Operational Rationality through Compilation of Anytime Algorithms

by

Shlomo Zilberstein
B.A. (Technion – Israel Institute of Technology) 1982

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

in the
GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:
Professor Stuart J. Russell, Chair
Professor Susan L. Graham
Professor Alice Agogino
Professor Thomas L. Dean

1993

Operational Rationality through Compilation of Anytime Algorithms

Contents

Table of Contents iii

List of Figures vii

List of Tables ix

Preface x

Acknowledgements xii

1 Introduction 1
- 1.1 The cost of deliberation . . . 1
- 1.2 Rationality in artificial agents . . . 2
  - 1.2.1 The failure of classical decision theory . . . 2
  - 1.2.2 Operational rationality . . . 5
- 1.3 Thesis . . . 6
  - 1.3.1 Historical perspective . . . 6
  - 1.3.2 Thesis statement . . . 7
- 1.4 Achieving operational rationality . . . 8
  - 1.4.1 Anytime algorithms, compilation and monitoring . . . 8
  - 1.4.2 The main problems addressed by this work . . . 9
- 1.5 Dissertation organization . . . 11

2 Background: Real-Time Decision Making 12
- 2.1 Reactive systems and universal planning . . . 12
- 2.2 Meta-level control of computation . . . 14
- 2.3 Anytime algorithms . . . 16
  - 2.3.1 Anytime path planning . . . 16
  - 2.3.2 Flexible computations . . . 18
  - 2.3.3 Approximate processing . . . 19
  - 2.3.4 Incremental approximate planning . . . 20
  - 2.3.5 Anytime query answering in relational databases . . . 21
- 2.4 Design-to-time scheduling . . . 22
- 2.5 Real-time problem solving . . . 23
  - 2.5.1 Real-time heuristic search . . . 23

2.5.2 Soft real-time 24
2.6 System support for approximate computation 26
2.7 Summary 27

3 Utility-Driven Real-Time Agents 28
3.1 Artificial agents 28
3.2 Goals versus utility functions 30
3.2.1 Partial goal satisfaction 30
3.2.2 Explicit and implicit utility 31
3.3 Real-time agents 32
3.3.1 Time-dependent utility 32
3.3.2 The cost of time 33
3.4 Evaluating utility-driven agents 34

4 Anytime Computation 35
4.1 Anytime algorithms 35
4.1.1 Measuring quality of results 35
4.1.2 Interruptible versus contract algorithms 37
4.1.3 The anytime traveling salesman 38
4.2 Performance profiles 40
4.2.1 Categories of performance profiles 40
4.2.2 Properties of performance profiles 41
4.2.3 Finding the performance profile of an algorithm 43
4.2.4 Representation of performance profiles 44
4.2.5 The library of performance profiles 46
4.3 Model of execution 47
4.3.1 Design goals 47
4.3.2 Assumptions about the programming environment 47
4.3.3 Running elementary anytime algorithms 48
4.4 Programming techniques 50
4.5 Theoretical aspects of approximate computation 53

5 Compilation of Anytime Algorithms 55
5.1 Why compilation? 55
5.1.1 Modularity and anytime computation 56
5.1.2 Minimizing the responsibility of the programmer 57
5.2 The compilation problem 57
5.3 Compilation examples 59
5.3.1 Composition of two algorithms 59
5.3.2 Linear composition of anytime algorithms 62
5.3.3 Local and global compilation 63
5.4 Conditional performance profiles 64
5.4.1 Special cases of conditional performance profiles 65
5.4.2 General conditional performance profiles 67
5.5 Compilation of functional composition 67

5.5.1 Complexity results 68
5.5.2 Optimality of local compilation 71
5.5.3 Additional programming operators 74
5.5.4 Repeated sub-expressions 76
5.5.5 Compilation of unrestricted composite expressions 78
5.5.6 Compilation under uncertainty 84
5.6 Compilation of other programming structures 85
5.6.1 Conditional structures 85
5.6.2 Compilation of boolean expressions 88
5.6.3 Compilation of loops 93
5.7 Summary 96

6 Run-Time Monitoring of Anytime Algorithms 97
6.1 The run-time system 97
6.1.1 Monitoring schemes 97
6.1.2 Uncertainty in real-time systems 98
6.1.3 Episodic problem solving 100
6.2 Monitoring contract algorithms 101
6.2.1 Re-allocating residual time 103
6.2.2 Adjusting contract time 104
6.2.3 A monitoring example 104
6.3 Monitoring interruptible algorithms 105
6.3.1 Interruptible anytime systems 105
6.3.2 Active monitoring using the value of computation 107
6.3.3 Dynamic adjustment of performance profiles 109
6.3.4 A monitoring example 110
6.4 Summary 111

7 Anytime Sensing and Anytime Action 112
7.1 Beyond episodic problem solving 112
7.1.1 The purely computational agent 112
7.1.2 Computation, perception and action 113
7.2 Anytime sensing 113
7.2.1 Sensing versus computation 114
7.2.2 Passive sensing in semi-static environments 114
7.2.3 Sensing as an independent process 115
7.2.4 Sensing as an information gathering action 115
7.3 Anytime action 116
7.3.1 Elementary anytime actions 116
7.3.2 Performance profiles of actions 117
7.3.3 Buying time in real-time domains 118
7.3.4 Controlling anytime action 118
7.4 Operational rationality over history 119

8 Application and Evaluation 120

8.1 Path planning and navigation in robotic systems . . . 120
  8.1.1 The environment . . . 120
  8.1.2 Anytime sensing . . . 122
  8.1.3 Anytime abstract planning . . . 123
  8.1.4 Compilation of sensing and planning . . . 129
  8.1.5 The run-time system . . . 130
8.2 Other applications . . . 131
  8.2.1 Anytime pragmatic diagnostic engine . . . 132
  8.2.2 Generate-and-test search . . . 133
8.3 Model evaluation . . . 135
  8.3.1 Constructing elementary anytime algorithms . . . 135
  8.3.2 Computing performance profiles . . . 136
  8.3.3 Compilation . . . 136
  8.3.4 Achieving operational rationality . . . 136

9 Conclusion 138
9.1 Contribution . . . 138
9.2 Further work . . . 139
  9.2.1 The scope of compilation . . . 139
  9.2.2 The theoretical framework . . . 139
  9.2.3 Programming support for anytime computation . . . 140
9.3 Design, deliberation and adaptation . . . 140

Glossary 142

Bibliography 144

List of Figures

1.1 Ideal, traditional, and anytime decision procedures 5
1.2 Path planning example 9
1.3 The conceptual layers of the model 9

2.1 Termination condition using rational meta-reasoning 15
2.2 Performance profiles of tour improvement and path planning 17
2.3 Optimal time allocation to base-level computation 19
2.4 Concord process structure 26

3.1 Artificial agents 29
3.2 Forms of compiled and uncompiled knowledge 30

4.1 Typical performance profiles 36
4.2 Performance profiles of interruptible and contract algorithms 38
4.3 The operation of randomized tour improvement 39
4.4 The anytime traveling salesman algorithm 39
4.5 Superior performance profiles 42
4.6 The quality map of the TSP algorithm 44
4.7 The expected performance profile of the TSP algorithm 45
4.8 Typical implementation of an interruptible anytime algorithm 49
4.9 The control of anytime computation 50

5.1 Compilation and monitoring 56
5.2 Performance profiles of two contract algorithms 59
5.3 Conditional performance profile of a TSP algorithm 67
5.4 Tree representation of a composite expression 72
5.5 Anytime composite module for speech recognition 76
5.6 Composite expressions and straight line code 77
5.7 DAG representation of composite expressions 78
5.8 Time allocation using a hill-climbing search 79
5.9 Time allocation with pre-determined time to repeated sub-expressions 80
5.10 Learning the allocation to repeated sub-expressions 82
5.11 Compilation of a boolean expression 89
5.12 Interruptible evaluation of a boolean expression 93
5.13 Compilation of an unbounded loop 94

6.1 Monitoring anytime computation 99
6.2 A sequence of residual sub-systems 103

8.1 Data flow diagram 121
8.2 The performance profile of the vision module 122
8.3 The anytime planning algorithm 124
8.4 The path finder 124
8.5 Optimal path 125
8.6 Abstract plans with perfect vision 126
8.7 The performance profile of the anytime planner 127
8.8 Abstract plans with imperfect vision 128
8.9 The conditional performance profile of the anytime planner 129
8.10 Compilation of vision and planning 130
8.11 Intra-frame optimization 131
8.12 The run-time display 132
8.13 Architectures for model based diagnosis 133
8.14 Distribution of solutions over time for S1-4 135

9.1 Real-time decision making 140

List of Tables

2.1 Approximate relational algebra operations 21

4.1 The performance distribution profile of the TSP algorithm 46

5.1 Optimal time allocation in boolean expression evaluation 90

Preface

Since my first encounter with a computer, I was fascinated by the possibility of writing programs that could really "think." A few weeks after my first BASIC lesson, I remember writing programs that could play and win simple games such as Nim and Mastermind. But those programs did not think. They were designed to follow a winning strategy that could be calculated by a simple formula. The computer's fast information retrieval and speed of computation were a substitute for thinking and left open the question of how to develop a program that could truly reason about its domain.

Years later, when I joined the Ph.D. program at Berkeley, I knew that my research was going to be in the area of automated reasoning. Believing that the answer to this fundamental problem lay within the realm of mathematical logic, I started a journey, in the words of Leibnitz, into "the universal algebra of all knowledge." Gradually however, it became apparent to me that logic, as attractive and elegant as it may seem, cannot capture the richness of practical reasoning. Practical reasoning is approximate, is resource bounded, and is interruptible. Logical reasoning is precise, is intractable, and is non-interruptible. I also found that classical decision theory that better deals with reasoning under uncertainty does not address adequately the problem of practical reasoning.

While at Berkeley, I became familiar with the work of Stuart Russell and Eric Wefald on principles of meta-reasoning. This work offered a new framework for automated reasoning that addressed the problem of limited computational resources. It suggested, like I. J. Good before, that practical reasoning should be based on a certain kind of limited rationality that takes into account computational resources. The global optimization problem of decision quality, as Herbert Simon claimed in the late 1950's, should be "to find the least-cost or best-return decision, net of computational costs." Consequently, I started to work on the construction of a practical model that would embody these principles. I was convinced that additional architectural restrictions must be made before the principle could be put into efficient use.

The work of Tom Dean and his students on time dependent planning triggered my inquiry into the possibility of building large real-time decision making systems using anytime algorithms. I suggested that building decision systems from anytime modules could be the architectural constraint. I felt that anytime algorithms offered the flexibility needed to construct a general and practical model of limited rationality. Anytime algorithms, or more generally, approximate algorithms, are as old as computer programming. However, it was not until the late 1980's that it was suggested that decision-theoretic control of anytime algorithms could be used for optimizing real-time problem solving. This idea, that met with considerable initial skepticism, has now been embraced by the artificial intelligence community.

My central research goal has been to bridge the gap between the simple task of developing elementary anytime algorithms and the complex task of constructing large systems that offer a similar tradeoff between computation time and quality of results. This effort has culminated in the construction of a model of "operational rationality." However, the problem of constructing programs that can really think still remains. Intelligence today is still achieved largely by design, not by autonomous learning and adaptation.

From my early naive attempts to make computers think, I have grown to appreciate the true complexity of this task. Now I find that it is precisely that complexity which stimulates me to continue this research.

Acknowledgements

I will always remember the years spent at Berkeley as an exciting period of personal growth as a scientist, as a teacher and, most importantly, as an individual. With much pleasure, I take the opportunity to thank the people who helped me through this fascinating period.

Stuart Russell, my teacher and advisor, introduced me to the enchanting world of artificial intelligence and made my studies challenging and enjoyable. His demonstrations of constant confidence in the face of research uncertainties on the one hand, and his permanent questioning of our methods on the other, have been and always will be inspiring to me.

The members of my dissertation committee were very helpful. Susan Graham was supportive during my entire graduate career. Alice Agogino gave me encouragement and useful advice. I am particularly thankful to Tom Dean of Brown University who inspired my initial interest in anytime algorithms and has continued to provide me with important insights and comments.

My teachers at Berkeley created a stimulating environment that I will greatly miss. In particular, I am grateful to Jitendra Malik and Lotfi Zadeh for broadening my understanding of artificial intelligence and to Michael Harrison and Raimund Seidel with whom I greatly enjoyed working as a teaching assistant during my first year.

The past and present members of RUGS, with whom I had many productive and enjoyable discussions, include Francesca Barrientos, Jeff Conroy, Marie desJardins, Othar Hansson, Tim Huang, Sonia Marx, Andrew Mayer, Ron Musick, Gary Ogasawara, Sudeshna Sarkar and the late Eric Wefald. They listened patiently to my evolving ideas and to countless practice talks and even had the patience to provide useful comments. My friends at Berkeley, especially Benjamin Fuchs and Yochai Konig, kept me from being overly dedicated to my research. My lifelong friends in Israel, Ilan, Menashe, Moishe, Oded and Oren, whom I greatly miss, always believed in my ability and encouraged me in all my endeavors.

The Computer Science Division staff was always supportive and helpful: Ellen Boyle, Teddy Diaz, Liza Gabato, Ani Nayman and Jean Root. Kathryn Crabtree, in particular, helped me navigate the intricate maze of Berkeley's bureaucracy with utmost ease.

My family always surrounded me with love and support. I wish to thank my parents, Arie and Esther, my sister Tal and my brothers Ronen and Niv. My extended family, Barbara and Sheldon Rothblatt and Abe and Fela Hirsch, have created for me a warm, “hamish” feeling at Berkeley. My wife, Karen, has changed my life in so many wonderful ways. To her, I dedicate this work.

Chapter 1

Introduction

> If the human brain was so simple that we could understand it, then *we* would be so simple that we could not.
>
> L. Watson

How can an artificial agent[1] react to a situation after performing the "right" amount of thinking? In this dissertation I develop a theoretical framework and a new programming paradigm that provide the answer to this question. The key component of the solution is the replacement of standard modules of a program by more flexible computation elements that are called *anytime algorithms* [Dean and Boddy, 1988; Horvitz, 1987]. In addition, the model includes an off-line compilation process and a run-time monitoring component that guarantee that the agent is performing the right amount of thinking in a well-defined, rigorous sense.

1.1 The cost of deliberation

Agents must limit the amount of thinking or deliberation since thinking has a cost associated with it. The overall performance of artificial agents can be improved by control of deliberation time. Two factors determine the cost of deliberation: the resources consumed by the process, primarily computation time, and constant change in the environment that may decrease the relevance of the outcome and hence reduce its value. In artificial agent construction, the cost of deliberation can be drastically reduced if the agent's reaction to any possible situation can be calculated and stored in a table. The result is a *reactive agent* whose behavior is determined by the state of the environment and a relatively fast look-up operation in a table. As I argue in Section 2.1, this architecture is not realistic for situations in which a robot is performing any "interesting" task in a real, physical environment. The size of the table required to guide the robot would be enormous, too large to construct or store using any modern computer. I therefore assume that an agent must perform some real-time problem solving and explicit deliberation, and is therefore a *deliberate agent*.

An important aspect of intelligence, traditionally ignored in the development of artificial agents, is the capability of the agent to factor the cost of deliberation into the deliberation process. People do it all the time. When one plans a trip to Japan, for example, the plan is not likely to include a specific action to be taken in case the Shinkansen train from Tokyo to Osaka is canceled. This possibility is not part of the plan

---

[1] An agent can be thought of as a robot situated in a particular environment and capable of translating perceptual input into actions that bring about a desired state. The notion of an artificial agent is defined and discussed in more detail in Chapter 3.

because it is very unlikely to happen given one's prior knowledge of trains in Japan. Yet, it is not impossible. However the time needed to plan an alternative action can be better utilized. Obviously, a plan that includes an alternative action for any possible problem is a better plan. It may help the agent reach the destination faster in case of an unexpected event. However, if one tries to consider the range of all possible problems that may occur on the trip, one would have to stay at home and plan forever. My primary research goal has been to formalize an efficient model in which the tradeoff between continued deliberation and commitment to action can be analyzed. Most people use common sense in order to decide on the "appropriate" amount of deliberation. Unfortunately, common sense reasoning is not a well understood process. In order to enable artificial agents to handle this problem, I develop in this dissertation a model that specifies the type of knowledge and reasoning procedures that can mechanize and optimize this process.

The capability of a system to reason about its own decision-making component in order to estimate the value of continued deliberation has been generally referred to as *meta-reasoning* [Batali, 1986; Davis, 1980; Dean and Boddy, 1988; Doyle, 1988; Genesereth, 1983; Horvitz, 1987; Russell and Wefald, 1989b]. Meta-reasoning, or reasoning about reasoning, can be used in various ways in order to improve the performance of a system: by selecting the most appropriate base level reasoning procedure in any given situation, by controlling a base level search procedure, or by dynamic allocation of computational resources to competing computation sequences. In the model developed in this dissertation, a meta-reasoning component was developed for the purpose of controlling the *deliberation time* of the flexible components of the base level.

1.2 Rationality in artificial agents

The search for a precise definition to the notion of "the right amount of thinking" leads naturally to the theory of rational choice. The notion of rationality has been widely discussed in philosophy, economics and artificial intelligence. In my work, I have used the decision-theoretic notion of rationality where probabilistic information about the possible outcomes of actions together with knowledge on the payoffs of these outcomes are used in order to make the best decision in terms of expected payoff. To compute the expected payoff, decision theory uses utility functions that specify the desirability of certain configurations of the world. The relationship between the given utility function and the behavior of the agent is the key question. Any method used by the agent that suggests the "right action" to do, so as to maximize the utility function, can be considered as a type of rationality. In this dissertation I show how a model based on compilation of anytime algorithms can be used in order to successfully implement a certain kind of limited rationality. I also show that the alternative views of rationality as a basis for agent construction are ill-defined, unrealistic, or both.

1.2.1 The failure of classical decision theory

The notion of rationality that I use in this dissertation stems from the pioneering work of von Neumann and Morgenstern [1947]. This work laid the foundation of what is called statistical decision theory. According to this theory, an agent faced with a choice of performing one out of several possible actions would select the action that maximizes the expected payoff. In other words, the agent would select the action that is most likely to transform the state of the world into a highly desired state according to the its utility function. An agent that performs actions that satisfy this theory is a *perfectly rational* agent. Since the theory allows for uncertainty concerning the outcome of each action, a perfectly rational agent may reach undesirable states, but in the long run, it would outperform any other agent in maximizing the utility function.

As attractive as it may seem, classical decision theory fails when used as the basic mechanism for implementing an artificial agent. It suffers from the following weaknesses:

1. *Ignoring the cost of deliberation*. Perfect rationality requires optimal decision making in virtually no time since, among other reasons, the world is changing while the agent is computing its next action. Inaction causes loss of utility. By assuming a static world that is "waiting" for the agent to make its optimal decision, the theory ignores a major aspect of any realistic domain.

2. *Exhaustive evaluation of all possible outcomes*. Classical decision theory is based on exhaustive evaluation of all the possible outcomes of all actions. This requires a lot of unnecessary computation. It is enough to establish the fact that one action is superior to all others without exact evaluation of all the alternatives. Exhaustive consideration of all the possible outcomes is not only inefficient but computationally impossible.

3. *Optimizing individual actions*. Since the overall performance of the agent is important, not the outcome of each individual action, it may be necessary to optimize over all possible sequences of actions over a certain period of time. However, applying classical decision theory to sequences of actions is too complicated. The size of the decision tree grows exponentially and it cannot be completely evaluated in any reasonable time.

4. *Learning and exploration*. An important aspect of any intelligent agent is the capability to improve its performance component by learning and exploration. The utility of learning is hard to determine in advance. Past experience can be used to estimate the effect of learning on performance. However the objective of exploration and learning is to improve the agent and transform it into a new, better system whose exact characteristics cannot be known in advance. The effect of learning may not be noticeable immediately, but rather in the long run. Therefore, it is hard to characterize the desirability of learning and exploration and to maintain the notion of perfect rationality.

As a result of these weaknesses, perfect rationality requires an agent that follows a precalculated optimal strategy[2]. It also requires that the agent retrieves the appropriate decision instantaneously. Unfortunately, due to the computational limits of the designer and his imperfect knowledge, the requirement to equip the agent with a perfect strategy is too strong and unrealistic. Therefore, artificial agents cannot be perfectly rational. They cannot manifest the best possible behavior even in relatively simple domains such as chess playing.

The failure of classical decision theory led the statistician I. J. Good [1971] to distinguish between perfect rationality, which he called "type I" rationality, and "type II" rationality which acknowledges the fact that the agent must deliberate before it can act. This type of rationality requires that the agent maximize its expected utility, taking into account the cost of deliberation. Similarly, referring to the problem of rational decision making in the field of economics, Simon [1976] says that: "The global optimization problem is to find the least-cost or best-return decision, *net* of computational costs." But neither Good nor Simon tell us how to achieve type II rationality. Moreover, while type II rationality may be a more realistic goal, it does not offer any simplification of the problem. In a way, type II rationality is simply more general and would produce type I rationality if the computational power available to the agent is unlimited. Hence, achieving type II rationality is an even harder task.

---

[2] In some competitive environments it can be shown that no optimal strategies exist. However we restrict the discussion to a single agent operating in a non-competitive environment.

A number of researchers in AI have addressed the problem of deliberation cost by suggesting various techniques that take into account the actual computational power of the agent. These techniques are generally referred to as “limited” or “bounded” rationality. While perfectly rational agents cannot be constructed at all, limited rationality can be constructed in principle. Just imagine all the possible implementations of agents by programming a given machine. One of them must be superior to all the others in maximizing the utility function and is therefore the desired implementation. However, direct construction of such an optimal agent is impossible under standard computer architectures since it transfers the problem into the design level and requires that the designer have infinite resources of computation. Consider, for example, a program that can play chess better than any other computer program or human being. Is there a way to effectively check whether the program is or is not the best possible, given the machine capabilities? If the program is modified so that one subroutine is running faster, is the latter program more “rational” just because of that? The conclusion is that even bounded rationality cannot be achieved in practice or even be verified. Beside computational power, additional architectural constraints must be assumed before a practical approach to rationality can be developed.

More recently, Russell, Subramanian and Parr proposed a definition of *bounded optimality* as a property of programs that govern the behavior of an agent *given* a computational device and a certain environment. To have this property, the expected utility of the program running on the device in the environment must be at least as high as that of all other programs. For a restricted class of programs, that consists of a sequence of decision procedures, a construction algorithm is proved to generate a bounded optimal program. Russell, Subramanian and Parr acknowledge that the strict notion of bounded optimality may be too strong to allow many interesting, general results to be obtained. Hence they suggest, just as in complexity theory, to replace bounded optimality by asymptotic bounded optimality. The latter case requires that the program just needs a faster machine to be as good as the best possible program on arbitrarily hard problems.

Another approach to bounded rationality is based on a meta-reasoning that treats computations as internal actions [Russell and Wefald, 1989b], as opposed to external actions that correspond to actual interaction with the environment. According to this approach, the agent spends some time on estimating the expected value of alternative computations. It then performs the best computation, provided that the expected value of this computation exceeds its cost. If there is no such computation the algorithm suggests the *current best action* based on previous computations. The evaluation of alternative computations is an internal problem that is solved in the same way using meta-meta-reasoning. The difficulty with such a uniform meta-level architecture is that any attempt to maximize the expected utility of the agent leads to an *infinite regress* problem [Batali, 1986; Doyle, 1988; Russell and Wefald, 1989b]. This problem arises as a result of optimizing a process that involves self-reference and recursive evaluation of internal computations without any justified way to truncate this recursive process and maintain overall optimality. However, limiting the number of meta-levels can be accepted as a reasonable architectural constraint. In fact, a number of applications of this approach, that assume a single meta-level layer, have been developed.

Finally, an important technique that simplifies the control of deliberation time is based on *anytime* [Dean and Boddy, 1988] or *flexible* [Horvitz, 1987] algorithms. To summarize its benefits, Figure 1.1 (based on [Russell and Wefald, 1991]) illustrates the differences between three alternative decision procedures. It shows the decision quality as a function of time for each method. In this particular example, the decision quality measures the effect the decision would have on the utility of the agent if applied at the current state. An ideal decision procedure yields maximal quality in no time. Hence, it is illustrated by a step function that rises to maximal quality at $t = 0$. Traditional decision procedures are either quality maximizing or time minimizing. That is, they either produce the maximal quality after a certain time or produce an acceptable decision as fast as possible. The former case is illustrated by a step function that

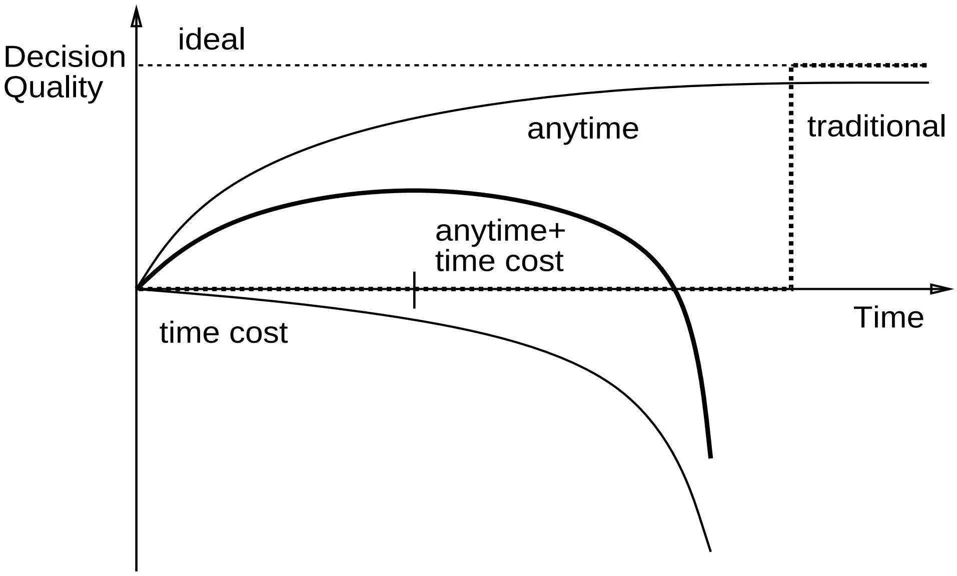


Figure 1.1: Ideal, traditional, and anytime decision procedures

rises to maximal quality at a certain time $t \gg 0$. Unfortunately, by that time the decision may have little value due to change in the environment. The time cost function describes the expected loss of utility as a function of time in the absence of a decision. In this example, the traditional decision procedure returns its result at a point where the time cost is very high, hence its *comprehensive value* is negative. Finally, an anytime decision procedure can generate sub-optimal decisions whose quality improves as computation time increases. When the value of these decisions is combined with the cost of time, an optimal time allocation can be determined that maximizes the comprehensive value.

It will become apparent to the reader hereinafter that the above description is somewhat oversimplified. The cost of time is not so easily determined and may not be separable from the characteristics of the decision procedure itself. However, the figure illustrates the main features of the three methods and the motivation for anytime computation.

1.2.2 Operational rationality

In the face of theoretical and practical limitations in implementing both Type I and Type II rationality, I suggest a more restricted, realistic model of rationality. The model is based on optimizing resource allocation to anytime algorithms. In this type of rationality, the performance components of the agent are determined by the designer of the system and are not *themselves* subject to the run-time optimization process. Optimization is only applied at the meta-level to control the deliberation time of the base-level performance components. Hence I propose the following definition:

Definition 1.1 *An agent is said to be* **operationally rational** *if it optimizes the allocation of resources to its performance components so as to maximize its overall expected utility in a particular domain.*

Operational rationality separates two, central aspects of agent construction: the development of the performance components and the optimization of performance. It makes algorithm development a design issue, not a run-time issue. The principles of rationality are applied only at run-time, to control the deliberation time of the performance components. In alternative approaches to rationality, these two aspects are inseparable. As a result, the task of rational agent construction becomes too complex[3].

Operational rationality still allows individual algorithms to be adaptive. The agent can learn and improve its performance over time. However, the concrete algorithms used and the flow of information between them are fixed and not part of the agent's self-optimizing problem.

The definition of operational rationality does not tell us how to achieve it. Many questions are left open. How can the deliberation time of the performance components be monitored? What types of knowledge and reasoning procedures are necessary to control the execution of the performance components? These questions form the core of the problem addressed by this dissertation.

1.3 Thesis

This section identifies the fundamental issues addressed by this dissertation. The main part – the thesis statement – is presented in the form of five claims that will be validated in the remaining chapters.

1.3.1 Historical perspective

By no means was decision theory the first attempt at formalizing the rules of reasoning. The mechanization of thought using formal systems evolved in the seventeenth century, long before the emergence of modern computers. In 1650, the English philosopher, Thomas Hobbes, proposed the idea that thinking is a computational process, analogous to arithmetic. It was probably the philosopher Gottfried Wilhelm Leibnitz [1646 - 1716] who envisioned for the first time the complete mechanization of intelligence. Leibnitz[4] published a book, *Dissertio de arte combinatorica* (Leipzig, 1666), in which he expressed his vision of "a universal algebra by which all knowledge, including moral and metaphysical truths, can some day be brought within a single deductive system." It took another two centuries before Gottlob Frege, in what is perhaps the most important single work ever written in logic [Frege, 1879], formulated the basis of predicate calculus. Frege described a system of logic in which derivations are carried out exclusively according to the form of expressions, an idea that became the basis of symbolic logic. Shortly after the development of the first electronic computers in the 1940's and 1950's, the founders of AI wrote programs that could perform elementary reasoning tasks, such as proving simple mathematical theorems and answering simple questions.

But, despite the fact that mathematical logic had already been well-formalized in the 1930's, it did not provide an effective framework for knowledge representation and reasoning in practical domains. Problems such as the intractability of automated theorem proving, the monotonicity of logical reasoning, and its inability to deal with uncertainty made it apparent that logic, as attractive and elegant as it may seem, cannot capture the richness of practical reasoning.

As a result, researchers tried several remedies: limiting knowledge expressibility in order to increase the efficiency of reasoning [Levesque, 1986]; formulating non-monotonic logics to overcome the monotonicity problem [Reiter, 1987]; and adding various measures of uncertainty to knowledge [Zadeh, 1975]. Other researchers abandoned logic completely and tried to find alternative representations and reasoning procedures such as Bayesian Networks [Pearl, 1988]. However, regardless of the method being

---

[3] Recall that the problems of program verification and program optimization are intractable. Any type of rationality that requires solving such problems is therefore impractical.

[4] See [Gardner, 1968], page 3.

used, what characterized artificial intelligence research on reasoning and planning systems over the past three decades was the development of systems whose complexity imposed a severe barrier on the size of the domains they could handle. From the early development of GPS and STRIPS through TWEAK and PRODIGY and more recent planning techniques[5], systems have been applied mainly to toy-worlds and could not be scaled up beyond that to handle real-world situations.

1.3.2 Thesis statement

The model that I will present in the following chapters has been primarily motivated by the observation that many AI systems suffer from lack of control over deliberation time. The failure of artificial intelligence to deliver expandable reasoning and planning systems is largely due to this problem. In the past, artificial intelligence systems had little control over the quality of their results and could not explicitly compromise accuracy in order to perform faster deliberation[6]. Such compromise, I argue, is essential for any rational decision making process.

> **Control of deliberation time is a key aspect that is missing in most AI systems and that limit their applicability.**

In an effort to overcome the barrier imposed by the complexity of reasoning, I have developed a model for intelligent control of deliberation. The model has successfully validated the following five claims:

Claim 1. Existence There exists an effective alternative to traditional algorithms, namely anytime computation, that offers a tradeoff between deliberation time and quality of results. This claim has been already validated by the work of Boddy and Dean, Horvitz, and Russell and Zilberstein in the area of automated reasoning, as well as by the work of Lesser, Pavlin and Durfee, Vrbsky, Liu and Smith, and others in the area of approximate computation. This work is presented in Chapters 2 and 4.

Claim 2. Feasibility Anytime algorithms can be efficiently constructed using standard programming techniques. This claim has been partly validated by a number of applications and is further discussed in Chapter 4.

Claim 3. Composability The principles of modularity can be applied to anytime computation. Large real-time systems can be composed of anytime components. The problem of time allocation within such systems can be handled by a special compilation technique. The validation of this claim constitutes the main contribution of this work. It is presented in Chapter 5.

Claim 4. Operational Rationality The performance of an agent composed of anytime algorithms can be efficiently optimized to yield an operationally rational agent. A meta-reasoner whose domain includes utility functions, domain descriptions, and performance profiles, can solve the optimization task. This claim has been partly validated by applications developed by Boddy and Dean that involve a small number of anytime algorithms. The validity of the claim with respect to large systems composed of anytime algorithms is presented in Chapter 6.

---

[5] For a survey of artificial intelligence planning systems and techniques see [Hendler *et al.*, 1990].

[6] Several exceptions will be described in Chapter 2.

Claim 5. Effectiveness Operational rationality is a powerful model that can effectively simplify the development of complex real-time systems. This is an immediate result of the previous claims and the major simplification they introduce into real-time system construction. This issue is discussed in Chapters 8 and 9.

1.4 Achieving operational rationality

This section outlines the model of operational rationality and the main problems that were raised by its implementation.

1.4.1 Anytime algorithms, compilation and monitoring

The fundamental property of an operationally rational agent is the capability to vary its deliberation time according to "time pressure." In order to achieve this capability, traditional algorithms, whose expected run-time is normally fixed, must be replaced by more flexible computation modules, namely *anytime algorithms*. Anytime algorithms are algorithms whose quality of results improves gradually as computation time increases. They introduce a continuous tradeoff between deliberation time and quality of results. The idea that such a tradeoff can be used in order to optimize the performance of real-time systems was independently developed by Dean and Boddy [1988], by Horvitz [1987], and by Lin *et al.* [1987]. In order to optimally control this degree of freedom, Boddy and Dean [1988] used *performance profiles* that characterize the dependency of output quality on run-time. I have extended this performance description by introducing *conditional performance profiles* that give a probabilistic description of the quality of the results of an algorithm as a function of run-time and input quality (or any set of input properties).

Conditional performance profiles are essential in order to project the effect of performance degradation within a system. Consider, for example, an anytime hierarchical planner whose quality of results is measured by the level of specificity of the plan. Obviously, the specificity of a plan affects its execution time and hence has influence on the efficiency of the agent. The performance of the planner depends on two factors: time allocation and the quality of its input, that is, the precision of the domain description. The conditional performance profile of this algorithm describes this dependency.

Some of the first applications of anytime algorithms were introduced by Boddy and Dean [1989] in solving a path planning problem, and by Horvitz [1987] in real-time decision making in the health care domain. The AI community reacted to this work with considerable skepticism, partly because of the difficulty of building large systems using anytime modules. In this dissertation I will introduce techniques that extend the use of anytime algorithms to the construction of complex real-time agents. It is unlikely that a complex system would be developed by implementing one, large, anytime algorithm. Systems are normally built from components that are developed and tested separately. In standard algorithms, the expected quality of the output is fixed, so composition can be implemented by a simple call-return mechanism. However, when algorithms have resource allocation as a degree of freedom, run-time scheduling and monitoring are required to guarantee optimal utilization of resources.

Figure 1.2 illustrates the composition of two anytime algorithms. It shows the performance profile of an anytime path planning algorithm that receives its input from an anytime vision module. The quality of vision is measured in terms of the precision of the domain description. The quality of path planning is measured in terms of specificity of the suggested plan. A special compilation scheme combines these two modules into one optimal anytime path planning algorithm that can automatically distribute any given amount of time between the two components so as to maximize the overall quality of the results.

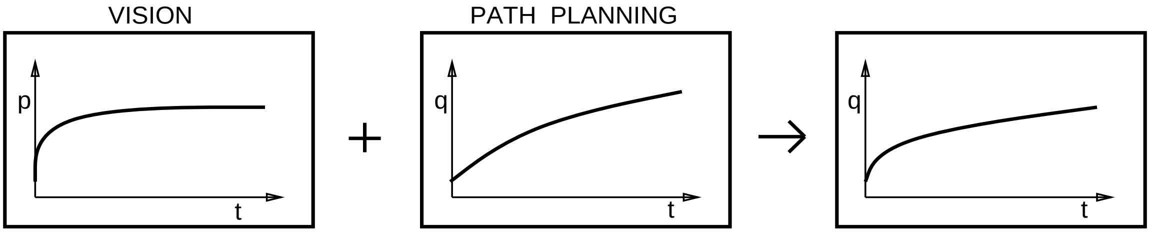


Figure 1.2: Path planning example

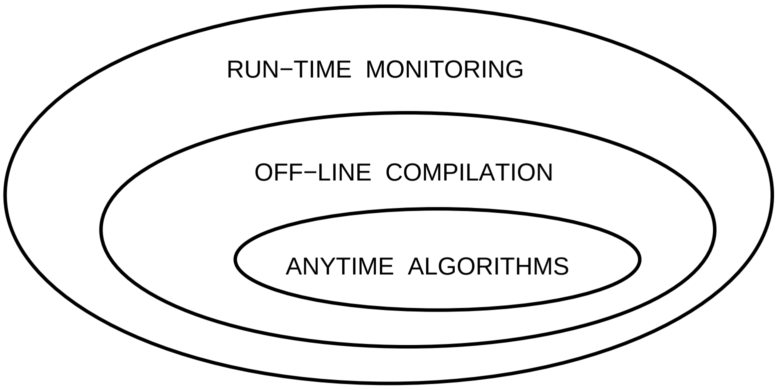


Figure 1.3: The conceptual layers of the model

Compilation produces *contract* algorithms which require the determination of the total run-time when activated. However, some real-time domains require *interruptible* algorithms whose total run-time is unknown in advance. This problem is solved by a standard technique to construct an interruptible algorithm once a contract algorithm is compiled.

Once a complete system is compiled into one anytime algorithm, the monitoring component of the run-time system is responsible for controlling the deliberation time of the system when operating in a particular environment. The complete model has three conceptual layers that are illustrated in Figure 1.3: anytime algorithms, off-line compilation that optimally combines anytime algorithms, and a run-time monitoring system that allocates resources to the components so as to optimize the utility of the complete system. These layers are described in detail in the following chapters.

This model of operational rationality introduces a new methodology to design complex real-time system. Instead of trying to design a system that would meet a specific set of time constraints, the design problem involves two orthogonal issues: decomposition of the total system into particular performance components and implementation of each basic component as an anytime algorithm. It will be shown in the following chapters that this approach greatly simplifies the construction of complex real-time systems.

1.4.2 The main problems addressed by this work

The implementation of the model of operational rationality as describe above was based on the solutions to the following key problems:

1. *Developing anytime algorithms.*

The use of anytime algorithms as basic blocks of complex systems calls for a new approach to algorithm construction. In human behavior and human problem solving, almost every activity has an anytime nature in the sense that we never commit ourselves to solving a problem without constantly reconsidering our methods and interrupting our activities. This kind of introspection is hard to formalize and mechanize. Existing software development techniques do not address this aspect of computation.

2. *Finding the performance profile of elementary anytime algorithms*.

   Performance profiles form the crucial meta-level knowledge needed for implementing operational rationality. Unfortunately, finding the performance profile of an *elementary anytime algorithm*[7] can be difficult and may require an extensive computation effort especially when it is based on simulation of the algorithm. In some cases, such as numerical analysis algorithms, the performance profile can be derived by direct analysis of the algorithm, but the general case is more complicated. Even more complicated is the task of finding conditional performance profiles that capture the dependency of output quality on run-time as well as on input quality. Finally, in most anytime algorithms there is a certain degree of uncertainty regarding the actual quality of results. Characterizing and representing this uncertainty is an important aspect of performance profiles.

3. *Compiling anytime algorithms*

   To allow modular system development, the performance profile of complex modules must be calculated based on the performance profiles of the components. Computing the best possible performance profile for the complete system involves solving a complex time allocation problem. This is a new kind of optimization problem that the compiler has to solve. A primary goal of this work was to mechanize the compilation of anytime algorithms and to solve it efficiently for large programs. Since the global optimization problem is shown to be NP-complete in the strong sense, local techniques must be utilized to reduce complexity. Establishing the optimality of such efficient local compilation techniques was an important part of this work.

4. *Anytime sensing and anytime actions*

   Since this dissertation is concerned with the development of artificial agents, the theory of anytime computation must be extended to deal with two additional key aspects of agent construction, namely sensing and action. Sensing, like computation, can be viewed an an information gathering act whose value can be modeled using similar tools. The modeling of actions as anytime interruptible activities is more complicated. In most existing systems, actions (such as: `(puton A B)`) are considered to be simple primitives, whose execution time is an insignificant constant. In practice, however, the execution time of an action is important and in many cases [8] actions have the property of graceful degradation of quality as a function of execution time. In that sense, actions are similar to anytime algorithms. The inclusion of anytime sensing and action in the model is an important step toward a definition of an architecture for artificial agent construction.

5. *Real-time scheduling and monitoring*

   When using anytime algorithms to control an artificial agent operating in a dynamic environment, the time allocation mechanism must take into account two sources of uncertainty. On the one hand,

---

[7] An elementary anytime algorithm is an anytime algorithm that does not use another anytime algorithm as a component.

[8] For example, when a “macro” action is actually implemented by alternating between computation and “micro” actions.

there is the uncertainty regarding the *actual* quality of the results produced by each component. The performance profile provides only *probabilistic* information on this aspect. On the other hand, there is the uncertainty regarding the state of the domain. The model of the environment used by the meta-level control is also probabilistic and there is always a possibility of a drastic change in time pressure. As a result, it is not enough to adopt a strategy of pre-determined fixed time allocation. Run-time scheduling and monitoring are necessary.

6. *Anytime algorithms and parallel computation*

   Since anytime algorithms are primarily designed to deal with complex real-time problems, it is only natural to utilize parallel machines for their implementation. The design of anytime algorithms using multi-processor systems introduces many open questions most of which deal with possible scheduling schemes. One such scheduling problem is to find the best scheduling scheme of a given non-interruptible algorithm on a machine with p processors to produce an interruptible algorithm with the best performance profile. Another scheduling problem is defined by using a multi-processor machine in order to run in parallel two different algorithms that solve the same problem: one with lower expected performance but a small possible deviation (i.e. the expected performance is almost guaranteed); the other with higher expected performance but a large possible deviation (i.e. the actual performance may be much worse than expected). By running these low-performance-low-risk and high-performance-high-risk algorithms in parallel, one can have the high expected performance together with a guarantee of a reasonable minimal performance.

1.5 Dissertation organization

In the following chapters I describe the details of the model of operational rationality and its implementation. In Chapter 2, I describe previous work on real-time decision making in the fields of artificial intelligence, control theory, economics and engineering. Chapter 3 defines the problem of constructing utility-driven real-time agents. In Chapter 4, I describe programming techniques to develop elementary anytime algorithms and their properties. An important distinction is made between contract and interruptible anytime algorithms. The chapter includes also the reduction theorem that shows how contract algorithms can be made interruptible. This important result allows us to solve the compilation problem in terms of contract algorithms and thus greatly simplify the problem. The central parts of the model are developed in Chapters 5 and 6. Chapter 5 focuses on the compilation process that automates the composition of anytime algorithms. Its main result is the development of an efficient local compilation technique whose complexity is linear in the size of the program. Local compilation is proved to yield global optimality for a large set of program structures. Chapter 6 describes the run-time monitoring component. In Chapter 7, I extend the notion of gradual improvement of quality to sensing and action. The application of the model and its evaluation are discussed in Chapter 8. Finally, in Chapter 9, I summarize the results and contributions of this work and identify three possible directions for further work. A brief glossary of specialized terminology appears at the end of the dissertation.

Chapter 2

Background: Real-Time Decision Making

> “Now! Now!” cried the Queen. “Faster! Faster!”
>
> Lewis Carrol, *Alice’s Adventures in Wonderland*

A considerable amount of work has been done on real-time decision making in the fields of artificial intelligence, decision theory, economics and engineering. In this chapter I will examine this work and identify the strengths and weaknesses of existing models. The focus of the analysis is on real-time decision making as a component of an architecture for artificial agent construction.

How do most AI systems cope with time constraints? In a comprehensive survey of real-time AI [Laffey *et al.*, 1988] that covered 48 systems, the authors claimed that “Currently, *ad hoc* techniques are used for making a system produce a response within a specified time interval.” Unfortunately, not much has been changed since that survey was conducted. The primary method for achieving real-time performance is based in many cases on speeding up individual algorithms in a generate-and-test manner. This method slows down the development of real-time systems and makes them inefficient when operating in dynamic environments. The wide variability of time pressure in dynamic environments makes it undesirable to design systems according to the worst case scenario. With the problem of artificial agent construction in mind, I will analyze each model concentrating on its potential to scale up and to handle successfully real-world situations. I start in Section 2.1 with an examination of the basic assumption that explicit deliberation and run-time problem solving are indeed necessary in real-time systems. Section 2.2 describes the work of Russell and Wefald on decision-theoretic control of inference. In many ways, the model developed in this dissertation is a refinement of Russell and Wefald’s framework. In Section 2.3, I describe several applications of anytime algorithms. Most notably, I discuss the work of Boddy and Dean, Horvitz, Lesser, Pavlin and Durfee, and Jane Liu who developed independently the first applications of such algorithms at a time when the notion of imprecise computation faced a large degree of scepticism. A closely related approach, Garvey and Lesser’s design-to-time scheduling, is presented in Section 2.4. Section 2.5 describes several related results in the area of real-time problem solving and Section 2.6 describes some experimental work on system support for approximate computation.

2.1 Reactive systems and universal planning

In an attempt to address the complexity of reasoning and planning in artificial intelligence, some researchers have proposed to limit the extent of run-time deliberation by using approaches such as *reactive planning* and *universal planning*. According to these approaches, explicit reasoning, problem solving and maintaining a world model are too complicated to handle at run-time and should be completely abandoned. Instead, believing that the world is its own best model, these researchers propose agents that are equipped with a mechanism that generates actions as an immediate response to interaction with the physical world.

Reactive planning [Brooks, 1986; Agre and Chapman, 1987] is the general approach of building systems that make the "right" decision by design. Agents based on these ideas are normally built from combinatorial circuits plus a small timing circuitry. The combinatorial circuitry may be adaptive, as in [Brooks, 1989], with the capability to converge on the desired behavior after a certain training period of interaction with the environment.

Universal planning [Schoppers, 1987] is a similar approach which is, in a sense, more restrictive than reactive planning since it does not allow for emergence of the desired behavior through interaction with the environment. Agents based on this approach determine what to do next by finding the current situation in a large table where the best action to be taken is stored. This leads to the following definition of a universal plan [Ginsberg, 1989]:

Definition 2.1 *A universal plan is a function,* $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{A}$*, from the set of possible situations* $\mathcal{S}$ *into the set of primitive actions* $\mathcal{A}$*.*

Both reactive planning and universal planning tend to transfer the complexity of planning to the design phase, thus ignoring the computational limits of the designer. While reactive planning can adaptively converge and produce some ordinary low-level behavior, it cannot be scaled up to deal with long-term planning in complex domains. As the complexity of the agent grows, the circuitry necessary to generate the correct response becomes too large and too complex to learn. Is it conceivable that chess-playing, medical diagnosis, or even path-planning for robot navigation could be performed by interaction with the environment alone, without any run-time deliberation? Will high-level strategic planning emerge "without reason"?

My dissertation is based on the assumption that some degree of real-time problem solving is unavoidable. This opinion is strongly supported by many researchers. Ginsberg [1989], for example, analyzed this problem and concluded that:

> "... even if the compile-time costs of the analysis are ignored, the size of the table must, in general, grow exponentially with the complexity of the domain. This growth makes it unlikely that this approach to planning will be able to deal with problems of an interesting size; one really needs the ability to do some amount of inference at run time."

A similar argument was presented years earlier by Herbert Simon referring to alternative views of rationality in the fields of economics and psychology. Simon made a distinction between *substantive rationality* and *procedural rationality*. Behavior is substantively rational if it is appropriate to the achievement of given goals in a particular environment. Like reactive planning, given a set of goals, the behavior is determined entirely by the characteristics of the environment. Behavior is procedurally rational when it is the outcome of appropriate deliberation. Its procedural rationality depends on the process that generated it. Simon[1] claims that:

> "... there is no point in prescribing a particular substantively rational solution if there exists no procedure for finding that solution with an acceptable amount of computing effort. So,

---

[1] See [Simon, 1982], page 428.

> for example, although there exist optimal (substantively rational) solutions for combinatorial problems of the traveling-salesman type, and although these solutions can be discovered by a finite enumeration of alternatives, actual computation of the optimum is infeasible for problems of any size and complexity. The combinatorial explosion of such problems simply outraces the capacities of computers, present and prospective."

Some researchers have proposed what may seem to be a natural synthesis of the two extremes: a system that would be reactive under high time pressure and would use classical reasoning methods when time is available. However, having to choose between the two extremes – between performing classical planning and not performing any planning at all – does not add much power to a system. Chapman [1989], for example, claims that:

> "As currently understood, planning is so inherently expensive – and reactive systems so inherently myopic – that even in combination they are useless."

Previous work on anytime algorithms and the model of operational rationality presented here extend the tradeoff that is offered by a simple synthesis of traditional and reactive planning. The reactive system and the fully deliberative one become two extreme points on a continuous scale of possible computational time offered by anytime algorithms.

2.2 Meta-level control of computation

Russell and Wefald [1989a, 1989b] developed a normative decision-theoretic approach for control of inference. The agent's objective is to maximize a given utility function defined over the states of the world: $U : \Omega \rightarrow \mathcal{R}$. The agent has a set, $\mathcal{A}$, of possible *base-level* actions that transform the environment. The outcome of a particular action, A, performed in state ω is denoted by $[A, \omega]$ or simply $[A]$ if ω is the current state. The agent's current default 'intention', typically the external action considered to have the highest utility, is denoted by α. The set $\mathcal{S}$ includes a sequence of computation actions that can be used to revise the agent's decision. At any given time, the agent has to choose whether to perform what is believed to be the best *external* action α, or to perform one of the computational actions, $S_1, ..., S_k$ that affect only the *internal* state of the agent. Since computation takes time, the *net value* of computation is the difference between the utility of the state resulting from the computation and the utility of the state resulting from performing the default external action α:

$$V(S_j) = U([S_j]) - U([\alpha]) \tag{2.1}$$

If S_j is a *complete* computation, resulting in a revised assessment of the best action, α_{S_j}, and a commitment to perform this action, then

$$U([S_j]) = U([\alpha_{S_j}, [S_j]]) \tag{2.2}$$

where $[\alpha_{S_j}, [S_j]]$ indicates the outcome of the action α_{S_j} in the state following the computation S_j. In the general case however S_j can be a *partial* computation affecting only the internal state but not immediately revising the assessment of the best action. In this case,

$$U([S_j]) = \sum_T Pr(T) U([\alpha_T, [S_j.T]]) \tag{2.3}$$

where T ranges over all possible complete computations following S_j, $S_j.T$ denotes the computation corresponding to S_j immediately followed by T, and $Pr(T)$ is the probability that the agent will perform the

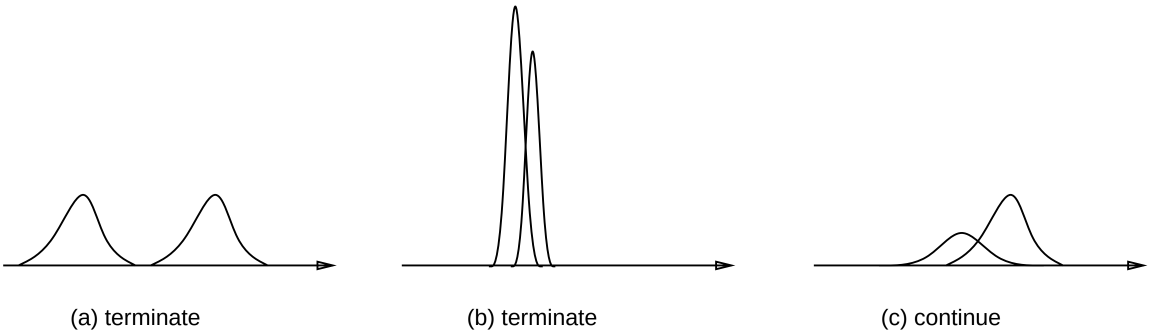


Figure 2.1: Termination condition using rational meta-reasoning

computation sequence T subsequent to S_j. A perfectly rational agent would select the computation that maximizes $U([\alpha_T, [S_j.T]]$, and this sequence would have probability 1. However, having limited computational resources, the agent cannot calculate the exact utilities and probabilities and must estimate them using some computational resources. Let $\hat{Q}^{\mathsf{S}}$ denote the agent's *estimate* of quantity Q following a computation S, typically based on the evidence produced by the computation. Then,

$$\hat{U}^{\mathsf{S}.S_j}([S_j]) = \sum_T \hat{Pr}^{\mathsf{S}.S_j}(T)\hat{U}^{\mathsf{S}.S_j}([\alpha_T, [S_j.T]]) \tag{2.4}$$

where S is the total computation preceding S_j, and

$$\hat{U}^{\mathsf{S}.S_j}([\alpha_T, [S_j.T]]) = \max_i \hat{U}^{\mathsf{S}.S_j.T}([A_i, [S_j.T]]) \tag{2.5}$$

where A_i ranges over all possible base-level actions in $\mathcal{A}$. Then the estimated net value of computation S_j, that is used by the meta-reasoning to decide whether further deliberation is valuable, becomes:

$$\hat{V}^{\mathsf{S}.S_j}(S_j) = \hat{U}^{\mathsf{S}.S_j}([S_j]) - \hat{U}^{\mathsf{S}.S_j}([\alpha]) \tag{2.6}$$

Of course, before the computation S_j is performed, $\hat{V}(S_j)$ is a random variable. The agent cannot know ahead of time what the exact value of $\hat{V}(S_j)$ will be, but the agent can estimate its *expectation*:

$$E[\hat{V}^{\mathsf{S}.S_j}(S_j)] = E[\hat{U}^{\mathsf{S}.S_j}([S_j])] - E[\hat{U}^{\mathsf{S}.S_j}([\alpha])] \tag{2.7}$$

Under certain assumptions, it is possible to capture the dependence of utility on time by a separate notion of the *cost of time*, so that the consideration of the quality of an action can be separated from considerations of time pressure. In such a case, the value of an action is measured by its *intrinsic utility*. The overall utility of a state is defined as the difference between the two:

$$\hat{U}([A_i, [S_j]]) = \hat{U}_I([A_i]) - TC(|S_j|) \tag{2.8}$$

where TC is the time cost function that depends only on $|S_j|$, the length (in elapsed time) of S_j.

Since there is a considerable uncertainty concerning the value of each action, the meta-reasoning component must be able to select among alternative actions without knowing their exact utilities. When the degree of uncertainty is too large to determine the best action, the meta-reasoning component may decide on additional deliberation to improve the utility estimates. Figure 2.1 [Russell and Wefald, 1989b] illustrates the three major situations that arise in evaluating the expected utility of two alternative actions. In case (a), one action is clearly superior to the other hence no more deliberation is necessary. In case (b), one action appears to be superior and the small possible difference between the utilities of the actions makes it undesirable to continue the computation. In case (c), one action appears to be superior, however the large uncertainty makes it desirable to continue the computation.

Russell and Wefald make several assumptions to simplify the analysis. First, they make the *meta-greedy assumption*, that the agent considers only single computation steps and chooses the one that appears to have the highest benefit. This computation may not be the optimal one considering a *sequence* of computation steps. Second, they make the *single-step assumption* that the agent will take at most one more search step. Their third assumption is the *subtree-independence assumption*, that a computational action can affect the expected utility estimate for exactly one base-level action. Under these assumptions, several search algorithms were developed, most notably MGSS*, and were proved superior to the best human-designed search algorithms for several games, such as Othello.

How does this model of rational meta-reasoning relate to the model of operational rationality? Russell and Wefald propose a rather general framework for meta-level control of reasoning. Operational rationality is more specific in terms of the type of meta-level knowledge that it uses, in terms of the characteristics of the computational elements, and in terms of the optimization problem that it defines. General meta-reasoning leaves some of these aspects to be decided in the context of the problem domain. It reasons about computational actions that must be identified in each particular domain. To summarize, operational rationality offers a more specific type of meta-level control of computation and one that is also easier to apply.

2.3 Anytime algorithms

The term “anytime algorithm” was coined by Tom Dean in the late 1980’s as part of his work on time dependent planning. There has been a considerable amount of work on designing and using algorithms that offer gradual improvement of quality of results, both before and after Dean’s coining of the term “anytime.” Nevertheless, very little work has capitalized on the additional degree of freedom offered by anytime algorithms – freedom in the very general sense that the algorithm offers to fulfill an entire spectrum of input-output specifications, over the full range of run-times, rather than just a single specification. In this section I describe five early applications of anytime algorithms and relate them to the model of operational rationality. I start with a description of Dean and Boddy’s work that identify some of the fundamental elements of my model, most notably, scheduling deliberation processes using expectations in the form of performance profiles. In addition, their work raised many of the problems that this dissertation addresses.

2.3.1 Anytime path planning

Boddy and Dean [1989] used anytime algorithms in order to solve a path planning problem involving a robot courier assigned the task of delivering packages to a set of locations. The robot operates in a domain, the *gridworld*, where each point is a location that may be occupied by the robot or by an obstacle. The robot can only move on to one of the four neighbors of its current position, provided that that neighbor is not already occupied. The robot has a map of the world that it can use for path planning. The utility of the robot’s performance is defined in terms of the time required to complete the entire set of deliveries.

The robot has to determine the order in which to visit the locations, referred to as a *tour*, and, given

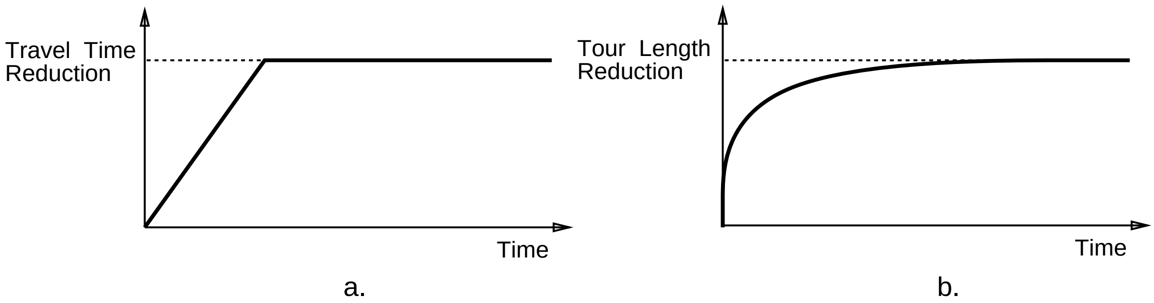


Figure 2.2: Performance profiles of tour improvement and path planning

a tour, it must plan paths between consecutive locations in the tour. To simplify the analysis, it is assumed that the robot's only concern is time; it seeks to minimize the total amount of time consumed both in sitting idle deliberating about what to do next and in actually moving about the environment. Furthermore, it is assumed that there is no advantage to the robot in starting off in some direction until it knows the first location to be visited on its tour, and, while the robot can deliberate about any subsequent paths while traversing a path, it must complete the planning for a given path before starting to traverse it.

The two primary components of the decision making process involve generating the tour and planning the paths between consecutive locations in the tour. The first is referred to as *tour improvement* and the second as *path planning*. Boddy and Dean employ iterative refinement approximation routines for solving each of these problems. For example, the algorithm for tour improvement is based on edge-exchange as suggested by Lin and Kernighan [1973]. It produces tours that are progressively closer to an optimal tour by exchanging small sets of edges such that the length of the overall tour decreases. The mean improvement in tour length after k exchanges can be approximated by a function of the form $f(k) = 1 - e^{-\lambda k}$, where λ depends on the size of the tour. The performance profile of this algorithm is derived by gathering statistics on its performance with random test cases. The complete algorithm starts out with an initial, randomly selected tour. Given the length of some initial tour and the expected reduction in length as a function of time spent in tour improvement and some assumptions on the performance of path planning, the algorithm can find exactly how much time to devote to tour improvement in order to minimize its overall time spent in stationary deliberation and combined deliberation and traversal.

Figure 2.2.a [Boddy and Dean, 1989] shows how the expected savings in travel time increases as a function of time spent in path planning. Figure 2.2.b shows how the expected length of the tour decreases as a fraction of the shortest tour for a given amount of time spent in tour improvement. In this context, Boddy and Dean introduced the term *performance profile* that describes the expected quality of the results of an anytime algorithm as a function of run-time.

Boddy and Dean's work demonstrates the applicability of anytime algorithms to solve time-dependent planning problems. Their work has inspired my initial interest in using anytime algorithms as the components of large real-time systems. Their analysis, however, does not provide answers to several important aspects of anytime computation that are essential for operational rationality, most notably, the general issue of composition of *dependent* anytime components. Boddy and Dean raise this as an unsolved problem. They admit that, in their example, "combining expectations for the two planning algorithms is straightforward. Other problems and other decompositions will require combining expectations in different ways." Referring to the same problem, Dean and Wellman [1991] conclude that:

> "there is currently no general theory of combining anytime algorithms. For cases in which the

> decision problems are dependent, there is not a great deal that we can say."

Composability of anytime computation is the most fundamental issue in my work. The compilation of anytime algorithms presented in Chapter 5 addresses exactly this problem of combining dependent anytime algorithms.

2.3.2 Flexible computations

Horvitz [1987] suggested a decision procedure that uses an anytime algorithm or what he calls *flexible computation* as its main problem solver. As with anytime algorithms, the value of the results produced using flexible computation is a function of the time spent on the computation. Horvitz separates the notion of *object-related value* from the notion of *comprehensive value*. The former is a measure of the value of the results apart from their particular use in the system while the latter refers to the overall utility of the response.

Horvitz demonstrates the use of flexible computation in the health care domain. Given information regarding a particular patient, he produces a graph that maps computation time to the precision of the distribution for a set of possible diagnoses. The object-related value is determined by considering the expected utility of the treatment based on inexact diagnosis of a particular quality. Using this information together with information on the reduction of object-related value as a function of the delay in administering treatment, one can derive the comprehensive value of computation. The comprehensive value has a global maximum at a particular time. This is the period of time the system should spend reasoning about the diagnosis so as to maximize the value of its conclusion to the patient. Although spending additional time on the problem may further increase the precision, the comprehensive value to the user will begin to decrease.

Horvitz and Breese [1990] generalized this approach to the problem of optimizing the performance of an agent presented with a single real-time problem. In the following description of their work, I borrowed the improved notation of [Dean and Wellman, 1991]. The value of the agent's response is determined by the quality of the answer and the total amount of time that it took to produce the answer, represented as:

$$t_b + t_p + t_s$$

where t_b is the deliberation time of the base level problem solver, t_p is the preparation time for base-level reasoning spent on such activities as algorithm selection or problem reformulation, and t_s is the time spent on scheduling the base-level preparation module and base-level problem solver.

Since t_s is some small constant, the comprehensive value, V_c and the object-related value, V_o, are functions of the total time spent on preparation and base-level problem solving. The cost of time associated with the delay in decision making, V_d, is an arbitrary function depending on the domain. The comprehensive value is simply the difference between the object-related value and the cost of time. The meta-level control is designed to maximize the comprehensive value finding the appropriate t_p and t_b.

$$\max_{t_p,t_b} V_c(t_p, t_b) = \max_{t_p,t_b}[V_o(t_p, t_b) - V_d(t_s + t_p + t_b)] \tag{2.9}$$

For example, Figure 2.3 [Horvitz and Breese, 1990] shows the comprehensive value in two cases where the object-related value is modeled by a negative exponential function and the cost of time is modeled by a linear function. t_b^* is the optimal allocation of time to the base-level problem solver in each case.

The model of operational rationality adds a number of important features to Horvitz and Breese's model of flexible computation. These features are summarized below:

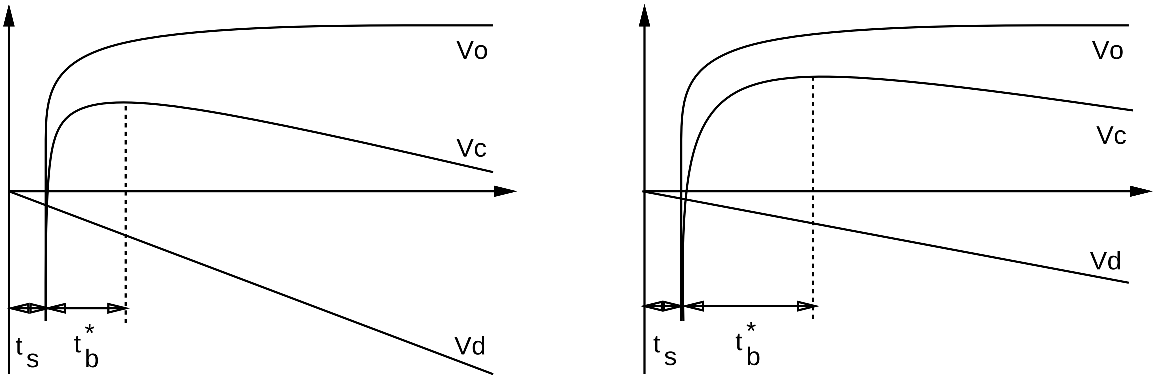


Figure 2.3: Optimal time allocation to base-level computation

1. The model of flexible computation does not address the general issue of composition of anytime algorithms. Even though it separates the preparation phase from problem solving, the model does not analyze the case where each component is an anytime algorithm and the possible effect that the quality of preparation might have on the quality of problem solving. As I mentioned earlier, it is not realistic to assume that large complex systems would be constructed based on one anytime algorithm. Hence solving the composition problem is essential for implementing any model of anytime computation.

2. The model of flexible computation assumes that the cost of delay, V_d, can be defined by a function that is independent of the other parts of the system. It uses subjective judgment in order to select such a function for any particular domain. This assumption ignores an important component of the cost of delay, what economists call the *opportunity cost*, which is the cost of choosing one course of action (continued deliberation in this case) over another (executing the current best action). The model of operational rationality does not require this assumption of a separate cost function. Its more general approach to factoring the time pressure into the deliberation process takes into account the opportunity cost.

3. Horvitz and Breese do not provide the run-time monitoring[2] mechanism that would re-evaluate the initial resource allocation in the context of the *actual* state of the world. The characterization of the object-related value of an algorithm and the cost of delay are all probabilistic and it is possible that a dynamic environment would require a faster response than originally anticipated (for example, due to unexpected deterioration in the patient's condition). It is also possible for the anytime problem solver to derive an optimal solution much faster than expected. To properly monitor resource allocation in such cases, the model of operational rationality includes a run-time monitoring component that may adjust resource allocation in response to such events.

2.3.3 Approximate processing

Lesser, Pavlin and Durfee [1988] proposed an approach for meeting real-time constraints in AI systems that is based on the following three observations:

---

[2] More recently, a monitoring component was added to the model by Horvitz and Rutledge [1991]. However, the monitoring problem addressed by this dissertation is more complicated because it may involve a large number of anytime components.

1. Time can be treated as a resource when making control decisions.
2. Plans can be used as ways of expressing control decisions.
3. Approximate processing can be used as a way of satisfying time constraints that cannot be achieved through normal processing.

Under this approach, a real-time problem solver estimates the time required to generate solutions and their quality. This estimate permits the system to anticipate whether the current objectives will be met in time. The system can then take corrective actions and form lower-quality solutions within the time constraints. These actions can involve modifying existing plans or forming different plans that utilize only rough data characteristics and approximate knowledge to achieve the desired speedup. A decision about how to change processing is situation dependent, based on the current state of processing and the domain-dependent solution criteria. The authors present a number of experiments that show how approximate processing helps a vehicle-monitoring problem solver meet its deadlines.

Lesser, Pavlin and Durfee suggest a number of general approximation techniques that offer a tradeoff between quality of results and computation time, similar to anytime algorithms. These techniques include: approximate search strategies that use corroboration and competition as criteria for pruning inferior alternatives in the search space, data approximation that limit the number of processing alternatives by taking an abstract view of data, and knowledge approximation that uses a single, less discriminating knowledge source to summarize several sources of knowledge.

Operational rationality complements this work by adding two important components: conditional performance profiles, that allow for better predictions of performance and better treatment of uncertainty, and efficient off-line compilation, that allows for better control of large systems. In addition, operational rationality offers an optimization mechanism rather than a “satisficing” criterion to measure problem-solving success.

2.3.4 Incremental approximate planning

Elkan [1990] suggested an abductive strategy for discovering and revising plausible plans. In his approach, candidate plans are found quickly by allowing them to depend on assumptions. His formalism makes explicit which antecedents of rules have the status of default conditions. Candidate plans are refined incrementally by trying to justify the assumptions on which they depend. This model was implemented by replacing the standard depth-first exploration strategy of Prolog with an iterative-deepening version. The result is an anytime algorithm for incremental approximate planning.

Elkan’s approach is hard to compare to the model of operational rationality developed here since he does not provide any quantitative analysis of his method. Such analysis would require generating the performance profile of the planner with respect to a specific problem domain[3]. The quality of an approximate plan can be measured in various ways: the probability of its correctness, the expected number of corrections needed to fix it, the expected cost of fixing it, or the expected time needed to achieve the goal using this plan. However, in Elkan’s system, that uses pure logic, it is hard to deal with such quality measures. Another problem with this approach is the fact that candidate plans that are found to be based on wrong assumptions

---

[3] Note that only one anytime algorithm is implemented, that is, the theorem prover. The performance profile of a theorem prover is inherently hard to find since it depends mostly on the input (what to prove) and the background theory (the current knowledge). A general theorem prover does not appear to be a good candidate to serve as an anytime component of a system.

Table 2.1: Approximate relational algebra operations

Approximate Operation	C_T	P_T
Union: $R_T = R_1 \cup R_2$	$C_T = C_1 \cup C_2$	$P_T = (P_1 \cup P_2) - C_T$
Difference: $R_T = R_1 - R_2$	$C_T = C_1 - R_2$	$P_T = (P_1 - R_2) \cup (P_2 \cap R_1)$
Select: $R_T = \sigma_{att=val} R_1$	$C_T = \sigma_{att=val} C_1$	$P_T = \sigma_{att=val} P_1$
Project: $R_T = \pi_{att} R_1$	$C_T = \pi_{att} C_1$	$P_T = \pi_{att} P_1$
Cart. Prod: $R_T = R_1 \times R_2$	$C_T = C_1 \times C_2$	$P_T = (R_1 \times R_2) - C_T$

are eliminated, without any process of “debugging,” and therefore the computation time that was spent on refining those plans is completely lost.

2.3.5 Anytime query answering in relational databases

Smith and Liu [1989] proposed a monotone query processing algorithm which derives approximate answers directly from relational algebra query expressions. Formally, an *approximate relation* R of a standard relation S is a subset of the Cartesian product of all the domains of S that can be partitioned into two blocks, the certain set C and the possible set P such that:

$$(1)\ C \subseteq S \quad and \quad (2)\ R = C \cup P \supseteq S \tag{2.10}$$

The algorithm assumes that the information stored in the database is complete and that the input data is precise. An incomplete answer to a query is generated when there is not enough time to complete processing the query, or because some relation that must be read to get the exact answer is not accessible. The algorithm works within the framework provided by a standard relational algebra query language and is based on an approximate relational data model.

Given a set of all approximate relations of a standard relation S, a partial order relation $\geq$ can be defined over the set as follows: the approximate relation $R_i = (C_i, P_i)$ is better than or equal to another approximate relation $R_j = (C_j, P_j)$, denoted as $R_i \geq R_j$, if $P_i \subseteq P_j$ and $C_i \supseteq C_j$. Standard relational algebra is replaced by approximate relational algebra that operates over approximate relations. The complete set of operations appear in Table 2.1. In the table, the C_T and P_T columns show the certain component and the possible component of the approximate result R_T. The operators in Table 2.1 are monotone, that is, the result of the operation is better when its operands are better [Vrbsky *et al.*, 1990].

Vrbsky and Liu have implemented the approximate query processing algorithm in a system called APPROXIMATE [Vrbsky and Liu, 1992]. The monotone query processing algorithm represents the query as a tree whose nodes represent relational operations. The operation associated with each leaf node of the tree is an approximate-read that returns a segment of the requested relation at a time. Approximate relational algebra is used in order to evaluate the tree. Initially, the certain set is empty for every approximate object and the possible set is the complete range of values for the particular object. After each approximate-read, a better approximate answer to the query is produced. The exact answer is returned if the system is allowed to run to completion. The latest, best available approximate answer is returned if query processing must be terminated before it is completed, hence the algorithm is interruptible.

APPROXIMATE demonstrates how anytime algorithms can be used for information retrieval. However, as with Elkan's approximate planning, it is difficult to derive the performance profile of the system due to its dependence on the contents of the database and the complexity of the query. It is also hard to evaluate the quality of an approximate relation and represent it quantitatively. For example, suppose that the system is presented with the query "How many free seats are available on flight EL AL 001?" and its approximate answer is "4 to 8." What is the quality of this approximate answer? Further work is required to define appropriate quality measures that would enable the construction of the performance profile of APPROXIMATE.

2.4 Design-to-time scheduling

Garvey and Lesser [1993] have developed a real-time scheduling approach called design-to-time scheduling. The methodology advocates generating the best possible solution under time pressure. In that sense, it shares the goals of operational rationality. Design-to-time also includes a monitoring component to handle the uncertainty regarding the actual quality of results produced so far. Design-to-time scheduling is based on a predefined set of solution methods with *discrete* duration and quality values, similar to the design-to-time algorithms proposed by D'Ambrosio [1989]. The overall problem is represented as a task structure in which every task may have a multiple set of dependent subtasks that can be combined to solve it. Each such set is considered as a *method* for solving the task. Two forms of approximate computation can be represented by a task structure: iterative refinement, where an approximate solution is generated quickly and can be refined through a number of iterations, and multiple methods, where a number of different algorithms are available for a task, each of which is generating a solution of a different quality. Each task group has a quality function associated with it that is based on the subtask relationship. Given a task structure, Garvey and Lesser developed a scheduling algorithm that finds execution methods for each task in the task structure, trying to maximize quality within the available time.

Design-to-time scheduling shares many of the underlying assumptions and techniques used by operational rationality. The differences are more in emphasis than in principles: design-to-time emphasizes a problem structure involving many different tasks and the solution is based on run-time scheduling. Operational rationality concentrates on a single task but it seems to handle task decomposition in a more informative manner. In particular, the use of conditional performance profiles allow for better predictions regarding the effect of approximate results in the components on the overall quality of the task. In addition, operational rationality solves the control problem by a combination of off-line compilation and run-time scheduling and monitoring.

An integration of the two approaches can be achieved in a number of ways. For example, the efficient compilation method, that will be presented in Chapter 5, can be used to derive an optimal contract algorithm for a task represented as a composite anytime module. Then, the design-to-time scheduler can be used with a number of alternative methods, each generated by a certain fixed allocation to the contract algorithm. This way, the resulting system can have the advantages of both approaches: the superior handling of interdependencies between modules and the efficient compilation of operational rationality, and the superior handling of multiple tasks with distinct temporal constraints of design-to-time scheduling. Another advantage of such integration is the reduction, through off-line compilation, of the number of tasks handled at run-time by the design-to-time scheduler.

2.5 Real-time problem solving

Real-time systems must not only produce correct results but also meet certain timing constraints. In traditional real-time systems, the timing constraints impose a *fixed* time allocation to the problem solving component so that the system can meet a certain *deadline*. For example, Laffey *et al.* [1988] define real-time systems by the capability to "guarantee a response after a fixed time has elapsed, where the fixed time is provided as part of the problem statement." This conservative approach leads to inflexible systems that may be under-utilized since in many domains there are no clear, rigid deadlines. Instead, the value of the results drops gradually over time and is situation-dependent. Operational rationality is based on a more general view of real-time systems, defined in Chapter 3, that is characterized by a time-dependent utility function. In this section I summarize related work on real-time problem solving and compare it to the operational rationality model.

2.5.1 Real-time heuristic search

Heuristic search is a fundamental problem-solving method in artificial intelligence. A single-agent search problem is characterized by an *initial state*, a set of possible actions whose application to a state generates the *successors* of that state, a *goal test* function that determines whether a particular state matches the goal, and a *heuristic function* that provides an estimate of the cost of reaching a goal state from any given state. The best known single-agent heuristic search algorithm is A*. It is a best-first search algorithm where the merit of a node, $f(n)$, is the sum of the actual cost of reaching that node from the initial state, $g(n)$, and the estimated cost of reaching a goal state from that node, $h(n)$. A* always finds the optimal solution if the heuristic function never overestimates the actual solution cost. Iterative-Deepening-A* (IDA*) [Korf, 1985] is a modification of A* that reduces its space complexity. Both A* and IDA*, however, take exponential time to run in practice. This cost of obtaining optimal solutions restrict the applicability of these algorithms.

Motivated by the observation that existing single-agent heuristic algorithms cannot be used in large-scale, real-time applications, Korf [1987, 1988, 1990] extended the standard A* algorithm so that its execution time can be controlled. The basic idea was to limit the search horizon so that the algorithm can commit to action in constant time, just like the *minimax* procedure is used in two-player games. In the single-agent case, the value of an internal node in the search tree is the minimum of all the heuristic estimates of its successors. Korf called this back-up procedure *minimin* and the pruning mechanism *alpha pruning* by analogy to *alpha-beta pruning*. The question now is how to use the minimin procedure in order to arrive at a solution. Real-Time-A* (RTA*) solves the problem using the following strategy: it uses minimin to select individual moves and backtracks to a previously visited state when the estimate of solving the problem from that state plus the cost of returning to that state is less than the estimated cost of going forward from the current state. RTA* is guaranteed to eventually find a solution under the following conditions:

Theorem 2.2 (Korf) *In a finite problem space with positive edge costs and finite heuristic values, in which a goal state is reachable from every state, RTA* will find a solution.*

An improved version of the algorithm, DTA*, was developed by Russell and Wefald [1991]. Both algorithms, however, involve a fixed depth limit as a parameter that controls the search. The constant time of move selection makes the algorithm react in "real-time." However, it does not provide any information about the quality of each step and about the time necessary to reach a solution. Therefore, real-time search algorithms can be embedded as modules in a larger system governed by an appropriate meta-reasoning component. Real-time search by itself does not provide any particular optimizing mechanism.

2.5.2 Soft real-time

Several researchers have examined the scheduling problem of real-time tasks that must meet certain timing requirements. Shih, Liu and Chung [1989, 1991] proposed a model of imprecise computation in which each task is decomposed into a *mandatory* subtask and an *optional* subtask. The mandatory subtask must be executed to produce results of any value; the optional subtask may be executed to increase the value of the results. This model is sometimes referred to as "soft real-time" as opposed to "hard real-time" where each task has a rigid deadline. Shih, Liu and Chung derived scheduling algorithms for this model assuming that the precision of the results improves either linearly or by steps through the execution of the optional subtask.

Alexander, Lim, Liu and Zhao [1992] have examined the performance of various scheduling policies for managing transient overload in an imprecise computation system. They use the same imprecise computation model. If the load on the computation system is low, the scheduler is designed to provide some prescribed balance of accuracy and response time. If the load is high, the scheduler is designed to keep response time bounded by sacrificing accuracy. The performance of the scheduler is measured using two metrics: normalized mean waiting time and normalized mean overload duration. This work demonstrates how anytime computation can be used to enable a system to maintain short waiting times when transient increase in load occurs.

Moiin and Smith [1992] generalized the imprecise computation model by allowing non-linear functions to describe the improvement of precision over time. They use an arbitrary *precision-value* function to model the precision of the results of an algorithm as a function of time. Another function, the *time-value* function, defines the value of achieving a task as a function of time. Moiin and Smith provide a case analysis of three particular functions to describe time-value (linear, quadratic and exponential decay) in conjunction with two functions to describe precision-value (linear and exponential improvement of precision). They give an approximate solution to the problem of finding an optimal schedule to a given set of tasks.

How does the work on soft real-time relate to the model of operational rationality? Optimal scheduling of a given set of *independent* jobs, each having a mandatory part and an optional part, is a simple case of anytime computation. In that sense, soft real-time addresses a small subset of the compositional language used to combine anytime algorithms. However, it is important to emphasize that scheduling imprecise computation, when the computational elements themselves arrive randomly at a certain rate, is beyond the scope of operational rationality. In operational rationality, the anytime computational components are part of a pre-determined program.

To summarize, here are the main advantages of operational rationality in solving the time allocation problem:

1. *Objective quality measures*

   The proposed "precision value" functions of the imprecise computation model are subjective. They are specified by the user and do not have a well-defined meaning. In contrast, performance profiles measure a concrete, well-defined aspect of the quality of the results. They are probability distributions calculated using concrete metrics rather than human intuition.

2. *Separation of quality and utility*

   In the model of operational rationality, quality of results and utility are separate entities. While quality is an objective measure of the performance of an algorithm, utility is an arbitrary subjective function. Utility can depend on the state of the environment as well as on the quality of results. For example, a plan for reaching a moving target has a value that depends on the new location of the target. If the target disappears, the value of the plan drops even if it is otherwise a high-quality plan. Moiin and Smith's model replaces the notion of utility by a simple "overall value of a task" which is a multiplication of the time-value, the precision-value, and, possibly, a weighting factor. This approach does not allow the overall value to be context-dependent.

3. *Composability*

   The imprecise computation model does not address the issue of composition of anytime modules. Instead, it deals with individual, independent tasks. Since the problem definition imposes timing constraints on the tasks, it allows the tasks to be *temporally dependent*, but it assumes that the results of each task and their qualities are independent. This is a major simplification that cannot be made when dealing with anytime algorithms as the components of a program to guide an artificial agent. This issue is addressed in Chapter 5.

4. *Run-time monitoring*

   The model of imprecise computation does not address the issue of uncertainty regarding the quality of the results of each imprecise task. As a result of this uncertainty, run-time monitoring is required to modify the optimal schedule as a response to the *actual* quality of results produced so far. This issue is addressed in Chapter 6.

5. *Sensing and action*

   Anytime sensing and anytime action are more complicated to analyze than anytime computation. They require an extension of the model of task execution used by Moiin and Smith. In particular, their model does not take into account the possible interaction between task execution and the environment. This issue is addressed in Chapter 7.

6. *Maximizing utility over history*

   Finally, further generalization of the imprecise computation model is needed to deal with optimization of performance over a "history" of execution of similar tasks, not just a single task or a given set of tasks. The work of Alexander *et al.* is a first step in this direction, however it addresses the case of independent tasks only. Optimization over history is another reason for active run-time monitoring. This issue is addressed in Chapter 7.

To summarize, soft real-time shares some of our motivation and goals. It solves a similar optimization problem: the problem of optimizing the execution value of a set of tasks based on subjective knowledge on their performance. Several simplifying assumptions limit the scope of soft real-time models. Most restricting is probably the assumption that there is no interaction between the tasks, and between the tasks and the environment. As a result, the model becomes inappropriate even when applied to the domain suggested in [Moiin and Smith, 1992]: a command and control system for threat analysis and target assignment. The radar component, designed to track objects, and the planning component, designed to perform target assignment, are clearly interdependent: the quality of the first clearly affects the quality of the second. The use of conditional performance profiles and dynamic scheduling seems essential for solving such problems.

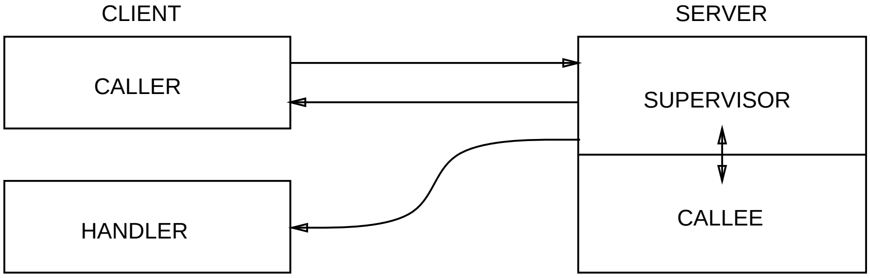


Figure 2.4: Concord process structure

2.6 System support for approximate computation

The wide-spread use of approximate computation will only happen when it becomes integrated into standard software engineering techniques. Given the importance of approximate computation in computer science in general, it is somewhat surprising that so little has been done to provide system support for models of imprecise computation. This issue was raised recently in the First IEEE Workshop on Imprecise and Approximate Computation (December 1992), and attracted much attention among the real-time programming community.

The Concord system, developed by Lin, Natarajan, Liu and Krauskopf [Lin *et al.*, 1987], is one of the few early attempts to address the issue of system support for approximate computations. A description of the system concludes this chapter.

The Concord system

Concord is a programming language that supports approximate computations. The run-time of each subroutine is controlled by the consumer of the results. Its development was motivated, like the model of operational rationality, by the problem of optimizing performance given limited computational resources. The basic assumption made by the developers of the system is that:

> "It is often enough to introduce the environment as a parameter of a computation and assume that it is unchanged throughout the computation."

Hence, a general parameter, E, is defined to be the subset of the environmental state which may affect the execution of a program P. A computation $\mathcal{C}$ is an instantiation of P by the following transition function:

$$\mathcal{C} \,:\, I \times S \times E \rightarrow O \times S \tag{2.11}$$

where S is the set of states of the program P, I is the set of input values, and O is the set of output values.

The main design issues involve the run-time environment structures needed to support flexible procedure calls. For this purpose, two new language primitives are defined. *Impresult* is used by the caller to define a handler for imprecise results and *impreturn* is used by the callee to return imprecise results. Figure 2.4 shows the data flow between the main components: the client that includes the caller and the result handler, and the server that includes the callee and its supervisor. For each procedure, a supervisor is used to record values of the approximate results obtained to date, together with a set of error indicators. When a procedure is terminated, its supervisor returns the best result found. Intermediate results are handled by the caller using a mechanism similar to exception handling. The handlers for imprecise results determine whether a result is acceptable or not; this decision is *local* to the caller, rather than being made in the context of a global utility function. In this sense, Concord actually performs some kind of satisficing rather than optimization.

The Concord model has several important disadvantages compared to the model of operational rationality. It leaves to the programmer the decision of what quality of results is acceptable; it does not mechanize the scheduling process but only provides tools for the programmer to perform this task; and it does not provide for simple cumulative development of more complex real-time systems. Nevertheless, Concord represents a pioneering project in the area of approximate computation. It is one of the first attempts to develop standard programming tools to support approximate computation.

2.7 Summary

Using the latest technology available today, real-time systems are hard to develop. Real-time AI is even harder. This view of the problem is expressed also by Laffey, Cox, Schmidt, Kao and Read [Laffey *et al.*, 1988] in their comprehensive survey of real-time AI. In their closing remarks, the authors of the survey say:

> “We concluded that one of the main reasons for this situation is that expert systems developers have often tried to apply traditional tools to applications for which they are not well suited. Tools specifically built for real-time monitoring and control applications need to be built. An immediate goal should be the development of high-performance inference engines that can guarantee response times.”

These kind of tools for real-time programming and monitoring are offered by the model of operational rationality that I have developed. This chapter shows that some aspects of the model, such as individual applications of anytime algorithms and control mechanisms for imprecise computation, have been developed independently. However, the survey of previous work shows also that none of these existing models attempted to put together all the aspects of operational rationality. The most central aspect of operational rationality, the composition of larger systems using anytime algorithms as components, has not been addressed at all. This capability, that I will present in detail in Chapter 5, is in my view a precondition to the wide spread use of anytime computation and to the simplification of the construction of real-time systems in general.

Chapter 3

Utility-Driven Real-Time Agents

> Artificial intelligence is the discipline that is concerned with programming computers to do clever, humanoid things — but not necessarily to do them in a humanoid way.
>
> Herbert A. Simon, *Models of Bounded Rationality*

What makes a system intelligent? Is it its expert-level performance in a *particular* domain or its ability to learn and improve its performance in *any* domain? Is it the system's body of knowledge and reasoning capability? Intelligence, as is well known, is easier to recognize than to define. I adopt the view of intelligent systems as agents whose intelligence is determined by the quality of their interaction with the environment. This view emphasizes the quality of the behavior of the system rather than its structure and internal mechanisms. This is the fundamental difference between artificial intelligence and cognitive science which is the study of the particular *mechanisms* of intelligent human behavior. In that sense, "artificial intelligence is a normative science of procedural rationality [while] cognitive science is a positive study of procedural rationality" [Simon, 1982]. My focus is therefore on the behavior of a system with respect to its goals. Intelligence becomes an objective measure of two factors: (1) the degree to which the system is maximizing its utility; and (2) the complexity of the environment in which the system is operating. In this chapter I spell out the fundamental problem addressed by the model of operational rationality: the construction of utility-driven real-time agents. I show how utility functions can be used for both guidance and evaluation of intelligent behavior.

3.1 Artificial agents

Any system can be viewed as an abstract artificial agent making decisions and acting in some physical or logical domain. An artificial agent is characterized by the capability to translate perceptual input into effective action that transforms the environment into a particular desired state. The set of all possible states of the environment in which the agent operates is designated by Ω. The desired states are also referred to as *goal* states. Input to the agent can be provided by an external user or it can be autonomously acquired. In both cases, the agent cab use its *sensors* in order to get information about the state of the environment. Based on this input, and its knowledge and reasoning capabilities, the agent selects a particular course of action that transforms the state of the environment so as to maximize the level of goal achievement.

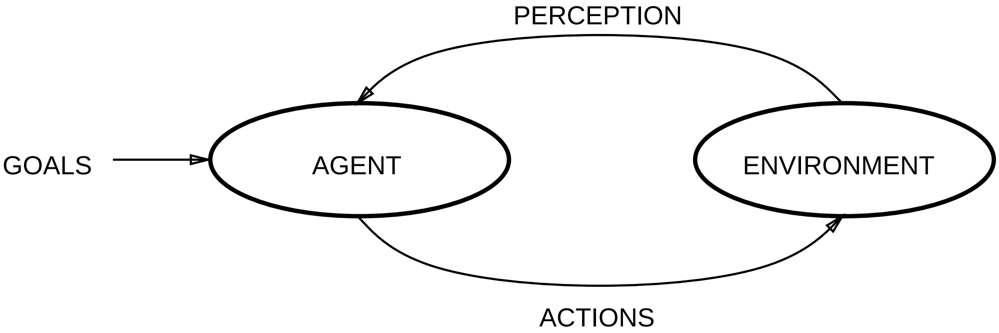


Figure 3.1: Artificial agents

Agent categories

Agents can be categorized according to their structure, that is, according to the mechanism used for action selection. If the agent follows a pre-determined strategy that states which *base-level* action to perform as a function of the state of the environment, then it is a *reactive agent*. If the agent uses any computational decision procedure, other than information retrieval, for selecting its actions, it is a *deliberative agent*. A deliberative agent that considers the outcome of actions and derives a set of constraints on future states before committing itself to an individual action is considered a *planning agent*. As the complexity of the domain of operation increases, deliberation and planning become essential parts of effective agent construction.

The distinction between reactive and deliberative agents is not always easy to define. On the borderline between the two types of agents one can find certain types of production systems that use a set of condition-action rules to generate their behavior. The rule interpreter carries out a matching operation to determine the next rule to be activated. When matching is fast and when it returns immediately the best external action, the system may be considered a reactive agent. The control structure of a production system becomes less reactive and more deliberative as matching cost grows, multiple rule matching is allowed, and certain policies are used to resolve conflicts between rules.

Russell [1989] presents a uniform view of agent deliberation that identifies six types of knowledge that agents can use. These classes of knowledge range from fully declarative to fully compiled representations. Figure 3.2 [Russell, 1989] shows the six types of knowledge, denoted by: A, B, C, D, E and F. Type A rules specify information that can be deduced about the current state. Type B rules specify information about the results of actions and their effect on the current state. Type C rules specify information regarding the utility of states. Type D rules specify the best action to be taken in certain situations. Type E rules specify utility of actions as a function of the current state. Type F rules specify the best action to be taken based on the results of actions. In addition, the decision-theoretic principle, labeled DT, uses knowledge of the utility of actions to conclude that one is the best. Based on the types of knowledge used by a particular agent, one can characterize its *execution architecture*. For example, a production system uses type D rules, and a goal-based system uses knowledge of type B and F. The combination of a number of execution architectures offers a tradeoff between execution time and decision quality [Ogasawara and Russell, 1993].

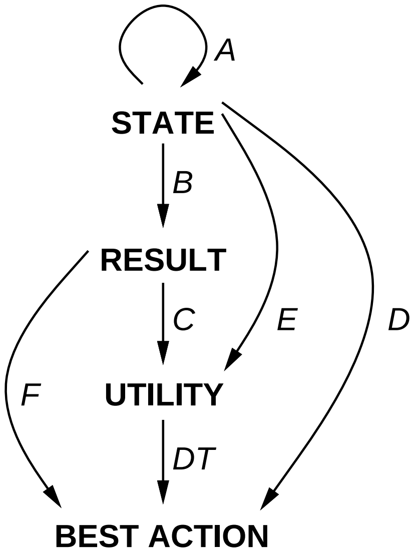


Figure 3.2: Forms of compiled and uncompiled knowledge

3.2 Goals versus utility functions

How can intelligence be measured or evaluated? Or to be more precise, given a particular agent, how can the degree of goal achievement be measured? A simple approach is based on counting the number of goals that are satisfied. For example, suppose that the set of goals includes `in(box1,room7)`, which means that the object named `box1` should be in the room named `room7`. Then the goal is achieved in any state of the environment for which the predicate is true. This has been the standard approach in early planning systems such as STRIPS and many of its descendants. With this approach there can be no partial satisfaction of a goal. An agent can either achieve a goal *completely* or fail to do so. This rigid approach is not suitable for complex situations where partial satisfaction of goals may be sufficient and even desirable given the cost of complete satisfaction. Consider, for example, the problem of finding a parking spot at a place one visits for the first time. The initial goal may be to find a free, unlimited parking space that is within a short walking distance from that place. Normally, after a short exploration period that yields no result, one settles for partial fulfillment of this goal rather than prolonging the exploration. In general, complete satisfaction may be either infeasible or uneconomical.

3.2.1 Partial goal satisfaction

Many reasons contribute to the fact that partial goal satisfaction might be preferred to complete satisfaction. The following list summarizes these reasons:

1. Problem complexity – given the agent’s computational resources, the problem complexity makes it too hard to find the best solution.
2. Cost of time – time pressure imposed on the agent does not allow enough time to find the best solution.
3. Uncertainty regarding the state of the environment – the agent cannot determine with absolute certainty that the goal was achieved because its perception of the environment provides only an approx-

imation of the current state.

4. Goal conflict – the desired goal imposes several constraints on the environment that cannot be satisfied simultaneously.

Since complete satisfaction of goals is not always possible, agents need a method for measuring *partial* satisfaction of goals so that they can maximize the degree of goal achievement. Such measure is provided by the use of *utility functions*. A utility function is a mapping from the set of states of the environment, Ω, into the set of real numbers. The utility of each state, measured in arbitrary units sometimes called *utils*, is a numeric evaluation of the degree of goal achievement in that state. For example, if the agent's goal is to deliver packages, its utility function may be:

$$\sum_{P_i \in DP} Value(P_i, T_i) \tag{3.1}$$

where T_i is the delivery time of package P_i, *DP* is the set of all delivered packages, and *Value* determines the time-dependent delivery value of each package.

Utility functions generalize the notion of goal achievement by allowing each goal to be achieved to a certain degree. The agent's set of goals is replaced by a new single goal which is to maximize its utility function. This approach solves the difficulties mentioned above, such as goal conflict, because even when two goals cannot be achieved together, the utility function can be maximized by partial satisfaction of both.

3.2.2 Explicit and implicit utility

Utility functions express the level of goal achievement for each possible state of the environment. An important distinction, that has been largely ignored in the past, should be made between *explicit* and *implicit* utility which are defined below:

Definition 3.1 *An* **explicit utility function** *over a set of states* Ω, $U_{exp} : \Omega \rightarrow \mathcal{R}$, *is a function that measures the degree of goal achievement in a state assuming that it is a terminal state.*

Definition 3.2 *An* **implicit utility function** *over a set of states* Ω, $U_{imp} : \Omega \rightarrow \mathcal{R}$, *is a function that measures the degree of goal achievement in a state assuming that it is an intermediate state in a problem solving episode.*

To understand the difference between explicit and implicit utility functions imagine an agent whose only goal is to get to from Berkeley to San Francisco. In the initial state, ω_0, the agent has $100 and a car that has only $1/8$ of a gallon of fuel. Consider the state ω_1 in which the agent is still in Berkeley, having the same amount of money and a map of local gas stations. The explicit utility of ω_1 is zero, since the agent has not advanced toward its final destination. When considered as a terminal state, ω_1 has no extra value due to the fact that the agent acquired the map. However, the implicit utility of ω_1 is high, because, having so little fuel, the agent must go directly to a gas station before it can do anything else. When considered as an intermediate state, ω_1 has some extra utility due to the fact that the agent acquired the map.

While explicit utility can be determined by a simple evaluation of certain, immediate features of the domain, such as the distance to San Francisco, implicit utility is hard to estimate. Implicit utility depends not only on the environment but also on the agent's capabilities and intentions. For example, the value of the map of local gas stations depends on the agent's capability to read a map and on its intention to get fuel before heading to San Francisco. Therefore, estimating the implicit utility of a given state may require complex planning and problem solving. I assume that normally only the explicit utility function is given as part of the problem definition. Computing the implicit utility is regarded as part of the problem solving process, not the problem definition.

Implicit utility is, in principle, a derivative of the explicit utility. For each intermediate state, the agent has expectations regarding the timing of the termination of the task and the level of goal achievement. The explicit utility of that expected termination state defines the implicit utility of an intermediate state. The utility of a particular action at any given time can also be estimated by the net effect it has on the implicit utility of the resulting state. Having made the distinction between implicit and explicit utility, I can now define the notion of a utility-driven agent.

Definition 3.3 *A* **utility-driven agent** *is an agent whose behavior is designed to maximize a given explicit utility function.*

Given an explicit utility function, as long as the agent has the capability to compute the possible effects of each action, the optimization of a single action is relatively simple. However, in order to optimize the agent's behavior over a certain segment of time, implicit utility must be computed and the task becomes more complicated. In the next chapter I will show how conditional performance profiles can simplify the task of projecting the quality of certain courses of action. The rest of this chapter discusses the role of utility functions in both guidance and evaluation of intelligent agents. Unless otherwise mentioned, the term utility will be used to indicate explicit utility.

3.3 Real-time agents

Having established a framework that evaluates intelligent agents using utility functions, I now turn to the notion of real-time agents. The classical definition of real-time systems emphasizes the capability of a system to produce its results after a fixed time has elapsed. In this work I have adopted a more general view of real-time systems that is based on the notion of time-dependent utility functions.

3.3.1 Time-dependent utility

Utility functions can be used as a rich language to describe the level of "time pressure" and its dependence on the situation. For example, a deadline can be imposed on the run-time of a system by having a sharp drop in utility at a certain point of time. If the value of a result r of a system in state s of the environment is defined by the function $V(r,s)$, and its fixed deadline is at T_0, then the following utility function can be used to capture both aspects:

$$U(r,s,t) = \begin{cases} V(r,s) & \text{if } t < T_0 \\ -\infty & \text{otherwise} \end{cases} \tag{3.2}$$

In addition to the capability to capture the notion of a deadline, utility functions allow for other types of time pressure to be described. Moreover, they allow for gradual decrease in the value of the results as a function of time. To describe this property, the following definition is used:

Definition 3.4 *A utility function $U(r,s,t)$, that measures the value of a result r in situation s at time t, is said to be* **time-dependent** *if*

$$\exists r,s,t_1,t_2 \; U(r,s,t_1) \neq U(r,s,t_2)$$

Time-dependent utility generalizes the traditional notion of a deadline and is sometimes referred to as a *soft deadline*. This approach has been used by several investigators [Horvitz, 1987; Boddy and Dean, 1989; Russell and Wefald, 1989b]. In some cases the utility function evaluates actions rather than states. For example, Horvitz and Rutledge [1991] introduce a system, Protos, to solve time-pressured medical problems. Protos can suggest a treatment by propagating observations about a patient's symptoms through a belief network. A time-dependent utility function, $u(A_iH_j, t)$, is used to specify the value of action A_i taken at time t when state H_j is true. Horvitz and Rutledge use linear and exponential functions to model the utility change over time:

$$u(A_iH_j, t) = u(A_iH_j, t_0)e^{-k_a t}$$

$$u(A_iH_j, t) = u(A_iH_j, t_0) - c_b t$$

where k_a and c_b are parameter constants derived through fitting a series of assessments to a functional form or through direct assessment. The use of time-dependent utility leads to the following definition of real-time agents:

Definition 3.5 *A utility-driven agent is said to be a* **real-time agent** *if its utility function is time-dependent.*

This definition unifies the analysis of all the systems operating under any type of time constraints, both with or without strict deadlines. Time-dependent utility provides a good mechanism to describe the time pressure in many computational tasks, for example, the computation of the next move in chess, path planning for robot control, reentry navigation for a space shuttle, financial planning and trading, and medical diagnosis in an intensive care unit. In many real-time domains there is no *fixed* deadline that should be imposed on the system. A traditional programming approach to real-time problem solving, that imposes on the system a strict deadline, must use a deadline that covers the *worst case*. But in many AI problem solving techniques there is a wide variance in computational effort between the best case and the worst case. For example, a medical diagnosis system may need to respond within 30 seconds when the patient is in critical condition but the same system may have 30 minutes if the patient condition is stabilized. Deadlines that are defined based on the worst case impose unnecessary constraints on such systems. Replacing deadlines by utility functions thus eliminates this deficiency.

3.3.2 The cost of time

An important aspect in real-time agent control is the cost of time [Russell and Wefald, 1989b]. It reflects the loss of utility due to deliberation and delay in action. The cost of time is determined by several factors: the agent's reasoning capabilities, the state of the environment, and the time-dependent utility function of the agent. In other words, the cost of time is not just the cost of computational time – it also reflects the expected utility gain due to the agent's deliberation and the expected utility loss due to the dynamics of the environment. Suppose that the function $U(r, s, t)$ is the utility of result r in situation s at time t. And let r_t be the result at time t and s_t be the state of the environment at time t. Then the cost of time is defined as follows:

Definition 3.6 *Given a utility function* $U(r, s, t)$*, the* **cost of time***,* $C(t)$*, is:*

$$C(t) = U(r_t, s_0, 0) - U(r_t, s_t, t)$$

In other words, the cost of time is the difference between the value of a result, assuming that it is available at the current state, and its value at the time it is actually produced by the system. Obviously there is a large degree of uncertainty regarding the state of the environment and the actual result generated by the system. This uncertainty is characterized respectively by the model of the environment and the conditional performance profile of the system. Hence, the *expected* cost of time can be estimated.

In many domains it is easier to estimate the cost of time directly rather than to construct a complete model of the environment. In such domains, the utility of the agent in a future state can be expressed by:

$$U(r_t, s_t, t) = U(r_t, s_0, 0) - C(t) \tag{3.3}$$

Hence, direct estimation of the cost of time can greatly simplify the meta-reasoning component. This issue is further discussed in Chapter 6.

3.4 Evaluating utility-driven agents

Utility functions are not only the key mechanism to define the *desired* behavior of an agent, but also the metric used to evaluate the quality of its *actual* behavior. Consider an agent $\mathcal{A}$ that is presented with an individual problem instance taken from a certain domain. Each problem instance includes a description of the initial state of the environment, ω_i. The agent transforms the environment, through a sequence of actions, into a final state, ω_f. I assume that if the utility function depends on time, the necessary temporal information is included in the state. A medical diagnosis program is an example of such an agent. The quality of the behavior is determined in this case by the expected utility of the final state over all possible problem instances:

$$Q(\mathcal{A}) = \sum_{\omega_i} Pr(I(\omega_i)) \sum_{\omega_f} Pr(F(\omega_f)|I(\omega_i)) U(\omega_f) \tag{3.4}$$

where $I(\omega)$ indicates that ω is the initial state and $F(\omega)$ indicates that ω is the final state. $U(\omega)$ is the explicit utility of ω. Note that this formula does not provide an effective method for agent evaluation. Further analysis is required to evaluate the probabilities that appear in the formula.

The above approach is useful when evaluating an agent based on a single problem solving instance. In many cases, however, it is more interesting to measure the performance of the agent over a longer period. Suppose that an agent is required to solve a sequence of problems. The total utility gain over a set of problems is important, not the utility gain from each individual problem instance. A robot performing local path planning as it navigates toward a certain destination is an example of such an agent. Another example is an agent that has a long term task, such as "to keep the room clean and organized." Evaluating the agent on a single action basis may not be the right thing to do. Finally, consider an agent that operates in an infinite loop or an agent whose operation time is unlimited in principle. A robot that delivers packages to different clients in a building is an example of such agent. Since packages are added to the delivery list all the time, the robot does not finish its job at any particular point. Assuming that the utility function depends on the value and urgency of each package, the quality of the behavior in this case can be measured by the average utility gain per time unit. A unifying approach to handle these cases is based on the definition of the utility over "histories" that describe the state of the environment over a particular period of time. The value of the agent depends then on its effect on the history of the environment. This approach to evaluate utility-driven agents is further discussed in Chapters 6 and 7.

In conclusion, this chapter shows how utility functions can replace simple goals in specifying the desired behavior of an agent. Furthermore, utility functions can be used to capture the time pressure in any domain and to evaluate the quality of the behavior produced by a particular agent.

Chapter 4

Anytime Computation

> Intelligence is not to make no mistakes but quickly to see how to make them good.
>
> Bertolt Brecht, *The Measures Taken*

Anytime algorithms expand upon the traditional view of a computational procedure as they offer to fulfill an entire spectrum of input-output specifications, over the full range of run-times, rather than just a single specification. Normally, the quality of the results of an anytime algorithm grows as computation time increases, hence anytime computation offers a tradeoff between resource consumption and output quality. This tradeoff plays a central role in the model of operational rationality. In this chapter I show how to develop anytime algorithms, how to calculate and represent the relationship between time allocation and quality of results, and how to actually execute them on a standard computer.

4.1 Anytime algorithms

Anytime algorithms generalize the standard call-return mechanism in computer programming. A standard procedure can be viewed as an implementation of a mapping from a set of inputs into a set of outputs. For each input that specifies a problem instance there is a particular element in the output set that is considered the *correct* solution to be returned by the procedure. Anytime algorithms can be viewed as an implementation of a mapping from a set of inputs and time allocation into a set of outputs. For each input that specifies a problem instance there is a set of possible solutions, each associated with a particular time allocation. The advantage of this generalization is that the computation can be interrupted at any time and still produce results of a certain quality, hence the name "anytime algorithm." The notion of interrupted computation is almost as old as computation itself. However, in the past, interruption was used primarily for two purposes: aborting the execution of an algorithm whose results are no longer necessary, or suspending the execution of an algorithm for a short time because a computation of higher priority must be performed. Anytime algorithms offer a third type of interruption: interruption of the execution of an algorithm whose results are considered "good enough" by their consumer.

Although the results produced by an anytime algorithm may be very useful, they are not considered "correct" in the traditional sense. The binary property of correctness must be replaced by a more flexible measure that characterizes the quality of each result.

4.1.1 Measuring quality of results

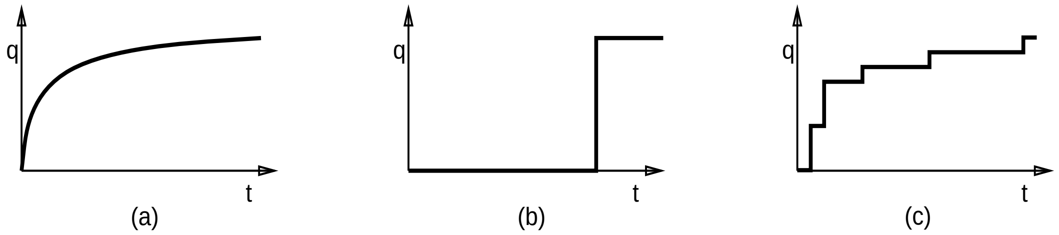


Figure 4.1: Typical performance profiles

The quality of the results produced by an anytime algorithm is characterized by its *performance profile*[1], which describes how the quality of the results depends on run-time. Any algorithm – both standard and anytime – has a performance profile. Figure 4.1 shows a typical performance profile of an anytime algorithm (a) and of a standard algorithm (b). The performance profile of the anytime algorithm shows that the quality of the results improves gradually over time, while with the standard algorithm, no results are available until its termination at which point the exact result is returned. Obviously, this example represents an ideal situation. In practice, the improvement in quality of an anytime algorithm may look like a step function, as in Figure 4.1 (c), rather than a smooth curve. In addition, there may be some uncertainty regarding the actual quality of the results for any particular time allocation. These issues and other aspects of representation of performance profiles are discussed in Section 4.2.

In order to draw performance profiles, objective metrics for measuring the quality of the output must be defined. Such quality measures specify the difference between the approximate result and the exact result. They are "objective" in the sense that they are a property of the algorithm itself, independent of its possible applications. Objective quality measures should not be confused with subjective utility functions that are also used in our model. The former are used to characterize the performance of an algorithm in absolute terms and the latter are used to define the desirability of the output of the complete system with respect to its design goals. Since the results of one anytime algorithm can be used as the input of another algorithm, the same quality measures that are attached to the results are used to characterize the possible variability in input quality.

From a pragmatic point of view, it may seem useful to define a single type of quality measure to be applied to all anytime algorithms. Such a unifying approach may simplify the meta-level control. However, in practice, different types of anytime algorithms tend to approach the exact result in completely different ways. Quality measures must match the nature of the algorithm they describe. As a result, the model of operational rationality allows arbitrary quality metrics to be used. In particular, the following three metrics where found useful:

1. **Certainty** – This metric reflects the degree of certainty that a result is correct. The degree of certainty can be expressed using probabilities, fuzzy set membership, or any other method of expressing uncertainty. For example, consider an anytime diagnosis algorithm that is based on combining more and more evidence as computation time increases. The certainty that the diagnosis is correct increases as a function of run-time. With this type of anytime algorithms, there is always a possibility that the correct results are *completely* different from the ones generated by the algorithm.

2. **Accuracy** – This metric reflects the degree of accuracy or how close is the approximate result to

---

[1] An exact definition of performance profiles will be given later in this chapter.

the exact answer. Normally with such algorithms, high quality provides a *guarantee* that the error is below a certain small upper bound. For example, Taylor series can be used for calculating the value of a certain function. The basic idea is to approximate the function by a polynomial in such a way that the resulting error is within some specified tolerance. Lagrange's remainder formula can be used to determine an upper bound on the error as a function of the iteration number. This error estimate determines the quality of the results.

3. **Specificity** – This metric reflects the level of detail of the result. In this case, the anytime algorithm always produces *correct* results, but the level of *detail* is increased over time. For example, consider a hierarchical planning algorithm that first returns a high level abstract plan. Each step in the abstract plan is a "macro" step that needs to be refined by further planning. As computation time increases, the level of detail in increased until the plan is composed of base-level steps only that can be easily followed. A detailed plan can be executed faster than an abstract plan and has higher quality.

When a particular anytime algorithm is constructed, it is often hard to classify uniquely its quality measure. Accuracy is typically used to measure quality in numerical domains and specificity in symbolic domains, but the former can be seen as a special case of the latter; an inaccurate numerical solution is very specific but incorrect, and could be mapped to an equally useful, correct statement that the solution lies within a certain interval. In addition, anytime algorithms can have multidimensional quality measures. For example, PAC algorithms for inductive learning are characterized by an uncertainty measure, δ, and a precision measure, ϵ. The advantage of the use of conditional performance profiles, as described in the next chapter, is that they allow for a uniform treatment of anytime algorithms, regardless of the particular type of their quality measure.

4.1.2 Interruptible versus contract algorithms

An important distinction should be made between two types of anytime algorithms: *interruptible* algorithms and *contract* algorithms. Interruptible algorithms produce results of the "advertised quality" even when interrupted unexpectedly; whereas contract algorithms, although capable of producing results whose quality varies with time allocation, must be given a particular time allocation in advance. If a contract algorithm is interrupted at any time shorter than the contract time, it may yield no useful results. Both interruptible and contract algorithms have been used in the past. Dean and Boddy's [1988] definition of anytime algorithms refers to the interruptible case. Korf's RTA* [1988] performs a depth-first or best-first search within a predetermined search horizon that is computed from the time allocation provided, and can therefore be considered a contract algorithm. Although this algorithm can produce results for any given time allocation, if it is interrupted before the expiration of the allocation, it may yield no results.

In general, every interruptible algorithm is trivially a contract algorithm, but the converse is not true. Intuitively, one tends to think about anytime algorithms as interruptible, whereas the greater freedom of design makes it easier to construct contract algorithms than interruptible ones. In the case of functional composition, for example, the construction of optimal contract algorithms can be solved by an efficient compilation process, while the construction of interruptible algorithms is much harder. Since in many domains with high time pressure the main decision component of an agent must be interruptible, the following reduction theorem is essential for the model of operational rationality. The reduction theorem allows for the construction of contract algorithms as an *intermediate* step, before the system is made interruptible.

Theorem 4.1 (Reduction) *For any contract algorithm $\mathcal{A}$, an interruptible algorithm $\mathcal{B}$ can be constructed such that for any particular input $q_{\mathcal{B}}(4t) \geq q_{\mathcal{A}}(t)$.*

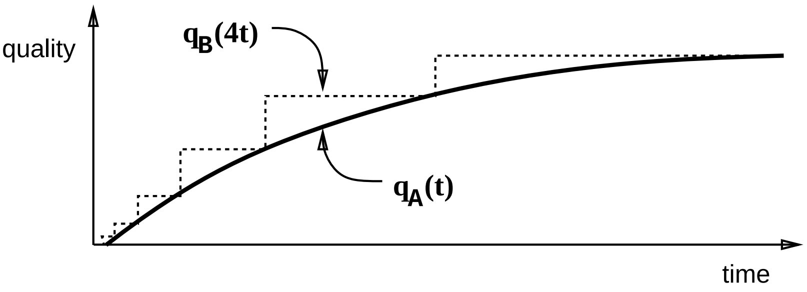


Figure 4.2: Performance profiles of interruptible and contract algorithms

Proof: Construct $\mathcal{B}$ by running $\mathcal{A}$ repeatedly with exponentially increasing time limits. If interrupted, return the best result generated so far. Let the sequence of run-time segments be $\tau, 2\tau, ..., 2^i\tau, ...$, and assume that the time overhead of the code required to control this loop can be ignored. Note also that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$. The worst case situation occurs when $\mathcal{B}$ is interrupted after almost $(2^n - 1)\tau$ time units, just before the last iteration terminates and the returned result is based on the previous iteration with a run-time of $2^{n-2}\tau$ time units. Since $\frac{2^n-1}{2^{n-2}} < 4$, the factor of 4 results. If one replaces the multiplier of time intervals by α, one gets a time ratio of: $\frac{\alpha^n - 1}{\alpha^{n-1} - \alpha^{n-2}}$. The lower bound of this expression is 4, for $\alpha = 2$, hence 2 is the optimal multiplier under this strategy. □

Note that τ may be arbitrarily small and should be in general the shortest run-time for which there is any improvement in the quality of the results of $\mathcal{A}$. Note also that the reduction theorem makes no assumption about the timing of the interrupt. When such information is available, for example if run-time is evenly distributed between 5 and 15 seconds, then a different scheduling scheme might be better.

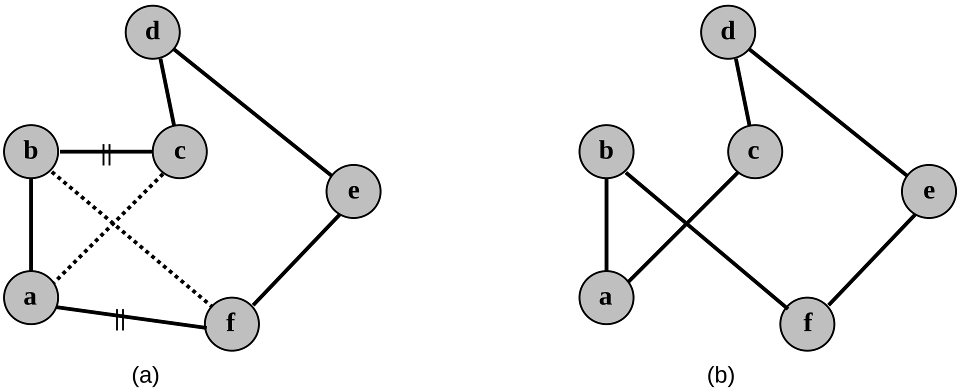
Figure 4.2 shows a typical performance profile for the contract algorithm $\mathcal{A}$, and the corresponding performance profile for the constructed interruptible algorithm $\mathcal{B}$, reduced along the time axis by a factor of 4.

As an example, consider the application of this construction method to Korf's RTA*, a contract algorithm. As the time allocation is increased exponentially, the algorithm will increase its depth bound by a constant; the construction therefore generates an iterative deepening search automatically.

4.1.3 The anytime traveling salesman

I now turn to an example of a particular anytime algorithm for solving a well-known combinatorial problem. The *traveling salesman problem* (TSP) involves a salesman that must visit n cities. If the problem is modeled as a complete graph with n vertices, the solution becomes a *tour*, or Hamiltonian cycle, visiting each city exactly once, starting and finishing at the same city. The cost function, $Cost(i, j)$, defines the cost of traveling directly from city i to city j. The problem is to find an optimal tour, that is, a tour with minimal total cost. The TSP is known to be NP-complete [Garey and Johnson, 1979]. Since all the known algorithms for solving this problem require exponential time in the worst case, it is impossible to find an optimal tour when the problem includes a large number of cities. Several efficient approximate algorithms have been developed for the TSP. Some, based on finding a minimum spanning tree, do not have the property of gradual improvement. Others are based on iterative gradual improvement. Such an interruptible anytime algorithm is described below.

The anytime traveling salesman algorithm is a randomized algorithm that repeatedly tries to perform a tour improvement step [Lin and Kernighan, 1973; Lawler *et al.*, 1987]. In the general case of tour



(a) (b)

Figure 4.3: The operation of randomized tour improvement

```
ANYTIME-TSP(V, iter)
    Tour ← INITIAL-TOUR(V)
    cost ← COST(Tour)
    REGISTER-RESULT(Tour)
    for $i \leftarrow 1$ to iter
        $e_1$ ← RANDOM-EDGE(Tour)
        $e_2$ ← RANDOM-EDGE(Tour)
        $\delta$ ← COST(Tour) − COST(SWITCH(Tour, $e_1$, $e_2$))
        if $\delta > 0$ then
            Tour ← SWITCH(Tour, $e_1$, $e_2$)
            cost ← cost − $\delta$
            REGISTER-RESULT(Tour)
    SIGNAL(TERMINATION)
    HALT
```

Figure 4.4: The anytime traveling salesman algorithm

improvement procedures, r edges in a feasible tour are exchanged for r edges not in that solution as long as the result remains a tour and the cost of that tour is less than the cost of the previous tour. I have implemented the algorithm for the case where $r = 2$. Figure 4.3 demonstrates one step of tour improvement. An existing tour, shown in part (a), visits the vertices in the following order: *a, b, c, d, e, f.* The algorithm selects two random edges of the graph, (b, c) and (f, a) in this example, and checks whether the following condition holds:

$$Cost(a, c) + Cost(f, b) < Cost(b, c) + Cost(f, a) \tag{4.1}$$

If this condition holds, the existing tour is replaced by the new tour, shown in part (b), *a, c, d, e, f, b, a.* The improvement condition guarantees that the new path has a lower cost. The algorithm starts with a random tour that is generated by simply taking a random ordering of the cities. Then the algorithm tries to reduce the cost by a sequence of random improvements. The result is an interruptible anytime algorithm shown in Figure 4.4. The performance profile of this algorithm will be presented in the following section.

4.2 Performance profiles

Performance profiles provide the crucial meta-level knowledge in the model of operational rationality. When an anytime algorithm is activated with a particular time allocation, the quality of its result falls within a certain range of possible values. The main reason for the uncertainty concerning the quality of the results, especially with deterministic algorithms, is the fact that the particular input to the algorithm is unknown. The performance profile specifies the quality distribution for any given time allocation. This quality distribution should always be interpreted with respect to a particular probability distribution of input instances. An algorithm may have several performance profiles, each characterizing its performance when operating in a different environment. For example, a particular path planning algorithm may have different performance profiles when applied to the corridors of a hospital and to a section of a warehouse. This section defines three types of performance profiles and discusses methods for their calculation and representation.

4.2.1 Categories of performance profiles

Given an anytime algorithm $\mathcal{A}$, let $q_{\mathcal{A}}(x, t)$ be the quality of results produced by $\mathcal{A}$ with input x and computation time t; let $q_{\mathcal{A}}(t)$ be the expected quality of results with computation time t; and let $p_{\mathcal{A},t}(q)$ be the probability (density function in the continuous case) that $\mathcal{A}$ with computation time t produces results of quality q. The most informative type of performance profile used in this work is the performance distribution profile defined below:

Definition 4.2 *The* **performance distribution profile (PDP)***, of an algorithm $\mathcal{A}$ is a function $D_{\mathcal{A}} : \mathcal{R}^+ \rightarrow Pr(\mathcal{R})$ that maps computation time to a probability distribution of the quality of the results.*

It may happen that the summation over all possible inputs produces too wide a range of qualities in which case the information provided by the performance profile is too general. In that case, one can use a *conditional performance profile* by partitioning the input domain into classes and storing a separate profile for each input class. The partitioning can be done using any attribute of the input that may influence performance, such as size or a complexity measure. Input classes of similar performance can also be derived automatically using Bayesian statistics by programs such as Autoclass [Cheeseman *et al.*, 1988]. Additional types of conditional performance profiles, used for compilation purposes, are discussed in Chapter 5.

Definition 4.3 *The* **expected performance profile (EPP)***, of an algorithm* $\mathcal{A}$ *is a function* $E_{\mathcal{A}} : \mathcal{R}^{+} \rightarrow \mathcal{R}$ *that maps computation time to the expected quality of the results.*

An expected performance profile is the most compact representation of performance information. It is the kind of performance information that was used by Boddy and Dean [1989] and by Horvitz [1987]. Note that:

$$E_{\mathcal{A}}(t) = \sum_{q} p_{\mathcal{A},t}(q)q = \sum_{x} Pr(x)q_{\mathcal{A}}(x,t) \tag{4.2}$$

Expected performance profiles are especially useful when the variance of the quality distribution is small. In such case,

$$f(E(q_1), E(q_2)) \approx E(f(q_1, q_2)) \tag{4.3}$$

hence expectations can be combined with high accuracy using expected performance profiles. In the special case where the variance of the distribution is zero (or infinitesimal), the anytime algorithm is said to has a *fixed performance*. For such algorithms, an expected performance profile offers a complete, accurate description of performance.

Definition 4.4 *The* **performance interval profile (PIP)***, of an algorithm* $\mathcal{A}$ *is a function* $I_{\mathcal{A}} : \mathcal{R}^{+} \rightarrow \mathcal{R}\times\mathcal{R}$ *that maps computation time to the upper and lower bounds of the quality of the results.*

Note that if $I_{\mathcal{A}}(t) = [L, U]$ then:

$$\forall x : L \leq q_{\mathcal{A}}(x,t) \leq U \tag{4.4}$$

Performance interval profiles offer a representation that is both compact and easy to manipulate. From the lower bounds on the qualities of the results of two algorithms, one can normally find a lower bound on the quality of their combined result. The same is not true of expected performance profiles. Hence, when a compact representation is preferred and the variance of the distribution is wide, performance interval profiles are useful.

4.2.2 Properties of performance profiles

I now turn to the definition of some basic properties of anytime algorithms and their performance profiles.

Definition 4.5 *The* **completion time** *of an anytime algorithm* $\mathcal{A}$ *is* t_c*, if* t_c *is the minimal time for which:*

$$\forall x\, \forall t : t > t_c \rightarrow q_{\mathcal{A}}(x,t) = q_{\mathcal{A}}(x,t_c)$$

Note that while the quality of results advertised by an expected performance profile is not guaranteed in general, it is guaranteed at completion time. This is an immediate consequence of the definition. It also reflects the intuitive notion of the completion of a computation. If t is the run-time for which the expected quality is maximal, then a time allocation of $t + \delta$ is required in order to guarantee that quality, where δ depends on the performance distribution of the algorithm. Note also that the quality of results achieved by an algorithm at its completion time is not necessarily equivalent to the quality of an optimal solution. Anytime algorithms are not required to return optimal solutions for a large, even infinite, time allocation. In many cases the flexibility offered by the algorithm is achieved at the expense of termination with sub-optimal results. For any given decidable problem, it is possible to “fix” an anytime algorithm so that it returns an optimal result in the limit. This can be achieved by simply switching from the anytime algorithm to a standard optimal algorithm after its termination (or at an earlier point). However, convergence to optimal results in the limit has very limited relevance to the construction of real-time systems.

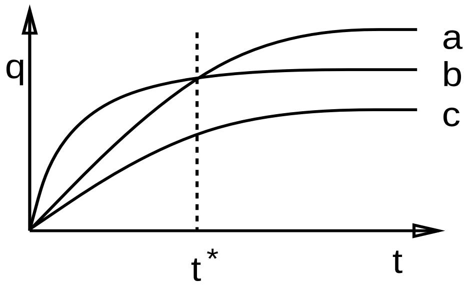


Figure 4.5: Superior performance profiles

Definition 4.6 *An anytime algorithm $\mathcal{A}$ is said to be* **pathological** *if its expected quality of results is not a monotonic non-decreasing function. That is,*

$$\exists t_1 \, \exists t_2 : t_1 < t_2 \;\; \wedge \;\; E_{\mathcal{A}}(t_1) > E_{\mathcal{A}}(t_2)$$

Unless otherwise mentioned, I assume that an algorithm is not pathological. Pathology in anytime computation can be easily removed when the quality of the results is simple to calculate. In such cases, the algorithm can be modified to return the best result generated so far instead of the most recent one. This modification clearly makes the algorithm non-pathological. For example, consider a path planning algorithm in which the quality of a generated path is determined by its length. Since the quality is simple to compute, it is easy to guarantee that the algorithm is not pathological. However, in chess playing programs, the quality of a move is not easily recognizable. That is exactly why extensive search is necessary to evaluate possible moves. In such cases fixing a pathological algorithm is much harder [Nau, 1983].

Definition 4.7 *A quality function is said to be* **normalized** *if the quality of an optimal result is 1 and:*

$$\forall x \, \forall t : 0 \leq q_{\mathcal{A}}(x,t) \leq 1$$

When the quality of results measures uncertainty using standard probabilities, the performance profile is trivially normalized. Error bounds can be normalized using a relative rather than an absolute measure. Levels of specificity can be normalized by measuring their relative contribution to the quality of the optimal (i.e. most specific) solution. Normalization of quality is sometimes useful for theorem proving purposes. However, in practice there is normally no need to limit an application to normalized performance profiles.

Definition 4.8 *Let $\mathcal{A}$ and $\mathcal{B}$ be two anytime algorithms that solve the same problem, then $\mathcal{B}$ is said to be* **superior** *to $\mathcal{A}$ if for every input x and every time allocation t:*

$$\forall x \, \forall t : q_{\mathcal{B}}(x,t) \geq q_{\mathcal{A}}(x,t)$$

The relationship of superiority between anytime algorithms is a partial order. Given two anytime algorithms that solve a certain problem, it is possible that neither of them is superior to the other. For example, in Figure 4.5, both performance profiles a and b are superior to c, but neither is a superior to b nor is b superior to a. To decide which one is "better," more knowledge on the distribution of time allocation is required. Given such information, one can compute the expected quality over all possible allocations and use that figure as a basis for comparison. Suppose that in a particular environment the function $f(t)$ is the density function for time allocations. That is,

$$Pr[T_1 \leq t \leq T_2] = \int_{T_1}^{T_2} f(t)dt \tag{4.5}$$

Given such a density function, any two performance profiles can be compared:

Definition 4.9 *Let $\mathcal{A}$ and $\mathcal{B}$ be two anytime algorithms that solve the same problem, then $\mathcal{B}$ is said to be* **stochastically superior** *to $\mathcal{A}$ over the time interval $[T_1, T_2]$ if:*

$$\int_{T_1}^{T_2} f(t)q_{\mathcal{B}}(t)dt \geq \int_{T_1}^{T_2} f(t)q_{\mathcal{A}}(t)dt$$

Definition 4.10 *Let $\mathcal{A}$ and $\mathcal{B}$ be two anytime algorithms that solve the same problem, then $\mathcal{B}$ is said to be* **ϵ-superior** *to $\mathcal{A}$ if for every input x and every time allocation t:*

$$\forall x \; \forall t : q_{\mathcal{B}}(x, t + \epsilon) \geq q_{\mathcal{A}}(x, t)$$

Definition 4.11 *Let $\mathcal{A}$ and $\mathcal{B}$ be two anytime algorithms that solve the same problem, then $\mathcal{B}$ is said to be* **equivalent** *to $\mathcal{A}$, denoted $\mathcal{B} \approx \mathcal{A}$, if there exists a small constant $\epsilon > 0$ such that $\mathcal{A}$ is ϵ-superior to $\mathcal{B}$ and $\mathcal{B}$ is ϵ-superior to $\mathcal{A}$.*

Proposition 4.12 *Let $\mathcal{A}$ and $\mathcal{B}$ be two anytime algorithms that solve the same problem with fixed performance. If neither algorithm is superior to the other, then there exists a contract anytime algorithm $\mathcal{C}$ that is ϵ-superior to both $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{C}$ is called the* **merger** *of $\mathcal{A}$ and $\mathcal{B}$.*

Proof: Construct $\mathcal{C}$ as follows:

```
$\mathcal{C}(x)$
    $t \leftarrow$ GET-TIME-LIMIT
    if $q_{\mathcal{A}}(t) > q_{\mathcal{B}}(t)$
        then AT($\mathcal{A}(x)$, CONTRACT, t)
        else AT($\mathcal{B}(x)$, CONTRACT, t)
```

The merger algorithm $\mathcal{C}$ checks first which algorithm produces a better result for the particular contract time. This decision is made based on the expected performance profiles of $\mathcal{A}$ and $\mathcal{B}$. Then it simply activates the better algorithm for that particular allocation. Since both $\mathcal{A}$ and $\mathcal{B}$ have fixed performances, $\mathcal{C}$ is guaranteed to produce results superior to both. □

Note that $\mathcal{A}$ and $\mathcal{B}$ can be either contract or interruptible algorithms since they are activated by $\mathcal{C}$ in contract mode. The reason why $\mathcal{C}$ is only ϵ-superior to $\mathcal{A}$ and $\mathcal{B}$ is because of the short additional time necessary to determine which algorithm should be activated.

4.2.3 Finding the performance profile of an algorithm

Suppose that a certain anytime algorithm is implemented on a certain machine. How can one determine its performance profile? In some cases, the performance profile can be calculated by performing a *structural analysis* of the algorithm. For example, in many iterative algorithms, such as Newton's method, the error

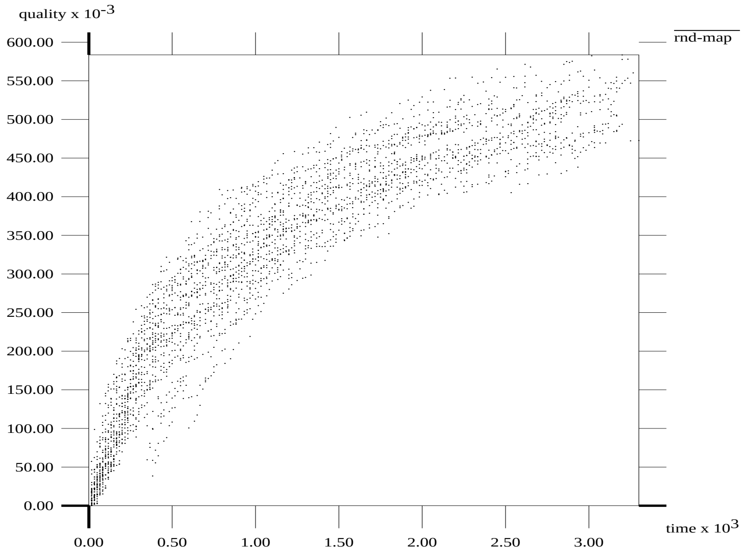


Figure 4.6: The quality map of the TSP algorithm

in the result is bounded by a function that depends on the number of iterations. In such cases, the performance profile can be calculated once the run-time of a single iteration is determined. In general, however, such structural analysis of the code is hard because the improvement in quality in each iteration and its run-time may be unpredictable. The randomized tour improvement algorithm from the previous section illustrates this problem. To overcome this difficulty, a general *simulation* method can be used. It is based on gathering statistics on the performance of the algorithm in many representative cases[2]. A third *adaptive* method combines simulation and learning. The system starts with an approximate performance profile that is determined using simulation with a limited number of examples. Then, as the system interacts with the environment, it updates the performance profile based on its experience. The advantage of this method is that it is automatically biased to measure the performance of the algorithm in the context of the *actual* application domain and using real input instances.

A quality map of an anytime algorithm summarizes the results of running the algorithm with randomly generated input instances. For example, Figure 4.6 shows the quality map of the randomized TSP algorithm. Each point (t, q) represents an instance for which quality q was achieved with run-time t. The quality of results in this experiment measures the percentage of tour length reduction with respect to the initial tour.

These statistics form the basis for the construction of the performance profile of the algorithm. The resulting expected performance profile is shown in Figure 4.7. Table 4.1 shows a tabular representation of the probability distribution profile of the algorithm.

4.2.4 Representation of performance profiles

Performance profiles can be represented either by a closed formula or as a table of discrete entries. This section discusses the two alternative representations.

---

[2] Representative problem instances are randomly generated based on prior knowledge of the problem domain.

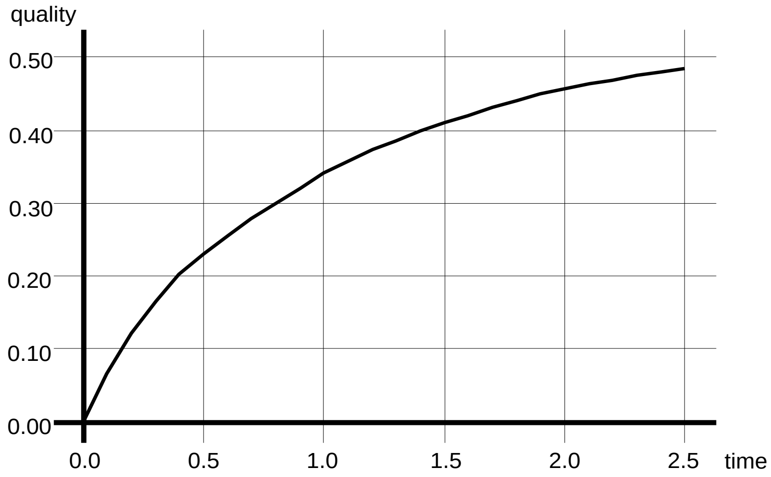


Figure 4.7: The expected performance profile of the TSP algorithm

Closed formula representation

Since performance profiles are normally monotone functions of time, they can be approximated using a certain family of functions. Once the quality map is known, the performance information can be derived by various curve fitting techniques. For example, Boddy and Dean [1989] used the function: $Q(t) = 1 - e^{-\lambda t}$ to model the expected performance of their anytime planner. Performance distribution profiles can be approximated using a similar method by using a certain family of distributions. For example, if the normal distribution is used, one can apply the same curve fitting techniques to approximate the mean and variance of the distribution as a function of time.

The advantage of using a closed formula representation of performance profiles is that symbolic compilation can be performed once a parametric representation of each profile is given. The result of such compilation can be used each time members of that family are compiled.

Closed formula representation has two major disadvantages:

1. It introduces error in performance information that is caused by the bias toward a certain family of functions.

2. It is hard to maintain closure under the compilation operation.

The problem of closure under compilation is especially important. The closure property requires that the result of compilation of two (or more) performance profiles that belong to a certain family would be a member of the same family (or at least it could be approximated by a function in that family). For example, when compiling two linear performance profiles one gets a non-linear performance profile and a linear approximation may not be sufficient (see Chapter 5). Hence, linear performance profiles are not closed under compilation. As a result of the disadvantages of closed formula representation, I prefer the use of the more flexible, discrete representation.

Discrete representation

The discrete representation of performance profiles is based on a table that specifies the discrete probability distribution of quality for certain possible time allocations. For this purpose, the complete range of qualities

Table 4.1: The performance distribution profile of the TSP algorithm

	quality											
time	.025	.075	.125	.175	.225	.275	.325	.375	.425	.475	.525	.575
0.0	1.00											
0.2	0.02	0.30	0.48	0.16	0.04							
0.4		0.04	0.12	0.24	0.36	0.24						
0.6			0.04	0.10	0.30	0.34	0.22					
0.8				0.02	0.16	0.34	0.30	0.14	0.04			
1.0					0.02	0.18	0.38	0.26	0.16			
1.2						0.06	0.24	0.40	0.28	0.02		
1.4							0.10	0.40	0.42	0.08		
1.6							0.04	0.30	0.44	0.20	0.02	
1.8								0.10	0.54	0.32	0.04	
2.0									0.44	0.48	0.08	
2.2									0.28	0.52	0.18	0.02
2.4									0.16	0.50	0.30	0.04

has to be divided into discrete qualities $q_1, ..., q_n$. The entry i, j in the table represents the discrete probability that with time allocation t_i the actual output quality q would be in the range $[q_j - \delta, q_j + \delta]$. The size of the table is a system parameter that controls the accuracy of performance information. Linear interpolation is used to find the quality when the run-time does not match exactly one of the table entries. For example, Table 4.1 shows the performance distribution profile of the randomized TSP algorithm.

4.2.5 The library of performance profiles

The development of an anytime algorithm is not considered complete before its performance profile is calculated and stored in the *anytime library*. This library keeps the meta-level information that is essential both for off-line compilation and for run-time monitoring. The construction of a standard anytime package that comes with a library of performance profiles is an important first step toward the integration of anytime computation into standard software engineering techniques. Behind such a library lies a vision of the widespread use of standard anytime algorithms for essentially every basic computational problem from sorting and searching to graph algorithms. The anytime library offers a set of reusable real-time programs, a notion that is almost self-contradictory with respect to current methodologies of developing real-time systems. Together with automatic compilation and monitoring, such a library can greatly simplify and accelerate the development of real-time systems.

Several important implementation issues regarding the construction of the anytime library have been ignored in the prototype implementation of the model. These issues include naming conventions and interface specification. Proper naming conventions are needed so that performance profiles are easily matched with the algorithms that they describe. Performance profiles in the library should be machine independent. They should specify performance with respect to a standard *virtual machine*. The library must also include information about relative performance of various computer systems so that the generic performance profile from the library can be *stretched* to describe the actual performance on a particular system. In addition, a standard set of interface operations should be defined for modification, information retrieval, and display of the entries of the library. The particular details of the implementation depend on the programming environment used and are beyond the scope of this dissertation.

4.3 Model of execution

How can an anytime algorithm be executed on a standard computer with minimal programmer intervention? In this section I define a model of execution for anytime computation that solves this problem. The crucial part of the model is the capability to communicate with a running algorithm, examine the quality of its results, and control its run-time accordingly. I limit the discussion to the execution of elementary anytime algorithms that do not include anytime algorithms as components. The treatment of compound algorithms is based on compilation and monitoring methods that are described in the following chapters.

4.3.1 Design goals

The model of execution was developed with the following design goals:

1. Anytime programs should include a minimal amount of "special code" to support the model. Such special code may be necessary in order to have access to the status and results of an algorithm while it is running, to activate an algorithm with a particular time allocation, to interrupt an algorithm and use its best result, or to find the performance profile. The minimal extra code should not make anytime programs hard to read and understand. Whenever possible, code to support special operations should be inserted automatically when an anytime algorithm is defined.

2. The programmer should not be responsible for inserting the code for time measurements or time allocation. All measurements should be performed automatically by the system. The responsibility of the programmer should be limited to qualitative aspects of algorithm development while the quantitative aspects of control and performance evaluation would be automatic.

3. The programmer should have to develop only one version of the program. None of the operations mentioned above, such as finding the performance profile of an elementary anytime algorithm, should require the programmer to alter the source code. Such modifications of code should be performed, whenever necessary, by automatic tools.

4. The programmer should be able to test individual anytime algorithms without activating the complete system. The run-time monitor developed to control the complete system should also be able to control individual algorithms. This may require the programmer to define an appropriate context that includes a utility function and a degenerate environment. This issue will be discussed in Chapter 6.

4.3.2 Assumptions about the programming environment

The model that is presented here can be realized in any programming environment that satisfies the following requirements:

1. The underlying programming language, $\mathcal{PL}$, must support the following features:
    (a) Functions are first class objects.

   (b) Functions can take optional and keyword arguments.
   (c) Execution is deterministic over time. That is, the run-time of any deterministic function is consistent over repeated activations with the same input.

2. The programming environment must include a real-time operating system, $\mathcal{OS}$, that supports the following operations:
   (a) A program can create processes and control their execution.
   (b) The scheduling of processes is based on priorities. At each point of time the process with the highest priority among all the ready processes is running.
   (c) The system maintains a real-time clock.
   (d) A process can sleep until a certain event occurs. The process becomes ready immediately when the event occurs.
   (e) Events can be triggered by any process or by the real-time clock.
3. The scheduler of processes must have the following properties:
   (a) It is an event-driven scheduler. If no event occurs, the process with highest priority remains active. If two ready processes have the same priority, one of them is randomly selected for execution.
   (b) When a process is running and a process with higher priority becomes ready, the latter becomes immediately active.
   (c) The effect of the overhead of the scheduler on the performance profiles of the running algorithms is negligible.

Note that while the simulated prototype of the model was implemented in Allegro Common Lisp, the actual model cannot be implemented in that language because it fails to satisfy the following assumptions: 1.c, 2.b, 2.d, 3.a, and 3.b. Obviously, the built-in garbage collection mechanism of Lisp makes it difficult to use that language for time-critical applications. Several existing programming environments, such as the Spring kernel [Stankovic and Ramamritham, 1989], that were especially designed for real-time applications, satisfy the requirements listed above.

4.3.3 Running elementary anytime algorithms

I now turn to a description of the model of execution with respect to the development of elementary anytime algorithms. As I mentioned earlier, the discussion here is limited to elementary anytime algorithms defined as follows:

Definition 4.13 *An* **elementary anytime algorithm** *is an anytime algorithm whose implementation does not use any other anytime algorithm as a component. A non-elementary anytime algorithm is also called a* **compound** *algorithm.*

Elementary anytime algorithms are standard programs in $\mathcal{PL}$. What makes them anytime algorithms is the mode of activation and the use of the special function: REGISTER-RESULT($Result$) whenever a new result is generated. This function records the new result so that it becomes available in case of interruption. It also

```
Foo(Input)
    Result ← Initialization-Step(Input)
    Register-Result(Result)
    while Continuation-Condition do
        Result ← Improvement-Step(Result)
        Register-Result(Result)
    Signal(TERMINATION)
    Halt
```

Figure 4.8: Typical implementation of an interruptible anytime algorithm

updates the status of the anytime algorithm. The registration operation is a fast atomic (non-interruptible) operation since interruption may create an inconsistent situation in which only part of the new result has been copied.

Practical experience shows that most elementary anytime algorithms are developed using a general interruptible scheme. Figure 4.8 shows the typical structure of such algorithms. After performing some initial computation, the algorithm register the first result. Then it enters a loop that repeatedly improves the result. At the end of each iteration, the algorithm updates the result. Once the algorithm terminates, and assuming it was not aborted earlier because of timeout or interrupt, it signals the fact that the computation is done.

Elementary contract algorithms have a different structure. In principle, they need to register a result just once, before the expiration of the contract. In order to guarantee that they generate a result before the expiration of the contract, these algorithms start with a call to Init-Control-Params. This procedure sets up the values of certain local parameters that determine the execution time of the algorithm. The parameters must be identified by the programmer and can limit, for example, the number of iterations, the search horizon, or the depth of graph exploration. How does a contract algorithm compute the mapping from contract time to these control parameters? The desired situation would be to use an automatic programming tool to perform this task, although no such tool is currently available.

Activation

While the programmer normally activates an elementary anytime algorithm just as if it were a standard algorithm, the model includes a special function that is actually used to control the execution time. Every activation of an elementary anytime algorithm in a program is replaced by the compiler (see Chapter 5) with a call to the following special function:

AT (*anytime-function-call*,
    activation-mode,
    time-limit,
    desired-quality)

where *anytime-function-call* is the original function to be activated (including its arguments), *activation-mode* is either CONTRACT or INTERRUPTIBLE, *time-limit* is the amount of time allocated (i.e. the con-

```
AT(anytime-function-call, activation-mode, time-limit, desired-quality)
    PID ← Create-Process (anytime-function-call, AT-PRIORITY)
    Set-Desired-Quality(PID, desired-quality)
    Initialize-Timer(RT, time-limit)
    if activation-mode = INTERRUPTIBLE then
        Enable(EXT-INTERRUPT)
    wait for event in [EXT-INTERRUPT or RT-EXPIRED or
                QUAL-ACHIEVED or TERMINATION]
        Result ← Current-Best-Result(PID)
        kill PID
        return Result
```

Figure 4.9: The control of anytime computation

tract time for a contract algorithm or a certain run-time limit for an interruptible algorithm), and *desired-quality* is the quality of results that is considered satisfactory by the consumer and, when reached, should cause the termination of the computation. Both *time-limit* and *desired-quality* are optional parameters.

Figure 4.9 shows the implementation of the AT function. It creates a process that runs the actual anytime algorithm and remains active as a control mechanism until the termination of the anytime algorithm. The termination can be signaled by any one of the following events: an external interrupt, expiration of the contract time, reaching the desired quality of results, or by a natural termination of the anytime algorithm.

To summarize, here is an example of an activation of a TSP algorithm as a contract algorithm with time limit of 1000 msec and desired quality of 0.78. The input to the algorithm is a random map of 200 cities:

```
MAP ← Create-Map(size = 200)
AT(Anytime-TSP(MAP,
    activation-mode = CONTRACT,
    time-limit = 1000,
    desired-quality = 0.78)
```

4.4 Programming techniques

Since elementary anytime algorithms serve as the basic blocks of the model of operational rationality, the success of the model largely depends on the capability to develop a large number of anytime algorithms that solve a wide range of problems. But, does anytime computation require a radical change in program design? Does it require a completely new set of programming techniques? I suggest that a large number of existing programming techniques offer a good basis for anytime computation.

To conclude this chapter, I show how several existing programming techniques can be used to develop anytime algorithms. I characterize these techniques as *indirect* programming techniques in the sense that they produce a sequence of approximate results rather than directly calculating the exact answer. Traditionally, such indirect programming techniques were motivated primarily by the fact that some problems do not have a closed form solution. In order to solve such problems, search and other *indirect* techniques must be used. The main difference between indirect computation in general and anytime computation is the emphasis of the latter on the value of producing a sequence of results, not as a means to reaching a satisfying solution, but as an end in itself. This difference in motivation means that some minor modifications might be necessary when using existing programming techniques to make it possible for intermediate results to become the final results of the computation.

Search algorithms

Search is the general problem solving technique that is based on systematic exploration of the space of possible solutions until a satisfying solution is reached. Search procedures differ in the order in which they explore the search space. As long as the search space is small, existing search procedures, such as A^*, can find an optimal solution. However, in most practical problems the search space is too large for finding optimal solutions. Thus, when using search as a basic mechanism for agent construction, the purpose of the search procedure becomes gathering information. Such information can be used in order to make a good selection of actions in a process that eventually converges on a solution. The run-time of each step can be controlled by limiting the search horizon. In principle, this forms the basis for an anytime algorithm since the more time is available for each step, the more information can be gathered before an action is selected and executed. However, monotonic increasing quality of results as a function of search effort is not guaranteed with many popular search procedures[3]. The RTA^* algorithm presented in Section 2.5 is an example of a search procedure that guarantees improved quality of results as a function of run-time when the quality is measured in terms of error bounds.

Randomized algorithms

In a wide range of applications, randomization offers an extremely important tool for the construction of algorithms [Karp, 1990]. There are two principal types of advantages that randomized algorithms often have. First, their execution time and space requirement can be smaller than that of the best deterministic algorithm known for the problem. But even more strikingly, they are simple to understand and implement. Many existing randomization techniques can be used to construct anytime algorithms whose quality of results improves in terms of degree of certainty.

A general randomized technique that has this property is sometimes called *abundance of witnesses*. It involves deciding whether the input data possesses a certain property: for example, whether an integer can be factored. Often, it is possible to establish the property by finding a certain object called a *witness*. While it may be hard to find a witness deterministically, it is often possible to show that witnesses are quite abundant in a certain probability space, and thus one can search efficiently for a witness by repeatedly sampling from the probability space. If the property holds, then a witness is very likely to be found within a few trials; thus, the failure of the algorithm to find a witness in a long series of trials gives strong evidence that the input does not have the required property. Two important types of algorithms fall under this category. A *Las Vegas algorithm* provides a solution with probability greater than 0.5 and never gives an incorrect solution. A weaker type of algorithm, known as *Monte Carlo algorithm*, can be used for situations where the algorithm makes a decision, and its output is either *yes* or *no*. A Monte Carlo algorithm is a

---

[3] The problem of decreased quality of results in spite of increased search effort has been studied by Nau [1985] and others. It is frequently referred to as *pathology* of search procedures.

randomized algorithm such that, if the answer is *yes*, the algorithm confirms it with probability larger than 0.5, but if the answer is *no*, it simply remains silent. Thus, on an input for which the answer is *no*, the algorithm will never give a definitive result. An anytime algorithm can be constructed in this case by repeating the activation of the trial phase. For example, there are several Monte Carlo algorithms for problems such as testing whether a given integer is composite, testing polynomial identities, or testing whether a graph has a perfect matching.

Another randomized programming technique that can be used to construct anytime algorithms is *fingerprinting*. This is a technique for representing a large data object by a short "fingerprint" computed for it. Under certain conditions, the fact that two objects have the same fingerprint is strong evidence that they are in fact identical. This can be used to solve pattern matching problems where the anytime algorithm uses several different fingerprints. As more and more fingerprints are generated and compared, the probability that the objects are identical increases.

Automated reasoning algorithms

Many useful algorithms for automated reasoning are based on accumulation of evidence. Such algorithms calculate the *support* of each candidate hypothesis based on observed evidence. The level of support replaces a more rigid binary truth value. This approach allows a system to accumulate evidence to support or reject a hypothesis in an incremental manner.

One approach, called *bounded conditioning*, is presented in Horvitz *et al.* [1989a]. Bounded conditioning monotonically refines the bounds on posterior probabilities in a belief network with computation and converges on final probabilities of interest. The approach allows a reasoner to exchange computational resources for incremental gains in inference quality. The algorithm solves a probabilistic inference problem in complex belief networks by breaking the problem into a set of mutually exclusive, tractable subproblems and ordering their solutions by the expected effect that each subproblem will have on the final answer.

Another approach, *variable precision logic* [Michalski and Winston, 1986], is concerned with problems of reasoning with incomplete information and resource constraints. Variable precision logic offers mechanisms for handling trade-offs between the precision of inferences and the computational efficiency of deriving them. Michalski and Winston address primarily the issue of variable certainty level and employ *censored production rules* as an underlying representational and computational mechanism. These censored production rules are created by augmenting ordinary production rules with an exception condition and are written in the form "if A then B unless C," where C is the exception condition. Systems using censored production rules are free to ignore the exception conditions when resources are tight. Given more time, the exception conditions are examined, lending credibility to high-speed answers or changing them. Some degree of quantitative analysis is added by augmenting censored rules with two parameters that indicate the certainty of the implication "if A then B." The parameter δ represents the certainty when the truth value of C is unknown, while γ is the certainty when C is known to be false.

Iterative approximation methods

The category of iterative methods includes a large number of approximation algorithms that are based on computing a series of results that get closer to the exact answer. These algorithms are obviously interruptible and in many cases the error (or quality) is directly related to the number of iterations. A classical example is Newton's method for finding the roots of an equation. Many basic iterative approximation methods can be found in standard texts in numerical analysis such as in Ralston and Rabinowitz [1978].

4.5 Theoretical aspects of approximate computation

The notion of approximate algorithms has been extensively analyzed by the theoretical computer science community. In this section I will discuss several interesting results in this field and their implications for the model of operational rationality. It should be emphasized that the following theoretical results refer to NP-complete problems only. They are motivated by the fact that NP-complete problems require super-polynomial time to solve and hence an approximation scheme may be the only alternative when the input size is large. The model of operational rationality, on the other hand, recognizes the fact that even with problems of polynomial complexity, it may be beneficial to use an anytime algorithm in order to build a more responsive real-time system. Hence, the applicability of the model goes far beyond the capability to deal with NP-completeness and intractability of computation. Nevertheless, theoretical results identify a sub-class of NP-complete problems for which a reasonable approximation scheme can produce high quality results in polynomial time.

A standard theoretical metric for the quality of an approximation algorithm is its *worst-case ratio* [Johnson, 1992]. The nearness to optimality of a given solution can be expressed as a ratio of its value to that of an optimal solution. The worst-case ratio for an approximation algorithm indicates just how far from 1 that ratio can be for a solution it generates. It is normally assumed that the numerator of the ratio is always the larger of the two solution values, so that worst-case ratios are always greater than or equal to 1 in both minimization and maximization problems. For many optimization problems, algorithms with small worst-case ratios were found, for example, a $3/2$ worst-case ratio for the Traveling Salesman problem under the triangle inequality [Christofides, 1976]. The approximation algorithm is based on solving a minimum weight perfect matching problem whose complexity is $O(n^3)$. A worst-case ratio of 2 is guaranteed by a faster approximation algorithm that is based on solving a minimum spanning tree problem whose complexity is $O(n \log n)$. For other problems, intractability results were obtained. If the triangle inequality is not assumed in the Traveling Salesman problem, for example, then guaranteeing a worst-case ratio of c for any constant c is just as hard as finding an optimal solution [Sahni and Gonzalez, 1976].

More recently, additional results regarding approximation algorithms were derived based on their connection to multiple provers [Johnson, 1992]. An interesting question to ask is what is the best possible ratio that can be guaranteed for a particular NP-complete problem in polynomial time, assuming that $P \neq NP$. There are two possible answers to this question. Either there is an *approximation threshold* $c > 1$ such that no polynomial time algorithm can guarantee a solution of quality c unless $P = NP$, or for every $c > 1$ there exists a polynomial time approximation algorithm with worst-case ratio c. In the later case, the problem is said to have a *polynomial time approximation scheme*. There are many examples of problems that have such schemes, for instance the KNAPSACK problem and the restriction of the maximum independent set problem to planar graphs.

Discussion

The relationship between these theoretical results and anytime computation is analogous to the relationship between NP-completeness results and traditional computation. Theory can tell us that some problems can be hard to approximate while others may have good polynomial approximation schemes. However, such results tell us very little about the actual capability to develop a good anytime algorithm to solve a problem. One deficiency of these theoretical results is their reliance on the worst-case ratio as a quality metric. In practice, the average-case quality with respect to a *concrete* probability distribution of input instances is much more important. The average-case quality is unfortunately much harder to derive using analytical tools and has not been much studied. Another deficiency of the theoretical analysis relates to the choice of quality measures. As defined in section 4.1, the quality measure of the results of an anytime algorithm reflects some important aspect of the output. Quality measures should directly relate to the usefulness of the results. The theoretical analysis allows a rather arbitrary measure. Therefore, a worst-case ratio of c may translate into a much better or much worse ratio when a more informative quality measure is selected. To summarize, when dealing with NP-complete problems, a theoretical analysis of approximate algorithms can save some work by indicating, for example, that a fixed worst-case ratio for a certain algorithm cannot be achieved using a polynomial time algorithm. But this kind of analysis is of very limited utility when the actual performance of an anytime algorithm needs to be characterized. It offers no alternatives to the statistical methods that I suggested earlier in constructing the performance profile of an algorithm.

Chapter 5

Compilation of Anytime Algorithms

> For the things we have to learn before we can do them, we learn by doing them.
>
> Aristotle, *Nicomachean Ethics*

I now turn from the examination of individual anytime algorithms to the problem of building large systems using anytime algorithms as components. Throughout this chapter, individual anytime algorithms will be treated as black boxes, characterized only by their performance profiles. The compilation process plays a central role in making operational rationality a modular model of anytime computation. It is the process that takes a module – composed of several elementary anytime algorithms – and makes it an optimal anytime algorithm. This process is illustrated in Figure 5.1. The input to the compiler includes a compound anytime module, that is, a module composed of several elementary anytime algorithms that does not include time allocation code and hence is not readily executable. In addition, the input includes the performance profiles of the elementary algorithms. The result of the compilation process is an executable anytime module that consists of a compiled version of the original module, a pre-defined run-time monitor, and the performance profile of the system that may include some auxiliary time allocation information. The compiled version includes some additional code to control the activation of the elementary components with an appropriate time allocation. Optimal scheduling of the elementary components may also require run-time monitoring. In fact, the complexity of the compilation task is largely determined by the choice of a run-time monitoring scheme. This relationship between compilation and monitoring is further examined in the following chapter.

I begin with an explanation of the need for compilation followed by a section categorizing the compilation problem. I then present a number of simple cases of compilation and their solutions. The notion of local compilation, that is performed on a single program fragment at a time, is introduced as a key mechanism to reduce the complexity of compilation of large programs. The complexity of the compilation of a rich compositional language is analyzed and proved to be NP-complete in the strong sense. However, local compilation, whose complexity is linear in the program size, is shown to be both efficient and optimal for a large class of programs that satisfy three basic assumptions. A number of approximate time allocation algorithms are shown to solve the compilation problem efficiently in the general case of functional composition. Finally, a number of extensions to the programming language are analyzed.

5.1 Why compilation?

Why is the compilation process so important in any model of anytime computation? The key issue addressed by the compilation process is the problem of allocating resources to the elementary components of

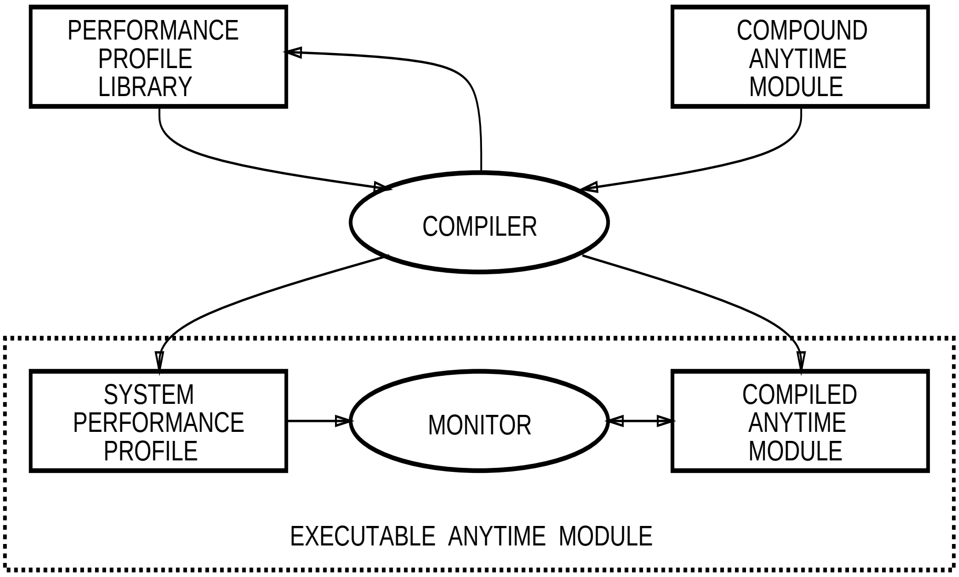

PERFORMANCE PROFILE LIBRARY
COMPOUND ANYTIME MODULE
COMPILER
SYSTEM PERFORMANCE PROFILE
MONITOR
COMPILED ANYTIME MODULE
EXECUTABLE ANYTIME MODULE

Figure 5.1: Compilation and monitoring

a module so as to optimize its behavior as an anytime algorithm. Unlike the traditional use of compilation in programming languages, the compilation of anytime algorithms is not used just as a translation mechanism. It fills in the gap created by introducing time allocation as a degree of freedom in computation. The rest of this section explains the importance of the compilation process and its design goals.

5.1.1 Modularity and anytime computation

Modularity is widely recognized as an important issue in system design and implementation. It allows the designer to decompose a large system into small, well-defined modules that can be developed and tested individually. Modularity also allows a separation between different aspects of the problem complexity, thus simplifying the task of each development group and allowing different parts of a system to be developed independently. Individual modules can be re-used in other systems to shorten the development time and reduce the costs.

The very idea of using anytime algorithms introduces a new kind of modularity into real-time system development. The modularity introduced by anytime algorithms is based on the separation between the development of the performance components and the optimization of their performance. In traditional design of real-time systems, the performance components must meet certain time constraints that are not always known at design time[1]. The result is a hand-tuning process that, hopefully, culminates with a working system. Anytime computation offers an alternative to this approach. By developing performance components that are responsive to a wide range of time allocations, one avoids the commitment to a particular performance level that might fail the system.

The main problem with modular system development is the integration of the components into one working system that meets its design goals. This integration problem is especially complex when deal-

---

[1] While the time constraints of the complete system are normally specified at design time, it is normally hard to derive from them appropriate time constraints for the components of the system.

ing with anytime computation. In standard algorithms, the expected quality of the output of each module is fixed, so composition can be implemented by a simple call-return mechanism. However, when algorithms have resource allocation as a degree of freedom, there arises the question of how to construct, for example, the optimal composition of two anytime algorithms, one of which feeds its output to the other. By solving mechanically this integration problem, the compilation process extends the principle of procedural abstraction and modularity to anytime computation.

5.1.2 Minimizing the responsibility of the programmer

Without the compilation process, the task of programming with anytime algorithms would have added a new difficulty to system development. The problem involves the activation and interruption of the components so as to optimize the performance of the complete system, or at least to make it executable. An important goal of the compilation process is to minimize the responsibility of the programmer regarding this optimization problem. Ideally, the programmer would be able to use elementary anytime algorithms as if they were standard algorithms, with all aspects of the scheduling problem solved by the compilation and monitoring components. In this respect, my model is different from existing systems for imprecise computation, such as Concord [Lin *et al.*, 1987], in the sense that the programmer does not have to determine what quality of results is desired in each situation or to schedule the components to achieve that quality.

5.2 The compilation problem

In this section I will characterize more precisely the compilation problem and its complexity. What aspects of anytime computation determine the complexity of compilation? To what extent can the compilation process be discussed in isolation? As Figure 5.1 shows, the resulting compiled anytime module may include a monitoring component. In fact, run-time monitoring is essential in many cases in order to guarantee the optimal quality of results as advertised by the compiled performance profile. However, throughout this chapter, I will examine the compilation process only, apart from the rest of the system. This separation between compilation and monitoring is only possible under certain assumptions. In this section I identify the various factors that affect the complexity of the compilation problem. Then, a certain class of compilation problems is defined that allows to defer the discussion of monitoring until the next chapter.

1. **Program structure** – The structure of a compound anytime module is a primary factor that determines the complexity of compilation and monitoring. Some programming structures, such as sequencing, are easier to handle. Other structures, such as recursive function calls, are quite difficult to compile and monitor. As with elementary anytime algorithms, the design goal is to minimize the programmer's responsibilities and duties. As a result, an effort is made to define compilation methods that depend on the semantics of the programming structure rather than on user provided information. This principle serves as a major guideline throughout this chapter.

2. **Type of performance profile** – The type of performance profile and its representation largely influence the compilation process. Highly informative performance profiles, such as the performance distribution profile, are more difficult to compile and manipulate. The complexity of the compilation is increased due to the complexity of the representation and the requirement that the resulting performance profile provides the same level of information. Simple performance profiles, such as the expected performance profile, are easier to handle but do not always have the closure property under compilation. The closure property guarantees that the performance profile provides enough

information to derive the same type of performance profile for a composed module. An expected performance profile, for example, does not have the closure property since, manipulation of a set of expected values does not yield, in general, the expected value of the result. In this respect, conditional performance profiles provide a useful, modular representation that simplifies the compilation problem.

3. **Type of anytime algorithm** – The type of algorithm used as input to the compiler and the desired type of the resulting algorithm have a direct effect on the compilation process. Contract algorithms are normally easier to construct both as elementary and as compound algorithms. Interruptible algorithms are more complicated. One can, of course, construct first a contract algorithm and then use the result of Theorem 4.1 to make it interruptible. However, with some programming structures it is advantageous to generate an interruptible algorithm directly and avoid the constant slowdown of the reduction theorem.

4. **Quality of intermediate results** – With both interruptible and contract anytime algorithms, the monitor can, in principle, examine the quality of intermediate results in order to modify the allocation of the remaining time. However, this requires a capability to determine the *actual* quality of intermediate results. The quality of intermediate results may be a simple aspect that can be quickly calculated. For example, in the case of a bin packing program whose quality function is the proportion of the container space filled with packages, the quality of an intermediate result can be easily calculated. In other cases, such as a chess playing program, the quality of a recommended move is not apparent from the move itself. Hence, the capability to determine the quality of intermediate results is an important factor in compilation and monitoring.

Depending on these factors, different types of compilation and monitoring strategies are needed. To simplify the discussion in this chapter, I will concentrate on compilation and discuss monitoring in the following chapter. However, compilation and monitoring have already been shown to be interdependent processes. In order to discuss them separately, the following distinction is made between two types of compilation processes:

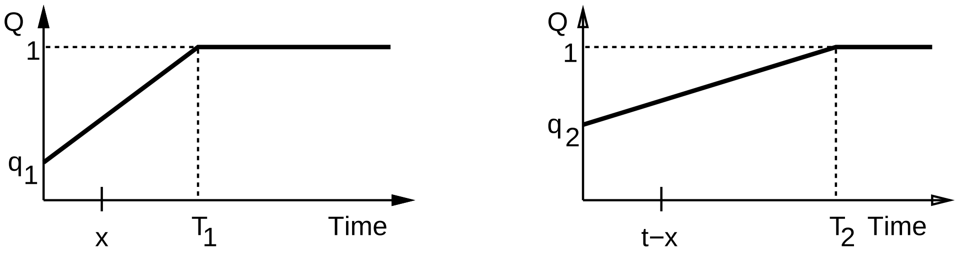
Definition 5.1 Compilation of anytime algorithms – Type I *is the process of deriving an optimal performance profile and related scheduling information for a compound anytime module assuming passive monitoring.*

Passive monitoring means that meta-level time allocation decisions are made *before* the activation of the anytime algorithms. Elementary algorithms are activated as contract algorithms only and allocation is not reconsidered before the termination of the contract or even by the termination of each subcontract. Obviously, the assumption of passive monitoring limits the capability to optimize the performance profile but it also simplifies the problem and allows us to consider compilation as an isolated issue. In the next chapter, I will expand the monitoring capability to allow active monitoring[2].

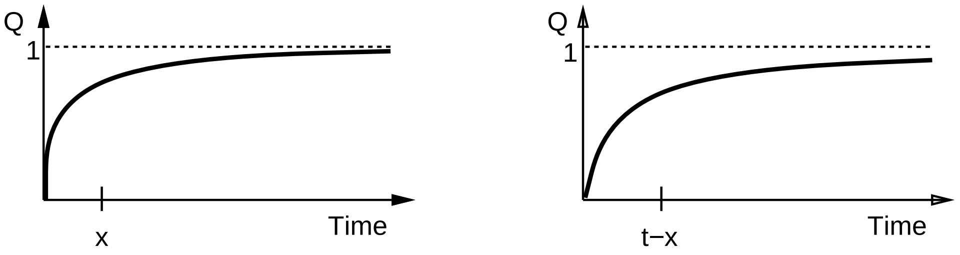
Definition 5.2 Compilation of anytime algorithms – Type II *is the process of deriving an optimal performance profile and related scheduling information for a compound anytime module assuming active monitoring.*

---

[2] Both passive and active monitoring are defined more formally in the next chapter.



(a) linear performance profiles



(b) exponential performance profiles

Figure 5.2: Performance profiles of two contract algorithms

The rest of this chapter concentrates on Type I Compilation. Throughout the chapter, the term compilation refers to Type I only. I begin with an introductory analysis of simple compilation problems. Type II compilation will be addressed in the next chapter.

5.3 Compilation examples

Before analyzing the general compilation problem, some basic examples will be examined. Starting with the composition of two algorithms, I will analyze the basic principles of compilation. Finally, an important component of this work, the notion of local compilation, will be introduced.

5.3.1 Composition of two algorithms

To begin, consider the composition of two anytime algorithms. Suppose that one algorithm takes the input and produces an intermediate result. This result is then used as input to another anytime algorithm which, in turn, produces the final result. Many systems can be implemented by a composition of a sequence of two or more algorithms. For example, an automated repair system can be composed of two algorithms: diagnosis and treatment. This can be represented in general by the following expression:

$$Output \leftarrow \mathcal{A}_2(\mathcal{A}_1(Input))$$

Suppose that the expression is composed of two contract anytime algorithms, $\mathcal{A}_1$ and $\mathcal{A}_2$, whose performance profiles are shown in Figure 5.2. The figure includes two sets of performance profiles that represent two separate cases. I start the analysis with the assumption that the performance profiles are fixed, that is, there is no uncertainty regarding the quality of the output for any given time allocation.

Case 1: Linear performance profiles

Figure 5.2(a) shows a set of two linear performance profiles. They start with an arbitrary initial quality q_i (that may be zero) and reach the maximal quality of 1 at time T_i. Hence they can be represented by:

$$Q_1(t) = q_1 + \alpha_1 t \qquad Q_2(t) = q_2 + \alpha_2 t$$

Assume that the output quality reflects the probability of producing correct results and that the success/failure of each module is independent of the success/failure of the other. Hence, the overall quality is the multiplication of the quality of $\mathcal{A}_1$ and the quality of $\mathcal{A}_2$. Since a contract algorithm is sought, the compilation process has to create the mapping:

$$\mathcal{T} : R^+ \rightarrow R^+ \times R^+ \tag{5.1}$$

and

$$\mathcal{PP} : R^+ \rightarrow [0, 1] \tag{5.2}$$

The first mapping specifies for each total allocation the amount of time that should be allocated to each algorithm so as to maximize the output quality[3]. The second mapping is the performance profile of the composed algorithm based on optimal time allocation.

For each total allocation, t, the compiler has to find the optimal allocation, x, to the first algorithm (which implies allocation $t - x$ to the second algorithm) such that the overall quality $Q(x)$ is maximal.

Theorem 5.3 *Given the performance profiles of $\mathcal{A}_1$ and $\mathcal{A}_2$, the optimal time allocation mapping is:*

$$\mathcal{T} : t \rightarrow \left(\frac{1}{2}\left(t - \frac{q_1}{\alpha_1} + \frac{q_2}{\alpha_2}\right), \frac{1}{2}\left(t + \frac{q_1}{\alpha_1} - \frac{q_2}{\alpha_2}\right)\right) \tag{5.3}$$

Proof: Since the overall output quality is:

$$Q(x) = -\alpha_1\alpha_2 x^2 + (\alpha_1\alpha_2 t - q_1\alpha_2 + q_2\alpha_1)x + q_1 q_2 + q_1\alpha_2 t \tag{5.4}$$

the maximal quality is achieved when $\frac{\partial Q}{\partial x} = 0$.
In other words:

$$-2\alpha_1\alpha_2 x + \alpha_1\alpha_2 t - q_1\alpha_2 + q_2\alpha_1 = 0 \tag{5.5}$$

The solution of this equation yields the above allocation. □

It should be noted that boundary conditions were ignored in this analysis. The following correction is therefore necessary to cover all cases:

1. If, as a result of the above mapping function, an algorithm gets more run-time than necessary for completion, the extra time should be allocated to the other algorithm (or ignored when both algorithms terminate).
2. If the time allocation to one algorithm is negative, all available time should go to the other algorithm and the allocation to that algorithm should be zero.

---

[3] Only the appropriate allocation to the first component is really necessary because the allocation to the second is simply the remaining time.

It is interesting to note also that in the special case where $q_i = 0$ (i.e. initial quality is zero), the optimal mapping allocates exactly half of the total time to each module *regardless* of α_1 and α_2.

Case 2: Exponential performance profiles

Figure 5.2(b) shows a set of two exponential performance profiles. These performance profiles are defined by:

$$Q_1(t) = 1 - e^{-\lambda_1 t} \qquad Q_2(t) = 1 - e^{-\lambda_2 t}$$

Assume that the output quality is the sum of the quality of $\mathcal{A}_1$ and the quality of $\mathcal{A}_2$. As in case 1, the compilation process has to create the optimal time allocation mapping and performance profile.

Theorem 5.4 *Given the performance profiles of $\mathcal{A}_1$ and $\mathcal{A}_2$, the optimal time allocation mapping is:*

$$\mathcal{T} : t \rightarrow \left(\frac{\ln\lambda_1 - \ln\lambda_2 + \lambda_2 t}{\lambda_1 + \lambda_2}, \frac{\ln\lambda_2 - \ln\lambda_1 + \lambda_1 t}{\lambda_1 + \lambda_2}\right) \tag{5.6}$$

Proof: Since the overall output quality is:

$$Q(x) = 1 - e^{-\lambda_1 x} + 1 - e^{-\lambda_2 (t-x)} \tag{5.7}$$

the maximal quality is achieved when $\frac{\partial Q}{\partial x} = 0$.
In other words:

$$\lambda_1 e^{-\lambda_1 x} - \lambda_2 e^{-\lambda_2 (t-x)} = 0 \tag{5.8}$$

The solution of this equation yields the above allocation. □

The other task of the compiler is to insert code in the original expression for proper activation of $\mathcal{A}_1$ and $\mathcal{A}_2$ as contract algorithms with the appropriate time allocation. This is done by replacing the simple function call by an anytime function call as explained in the previous chapter. The time allocation is determined by the total allocation and the compiled time allocation mapping. In the future, I will not always distinguish between the two mappings generated by the compilation process but will refer to both as the compiled performance profile. In practice, the two mappings are both calculated and stored together.

Summary

The compilation of a simple example – the composition of two modules – has been analyzed. This example demonstrates several general issues in compilation. To summarize these issues, two aspects of compilation that are related to the representation of performance profiles are discussed below:

1. When performance profiles are represented using a certain formula, as in the above example, the compilation problem involves solving a differential equation. The complexity of the equation, in terms of both size and number of variables, grows as a function of the number of elementary algorithms that are compiled. If a different representation is used, such as tabular discrete approximation, then the compilation problem becomes a search problem in a discrete domain whose size grows exponentially with the number of modules. The problem of exponential growth in the compilation complexity is addressed in the following section.

2. When performance profiles are represented using formulas, it is advantageous to use a *homogeneous representation* by using one family of functions for representing the performance profiles of all the

elementary algorithms. If a homogeneous representation is used, the compilation problem can be solved once for that family, thus accelerating the implementation. This raises the question of whether the chosen family is closed under compilation, that is, whether the quality of a compiled module is a function of the same family. The answer to this question depends not only on the family used, but also on the programming construct and the way quality is combined. In the example above, for instance, linear performance profiles produced a compiled non-linear (quadratic) profile. So, the family of linear functions is not closed under compilation when multiplicative quality combination is used. But the family of polynomials (of unbounded degree) is obviously closed under compilation when any polynomial is used as the quality combination function.

5.3.2 Linear composition of anytime algorithms

I now turn to an extension of the previous example where the program consists of a composition of n steps $\mathcal{A}_1, ..., \mathcal{A}_n$. Each step is an anytime contract algorithm whose performance profile is given. The goal of the compiler is to derive an optimal contract algorithm for the complete expression:

$$Output \leftarrow \mathcal{A}_n(...\mathcal{A}_2(\mathcal{A}_1(Input)))$$

Now, the time allocation problem is to find, for each total allocation of time t, the allocations: $t_1, ..., t_n$, $(t_1 + ... + t_n = t)$ that maximize the quality of the output.

If one assumes, like in the previous example, that the performance profiles are represented by a certain family of functions, one can use calculus in order to find the optimal allocation. But, instead of finding the global maxima of a quality function of one variable, as in the two module case, one must handle a quality function of $n - 1$ variables. The optimal allocation can be derived by solving a system of differential equations of the form:

$$\frac{\partial Q}{\partial t_1} = 0 \; ... \; \frac{\partial Q}{\partial t_{n-1}} = 0$$

The allocation problem becomes more complicated in the general case where a discrete tabular representation is used for performance profiles. In that case, I assume that time allocation is also discrete. The optimization problem becomes the problem of finding the best way to distribute t time units between n modules so as to maximize the overall quality function. The number of different possible time allocations to be considered is:

$$\frac{(t + n - 1)!}{t!(n - 1)!}$$

which is exponential in both n and t. It is therefore important to find ways to reduce the complexity of the global optimization problem, while preserving global optimality whenever possible. A key mechanism for achieving this goal is *local compilation*. It is presented in the following section.

Several extensions of this linear composition example can be similarly analyzed. Consider, for example, the case where there is some uncertainty regarding the actual quality of results of each algorithm, however, the performance profiles express only the *expected* quality as a function of time. In this case, the above compilation scheme remains valid only when certain conditions are met. These conditions will be examined later in this chapter. In addition, more general compilation and monitoring strategies will be described in the next chapter.

5.3.3 Local and global compilation

The compilation examples presented so far in this section demonstrate a fundamental problem of compilation, that is, the complexity of the optimization problem tends to grow exponentially with the size of the program. In order to overcome this difficulty, I propose to replace the global optimization problem with a set of local optimization problems whose complexity is polynomial or even constant. The number of simpler optimization problems grows *linearly* with the size of the program being compiled, hence the total amount of work becomes polynomial.

Definition 5.5 Local compilation *is the process of optimizing the quality of the output of each programming construct by considering only the performance profiles of its immediate sub-components.*

Local compilation solves the same problem as global compilation except for the fact that its scope is limited to one programming structure at a time. While global compilation derives the best time allocation to the elementary components, local compilation treats the immediate sub-components as if they were elementary anytime algorithms. If these components are not elementary, their performance profiles are derived using local compilation as well.

Since local compilation is much more efficient than global compilation, and since the number of times it needs to be performed is proportional to the size of the program, it offers a major reduction in the complexity of compilation in general. In fact, it makes the whole concept of compilation of anytime algorithms realistic for large programs. It also raises the question of how the resulting performance profile compares to the globally optimal performance profile.

Definition 5.6 *Local compilation is said to be optimal with respect to a particular program structure if it always achieves a globally optimal expected performance.*

Preserving global optimality is a non-trivial property of local compilation. The following theorem asserts the global optimality of local compilation with respect to the linear composition structure that was examined earlier in this section.

Theorem 5.7 Optimality of local compilation of linear composition: *Let $\mathcal{A}$ be a linear composition of the anytime algorithms: $\mathcal{A}_1, ..., \mathcal{A}_n$, such that for any time allocation $t = t_1 + ... + t_n$:*

$$Q_{\mathcal{A}}(t) = Q_1(t_1) \circ (Q_2(t_2) \circ (... \circ Q_n(t_n)))$$

where $\circ$ is an arbitrary non-decreasing binary operation, then the performance profile derived by a series of local compilations is globally optimal.

Proof: Let $Q^G_{[i,n]}(t)$ be the performance profile derived by global compilation of the algorithms $\mathcal{A}_i, ..., \mathcal{A}_n$. Similarly, let $Q^L_{[i,n]}(t)$ be the performance profile derived by local compilation. For global compilation, the performance profiles of all the elementary components are considered. For local compilation, the only two performance profiles that are considered are the performance profile of the additional algorithm, $Q_{i-1}(t)$, and the compiled performance profile of the rest, $Q^L_{[i,n]}$.
Using this notation, the theorem asserts that:

$$Q^L_{[1,n]}(t) = Q^G_{[1,n]}(t) \tag{5.9}$$

The proof is by induction on the number of algorithms. For one algorithm the claim is trivially true. For two algorithms, local compilation is identical to global compilation and hence the claim is also true. Now, assume that the claim is true for compilation of $n - 1$ algorithms and consider the compilation of n algorithms.

Let t be the total allocation and let $t_1, ..., t_n$ be the allocations to the components based on global compilation. Let $r = t_2 + ... + t_n$ be the total allocation to all the algorithms except the first, then by definition of the quality function:

$$Q^G_{[1,n]}(t) = Q_1(t_1) \circ (Q_2(t_2) \circ (... \circ Q_n(t_n))) \tag{5.10}$$

By monotonicity of $\circ$:

$$\leq Q_1(t_1) \circ Q^G_{[2,n]}(r) \tag{5.11}$$

By the induction hypothesis:

$$= Q_1(t_1) \circ Q^L_{[2,n]}(r) \tag{5.12}$$

By local compilation:

$$\leq Q^L_{[1,n]}(t) \tag{5.13}$$

By global compilation:

$$\leq Q^G_{[1,n]}(t) \tag{5.14}$$

Hence $Q^L_{[1,n]}(t) = Q^G_{[1,n]}(t)$ and the theorem is proved. $\Box$

Later in this chapter I will prove a stronger version of this theorem for tree-structured programs. However, it is important to emphasize that even when local compilation is non-optimal, it hardly has any alternative. For large programs, the global optimization problem becomes exceedingly hard to solve. This fact necessitates the use of non-optimal time allocation techniques such as local compilation.

5.4 Conditional performance profiles

Do performance profiles provide all the necessary information for the compilation process? I suggest that for some programming structures the dependency of performance on time allocation alone is insufficient. In order to be able to properly combine anytime algorithms in general, one has to take into account that the quality of the results of an algorithm depends not only on time allocation but also on input quality. In other words, by reducing the allocation of time to a certain module, one affects not only the quality of the result of that module but also the quality of the output of any module that uses that result as input. When standard programming operators are analyzed, this dependency could be automatically determined by the semantics of the operator. But when user defined algorithms are used, the dependency may only be determined empirically, based on an analysis of test cases. For example, a planning algorithm would produce a better plan (in terms of reliability, correctness, and execution efficiency) if its input, the domain description, is more accurate or more detailed. In general, many input properties other than quality can affect the quality of the results. For example, the quality of a plan is affected by the complexity of the domain, even when a perfect domain description is assumed. Nevertheless, in the development of this model, I have concentrated on the dependency on input quality since this property of the input is typically determined by the time allocation to other algorithms. Hence, it is directly controlled by the compilation process.

In this section, I introduce the notion of a conditional performance profile that is used to capture the dependency of performance on input quality. The knowledge provided by conditional performance profiles makes it possible to analyze the cumulative effect of time allocation to a certain component on the performance of the complete system.

A conditional performance profile consists of a mapping from input quality and run-time to probability distribution of output quality (or any other probabilistic characterization of the quality of the output, as in the simple performance profile case):

$$\mathcal{CPP} \,:\, Q_{in} \times T \rightarrow Pr(Q_{out}) \tag{5.15}$$

It should be noted that when an algorithm takes several inputs with varying quality, then Q_{in} is represented as a *vector* of qualities, each corresponding to one input. A simplified form of conditional performance profile may assume a single input quality measure regardless of the number of inputs. In such a case, the single quality measure can relate to a certain function of the vector of qualities, for example their geometric average. The justification of using one quality measure only, besides compact representation, is that the purpose of the compiler is to allocate time to the components so as to balance their qualities and contributions to the performance of the system. This means that under time pressure all the components "suffer" to a similar extent. Therefore, one indication of input quality may be sufficient.

5.4.1 Special cases of conditional performance profiles

By definition, a conditional performance profile is a two dimensional mapping. However, it can be represented in some cases using a projection of a simple (one dimensional) performance profile that depends on time allocation only. This representation simplifies the construction and use of conditional performance profiles. The rest of this section describes several such cases.

Homogeneous algorithms

Definition 5.8 *An anytime algorithm is said to be* **homogeneous** *if its output is represented the same way as its input.*

Homogeneous algorithms can take their output as an input. The quality of the output generated in one run becomes the input quality for a successive run. If it is assumed that the quality of the output is always higher than the quality of the input, the following property can be proved for homogeneous algorithms:

Theorem 5.9 *Let $\mathcal{A}$ be a homogeneous algorithm and let $\mathcal{PP}_{\mathcal{A}}(t)$ be the performance profile of $\mathcal{A}$ when fed with input whose quality is minimal. Then the conditional performance profile of $\mathcal{A}$ can be expressed by:*

$$\mathcal{CPP}_{\mathcal{A}}(q,t) = \mathcal{PP}_{\mathcal{A}}(\mathcal{PP}_{\mathcal{A}}^{-1}(q) + t) \tag{5.16}$$

Proof: Any problem with initial input quality q can be considered as the output of $\mathcal{A}$ generated for the same problem with minimal initial input quality[4] and time allocation $\mathcal{PP}_{\mathcal{A}}^{-1}(q)$. Note that since $\mathcal{PP}_{\mathcal{A}}(q)$ is a strictly increasing function, its inverse is well defined. Therefore, the equivalent of activating $\mathcal{A}$ with initial quality q and time allocation t is activating the same algorithm with minimal initial quality but with increased time allocation to bring it first to quality q. Hence the property holds. □

---

[4] Minimal initial input quality depends on the particular domain and on the quality measure that is used. It is normally zero.

Here are some examples of homogeneous anytime algorithms for which the theorem above can be used to capture the dependency on input quality:

Example 1: Anytime sorting
Consider an anytime sorting algorithm that is based on the standard quicksort algorithm. Initially the input is represented as one segment of unsorted elements. The anytime algorithm performs repeatedly the following step. It takes the largest unsorted segment and splits it into two segments such that all the elements in the first are smaller than all the elements in the second. Suppose that the quality of a result is $1 - (k/n)$ where k is the total number of elements in all unordered segments and n is the total number of elements to be sorted. The initial input is an unordered array. Its quality according to the above definition is zero. However, one may want to allow a situation where the input is already divided into several segments, some of which are already ordered. Clearly, a simple performance profile with the theorem above is sufficient to construct the conditional performance profile of the algorithm.

Example 2: Anytime hierarchical planning
Consider an anytime algorithm that starts with a high level abstract plan. In each iteration, the algorithm selects the worst segment of the plan and replaces it with two segments planned at a lower level of abstraction whose concatenation is a refinement of the original plan. Suppose that the quality of an abstract plan corresponds to the abstraction level of all its parts. The result is a homogeneous planning algorithm.

Example 3: Anytime iterative approximation
Consider an algorithm that computes an approximation of a certain mathematical operation by repeatedly performing a computation step that reduces the error in the result: for instance, Newton's method for finding the simple roots of real equations. This method starts with an approximation x_1 to the root of $F(x) = 0$, and calculates successively better approximations by the following formula:

$$x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)}$$

In general, x_{n+1} has more correct digits than x_n has. If one uses the number of consecutive correct digits (from the left) as the quality, one gets a homogeneous anytime algorithm.

Multiplicative performance profiles

Definition 5.10 *Let $\mathcal{A}$ be an anytime algorithm and let $\mathcal{PP}_{\mathcal{A}}(t)$ be the performance profile of $\mathcal{A}$ when fed with input whose quality is maximal. Then $\mathcal{A}$ is said to have a* **multiplicative** *conditional performance profile if its conditional performance profile can be expressed by the following product:*

$$\mathcal{CPP}_{\mathcal{A}}(q, t) = q\mathcal{PP}_{\mathcal{A}}(t) \tag{5.17}$$

Example 1: Probabilistic reasoning:
Suppose that an algorithm is fed with a hypothesis whose quality measures the probability that the hypothesis is true. The anytime algorithm computes an action based on the assumption that the hypothesis is correct. The performance profile of the algorithm expresses the value of the action as a function of time (assume that the action has no value if the hypothesis is wrong and that the value of the action as a function of time

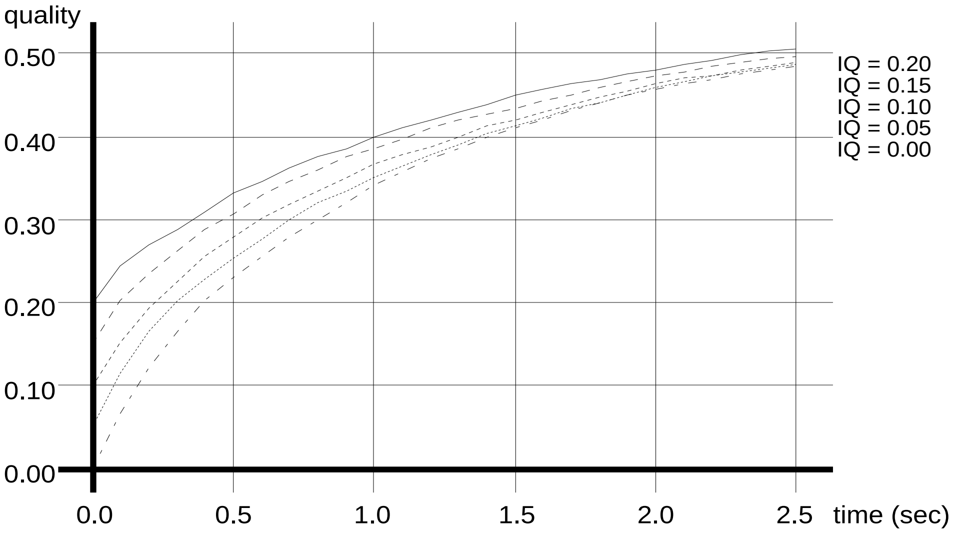


Figure 5.3: Conditional performance profile of a TSP algorithm

is independent of the hypothesis). In this case, the conditional performance profile is multiplicative since:

$$\mathcal{CPP}_{\mathcal{A}}(q,t) = qPP_{\mathcal{A}}(t) + (1-q)0 = q\mathcal{PP}_{\mathcal{A}}(t) \tag{5.18}$$

5.4.2 General conditional performance profiles

Unfortunately, most anytime algorithms are neither homogeneous nor do they have a multiplicative conditional performance profile. The dependency of output quality on input quality is rather arbitrary and cannot be analyzed by looking at the code of the algorithm. The only alternative left is to capture the dependency using statistical methods similar to the methods used in order to capture the dependency on run-time in constructing regular performance profiles.

Figure 5.3 shows the conditional performance profile of the ANYTIME-TSP algorithm that was introduced in Section 4.1.3. Each curve describes the expected quality of the result as a function of run-time for a particular initial input quality. Input with a particular desired initial quality was generated using a different TSP algorithm (“sequential tour improvement”). Hence, the example is not an instance of a homogeneous algorithm. Had I used the same algorithm to generate problems of arbitrary initial quality, the performance profile for initial quality zero could be used to represent the general conditional performance profile and it would become a typical example of a homogeneous algorithm.

5.5 Compilation of functional composition

Having defined the notion of a conditional performance profile, I can now turn to the general compilation problem of functional composition. In functional composition, each expression to be compiled is composed of an anytime function whose arguments may be either the result of functional composition or input variables. The compilation task involves finding for each total allocation t, the best way to schedule the components so as to optimize the expected quality of the result of the complete expression.

Let $\mathcal{F}$ be a set of anytime functions. Assume that all function parameters are passed by value and that functions have no side-effects (as in pure functional programming). Let $\mathcal{I}$ be a set of input variables. Then, the notion of a composite expression is defined as follows:

Definition 5.11 *A* **composite expression** *over $\mathcal{F}$ with input $\mathcal{I}$ is:*

1. *An expression $f(i_1, ..., i_n)$ where $f \in \mathcal{F}$ is a function of n arguments and $i_1, ..., i_n \in \mathcal{I}$.*
2. *An expression $f(g_1, ..., g_n)$ where $f \in \mathcal{F}$ is a function of n arguments and each g_i is a composite expression or an input variable.*

Suppose that each function $f \in \mathcal{F}$ has a conditional performance profile associated with it that specifies the quality of its output as a function of time allocation to that function and the qualities of its inputs. I will later show that the following results apply to a more general definition of composite expressions in which some of the functions are actually programming operators. The only requirement is that each operator has a "standard" conditional performance profile associated with it that describes the dependency of output quality on time allocation and input quality. But, for the simplicity of the discussion, I will first restrict it to simple composite expressions. I start this section with an analysis of the complexity of the time allocation problem.

5.5.1 Complexity results

The purpose of this section is to examine the computational complexity of compilation of composite expressions. I will show that the general problem (i.e. when evaluation of repeated sub-expressions is optimized) is NP-complete in the strong sense. This fact justifies the use of the approximate allocation techniques of the previous section. The efficiency of local compilation and its global optimality do not contradict this result since local compilation only applies to tree-structured expressions. In that case, the strong NP-completeness result does not hold but a similar transformation to another problem shows that the compilation problem remains NP-complete but pseudo-polynomial. In fact, I show that the local compilation technique is a dynamic program that solves the global optimization problem for tree-structured expressions.

Strong NP-completeness

I start with a short review of related definitions from computational complexity (for a complete discussion of these terms see [Garey and Johnson, 1979]). Given a decision problem Π, let $length(I)$ denote the number of symbols used to describe an instance I of Π under a reasonable encoding scheme, and let $max(I)$ denote the largest number in I. An algorithm that solves Π is said to be *pseudo-polynomial* if its time complexity function is bounded by a polynomial function of $length(I)$ and $max(I)$. Pseudo-polynomial algorithms are very useful since they display "exponential behavior" only when the input instances include "exponentially large" numbers. Otherwise, they may serve almost as well as polynomial time algorithms.

Given a polynomial p (over the integers), let Π_p denote the subproblem of Π obtained by restricting Π to only those instances I that satisfy $max(I) \leq p(length(I))$. If Π_p is NP-complete than Π is *NP-complete in the strong sense*. If Π is NP-complete in the strong sense, then it cannot be solved by a pseudo-polynomial time algorithm unless $P = NP$.

Compilation as a decision problem

The compilation problem is normally defined as an optimization problem, that is, a problem of finding a schedule of a set of components that would yield maximal output quality. But, in order to prove the NP-completeness results, it is more convenient to refer to the decision problem variant of the contract compilation problem. Given a composite expression e, the conditional performance profiles of its components, and a total allocation B, the decision problem is whether there exists a schedule of the components that yields output quality greater than or equal to K. To begin, consider the general problem of global compilation of composite expressions, or GCCE. The first result asserts the following property:

Theorem 5.12 *The GCCE problem is NP-complete in the strong sense.*

Proof: The GCCE problem is clearly NP since, given a particular allocation to the components, it is easy to determine in linear time the output quality of the expression. Hence, the verification problem is polynomial and the decision problem is NP. The rest of the proof is by transformation from the PARTIALLY ORDERED KNAPSACK problem, an NP-complete problem in the strong sense [Garey and Johnson, 1979] defined as follows:

INSTANCE: Finite set U, partial order $\prec$ on U, for each $u \in U$ a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$, and positive integers B and K.
QUESTION: Is there a subset $U' \subseteq U$ such that if $u \in U'$ and $u' \prec u$, then $u' \in U'$, and such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

An instance of the PARTIALLY ORDERED KNAPSACK problem will be directly transformed into a DAG representation of a composite expression. To define the construction of the DAG, the notion of a maximal element in a partially ordered set must be defined:

Definition 5.13 *An element $u \in U$ is a* **maximal element** *of U if there is no other element $u' \in U$ such that $u \prec u'$.*

The notion of a minimal element is defined in an analogous way. Every partially ordered set has at least one maximal element and at least one minimal element. Now, the construction of the DAG can be defined. For each $u \in U$ the DAG will contain a corresponding computational node. A direct arc goes from u_1 to u_2 if and only if u_1 is a maximal element of the set $\{u | u \prec u_2\}$ of all elements smaller than u_2. In addition, the DAG has a “root” node r with a directed arc from every other node $u \in U$ to r. The conditional performance profile of a node $u \in U$ is:

$$Q_u(t, q_1, ..., q_n) = \begin{cases} v(u) & \text{if } t \geq s(u) \text{ and } \forall i : q_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.19}$$

where $q_1, ..., q_n$ are the qualities of the nodes that have a directed arc to u. If there is no such node, that is, u is a minimal element of U, then its performance profile is simpler:

$$Q_u(t) = \begin{cases} v(u) & \text{if } t \geq s(u) \\ 0 & \text{otherwise} \end{cases} \tag{5.20}$$

The conditional performance profile of r then becomes the following:

$$Q_r(t, q_1, ..., q_k) = \sum_{i=1}^{k} q_i \tag{5.21}$$

The overall output quality Q_{out} is the quality of the root node r.

It is easy to see that the construction of the DAG can be accomplished in polynomial time. All that is left to show is that the answer to the PARTIALLY ORDERED KNAPSACK problem is “yes” if and only if the answer to the corresponding GCCE problem is “yes.”

If the answer to the GCCE problem is positive (with contract time B and minimal output quality K), than define U' as the set of nodes $u' \in U$ whose “output quality” in the DAG is positive. The sum of the output qualities of all the modules, except the root, must be at least K. Each module can only contribute its value to the output quality (when its allocation is at least its size). In addition, the output quality of an internal node of the DAG is “enabled” only when all its inputs have positive quality, that is, all the elements smaller than it are included. Therefore the condition that $u' \in U'$ when $u \in U'$ and $u' \prec u$ is satisfied. Finally, since the total allocation is B, $\sum_{u \in U'} s(u) \leq B$, and since the output quality is at least K, $\sum_{u \in U'} v(u) \geq K$, the answer to the PARTIALLY ORDERED KNAPSACK problem is also positive.

If the answer to the PARTIALLY ORDERED KNAPSACK problem is positive (with knapsack size B and minimal value K), then simply allocate to each computational node $u' \in U'$ an amount of time equal to its size. The definition of the PARTIALLY ORDERED KNAPSACK problem and the transformation to the DAG guarantee that the output quality of each u' would be equal to its value $s(u')$. Hence a minimal output quality of K is guaranteed and the answer to the GCCE problem is also positive.

Now, since the PARTIALLY ORDERED KNAPSACK problem is NP-complete in the strong sense, and since the above transformation is polynomial, the GCCE problem is NP-complete in the strong sense. □

Note that the strong NP-complete result implies that the general compilation problem is not pseudo-polynomial. Hence the approximate time allocation algorithms that I will present later in this section are necessary to solve the general compilation problem.

I now turn to the analysis of the tree-structured case of the compilation problem, referred to as tree-structured GCCE. In this case, the graph representation of a composite expression is restricted to a directed tree. This restriction does not allow any repetition of sub-expressions since any such repetition changes the representation from a directed tree to a directed acyclic graph. I will show that the tree-structured GCCE is NP-complete.

Theorem 5.14 *The tree-structured GCCE problem is NP-complete.*

Proof: The tree-structured GCCE problem is clearly NP since, given a particular allocation to the components, it is easy to determine in linear time the output quality of the expression. The verification problem is polynomial and hence the problem is NP. The rest of the NP-completeness proof is by transformation from the KNAPSACK problem [Garey and Johnson, 1979; Karp, 1972], defined as follows:

INSTANCE: Finite set U, for each $u \in U$ a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$, and positive integers B and K.
QUESTION: Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

An instance of the KNAPSACK problem will be transformed into a tree-structured GCCE problem by constructing a binary tree whose leaves are the elements of U. Each element $u \in U$ corresponds to one leaf of the tree (one can add leaf nodes of zero size and value to make the number of leaves an exact power of 2). The performance profile of each leaf node is:

$$Q_u(t) = \begin{cases} v(u) & \text{if } t \geq s(u) \\ 0 & \text{otherwise} \end{cases} \tag{5.22}$$

Now, $O(|U|)$ internal nodes are added to construct a complete binary tree. The conditional performance profile of each internal node w is the sum of the qualities of its left and right branches:

$$Q_w(t, q_1, q_2) = q_1 + q_2 \tag{5.23}$$

Note that internal nodes of the tree do not consume any computation time. The output quality Q_{out} is the quality of the root node which is actually the sum of values of all the elements of U whose allocation exceeds their size.

It is easy to see that the construction of the tree can be accomplished in polynomial time. To complete the proof, one needs to show that the answer to the KNAPSACK problem is "yes" if and only if the answer to the corresponding tree-structured GCCE problem is "yes." This is trivially true when one sets the contract time to B and the minimal output quality to K. The exact proof is very similar to the previous one. Hence the tree-structured GCCE problem is NP-complete. □

The KNAPSACK problem itself is pseudo-polynomial. In fact, the problem can be solved by a simple dynamic programming algorithm. This raises the question of whether the compilation problem of tree-structured expressions is also pseudo-polynomial. The answer to this question is positive under the following two assumptions:

1. The bounded degree assumption:
   The tree has a bounded degree. In other words, the number of inputs to each function is bounded.

2. Input monotonicity assumption:
   Each conditional performance profile is a monotonic non-decreasing function of input quality. In other words, if $Q(t, q)$ is a conditional performance profile, then for any t and $q \geq > q'$, $Q(t, q) \geq Q(t, q')$.

These assumptions lead to the following result:

Theorem 5.15 *The tree-structured GCCE problem is pseudo-polynomial under the bounded degree and the input monotonicity assumptions.*

Proof: The problem remains NP-complete since the bounded degree and the input monotonicity assumptions were not violated by the transformation of the NP-completeness proof. The rest of the proof is based on the introduction of an efficient local compilation algorithm that solves the time allocation problem for this case in polynomial time. The algorithm and the proof of its optimality are presented in the following section. □

5.5.2 Optimality of local compilation

My goal in this section is to prove the optimality of local compilation of tree-structured functional programs under the bounded degree and the input monotonicity assumptions. Without limiting the generality of the discussion, I will consider binary functions only and assume that the composite expression is a complete binary tree. The leaves of the tree are functions that take input variables as inputs and the internal nodes are functions that take composite expressions as inputs.

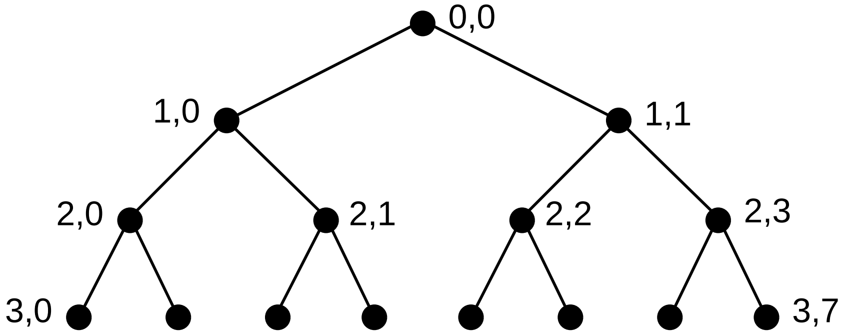


Figure 5.4: Tree representation of a composite expression

Let $f_{i,j}$ denote the j^{th} function on the i^{th} level of the tree. The root node is denoted accordingly by $f_{0,0}$. If the tree is of depth n, then the nodes corresponding to $f_{n,0}, ..., f_{n,2^n-1}$ are leaf nodes whose inputs are input variables. For any other node $f_{i,j},\ 0 \leq i \leq n-1,\ 0 \leq j \leq 2^i - 1$, the inputs are: $f_{i+1,2j}$ and $f_{i+1,2j+1}$ as shown in Figure 5.4.

Corresponding to each node of the binary tree is a conditional performance profile $Q_{i,j}(q_1, q_2, t)$ which characterizes the output quality for that node as a function of its input qualities, q_1 and q_2, and time allocation t.

Given a composite expression e of depth n, and a particular input quality, the global compilation problem is to find the optimal time allocation to all the nodes of the tree that would maximize the quality of the output of the root node:

$$Q_e^G(t) = arg \max_{t_{i,j}} Q_{0,0}(.), \quad \sum_{0 \leq i \leq n} \sum_{0 \leq j \leq 2^i - 1} t_{i,j} = t \tag{5.24}$$

where $Q_{0,0}(.)$ denotes the result of replacing (in the expression e) every function by its conditional performance profile and every input variable by its quality.

What is the complexity of global compilation? The previous complexity results imply that the problem is NP-complete. But no particular algorithm for solving the problem has been considered yet. To define such an algorithm, assume a discrete tabular representation of performance profiles. This representation reduces the compilation problem to the problem of calculating all the entries of a particular table. The indices of the table range over time allocation and input quality[5]. The size of the table is a system parameter that controls the error in quality calculation. The complexity of global compilation is therefore determined by the amount of work needed to compute each entry of the table, that is, the complexity of solving Equation 5.24 above. If τ is the number of discrete time units to be allocated and if the size of the expression (i.e. the number of nodes in the tree) is κ, then the number of different possible time allocations to be considered is:

$$\frac{(\tau + \kappa - 1)!}{\tau!(\kappa - 1)!}$$

For each allocation the output quality has to be calculated in time $O(\kappa)$. The overall work for each entry of the table is therefore exponential in both τ and κ. Hence a naive approach to global compilation is not

---

[5] Input quality here refers to the system input. In most problems it corresponds to a single quality measure although in principle it may be a vector of values each corresponding to a single input variable.

realistic for large composite expressions.

Given a particular composite expression e of depth n, a local compilation scheme for e is defined by induction on its structure. For a leaf node, the locally compiled performance profile is the conditional performance profile associated with that node:

$$Q^L_{n,j}(t) = Q_{n,j}(q_{n,j,1}, q_{n,j,2}, t), \quad 0 \leq j \leq 2^n - 1 \tag{5.25}$$

where $q_{n,j,1}$ and $q_{n,j,2}$ are the qualities of the two inputs of the particular function. For each internal node, the locally compiled performance profile is defined using the performance profiles of its immediate inputs:

$$Q^L_{i,j}(t) = arg \max_{t_1,t_2} \{Q_{i,j}(Q^L_{i+1,2j}(t_1), Q^L_{i+1,2j+1}(t_2), t - t_1 - t_2)\} \tag{5.26}$$

Finally, the performance profile of e (as a result of local compilation) is denoted by the following expression:

$$Q^L_e(t) = Q^L_{0,0}(t) \tag{5.27}$$

Note that the external input quality was deliberately omitted in this notation since the focus is on the result of local compilation for a *given* input quality.

What is the complexity of local compilation? Using the discrete representation described above, local compilation requires $O(\tau^2)$ work per (internal) node of the tree. Hence the total amount of work is $O(\kappa\tau^2)$ (for each *given* input quality). In terms of space requirements, even though local compilation requires $O(\kappa)$ separate performance profiles (one for each internal node of the tree), its total space requirement is only a constant factor more than the space requirement of global compilation. The reason is that a compiled global performance profile needs to specify the allocation to *each* node of the tree (i.e. κ elements) for each total allocation while a compiled local performance profile needs to specify only the allocation to the immediate successors of each node and to the node itself (i.e. three elements). To summarize, local compilation has the same space requirements as global compilation but it reduces the time complexity of the optimization problem from exponential to polynomial. Moreover, the complexity is linear in the size of the program.

How does the quality produced by local compilation compares with the quality of global compilation? The following theorem guarantees that the final result of both compilation schemes is the same:

Theorem 5.16 Optimality of local compilation of composite expressions: *Let e be a composite expression of an arbitrary depth n whose conditional performance profiles satisfy the input monotonicity assumption, then for any input and total time allocation t:*

$$Q^L_e(t) = Q^G_e(t)$$

Proof: By induction on the depth of the tree. For trees of depth 1 the claim is trivially true because both compilation schemes solve the same optimization problem. Suppose that the claim is true for trees of depth $n - 1$ or less. Let e be an expression of depth n, and let $t_{i,j}$ be the allocations to $f_{i,j}$ based on global compilation and resulting in a global optimum. Let t_l and t_r be respectively the total allocation to the left and right subtrees of the root node:

$$t_l = \sum_{i=1}^{n} \sum_{j=0}^{2^{i-1}-1} t_{i,j} \tag{5.28}$$

$$t_r = \sum_{i=1}^{n} \sum_{j=2^i-1}^{2^i-1} t_{i,j} \tag{5.29}$$

$$t = t_l + t_r + t_{0,0} \tag{5.30}$$

Then:

$$Q_e^G(t) =$$

By definition and monotonicity:

$$= Q_{0,0}(Q_{1,0}^G(t_l), Q_{1,1}^G(t_r), t_{0,0}) \tag{5.31}$$

By the induction hypothesis:

$$= Q_{0,0}(Q_{1,0}^L(t_l), Q_{1,1}^L(t_r), t_{0,0}) \tag{5.32}$$

By local compilation:

$$\leq Q_{0,0}^L(t) \tag{5.33}$$

By definition:

$$= Q_e^L(t) \tag{5.34}$$

By global compilation:

$$\leq Q_e^G(t) \tag{5.35}$$

Hence $Q_e^L(t) = Q_e^G(t)$ □

Note that the input monotonicity assumption is required to guarantee the optimality of local compilation. The bounded degree assumption, on the other hand, is only used to guarantee that the complexity of local compilation of each internal node is polynomial in t, and hence shows that the problem is pseudo-polynomial. Both the bounded degree and the input monotonicity of performance profiles are not only reasonable assumptions but also desirable from a methodological point of view. The bounded degree assumption limits the number of inputs to each algorithm by a certain constant, a principle that has been long recognized as a good programming practice in the development of modular systems. The input monotonicity of performance profiles is a desirable property in general. It supports the selection of performance metrics that correlate with the intuitive notion of quality rather than being random features of the results.

5.5.3 Additional programming operators

The optimality of local compilation of composite expressions makes it an attractive programming construct. But, can the result be extended to include additional programming operators? To begin, I examine the possibility of replacing some of the functions in a composite expression by standard programming operators. The validity of the theorem will be preserved if each new operator is defined as a regular function in a composite expression. In other words, each language operator, ϕ, must produce a result whose quality depends on the qualities of its inputs and time allocation to the evaluation of the operator itself, ϵ_ϕ. Several useful operators have this property. Their evaluation time is normally a small constant time that can be ignored in some applications. Their conditional performance profiles are normally represented as step functions. Here are some examples:

1. The operator **one-of** is defined as follows: its output is the result of its single component with the highest quality and its quality is the quality of that component. The conditional performance profile

of **one-of** is:

$$Q_{\mathbf{oneof}}(q_1, ..., q_n, t) = \begin{cases} max(q_1, ..., q_n) & \text{if } t > \epsilon_{\mathbf{oneof}} \\ 0 & \text{otherwise} \end{cases} \tag{5.36}$$

This models a situation in which several alternative methods can be used to solve the same problem. For example, suppose that one needs to transport n identical packages using a certain container. The components of **one-of** might be several alternative bin packing algorithms. The quality of each algorithm is measured by the percentage of packages that can be packed in the given container as a function of computation time and the total volume of the packages. Obviously, the total number of packages that can be transported is proportional to the maximal quality among all the individual bin packing algorithms.

2. The operator **all** is defined as follows: its output is the result of its single component with the lowest quality and its quality is the quality of that component. The conditional performance profile of **all** is:

$$Q_{\mathbf{all}}(q_1, ..., q_n, t) = \begin{cases} min(q_1, ..., q_n) & \text{if } t > \epsilon_{\mathbf{all}} \\ 0 & \text{otherwise} \end{cases} \tag{5.37}$$

This models a situation in which several sub-problems *must* be solved and *all* the solutions are essential in order to solve the original problem. Moreover, the quality of the worst solution imposes an upper bound on the total performance. For example, suppose that one needs to transport n identical packages using, sequentially, k different containers. The components of **all** might be k bin packing algorithms, one for each container type. The quality of each component is measured as in the previous example by the percentage of the number of packages that can be packed in the corresponding container. Obviously, the total number of packages that can be transported (in a single shipment) is proportional to the minimal quality among all the bin packing algorithms.

3. The operator **dac** (divide and conquer) is defined as follows: its output is the result of RCF applied to the results of its components and its quality is the result of QCF applied to the qualities of the components. The conditional performance profile of **dac** is:

$$Q_{\mathbf{dac}}(D, RCF, QCF, q_1, ..., q_n, t) = \begin{cases} QCF(q_1, ..., q_n) & \text{if } t > \epsilon_{\mathbf{dac}} \\ 0 & \text{otherwise} \end{cases} \tag{5.38}$$

This models a situation in which a problem is solved by dividing the input problem into several simpler sub-problems, solving each sub-problem, and generating the output from the results of the sub-problems. The function D is used to divide the input problem into sub-problems; the function RCF (Result Combination Function) is used to determine the overall result based on the results of the components; and the function QCF (Quality Combination Function) is used to determine the overall quality based on the qualities of the solutions to the sub-problems. $\epsilon_{\mathbf{dac}}$ is (normally) a constant time necessary in order to generate the sub-problems and in order to combine their results. For example, a path planning algorithm that has to find a path between P_1 and P_2 in an environment with random obstacles can determine a third position P_3 between the start and goal position and find paths from P_1 to P_3 and from P_3 to P_2. Assuming that the quality of each component is l_{opt}/l (i.e. the length of the optimal path divided by the length of the calculated path), then RCF is a simple concatenation

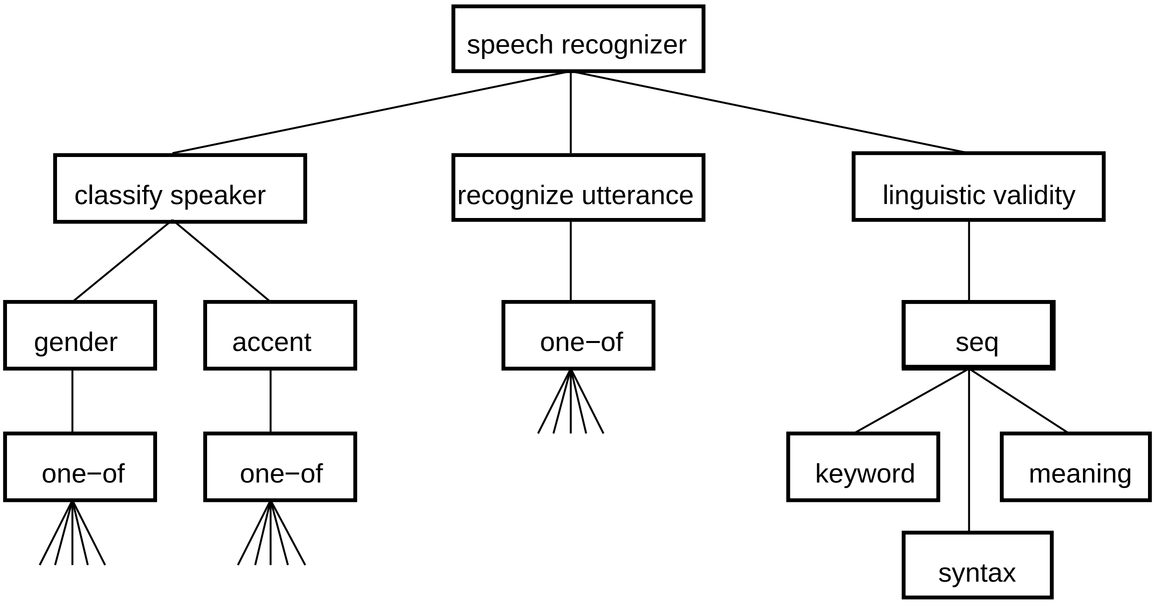


Figure 5.5: Anytime composite module for speech recognition

operation and:

$$QFC(q_1, q_2) = k \frac{2q_1q_2}{q_1 + q_2} \tag{5.39}$$

where q_1 and q_2 are the qualities of the components and $0 < k < 1$ is a factor that determines the effect of the choice of P_3 on the quality of the complete path.

To summarize, the optimality of local compilation provides a powerful tool to compile large composite expressions that may include a variety of standard programming operators in addition to user-defined elementary anytime algorithms. Figure 5.5 shows an example of such a program. It is an anytime module for speech recognition whose elementary components are anytime algorithms. The program has three main components: a module that classifies the speaker, a module that generates possible symbolic representations of the utterance, and a module that checks the linguistic validity of these representations. Each of the one-of operators represents a set of alternative methods of implementation for a particular function, for example, a neural network implementation versus a knowledge base implementation.

5.5.4 Repeated sub-expressions

The analysis of functional composition so far has not taken into account the possibility that a composite expression may have a sub-expression that appears several times. Such a sub-expression is called a *repeated sub-expression*. Using the tree representation, a repeated sub-expression corresponds to a sub-tree that appears several times. The tree-structured analysis is based on the assumption that all the nodes of the tree are evaluated while, with repeated sub-expressions, one should allocate time only once in order to evaluate all the copies of a repeated sub-expression[6]. For example, consider the following composite expression:

---

[6] Every sub-expression of a repeated sub-expression is obviously a repeated sub-expression as well. To remove any ambiguity, I will use the term "repeated sub-expression" only with respect to *maximal repeated sub-expression*, that is, repeated sub-expressions

```
F($x$)
    $a \leftarrow$ A($x$)
    $b \leftarrow$ B($a$)
    $c \leftarrow$ C($a$)
    $d \leftarrow$ D($b, c$)
    $e \leftarrow$ E(d)
    return $e$
```

(a) Definition of F using straight line code

```
F($x$)
    return E(D(B(A($x$)),C(A($x$))))
```

(b) Definition of F using functional composition

Figure 5.6: Composite expressions and straight line code

$$\text{E(D(B(A(}x\text{)),C(A(}x\text{))))}$$

The sub-expression A(x) appears twice and an efficient compiler should not allocate time to both copies.

Functional composition with repeated sub-expressions can be represented efficiently as a *straight line program* with anytime algorithms as basic operations. A straight line program is a sequence of expressions of the following form:

$$u \leftarrow F(v_1, ..., v_n)$$

where u is a new local variable and each v_i is either an input variable or an existing local variable. The last local variable defined by the sequence is considered to be the *result* of the sequence. There is a one-to-one mapping between composite expressions and straight line code. Given a composite expression e, the corresponding straight line code is defined by induction on the structure of e:

1. An input variable corresponds to an empty program.
2. An expression of the form $F(g_1, ..., g_n)$, where each g_i is a composite expression or an input variable, corresponds to the program that is composed of the concatenation of the programs corresponding to $g_1, ..., g_n$ followed by

   $$u \leftarrow F(r_1, ..., r_n)$$

   where u is a new local variable and r_j is the result of the program corresponding to g_j, or g_j itself if it is an input variable.

The inverse transformation is straightforward. A straight line program records all the intermediate results and hence can reuse the result of a sub-expression when it appears several times.

For example, Figure 5.6 shows a definition of a function, F, both as a straight line program, (a), and as a composite expression, (b). Every straight line program has a corresponding Directed Acyclic Graph

---

that are not proper sub-expressions of a larger repeated sub-expression.

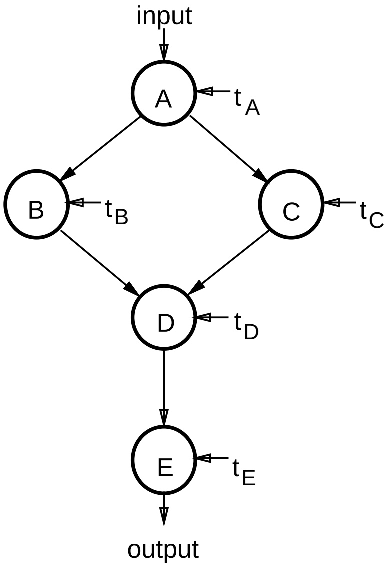


Figure 5.7: DAG representation of composite expressions

(DAG) representation where each node corresponds to one assignment of a value to a new local variable. A directed arc goes from node v_i to node v_j if the assignment expression of v_j uses the local variable v_i as an argument. Figure 5.7 shows the DAG representation of F.

As in the tree-structured case, the purpose of the compilation is to compute a time allocation mapping that would specify for each input quality and total allocation of time the best apportionment of time to the components so as to maximize the expected quality of the output. However, the DAG representation makes it hard to apply local compilation. The problem arises since local compilation is only possible when one can repeatedly break a program into sub-programs whose execution intervals are disjoint, so that allocating a certain amount of time to one sub-program does not affect in any way the evaluation and quality of the other sub-programs. While this claim is true about tree-structured programs, it is not a property of DAGs. Consider, for example, the expression represented by Figure 5.7. Although B and C are the ancestors of D, their time allocations cannot be considered independently since they both use the same sub-expression, A(x). This problem is addressed by the following section.

5.5.5 Compilation of unrestricted composite expressions

An efficient representation of a composite expression corresponds to a DAG rather than to a tree. Unfortunately, the DAG representation imposes a severe restriction on the capability to apply local compilation. In this section I will present three time allocation methods that solve this problem. The first method is based on an efficient algorithm that finds a solution to the global compilation problem directly, but does not guarantee global optimality. The second method is based on determining first the allocation to the repeated sub-expressions and then using standard local compilation to determine the allocation to the other components. The third method is based on learning the allocation to the repeated sub-expression based on repeated application of standard local compilation. Finally, the complexity and optimality of the three methods will be contrasted.

```
for each $Q_{in} \in$ QUALS-TABLE do
    for each $T \in$ TIME-TABLE do
        $s \leftarrow$ INITIAL-RESOLUTION($T$)
        $t_i \leftarrow T/n \quad \forall i : 1 \leq i \leq n$
        repeat
            while $\exists i, j$ such that
                    $E(Q_{out}(Q_{in}, t_1, ..., t_i - s, ..., t_j + s, ..., t_n)) >$
                    $E(Q_{out}(Q_{in}, t_1, ..., t_n))$
                let $i, j$ be the ones that maximize $E(CPP_{\mathcal{M}})$
                $t_i \leftarrow t_i - s$
                $t_j \leftarrow t_j + s$
            $s \leftarrow s/2$
        until $s < \epsilon$
        $\mathcal{T}[Q_{in}, T] \leftarrow (t_1, ..., t_n)$
```

Figure 5.8: Time allocation using a hill-climbing search

Method 1: Time allocation using a hill-climbing search

While local compilation cannot be applied to DAGs directly, global compilation works exactly the same way as it works with trees. For each particular sequence of time allocations to all the components of a DAG, the quality of the output can be computed using the conditional performance profiles of the components. This can be done in linear time in the size of the graph. However, the number of possible allocations to the components grows exponentially. This difficulty can be removed by limiting the search space.

Consider again the definition of the function F. Given a total allocation t, the compiler has to determine the suballocations t_A, t_B, t_C, t_D and t_E ($t_A + t_B + t_C + t_D + t_E = t$) to the modules A, B, C, D and E respectively that maximize the expected quality of the output. For each given allocation of time, the expected quality of the output can be calculated based on the DAG representation and the conditional performance profiles of the elementary anytime functions. In order to find an optimal allocation, I have implemented the following search algorithm.

The time allocation algorithm shown in Figure 5.8 is based on a hill-climbing search. It starts with an equal amount of time allocated to each component of the DAG. Then it considers trading s time units between two modules so as to increase the expected quality of the output. As long as it can improve the expected quality, it trades s time units between the two modules that have maximal effect on output quality. When no such improvement is possible with the current value of s, it divides s by 2 until s reaches a certain minimal value, ϵ. At that point, it reaches a local maximum and returns the best time allocation it found. As with any hill-climbing algorithm, it suffers from the problem of converging on a local maximum. An analysis of the algorithm shows that simple properties of the conditional performance profiles of the components, such as monotonicity, are not sufficient to guarantee global optimality.

```
for each $Q_{in} \in$ QUALS-TABLE do
    for each $T \in$ TIME-TABLE do
        $Q_{max} \leftarrow 0$
        $r_{max} \leftarrow 0$
        for $r \leftarrow 0$ to $T$ step $\epsilon$
            $t \leftarrow T - r$
            ADJUST-PP($r$)
            APPLY-LOCAL-COMPILATION($e, t$)
            $Q \leftarrow Q_{out}(Q_{in}, (r|t_1, ..., t_n))$
            if $Q > Q_{max}$ then do
                $Q_{max} \leftarrow Q$
                $A_{opt} \leftarrow (r|t_1, ..., t_n)$
        $\mathcal{T}[Q_{in}, T] \leftarrow A_{opt}$
```

Figure 5.9: Time allocation with pre-determined time to repeated sub-expressions

Complexity

As in the earlier analysis of local and global compilation, I assume a discrete tabular representation and look at the complexity of computing each entry in the table representing the compiled performance profile. Again, let the total number of modules be κ, and let the maximal number of discrete time units to be allocated be $\tau = T_{max}/\epsilon$. The complexity of the algorithm is then:

$$O(\kappa^3 log\tau)$$

This is due to the fact that for each search resolution s, the algorithm needs to find the optimal pair of modules for trading time. This is done in $O(\kappa^2)$ by considering every possible pair. This step repeats only a constant number of times. Finding the expected quality of the output is performed in $O(\kappa)$ and the number of time resolution measures is $O(log\tau)$. Hence we get the above overall complexity.

Method 2: Pre-determined allocation to repeated sub-expressions

The second method is based on fixing the allocation to each repeated sub-expression before computing the allocation to the other components. The allocation to the other components is determined based on standard local compilation. Time allocation is made only once to all the copies of each repeated sub-expression. Once that allocation is decided, the complete expression is treated as a tree rather than a DAG and the efficient local compilation scheme is used.

Let e be a composite expression. To begin, assume that e has only one repeated sub-expression e' that appears $m > 1$ times in e. The copies of e' are denoted by $e'_1, ..., e'_m$. Figure 5.9 shows the time allocation algorithm. Its central idea is to *reserve* a certain amount of time r, out of the total allocation t, for evaluating a single copy of the repeated sub-expression e'. All the other copies may "enjoy for free" the result of this evaluation. The fact that r time units are reserved for e' is communicated to the local compilation process by adjusting the performance profile of e'. The new performance profile is a step function that returns quality $Q_{e'}(r)$ at zero time and provides no further improvement of quality. Since no improvement of quality is possible, an optimal schedule would not allocate any time to any of the copies and hence standard local compilation is guaranteed to allocate the remaining time optimally to the *other* components. The algorithm performs a search to find the best pre-determined reserved time r for which the output quality is maximal.

I now show that if the conditional performance profiles of all the components are monotonic, then any optimal schedule has the following property:

Lemma 5.17 *Any optimal schedule for the evaluation of e allocates time to a single copy of e'.*

Proof: Suppose that there is an optimal schedule in which more than one copy of e' is evaluated. Let $r_1, ..., r_m$ be the allocations to the m copies, and let $r = \sum r_i$. By the monotonicity of the performance profile of e', the quality achieved by allocating r time units to a single copy is greater than any of the qualities achieved with allocations $r_1, ..., r_m$. Hence, by substituting the result of that single copy for all the copies without changing the allocation to the other components, and by the monotonicity of the conditional performance profiles, it is apparent that the output quality would increase. This contradicts the optimality of the original schedule. Therefore, time must be allocated to a single copy only. □

Having established the fact that any optimal schedule must activate e' only once with some allocation r, I define a two phase optimization problem. The first phase determines the optimal r and the second finds the optimal allocation to the other components. The optimality of method 2 can then be established:

Theorem 5.18 Optimality of Method 2: *Let e be a composite expression with a single repeated sub-expression e', then method 2 returns a globally optimal time allocation schedule for evaluating e.*

Proof: By Lemma 5.17, any global schedule allocates r time units to a single copy of e'. Since the algorithm performs search over the complete range of r, and since local compilation yields optimal results for trees, Method 2 is guaranteed to find the globally optimal schedule. □

Complexity

Assuming the same discrete representation as in method 1, I now look at the complexity of computing each entry in the table representing the compiled performance profile. Again, let the total number of modules be κ and let the maximal number of discrete time units to be allocated be $\tau = T_{max}/\epsilon$. The complexity of the algorithm is then:

$$O(\kappa\tau^3)$$

This is due to the fact that the complexity of the search for the optimal value of r is $O(\tau)$ and the most complicated step inside the loop is the local compilation step with complexity $O(\kappa\tau^2)$.

To extend this method to work with p different repeated sub-expressions, the algorithm must consider any possible pre-determined allocation to (single copies of) the repeated sub-expressions. The complexity of this step becomes τ^p when $p \ll \tau$. And, the overall complexity becomes

$$O(\kappa\tau^{(p+2)})$$

Method 3: Learning the allocation to repeated sub-expressions

The third method is based on learning the allocation to repeated sub-expressions through standard local compilation. To be able to apply local compilation, the algorithm first ignores the repetition of sub-expressions

```
for each $Q_{in} \in$ QUALS-TABLE do
    for each $T \in$ TIME-TABLE do
        $r \leftarrow 0$
        repeat
            $t \leftarrow T - r$
            SHIFT-PP($r$)
            APPLY-LOCAL-COMPILATION($e, t$)
            Let $r_1, ..., r_m$ be the allocations to $e_1, ..., e_m$
            $r \leftarrow r + \max\{r_i\}$
        until $\sum r_i = 0$
        $\mathcal{T}[Q_{in}, T] \leftarrow (r | t_1, ..., t_n)$
```

Figure 5.10: Learning the allocation to repeated sub-expressions

and uses standard local compilation. Then, it applies a series of performance profile adjustments that converge on a single allocation to each repeated sub-expression.

Again, let e be a composite expression. As with method 2, I consider first the case where e has only one repeated sub-expression e' with copies $e'_1, ..., e'_m$. Figure 5.10 shows the time allocation algorithm. It learns the allocation r to a single copy of e'. Starting with $r = 0$, the algorithm repeatedly increases r until local compilation allocates no additional time to the copies of e'. In each iteration, the current value of r is used to determine how much time to reserve for evaluating e'. The fact that r time units are reserved for e' is communicated to the local compilation process by adjusting the performance profile of e'. The time origin of the performance profile is shifted r units to the right. Standard local compilation is then applied and the optimal allocation to *all* the components is computed. Suppose that, based on the adjusted performance profile, the allocations to the m copies of e' are $r_1, ..., r_m$. Then, the maximal allocation among those is used to increase the value of r. This process is repeated until no additional time is allocated to any of the copies beyond the reserved time allocation r.

At first look, Method 3 may seem to converge on an optimal schedule, however the following theorem shows that it may not.

Theorem 5.19 *Let e be a composite expression with a single repeated sub-expression e', then Method 3 does not necessarily returns a globally optimal time allocation schedule for evaluating e.*

Proof: By example. Let e be the following expression:

$$e = B(A(x), A(x))$$

where the unconditional performance profile of A is the step function:

$$q_A(t) = \begin{cases} 0.0 & \text{if } 0 \le t < 1 \\ 0.5 & \text{if } 1 \le t < 2 \\ 1.0 & \text{if } 2 \le t \end{cases} \tag{5.40}$$

and the conditional performance profile of B is the step function:

$$q_B(q_1, q_2, t) = \begin{cases} 0.0 & \text{if } (t < 1) \;\vee\; (q_1 < 0.5) \;\vee\; (q_2 < 0.5) \\ 0.5 & \text{if } (1 \leq t < 2) \;\wedge\; (0.5 \leq q_1, q_2 < 1) \\ 0.8 & \text{if } (2 \leq t) \;\wedge\; (0.5 \leq q_1, q_2 < 1) \\ 1.0 & \text{if } (1 \leq t) \;\wedge\; (q_1 = 1) \;\wedge\; (q_2 = 1) \end{cases} \tag{5.41}$$

For a total allocation of $t = 3$, if A gets 2 time units and B gets 1 unit, then the output quality is 1.0. Since 1.0 is the maximum output quality, it is obvious that the above schedule is optimal. How would the above time allocation algorithm behave in this case? The algorithm will first allocate 1 unit to each module resulting in output quality of 0.5. In the next iteration it will reserve $r = 1$ time unit for A and allocate optimally the 2 remaining time units. Allocating the complete remaining time to the two copies of A would leave no time for B and result in an output quality of zero. At the same time, allocating less than one unit to either copy would yield no improvement in the output quality. Therefore, the 2 remaining time units will be allocated to B, resulting in an improved quality of 0.8. Since the allocation to both copies of A is zero, the algorithm terminates at this point with a sub-optimal schedule. □

Note that this example has some implications for the capability to improve Method 2. It may seem natural to try to determine the optimal r in Method 2 using a more guided search, such as a binary search. But this example shows that it may be hard to determine whether r is too small or too big. In particular, the fact that additional time is allocated by local compilation to the components, beyond the reserved time, shows that r is too small. But, if no additional time is allocated to the components, r may be either too big or too small.

What are the conditions that would guarantee the optimality of Method 3 and would allow the use of binary search in Method 2? This is left as an open question at this point. A good direction toward establishing such conditions is to investigate the situation in which conditional performance profiles are all convex. That is, their second derivative is continuous and negative. This assumption may be sufficient to show that r is optimal if and only if it is the minimal allocation for which local compilation allocates no additional time to the copies of e'. Such property would both guarantee the optimality of Method 3 and simplify Method 2.

Complexity

Finally, I determine the complexity of computing each entry in the table representing the compiled performance profile. Again, let the total number of modules be κ, and let the maximal number of discrete time units to be allocated be $\tau = T_{max}/\epsilon$. The complexity of the algorithm is then

$$O(\kappa\tau^3)$$

This is due to the fact that the complexity of the search for r is $O(\tau)$ (since r may be incremented by 1 unit of time in each iteration). The most complicated step inside the loop is the local compilation with complexity $O(\kappa\tau^2)$. Note that in practice the convergence of the search for r is much faster than $O(\tau)$.

The extension to multiple repeated expressions is straightforward. The algorithm needs to maintain a sequence of reserved allocations for each repeated sub-expression. The rest of the algorithm is the same. The advantage of method 3 is that its complexity remains the same with any number of repeated sub-expressions. This is due to the fact that a single loop is used to update all the reserved allocations to the repeated sub-expressions and the worst case complexity of that loop remains $O(\tau)$.

Summary

I have examined three time allocation algorithms designed to cope with the difficulty of compiling DAGs. The first algorithm has a complexity $O(\kappa^3 log\tau)$ but finds only local optimum. The second algorithm has complexity $O(\kappa\tau^{(p+2)})$ and the third $O(\kappa\tau^3)$. Since $\kappa \ll \tau$ the first algorithm is the most efficient one. Method 2 is superior since it guarantees optimality, but its complexity grows exponentially as the number of different repeated sub-expressions grows. To address this problem, the last method can be used. Its complexity remains the same for any number of repeated sub-expressions but it does not guarantee global optimality. However, since it uses local compilation to determine the allocation to the rest of the components, it has a better chance of getting closer to the global optimum than the first method.

5.5.6 Compilation under uncertainty

The discussion of compilation in this chapter was restricted to "Type I" where no active monitoring is assumed. That is, the compilation was restricted to the problem of finding an optimal static schedule for the components that would maximize the expected quality of the output. However, in the analysis of functional composition so far, I used fixed conditional performance profiles only. These performance profiles allow no uncertainty regarding the quality of the output, once the quality of the inputs and the time allocation are determined. This analysis remains valid in cases where the output quality variance is narrow. But in general, the compilation scheme has to be extended to work with probability distribution profiles.

Let e be a composite expression of size n. Consider a particular time allocation $(t_1, ..., t_n)$, where t_i is the time allocation to module i. What is the probabilistic description of the quality of the output given this particular time allocation? When fixed conditional performance profiles are used, the answer to this question can be computed by using an expression similar to e where each function is replaced by its conditional performance profile and each input variable is replaced by its quality. With performance distribution profiles (pdps), the answer to this question becomes more complicated. Before answering it, I need to discuss the representation of pdps.

To represent a pdp, an extension of the discrete tabular representation of fixed performance profiles can be used. Assume that all quality measures are normalized to be in the interval $[0, 1]$. The interval is divided into ℓ discrete qualities, $q_1...q_\ell$. A result is of quality $q_i = (2i-1)/2\ell$ if its actual quality measure q is in the range: $(i-1)/\ell < q \leq i/\ell$. Now, a conditional pdp is a mapping from input quality and run-time into a (standardized) discrete probability distribution over qualities. Let A be a function of one argument and let q represent the quality of its input. Using the previous notation, $Q_A(q, t)$ becomes a vector of probabilities instead of a scalar. Let $Q_A(q, t)[i]$ be the i^{th} entry of this vector which expresses the probability of the output quality being q_i.

Now, consider the simple case of composition represented by the following expression:

$$C(A(x), B(x))$$

where the conditional pdps of A, B and C are given. For any particular allocation to the components, the probability distribution of the output quality can be derived using marginalization over all the possible output qualities of A and B. That is, for a given total allocation t, with sub-allocation of t_A to module A, t_B to module B, and t_C to module C, the probability distribution of the output can be computed in the following way:

$$Q_{out}(q,(t_A,t_B,t_C))[k] = \sum_{i=1}^{\ell}\sum_{j=1}^{\ell} Q_A(q,t_A)[i]Q_B(q,t_B)[j]Q_C(q_i,q_j,t_C)[k] \tag{5.42}$$

This approach can be extended to any expression of any size. To compute the probability of a particular output quality, one need to marginalize over all the possible qualities of intermediate results. The time complexity of generating the table representing the probability distribution of the output for e is $O(\ell^p)$, where p is the number of modules. The complexity of the same task with fixed performance profiles is $O(p)$. This exponential growth means that it may be impractical to apply this computation globally to a large expression. However, similar to local compilation, the probability distribution of the output can be computed for each node based on its immediate inputs. For binary trees the total complexity becomes $O(p\ell^3)$.

It should be noted that when searching for optimal apportionment of time to the components, one has to compare two possible probability distributions of the output quality rather than two scalar output qualities. This comparison is not as straightforward as it may seem. One can, of course, compare the *expected qualities*. However, recall that the quality of the result of a decision method is used in conjunction with a description of the environment and a utility function to determine its actual value. The better the quality the better the value, but a better expected quality does not necessarily correspond to a better expected utility. In some cases, a distribution of qualities with a slightly lower expected value may be preferred because it guarantees a narrow variance and possibly a higher expected utility. To overcome this difficulty, the utility function of the system and a typical description of the environment can be used at compile time so that quality distributions are compared based on their resulting utility rather than on their quality.

5.6 Compilation of other programming structures

In this section I will examine the compilation of additional programming structures. The first part analyzes conditional structures. The second part examines the compilation of boolean expressions. Finally, the last part of this section analyzes the compilation of loops.

5.6.1 Conditional structures

Conditional execution of code is a fundamental structure in programming. I start this section with an analysis of several types of conditionals. To begin, consider a modification of straight line programs that allows each assignment in the program to be of the form,

$$u \leftarrow \textbf{if } C \textbf{ then } F(v_1,...,v_n) \textbf{ else } G(w_1,...,w_m)$$

Even this most basic conditional statement raises several questions regarding the anytime nature of the components:

1. Is the condition C an anytime algorithm?
2. Does C return a truth value or a probability?
3. What is the quality of the result generated by the wrong branch of the structure (e.g. the result of F when C is *false*)?

4. Do we only evaluate one branch or both?

Depending on the answers to these questions, one can design an appropriate compilation scheme. Consider the following cases:

Case 1: Fixed-time correct-answer

In this case, the evaluation of the condition takes constant time (t_C) after which it returns the *correct* truth value. When the value is *true*, the quality of the result is the quality of F. Otherwise, it is the quality of G. No special assumption is made about the utility of the wrong branch[7] since, when conditional expressions are evaluated correctly, the right branches are always selected. For example, consider the problem of bin packing with a given set of n packages. The condition determines whether the packages are convex, in which case algorithm F solves the problem, or non-convex, in which case algorithm G solves the problem.

Since the condition cannot be evaluated at compile time, assume that the prior probability that the condition is *true*, P_C, is known. The expected performance profile of the **if** structure is then,

$$Q_{\mathbf{if}}(t) = \begin{cases} P_C Q_F(t - t_C) + (1 - P_C) Q_G(t - t_C) & \text{if } t > t_C \\ 0 & \text{otherwise} \end{cases} \tag{5.43}$$

Given its performance profile, the conditional assignment can be treated just as any other assignment in a straight line program. Based on this performance profile one can use the efficient compilation methods for DAGs presented in the previous section.

Case 2: Fixed-time probabilistic-answer

In this case, the evaluation of the condition takes constant time (t_C) after which it returns the probability, p, that the answer is "*true*." The correct truth value cannot be determined by the program. I assume that in this case F and G can generate useful results *both* when the correct condition is *true* and when it is *false*. The performance profiles however are different in each case. $Q_{F,true}(f)$ characterizes the quality of F when the condition is *true*, and $Q_{F,false}(t)$ specifies the quality when the condition is *false*. For example, consider the bin packing example of the previous case. Assume that the bin packing algorithm for convex packages can still handle non-convex packages (by computing their convex hull) but its performance in that case is inferior to the performance of the specialized algorithm. Similarly, the bin packing algorithm for non-convex packages can, obviously, handle convex packages but not as efficiently as the specialized algorithm.

If the total allocation is t and the returned probability of the condition being true is p, the algorithm decides at run time whether to allocate the remaining time to F or to G based on comparing their expected values. When F is activated (and $t > t_C$) the expected quality of the results is:

$$pQ_{F,true}(t - t_C) + (1 - p)Q_{F,false}(t - t_C) \tag{5.44}$$

When G is activated (and $t > t_C$) the expected quality of the results is:

$$pQ_{G,true}(t - t_C) + (1 - p)Q_{G,false}(t - t_C) \tag{5.45}$$

---

[7] However, it is a trivial property of any "good" program that the right branch has a higher utility than any wrong branch.

Based on the branch that has the higher expected utility, the run-time system can determine which method should be activated. If one knows the probability distribution of the returned value of the condition, p, one can construct the performance profile of the **if** structure and then use one of the standard compilation methods for DAGs as in case 1.

Case 3: Anytime conditional

In this case, the condition C itself is an anytime algorithm that returns a truth value. Its performance profile describes the probability of correctness as a function of time and truth value[8]:

$$\begin{aligned} Q_{C,true}(t) &= p(\text{Correct Answer} = true \mid C = true) \quad (5.46) \\ Q_{C,false}(t) &= p(\text{Correct Answer} = false \mid C = false) \quad (5.47) \end{aligned}$$

Again, I assume that the correct truth value cannot be determined by the program and that the the quality of the result is the quality of F when the correct condition value is *true*, and the quality of G otherwise. I also assume that the result of the wrong branch has zero quality and that the algorithm executes F when C returns *true* and that it executes G otherwise. For example, suppose that the condition is used to select the correct speech recognition procedure for a particular person based on a certain classification. The condition in this case is an anytime algorithm that determines the membership in one out of two possible classes (male or female, for instance). Once the utterance is classified, an appropriate anytime speech recognition procedure is used. Assume that if the classification is wrong, then the speech recognition procedure fails. Otherwise, its probability of success is determined by its performance profile.

Let $P_C(t)$ be the prior probability that the condition returns *true* with time allocation t. Then, for each total allocation t, the optimal allocation x to the conditional part can be determined by solving the following equation:

$$arg \max_{0 \leq x \leq t} \{P_C(x) Q_{C,true}(x) Q_F(t-x) + (1 - P_C(x)) Q_{C,false}(x) Q_G(t-x)\} \quad (5.48)$$

Once the best allocation to the condition is determined, local compilation of the **if** structure can be completed and its performance profile can be determined. Then, the standard compilation methods for DAGs can be used as in the previous cases.

Summary

Several alternative conditional structures have been analyzed. The analysis shows that the compilation of DAGs can be applied by first deriving the performance profile of the conditional structure, using local compilation techniques, and then treating it as a standard component of the DAG. This result can be extended to include multi-case conditional structures such as the following case construct:

$$u \leftarrow \ \textbf{case}\ C\ \textbf{of}\ \{v_1 : F_1 \ ; \ v_2 : F_2 \ ; \ \ldots \ ; \ v_n : F_n\}$$

Although all the cases that were considered above can be applied to this construct, the first case seems to be the most useful one. It would allow $F_1, \ldots, F_n$ to be anytime algorithms while C must be an expression whose evaluation returns the exact answer within a fixed run-time.

---

[8] This kind of performance profiles and their compilation are discussed at length in the following part on boolean expressions.

Multiple versions of conditional constructs raise an important methodological question: How would the programmer indicate the version of conditional structure that is used. One approach is to limit the semantics of all **if** structures in a program to only one case depending on the application. Another approach is to use different keywords for different types of conditional structures. Finally, it should be noted that the last case, where the conditional part is an anytime algorithm, could be automatically recognized since the compiler has information in the anytime library on all the anytime components.

5.6.2 Compilation of boolean expressions

This section examines the compilation of boolean expressions which are composed of anytime boolean functions defined as follows:

Definition 5.20 *An* **anytime boolean function** *is a function that returns either T (true) or F (false). Its performance profile determines the probability that the answer is correct as a function of the returned truth value and time.*

Let f be an anytime boolean function that takes input I and calculates the truth value of the relation $r(I)$. For any possible time allocation, t, the performance profile of f specifies:

$$Q_f(T,t) = p(r(I) = T | f = T)$$

$$Q_f(F,t) = p(r(I) = F | f = F)$$

obviously,

$$1 - Q_f(T,t) = p(r(I) = F | f = T)$$

$$1 - Q_f(F,t) = p(r(I) = T | f = F)$$

Note also that the performance profile has to satisfy the property that $Q_f(T,0) = 1 - Q_f(F,0)$ since with zero allocation the algorithm has no chance to produce any result other than a default answer. The default answer should be the most likely answer based on the *prior probability distribution*. However, for any $t > 0$, it is possible that: $Q_f(T,t) \neq 1 - Q_f(F,t)$.

The definition allows performance profiles to depend on the results themselves since such dependency exists in many general techniques for implementing anytime boolean functions. For example, when a Monte Carlo algorithm returns a positive answer, it must be correct (i.e. $Q_f(T,t) = 1$). However, when the answer is negative, it is only because the algorithm remains silent. Its failure to give a positive answer in a series of trials gives evidence that the correct answer is negative. Therefore the only meaningful performance profile is for the case in which the algorithm returns F (i.e. $Q_f(F,t)$).

Note that the fact that the performance profile of each elementary algorithm depends on time as well as on the result produced by the algorithm makes the compilation task more complicated. Normally, the result of a module is a function of the results of the elementary components (only!) and the quality is a function of the qualities (only!). However, this kind of separability is not simple to achieve with boolean expressions, even if one assumes that the elementary performance profiles are independent of the results. For example, consider the following expression:

$$e = (f_1 \wedge f_2 \wedge f_3)$$

Suppose that the results of the functions are independent, that each function returns the same truth value and that the probability of correctness is $p = 0.7$ for all three functions. If the answer is F, then $p(e = F) =$

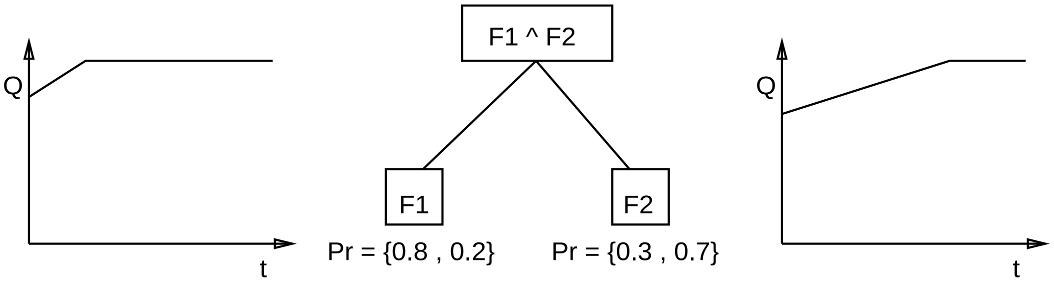


Figure 5.11: Compilation of a boolean expression

0.973 and the answer for the total expression should be the standard result of $(F \wedge F \wedge F)$. However, if the answer is T, then $p(e = T) = 0.343$ and $p(e = F) = 0.657$. Hence, the more likely answer to the total expression is F which is the negation of the standard result of $(T \wedge T \wedge T)$. In conclusion, both the result of a boolean expression and its quality are defined as a function of the results and qualities of the elementary components. For the purpose of analyzing the compilation of boolean expressions, the following definition is used:

Definition 5.21 *A* **boolean expression** *over a set, $\mathcal{S}$, of elementary anytime boolean functions is:*

1. *Any elementary anytime boolean function $e \in \mathcal{S}$.*
2. *The expression $\neg e$ where e is a boolean expression.*
3. *The expression $(e_1 \wedge e_2)$ where e_1 and e_2 are boolean expressions.*
4. *The expression $(e_1 \vee e_2)$ where e_1 and e_2 are boolean expressions.*

Anytime evaluation of boolean expressions

Given a boolean expression over a set $\{f_1, ..., f_n\}$ of elementary boolean functions, two methods will be shown for time allocation to the components. Both methods are based on the following assumptions:

1. The truth values returned by the elementary boolean functions are independent, that is, $p(f_i|f_j) = p(f_i)$.
2. The time needed to compute the truth value of each function is much longer than the time needed to compute the value of the expression once the truth values of all the functions are known.

The first assumption simplifies the combination of probabilities. The second assumption allows us to allocate all the available time to the anytime components since the evaluation time of the boolean expression itself is negligible.

Method 1:

The first method produces a contract algorithm. It is based on a compilation process that produces the allocation to the components for each given total allocation. Consider first the simple example shown in Figure 5.11. Given the expression,

$$e = (F_1 \wedge F_2) \tag{5.49}$$

Table 5.1: Optimal time allocation in boolean expression evaluation

Total Time	t_1	t_2	E(Quality)
1	0	1	0.7498
4	0	4	0.7694
7	0	7	0.7891
10	0	10	0.8088
13	0	13	0.8285
16	0	16	0.8482
19	0	19	0.8678
22	2	20	0.8881
25	3	22	0.9093
28	5	23	0.9315
31	6	25	0.9547
34	6	28	0.9783
37	7	30	1.0000

where the qualities of of F_1 and F_2 are described by the following functions:

$$Q_1(t) = \min\{0.8 + 0.03t, 1\} \tag{5.50}$$

$$Q_2(t) = \min\{0.7 + 0.01t, 1\} \tag{5.51}$$

the compilation problem is to find for each total allocation the best way to allocate time to the components so as to maximize the expected confidence level in the result of the whole expression. In order to find the best allocation for any given total time t, we can use a search method similar to methods used in the previous section. The only difference is the way in which the expected quality of the whole expression is calculated for each particular allocation to the components. While in functional composition one could compute the quality of the output based on the qualities of the components only, here the particular results of the components are needed as well. The result of each component is, of course, unknown at compile time. However, based on the prior probability distribution of each component, the joint probability distribution can be derived. The new confidence level of each component is known from the performance profiles of the components. Hence, for each possible set of results of F_1 and F_2, one can compute the new probability distribution (or confidence level) of the result and, accordingly, the new output quality. Finally, the expected quality of the whole expression can be computed based on the joint probability distribution of the results of the components. Table 5.1 shows the time allocation that was computed using this compilation technique. A tabular representation of the compiled performance profile is a refinement of this table along the time axis.

The above compilation method can be extended to a global compilation scheme for boolean expressions. Its complexity, however, grows exponentially with the size of the expression. Local compilation can only be applied under certain assumptions. When compiling a single boolean function, it is sufficient to assume that the results of the individual components are independent. But, when one tries to extend the above method to DAGs, the results of internal nodes may be dependent. Therefore, to apply local compilation one must assume *subtree independence*. This assumption restricts the structure of the expression to a tree and requires that the truth values of any two disjoint subtrees are independent. In other words, the subsets of elementary anytime functions used by any two sub-expressions must be disjoint. For such expressions, the following local compilation scheme is applicable.

To formalize local compilation of boolean expressions, the following notation is needed. Let v_i represent a possible truth value of a boolean expression e_i, that is $v_i \in \{T, F\}$. A binary probability distribution is denoted by a pair: $\{p, 1-p\}$ where the first component is the probability of T and the second the probability of F. The function $Dist(v_i, q_i)$ is a binary probability distribution such that $p(e_i = v_i) = q_i$. For example, $Dist(T, 0.8)$ is $\{0.8, 0.2\}$ and $Dist(F, 0.9)$ is $\{0.1, 0.9\}$. The boolean operators are extended to probability distributions rather than truth values in the following way:

$$\neg\{p_1, 1-p_1\} = \{1-p_1, p_1\} \tag{5.52}$$

$$\{p_1, 1-p_1\} \wedge \{p_2, 1-p_2\} = \{p_1p_2, 1-p_1p_2\} \tag{5.53}$$

$$\{p_1, 1-p_1\} \vee \{p_2, 1-p_2\} = \{1-(1-p_1)(1-p_2), (1-p_1)(1-p_2)\} \tag{5.54}$$

The quality of a probability distribution is simply $Qual(\{p, 1-p\}) = \max(p, 1-p)$.

Local compilation is defined by induction on the structure of the boolean expression. I assume that a prior probability distribution for the elementary functions is known. The prior probability distribution of each node can be easily calculated. Local compilation works as follows:

1. If the expression is an elementary anytime boolean function, then its performance profile is given. Hence no compilation is necessary.

2. If the expression is of the form $\neg e$, then its performance profile is the same as the performance profile of e.

3. If the expression is of the form $(e_1 \wedge e_2)$, then for each total allocation t, the best quality of result is achieved by solving the equation:

$$arg \max_{0 \le x \le t} \{Q_\wedge(Q_1(x), Q_2(t-x))\} \tag{5.55}$$

   where Q_1 and Q_2 are the (possibly compiled) performance profiles of e_1 and e_2 respectively. The equation also defines the best time allocation to the components. The function $Q_\wedge$ is defined as follows:

$$Q_\wedge(q_1, q_2) = \sum_{v_1=T,F} \sum_{v_2=T,F} p(e_1 = v_1)p(e_2 = v_2)Qual(Dist(v_1, q_1) \wedge Dist(v_2, q_2)) \tag{5.56}$$

4. If the expression is of the form $(e_1 \vee e_2)$, then for each total allocation t, the best quality of result is achieved by solving a similar equation:

$$arg \max_{0 \le x \le t} \{Q_\vee(Q_1(x), Q_2(t-x))\} \tag{5.57}$$

   where $Q_\vee$ is defined as follows:

$$Q_\vee(q_1, q_2) = \sum_{v_1=T,F} \sum_{v_2=T,F} p(e_1 = v_1)p(e_2 = v_2)Qual(Dist(v_1, q_1) \vee Dist(v_2, q_2)) \tag{5.58}$$

Definition 5.22 *A local compilation scheme is said to be* **consistent** *with respect to a certain equivalence relation, if it yields the same performance profile for all the members of each equivalence class.*

Consistency of local compilation is an important property. It guarantees that compilation is not sensitive to trivial representation changes. Consider, for example, the equivalence relation over boolean expressions that holds for certain expressions if one can be derived from the other using the commutative, associative, and distributive rules. Is local compilation of boolean expressions consistent with respect to this equivalence class? Empirical results suggest that the answer is positive. For example, local compilation of the two expressions $(F_1 \wedge F_2) \wedge F_3$ and $F_1 \wedge (F_2 \wedge F_3)$ yields the same allocation to all three components for each total allocation. However, both the consistency of local compilation and its global optimality are yet to be proved. Note that optimality of local compilation implies consistency, but the converse is not true.

Finally, as with composite expressions, global compilation is impractical for large expressions. Therefore, when the subtree independence assumption does not hold, local compilation should still be used as an approximate compilation method.

Method 2:

The second method for evaluating an anytime boolean expression is based on direct construction of an interruptible algorithm. Any boolean expression over $f_1, ..., f_n$, can be represented as a DAG. The method is based on a greedy algorithm that repeatedly selects a single leaf node whose computational effect on the quality of the expression is maximal and allocates a fixed amount of time to that node.

How does the algorithm select the leaf node with maximal effect? It simply considers every alternative. For each candidate, the new expected quality of the expression can be calculated based on the current probability distribution associated with the leaf nodes and the performance profile of the candidate. The output of the candidate node is unknown but the current distribution associated with it can be used to compute the expected quality of the whole expression. The interruptible greedy algorithm is similar to global compilation with a fixed time allocation, equal to the time quota of each iteration. What simplifies the search problem is the restriction that only one component gets the whole allocation in each iteration.

An interesting variation of this greedy algorithm is the special case where the performance profiles of all the elementary components are step functions. That is, the components themselves are not anytime algorithms. By allocating a certain fixed amount of time to each component, one can get the *exact* truth value of that node. In this particular case, the greedy algorithm creates an *ordering* of the components so that early ones have greater effect on the quality of the result. The ordering can be calculated once in advance or the next candidate can be picked at run-time based on the *actual* truth values of the other components.

For example, consider the following expression:

$$F_1 \vee (F_2 \wedge F_3)$$

Figure 5.12 shows the corresponding graph and the prior probability distributions of the leaf nodes and the internal nodes. Suppose that, by allocating one unit of time, the exact truth value of any leaf node can be computed. Which node should get the first time unit? The initial quality of the root node is 0.875. The greedy algorithm would simply consider the new expected quality as a result of evaluating each possible node:

1. If F_1 is evaluated first, the expected quality is:

$$\textit{Expected Quality} = 0.8 \cdot 1.0 + 0.2 \cdot 0.625 = 0.925$$

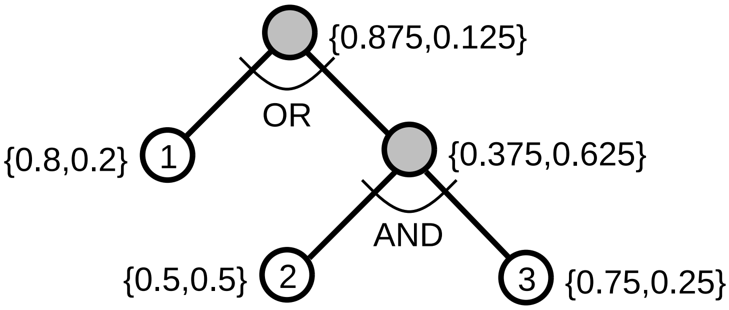


Figure 5.12: Interruptible evaluation of a boolean expression

2. If F_2 is evaluated first, then the expected quality is:

$$\textit{Expected Quality} = 0.5 \cdot 0.95 + 0.5 \cdot 0.8 = 0.875$$

3. If F_3 is evaluated first, then the expected quality is:

$$\textit{Expected Quality} = 0.75 \cdot 0.9 + 0.25 \cdot 0.8 = 0.875$$

Therefore the algorithm would select F_1 for evaluation in the first iteration. If the result of F_1 is T, then the result of the expression is T with probability 1 and the computation terminates. Otherwise, the algorithm will have to select between evaluation of F_2 and F_3. The initial quality becomes 0.625 and the expected qualities of the computations are as follows:

1. If F_2 is evaluated first, the expected quality is:

$$\textit{Expected Quality} = 0.5 \cdot 0.75 + 0.5 \cdot 1.0 = 0.875$$

2. If F_3 is evaluated first, the expected quality is:

$$\textit{Expected Quality} = 0.75 \cdot 0.5 + 0.25 \cdot 1.0 = 0.625$$

Therefore the algorithm would select F_2 for evaluation. If the result is F, then the computation terminates. Otherwise the last function would be evaluated. Note that in this particular example, the optimal order of evaluation is F_1, F_2, F_3 in all cases, thus the order can be determined by an off-line computation.

The compilation of boolean expressions raises many interesting questions. How does the performance profile of the contract method relate to the performance profile produced by the interruptible evaluation? What is the effect of reducing the time step of the interruptible evaluation on the performance profile? Note that the time necessary to select the next component for allocation is considered to be negligible. This assumption becomes invalid as the time step becomes infinitesimal.

5.6.3 Compilation of loops

The compilation of loops is more complicated than the compilation of other programming structures. As with the previous structures, the key question is the relationship between the quality of the loop and the

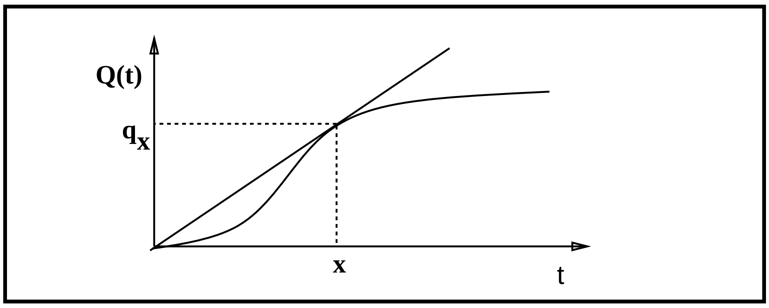


Figure 5.13: Compilation of an unbounded loop

quality of its components, that is, the body and exit condition. I will generally assume that the body of the loop is an anytime algorithm whose performance profile is given. Several loop structures are analyzed below.

Unbounded loops

Any system that repeatedly performs a complex task can be implemented using a loop through a sequential anytime process. Examples include operating systems, part-picking robots, and network communication servers. In these cases, an unbounded loop is an adequate model:

$$\textbf{loop } S$$

Assuming that S is an anytime algorithm whose performance profile is given and that utility (or quality) is additive over repeated activation of S, time allocation should maximize the utility gain per time unit, that is, at each iteration x is selected such that

$$\max_{0 \leq x \leq t} \{Q_S(x)/x\} \tag{5.59}$$

where $Q_S(x)$ is the performance profile of the body of the loop. This amounts to stopping the sequence when it reaches the point of contact of the steepest tangent to the performance profile as shown in Figure 5.13.

What happens if the result of each iteration is part of the input to the next one? Obviously, quality is not additive any longer and the optimization problem is much more complicated. Suppose that $Q_S(q, t)$ is the conditional performance profile of the body of the loop. For any fixed number of iterations, n, the loop can be viewed as a simple linear composition of a single function duplicated n times. Since the optimality of local compilation is guaranteed for such cases (by Theorem 5.16), a sequence of performance profiles, $Q_S^n(q, t)$, can be defined such that each element is the result of local compilation of the previous one and $Q_S(q, t)$. For any given input quality q and total allocation t, the best number of iterations is determined by:

$$\max_n \{Q_S^n(q, t)\} \tag{5.60}$$

Once n is determined, the particular sequence of allocations can be derived by local compilation of the n duplicates.

Fixed length loops

Fixed length loops are loops that are executed a fixed number of times, although the number of times may be determined at run-time. Their general structure is

$$\textbf{for}\ i = 1\ \textbf{to}\ n\ \textbf{do}\ S(i)$$

I will consider the case where such loops are used for applying a certain operation to all the members of a certain set or array. The total quality is additive and the qualities of individual iterations are similar. In addition, I assume that the performance profile of each iteration grows faster at early stages of execution. Based on these assumptions, all iterations must be performed and time should be divided equally between them. If a contract algorithm is constructed, then the total time allocation should be divided equally between the iterations. If an interruptible algorithm is constructed, then the best strategy is to allocate small amounts of time increments to the evaluation of each $S(i)$.

The above description is rather general, but it does not cover all the cases. It is also possible, for example, that a fixed length loop would apply each iteration to the results of the previous one. In that case, the local compilation scheme described previously for unbounded loops would apply. In fact, the case of fixed length loops is less complicated since the number of iterations is known at activation time.

Conditional loops

Finally, consider the compilation of conditional loops. In particular, consider the following structure:

$$\textbf{while}\ C(e)\ \textbf{do}\ S$$

There are many possible ways to define the behavior of such anytime structures. The conditional part may be an anytime algorithm and its returned value may be a probability rather than a truth value. The execution of the body of the loop may have a negative effect when the condition does not hold or it may have no effect in that case. The quality of the whole structure may be additive over individual iterations or each iteration may contribute to the overall quality in a more complex way. As a result, the compilation of loops of this kind is hard to perform in general. Instead, I describe some particular examples that show how compilation might be implemented.

1. One possible use of conditional loops is in situations where the condition determines whether any input is ready to be processed and the body implements a certain processing step. For example,

   ```
   while not EMPTY(Query-Queue) do
       PROCESS(POP(Query-Queue))
   ```

   Typically in such situations the utility is additive over the individual queries. Assume that the time necessary to evaluate the condition is negligible. In such a case, it seems that the control of the execution of the loop becomes a dynamic monitoring problem. The monitor, using a model of the environment, could determine the time pressure when a new query arrives. The length of the queue can be used as an additional factor to determine the time pressure. Then, a particular contract time can be derived using the performance profile of the body of the loop. Hence, off-line compilation does not seem to be useful in this case.

2. Another possible use of conditional loops is in situations where the condition enables the operation of the body of the loop. The run-time system may have some control over the status of the condition. In the context of anytime computation, both the condition and the body may consume variable resources. For example,

```
while (IDENTIFY-TARGET(T) $\wedge$ NOT-REACHABLE(T)) do
    MOVE-TOWARDS-TARGET(T)
```

   In such a situation, the performance profile of the body may express the probability of reaching the target as a function of distance and time *assuming* that the target is static. The performance profile of the condition may express the probability of keeping track of the target as a function of tracking time. In this case, compilation is possible. It will result in a certain degradation in the performance profile of the body due to the need for constant tracking. In addition, this type of compilation requires active monitoring.

3. Finally, conditional loops can be translated into the following representation:

```
loop
    if $C(e)$ then $S$ else exit
```

   The advantage of this representation is that the compilation of conditional loops is transformed into a two-step compilation process. The first step is the compilation of the conditional structure and the second is the compilation of the unbounded loop.

To summarize, there is a wide range of possible loop constructs, some of which can be efficiently compiled. The control of other structures becomes a run-time monitoring problem. Practical experience shows that the open ended types of loops, that are more difficult to compile, are useful at the very top level of an anytime computation system. I will further examine the top level of anytime systems and their activation in the next two chapters.

5.7 Summary

This chapter examined the possibility of producing the performance profile of large programs based on the performance profiles of their components. Such an off-line compilation process is crucial for the efficient implementation of the meta-level control that supports operational rationality. The global compilation problem is well defined for many programming structures, but its solution can be rather expensive. Its time complexity tends to grow exponentially with the size of the program. To cope with this exponential growth, I have developed local compilation methods that are performed on one program structure at a time. Local compilation is not only more efficient but also supports modular development of anytime algorithms. In addition, the global optimality of local compilation has been established for the general case of functional composition without repeated sub-expressions.

Chapter 6

Run-Time Monitoring of Anytime Algorithms

> What is actual is actual only for one time
> And only for one place.
>
> T. S. Eliot, *Ash Wednesday*

Monitoring plays a central role in anytime computation because it complements anytime algorithms with a mechanism that determines their run-time. Without such a mechanism, the anytime components of a system are worthless. The last two chapters concentrated on the construction of anytime performance components and on their compilation. In this chapter I examine the monitoring problem and develop several monitoring strategies. In particular, I show that for a certain class of problems it is sufficient to make all the monitoring decisions when the system is activated. In other domains, active monitoring, a more complex mechanism, is essential to guarantee operational rationality in the face of uncertainty.

6.1 The run-time system

The run-time system complements anytime algorithms with a monitoring mechanism that determines their run-time. In this section I define the general notion of a monitoring scheme and distinguish between passive and active monitoring. Then I analyze the conditions under which active monitoring is necessary. Finally, I determine the temporal scope and goal of the monitoring system.

6.1.1 Monitoring schemes

Given a compound anytime program, $\mathcal{P}$, whose elementary anytime components are $E = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$, a monitoring scheme is defined as a mapping that determines a certain time allocation for each activation of an elementary component.

Definition 6.1 *A* **monitoring scheme** *for a program $\mathcal{P}$ is a mapping:*

$$\mathcal{M} : E \times Z^+ \rightarrow R^+$$

where E is the set of elementary components of $\mathcal{P}$.

In other words, $\mathcal{M}(i, j)$ is the time allocation to the j^{th} activation of the i^{th} component. A monitoring scheme supplies the necessary information to make a compound anytime program executable in a well defined way. It fixes the degree of freedom associated with anytime algorithms.

Much of this chapter is dedicated to the development of various monitoring schemes and to the analysis of their properties. An important distinction is made between *passive* and *active* monitoring.

Definition 6.2 *A monitoring scheme is said to be* **passive** *if the corresponding time allocation mapping is completely determined prior to the activation of the system.*

Definition 6.3 *A monitoring scheme is said to be* **active** *if it is not passive. That is, the corresponding time allocation mapping is partially determined while the system is active.*

Under active monitoring, some scheduling decisions are made at run-time. Such decisions are based on the *actual* quality of results produced by the anytime components and based on the *actual* change that occurred in the environment. It should be emphasized that compilation of anytime algorithms remains as essential in active monitoring as it is necessary in passive monitoring. The run-time scheduling decisions are made using compiled performance profiles both in order to determine an initial time-allocation to each component and in order to determine whether time allocation should be revised.

Figure 6.1 shows the general structure of the run-time system and the data flow between its main components. The monitor is the central component that makes run-time scheduling decisions. It represents the implementation of a particular monitoring scheme. An independent process is used in order to update the state of the environment based on sensory input. This process is performed in parallel to the execution of the main decision procedure, but it does not consume the same computational resources. The state of the environment, or more precisely, a set of high-level features of the environment, is used together with a model of the environment, the current best results, and the performance profile of the main decision procedure in order to determine the value of continued computation. The value of continued computation can be estimated for the complete system or for individual modules. In the former case, the monitor may decide to stop the execution of the main decision procedure and return the current best result. In the latter case, the monitor may decide to transfer control to another anytime component or it may interrupt the execution of the complete system. Specific examples of monitoring policies are presented in the following sections.

The main reason why complicated active monitoring is necessary in control of anytime algorithms is the problem of uncertainty. In an entirely deterministic world, passive monitoring can yield optimal performance and hence satisfy the operational rationality criterion. However, in unpredictable domains there is much to be gained in performance by introducing an active monitoring component. In the next section, the two main sources of uncertainty in real-time systems are described.

6.1.2 Uncertainty in real-time systems

Two primary sources of uncertainty affect the operation of real-time systems. The first source is internal to the system. It is caused by the unpredictable behavior of the system itself. The second source is external. It is caused by unpredictable changes in the environment. Obviously, each source may contribute a variable degree of uncertainty depending on the problem domain and the implementation of the system.

1. *Uncertainty regarding the performance of the system*

   In the general case of an anytime algorithm, the quality of the results may vary for any fixed time allocation. A performance distribution profile describes the distribution of quality for any time allocation.

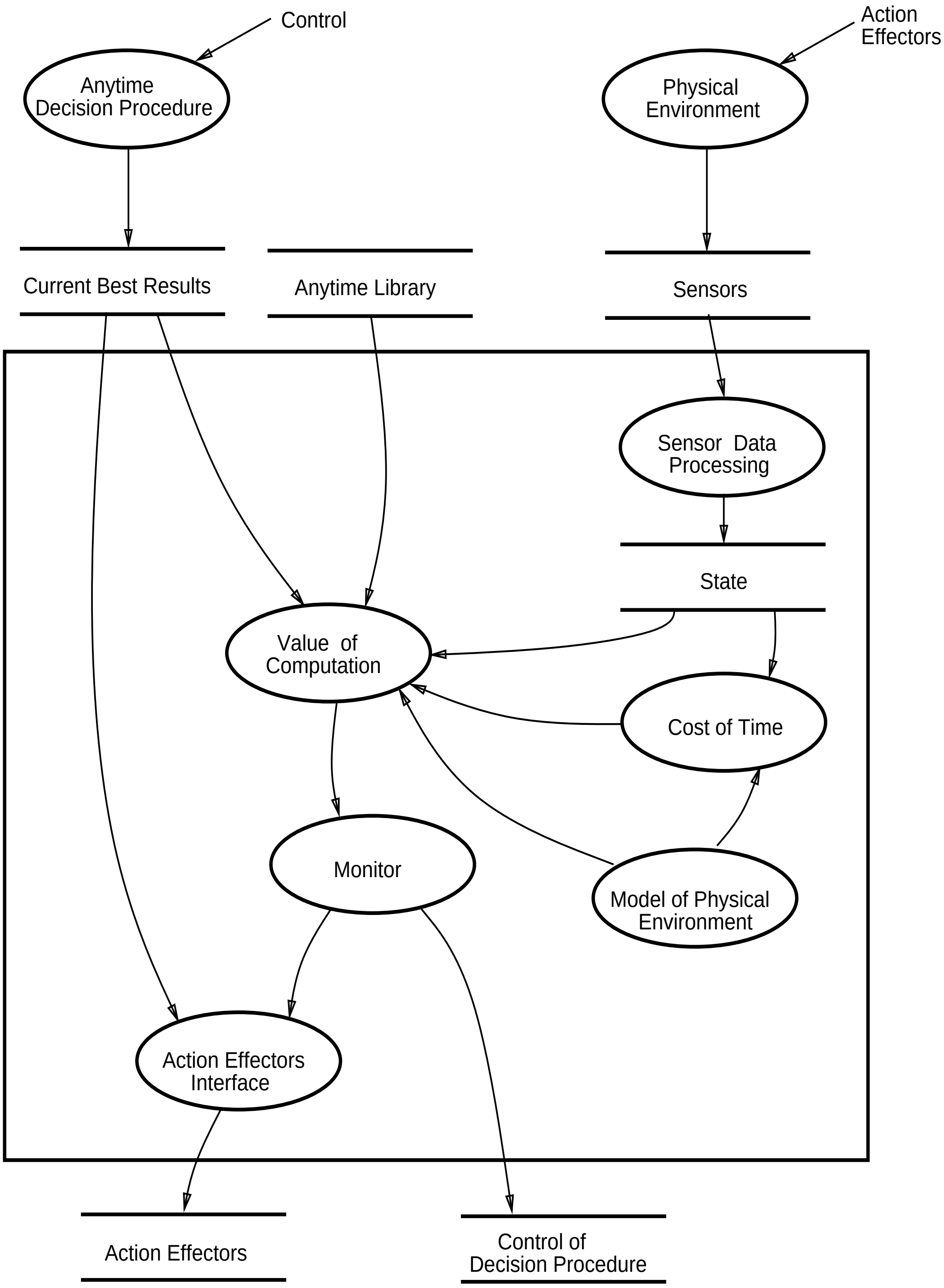


Figure 6.1: Monitoring anytime computation

In situations where the variance of the distribution is small or where the actual quality is bounded by a small δ around the expected quality, this uncertainty may have little effect on meta-level control of computation. However, with a large variance there is much to be gained by using active monitoring. A monitor may use the quality of the *actual* result to correct the time allocation to the components of the system.

2. *Uncertainty regarding the environment*

   The desired run-time of a system is determined by its performance profile as well as by the time pressure and by the characteristics of the environment in which it operates. The source of the time pressure is change in the environment that may render the result of the computation useless. Change in the environment creates time pressure but does not always bring about uncertainty. The source of uncertainty is either the presence of stochastic events or the use of approximate models of the environment. An agent that provides a certain service, such as package delivery, in an environment where requests for the service arrive at a non-deterministic rate, operates in a stochastic environment. An agent that performs medical diagnosis operates in an environment that has an approximate model. In both cases, there is uncertainty regarding the future states of the environment and, as a result, any initial contract time may need to be revised. In some situations, an interruptible algorithm must be used due to great variability in time pressure. Active monitoring, however, may be used to control anytime computation in such environments.

The two sources of uncertainty mentioned above are characterized by two separate knowledge sources. Uncertainty regarding the performance of the system is characterized by the performance distribution profile of the system. Uncertainty regarding the future state of the environment is characterized by the model of the environment. Active monitoring is required in the presence of any one of these sources of uncertainty. However, the type of monitoring may vary as a function of the source of uncertainty and the degree of uncertainty.

6.1.3 Episodic problem solving

Operational rationality has been defined as an optimization problem whose ultimate goal is to generate a utility maximizing behavior. However, the time segment over which the agent's behavior is optimized has not been clearly specified. In some cases, the answer to this question is obvious. The agent is presented with a *single* task and the time segment is the period of time required by the agent to achieve that task. For example, consider a robot that delivers packages and whose utility function is the sum of the time-dependent delivery values of the packages. The behavior of the robot is optimized over a single instance of the problem. That is, the robot is presented with a map, a set of packages to be delivered and their time-dependent delivery values. Operational rationality is achieved by optimizing resource allocation to the computational components so as to maximize the utility function. When solving the optimization problem, the input task is considered in isolation. The fact that the same agent may be involved in a completely different activity which may be more beneficial is not considered. Moreover, the fact that another batch of packages may be waiting in a queue for delivery is ignored as well. I call the situation where the achievement of a *single* task is optimized, *episodic problem solving*. Note that the task may be arbitrarily complicated and may include a number of sub-tasks. What makes a problem solving process episodic is not the simplicity of the task, but the fact that the *complete* task is specified as input to the system and the optimization is performed over that task only.

In episodic problem solving, the alternative use of the agent is ignored. In particular, its availability for solving further problems immediately after accomplishing the current task is disregarded. In the rest of this chapter I will focus mainly on monitoring of episodic problem solving. Optimizing behavior over a larger time segments requires to extend the framework of anytime computation to the other components of the agent, namely sensing and action. This extension is examined in the following chapter. It should be noted that, to some extent, the possibility of using the agent for solving other tasks can be factored into the model using a modified utility function. Such a modification must increase the cost of time as a function of the contents of the queue of waiting tasks.

The notions of passive and active monitoring have been defined and the temporal scope of the monitoring component has been restricted to episodic problem solving. The rest of this chapter examines several monitoring schemes. It is divided into two main parts according to the type of system being monitored – contract or interruptible.

6.2 Monitoring contract algorithms

It is easier to construct contract algorithms than interruptible ones, both as elementary and as compound algorithms. Therefore, I will examine first the monitoring problem assuming that the complete system is presented as a contract algorithm, $\mathcal{A}$. The conditional performance profile of the system is $Q_{\mathcal{A}}(q,t)$ where q is the input quality and t is the time allocation. Recall that $Q_{\mathcal{A}}(q,t)$ represents, in the general case, a probability distribution. When a discrete representation is used, $Q_{\mathcal{A}}(q,t)[q_i]$ denotes the probability of output quality q_i.

Let S_0 be the current state of the domain and let S_t represent the state of the domain at time t, let q_t represent the quality of the result of the contract anytime algorithm at time t. $U_{\mathcal{A}}(S,t,q)$ represents the utility of a result of quality q in state S at time t. This utility function is given as part of the problem description. The purpose of the monitor is to maximize the expected utility of the result, that is, to find t for which $U_{\mathcal{A}}(S_t,t,q_t)$ is maximal. Contract algorithms are especially useful in a particular type of domains which is defined as follows:

Definition 6.4 *A domain is said to have* **predictable utility** *if $U_{\mathcal{A}}(S_t,t,q)$ can be determined for any future time, t, and quality of results, q, once the current state of the domain, S_0, is known.*

The notion of predictable utility is a property of domains. The same utility function can be predictable in one domain and unpredictable in another. What makes a domain predictable is the capability to determine the exact value of results of a particular quality at any future time. Hence, the state of the domain may change, even in an unpredictable way, and utility may still be predictable. To explain this situation, I define a function, $f(S)$, that isolates the features of a state that determine its utility. In other words,

$$\forall S_1, S_2 \;\; f(S_1) = f(S_2) \;\Rightarrow\; U_{\mathcal{A}}(S_1,t,q) = U_{\mathcal{A}}(S_2,t,q) \tag{6.1}$$

Consider the domain of traffic on a particular road. The state of the domain is defined by the location and velocity of each vehicle and $f(S)$ may be, for example, the traffic density. Using the function f, it is easy to show that a domain with predictable utility is a domain for which $f(S_t)$ can be determined once the current state, S_0, is known. In general, three typical cases of such domains can be identified:

1. A static domain is obviously predictable since $S_t = S_0$ and $f(S_t) = f(S_0)$. For example, the game of chess constitutes a static domain.

2. A domain that has a deterministic model is predictable since future states can be uniquely determined and hence $f(S_t)$ can be determined. For example, a domain that includes moving objects has a deterministic model when the velocity of each object is constant.

3. A domain for which there is a deterministic model to compute $f(S_t)$, once the current state is known, is predictable. Note that this does not require a deterministic model of the domain itself. An important sub-class is all the domains for which $f(S) = \emptyset$, that is, domains in which the utility function depends only on time.

The initial contract time

The first step in monitoring of contract algorithms involves the calculation of the initial contract time. Due to uncertainty concerning the quality of the result of the algorithm, the expected utility of the result at time t is represented by:

$$U'_{\mathcal{A}}(S_t, t) = \sum_i Q_{\mathcal{A}}(q, t)[q_i] U_{\mathcal{A}}(S_t, t, q_i) \tag{6.2}$$

The probability distribution of future output quality is provided by the performance profile of the algorithm. Hence, an initial contract time, t_c, can be determined before the system is activated by solving the following equation:

$$t_c = arg \max_t \{U'_{\mathcal{A}}(S_t, t)\} \tag{6.3}$$

Passive monitoring means that this initial contract time is used to determine, using the compiled performance profile of the system, the ultimate allocation to each component.

In some cases, it is possible to separate the value of the results from the time used to generate them. In such cases, one can express the comprehensive utility function, $U_{\mathcal{A}}(S, t, q)$ as the difference between two functions:

$$U_{\mathcal{A}}(S_t, t, q) = V_{\mathcal{A}}(S_0, q) - Cost(S_0, t) \tag{6.4}$$

where $V_{\mathcal{A}}(S, q)$ is the value of a result of quality q in a particular state S (termed *intrinsic utility* [Russell and Wefald, 1989b]) and $Cost(S, t)$ is the cost of t time units provided that the current state is S. Similar to the expected utility, the expected intrinsic utility for any allocation of time can be calculated using the performance profile of the algorithm:

$$V'_{\mathcal{A}}(S, t) = \sum_i Q_{\mathcal{A}}(q, t)[q_i] V_{\mathcal{A}}(S, q_i) \tag{6.5}$$

Finally, the initial contract time can be determined by solving the following equation:

$$t_c = arg \max_t \{V'_{\mathcal{A}}(S_0, t) - Cost(S_0, t)\} \tag{6.6}$$

Once an initial contract time is determined, several monitoring policies can be applied. The most trivial one is the *fixed-contract* strategy that leads to a passive monitoring scheme. Under this strategy, the initial contract time and the compiled performance profile of the system are used to determine the allocation to the components. This allocation remains constant until the termination of the problem solving episode. The fixed-contract policy is optimal under the following conditions:

Theorem 6.5 Optimality of monitoring of contract algorithms. *The fixed-contract monitoring strategy is optimal when the domain has predictable utility and the system has a fixed performance profile.*

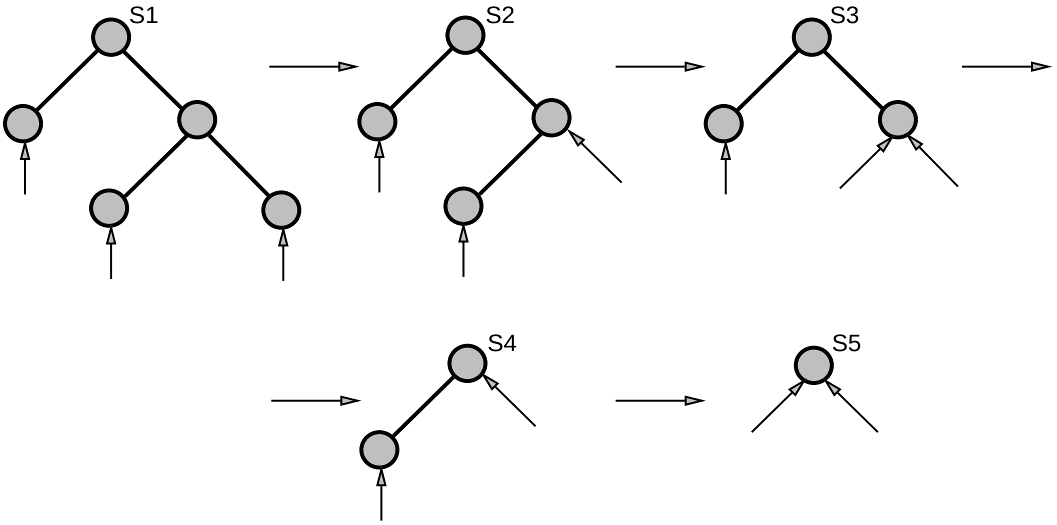


Figure 6.2: A sequence of residual sub-systems

Proof: This result is rather trivial since, when the domain has predictable utility and the system's performance profile is fixed, utility of results at any future time can be determined. The initial contract time, that maximizes the comprehensive utility, remains the same during the computation and no additional scheduling decision can improve the performance of the system. □.

The implication of this theorem is that operational rationality can be achieved in certain domains using a simple, passive monitoring scheme. More interestingly, however, is the applicability of this monitoring method in situations that approximately meet the conditions of the theorem. For example, the fixed-contract approach is efficient when the performance profile of the system is not fixed but the performance variance is small.

The rest of this section discusses two extensions to the fixed-contract policy for cases with high degree of uncertainty regarding the quality of the results. In such cases, the initial contract time must be altered by an active monitoring component.

6.2.1 Re-allocating residual time

The first type of active monitoring that I analyze involves reallocation of residual time among the remaining anytime algorithms. Suppose that a system, composed of several elementary contract algorithms, is compiled into an optimal compound contract algorithm. Since the results of the elementary contract algorithms are not available during their execution, the only point of time where active monitoring can take place is *between* activations of the elementary components. Based on the structure of the system, an execution order can be defined for the elementary components. The execution of any elementary component can be viewed as a transformation of a node in the graph representing the program from a computational node to an external "input" of a certain quality. This transformation is shown in Figure 6.2. The quality of the new input is only known when the corresponding elementary component terminates. Based on the actual quality, the remaining time (with respect to the global contract) can be reallocated among the remaining computational components to yield a performance improvement with respect to allocation that was based on the probabilistic knowledge of quality of intermediate results.

In order to be able to allocate time optimally to each component, the monitor needs to access not only the performance profile of the complete system, but also the performance profiles of the residual sub-systems. The compilation problem has to be solved for each residual system. For example, for the system modeled by Figure 6.2, five performance profiles must be calculated. These performance profiles can be derived using the standard local compilation technique. The only difference is that the compiler does not need to store the allocation to all the components but only the allocation to the next component in the activation order. This multi-compilation and monitoring scheme will be demonstrated by an example at the end of this section.

6.2.2 Adjusting contract time

The second type of active monitoring for contract algorithms involves adjustments to the original contract time. As before, once an elementary component terminates, the monitor can consider its output as an input to a smaller residual system composed of the remaining anytime algorithms. By solving the previous equation that determines the contract time for the residual system, a better contract time can be determined that takes into account the actual quality of the intermediate results generated so far.

If the elementary components are interruptible, the contract time can be adjusted *while* an elementary component is running. Given the quality of the results generated by that component and its performance profile[1], a new contract may be determined. In that case, the new contract may affect the termination time of the currently active module in addition to affecting the run-time of future modules.

Note that in domains with predictable utility, the only factor that may affect the execution time of a module is the actual qualities produced by *previous* modules and the module itself. But, once a module terminates, there is no need to consider reactivating it with larger time allocation. This fact simplifies the scheduling of anytime computation in such domains.

6.2.3 A monitoring example

The following example demonstrates how the two principles of active monitoring, re-allocating residual time and adjusting contract time, can be used in practice. Suppose that a speech recognition system is composed of two anytime modules. The first module classifies the speaker and the second module uses the classification in order to set up the initial parameters of the recognition phase. The better the classification, the faster the recognition system converges on the correct sequence of words. The system's top level can be represented by the following expression:

RECOGNIZE(*Utterance*, CLASS(*Utterance*))

The system must operate in real-time with only a small constant delay allowed between the utterance and its computed word representation. The overall utility is determined by the delay in computing the representation of an utterance and the probability that the representation is correct.

A standard compilation technique looks at the complete system and determines for each total allocation the allocation to each component that maximizes the overall expected utility. If passive monitoring is used, an optimal fixed contract is determined by the monitor and the components are activated accordingly. No further scheduling decisions take place.

With active monitoring, time is only allocated to the first component, the classifier. When it terminates, the recognition step does not receive the remaining contract time. Instead, it is viewed by the

---

[1] The performance profile may require some dynamic adjustments as well. Such adjustments are discussed later in this chapter.

monitor as a residual sub-system whose inputs include both the utterance and the speaker's classification. Each input now has a particular quality that the monitor can use in order to derive a new contract time. Since the conditional performance profile of the recognition phase depends on the quality of its inputs, the new information regarding the actual quality of the classification can result in either an increase or decrease in the time allocation to that module. The actual effect is determined by calculating the new optimal contract time for the residual system.

To summarize, when the performance profile of a system has a wide range of possible qualities, a performance improvement can be achieved by active monitoring. Two techniques of active monitoring of contract algorithms were introduced: re-allocating residual time and adjusting contract time. To implement these techniques, the monitor must calculate a new contract time before activating each component. In other words, the process that is normally performed once in the fixed-contract case must be repeated. For this purpose, the monitor needs a set of performance profiles, one for each residual sub-system. These performance profiles are derived by standard compilation of each residual sub-system according to a pre-determined evaluation order.

6.3 Monitoring interruptible algorithms

Now I turn to the problem of monitoring interruptible anytime computation. The use of interruptible algorithms is necessary in domains whose utility function is not predictable and cannot be approximated by a predictable utility function. Such domains are characterized by non-deterministic rapid change. Medical diagnosis in an intensive care unit, trading in the stock exchange market, and vehicle control on a highway are examples of such domains. Many possible events can change the state of such domains and the timing of their occurrence is essentially unpredictable. Consequently, accurate projection into the far future is very limited and the previous fixed-contract approach fails. Such domains require interruptible decision making. I start this section with a look at the construction of interruptible systems.

6.3.1 Interruptible anytime systems

The question of generating interruptible anytime systems must be addressed before I can proceed to the monitoring part. Obviously, elementary anytime algorithms that are interruptible can be constructed using the programming techniques described in Chapter 4. However, when non-elementary modules are considered, one needs to apply one of the following two approaches:

1. Generate a contract algorithm first and use the construction of Theorem 4.1. The compilation of contract algorithms, a much easier task, has been extensively analyzed in the previous chapter. By compiling the main decision procedure as a contract algorithm and then running it with exponentially increasing time limits (as described in the proof of Theorem 4.1), one can create an interruptible system. This general approach suffers from a constant slowdown because of the reduction from contract to interruptible algorithm.

2. Use special compilation methods that directly result in an interruptible system. Direct compilation of interruptible algorithms is only possible with certain types of programming constructs. In the general case of functional composition, or even in the case of linear composition, interruptible algorithms require that the elementary components be activated and interrupted many times. This is due to the fact that only one module, the root of the DAG, produces the output of the system. If that module is not activated early on and the system is interrupted, no result is available. If it is activated early

on and the system is not interrupted, then execution must return to earlier modules to produce higher quality. When repeated activation of modules is necessary, the contract-to-interruptible reduction seems to be a good implementation. However, in some particular cases, better performance can be achieved by re-using results generated by previous contracts. In such cases, direct compilation might be beneficial.

Direct compilation of interruptible algorithms

A number of programming structures allow the use of special compilation methods that yield an interruptible algorithm directly. Here is a brief summary of these structures.

1. Pipelining of results in composite expressions.

   Some interruptible anytime algorithms, such as image processing algorithms, accept large data objects as input. The quality of the output may correspond to the percentage of input data that has been *completely* processed. When a composition of such algorithms is given, it is easy to construct an interruptible algorithm by simply allocating time to the components in an incremental way, using pipelines to transfer partial output of one module to its consumers. If the original performance profiles are $Q_1(t), Q_2(t), ..., Q_n(t)$, then the compiled performance profile is determined by the equation:

   $$Q(t) = Q_1(t_1) = Q_2(t_2) = ... = Q_n(t_n) \quad | \quad t = t_1 + t_2 + ... + t_n \tag{6.7}$$

   Matching the output qualities guarantees optimal balancing of input/output sizes. The equation can be solved efficiently by a binary search for the value of $Q(t)$ when the performance profiles of the elementary components are not pathological

2. Compilation of a set of interruptible independent jobs.

   A set of interruptible independent jobs is a set of interruptible jobs for which the quality of each job depends on time allocation only. The overall utility is the sum of the qualities of all the jobs. An interruptible system that executes these jobs can be constructed by scheduling the components according to the derivatives of their performance profiles. The algorithm with the steepest performance gain should be selected for execution until the derivative of its performance profile falls below the derivative of another task. The resulting performance profile and the corresponding schedule can be calculated by an off-line compilation process. This performance profile is optimal when $\partial Q_i(t) \partial t$ is a monotone non-increasing function.

3. Compilation of production systems.

   Russell and Subramanian [1993] studied the compilation problem of a special case of production systems in which production rules are matched in a fixed sequence. Each rule has associated with it a match time and a quality which corresponds to the utility of the rule's recommended action when the match succeeds. This architecture is similar to the compilation of a **one-of** structure where each one of the individual methods is a conditional statement. Russell and Subramanian analyze both fixed and stochastic deadlines. For the latter case, they provide a dynamic programming algorithm for obtaining an optimal sequence of decision rules and thus produce the best interruptible algorithm for any given stochastic deadline distribution.

4. Compilation of loops as interruptible algorithms.

> Certain types of iterative structures, in which each iteration gradually improves the quality of the result, offer a simple basis for the construction of interruptible algorithms. If the body of the loop requires a constant time, then the construction of the interruptible algorithm is trivial. Otherwise, the time allocation to each iteration has to be determined based on its conditional performance profile and the stochastic deadline distribution.

To summarize, in several special cases one can construct interruptible algorithms using direct compilation methods. In all other cases, the reduction theorem can be used to construct interruptible algorithms from contract ones. Both direct compilation techniques and the contract-to-interruptible reduction require some type of monitoring to construct the interruptible algorithm. The type of active monitoring discussed below is applied *after* the basic interruptible algorithm is constructed.

6.3.2 Active monitoring using the value of computation

Consider a system whose main decision component is an interruptible anytime algorithm, $\mathcal{A}$. The conditional probabilistic performance profile of the algorithm is $Q_{\mathcal{A}}(q,t)$ where q is the input quality and t is the time allocation. As before, $Q_{\mathcal{A}}(q,t)$ is a probability distribution and $Q_{\mathcal{A}}(q,t)[q_i]$ denotes the probability of output quality q_i.

Let S be the current state of the domain. Let S_t be the state of the domain at time t. And, let q_t represent the quality of the result of the interruptible anytime algorithm at time t. $U_{\mathcal{A}}(S,t,q)$ represents the utility of a result of quality q in state S at time t. The purpose of the monitor is to maximize the expected utility by interrupting the main decision procedure at the "right" time. Due to the high level of uncertainty in rapidly changing domains, the monitor must constantly assess the value of continued computation by calculating the net expected gain from continued computation given the current best results and the current state of the domain. This is done in the following way:

Due to the uncertainty concerning the quality of the result of the algorithm, the expected utility of the result in a given future state S_t at some future time t is represented by:

$$U'_{\mathcal{A}}(S_t,t) = \sum_i Q_{\mathcal{A}}(q,t)[q_i] U_{\mathcal{A}}(S_t,t,q_i) \tag{6.8}$$

The probability distribution of future output quality is provided by the performance profile of the algorithm. Due to the uncertainty concerning the future state of the domain, the expected utility of the results at some future time t is represented by:

$$U''_{\mathcal{A}}(t) = \sum_S p(S_t = S) U'_{\mathcal{A}}(S,t) \tag{6.9}$$

The probability distribution of the future state of the domain is provided by the model of the environment.

Finally, the condition for continuing the computation at time t for an additional Δt time units is therefore $VOC > 0$ where:

$$VOC = U''_{\mathcal{A}}(t+\Delta t) - U''_{\mathcal{A}}(t) \tag{6.10}$$

Similar to monitoring of contract algorithms, monitoring of interruptible systems can be simplified when it is possible to separate the value of the results from the time used to generate them. In such cases, one can express the comprehensive utility function, $U_{\mathcal{A}}(S,t,q)$, as the difference between two functions:

$$U_{\mathcal{A}}(S_t,t,q) = V_{\mathcal{A}}(S,q) - Cost([t_c,t]) \tag{6.11}$$

where $V_{\mathcal{A}}(S,q)$ is the intrinsic utility function, S is the current state, t_c is the current time, and $Cost([t_c,t])$ is the cost of the time interval $[t_c,t]$. Under this separability assumption, the intrinsic value of allocating a certain amount of time t to the interruptible system (resulting in domain state S) is:

$$V'_{\mathcal{A}}(S,t) = \sum_i Q_{\mathcal{A}}(q,t)[q_i]V_{\mathcal{A}}(S,q_i) \tag{6.12}$$

Hence, the intrinsic value of allocating a certain time t in the current state is:

$$V''_{\mathcal{A}}(t) = \sum_S p(S_t = S)V'_{\mathcal{A}}(S,t) \tag{6.13}$$

And the condition for continuing the computation at time t for an additional Δt time units is again $VOC > 0$ where:

$$VOC = V''_{\mathcal{A}}(t+\Delta t) - V''_{\mathcal{A}}(t) - Cost([t,t+\Delta t]) \tag{6.14}$$

Discussion

A control mechanism has been developed for interruptible algorithms that is based on the estimation of the current value of computation. Note that the condition for termination is “temporally local” in the sense that the value of computation might be negative for a small amount of computation time, Δt, and yet larger computation time might have positive net value. In highly unpredictable domains, it might be justified to make time allocation decisions based only on predictions of the immediate future. But in more stable situations, the monitor should consider several values of Δt before terminating the computation. The following theorem asserts the optimality of the above monitoring policy under certain assumptions about the intrinsic value and time cost functions.

Theorem 6.6 Optimality of monitoring of interruptible algorithms. *Monitoring interruptible algorithms using the value of computation criterion is optimal when* $\Delta t \rightarrow 0$ *and when the intrinsic value function is monotonically increasing and concave down and the time cost function is monotonically increasing and concave up.*

Proof: A function q is called *concave up* on a given interval I if it is continuous, piecewise differentiable, and $\forall x,y \in I$ for which $q'(x)$ and $q'(y)$ exist, $(x < y) \Rightarrow (q'(x) \leq q'(y))$. It is called *concave down* if $\forall x,y \in I$ for which $q'(x)$ and $q'(y)$ exist, $(x < y) \Rightarrow (q'(x) \geq q'(y))$. Note that the assumption of monotonically increasing and concave down intrinsic value function is identical to the assumption of Dean and Wellman[2] that performance profiles have the property of *diminishing returns*.

Now, suppose that the current time is t_1 and that

$$VOC = V''_{\mathcal{A}}(t_1+\Delta t) - V''_{\mathcal{A}}(t_1) - Cost([t_1,t_1+\Delta t]) \leq 0 \tag{6.15}$$

Since the intrinsic value function is concave down, it is guaranteed that for any future time $t_2 > t_1$:

$$V''_{\mathcal{A}}(t_2+\Delta t) - V''_{\mathcal{A}}(t_2) \leq V''_{\mathcal{A}}(t_1+\Delta t) - V''_{\mathcal{A}}(t_1) \tag{6.16}$$

---

[2] See [Dean and Wellman, 1991], Chapter 8, page 364

Since the time cost function is concave up, it is guaranteed that for any future time $t_2 > t_1$:

$$Cost([t_2, t_2 + \Delta t]) \geq Cost([t_1, t_1 + \Delta t]) \tag{6.17}$$

Hence, it is guaranteed that for any future time t_2:

$$VOC = V''_{\mathcal{A}}(t_2 + \Delta t) - V''_{\mathcal{A}}(t_2) - Cost([t_2, t_2 + \Delta t]) \leq 0 \tag{6.18}$$

And therefore termination at the current time is an optimal decision. □

How realistic are the assumptions of a concave down intrinsic value and a concave up time-cost? I suggest that these assumptions are valid in many real domains because of the nature of anytime computation. First, it is quite normal for a system to have more significant gains in quality in the beginning of the computation with gradually decreasing improvement rate towards its completion time. Second, the cost of time typically grows at a slow rate at the beginning of the computation and at a faster rate as the time approaches the “hard deadline” of the application.

6.3.3 Dynamic adjustment of performance profiles

A single performance profile has been used until now in the analysis of monitoring interruptible algorithms. This performance profile specified for any future time t a fixed quality distribution, regardless of the history of results already generated by the algorithm. Obviously, the history of results generated so far may provide strong evidence regarding future results. Therefore, a more informative characterization of the future performance of an interruptible algorithm may be captured by the following mapping:

$$\mathcal{CPP}^* : Q_{in} \times Q^* \times T \rightarrow Pr(Q_{out}) \tag{6.19}$$

where Q_{in} is a measure of the input quality, Q^* is a sequence of qualities of the results produced so far, and T is the remaining execution time.

Although the above mapping is a more informative representation, it is unrealistic since it requires that all the past qualities be stored and taken into account. Fortunately, the dependency of future results on the results generated so far can be captured in a far more efficient way in many domains. In others, the problem may not be relevant at all since the actual quality of previous results cannot be measured directly. The following list summarizes four typical situations:

1. *The actual quality of the results is unknown.*

   In this case Q^* is simply unavailable and one must rely on a standard performance profile. For example, consider an interruptible randomized algorithm for primality testing, or Monte Carlo algorithms in general. By the nature of such algorithms the quality of results is a function of the number of iterations (or time) and cannot be adjusted since the answer produced by a single run of the algorithm does not have a measurable objective quality.

2. *The actual quality of the results is known but has no effect on the quality of future results.*

   In this case Q^* is irrelevant and a standard performance profile may be used. For example, consider an interruptible algorithm whose implementation is based on activating several *independent* methods. The methods are ordered according to their expected quality and time requirements. The *actual* quality of a method, as much as it may deviate from the expected value for that method, has no influence on the future quality of successive methods. It should be noted that while the expected performance of future methods remains the same, the best result generated rather than the last one can be returned by such algorithm. Hence, the dynamic performance profile at each point is composed of the maximum of the quality of the best result so far and the original performance profile. This minor correction does not require that Q^* be saved.

3. *The quality of previous results has some effect on the quality of future results, but the dependency is on the best result only.*

   In this case, Q^* is not needed. Instead, a single number representing the best result so far has to be stored. Therefore, the representation of the performance profile is not radically more complicated. For example, consider iterative approximation methods (e.g. Newton's method). In such methods, the quality is a reflection of the error or distance from the correct result. The error reduction in each iteration is bounded by an expression that depends on the previous error bound. This case has two interesting sub-cases:

   (a) When homogeneous algorithms are used, that is, algorithms whose input and output have the same representation, a standard conditional performance profile is sufficient to capture the dependence on the quality of intermediate results. The correction of the performance profile can be implemented by a horizontal shift of the performance profile each time a new result is generated. For example, consider the traveling salesman algorithm of Chapter 4 where the problem is a random path and a solution is an improved, shorter path represented in the same way.

   (b) When non-homogeneous algorithms are used, it is still possible that the dependency on the best result so far can be captured by a simple shift of the origin of the performance profile to the point $(t, Q(t))$ where $Q(t)$ is the quality of the best result so far.

4. *The quality of previous results offers some evidence regarding the quality of future results. The dependency can be summarized by a function of the slope of the actual performance profile.*

   The slope of the actual performance profile can be computed based on the sequence of qualities. It does not require that the entire sequence be saved. For example, consider a speech recognition program whose overall quality relates to the probability of correct representation of an utterance as a function of computation time. Obviously, the slower the speaker, the less data the system needs to process per time unit. Hence, performance is improved with slow speakers. The effect of the speaker on the performance profile is linear and can be determined based on the actual performance of the system over previous utterances of the same speaker.

In addition to the above situations, there is a class of algorithms for which the quality of previous results has some effect on the quality of future results, but the dependency can be ignored since the deviation from the expected performance is negligible. In other words, the performance of the algorithm is strongly dependent on time allocation with small deviation. In such cases, the quality of future results is highly predictable based on a regular performance profile and no adjustment is necessary.

6.3.4 A monitoring example

Suppose that a robot is performing automatic diagnosis and repair of machines in a factory. Once a machine is malfunctioning, it is shut down automatically and reported to the robot technician. The robot then performs hierarchical diagnosis to determine the defective component of the machine. The smallest components of the machine that can be identified as defective by the robot are called the elementary components.

When an elementary component is identified as defective, the robot simply replaces that component. The robot can also replace larger assemblies that include defective components. The time and cost of replacing a part are known in advance and depend on the defective part only. The machine has a hierarchical structure in which each component has several sub-components. The diagnosis algorithm can be interrupted at any time to yield the identification of the sub-system that contains the defective part. The more time that is available for diagnosis, the more specific the diagnosis will be.

Each machine has an associated function that determines the loss of production as a result of delay in repair. The function may be non-linear since there may be complex dependencies among various machines in the factory. In addition, the function depends on the current operational needs of the factory which change constantly. As a result, the robot technician operates under rapidly changing time pressure in a domain with unpredictable utility. Therefore, interruptible algorithms must be used.

With active monitoring, time is allocated to the diagnosis module based on the value of computation criterion. At each point of time, the most specific defective sub-component identified so far, S_i, and its replacement time and cost are known to the monitor. The monitor needs to determine whether to allow continued deliberation by calculating the value of computation. The value of computation in this case is the difference between the expected saving in repair costs due to a more specific diagnosis and the cost of further delay in making the machine operational.

Comparison to contract algorithms

The above example could be handled also using a contract algorithm. The greater efficiency of active monitoring of interruptible algorithms becomes significant in unpredictable situations with great variability in algorithm performance and in the time cost. With respect to the above example, while the level of specificity of a diagnosis may be a relatively stable function of time, the more relevant aspect is the *actual* cost of replacing the defective part. This cost can vary over a large range of values. For example, a memory component may be much cheaper than a processing component, even if both are characterized by similar specificity levels. In addition, this domain is characterized by great variability in time cost. Hence, by constantly re-evaluating the value of computation based on the *actual* defective part and the *current* time cost, the performance of the robot is better optimized.

6.4 Summary

The monitoring problem has been examined in two types of domains. One type is characterized by the predictability of utility change over time. High predictability of utility allows an efficient use of contract algorithms modified by various strategies for contract adjustment. The second type of domain is characterized by rapid change and a high level of uncertainty. In such domains, monitoring must be based on the use of interruptible algorithms and the value of computation criterion. But, how should one handle domains with moderate change and some degree of predictability of future utility? The monitoring approach that is based on contract algorithms does not generalize beyond predictable domains. Fortunately, the use of interruptible algorithms is general enough to cover all types of domains. The only problem with interruptible algorithms is the performance degradation as a result of converting contract algorithms into interruptible ones. This performance degradation can be minimized by scheduling contract algorithms on a parallel machine, even one with a small number of processors. Another approach is to try to integrate the two monitoring schemes and to use prior information about the distribution of stochastic deadlines in the particular problem domain.

Chapter 7

Anytime Sensing and Anytime Action

> I have striven not to laugh at human actions, not to weep at them, nor to hate them, but to understand them.
>
> Baruch Spinoza, *Tractatus Politicus*

In this chapter I return to the fundamental question posed in the introduction: how can an artificial agent react to a situation after performing the right amount of thinking? In the previous chapters I developed a formal definition of the notion of the "right amount of thinking" in terms of the value of computation. I showed how anytime algorithms, together with an appropriate compilation and monitoring scheme, can be used to optimize the agent's decision procedure. However, until now the decision making component of the agent was studied in isolation. In practice, two additional processes define the quality of the behavior of an agent, namely sensing and action. Sensing involves gathering information about the state of the domain and action involves applying the results of computation to the domain and changing its state in order to achieve a set of goals. In this chapter, I will extend the notion of gradual improvement to sensing and action and will show how to integrate them into the model of operational rationality.

7.1 Beyond episodic problem solving

The main reason for incorporating sensing and action into the model is in order to extend its scope beyond episodic problem solving. To achieve this goal, the principles of operational rationality must be extended to larger fragments of the life time of an agent. But, as one tries to solve the utility optimization problem over larger segments of time, or histories, once must address additional aspects of agent construction, namely sensing and action. To understand the effects of sensing and action on the control of deliberation, it is useful to distinguish between *purely computational agents* and other, more general agents.

7.1.1 The purely computational agent

A purely computational agent is an agent that performs a certain computational service (i.e. solving a particular problem) and whose utility function is defined directly in terms of the time-dependent quality of that service. For example, the kernel of an operating system, a standard compiler, and a automatic text-translation system are all purely computational agents. A purely computational agent must present its results by writing them on a certain output device and, in a sense, this constitutes an action rather than a computation. Yet, it is useful to look at these agents as purely computational in a sense that their task terminates when they output the results. With purely computational agents, operational rationality can be achieved without knowing exactly how the results are used and in what ways they affect the world. The utility function relates to the results themselves, not to the results in the context of a particular state of a certain domain. For this kind of agents, compilation and monitoring, as they were described so far, are sufficient even when a large time segment is considered. A purely computational agent that operates on a sequence of independent problems can be simply monitored as a type of loop as discussed earlier.

7.1.2 Computation, perception and action

The construction of a general artificial agent requires a model that extends beyond the purely computational agent boundary. A robot that performs a certain task in a given environment, such as package delivery in an office building, cannot be analyzed based on its planning capability or decision quality alone. The capability of the robot to carry out its plans is as important as its planning capability. The degree of task achievement is the ultimate measure of performance, not the quality of "thinking" (although there may be a strong correlation between the two). The additional aspects of the robot's behavior, sensing and action, affect its operational rationality. The rest of this chapter gradually expands the model of operational rationality to include certain types of sensing and action. The complete analysis of perception and action is hard and requires the solution of several open questions. This dissertation does not include complete answers to all the questions, but it discusses the problems and outlines some directions toward their solutions.

7.2 Anytime sensing

Sensing means that some information about the current state of the domain of operation is gathered and used by the agent after its initial activation point. I start with the analysis of anytime sensing since it appears to be much closer than action to anytime computation. Sensing is essential in agent construction for three primary reasons:

1. Change in the environment that may affect the immediate goal, the time pressure and the desirability of continued computation.
2. Uncertainty about the actual state of the environment due to past sensory measurement errors.
3. Uncertainty about the actual state of the environment due to inexact environment modeling.

The notion of anytime sensing is a natural extension of traditional sensing. The quality of a sensing procedure can be measured in a similar way to the quality of computation: in terms of certainty regarding the domain description that it produces, in terms of accuracy of the domain description, or in terms of level of specificity. Gradual improvement in sensing quality can be achieved by varying the amount of data collected about the domain or by using anytime algorithms to extract the domain description from the raw data. The amount of sampled data can be controlled by varying the sampling resolution or by varying the number of sampled features. For example, a vision module can produce a varying quality description of a scene by changing the resolution of the gray level samples, by selecting a certain set of features to construct the intrinsic image (i.e. illumination and depth but not reflection or orientation), or by applying a certain set of analysis methods (i.e. visual line analysis and texture analysis but not motion analysis or stereo disparity analysis). Hence, it seems that elementary anytime sensing algorithms can be constructed in a similar way to the construction of elementary anytime algorithms. In this section I will examine the effect of integrating anytime sensing into the model of operational rationality and will show several methods of controlling the tradeoff offered by anytime sensing.

7.2.1 Sensing versus computation

To some extent, sensing is similar to computation and can be similarly compiled and controlled. Both computation and sensing can be viewed as information gathering activities. The former provides information based on manipulating knowledge already available to the agent, while the latter provides information based on activation of sensory devices. The performance profiles of both activities are similar in nature. They both describe the probability distribution of quality of the results as a function of time – computation time in the first and sensing time in the second. Sensing time is the total amount of time consumed by the sensing device including the computation time needed to translate the raw data into a more meaningful representation of the state of the domain.

Despite this fundamental similarity, there are several significant differences between computation and sensing. First, the results of a computation have a fixed objective quality which is a measure of the distance between the approximate result and the exact result. This quality remains the same as time passes. The question of whether the results of a computation are used immediately or not does not affect their objective quality since it is defined with respect to a static problem description. In that respect sensing quality is different since it is defined with respect to a potentially dynamic environment that it describes. Sensing provides information whose accuracy in describing the current state of the domain depends on sensing time as well as the time the results are used. Since the state of the domain may change, the validity and quality of sensory information deteriorates over time. For example, a map of objects in front of a moving car becomes irrelevant a few seconds later as the car passes these objects or hits them. It is true that computation may suffer as well from a similar problem. The quality of a plan for achieving a certain goal may deteriorate as computation time increases, since the facts on which the planning process is based may become invalid. But the quality of planning can be measured with respect to a *given* domain description and the deterioration of its utility can be described by a different component of the model. This separation cannot be applied to sensing since sensing quality is *always* with respect to the current state of the domain.

Another important difference between sensing and computation is the fact that computations do not change the environment. Sensing sometimes does. Ideally, sensing devices could be used for information gathering only. However, some types of sensors interact with the environment and change the environment, possibly in an undesirable way. For example, in medical diagnosis, some tests may affect the condition of the patient or even endanger the patient's life.

To summarize, while sensing may seem very similar to computation, its inherent quality deterioration and its possible effect on the state of the domain make it harder to analyze and control. The rest of this section describes several approaches to sensing and their monitoring schemes.

7.2.2 Passive sensing in semi-static environments

The analysis and control of anytime sensing can be simplified by making two assumptions that eliminate the differences between sensing and computation. The first assumption is that sensing is passive, that is, sensing has no effect on the state of the domain. For example, visual sensors, such as sonars, have no significant effect on the state of the domain. The second assumption is that the environment is semi-static, that is, during a single problem solving episode the state of the environment can be considered static. Under these two assumptions sensing can be treated as computation and incorporated into the model of operational rationality. The compilation and monitoring techniques that were introduced in the previous chapters can handle this type of anytime sensing.

In Chapter 8, I will describe an application of the model in which an anytime vision module is used as part of a mobile robot navigation system. Since the vision module and the environment satisfy the two assumptions above, standard compilation is applied in order to combine the vision module with an anytime abstract planning module.

7.2.3 Sensing as an independent process

Another approach to sensing is to isolate the process and to assume that it is performed independently and in parallel to the computational decision component. This approach is justified by the fact that sensing normally uses a specialized type of hardware, such as a camera, and a specialized type of computation elements, such as signal processors or neural networks. As a result, it may not be realistic to expect any kind of resource sharing between sensing and computation. Once such resource sharing is excluded, operational rationality can be achieved assuming an independent sensing process whose only goal is to maintain a current description of the state of the environment. This is in fact the assumption used in the analysis of monitoring in the previous chapter, where time is allocated only to computation. To control the computational task, the monitor may use the description of the *current* state of the domain.

Suppose that the state of the environment is used whenever the main decision procedure is activated with a certain contract time. The conditional performance profile of the main decision procedure is represented by $Q_{out}(q_{sensing}, t)$ where $q_{sensing}$ is the quality of the sensing process and t is the time allocation. Suppose also that the performance profile of the sensing process is represented by $Q_{sensing}(t)$. Then, when sensing is performed in parallel, the unconditional performance profile of the decision procedure is simply:

$$Q'_{out}(t) = Q_{out}(Q_{sensing}(t), t) \tag{7.1}$$

The control of the anytime sensing process is rather simple. Based on the desired contract time for the next cycle of the decision procedure, the sensing monitor has to allocate time to the sensing module.

The assumption of an independent sensing process is useful as long as the sensing component does not require any guidance, for example, when the sensor consists of a camera that overlooks the entire domain from above and produces a complete domain description in each cycle. However, if a sensor consists of a camera mounted on a mobile robot with a limited field of view, sensing cannot remain an independent process since the camera needs to be aimed at a particular direction based on the immediate plan of the robot. Since the information gathered by such sensors is limited to a certain section of the domain, they must be coordinated so that the sensory input covers the most relevant section of the domain. Such sensing activity is discussed in the following section.

7.2.4 Sensing as an information gathering action

The most general view of sensing is as a type of action whose primary goal is information gathering but whose potential effect on the domain must be analyzed in a similar way to the analysis of actions. Optimization of the agent's behavior requires intelligent use of the sensing devices so that the information gathered has higher value and relevance to the situation. For example, a mobile robot operating in an office environment may need to move some boxes just in order to determine whether a certain package is located behind these boxes. As a result, there may be a complex relationship, and sometimes interference, between standard actions and sensing actions. So, rather than being an independent component, general sensing must be analyzed as an integral part of the planning problem. This type of sensing is much closer to action and will thus be analyzed in the following section.

To summarize, the fundamental goal of sensing processes is to gather information, much like in the case of computational processes. In fact, in some situations anytime sensing modules can be controlled just as any computational modules. However, more general sensing must be viewed as a type of action whose control is more complicated.

7.3 Anytime action

Action is much more difficult than computation or perception when analyzed as an anytime process. While computation and perception has no intentional effect on the state of the domain, actions are designed to transform the state of the domain. While useless computation paths may be simply abandoned, useless actions may be destructive and sometimes irreversible. As a result, some anytime programming methods and monitoring techniques cannot apply directly to anytime action. The contract-to-interruptible conversion, for example, cannot apply to actions since re-initiating an action may result in different effects because of the changed initial state. As an example of this difficulty, consider gem polishing with increasingly fine abrasives. Once a fine abrasive is used, it is normally undesired to return to a coarse abrasive even if more time is available for the action. As a result, the extension of the principle of operational rationality to action requires significant modification of the model. The purpose of this section is to introduce the notion of anytime action and show how it could be integrated into the model of operational rationality.

7.3.1 Elementary anytime actions

The quality of an action is typically defined by the expected degree of goal achievement. The degree of goal achievement can be measured, just as in the case of anytime computation, in terms of certainty, accuracy or specificity. Elementary anytime actions can be constructed using several standard paradigms:

1. *A loop of corrective actions*

   Many types of actions can be implemented as a sequence of corrective actions that are designed to achieve a certain goal. Consider the class of actions in which the accuracy of goal achievement relates to the accuracy of positioning a certain physical object. Typically, the error in the position is (bounded by) a function of the movement size and speed. For example, the angular error in a camera rotation may depend on the angle of rotation and on the speed of rotation. In such cases, it is common to reduce error by a series of corrective actions, each composed of a smaller movement size at a slower speed. As a result, the position error is constantly reduced. This process is repeated until a certain minimal error is reached or until the process is interrupted.

2. *A loop of refinement actions*

   Similar to the previous case, gradually improving quality can be achieved in actions by repeatedly applying the same action with more refined setup to increase the accuracy of the outcome. The gem polishing example of the previous section is a good example. In gem polishing, it is common to repeat the polishing procedure with increasingly fine abrasives. In such cases, the quality of the action is a direct function of the refinement level. By limiting each phase to a fixed amount of time, a fixed performance profile can be easily derived.

3. *A sequence of low-level actions*

   Actions are sometimes implemented by a sequence of low-level steps or motor movements, and the degree of goal achievement depends on the completion of all the necessary phases. Gradual improvement over time can be achieved by performing a subset of the low-level steps. For example, when a robot needs to get closer to an object in order to better identify it, the motion toward the object can be interrupted at any time and the remaining distance can be used to calculate the quality of the action. Another example is the initialization process of an autonomous vehicle. Certain steps, such as self-testing and instrument calibration, may be omitted under time preassure, allowing the construction of an anytime action.

4. *Multiple methods*

   As in the case of computation, anytime actions can be sometimes implemented using a set of alternative methods that offer a different execution time and performance constraints. For example, in an intensive care unit, several different methods can be normally used to stablize the condition of the patient. A life endangering, but fast procedure may be preferred under extreme time presure. An important difference between action and computation is that in the case of action, once a particular method is applied, it may restrict the future application of alternative methods. For example, the prescription of a certain drug to a patient normally restricts the application other possible treatments, because the combination of the associated drugs may have a dangerous effect.

To summarize, four general methods to construct anytime actions were described. Elementary anytime actions tend to be interruptible. A loop of corrective actions and a loop of refinement actions are always interruptible, but a sequence of low-level actions or a set of alternative methods may not be interruptible. For example, if gradual improvement is achieved by skipping non-critical low-level actions without being able to return to steps that were omitted, then the outcome may be a contract algorithm that cannot be converted to an interruptible one.

7.3.2 Performance profiles of actions

Performance profiles of elementary anytime actions describe the probability distribution of quality as a function of time. The notion of conditional performance profile is as useful as with anytime computation and is defined in the same way. The construction of performance profiles for anytime actions, however, appears to be more complicated for several reasons:

1. Gathering statistics, when necessary, is more difficult. While the performance of an algorithm can be observed by activating it with many input instances, actions need to be performed in a certain domain. To learn the performance profile of an action, it must be performed many times in a real or simulated domain. As it seems unlikely that the real domain would be used for this purpose, the construction of the performance profile requires the development of a simulated environment in addition to the implementation of the procedure that performs the action itself.

2. Objective quality measures are hard to identify. While accuracy in computation is directly related to deviation from the correct answer, it is harder to measure the accuracy of action. The main reason is the fact that the quality of an action is related to the degree of goal achievement and hence the same action can have different qualities with respect to different goals. While this problem may arise in computation as well, it is not as common since a computation has a well defined goal (of solving

particular problem). But a small set of actions is normally used in order to achieve a large set of possible goals. For example, when an agent moves toward a target, its goal may be to get a better view of the target or to grasp the target. The quality of the action is different depending on the actual goal. One way to deal with this difficulty is to use a different quality measures for different goals.

3. Actions in many cases can be performed at a variable speed, where higher speed may reduce the time segment of the action but it may also increase the energy consumption[1]. For example, when moving toward a target position, a mobile robot has to accelerate, move at a certain speed, and slow down. The acceleration rate and speed of movement at any point can be controlled to vary the timing and quality of goal achievement. In many cases the speed of execution and its effect on the performance profile are important and must be determined by the meta-level control. Therefore, a performance profile that is conditioned on the speed of execution must be used.

Standard conditional performance profiles can be used to characterize the behavior of anytime actions. But several unique aspects of actions make it more difficult to construct their performance profiles.

7.3.3 Buying time in real-time domains

An interesting aspect of action, that has no parallel in computation, is the capability to change the degree of time pressure or to "buy time." In many domains it is possible to perform a certain action whose main or only value is to buy time for further planning. For example, in dialogues, any action which keeps the other agents from speaking will give the agent more planning time. In an intensive care unit, temporary treatment that is intended to stabilize the condition of the patient is common prior to complete diagnosis. Similarly, in an air traffic control situation, placing some aircraft in a holding pattern to allow more time for safe scheduling is commonly used. So, in addition to global goal achievement, action can be used to modify the domain so as to reduce time pressure. The question is, how do this and other special aspects of action affect its monitoring? This question is addressed in the following section.

7.3.4 Controlling anytime action

Operational rationality deliberately reduces the meta-level control of an agent to a resource allocation problem. This goal is maintained as the framework is expanded to include sensing and action. In other words, the problem of action selection is considered a planning problem, not subject to the optimization process performed by the model of operational rationality. This optimization problem is only responsible to determine the amount of time that should be allocated to elementary anytime modules based on their performance profiles. So, when the possibility of using an action to buy time is considered, the purpose of the meta-level control is limited to allocating the optimal amount of time to this action but not to selecting the action over other possible actions. The meta-level control does *not* solve the planning problem of the agent.

How does the agent select actions? The most natural way to extend the model is by performing action selection in an analogous way to computation selection. That is, by explicit programming. According to this approach, each anytime action is actually implemented as a procedure embedded in the complete program that generates the agent's behavior. For example, consider the following high level program to control a robot that collects empty cans:

---

[1] Variable action speed is somewhat analogous to variable computation speed achieved by varying the computational resources used (e.g. the number of processors). However, parallelizing anytime algorithms is a non-trivial issue that has not been included in the model.

```
X ← LOCATE-OBJECT(Can)
if REACHABLE-OBJECT(X) then
    while not GRASPABLE-OBJECT(X) do
        MOVE-TOWARD-OBJECT(X)
    GRASP-OBJECT(X)
```

Under this approach, compilation is used separately on homogeneous program fragments, that is, on computation, on sensing, and on action. Each one of them is considered an instance of a *generalized action* and a candidate module for execution. Such generalized action may be an internal action in the form of a computation, an information gathering sensing action, or an external action. Each generalized actions may be an anytime module whose (possibly compiled) conditional performance profile is known to the monitor. A relatively simple monitoring scheme can be implemented by scheduling the execution of a single selected action at a time. For this purpose, the monitor needs to determine the *context of execution*, which is a set of aspects of the current state of the domain that affect the resource allocation to the selected generalized action. Then, execution is monitored following the episodic problem solving approach.

An extension of this monitoring strategy allows the generalized action to be selected by a meta-level planning component rather than being a component of a fixed program. The planner itself may be an anytime algorithm in which case a multi-level monitoring scheme may be required: one for allocating time to the planner based on its performance profile and the state of the domain and another for allocating time to the selected generalized action. These monitoring techniques optimize the allocation of time to a single generalized action at a time. The question of how to optimize the agent's behavior over a longer time segment remains open.

7.4 Operational rationality over history

The notion of bounded optimality or operational rationality over a long time segment, or history, is yet to be defined and solved. The ultimate goal is to extend the theoretical framework of operational rationality to allow building agents whose decisions are made using anytime computation, whose perception is based on anytime sensing, and whose interaction with the environment is implemented as anytime actions. This long term goal presents a number of difficulties. One difficulty with implementing operational rationality over histories is the estimation of the utility of learning and domain exploration. Learning may have negative effect on performance in the short run but, it has very high utility in the long run. Estimating the utility of learning is much harder than estimating the quality of results of a given algorithm. By their nature, learning and exploration lead to failures and false generalizations. It is hard to select performance metrics for learning activities, let alone characterize them quantitatively.

Another difficulty is due to the need to make predictions regarding the state of the domain far into the future. In the face of uncertainty regarding the current state of the domain and the performance of the system, it is hard to make such predictions. The exponential growth of the number of possible states makes such prediction even harder. Abstraction seems to be a possible mechanism to handle this difficulty. That is, by making predictions that correspond to a large set of possible states, rather than reasoning about individual states, one can reduce the complexity of the problem. I will return to these problems in the concluding chapter.

Chapter 8

Application and Evaluation

> Rationalism is an adventure in the clarification of thought.
>
> Alfred North Whitehead, *Process and Reality*

This chapter describes several applications of the model of anytime computation. In each application, the implementation of the key processes, compilation and monitoring, will be examined. Of special interest is the representation and learning of performance profiles and the use of conditional performance profiles. Section 8.1 describes my own experience in using the model to implement a navigation system for a mobile robot. In Section 8.2, I describe two additional applications that were developed by Anita Pos [1992, 1993] and by Coulon *et al.* [1992]. Finally, in Section 8.3, I evaluate the model based on the results of these experiments. The purpose of this evaluation is to examine the basic assumptions of the model and its applicability.

8.1 Path planning and navigation in robotic systems

In order to demonstrate the model of anytime computation, I have selected one of the fundamental problems facing any autonomous mobile robot: the capability to plan its own motion with noisy sensors. For this purpose, I have implemented a simulated environment and a system composed of anytime sensing and planning modules. Figure 8.1 shows the data flow between the main components of the navigation system. Sensory input is used to update the description of the environment. This description is used both as input to the anytime planner and as one of the factors that determine the allocation of time by the monitor. The other factors are the compiled performance profiles of sensing and planning, the model of the environment, and the quality of the current best plan.

I start with a description of the environment and the anytime sensing and planning modules. Then, I explain how the compilation process was used to optimally integrate the components of the navigation system. Finally, I describe the run-time monitor and experimental results.

8.1.1 The environment

A robot is situated in a simulated, two dimensional environment with random rectangular obstacles. The robot does not have an exact map of the environment, but it has a vision capability that allows it to create an approximate map. The accuracy of the domain description, produced by the vision mechanism, depends on

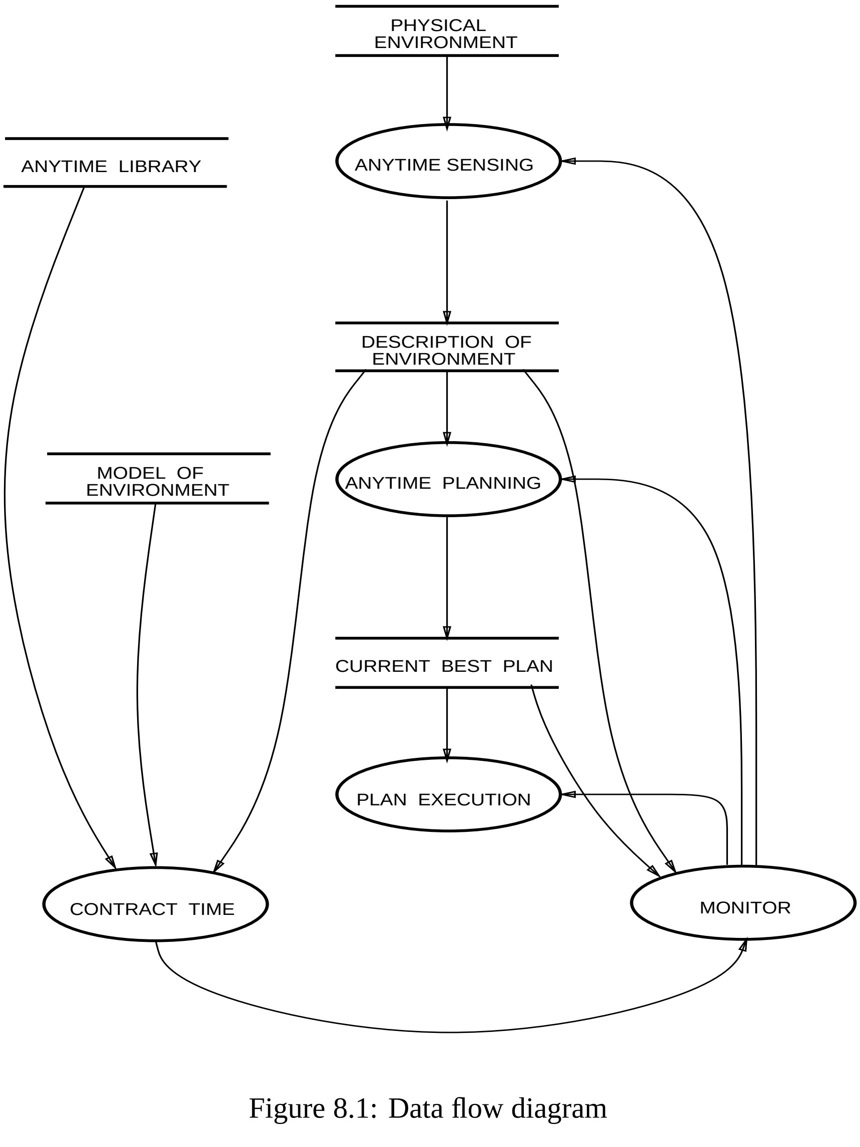


Figure 8.1: Data flow diagram

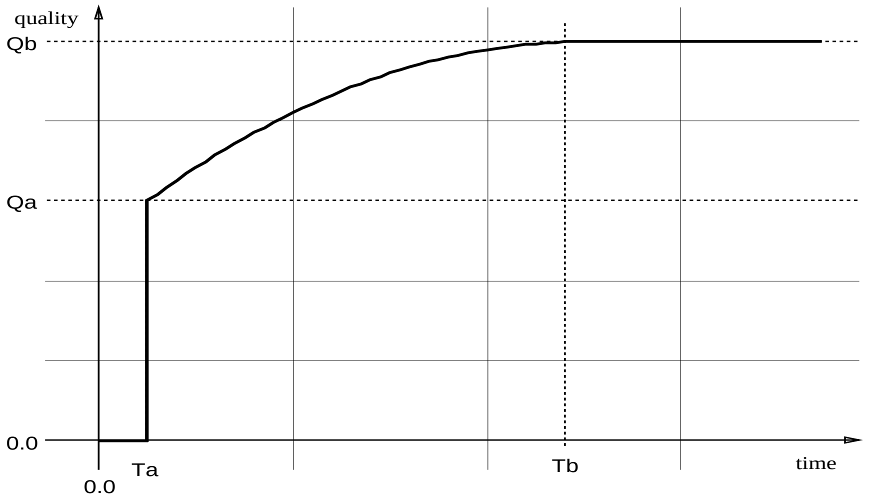


Figure 8.2: The performance profile of the vision module

the time allocated to the vision module. The environment is represented by a matrix of elementary positions. The robot can move between adjacent cells of the matrix at varying speeds which affect the execution time of the plan as well as the energy consumption. Since both sensing and planning are imprecise, it is possible that a plan would lead the robot through a position that is actually blocked by an obstacle. I assume that the robot has a capability to detect this situation using an alternative sensing capability (i.e. sonar) when it is close to the blocked position. In that case, the robot has to modify the plan at run-time in order to avoid hitting the obstacle.

When the simulation starts, the robot is presented with a certain task that requires it to move to a particular position and perform a specific job. Associated with each task is a reward function that determines the value of the task as a function of completion time. The system is designed to control the movement of the robot, that is, to determine its direction and speed at each point of time while maximizing the overall utility. The overall utility depends on the value of the task (a time dependent function) and on the amount of energy consumed in order to complete it.

To simplify the discussion, I assume first that the description of the environment is produced by a *global sensing module*, that is, the vision module has access to the complete environment (i.e. using a camera that is watching the domain from above). However, in a more realistic situation, sensing is limited to a small, local segment of the complete environment. That situation will be examined later, when the run-time system is described. At that point, the sensing and planning modules will be applied repeatedly to segments of a larger domain. The monitor has to determine at each point how much time to allocate to vision and path-planning based on factors such as the current location of the robot, the estimated distance to the goal position, the urgency of the task, and the quality of the plan produced so far.

8.1.2 Anytime sensing

A primary goal of this application has been to extend the notion of gradual improvement of quality to sensing. The supposition that sensors produce a perfect domain description, as much as the assumption of perfect planning and plan execution, constitutes a major disadvantage in any model for real-world robot control. In my model, the presence of sensory errors is not only acceptable, but is considered the normal situation. Moreover, in order to optimally control the quality of sensing, the model includes a quantitative evaluation of its effect on the quality and performance of the other components of the system.

This section describes the module of anytime sensing that I implemented. It produces a domain description whose quality expresses the probability that an elementary (base level) position is wrongly identified, that is, identified as free space while actually blocked by an obstacle or vice versa. It is assumed that within the area in which the sensors are effective, the quality of sensing is not affected by the robot's position. The general model, however, does not require this assumption.

Figure 8.2 shows the performance profile of the vision module. It is characterized by several parameters: T_a, T_b, Q_a, Q_b. T_a is the minimal amount of time needed for the sensor to produce an initial domain description with quality Q_a. Given a shorter run-time, the sensor does not produce any description of the domain. For a run-time t, $T_a \leq t \leq T_b$, the quality of vision improves from Q_a to the maximal quality Q_b, which is 1.00 in this example.

Notice that when a sensor is interrupted at any time shorter than T_a, it is still possible for the system to operate using prior knowledge. For example, it may assume that every position is free in a region with scarce obstacles.

8.1.3 Anytime abstract planning

Path planning is performed using a variant of the coarse-to-fine search algorithm [Lozano-Pérez and Brooks, 1984] that allows for unresolved path segments. In order to make this algorithm interruptible, a hierarchy of abstraction levels of the domain description is used. This allows the algorithm to find quickly a low quality plan and then repeatedly refine it by replanning a segment of the plan in more detail. The rest of this section describes the algorithm and its performance profile.

Abstract description of the domain

In a hierarchical (quad-trees) representation, the n^{th} level of abstraction corresponds to a certain coarse grid in which every position, (i, j), is an abstraction of a $2^n \times 2^n$ matrix of base-level positions. Each high level position has a certain degree of "obstacleness" associated with it which is simply the proportion of the matrix that is covered by obstacles.

A general position in this two dimensional domain has therefore three components: $(x \;\; y \;\; l)$: where x and y are the coordinates and l is the level of abstraction. The position $(3 \;\; 3 \;\; 1)$, for example, corresponds to the following set of base level positions: $(6 \;\; 6 \;\; 0)$ $(6 \;\; 7 \;\; 0)$ $(7 \;\; 6 \;\; 0)$ $(7 \;\; 7 \;\; 0)$. If one of these positions is blocked by an obstacle and the rest are free, then the "obstacleness" of $(3 \;\; 3 \;\; 1)$ is 0.25.

The anytime planning algorithm

The interruptible anytime planner (ATP) constructs a series of plans whose quality improves over time. It starts with a plan generated by performing best-first search at the highest level of abstraction. Then, it repeatedly refines the plan created so far by selecting the worst segment of the plan, dividing it into two segments (of identical length), and replacing each one of those segments by more detailed plans at a lower abstraction level. The worst segment of the plan is selected according to the degree to which the segment is blocked by obstacles and according to its abstraction level. A special data structure, called a multi-path, is used in order to keep intermediate results. It is a list of successive path segments of arbitrary abstraction level. The algorithm is shown in Figure 8.3.

```
ATP(start, goal, domain-description)
    multi-path ← [SEGMENTIZE(start),
        PATH-FINDER(PROJECT(start, $L_{max}$), PROJECT(goal, $L_{max}$), domain-description),
        SEGMENTIZE(goal)]
    REGISTER-RESULT(multi-path)
    while REFINABLE(multi-path) do
        REFINE(WORST-SEGMENT(multi-path), domain-description)
        REGISTER-RESULT(multi-path)
    SIGNAL(TERMINATION)
    HALT
```

Figure 8.3: The anytime planning algorithm

```
PATH-FINDER(start, goal, domain-description)
    level ← LEVEL-OF(start)
    size ← ACTUAL-DOMAIN-SIZE/$2^{level}$
    domain ← NTH-ABSTRACTION-LEVEL(level, domain-description)
    close ← UNVISITED(domain)
    open ← [MAKE-STATE(start)]
    best-path ← best-first-search(domain, open, close, goal)
    return [start | best-path]
```

Figure 8.4: The path finder

Note that *start* and *goal* are the start and goal positions, and L_{max} is the maximal abstraction level. The length of each segment of an intermediate plan is invariant. It depends only on the length of the initial path at the highest level of abstraction. As a result, the run-time of the refinement step is approximately the same for any segment of the plan regardless of its level of abstraction.

The PATH-FINDER is a search procedure that returns the best path between any two positions in the same abstraction level. The path is represented as a list of positions at the same abstraction level as the start and goal positions. A base-level path must be obstacle-free and hence, it is a route that the robot can follow. A path at a higher level of abstraction, on the other hand, is the result of a best-first search that minimizes the length as well as the obstacleness of the result. It does not correspond to a particular list of base-level positions that the robot can follow. The particular positions are determined at execution time. The algorithm is shown in Figure 8.4.

Plan execution

In order to follow an abstract path, the robot must use an obstacle avoidance procedure that controls its movement whenever the planned route is blocked. The robot can sense that the planned route is blocked

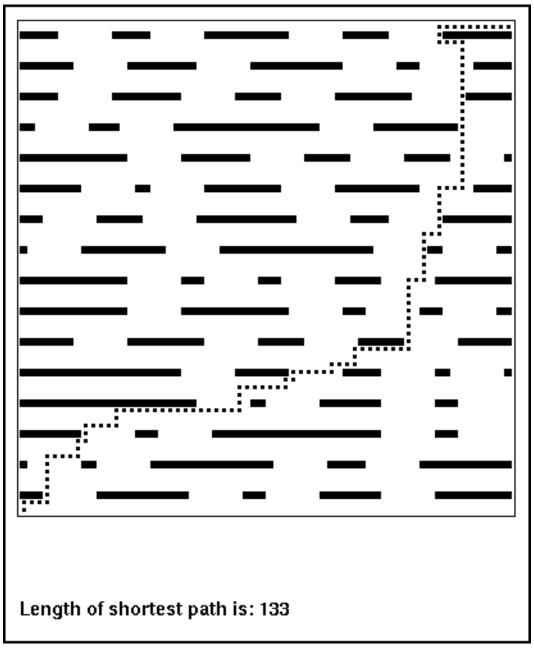


Figure 8.5: Optimal path

as it reaches an obstacle (using a different kind of sensing method). Navigation using obstacle avoidance alone is not efficient and may lengthen the route. In this implementation, as long as there exists a path that connects the start and goal positions, the obstacle avoidance procedure alone can bring the robot to its destination. Therefore, any abstract plan is executable. Obstacle avoidance is clearly not a smart navigation method, but it can always substitute for missing details in an abstract plan. The quality of an abstract plan $\mathcal{P}$ is defined as follows:

$$\mathbf{Quality}(\mathcal{P}) = \frac{\textbf{route-length}(\mathcal{P}_{opt})}{\textbf{route-length}(\mathcal{P})}$$

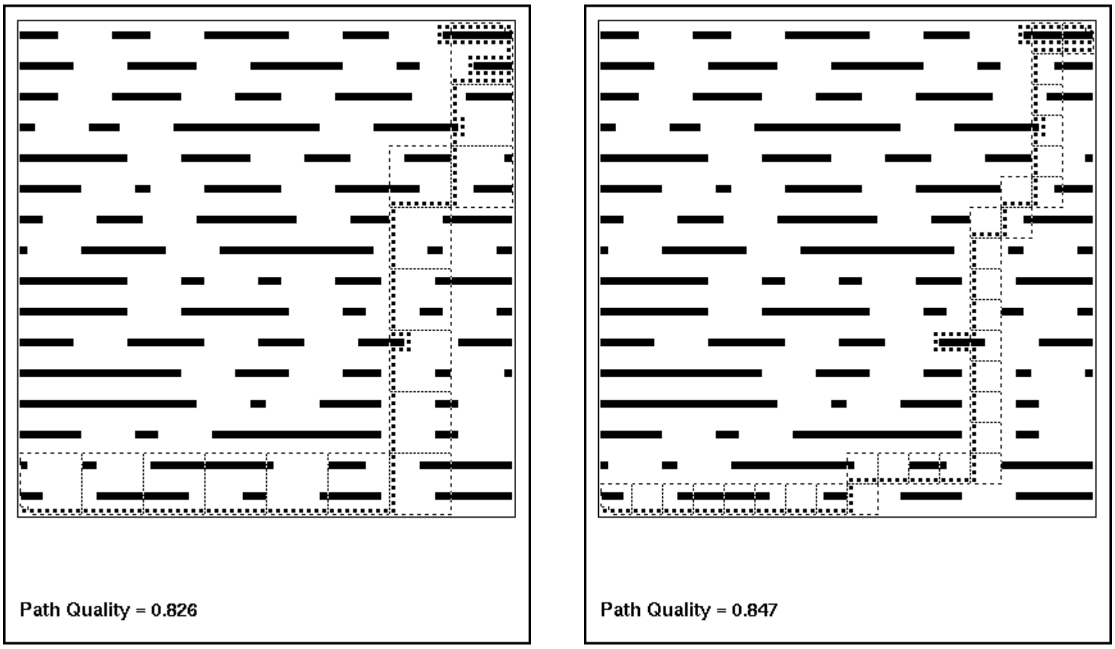
where **route-length**$(\mathcal{P})$ is the length of the route generated when the robot is guided by the plan $\mathcal{P}$, and $\mathcal{P}_{opt}$ is the optimal plan. Note that the higher the level of abstraction, the lower the quality of the plan. At the same time, high-level abstract planning reduces the search space and hence it is performed much faster.

The notion of executable abstract plans – regardless of their level of detail – is made possible by using plans as suggestions that direct the base level execution mechanism but do not impel a particular behavior. This idea was promoted by Agre and Chapman [1990] and was experimentally supported by Gat [1992]. Uncertainty alone makes it impossible to use plans except as a guidance mechanism.

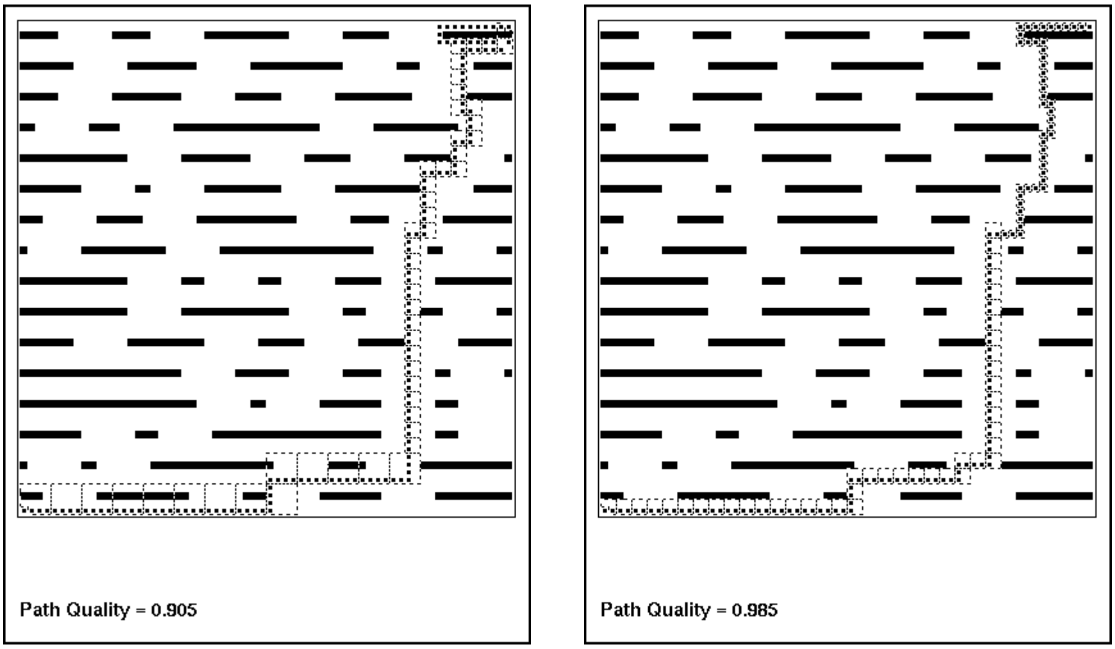
Performance with perfect vision

What is the performance of the abstract planner? First, I will examine the performance under the assumption of perfect domain description. Then, I will examine the effect of degradation in vision on the quality of planning.

Figure 8.5 shows the path found by the path finder when activated with start and goal positions being the lower left and upper right positions respectively. The search level is zero (base level) hence the path shown is optimal (i.e. it is a shortest path).



(a) Level 3 plan (b) Level 2 plan



(c) Level 2/1 plan (d) Level 1/0 plan

Figure 8.6: Abstract plans with perfect vision

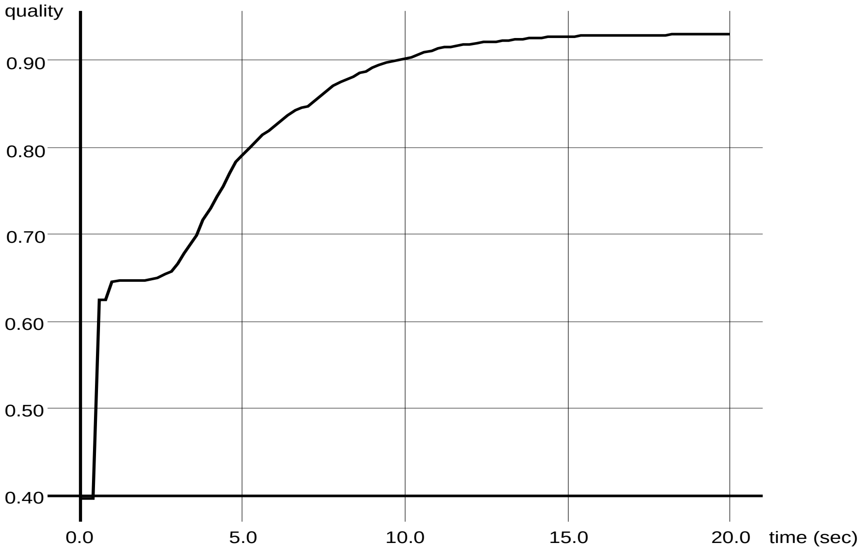


Figure 8.7: The performance profile of the anytime planner

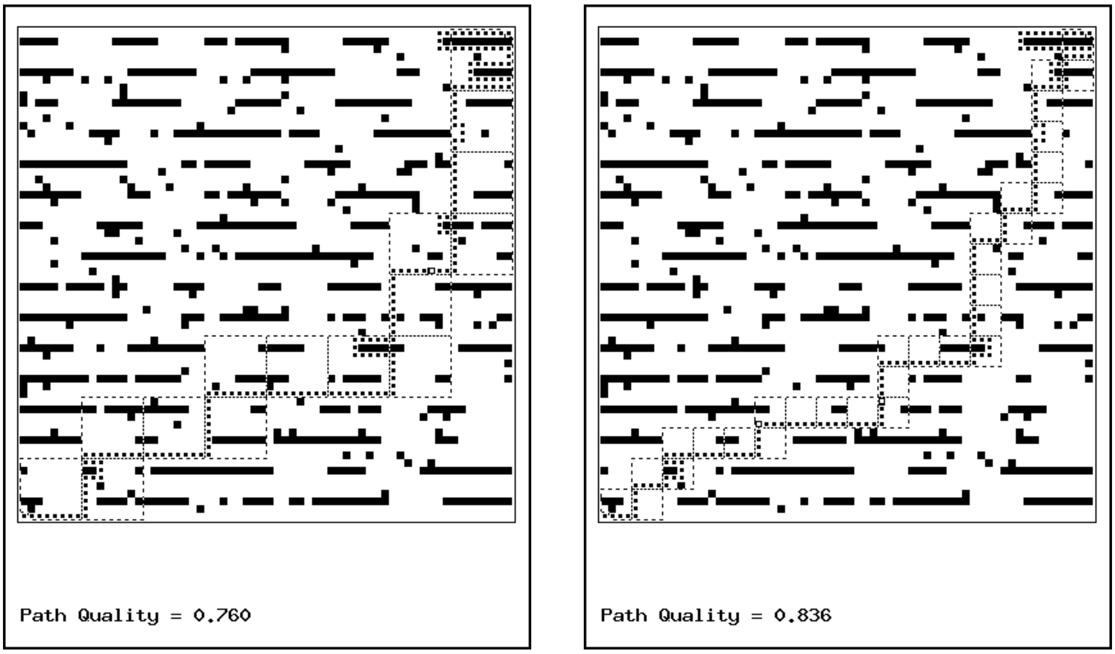
Figure 8.6 shows the paths generated by the path finder when activated with the same start and goal positions but starting at the highest abstraction level. The upper-left frame (a) shows (by the large squares drawn in broken line) an abstract plan at level 3. The quality of the plan, 0.826, is determined by the length of the route the robot would have followed if guided by this plan (shown in the figure by a heavy broken line) compared to the length of the shortest route. The upper-right frame (b) shows a more precise abstract plan with segments at level 2. Its quality is 0.847. The lower-left frame (c) shows an abstract plan with segments at levels 2 and 1. Quality reaches 0.905. Finally, the lower-right frame (d) shows a detailed plan with segments at levels 1 and 0. Although further refinement is possible, it has no effect on the quality of the plan in this particular example. Notice that the quality of the plan reached 0.985 – almost as good as the quality of the shortest path.

The typical performance of the planner is summarized by its performance profile in Figure 8.7. The graph shows the expected quality of the plan as a function of run-time. When run to completion, the expected quality of the plan produced by the abstract planner is 0.93. At the same time, its expected run-time is only 27% of the the expected run-time needed to compute the optimal path using the standard $\mathcal{A}^*$ algorithm. These figures show that anytime algorithms offer both more flexibility and a better cost/performance ratio.

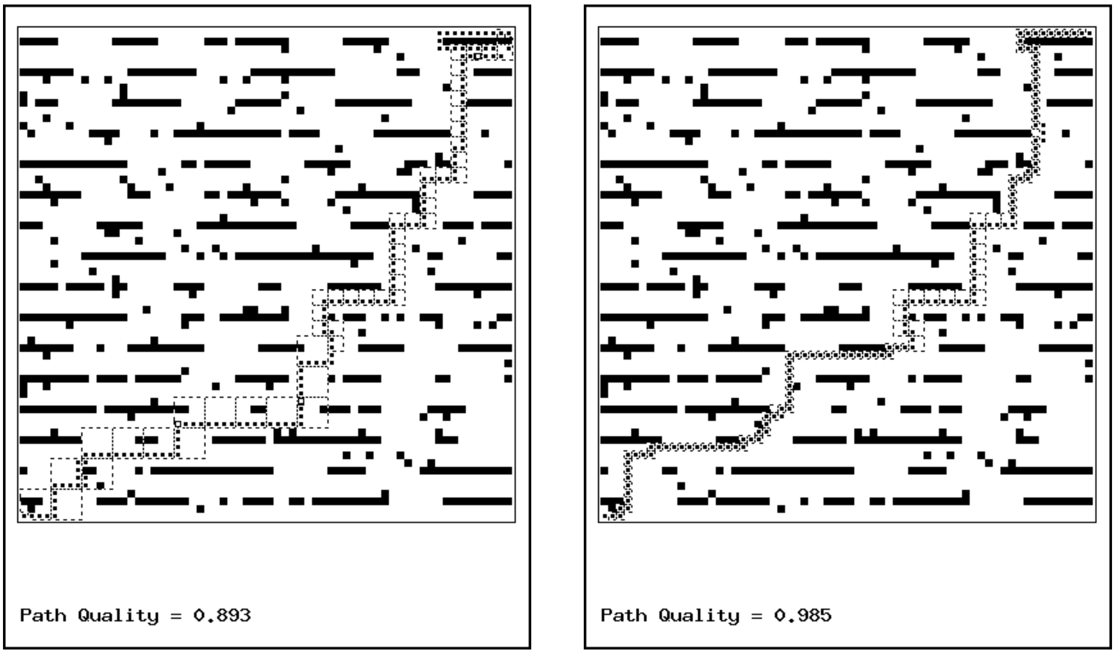
Performance with imperfect vision

I now examine the effect of vision errors on the quality of planning. Suppose, for example, that the quality of the domain description is 0.96. This figure is a measure of the sensor's noise level as described in the previous section. The physical domain is identical to the one used in the previous example. However, the map constructed by the vision module is erroneous.

Figure 8.8 shows snapshots of the plans generated by the algorithm and their qualities. Notice that as a result of lower quality of sensing, the quality of the initial plan is only 0.760 compared to 0.826 with perfect vision. In this particular example, the planner completed its execution with a plan of the same quality as in the perfect vision case, but it took more time. The lower-right frame (d) shows a plan of the



(a) Level 3 plan (b) Level 2 plan



(c) Level 2/1 plan (d) Level 1/0 plan

Figure 8.8: Abstract plans with imperfect vision

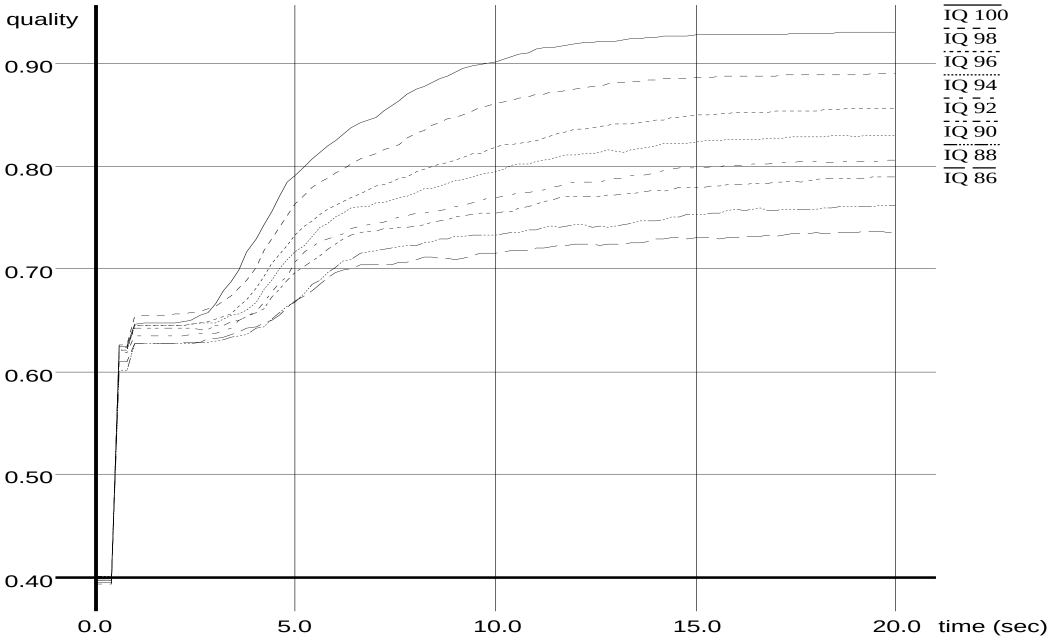


Figure 8.9: The conditional performance profile of the anytime planner

same quality (0.985) as in the perfect vision case, but one can see that in order to reach this quality, the abstract planner had to refine the plan until it was almost entirely in level 0. To summarize, as a result of the error in the domain description, the planner terminates, on average, with a plan of lower quality.

Based on statistics gathered by running the planning algorithm many times on randomly generated domains, its conditional performance profile can be derived. The conditional performance profile describes the expected quality of a plan based on the quality of the domain description and run-time. Figure 8.9 shows the conditional performance profile of the abstract planner. Each curve corresponds to a particular quality of vision and shows the expected plan quality as a function of run-time. The use of the conditional performance profile in order to determine optimal time allocation is further discussed below.

8.1.4 Compilation of sensing and planning

The composition of planning and sensing is a simple example of a compilation of a straight line program as discussed in Section 5.5:

ATP(*start*, *goal*, DOMAIN-DESCRIPTION(*sensor*))

For optimizing time allocation, I implemented the hill-climbing algorithm described in Section 5.5.

Figure 8.10 shows the performance profile that was produced by compiling the sensing and planning modules. Also shown in that figure (for comparison) are the performance profiles of two other modules: MIN, that allocates to vision a minimal amount of time, T_a, and MAX, that allocates to vision a maximal amount of time, T_b. The compiled performance profile is superior to both. It is closer to MIN with small allocations of time and is closer to MAX in the limit.

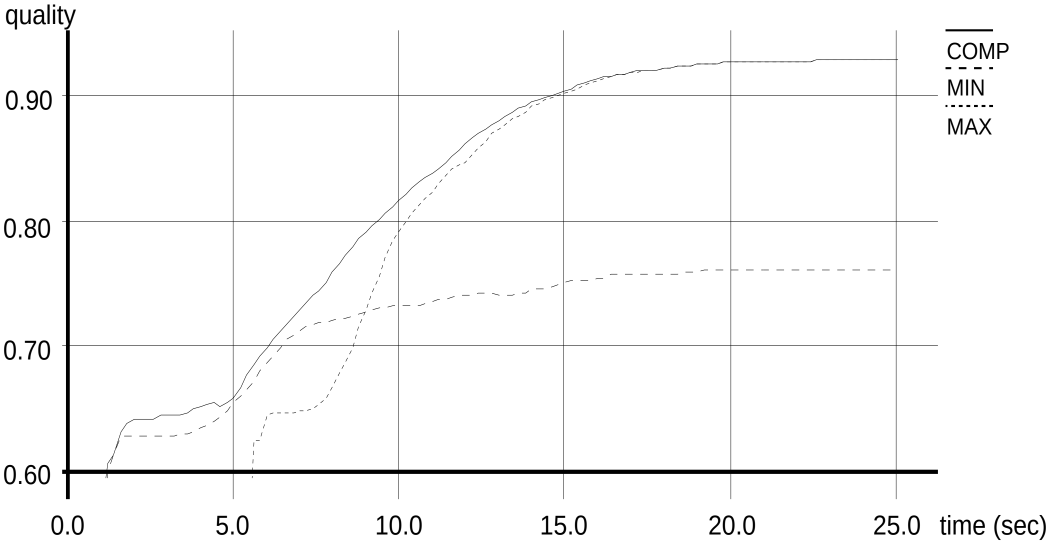


Figure 8.10: Compilation of vision and planning

8.1.5 The run-time system

Systems composed of anytime algorithms require constant monitoring. The compilation process provides the necessary meta-level information to make the run-time monitoring more efficient, but it is not a substitute for monitoring. In this section I explain how the run-time system controls the time allocation to the anytime modules.

The optimization of the long-term behavior of the robot is performed by dividing a complex task into a series of small sensing, planning and plan execution episodes called *frames*. For each episode, the anytime sensing and planning modules that were described earlier are used. Since, in many cases, sensing capability is limited to a small, local segment of the environment, it is only natural to break the navigation problem into small episodes. Each episode is monitored using dynamic readjustment of contract time as described in Chapter 6.

The task of the meta-level control is to determine the optimal initial contract time for each frame. This decision – *inter-frame optimization* – is made in the following way: let t be the current time (real-time since the beginning of the execution of the task), let f_t be the (estimated) number of frames left at time t for planning and execution, let t_c be the contract time for the next cycle of planning and execution, and let e_t be the energy used so far for plan execution. Then:

$$t_c = argmax_{t_i}\{VOT(t, f_t, t_i) - COE(e_t, f_t, t_i)\}$$

where VOT is the expected value of the task and COE is the expected cost of energy. Note that both the performance profile of the system and a model of the environment are necessary in order to compute these functions.

Once an initial contract time is determined, the system starts allocating resources to sensing, planning and plan execution. At the same time, it continues to monitor the performance of the anytime modules. This constant monitoring is necessary because of the uncertainty concerning the *actual* quality of plans and

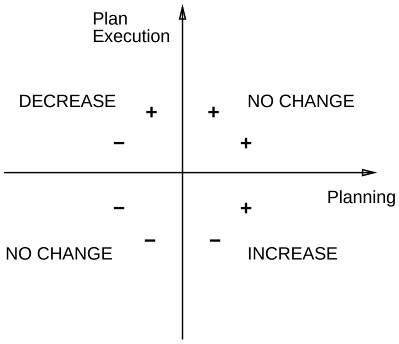


Figure 8.11: Intra-frame optimization

the *actual* time necessary to execute them. The purpose of the meta-level control in this phase is to reach an optimal plan quality for the next frame while executing a previously derived plan. For this purpose, it can modify the initial contract time. This decision – *intra-frame optimization* – is made in the following way:

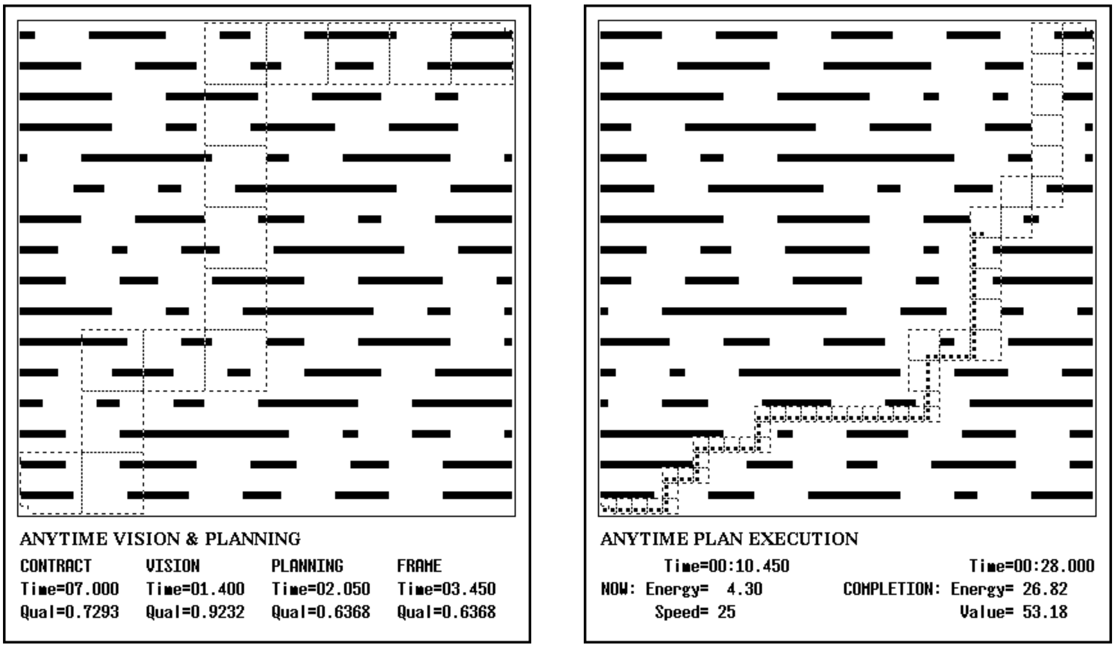
The monitor determines at each point whether planning is ahead of or behind expectations by comparing the (estimated) plan quality to the quality advertised by the performance profile. It also determines whether plan execution is ahead of or behind expectations by comparing the (estimated) execution time to the frame contract time. Figure 8.11 shows how the resource allocation decision is made. In this figure, $+$ represents a process that advanced faster than initially expected and – represents a process whose performance is below expectations. If planning is ahead of expectations and plan execution is behind, the monitor accelerates plan execution by allocating more resources (energy) to plan execution. If planning is ahead and plan execution is behind, the monitor slows down plan execution by reducing resource consumption.

This monitoring strategy can be modified in various ways. For example, one can consider planning more than one frame ahead when plan execution is slow. Another possibility in this case is to replan part of the plan to accelerate its execution. However, experiments with the above domain show that the monitoring strategy that was implemented is sufficient in order to achieve (within 4% error) the optimal task value that the system computes when presented with the task.

Figure 8.12 shows the display of the run-time system. The left frame shows an intermediate state of sensing and planning and includes the best plan so far, its expected quality, the contract time, frame time, sensing time and its quality. The right frame shows the plan execution (at the same time) and includes the path followed by the robot, the current time and the energy consumed so far. It also shows the expected task completion time and value.

8.2 Other applications

In this section I describe two additional applications of the model of anytime computation. Both applications were influenced by Russell and Zilberstein's [1991] paper on the composition of real-time systems. The first project involves the construction of a diagnostic system by Anita Pos and René Bakker from the University of Twente, The Netherlands. The second project involves the analysis of anytime generate-and-test algorithms by a research group at the German National Research Institute for Computer Science (GMD).



```
ANYTIME VISION & PLANNING
CONTRACT      VISION        PLANNING      FRAME
Time=07.000   Time=01.400   Time=02.050   Time=03.450
Qual=0.7293   Qual=0.9232   Qual=0.6368   Qual=0.6368
```

```
ANYTIME PLAN EXECUTION
       Time=00:10.450                       Time=00:28.000
NOW: Energy=  4.30          COMPLETION: Energy= 26.82
      Speed= 25                          Value= 53.18
```

Figure 8.12: The run-time display

8.2.1 Anytime pragmatic diagnostic engine

Model-based diagnostic methods [Davis and Hamscher, 1988] identify defective components in a technical system by a guided series of tests and probes. Advice on informative probes and tests is given using diagnostic hypotheses that are based on observations and a model of the system. The goal of model-based diagnosis is to locate the defective components using a small number of probes and tests.

The General Diagnostic Engine [de Kleer and Williams, 1987] (GDE) is a basic method for model-based diagnostic reasoning. In GDE, observations and a model of a system are used in order to derive *conflicts*[1]. These conflicts are transformed to *diagnoses*[2]. The process of observing, conflict generation, transformation to diagnoses, and probe advice is repeated until the defective components are identified. This process is shown in Figure 8.13(a).

GDE has a high computational complexity – O(2^n), where n is the number of components. As a result, its applicability is limited to small-scale applications [de Kleer, 1991]. To overcome this difficulty, Bakker and Bourseau [1992] have developed a model-based diagnostic method, called Pragmatic Diagnostic Engine (PDE), whose computational complexity is O(n^2). PDE is similar to GDE, except for omitting the stage of generating all diagnoses before determining the best measurement-point. Probe advice is given on the basis of the most relevant conflicts, called *obvious* and *semi-obvious* conflicts[3]. This process is shown in Figure 8.13(b).

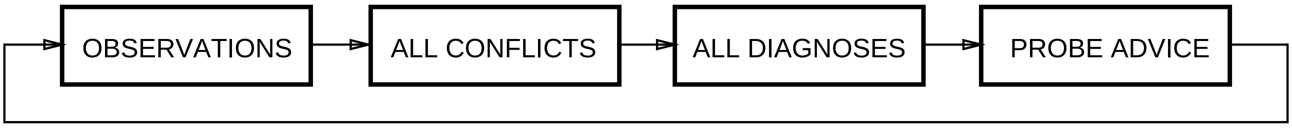
In order to construct a real-time diagnostic system, Anita Pos [Pos, 1992] has applied the model of compilation of anytime algorithms to the PDE architecture. PDE can be analyzed as a composition of two (anytime) modules. In the first module, a subset of all conflicts is determined. Pos implements this module

---

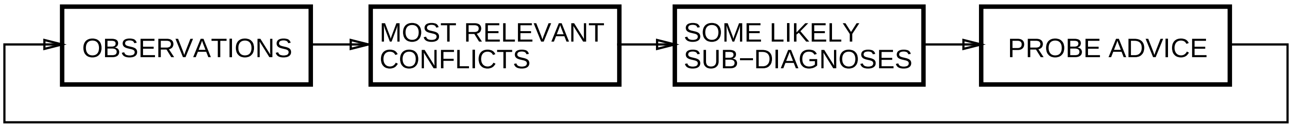
[1] A conflict is a set of components of which at least one has to be defective.

[2] A diagnosis is a set of defective components that might explain the deviating behavior of the system.

[3] An obvious (semi-obvious) conflict is a conflict that is computed using no more than one (two) observed outputs.



(a) The GDE architecture



(b) The PDE architecture

Figure 8.13: Architectures for model based diagnosis

by a contract form of breadth-first search. The second module consists of a repeated loop that determines which measurement should be taken next, takes that measurement and assimilates the new information into the current set of conflicts. Finally, the resulting diagnoses are reported.

To represent the performance information, Pos uses a combination of expected performance profile and performance interval profile (defined in Section 4.2). The combined performance profile is named Statistical Performance Profile or PSP. It records not only upper and lower bounds but also the mean of the sample set. For the purpose of decision making, the mean is used instead of the center of the interval.

The quality combination function (set multiplication) maps the mean of each sub-process to the mean of the complete process, so that the resulting performance profile is again a PSP. An interesting result of the above decomposition is that interaction with the user, who has to actually take measurements, has to be considered in determining the performance profile of the second sub-process. This is achieved using expert knowledge and statistical experiments to determine the average time necessary to take each measurement. This knowledge is then incorporated into the performance profile.

Two versions of the diagnostic system have been implemented: one by constructing a contract algorithm and the other by making the contract system interruptible using Theorem 4.1. The actual slow down factor of the interruptible system was approximately 2, much better than the worst case theoretical ratio of 4.

8.2.2 Generate-and-test search

Coulon *et al.* [1992] have analyzed several control strategies to perform a generate-and-test search at any time. One of their goals is to examine the applicability of the model of anytime computation to the composition of generate-and-test modules. The data flow of information is as follows:

input $\Longrightarrow$ *GENERATE* $\Longrightarrow$ *hypotheses* $\Longrightarrow$ *TEST* $\Longrightarrow$ *solutions*

The problem definition includes a stopping criterion for the system. The stopping criterion is a disjunction of a lower bound on the number of solutions, sol_{min}, and an upper bound on time allocation, t_{max}. Given a stopping criterion, the goal of the system is to maximize the number of solutions s and minimize computation time t. The expected utility is defined as follows:

$$U(s,t) = s/t$$

To simplify the analysis, the following four assumptions are made: (A1) all generated hypotheses are tested before a new set may be generated; (A2) a non-trivial constant time, $t_{switch}/2$, is required for switching from the generate module to testing and vice versa; (A3) generating a hypothesis and testing it require constant times (t_{gen} and t_{test} respectively) for all hypotheses; and (A4) solutions are distributed uniformly among the hypotheses. Four possible control strategies are presented and analyzed:

Strategy 1 – directly generate the right number of hypotheses
If one knows the probability p that a generated hypothesis becomes a solution, one can estimate the number of hypotheses needed to obtain sol_{min} solutions. The authors use p in order to define a control strategy that does not require switching back from testing to hypothesis generation. It performs only one iteration that generates sol_{min}/p hypotheses and tests them. The paper claims that this strategy is optimal. However, since it does not satisfy the stop criterion mentioned above nor does it guarantee termination with sol_{min} solutions, the claim of optimality is inaccurate.

Strategy 2 – eager generation
This strategy first generates all the possible hypotheses and then tests them, producing all the solutions (whose number is sol_{all}). As with Strategy 1, it also has the “advantage” of not switching back from test to generate. Since a realistic domain may have a very large, sometimes infinite, hypothesis set, the generation of all hypotheses seems impractical.

Strategy 3 – lazy generation
This strategy generates one hypothesis at a time, tests it and switches back to generate. The expected time to compute sol_{min} hypotheses is:

$$\lceil\frac{sol_{min}}{p}\rceil(t_{gen} + t_{test} + t_{switch})$$

where $\lceil\cdot\rceil$ is the ceiling function. Obviously, if $t_{switch} = 0$, this strategy is optimal.

Strategy 4 – generate the number of missing solutions
This strategy generates in each iteration as many hypotheses as the number of missing solutions.

The authors compare the strategies based on their performance as shown in Figure 8.14. Their final analysis identifies two cases of termination: by reaching $sol \geq sol_{min}$ and by reaching $t \leq t_{max}$. In the first case, the choice of strategy depends on the proportion of solution p, the ratio between t_{switch} and $t_{gen} + t_{test}$, and the ratio between sol_{all} and sol_{min}. When p is known, the authors claim that Strategy 1 is optimal. In the second case, the authors claim that the methods have different properties in terms of risk and performance and that the last strategy represents a compromise that offers high-payoff/high-risk in the beginning and gradually becomes low-payoff/low-risk. The authors say that this behavior is advantageous since the chance of running into t_{max} increases over time. The last conclusion confuses the problem definition with the notion of interruptibility. When a stopping criterion is given (specifying the maximal time allocation), the meaning of high-risk (or low-risk) is unclear. The problem definition is based on satisfying a stopping criterion rather than on a stochastic deadline.

Coulon *et al.* conclude that only when the probability p of a hypothesis being a solution is known, is an optimal strategy (based on compilation) possible. When p is unknown, it means that the performance

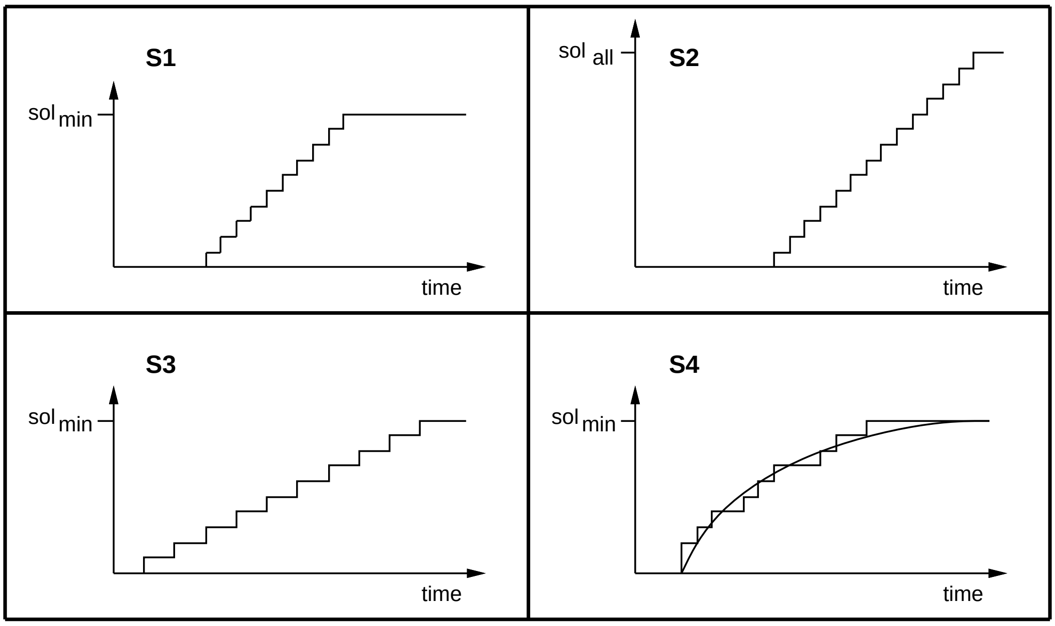


Figure 8.14: Distribution of solutions over time for S1-4

profile is unknown and hence compilation is obviously impossible. This presents an interesting question of how to utilize systems composed of anytime algorithms when the performance profiles are not given. A reasonable approach would be to learn the performance profile while the system is working and adjust the time allocation to reflect changes in the performance profile. An initial estimate of the performance profile is needed.

8.3 Model evaluation

I conclude this chapter with a summary of the experience that I had with the development and application of the model of operational rationality. Most of my conclusions relate to my own work on the mobile robot navigation system. An extensive theoretical analysis of the model and its capabilities appears in the previous chapters. The purpose of this section is to assess the potential of operational rationality to become a practical method capable of both simplifying the development of real-time systems and optimizing their behavior.

8.3.1 Constructing elementary anytime algorithms

The construction of elementary anytime algorithms has been found to be as simple as standard programming. It does not impose any new limitations on the programmer beyond the simple requirement of registering intermediate results. Standard programming methods, some of which were surveyed in Chapter 4, extend naturally to anytime computation.

Another important aspect of anytime computation that seems to pose no difficulty is the selection of an appropriate quality measure to characterize the performance of the algorithm. Although more extensive research and many more applications are needed to examine this aspect, preliminary experience shows that anytime algorithms have “natural” quality measures. In optimization problems, such as minimal cost path finding, the natural quality measure of any given path is the ratio between its length and the length of an optimal solution. This approach has been used successfully in the mobile robot navigation system. A possible difficulty may arise when the optimal solution is too hard to compute, even by an off-line simulation program. In such a case one, can use the ratio between the length of the path and the *distance* between the start and goal positions as a reasonable quality measure.

8.3.2 Computing performance profiles

The computation of performance profiles is, at the moment, a somewhat tedious task. Many decisions regarding the representation of the performance profile are made by the programmer. These decisions include the selection of the range of time allocation, the resolution of time and quality, and the choice of an appropriate range of initial input qualities for the construction of the conditional performance profile. My experience shows that in the prototype system the single aspect that has the highest effect on the development cycle is the computation of the performance profiles. It takes hours to gather the essential statistics to build the quality map and construct the performance profile, even after determining the parameters of the representation. This process, which is completely automated, cannot be avoided when using anytime algorithms. The system however can be debugged at the same time using an older (or an approximate) performance profile. Inaccurate performance profiles affect only the performance of the system but not its logical operation.

Another interesting difficulty that was observed by Anita Pos relates to the construction of performance profiles when interaction with a user is part of the main line of the algorithm. For example, in the second phase of her diagnostic system, the anytime algorithm consists of a loop that determines which measurement should be taken next, takes that measurement and assimilates the new information into the current set of conflicts. Measurements are actually taken by the user. To construct the performance profile, Pos used the knowledge of an expert and statistical experiments to determine the average time necessary for manual measurements.

8.3.3 Compilation

Local compilation of anytime algorithms is, in fact, a simple, fast process. The relatively small size of the application that I developed made it possible to apply both globally optimal and approximate compilation methods. In fact, I have compared the efficient hill-climbing time allocation algorithm and the algorithm that uses complete search to guarantee optimality. I found that the results were practically the same in terms of the allocation to the components and the overall quality. Only one entry in the tables representing the performance profiles was slightly different. More experience is needed, however, to test the performance of the hill-climbing allocation algorithm and to determine how close it is to the optimal output quality.

8.3.4 Achieving operational rationality

The applications developed so far show that operational rationality can be achieved in practical domains. This is an important result of this dissertation since the very principle of operational rationality and anytime computation offer, by their nature, a performance improvement over traditional approaches to real-time system development.

Recall, however, that the performance of the agent is optimized *given* a certain set of performance components and a system design that are decided by the developer. Therefore, one cannot immediately conclude that the performance achieved by an operationally rational agent is best in absolute terms. But it is clearly an efficient tool to achieve superior performance with respect to agents whose components are based on producing output of fixed expected quality.

Several fundamental aspects of the model of operational rationality are still hard to evaluate. These aspects can be evaluated only after a large number of applications are developed. This leaves some open questions regarding the potential of the model. For example, to what degree could one create a large, general purpose library of anytime algorithms? To what degree would anytime computation and automatic compilation simplify the development of real-time systems? I will return to these questions in my concluding remarks.

Chapter 9

Conclusion

> Civilization advances by extending the number of important operations which we can perform without thinking about them.
>
> Alfred North Whitehead, *An Introduction to Mathematics*

I have examined the problem of real-time decision making by intelligent agents. The result of this examination has been the development of an efficient model of bounded optimality that is based on anytime computation, off-line compilation, and run-time monitoring. In this chapter I will summarize the contribution of this work and identify the main aspects of the model to be studied and refined in the future.

9.1 Contribution

The model of operational rationality offers both a methodological and a practical contribution to the field of real-time decision making and to artificial intelligence in general. The main aspects of this contribution are summarized below:

Design of complex real-time systems

Operational rationality offers a modular approach to the design of complex real-time systems. Separating the design of the system from the optimization of its performance introduces a new type of modularity and a large degree of simplification. Many researchers in the real-time community believe that the development of real-time systems is incompatible with the principles of abstraction since abstraction emphasizes the functional requirements but ignores the timing constraints of the system. The model of operational rationality shows how the two aspects can be addressed independently and hence it encourages the use of abstraction in real-time system design.

Foundations of anytime computation

Operational rationality is largely based on anytime computation. The optimal scheduling and control of anytime algorithms forms the center of the model. The utility of approximate computation has been long appreciated by the computer science community. However, it has also been recognized that a major obstacle to the wide spread use of approximate computation is its incompatibility with standard software engineering principles. The principal problems have involved the estimation of the cumulative effect of error in the system, the control of approximate computation, and the great degree of unpredictability associated with approximate computation. By introducing modularity into anytime computation and by mechanizing the scheduling task, operational rationality constitutes an important step toward the complete integration of approximate computation into standard system development methodologies. Performance profiles give the designer a high degree of performance predictability and the standard library of anytime algorithms encourages sharing and re-using anytime modules among different applications.

Resource bounded reasoning

The problem of optimal decision making with limited resources has been recognized as a hard problem in artificial intelligence, in engineering, in economics and in philosophy. This dissertation presents a general approach to solving this problem in two steps. First, certain structural constraints on the agent are established. Then, and only then, the question of optimal decision making can be considered in the context of those structural constraints. Since the chosen architecture can be arbitrarily restricted, the corresponding optimization problem can be arbitrarily simple. The general trend should be to investigate the solutions one obtains as the constraints on the agent's architecture are relaxed. Operational rationality offers a solution to the problem of resource bounded reasoning that fits this framework. Its structural premise is that the agent is designed by composing anytime algorithms.

Machine independent real-time systems

Finally, operational rationality defines real-time systems by a time-dependent utility function. The dynamic monitoring and the use of anytime computation make it possible to construct machine independent real-time systems, a concept that has been considered a self-contradiction in the past. When an anytime system is installed on a slower machine, it would automatically adjust the time allocation to its components to maximize its utility. In response to a reduction in computational power, such systems offer a corresponding reduction in performance rather than failing to produce results at all.

9.2 Further work

Further work is required to generalize the components of the model of operational rationality and to further validate its effectiveness. This section identifies the three major directions such work could take.

9.2.1 The scope of compilation

While the compilation process has been developed and theoretically analyzed for large programs, its practical use has been limited to small programs. Larger applications of the model need to be examined in order to validate the vital role of local compilation. In addition, the scope of compilation needs to be extended to include more programming structures. The compilation of recursive anytime functions, for example, is yet to be realized.

9.2.2 The theoretical framework

The theoretical framework of anytime computation has to be further extended to include anytime sensing and anytime action. To fully integrate the notion of action into the optimization problem, one needs to

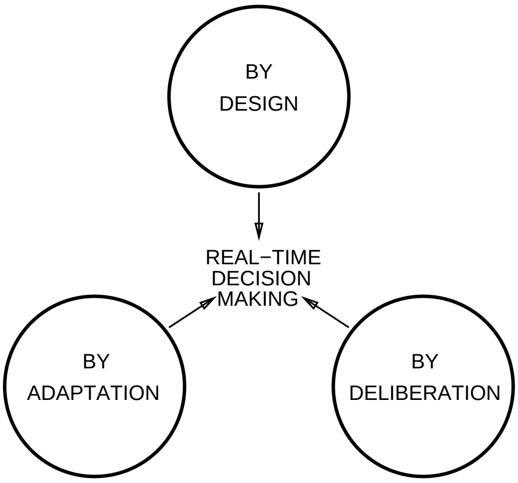


Figure 9.1: Real-time decision making

examine the optimal behavior of agents over histories. This requires the development of mechanisms to handle the exponential complexity of projection into the far future.

9.2.3 Programming support for anytime computation

Anytime computation requires a number of aspects of system development to be re-defined. One important aspect is algorithm specification. In traditional programming, algorithm specification is based on binary truth criteria that determine when an answer is correct, *independent* of its computation time. This kind of specification is no longer useful with anytime computation. A performance profile seems to be a more adequate mechanism for algorithm specification. But, as erroneous results become permissible in program development, a distinction must be drawn between an inefficient anytime algorithm and a programming bug. This distinction can be based on a *minimal* performance profile that sets up a lower bound on the performance of an acceptable algorithm.

Another aspect is the integration of all aspects of anytime programming into the programming environment. The Concord system of Chapter 2 represents a preliminary step in this direction, but many questions are still left open. How would the performance profiles in the library be linked to the algorithms they describe? How would the programmer indicate the type of compilation method to be used? What degree of control would the programmer have over the modification of the library to better match a particular problem domain? Further work and experience with anytime computation are needed to answer these questions.

Finally, debugging and testing tools for anytime algorithms must be developed that would address the special characteristics of these algorithms.

9.3 Design, deliberation and adaptation

The problem of real-time decision making has been addressed by many researchers over the years. Three basic mechanisms to solve this problem can be identified [Russell and Wefald, 1989b], as shown in Figure 9.1.

1. Real-time decision making by design – in which the *designer* possesses the computational and informational resources required to find optimal solutions and uses them in order to build a system that “does the right thing.”
2. Real-time decision making by deliberation – in which the agent itself performs explicit deliberation in order to make decisions, possibly using compilation to improve its reactivity over time.
3. Real-time decision making by adaptation – in which the agent is equipped with a mechanism to adjust its behavior in response to feedback from the environment so that the quality of its decisions improves over time.

The challenge of artificial intelligence, I would argue, is to reduce the burden on the designer by moving primary system construction mechanisms from the first category into the other two. However, in the current state of artificial intelligence, sophisticated tasks are achieved largely by design. Explicit deliberation and adaptation techniques are simply not fast enough to implement intelligent agents, let alone real-time agents. In that respect, operational rationality offers a significant new direction. Operational rationality is based on a combination of the three mechanisms: the outline of the system and its performance components are solved by design; the resource allocation problem to the components is solved by meta-level deliberation; and the information regarding the performance of the elementary components is constructed by adaptation in the context of a particular problem domain.

The field of artificial intelligence has been at a crossroads in recent years. Classical techniques, that have been proven inadequate, are being gradually replaced by techniques that better address the problems of uncertainty, incomplete information and bounded computational resources. The model of operational rationality addresses these issues. Further research in this area will contribute to our understanding of the limits and capabilities of intelligent agents.

Glossary

Many of the basic concepts and terms in the area of anytime computation are defined for the first time in this dissertation. Other terms have been used in the past but still lack a generally accepted definition. For this reason, I have included the following short summary of specialized terminology. The definitions here are informal and are intended for clarification purposes only. Exact definitions can be found in the body of this dissertation.

anytime action An external action of an agent whose degree of goal achievement improves gradually as execution time increases.

anytime algorithm An algorithm whose quality of results improves gradually as computation time increases.

anytime computation A model of computation that allows using anytime algorithms as basic components and includes mechanisms for automatic scheduling the components so as to maximize a certain objective function.

anytime sensing A sensing procedure whose quality of domain description improves gradually as sensing time increases.

compilation of anytime algorithms An off-line process that inserts time allocation code in a compound anytime algorithm and prepares auxiliary meta-level information for efficient scheduling of the components. The meta-level performance information is computed by the compiler using the performance profiles of the elementary anytime components.

completion time of an anytime algorithm. The minimal amount of time required by an anytime algorithm to guarantee that output quality reaches its maximal value and that no further computation can improve the quality of the output.

compound anytime algorithm An anytime algorithm composed of one of more elementary anytime algorithms using certain program composition operator.

comprehensive value of a computation The net value of the information produced by a computation. Sometimes represented as the difference between the intrinsic value of the information and the cost of resources consumed by the computation.

conditional performance profile of an anytime algorithm. A mapping that determines the probabilistic characterization of the quality of the output of an anytime algorithm as a function of run-time and a set of input attributes, usually the quality of the input.

contract anytime algorithm An anytime algorithm that returns results as characterized by its performance profile when the time allocation is determined in advance, before activating the algorithm, and is known to the algorithm itself.

deliberation value of an interruptible anytime decision system. The marginal value of continued computation. The difference between the comprehensive value of taking an action in the future based on further deliberation and the comprehensive value of taking immediate action based on the current results. A negative value indicates that deliberation should be interrupted and the current best results should be used.

episodic problem solving A problem solving approach in which the complete description of a problem instance is introduced to the problem solver as input and no further input is considered before the termination of the problem solving process.

generalized action A common reference to any aspect of the behavior of an agent as an action: computations are internal actions, base-level actions are external, and sensing activity is information gathering action.

intrinsic value of a computation The value of the results of a computation with respect to a fixed predetermined problem description, regardless of the time consumed by the computation and the possibility that the problem description may not remain accurate as the environment changes.

interruptible anytime algorithm An anytime algorithm that returns results as characterized by its performance profile when interrupted at an arbitrary point after its activation.

local compilation method A compilation method that is performed by considering only one program construct at a time and using only the performance profiles of its immediate components. The immediate components are treated as if they are elementary anytime algorithms.

operational rationality The theory of scheduling anytime computation so as to maximize the degree of goal achievement determined by a certain utility function.

pathological anytime algorithm An anytime algorithm whose quality of results is not a non-decreasing function of time.

performance profile of an anytime algorithm. A mapping that determines the probabilistic characterization of the quality of the output of an anytime algorithm as a function of run-time.

performance profile library A database where the (conditional) performance profiles of all the elementary anytime algorithms are originally stored and where the compiler stores the (conditional) performance profiles of compound anytime algorithms.

quality map of an anytime algorithm. A set of pairs each indicating a particular quality of results that was achieved by running the algorithm with a particular time allocation. Quality map is always defined with respect to a particular distribution of input instances and a particular input quality. Used to construct the conditional performance profile of an anytime algorithm.

Bibliography

[Agogino *et al.*, 1988] A. M. Agogino, S. Srinivas and K. M. Schneider. Multiple sensor expert system for diagnostic reasoning, monitoring and control of mechanical systems. In *Mechanical Systems and Signal Processing*, **2**:165–185, 1988.

[Agogino, 1989] A. M. Agogino. Real-time reasoning about time constraints and model precision in complex, distributed mechanical systems. In *Proceedings of the AAAI Spring Symposium on AI and Limited Rationality*, Stanford, California, 1989.

[Agre and Chapman, 1987] P. E. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 268–272, Seattle, Washington, 1987.

[Agre and Chapman, 1990] P. E. Agre and D. Chapman. What are plans for? In *Robotics and Autonomous Systems*, **6**:17–34, 1990.

[Alexander *et al.*, 1992] P. D. Alexander, C. C. Lim, J. W. S. Liu and W. Zhao. Managing transient overload in an imprecise computation system. In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, pp. 1–5, Phoenix, Arizona, 1992.

[Bacchus, 1988] F. Bacchus. *Representing and Reasoning with Probabilistic Knowledge*. Research Report CS-88-31, Department of Computer Science, University of Alberta, Edmonton, Alberta 1988.

[Bakker and Bourseau, 1992] R. R. Bakker and M. Bourseau. Pragmatic reasoning in model-based diagnosis. In proceedings of *The 10th European Conference on Artificial Intelligence*, pp. 734–738, Vienna, Austria, 1992.

[Batali, 1986] J. Batali. Reasoning about control in software meta-level architectures. In *Proceedings of the First Workshop on Meta-Architectures and Reflection*, Sardinia, 1986.

[Boddy and Dean, 1989] M. Boddy and T. L. Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 979–984, Detroit, Michigan, 1989.

[Boddy, 1991] M. Boddy. Anytime problem solving using dynamic programming. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 738–743, Anaheim, California, 1991.

[Bresina and Drummond, 1990] J. Bresina and M. Drummond. Integrating planning and reaction. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain Environments*, Palo Alto, California, 1990.

[Brooks, 1986] R. A. Brooks. A robust, layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*,**2**(1):14–23, 1986.

[Brooks, 1989] R. A. Brooks. A robot that walks: Emergent behavior from a carefully evolved network. *Neural Computation*,**1**(2):253–262, 1989.

[Brooks, 1991] R. A. Brooks. Intelligence without reason. Computer and Thought Lecture. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 569–595, Sydney, Australia, 1991.

[Bylander, 1992] T. Bylander. Complexity results for extended planning. In *Proceedings of the First International Conference on AI Planning Systems*, pp. 20–27, College Park, Maryland, 1992.

[Chapman, 1989] D. Chapman. Penguins can make cake. *AI magazine* **10**(4):45–50, Winter 1989.

[Charniak and McDermott, 1985] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Reading, Massachusetts: Addison-Wesley, 1985.

[Cheeseman *et al.*, 1988] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor and D. Freeman. Autoclass: a Bayesian classification system. In *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[Cherniak, 1986] C. Cherniak. *Minimal Rationality*. Cambridge, Massachusetts: MIT Press, 1986.

[Christofides, 1976] N. Christofides. *Worst-case analysis of a new heuristic for the traveling salesman problem*. Technical Report, Graduate School of Industrial Administration, Carnehie-Mellon University, Pittsburgh, Pennsylvania, 1976.

[Chung *et al.*, 1990] J-Y. Chung, J. W. S. Liu and K-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, **39**(9):1156–1174, 1990.

[Conroy, 1991] J. M. Conroy. *Decision-theoretic control of search in probabilistic domains*. MS Report, Computer Science Division, University of California, Berkeley, 1991.

[Coulon *et al.*, 1992] C. H. Coulon, F. van Harmelen, W. Karbach and A. Vob. Controlling generate & test in any time. In proceedings of *The ECAI-92 Workshop on Advances in Real-Time Expert System Technologies*, Vienna, Austria, 1992.

[D'Ambrosio, 1989] B. D'Ambrosio. Resource bounded agents in an uncertain world. In *Proceedings of the Workshop on Real-Time Artificial Intelligence Problems*, (IJCAI-89), Detroit, Michigan, 1989.

[Davis, 1980] R. Davis. Meta-rules: Reasoning about control. *Artificial Intelligence* **15**:179–222, 1980.

[Davis and Hamscher, 1988] R. Davis and W. Hamscher. Model-Based Reasoning: Troubleshooting. In Shrobe, H. E. (Ed.), *Exploring Artificial Intelligence*, pp. 297–346, San Mateo, California: Morgan Kaufmann, 1988.

[Dean, 1987] T. L. Dean. Intractability and time-dependent planning. In *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, M. P. Georgeff and A. L. Lansky (eds.), Los Altos, California: Morgan Kaufmann, 1987.

[Dean and Boddy, 1988] T. L. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 49–54, Minneapolis, Minnesota, 1988.

[Dean and Wellman, 1991] T. L. Dean and M. P. Wellman. *Planning and Control*. San Mateo, California: Morgan Kaufmann, 1991.

[Dean *et al.*, 1993] T. L. Dean, L. Kaelbling, J. Kirman and A. Nicholson. Planning with deadlines in stochastic domains. To appear in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, D.C., 1993.

[Decker *et al.*, 1990] K. S. Decker, V. R. Lesser and R. C. Whitehair. Extending a blackboard architecture for approximate processing. In *Journal of Real-Time Systems*, **2**(1/2):47–79, 1990.

[de Kleer and Williams, 1987] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence* **32**: 97–130, 1987.

[de Kleer, 1991] J. de Kleer. Focusing on probable diagnoses. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 842–848, Anaheim, California, 1991.

[Doyle, 1988] J. Doyle. *Artificial intelligence and rational self-government*. Technical Report CMU-CS-88-124, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1988.

[Doyle, 1990] J. Doyle. Rationality and its roles in reasoning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 1093–1100, Boston, Massachusetts, 1990.

[Elkan, 1990] C. Elkan. Incremental, approximate planning: Abductive default reasoning. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain Environments*, Palo Alto, California, 1990.

[Fehling and Russell, 1989] M. Fehling and S. J. Russell (eds.). *Proceedings of the AAAI Spring Symposium on Limited Rationality*. Stanford, California, 1989.

[Frege, 1879] G. Frege. Begriffsschrift, a formula language modeled upon that of arithmetic, for pure thought. in van Heijenoort, J. (Ed.), *Frege and Gödel: Two Fundamental Texts in Mathematical logic*, pp. 1–82, Cambridge, Massachusetts: Harvard University Press, 1970.

[Gardner, 1968] M. Gardner. *Logic Machines, Diagrams and Boolean Algebra*. New York: Dover Publications, Inc., 1968.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, California: W. H. Freeman and Company, 1979.

[Garvey and Lesser, 1992] A. Garvey and V. Lesser. Scheduling satisficing tasks with a focus on design-to-time scheduling. In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, pp. 25–29, Phoenix, Arizona, 1992.

[Garvey and Lesser, 1993] A. Garvey and V. Lesser. Design-to-time real-time scheduling. To appear in *IEEE Transactions on Systems, Man and Cybernetics*, 1993.

[Gat, 1992] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 809–815, San Jose, California, 1992.

[Ginsberg, 1989] M. L. Ginsberg. Universal planning: An (almost) universally bad idea. *AI magazine* **10**(4):40–44, Winter 1989.

[Genesereth, 1983] M. R. Genesereth. An overview of metalevel architectures. In *Proceedings of the Third National Conference on Artificial Intelligence*, pp. 119–123, Washington, D.C., 1983.

[Genesereth and Nilsson, 1987] M. R. Genesereth and N. J. Nilsson. *Logical foundation of artificial intelligence*. Los Altos, California: Morgan Kaufmann, 1987.

[Good, 1971] I. J. Good. Twenty-seven principles of rationality. In *Foundations of Statistical Inference*, V. P. Godambe and D. A. Sprott (eds.), Toronto: Holt, Rinehart and Winston, 1971.

[Haussler, 1990] D. Haussler. Probably approximately correct learning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 1101–1108, Boston, Massachusetts, 1990.

[Hayes-Roth *et al.*, 1989] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett and A. Siever. Intelligent monitoring and control. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 243–249, Detroit, Michigan, 1989.

[Hendler, 1989] J. A. Hendler. Real-time planning. In *Proceedings of the AAAI Spring Symposium on Planning and Search*, Stanford, California, 1989.

[Hendler *et al.*, 1990] J. A. Hendler, A. Tate and M. Drummond. AI planning: Systems and techniques. *AI Magazine* **11**(2):61–77, Summer 1990.

[Horvitz, 1987] E. J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the 1987 Workshop on Uncertainty in Artificial Intelligence*, Seattle, Washington, 1987.

[Horvitz *et al.*, 1989a] E. J. Horvitz, H. J. Suermondt and G. F. Cooper. Bounded conditioning: Flexible inference for decision under scarce resources. In *Proceedings of the 1989 Workshop on Uncertainty in Artificial Intelligence*, pp. 182–193, Windsor, Ontario, 1989.

[Horvitz *et al.*, 1989b] E. J. Horvitz, G. F. Cooper and D. E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 1121–1127, Detroit, Michigan, 1989.

[Horvitz and Breese, 1990] E. J. Horvitz and J. S. Breese. *Ideal partition of resources for metareasoning*. Technical Report KSL-90-26, Stanford Knowledge Systems Laboratory, Stanford, California, 1990.

[Horvitz and Rutledge, 1991] E. J. Horvitz and G. Rutledge. Time-dependent utility and action under uncertainty. In *Proceedings of Seventh Conference on Uncertainty in Artificial Intelligence*, pp. 151–158, Los Angeles, California, 1991.

[Howard, 1966] R. A. Howard. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, **SSC-2**(1):22–26, 1966.

[Johnson, 1992] D. S. Johnson. The NP-completeness column: An ongoing guide. To appear in *J. Algorithms*, 1993.

[Karp, 1972] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher (eds.), pp. 85–103, New York: Plenum Press, 1972.

[Karp, 1990] R. M. Karp. *An Introduction to Randomized Algorithms*. Technical Report TR-90-024, International Computer Science Institute, Berkeley, California, 1990.

[Korf, 1985] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* **27**: 97–109, 1985.

[Korf, 1987] R. E. Korf. Real-time heuristic search: First results. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 133–138, Seattle, Washington, 1987.

[Korf, 1988] R. E. Korf. Real-time heuristic search: New results. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 139–144, Minneapolis, Minnesota, 1988.

[Korf, 1990] R. E. Korf. Real-time heuristic search. *Artificial Intelligence* **42**(3): 189–212, 1990.

[Kutlukan *et al.*, 1992] E. Kutlukan, S. N. Dana and V. S. Subrahmanian. When is planning decidable? In *Proceedings of the First International Conference on AI Planning Systems*, pp. 222–227, College Park, Maryland, 1992.

[Laffey *et al.*, 1988] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao and J. Y. Read. Real-time knowledge based systems. *AI Magazine* **9**(1):27–45, Spring 1988.

[Latombe, 1991] J. Latombe. *Robot Motion Planning*. Boston: Kluwer Academic, 1991.

[Lawler *et al.*, 1987] E. L. Lawler *et al.* (eds.). *The traveling salesman problem: a guided tour of combinatorial optimization*. New York: Wiley, 1987.

[Lesser *et al.*, 1988] V. Lesser, J. Pavlin and E. Durfee. Approximate processing in real-time problem-solving. *AI Magazine* **9**(1):49–61, Spring 1988.

[Levesque, 1986] H. J. Levesque. Making believers out of computers. *Artificial Intelligence*, **30**(1):81–108, 1986.

[Levesque, 1989] H. J. Levesque. *Logic and the complexity of reasoning*. Technical Report KRR-TR-89-2, Department of Computer Science, University of Toronto, Toronto, Ontario, 1989.

[Lin and Kernighan, 1973] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the Traveling Salesman problem. *Operation Research* **21**:498–516, 1973.

[Lin *et al.*, 1987] K. J. Lin, S. Natarajan, J. W. S. Liu and T. Krauskopf. Concord: A system of imprecise computations. In *Proceedings of COMPSAC '87*, pp. 75–81, Tokyo, Japan, 1987.

[Liu *et al.*, 1991] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung and W. Zhao. Algorithms for scheduling imprecise computations. In *IEEE Computer*, **24**:58–68, 1991.

[Lipman, 1989] B. Lipman. How to decide how to decide how to ... Limited rationality in decisions and games. In *Proceedings of AAAI Symposium on AI and Limited Rationality*, Stanford, California, 1989.

[Lozano-Pérez and Brooks, 1984] T. Lozano-Pérez and R. A. Brooks. In *Solid Modeling by Computers*, M. S. Pickett and J. W. Boyse (eds.), pp. 293–327, Plenum Press, New York, 1984.

[McDermott, 1992] D. McDermott. Robot planning. *AI magazine* **13**(2), Summer 1992.

[Michalski and Winston, 1986] R. S. Michalski and P. H. Winston. Variable precision logic. *Artificial Intelligence* **29**(2):121-146, 1986.

[Mitchell, 1990] T. Mitchell. Becoming increasingly reactive. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 1051–1058, Boston, Massachusetts, 1990.

[Moiin and Smith, 1992] H. Moiin and P. M. M. Smith. Better late than never. Department of Electrical and Computer Engineering, University of California, Santa Barbara, (personal communication).

[Nau, 1983] D. S. Nau. Pathology in game trees revisited and an alternative to minimaxing. *Artificial Intelligence,* **21**:221-244, 1983.

[Nilsson, 1991] N. J. Nilsson. Logic and artificial intelligence. *Artificial Intelligence* **45**:31–56, 1991.

[Ogasawara and Russell, 1993] G. H. Ogasawara and S. J. Russell. Planning using multiple execution architectures. To appear in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1993.

[Parr *et al.*, 1992] R. Parr, S. J. Russell and M. Malone. *The RALPH System*. Computer Science Division, University of California, Berkeley, (unpublished manuscript).

[Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Los Altos, California: Morgan-Kaufmann, 1988.

[Pos, 1992] A. Pos. *Anytime Pragmatic Diagnostic Engine*. Department of Computer Science, University of Twente, The Netherlands, (personal communication).

[Pos, 1993] A. Pos. *Time-Constrained Model-Based Diagnosis*. Master Thesis, Department of Computer Science, University of Twente, The Netherlands, 1993.

[Raiffa and Schlaifer, 1961] H. Raiffa and R. Schlaifer. *Applied Statistical Decision Theory*. Harvard University Press, 1961.

[Ralston and Rabinowitz, 1978] A. Ralston and P. Rabinowitz. *A first course in numerical analysis*. 2nd ed., New York: McGraw-Hill, 1978.

[Reiter, 1987] R. Reiter. Nonmonotonic reasoning. *Annual Review of Computer Science,* **2**:147–186, Palo Alto, California: Annual Reviews Inc, 1987.

[Robert, 1986] F. Robert. *Discrete Iterations: A Metric Study*. New York: Springer-Verlag, 1986.

[Russell, 1989] S. J. Russell. Execution architectures and compilation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan, 1989.

[Russell and Wefald, 1989a] S. J. Russell and E. H. Wefald. On optimal game-tree search using rational meta-reasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 334–340, Detroit, Michigan, 1989.

[Russell and Wefald, 1989b] S. J. Russell and E. H. Wefald. Principles of metareasoning. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, R.J. Brachman *et al.* (eds.), San Mateo, California: Morgan Kaufmann, 1989.

[Russell, 1991] S. J. Russell. An architecture for bounded rationality. In *Proceedings of the AAAI Spring Symposium on Integrated Architectures for Intelligent Agents*, Stanford, California, 1991.

[Russell and Wefald, 1991] S. J. Russell and E. H. Wefald. *Do the Right Thing: Studies in limited rationality*. Cambridge, Massachusetts: MIT Press, 1991.

[Russell and Zilberstein, 1991] S. J. Russell and S. Zilberstein. Composing real-time systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 212–217, Sydney, Australia, 1991.

[Russell and Subramanian, 1993] S. J. Russell and D. Subramanian. On provably RALPHs. In E. Baum (Ed.) *Computational Learning and Cognition: Proceedings of the Third NEC Research Symposium*, SIAM Press, 1993.

[Russell *et al.*, 1993] S. J. Russell, D. Subramanian and R. Parr. Provably bounded optimal agents. To appear in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1993.

[Sacerdoti, 1974] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* **5**, 1974.

[Sahni and Gonzalez, 1976] S. Sahni and T. Gonzalez. P-complete approximation problems. *J. Assoc. Comput. Mach.* **23**:555–565, 1976.

[Schoppers, 1987] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 1039–1046, Milan, Italy, 1987.

[Schoppers, 1989] M. J. Schoppers. In defense of reaction plans as caches. *AI magazine* **10**(4):51–60, Winter 1989.

[Shih *et al.*, 1989] W-K. Shih, J. W. S. Liu and J-Y. Chung. Fast algorithms for scheduling imprecise computations. In *Proceedings of the Real-Time Systems Symposium*, pp. 12–19, IEEE, 1989.

[Shih *et al.*, 1991] W-K. Shih, J. W. S. Liu and J-Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal on Computing*, **20**(3):537–552, 1991.

[Shoham, 1989] Y. Shoham. Time for action: On the relation between time, knowledge and action. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 954–959, Detroit, Michigan, 1989.

[Simon, 1955] H. A. Simon. A behavioral model of rational choice. *Quarterly Journal of Economics*, **69**:99–118, 1955.

[Simon, 1976] H. A. Simon. On how to decide what to do. In [Simon, 1982].

[Simon, 1982] H. A. Simon. *Models of bounded rationality, Volume 2*. Cambridge, Massachusetts: MIT Press, 1982.

[Smith, 1986] B. C. Smith. Varieties of self-reference. In Halpern, J. (Ed.) *Theoretical Aspects of Reasoning about Knowledge*. Los Altos, California: Morgan Kaufmann, 1986.

[Smith and Liu, 1989] K. P. Smith and J. W. S. Liu. Monotonically improving approximate answers to relational algebra queries. *COMPSAC-89*, Orlando, Florida, 1989.

[Stankovic and Ramamritham, 1989] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A new paradigm for real-time operating systems. *ACM Operating Systems Review*, **23**(3):54–71, 1989.

[Valiant, 1984] L. G. Valiant. A Theory of the Learnable. *Communications of the ACM*, **27**(11):1134–1142, 1984.

[von Neumann and Morgenstern, 1947] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton, New Jersey: Princeton University Press, 1947.

[Vrbsky *et al.*, 1990] S. V. Vrbsky, J. W. S. Liu and K. P. Smith. *An object-oriented query processor that returns monotonically improving approximate answers*. Technical Report UIUCDCS-R-90-1568, University of Illinois at Urbana-Champaign, 1990.

[Vrbsky and Liu, 1992] S. V. Vrbsky and J. W. S. Liu. Producing monotonically improving approximate answers to database queries. In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation,* pp. 72–76, Phoenix, Arizona, 1992.

[Wellman and Doyle, 1991] M. P. Wellman and J. Doyle. Preferential semantics for goals. In *Proceedings of the Ninth National Conference on Artificial Intelligence,* Anaheim, California, 1991.

[Zadeh, 1975] L. A. Zadeh. Fuzzy logic and approximate reasoning. *Synthese* **30**:407-428, 1975.

[Zilberstein, 1991] S. Zilberstein. Integrating hybrid reasoners through compilation of anytime algorithms. In *Proceedings of the AAAI Fall Symposium on Principles of Hybrid Reasoning,* pp. 143-147, Pacific Grove, California, 1991.

[Zilberstein and Russell, 1992a] S. Zilberstein and S. J. Russell. Efficient resource-bounded reasoning in AT-RALPH. In *Proceedings of the First International Conference on AI Planning Systems,* pp. 260–266, College Park, Maryland, 1992.

[Zilberstein and Russell, 1992b] S. Zilberstein and S. J. Russell. Reasoning about optimal time allocation using conditional performance profiles. In *Proceedings of the AAAI-92 Workshop on Implementation of Temporal Reasoning,* pp. 191–197, San Jose, California, 1992.

[Zilberstein and Russell, 1992c] S. Zilberstein and S. J. Russell. Control of mobile robots using anytime computation. In *Proceedings of the AAAI Fall Symposium on Applications of Artificial Intelligence to Real-World Autonomous Mobile Robots,* pp. 200–207, Cambridge, Massachusetts, 1992.

[Zilberstein and Russell, 1992d] S. Zilberstein and S. J. Russell. Constructing utility-driven real-time systems using anytime algorithms. In *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, pp. 6–10, Phoenix, Arizona, 1992.

[Zilberstein and Russell, 1993] S. Zilberstein and S. J. Russell. Anytime sensing, planning and action: A practical model for robot control. To appear in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1993.