SENSITIVITY ANALYSIS FOR COMBINATORIAL OPTIMIZATION

by

Daniel Mier Gusfield

Memorandum No. UCB/ERL M80/22

27 May 1980

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Sensitivity Analysis for Combinatorial Optimization

Daniel Mier Gusfield

Abstract

Sensitivity analysis is the study of how functions, algorithms, or solutions to problems change in response to perturbations or modifications in the problem input or problem structure. The importance of sensitivity analysis is primarily due to the common occurrence of long sequences of problem instances, each instance differing from the others by small or structured modifications of the problem data.

This dissertation focuses on sensitivity analysis for problems arising in combinatorial optimization. We begin with a discussion of how perturbed problems arise, and why sensitivity analysis is important, and then review the major questions and previous results in sensitivity analysis for combinatorial problems.

The original results are in four sets: The first results concern matroid problems where the data is successively modified in a very structured way, allowing a strong characterization of the resulting solutions, and fast algorithms for reoptimization after a data modification. The second result is a fast algorithm to solve the linear parametric programming problem for many combinatorial problems. The key feature of the algorithm is that its running time is polynomial in the size of the output, and hence a large improvement over the parametric simplex method. The third set of results are bounds for certain counting problems in parametric programming. The parametric minimum spanning tree, and parametric shortest path problems are examined in detail, and the results are applied to show certain limitations in the generalized Lagrange multiplier method. The fourth set of results are simple constructions which solve the *net-*

work flow synthesis problem in a way that allows very rapid solutions to several sensitivity analysis questions. The constructions also have several other desirable properties. The dissertation concludes with a few suggestions for further research, listing a few promising open problems.

Acknowledgements

I would like to thank the people who have contributed the most to my formal graduate education at Berkeley: My advisor, Dick Karp, who has been extremely generous with his time, money, and critical attention, and who on several occasions pulled me out of the fire and back into the frying pan; Gene Lawler, who has been a constant supporter and booster; and Mike Sipser, Barbara Simons, and David Lichtenstein, who have been good listeners, good company, and good friends. I would like to thank all the other members of my extended cohort at Berkeley, particularly Jane Hagstrom, Faith Fich, Bob MacGregor, Ravi Kannan, Chip Martel, Silvio Micali, Vijay Vazirani, Bruce Char and Shafi Goldwasser. I would also like to thank Ilan Adler for serving on my committee, and K.G. Murty of the University of Michigan for several stimulating conversations on sensitivity analysis.

Table of Contents

Chapter I: INTRODUCTION TO SENSITIVITY ANALYSIS FOR COMBINATORIAL OPTIMIZATION

Sensitivity analysis is the study of how functions, algorithms, or solutions to problems change in response to perturbations or modifications in the problem input or problem structure. The importance of sensitivity analysis is primarily due to occurrences of long sequences of problem instances, each instance differing from the others by small or structured modifications of the problem data.

The importance of sensitivity analysis is well recognized in numerical computing, but has been given very little attention in combinatorial computing. This dissertation focuses on sensitivity analysis for problems arising in combinatorial optimization. The structure of the dissertation is the following:

In chapter I, the introduction, we begin with a discussion of how perturbed problems arise, and why sensitivity analysis is important. We then categorize some common forms of sensitivity analysis, and some general research approaches. Finally we survey previous results and methods in the area of sensitivity analysis for combinatorial optimization.

Chapter II begins the original work of the dissertation. We examine problems which arise when many successive matroid optimization problems are solved, each problem instance differing from the others in a highly structured way, allowing strong results and fast optimization methods.

Chapter III presents algorithms for determining all optimal solutions to problems in which the problem data is given as linear functions of a variable λ. The *parametric programming problem* is to find all the optimal solutions as a function of λ. The main result of the chapter is a method which solves the parametric programming problem on-line in polynomial time per solution, for many combinatorial problems. This is an improvement over previous methods, and leads to polynomial solutions of other optimization problems. One example, a problem in program module distribution, is included in chapter III.

Chapter IV discusses the number of solutions possible in various parametric programming problems. The main results are upper bounds for the parametric minimum spanning tree and shortest path problems. We use these results to show the limitations of the generalized Lagrange multiplier method for certain problems. We also detail some unsuccessful efforts at improving the bounds for the parametric minimum spanning tree problem.

Chapter V presents a new solution to the *Network Synthesis Problem*. The solution allows very rapid answers to certain sensitivity analysis questions, and has several other desirable properties.

Chapter VI mentions some open questions and promising areas for future research.

1.1 WHY SENSITIVITY ANALYSIS?

The naive view of the way that algorithms for combinatorial optimization are used is that each call on the algorithm presents a problem instance which is unrelated to past or future calls on the algorithm. Further, the data presented to the algorithm is viewed as correct and complete, and the algorithm is viewed as returning optimal solutions which are usable without further interaction.

In practice however, it is common that combinatorial algorithms are components of larger systems, and are often used to solve long sequences of closely related problem instances, each instance differing from the others by small or structured modifications of the problem data. Tools for sensitivity analysis are useful to deal with such sequences of repeated modifications and reoptimizations.

In the next pages we survey several reasons for long sequences of successive modifications and reoptimizations. The first three arise from naturally occurring applications, and the others from mathematical programming.

1.1.1 Naturally occurring problem sequences

1. **Inexact data or model:** In practice, problem data is often not known exactly, and optimization models rarely capture all the problem constraints and relationships. Results based on a single optimization therefore can't be completely trusted. Instead, the optimization problem is repeatedly solved with different choices of data, or different choices of the model. The optimization algorithm is used to search for *robust* solutions, or to give a picture of the sensitivity of the solution to different data or model assumptions. In many applications, the use of an optimization algorithm for optimization is secondary to its use in contingency planning.

A recent advertisement [SCI] by IBM highlights the use of optimization algorithms for sensitivity analysis. The application is the design of large tractors, using computers to model the performance of different design proposals. The advertisement ends with the following quote: "While these computer programs can't be expected to describe the physical product with absolute precision, ... they are good at reflecting change, showing clearly the *sensitivity* of important performance parameters to variations in the design".

2. **Changing data or model:** Even if the model and data are exact, in many applications they are subject to change; the problem may itself be dynamic and

changing. Consider, for example, routing in computer networks. Message routes must be repeatedly determined because the congestion in the network changes, or certain nodes or links fail, or additional capacity is added. The dynamic nature of many applications requires repeated reoptimization after changes in the problem description.

3. **Conflicting objectives:** In many applications there is more than one objective function, and often these different objectives conflict. There may be no "optimal solution", but rather many solutions which are "good" with respect to all the objectives. Repeated calls on the optimization algorithm are used to find good solutions, or to reflect the sensitivity of the solutions to different weightings or orderings of the objective functions.

1.1.2 Problem sequences in mathematical programming

4. **Heuristics for NP - hard problems:** Most of the combinatorial optimization problems of greatest interest are NP - hard, and therefore no efficient exact solution methods for them are known, and none are likely. In practice these problems are often tackled by embedding fast algorithms for related problems in the inner loop of a larger program. The fast algorithms are called repeatedly with successively modified data. Four common heuristic methods are of this type:

a. **Interactive optimization:** Most problems in network *design* are NP - hard, while many useful problems in network *analysis* are polynomial. Then one heuristic for such design problems is to successively propose a design, analyze it using fast algorithms, propose a modification to the design, and so on until an acceptable design is arrived at. The modifications are done by a human, and the repeated analysis is done by fast computer programs.

b. **Hill climbing:** Hill climbing also involves successive solution proposals, analyses, and modifications. In contrast to interactive optimization, however, the process is completely program controlled. Given a proposed solution, the inner loop analyzes its cost or feasibility and finds a change which improves the value of the objective function or removes some violated condition. Hill climbing methods can be quite involved, as in Shen Lin's [SHE] 3-opting for the travelling salesman problem.

c. **Branch and bound:** Branch and bound is a commonly proposed method for NP - hard problems. Bounds are successively obtained by calling a fast algorithm for a relaxed version of the NP - hard problem. Often each call presents a problem instance which differs from the preceding instance by only a small modification of the data. For example in the travelling salesman problem, the bounding problem is often a derivative of bipartite or general matching, or is a close relative of the minimum spanning tree problem. In the matching case, the successive problems differ by the insertion or deletion of a few edges, or by the modification of a few edge weights. In the spanning tree case, the problems differ by the modification of edge weights, or by the imposition of degree constraints on nodes in the tree.

d. **Lagrangian relaxation:**

In this method, a hard problem is tackled by solving a sequence of easy problems which result from the relaxation of certain constraints. Terms are added to the objective function to reflect violations of any of the constraints, and each term i is multiplied by a variable λ_i. A sequence of such problems is solved, each with different values of λ_i, in an attempt to home in on a solution to the original hard problem [SHA].

The abstract from a recent paper "Optimal Set Partitioning, Matchings and Lagrangian Duality" [NEM] illustrates this perfectly: "We formulate the set partitioning problem as a matching problem with simple side constraints. As a result we obtain a Lagrangian relaxation of the set partitioning problem in which the primal problem is a matching problem. To solve the Lagrangian dual we must solve a *sequence* of matching problems each with different edge weights... successive matching problems differ in only two edge-weights. This enables us to use *sensitivity analysis* to modify one optimal matching to obtain the next one."

5. **Decomposition:** In this method, large mathematical programming problems are decomposed into several smaller subproblems, and each is solved separately. Generally, these separate solutions don't immediately combine to solve the larger problem, but the subproblems can be successively modified and resolved in ways that lead to a solution for the larger problem.

6. **Problems that reduce to parametric programming:** A large class of integer programming problems can be reduced in non-trivial ways to parametric programming problems. More will be said on this in the section on minimum ratio optimization.

1.2 GENERAL RESEARCH APPROACHES

With the perspective that long sequences of closely related problem instances are common, what kinds of results can be obtained to improve sensitivity analysis methods, to speed up successive computations, and to better understand such sequences of problems? We list a few research approaches, noting the ones included in this thesis. Several of these approaches will be further illustrated in section 1.3 on previous results.

1. Factoring and preprocessing

One approach is to identify a part of the problem that remains constant over the sequence of problem instances. By preprocessing the constant part, it may be possible to speed up the sequence of computations. When the successive computations must be done on-line, or in little space, any improvement due to preprocessing can be extremely valuable. Chapter II gives a preprocessing method for a class of problems that arise in matroid optimization.

2. Interval optimization

Combinatorial optimization methods are usually developed to solve problems for a single data point. However, it is often possible to develop equally fast methods that solve the problem for a given data point, and in addition return intervals of data such that the solution remains optimal for any data point in the interval. Chapter V presents such an algorithm for the network synthesis problem.

3. Successively improving complexity

It is often possible to develop methods that use part of the work of one computation to speed up the computations that follow it. The key problems are how to recognize what can be used, and how to balance storage against recomputation. In chapter III we present a method for parametric programming in which each computation uses as much information as possible from the preceding computations.

4. Structural results

A very important research approach is to establish results relating the structure of the data of a problem to the structure of the solutions, i.e. characterize the form of the solutions given the form of the data. Such characterizations can be extremely valuable in identifying what kinds of data changes affect the optimal solutions, and in what way, and hence can speed up recomputation after data changes, or indicate that no recomputation is necessary. The matroid and the network flow synthesis results of chapters II and V are each partly structural.

5. Transparent algorithms

For purposes of sensitivity analysis, it is desirable to have algorithms look like *transparent* boxes rather than black boxes. We want a clear view of what the algorithm does to the input in producing its output. The output can then be written in terms of the input, and certain sensitivity analysis questions can be answered very efficiently. In chapter V we present a transparent algorithm for the network flow synthesis problem which permits extremely fast re-optimization after certain modifications, whereas the previous published algorithms require recomputation.

6. Complexity and counting results

Complexity results address the question of how much one must pay for sensitivity analysis. The thrust of five approaches above has been to solve a problem and, at little extra expense, get information which will speed up the solution of other related problems. Limitations exist on how much information is attainable without major increase in cost, and results in this area should be possible. More will be said on this in the next section.

Counting results answer the question "From a class of distinct inputs for a function or algorithm, how many distinct outputs are possible?" Counting results are useful if enumeration of all distinct outputs is desired, and can establish the complexity of certain optimization algorithms that involve enumeration. Chapter IV is devoted to counting questions where the input is generated by a parameter λ.

1.3 PREVIOUS RESULTS

In this section we survey previous papers and textbook methods for sensitivity analysis for combinatorial optimization. The last four methods, parametric linear programming, the Eisner - Severance technique, the fractional linear programming method, and Meggido's method, will be needed in the sequel, and are given in greater detail.

1.3.1 Incremental Computing

Several papers have addressed the problem of updating the solution to a combinatorial optimization problem after some local modification of the problem description. For graph problems, a local modification is the insertion or deletion of new nodes or edges, or the modification of edge costs. In geometric computing, a local modification is the insertion or deletion of new points in some space.

Chin and Houck [CHI] and Spira and Pan [SPI] present $O(n)$ algorithms to update the minimum spanning tree after the addition of one new node and edges incident with it. Even and Shiloach [EVE] give an algorithm for the on-line maintenance of the connected components of a graph as edges are successively deleted. The complexity for q deletions is $O(q+ne)$ where n is the number of nodes and e the number of edges in the initial graph. A paper by Webber [WEB] gives an $O(n^2)$ method to update an optimal matching after the modification of one edge weight, and this method is used in [NEM]. A paper by Goto and Vincentelli [GOT] gives an updating method for the shortest path problem. A recent thesis by Cheston [CHE] examines the advantages of updating methods verses recomputation from scratch after a local modification. For the problems examined in that thesis, recomputation was faster than naive updating methods. However, there may be more clever ways to do updating, and so the question remains open.

In the area of geometric computing, there has been recent work in dynamically maintaining information about geometrical figures, as parts of the figure are modified. For example, the convex hull of a set of points is maintained as points are added or deleted. Work in this area has been reported by [VAN1],

[VAN2], [VAN3], and [BEN].

1.3.2 All for one results

Suppose T is the minimum spanning tree of a weighted graph G, and e is an edge of G in T. The *e deletion problem* P_e is: Given T, determine the minimum spanning tree of G - {e}. The *all deletion problem* P_a is: Given T, solve problem P_e (n - 1) times, once for each edge e in T.

If problem P_e can be solved in time q, then problem P_a can be solved naively in time (n-1)q. By a more clever approach, problem P_a can be solved as quickly as problem P_e, i.e. in time $O(n^2)$. Such results are called "all for one" results, and have been obtained for minimum spanning tree and shortest path problems [SHI],[CHI],[CHE].

1.3.3 Efficient solutions in bi-criteria optimization

If P is a problem with two objective functions, then a solution x is called efficient if there is no other solution y which is better than x in both criteria. If the two criteria conflict or are incomparable, then all efficient solutions are often desired. Much work has been done in this area for certain optimization problems, but very little in combinatorial optimization. One paper [VW] examines the problem of two conflicting objectives in a simple scheduling problem. The paper presents a pseudo-polynomial algorithm for this problem, but an $O(n^2)$ algorithm is easily obtained. Thuente [THU] looks at the bi-criteria problem for shortest paths.

1.3.4 Unrestricted parametric programming

One very general form of parametric programming is to let the problem costs be completely variable, and then to solve the problem as a function of the variable costs. This is generally an intractable task, due simply to the size of the output, but in restricted forms it can be useful. Somers [SOM] and Walters [WAL] have done work in this area. Somers examines network flow problems, but the ideas extend to other problems as well [MU]. We illustrate her approach with the s to t shortest path problem.

Let $M(\delta_i)$ be the shortest path from s to t when edge i has cost $\delta_i \geq 0$, and all other edges have known fixed costs. Then $M(\delta_i) = \mathrm{MIN}[\ M(0) + \delta_i,\ M(\infty)\ ]$. When $M(0) + \delta_i < M(\infty)$, $M(0) + \delta_i$ is the minimum cost of all paths which go through edge i, and $M(\infty)$ is the minimum cost of all paths which don't.

This approach generalizes naturally to more than one variable edge cost. Let δ_i and δ_j be the variable edge costs of two edges i and j, and let $M[\delta_i,\delta_j]$ be the cost of the shortest s to t path when edge i is set to δ_i and edge j is set to δ_j. Then $M[\delta_i,\delta_j] = \mathrm{MIN}[\ M(0,0) + \delta_i + \delta_j,\ M(0,\infty) + \delta_i,\ M(\infty,0) + \delta_j,\ M(\infty,\infty)\ ]$.

If there are k edges with variable costs, then the shortest path expression is the minimum of 2^k sub-expressions. This method is then essentially enumeration over all way to include or exclude the k edges in the path selection. For computing purposes, this method is very unsatisfactory unless k is very small.

1.3.5 Complexity and bad examples

Several results in the complexity of sensitivity analysis have been reported. A paper by Spira and Pan [SPI] addresses the complexity of incremental computing for minimum spanning tree problems. Jerislow [JER] shows an NP - completeness result for a parametric optimization problem. Murty [MU2] gives an example showing that the parametric linear programming problem can have an exponential number of optimal solutions. This result is implicit in the earlier work of Zadeh [ZAD] in which he gives exponential examples for the network simplex and other minimum cost flow methods. By parameterizing the lower bound of one edge in his example, an exponential parametric programming example is obtained. Rosenthal [ROS1] [ROS2] examines sensitivity analysis in problems related to matrix multiplication, and obtains optimal algorithms for a certain model of complexity. Johnson [JOH] gives a clever reduction showing that the problem of finding the k-th cheapest spanning tree of a weighted graph is NP - complete.

1.3.6 Parametric Linear Programming

Let A be an m × n matrix, c an n length vector, and b an m length vector. The linear programming problem is then to find an n length real vector x to minimize $c^t x$ such that $Ax \leq b$, $x \geq 0$.

Associated with each decision variable x_i are two costs S(i) and T(i). Then the function P(λ) is defined as:

Given a value for the variable λ, P() = the minimum value of

$$\sum_i S(i)x_i + \lambda \sum_i T(i)x_i$$

such that $Ax \leq b$, $x \geq 0$.

The parametric linear programming problem is to determine P(λ) as a function of λ. Note that P(λ) is a piecewise linear convex function of λ. Points of discontinuity of P(λ) are called *breakpoints*.

The Parametric Simplex Method

The textbook method for determining P(λ) is the **Parametric Simplex Method** [MU]. Let basis B be an m × m full rank submatrix of A. Let $A_{.i}$ denote the ith column of A, and if column i is in B then the variable x_i is called a basic variable. Let S_B and T_B be m length vectors containing the S and T costs of the basic variables, and let B^{-1} be the inverse matrix of B. Then the cost, as a function of λ, of variable i with respect to basis B, denoted C(i,B,λ) is

$$S(i) - S_B B^{-1} A_{.i} + \lambda[T(i) - T_B B^{-1} A_{.i}]$$

Suppose basis B is proved optimal by the simplex method for the value of λ equal to λ'. We want to determine the maximum value of $\lambda \geqq \lambda'$ for which B is optimal. The parametric simplex method determines this by successively solving the following problem: What is the maximum value of $\lambda \geqq \lambda'$ for which the simplex method *proves* the optimality of B? i.e. What is the largest value of $\lambda \geqq \lambda'$ such that $C(i,B,\lambda) \geqq 0$ for all i?

The parametric simplex method determines $P(\lambda)$ for $\lambda \geqq 0$ as follows:

1. Set $\lambda' = 0$ and compute $P(0)$. Let B be the optimal basis. Then $C(i,B,0) \geqq 0$ for all i.

2. Find λ^*, the smallest value of $\lambda > \lambda'$ such that $C(i,B,\lambda^*) \geqq 0$ for all i, and $C(i,B,\lambda^* + \varepsilon)$ for some i and all $\varepsilon > 0$. If no such λ^* exists, then B is optimal for all values of λ between λ' and ∞. In that case, output B and ∞ and stop. Else if i is a variable such that $C(i,B,\lambda^* + \varepsilon) < 0$ for all $\varepsilon > 0$, then a new basis B^* is created by pivoting in variable i. Note that $C(j,B,\lambda) = 0$ for any λ if j is a basic variable in B, and so the above variable i must be non-basic.

3. Output B and λ^*; Set B to B^* and λ' to λ^* and go to 2.

Problems with the Parametric Simplex Method

The major problem with this method is that pivots in step 2 do not necessarily determine breakpoints of $P(\lambda)$. In fact in most applications, very few of the pivots determine breakpoints,and so we can't relate the amount of work (number of pivots) of the algorithm to the number of breakpoints.

The problem is due to primal degeneracy, and the fact that for a given value of λ, a primal optimal solution may not be dual feasible. Therefore, the parametric simplex method may make many degenerate pivots before making a non-degenerate pivot corresponding to a real breakpoint of $P(\lambda)$. To make this point clear, consider the parametric s to t shortest path problem is a graph G.

One formulation of this problem as a linear program is as a minimum-cost flow problem where s has a supply of one unit, t has a demand of one unit, and all other nodes have no supply or demand. This is a highly degenerate problem since the optimal basis will be a tree of shortest paths from s to all other nodes in G, and the flow on most edges of the tree will be zero. As λ varies, the basis will change whenever a path from s to any other node is no longer shortest, even though the shortest path to t is unaffected. Hence we do not have a polynomial bound on the number of pivots needed by the parametric simplex method to find one breakpoint in the s to t shortest path problem.

If the linear programming problem is totally non-degenerate, then every pivot of the parametric simplex method will yield a different primal solution. However, the method still suffers from a second form of degeneracy. Even if every pivot is non-degenerate, the method may do many pivots for one breakpoint because there may be many different solutions which are all optimal at only one point (see figure 1.1).

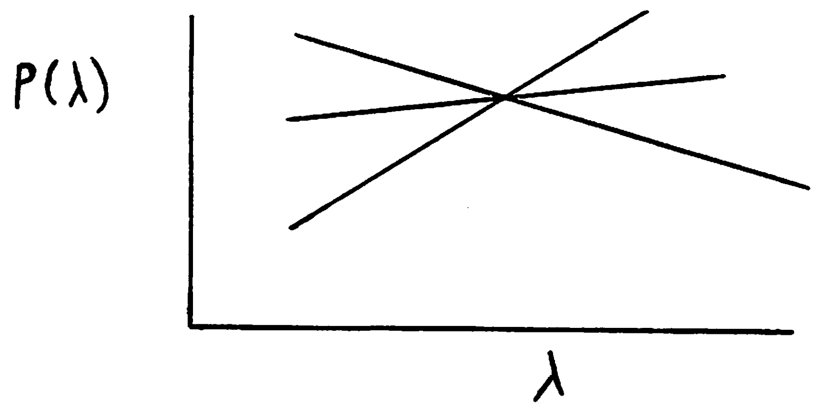


Figure 1.1:

There is no mechanism in the parametric simplex method to keep it from discovering multiple solutions at a given value of λ.

In the case of multi-dimensional parametric programming, the degeneracy problem becomes even more severe. In an m - parameter problem, a primal solution is optimal in a convex region C of m - dimensional space. However, each basis may be *proven* optimal by the simplex method for only some sub-region of C. Then, in order to find C, the subregions corresponding to the same primal solution must be pieced together. The thesis of Walters [WAL], and the paper of Haneveld, Meer and Peters [HAN] are primarily concerned with this problem.

In chapter III we will present parametric methods for many combinatorial problems which completely avoid the problems of degeneracy.

1.3.7 The Eisner and Severance Technique

Let P be an unparametrized problem, and let X be the set of all feasible solutions to P. Associated with every decision variable i are two costs S(i) and T(i). Define for each $x \in X$,

$$S(x) \equiv \sum_i S(i) x_i \quad \text{and} \quad T(x) \equiv \sum_i T(i) x_i$$

The function P(λ) is the following: For a given value of λ, P(λ) = minimum value of S(x) + λT(x) taken over all $x \in X$.

The parametric problem P(X,λ) is the problem of determining P(λ) as a function of λ. Note that P(λ) is a piecewise linear convex function of λ. Points of discontinuity of P(λ) are called *breakpoints.*

The following method determines P(λ) for any problem P(X,λ), assuming P(λ) has a finite number of breakpoints. Further, if P(λ) contains k breakpoints between the minimum (λmin) and maximum (λmax) ranges of λ, then the method needs to evaluate P(λ) at most 2k times. We call the method the Eisner - Severance (ES) method because a version of it can be found in [ES]. However, similar versions can be found in [NAU], [GEO], and it is not clear where the method originated. The method presented here differs from the above versions in the use of a stack which allows a clean proof of correctness and timing.

Suppose basis B is proved optimal by the simplex method for the value of λ equal to λ'. We want to determine the maximum value of $\lambda \geqq \lambda'$ for which B is optimal. The parametric simplex method determines this by successively solving the following problem: What is the maximum value of $\lambda \geqq \lambda'$ for which the simplex method *proves* the optimality of B? i.e. What is the largest value of $\lambda \geqq \lambda'$ such that $C(i,B,\lambda) \geqq 0$ for all i?

The parametric simplex method determines $P(\lambda)$ for $\lambda \geqq 0$ as follows:

1. Set $\lambda' = 0$ and compute $P(0)$. Let B be the optimal basis. Then $C(i,B,0) \geqq 0$ for all i.

2. Find λ^*, the smallest value of $\lambda > \lambda'$ such that $C(i,B,\lambda^*) \geqq 0$ for all i, and $C(i,B,\lambda^* + \varepsilon)$ for some i and all $\varepsilon > 0$. If no such λ^* exists, then B is optimal for all values of λ between λ' and ∞. In that case, output B and ∞ and stop. Else if i is a variable such that $C(i,B,\lambda^* + \varepsilon) < 0$ for all $\varepsilon > 0$, then a new basis B^* is created by pivoting in variable i. Note that $C(j,B,\lambda) = 0$ for any λ if j is a basic variable in B, and so the above variable i must be non-basic.

3. Output B and λ^*; Set B to B^* and λ' to λ^* and go to 2.

Problems with the Parametric Simplex Method

The major problem with this method is that pivots in step 2 do not necessarily determine breakpoints of $P(\lambda)$. In fact in most applications, very few of the pivots determine breakpoints,and so we can't relate the amount of work (number of pivots) of the algorithm to the number of breakpoints.

The problem is due to primal degeneracy, and the fact that for a given value of λ, a primal optimal solution may not be dual feasible. Therefore, the parametric simplex method may make many degenerate pivots before making a non-degenerate pivot corresponding to a real breakpoint of $P(\lambda)$. To make this point clear, consider the parametric s to t shortest path problem is a graph G.

solution is optimal in a convex region C of m - dimensional space. However, each basis may be *proven* optimal by the simplex method for only some sub-region of C. Then, in order to find C, the subregions corresponding to the same primal solution must be pieced together. The thesis of Walters [WAL], and the paper of Haneveld, Meer and Peters [HAN] are primarily concerned with this problem.

In chapter III we will present parametric methods for many combinatorial problems which completely avoid the problems of degeneracy.

1.3.7 The Eisner and Severance Technique

Let P be an unparametrized problem, and let X be the set of all feasible solutions to P. Associated with every decision variable i are two costs S(i) and T(i). Define for each x ∈ X,

$$S(x) \equiv \sum_i S(i)x_i \text{ and } T(x) \equiv \sum_i T(i)x_i$$

The function P(λ) is the following: For a given value of λ, P(λ) = minimum value of S(x) + λT(x) taken over all x ∈ X.

The parametric problem P(X,λ) is the problem of determining P(λ) as a function of λ. Note that P(λ) is a piecewise linear convex function of λ. Points of discontinuity of P(λ) are called *breakpoints*.

The following method determines P(λ) for any problem P(X,λ), assuming P(λ) has a finite number of breakpoints. Further, if P(λ) contains k breakpoints between the minimum (λmin) and maximum (λmax) ranges of λ, then the method needs to evaluate P(λ) at most 2k times. We call the method the Eisner - Severance (ES) method because a version of it can be found in [ES]. However, similar versions can be found in [NAU], [GEO], and it is not clear where the method originated. The method presented here differs from the above versions in the use of a stack which allows a clean proof of correctness and timing.

The Eisner and Severance Technique

1. Evaluate $P(\lambda min)$. Let $L \equiv \{S(L), T(L)\}$ denote the S and T costs of the optimal solution at λmin. Output λmin and L.

2. Evaluate $P(\lambda max)$. Let $R \equiv \{S(R), T(R)\}$ denote the S and T costs of the optimal solution at λmax. Set λ_R to λmax.

3. Set λ^* to be the intersection point of the solutions L and R. i.e. $S(L) + \lambda^* T(L) = S(R) + \lambda^* T(R)$.

3a. If $\lambda^* = \lambda_R$ go to 4bi.

3b. Evaluate $P(\lambda^*)$ and let $C \equiv \{S(C), T(C)\}$ denote the cost of the optimal solution at λ^*.

4a. If L is not optimal at λ^*, then stack R and λ_R on stack SK. One element of SK then consists of the triple $(S(R), T(R), \lambda_R)$. Set R to C; set λ_R to λ^* and go to step 3a.

4b. If L is optimal at λ^* then

i. Output L and λ^*.

ii. Set L to R.

iii. If SK is empty, then output L and λmax and stop. Else pop SK; set R and λ_R to the popped values; go to step 3.

Theorem 1.1: The ES method correctly discovers the function $P(\lambda)$, and does not evaluate $P(\lambda)$ at more than 2k points, where k is the number of breakpoints of $P(\lambda)$ between λmin and λmax.

Proof: First we prove that the ES method works correctly. Note that at any point in the computation, λ_R, the known optimality point of the current R, is to the left of all values of λ which are stacked on SK. Further, R is not optimal at any of the stacked values of λ. Hence until step 4b is executed for the first time, the algorithm is generating distinct solutions which lie on $P(\lambda)$. The number of breakpoints, hence solutions on $P(\lambda)$, is finite and so the algorithm must eventually execute step 4b. At that point, the output L is optimal for a range of λ beginning on the left with λmin, and ending on the right with λ^*.

After 4biii is executed, either all of $P(\lambda)$ has been identified, or the state of the computation is identical to a state occurring if λmin had been set initially to λ^*. In the latter case, the correctness of the algorithm follows from the above argument for the original λmin.

Now we examine the number of evaluations of $P(\lambda)$ needed by the ES method. Suppose the test in step 3a is never true. Then we claim that every evaluation of $P(\lambda^*)$ in step 3 leads either to the identification of a previously unidentified breakpoint, or to the generation of a solution which is optimal along a line segment L_s of $P(\lambda)$, such that no previously generated solution is optimal in the interior of L_s. It then follows that no more than 2k evaluations of $P(\lambda)$ are required.

To see the first part of the claim, note that the algorithm never evaluates $P(\lambda)$ twice at the same value of λ. Hence if step 4b follows step 3, a previously unidentified breakpoint is recognized.

To see the second part of the claim, recall that C, generated in step 3, is optimal at λ^*. No solution which is stacked on SK is optimal at λ^*, and neither is any solution which has been unstacked. Therefore, C is the first generated solution which is optimal at λ^*. Now the test of step 3a is never true by assumption, and so λ^* must be in the interior of some line segment L_s of $P(\lambda)$, and no previously generated solution is optimal in the interior of L_s. Hence under the assumption that the test in step 3a is never true, at most 2k evaluations of $P(\lambda)$ are needed to determine all of $P(\lambda)$.

If the test at step 3a is ever true, then not every evaluation of $P(\lambda)$ leads directly to the identification of a breakpoint or a line segment on $P(\lambda)$, and the above argument does not hold. Instead, when $\lambda^* = \lambda_R$ in step 3a, λ_R is recognized as a breakpoint even though $P(\lambda_R)$ may have been evaluated much earlier. The cost of recognizing the breakpoint λ_R is still, however, one evaluation of $P(\lambda)$, since step 3a does not involve an additional evaluation of $P(\lambda)$. Note that $\lambda^* = \lambda_R$ in step 3a only when the earlier evaluation of $P(\lambda_R)$ produced a solution optimal only at λ_R (see figure 1.2).

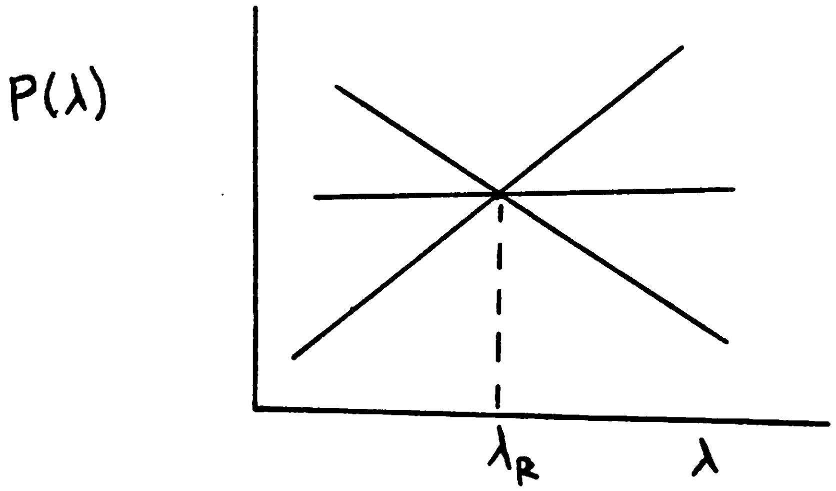


Figure 1.2:

Advantages and Defects of the ES method

The main feature of the ES method is that it determines $P(\lambda)$ for any problem $P(X,\lambda)$ with a linear parametric cost function, and further, it evaluates $P(\lambda)$ at most twice per breakpoint, totally avoiding the problems of degeneracy in the parametric simplex method. If the evaluation of $P(\lambda)$ can be done in polynomial time, then $P(\lambda)$ can be determined in time which is polynomial in the number of breakpoints. However, the ES method often cannot be used to find successive breakpoints on - line in polynomial time. Although the breakpoints are output in left to right order (due to the use of stack SK), the work to find the first breakpoint may be on the order of the work needed to find them all. Further, it is often useful to locate the first breakpoint greater than some value of λ, and the ES method cannot solve this problem efficiently. An additional defect of the method is that it does not generalize nicely to multi- parametric problems where two or more parameters are varied.

In chapter 3 we discuss parametric programming methods which can be used on line, can determine the next breakpoint greater than a given λ, and do generalize to more than one parameter. In contrast to the ES method, however, they do not work for all problems with linear parametric cost functions.

1.3.8 Minimum Ratio Optimization

Given an instance of problem P containing n decision variables, let solution x be an n-length vector, and let X be the set of all solutions to problem P.

For every decision variable i, associate two cost variables $S(i) > 0$ and $T(i) < 0$, and recall that for solution x

$$S(x) \equiv \sum_i S(i)x_i \quad \text{and} \quad T(x) \equiv \sum_i T(i)x_i$$

where x_i is the ith component of vector x.

Problem R(X): The Minimum Ratio Optimization problem for X, denoted R(X) is to find $x \in X$ such that the ratio $S(x)/T(x)$ is minimized over all feasible solutions in X. Let z denote the optimal ratio. The function $P(\lambda)$ is defined as before i.e.

Given a fixed value of λ, $P(\lambda)$ is the minimum of $S(x) + \lambda T(x)$ over all $x \in X$. Recall that $P(\lambda)$ is a piecewise linear convex function of λ. Note that $T(i) < 0$ implies that $P(\lambda)$ is a decreasing function of λ.

We describe two solution methods for problem R. Both rely on successive computations of $P(\lambda)$ for varying values of λ. The idea is that for a given value of λ,

$$z = \lambda \text{ if and only if } P(\lambda) = 0$$

$$z > \lambda \text{ if and only if } P(\lambda) > 0$$

$$z < \lambda \text{ if and only if } P(\lambda) < 0$$

Hence, to solve R(X), it is sufficient to find the value of λ such that $P(\lambda) = 0$. Many papers [MEG] [PIC1] [CHA1] [CHA2] [ORL] have exploited this, reducing specially structured integer programming problems to the minimum ratio problem for some underlying problem P. Picard [PIC2], with other authors, in particular have found numerous integer programming problems that can be solved this way, where P is a network flow model called *provisioning*. Picard's techniques give polynomial algorithms for many integer programming problems, however many of Picard's results can be accelerated by a judicious use of binary search.

We now review two known methods to locate λ such that $P(\lambda) = 0$, and suggest some improvements to the second method.

1.3.9 Fractional Linear Programming

The following method to solve R(X) is used when X is the set of feasible solutions of a general linear programming problem, but can be used for other problems as well.

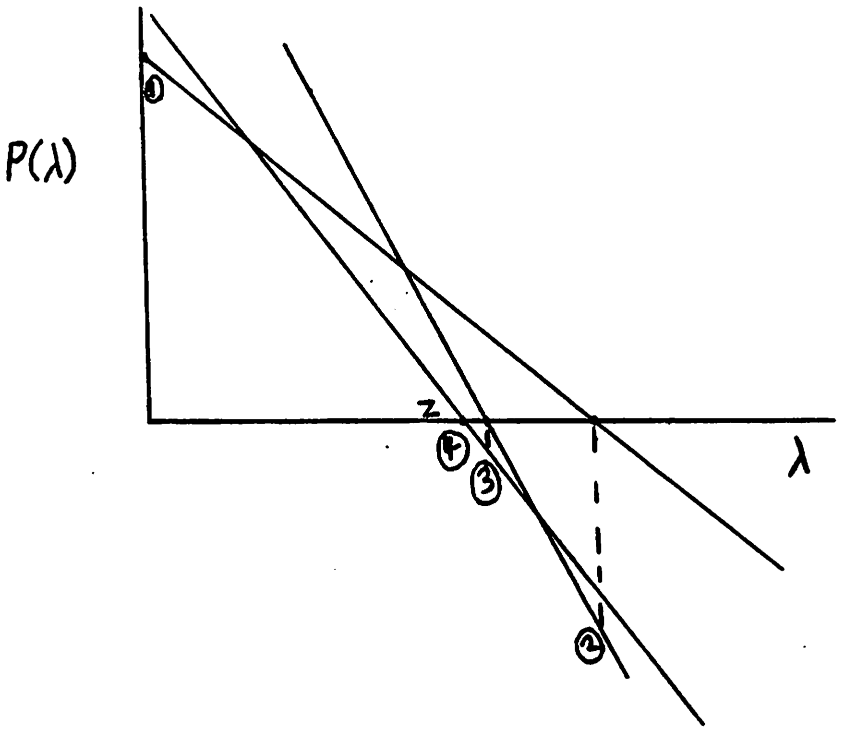
1. Compute P(0) and let x be the resulting optimal solution with costs S(x) and T(x). Set λ^* to $S(x)/T(x)$.

2. Compute $P(\lambda^*)$ and let x be the resulting optimal solution with costs S(x) and T(x).

3. Set λ' to $S(x)/T(x)$. λ' is now the value of λ such that $S(x) + \lambda T(x) = 0$.

4. If $\lambda' < \lambda^*$ then set λ^* to λ' and go to step 2. If $\lambda' = \lambda^*$ then stop. $P(\lambda^*) = 0$, $\lambda^* = z$, and x is the solution to R(X).

Figure 1.3 shows an example of the fractional linear programming method.



Four optimizations used to locate z.

Figure 1.3

Fact: The fractional linear programming method finds λ such that $P(\lambda) = 0$ with at most k+1 evaluations of $P(\lambda)$, where k is the number of breakpoints of $P(\lambda)$. The point is that until z is reached, λ' decreases each time step 2 is executed.

The advantage of this method is that it works for general linear programming problems, and if k is small, it works quickly. The disadvantage is that for problems with a large number of breakpoints, many computations of $P(\lambda)$ may be needed. If k is exponential, then the above bound on the number of evaluations of $P(\lambda)$ cannot yield polynomial bounds on the cost of solving $R(X)$.

1.3.10 Megiddo Method

We now describe a method due to N. Megiddo [Meg] that solves R(X) for a large class of combinatorial problems. Megiddo does not characterize which problems are solvable by this method, but they are easy to recognize. One sufficient condition is the existence of an algorithm for the non-ratio problem P using only the following arithmetic operations: additions and subtractions, and multiplications between one program variable and one constant. Many combinatorial algorithms are of this type.

Let P be the non-ratio problem associated with R(X), and let A be an algorithm that solves P. Assume that the arithmetic in A consists only of additions and subtractions, and multiplications between one program variable and one constant. If we know z, the value of λ such that $P(\lambda) = 0$, then x, the solution of R(X), can be found by using A to compute P(z). The Megiddo method takes the opposite approach. It tries to force A to behave as if z is known, and hence to discover x, even though z is not known until the end of the algorithm. Before giving a formal description of the method we give an intuitive description.

Consider algorithm A written as a *binary decision tree* T. We want to discover the path Q through T that A would take when computing P(z), but we don't know z. The method executes algorithm A symbolicly, carrying along linear forms for the values of the variables. The values can be represented by linear forms since they began as linear forms, and the permitted arithmetic does not change this. When A reaches a branch point in T, it isn't clear which way to branch since z isn't known. What is needed is an oracle to tell the algorithm which way A would branch when computing P(z). Suppose the test in the branch is a comparison between two program variables, say g and h, and let $g(\lambda)$ and $h(\lambda)$ be the linear forms representing the values of g and h at the branch point. The oracle must tell the algorithm whether g(z) is greater, less or equal to h(z).

However, this can be determined as follows: Let λ^* be such that $g(\lambda^*) = h(\lambda^*)$, and compute $P(\lambda^*)$. If $P(\lambda^*) = 0$, then z is λ^*. If $P(\lambda^*) > 0$, then the relationship of g(z) to h(z) is the unique relationship of $g(\lambda)$ to $h(\lambda)$ on the left of λ^*. If $P(\lambda^*) > 0$, then the relationship of g(z) to h(z) is the unique relationship of $g(\lambda)$ to $h(\lambda)$ on the right of λ^*. Hence the proper branch can be made without the explicit knowledge of z.

Formal Description

In order to describe the workings of the Megiddo method we need to describe the structure of algorithm A, and the program variables of A. We assume that algorithm A consists of four types of statements: I/O statements, assignment statements, arithmetic statements, and branch on compare statements. The I/O statements read in or out the value of a program variable. Assignment statements change the value of one program variable. Arithmetic statements are of the form "$g \leftarrow h \bigcirc k$", where $\bigcirc$ is either +, - or ×. In the case of multiply, at most one of h or k is a program variable, the other being a constant. Branch statements are of the form: "if g rel h then branch 1 else branch 2", where "rel" is one of $>, <, =, \leq$, or $\geq$. We define *straight line code* as a sequence of statements not containing any branch statements.

Now we look at the program variables of A. Let V be the set of program variables of A, and let $R \subseteq V$ be the set of variables whose value is ever read in from the problem data. For example, in the shortest path problem, R is the set of program variables representing the costs of the edges in the problem. Let $R^* \subseteq V$ be the set of program variables whose value is effected by some variable in R. That is, R^* is formed as follows: R is in R^*. If v is a program variable on the left-hand side of an assignment or arithmetic statement whose right-hand side contains a variable in R^*, then v is put into R^*. R^* contains no variables not identified in this way. For example, in the Dykstra algorithm, the node labels are in R^* since each node label is set equal to the sum of an edge length plus another node label. In most combinatorial algorithms, $V - R^*$ is the set of decision variables which define the solution x. In the Dykstra algorithm, $V - R^* = x$, where $x_i = 1$ if edge i is in the shortest path tree, and $x_i = 0$ if not. We will assume that in A, no variable in R^* is ever assigned the value of a variable in $V - R^*$, and that no arithmetic operations involve variables in $V - R^*$.

Preliminaries for M(A)

For each program variable $g \in R^*$, create two program variables S_g and T_g, and define $g(\lambda) \equiv S_g + \lambda T_g$, the linear form representing the value of program variable g as a function of λ. Create a single program variable g for every $g \in V - R^*$. "*Simulate straight line code of A*" means that a portion of A containing straight line code is to be simulated as follows:

1. For each input statement "read g", execute the two statements "read S_g, read T_g". For each output statement "write g", execute "write S_g, write T_g" if $g \in R^*$, and execute "write g" if g is in $V - R^*$.

2. For each assignment statement "$g \leftarrow h$" execute "$g \leftarrow h$" if $g,h \in V - R^*$, and execute "$S_g \leftarrow S_h$, $T_g \leftarrow T_h$" if $g,h \in R^*$, and execute "$S_g \leftarrow h$, $T_g \leftarrow 0$" if $g \in R^*$ and h is a constant.

3. For each arithmetic statement "$g \leftarrow h \bigcirc k$" execute "$S_g \leftarrow S_h \bigcirc S_k$, $T_g \leftarrow T_h \bigcirc T_k$" if $g,h,k \in R^*$. If $g,h \in R^*$ and k is a constant, then execute "$g \leftarrow h \pm k$" as "$S_g \leftarrow S_h \pm k$", and execute "$g \leftarrow h \times k$" as "$S_g \leftarrow S_h \times k$, $T_g \leftarrow T_h \times k$".

Algorithm M(A)

Input: Costs for each variable i given as pairs S(i), T(i).

Output: z and x. x is the optimal solution at point z.

1. Denote interval I as [L,R]. Set L to 0 and R to ∞. Note that $z \in I$. Set pc to 1.

2. Simulate the straight line code of A starting at point pc until a branch statement is reached. If the end of A is reached, go to step 6.

3. At a branch statement, two linear forms, $g(\lambda)$ and $h(\lambda)$, representing the values of two program variables g and h, are compared. Solve for λ^*, the unique value of λ such that $g(\lambda^*) = h(\lambda^*)$. If $\lambda^* \in I$, go to step 4. If $\lambda^* \notin I$, then either $g(\lambda) > h(\lambda)$ or $g(\lambda) < h(\lambda)$ for all $\lambda \in I$. This determines whether g(z) is greater or less than h(z) at this branch, and hence determines the correct branch that algorithm A would make in computing P(z). Branch to the appropriate code in A; set pc to the start of that code, and go to step 2.

4. If $\lambda^* \in I$, then for $\lambda \in I$, $g(\lambda) > h(\lambda)$ on one side of λ^*, and $g(\lambda) < h(\lambda)$ on the other. To determine the correct branching of A at this branch statement (i.e. which branch A would take in computing P(z)), compute $P(\lambda^*)$ in a separate running of algorithm A. For clarity, assume that $P(\lambda^*)$ is computed using A', a copy of algorithm A. Then there are three cases:

4i: If $P(\lambda^*) = 0$ then $\lambda^* = z$. Terminate the algorithm.

4ii: If $P(\lambda^*) > 0$ then $z > \lambda^*$. Set L to λ^*, and determine the correct branch for A at this branch statement. Branch to the appropriate code in A; set pc to the start of that code, and return to step 2.

4iii. If $P(\lambda^*) < 0$ then $z < \lambda^*$. Set R to λ^* and determine the correct branching of A. Branch to the appropriate code in A; set pc to the start of that code, and go to step 2.

5. Let $y \in X$ be the solution computed by the actions of A above, and let [L,R] be the final endpoints of I. Then y is the optimal solution for R(X), and $z = S(y)/T(y)$. Set x to y and output x and z.

Complexity of M(A)

The complexity of M(A) is at most the square of the complexity of A; If A contains d branch statements, then the complexity of M(A) is bounded by dC(A), where C(A) is the complexity of A. Hence, if A is polynomially bounded, then M(A) is also polynomially bounded.

Megiddo suggests two techniques for reducing the running time of M(A) on typical combinatorial problems. The first technique is useful when actions at a branch statement do not affect the future execution of A until well after that branch. For example, consider the statements:

"if $g > h$ then branch 1 else branch 2".

branch 1: $g \leftarrow 5 - h$

branch 2: $g \leftarrow 50 + h$

If g is not used again until much later, the determination of the proper assignment can be delayed. In this way, values of λ^* and questions "is z greater, less or equal to λ^*" can be buffered and resolved by binary search over the buffered values. This reduces the number of runs of algorithm A', and hence the running time of M(A). For example, the complexity of the minimum ratio cycle problem is reduced from $O(n^6)$ to $O(n^4 \log n)$ in this way.

Megiddo's second improvement is for problems such as minimum spanning tree problems, in which the A algorithm involves no arithmetic. The details of this improvement can be found in [Meg].

Refinements

We describe two refinements of M(A) which do not improve the order of magnitude of the complexity, but which are cheap to implement and will likely speed up M(A) in practice.

1. Let x be the optimal solution associated with $P(\lambda^*)$ computed in step 4. In step 4iii, R can be set to $S(x)/T(x) < \lambda^*$. See figure 1.4. This accelerates the shrinking of L and may reduce the number of executions of A'. This seems particularly true when used together with the binary search improvement above.

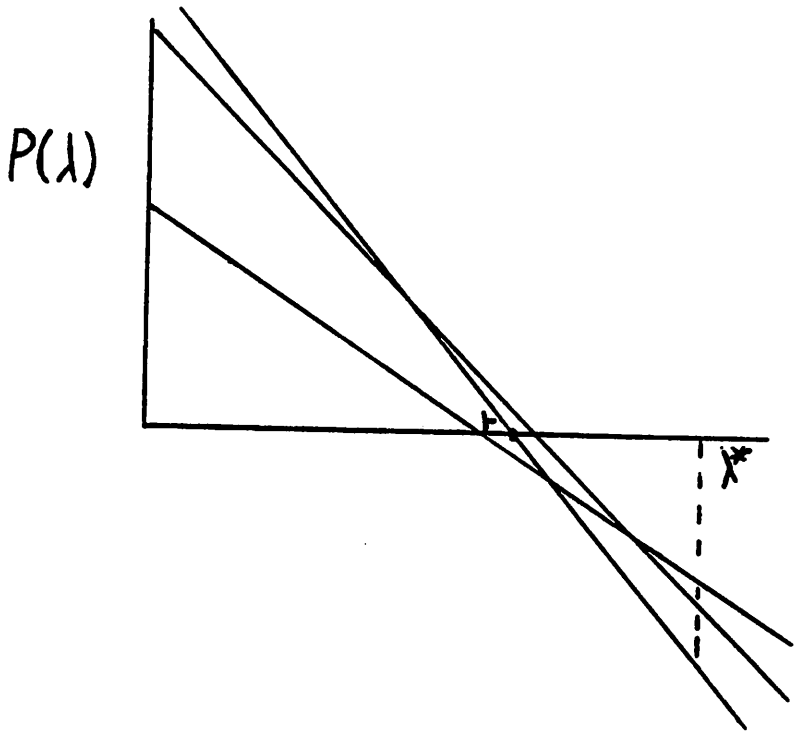


Figure 1.4

2. It is generally unnecessary to run all of A' when $P(\lambda^*)$ is computed in step 4. A' needs only be run forward from the point in A' corresponding to the branch statement at pc where $P(\lambda^*)$ must be determined (see figure 1.5). To prove this, note that the actions of A taken in M(A) are consistent with the actions of A' in the computation of $P(\lambda^*)$, until pc is reached. Therefore,

these computations need not be repeated. To get the values of program variables for A' at pc, compute g(λ*) at pc for all program variables g in A.

Essentially, when A reaches a branch statement, the correct action can be determined by plugging in λ* and completing the computation of A with the resulting constant data. This is cheap to implement, and is certainly an improvement in practice.

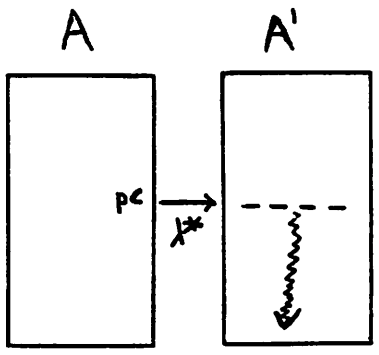


Figure 1.5

Chapter II: MATROID OPTIMIZATION WITH THE INTERLEAVING OF TWO ORDERED SETS

Introduction

In this chapter we examine a problem with special structure allowing very strong results. We consider problems which arise when many successive matroid optimization (minimum cost base) problems must be solved, each problem differing from the others in a structured manner. We present theorems characterizing the structure of the successive optimal solutions, and a preprocessing method allowing very rapid solutions to the successive optimization problems.

Given a matroid M with element set E partitioned into two sets R and W, consider the set of all orderings of E which preserve the internal orders of R and W respectively. Associated with each such ordering is a minimum weight base of M. In this chapter we study the class of all minimum weight bases associated with the above set of orderings. We characterize the structure of this class of bases, and present an algorithm which yields a representation of them. We relate these results to the problem of repeatedly finding a minimum weight base of M, where the element weights are successively drawn from a restricted class of possible weights. We further use the algorithm to efficiently solve the following parametric problem: For M a matroid with $E = R \cup W$, and a weight defined for each element in E, find for all feasible q, the minimum weight base of M containing exactly q elements of R.

Basic definitions for matroids and graphs can be found in Lawler [LAW]. The reader unfamiliar with matroids, but comfortable with graphs can follow the chapter by specializing the results to the minimum spanning tree problem, substituting graphs for matroids, edges for elements, spanning trees for bases, and cycles for circuits.

2.1 Problem Definitions

Let $M = (E,I)$ be a matroid with element set E of size n, partitioned into two ordered sets R (red) $= (R_1, R_2 ,.., R_s)$ and W (white) $= (W_1, W_2 ,.., W_t)$, where $t = n - s$. A weighting of E, denoted $C(E)$, is defined as a map from the reals onto E. For an element E_i of E, $C(E_i)$ is called the weight of E_i. The interleavings of E, denoted $I(R,W)$, consist of all weightings such that $C(R_1) < C(R_2) < ... < C(R_s)$ and $C(W_1) < C(W_2) < ... < C(W_t)$. The order of the weights of the red elements is preserved by $I(R,W)$, as is the order of the weights of the white elements, but the two sets may be ordered together by any interleaving of R and W.

Given $C(E)$, the weight of a base B of M is defined as the sum of the weights of the elements of B, and the minimum weight base is the base with minimum weight over all bases of M. For every interleaving in $I(R,W)$ there is an associated minimum weight base, and we define $B(R,W)$ as the set of all bases such that each is the associated minimum weight base for some interleaving. That is, for every B in $B(R,W)$ there is an interleaving $C(E)$ in $I(R,W)$ for which B is the minimum weight base. The following theorem characterizes the structure of the set $B(R,W)$.

2.2 Pairing Theorem

Theorem 2.1 Given the ordered sets R and W; the elements of set E can be partitioned into the following three classes:

i. elements which appear in no base in B(R,W)

ii. elements which appear in every base in B(R,W)

iii. an equal number of red and white elements mated one - one into red white pairs (R_i, W_j) such that for any base B in B(R,W), R_i is in B if and only if W_j is not. Further, R_i is in B whenever $C(R_i) < C(W_j)$; W_i is in B whenever $C(R_i) > C(W_j)$.

Hence for any interleaving, the associated minimum weight base contains all elements from class ii, and exactly one element from each pair in class iii.

Proof of the Pairing Theorem

The proof of theorem 2.1 is in three parts; recognizing the three classes; proving equal cardinality of the red and white elements of class iii; and pairing the class iii elements into red - white pairs so that for any base B in B(R,W) exactly one element of each pair is in B. The following fact will be needed throughout.

Fact: Given an weighting C(E) of the elements E in M, an element e is in the minimum weight base of M, if and only if e is not the maximum weight element of any circuit in M.

Lemma 1: Classes i and ii can be recognized by two minimum weight base computations.

Proof of Lemma 1: Consider the interleaving C'(E) in which every red element has less weight than every white element, and let B' be the associated minimum weight base. For such weights, the above fact implies that any red element R_i not included in B' is the maximum weight element of some circuit S containing only red elements. The interleavings preserve the order of the weights of the red elements, hence R_i will be the maximum weight element of S for any interleaving, and will be in no base in B(R,W). Conversely, no red element in B' is in class i, and therefore the red elements of class i are exactly those red elements omitted from B'. Further, any white element W_j in B' will be in every base of B(R,W). Since W_j is not the maximum weight element of any circuit when all red elements weigh less than all white elements, there is no interleaving in which W_j is the maximum element of some circuit of M. Conversely all white elements of class ii are in B', and therefore the white elements in B' are the white elements of class ii.

In a similar way, the white elements of class i and the red elements of class ii are recognized by considering all red weights greater than all white weights, and the associated minimum weight base B''. ▪

Class iii is defined as the set of elements of E not in classes i or ii.

Lemma 2: Class iii contains an equal number of red and white elements.

Proof of Lemma 2: Let B' and B'' be as in Lemma 1. The reds omitted from B' are exactly the red set of class i, hence B' contains all the red elements of classes ii and iii. Further, the white elements in B' are exactly the whites of class ii, and so B' consists of the red and white elements of class ii, and the red elements of class iii.

Similarly, B'' consists of the red and white elements of class ii, and the white elements of class iii. B' and B'' have equal cardinality, and so class iii contains an equal number of red and white elements. ▪

Pairing Algorithm

Let k be the number of elements of each color in class iii. We now give an algorithm which finds k red - white pairs of class iii elements, such that at most one element from each pair is present in any base in B(R,W). Let B' be as in Lemma 1, and assume WLOG that the white elements of class iii are $W_1, W_2 ,.., W_k$. Set B_0 equal to B' and repeat the following for j = 1 through k:

a) Add W_j to B_{j-1}, creating the unique circuit C_j.

b) Find the red element R_i of largest index among the red elements of C_j. We will show below that such a red element exists, and is in class iii.

c) Pair R_i with W_j. Remove R_i from the result of step a), yielding the base B_j.

Figure 2.1 gives an example of the algorithm and the resulting pairing for a graphic matroid, where the minimum weight base problem is the minimum spanning tree problem.

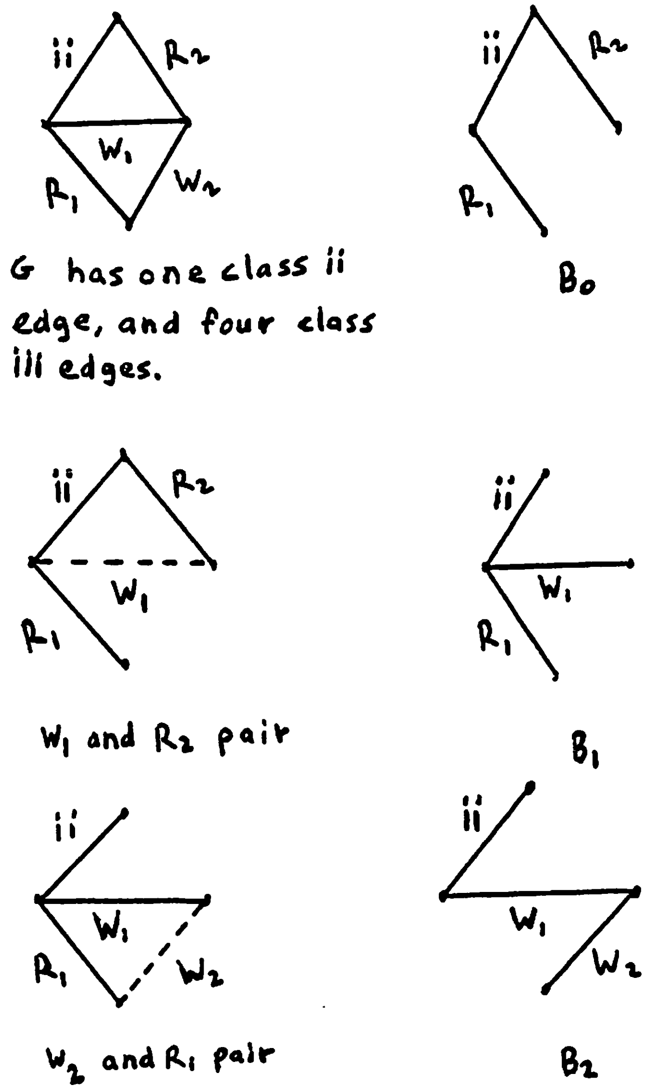


Figure 2.1

Lemma 3: The pairing algorithm pairs the elements in class iii, and for any base B in B(R,W) at most one element from each pair is in B. In fact, for B associated with the interleaving C(E), only the least weight element of each pair can appear in B.

Proof of Lemma 3: Consider W_j and the circuit C_j resulting from adding W_j to B_{j-1} in step a). By construction, W_j is the white element of maximum index in C_j, and hence for any interleaving, the maximum weight white element in C_j. C_j must contain at least one red element, or else W_j would be the maximum weight element in C_j for all interleavings, and would be in class i. Let R_i be the red element paired to W_j in step c) of the algorithm. For any interleaving, R_i is the maximum weight red element in C_j, and hence for some interleaving, the maximum weight element in C_j. Therefore, R_i can't be in class ii, and must be in class iii.

The above shows that for any interleaving C(E), the maximum weight element of circuit C_j is either R_i or W_j. Therefore, for any B in B(R,W), at most one of those elements can be in B. In fact, for interleaving C(E) and associated base B, only the minimum weight element of the pair can be in B. ▪

To complete the proof of the pairing theorem, note that every base B in B(R,W) contains exactly k class iii elements. Since there are k class iii pairs, and at most one element from each is in B, exactly one element from each pair is in B. This completes the proof of the pairing theorem.

Improved Algorithm

In the above presentation, classes i and ii were recognized in two passes through E, and the class iii elements were paired in a different algorithm. These tasks were done separately for clarity, but can be done together in one pass through E. Let B_0 be the empty set, and apply the pairing algorithm using all elements E in the order $R_1, \ldots, R_s, W_1, \ldots, W_t$. Any element completing a monochromatic circuit is rejected and placed in class i. Any white element completing a bi - chromatic circuit is paired with the red element of maximum index in the circuit, and the red element is deleted. Finally, any white element not completing a circuit is placed in class i, as is any red element which remains in the base after all the white elements have been examined. Viewed this way, the pairing algorithm is seen as an extension of the Greedy algorithm [Law] for the minimum weight base of a matroid.

2.3 The Matroid Selection Problem

Let M be a matroid with element set E partitioned into sets R and W, with real valued weights C(E) assigned to E. The matroid selection problem is to find, for all feasible q, the minimum weight base B of M, subject to the constraint that B contain exactly q elements from R. This problem is efficiently solved using the pairing theorem and algorithm.

To solve the selection problem, we modify the weights of R by adding the variable λ to the weights of all elements in R. Then any base containing r elements from R, has modified weight equal to its original weight plus $r \times \lambda$. Therefore, for fixed λ, if the resulting minimum modified weight base B has r elements of R, then B solves the selection problem for $q = r$. Varying λ from $-\infty$ to $+\infty$ produces a sequence of minimum modified weight bases corresponding to the solutions of the selection problem, with q varying from its largest possible value to its smallest possible value.

Adding λ to the weights of R preserves the order of the R set weights, while the weights and order of the W set remains constant. Hence as λ varies the order of the weights of E defines a class of interleavings of R and W. Then if R_i and W_j are paired elements of class iii, and B is the minimum weight base associated with a given value of λ, the pairing theorem states the R_i is in B if and only if $C(R_i) + \lambda < C(W_j)$. The selection problem can be efficiently solved as follows:

a) Given R, W and C(E) find classes i, ii and the pairings of class iii.

b) For each (R_i, W_j) pair of class iii, compute $\lambda_{ij} = C(W_j) - C(R_i)$, and sort the λ_{ij} values.

c) Suppose the number of R elements of class ii is p. Then the minimum weight base with exactly q elements from R consists of the class ii elements, the red elements of the q - p pairs with smallest λ_{ij} values, and the white elements of the remaining pairs.

d) The minimum value for q is p, and the maximum value is p + k, and c) implies that any value of q between p and s + k is feasible.

It is possible to improve the efficiency of this method in the context of particular matroids. For example, Gabow and Tarjan [GT] have improved the speed of the pairing algorithm for graphic matroids, and have presented other methods to solve the selection problem for other specific matroids.

2.4 Successive Modification

Many computing applications involve the repeated computation of a given function, where the values of the function variables are successively modified, or are drawn from some class of possible values. For such applications, it is desirable to devise methods to "preprocess" the data in order to speed up the successive computations.

Consider the situation in which the successive modification consists of choosing different interleavings in a minimum weight base problem. A typical special case of this model is the situation in which the weights of some elements are held constant, while the remaining weights are given by an order preserving function of some parameter. The pairing algorithm can be used to preprocess the elements so that when actual interleaving is specified, the minimum weight base can be computed at the cost of at most m comparisons, where m is the size of the base. The complexity of the pairing algorithm will vary for different matroids, and depends on the difficulty of finding circuits. However, once the pairing is completed, the minimum weight base problem reduces to a simple problem of comparisons, regardless of the matroid. This is a substantial algorithmic improvement for many matroids where finding circuits is expensive. Further, the pairing gives a compact representation of all the bases in B(R,W), which is useful for computer applications where the bases must be easily stored and retrieved.

Chapter III: ALGORITHMS FOR PARAMETRIC PROGRAMMING

In chapter I we discussed two methods (the parametric simplex and the Eisner - Severance method) for parametric programming. In this chapter we examine three additional methods to solve the parametric programming problem for combinatorial problems. We first discuss two general methods that work for some combinatorial problems, and then discuss a method which solves the parametric programming problem on-line in polynomial time per breakpoint for many combinatorial problems.

Let P be an unparametrized problem, and let X be the set of all feasible solutions to P. Associated with every decision variable i are two costs S(i) and T(i). Define for solution x,

$$S(x) \equiv \sum_i S(i)x_i \text{ and } T(x) \equiv \sum_i T(i)x_i$$

The function $P(\lambda)$ is the following: For a given value of λ, $P(\lambda)$ = minimum value of $S(x) + \lambda T(x)$ taken over all $x \in X$.

The parametric problem $P(X,\lambda)$ is the problem of determining $P(\lambda)$ as a function of λ. Note that $P(\lambda)$ is a piecewise linear convex function of λ. Points of discontinuity in $P(\lambda)$ are called a *breakpoints*.

3.1 Method I

Let x be an optimal solution to a parameterized problem P at λ^1. We can find λ^2, the largest value of λ such that x is still optimal, as follows: characterize the set of all possible changes in x that could occur at λ^2, and search over this set to find the correct change. We illustrate the idea on the directed shortest path problem from s to all other nodes in a graph G. Let x be a shortest path tree at λ^1. Then at λ^2 the new optimal tree will differ from x by the addition of one edge directed into a node j, and the deletion of one edge directed into j. λ^2 can then be found as follows:

1. For each node i, let $D(i,\lambda)$ be the cost, as a function of λ, from s to i in x.
2. For each edge $e = (i,j) \notin x$, let $S(e,\lambda) = D(i,\lambda) + D(e,\lambda)$, where $D(e,\lambda)$ is the cost of edge e as a function of λ. Let λ_e be the value of λ such that $S(e,\lambda) = D(j,\lambda)$. Compute λ_e for all edges $e \notin x$.
3. $\lambda^2 = \text{Min}\ \lambda_e$. Let $a = (i,j) \notin x$ be the edge such that $\lambda^2 = \lambda_a$, and let $b \in x$ be the unique edge in x directed into node j. Then x^2, the next optimal tree, is found by removing edge b from x and adding edge a.

Implementation

This method can be used to find successive breakpoints on-line in left to right order. The key implementation issue is the work required to update the $D(i,\lambda)$ and $S(i,\lambda)$ values after a change in the tree, and the work required to find the minimum $S(e,\lambda)$ value. If the distance to node b is changed, then $D(i,\lambda)$ changes for every node in the subtree rooted at b, and $S(e,\lambda)$ changes for every edge e into or out of that subtree. A naive implementation of the updating requires $O(n^2)$ time per breakpoint. In the case of strictly positive edge costs, this is the same as the Eisner-Severance method, but for negative costs it is an improvement.

It is not always possible to characterize the type of change that occurs at each breakpoint, and not always easy to carry out the updating. For example, in the s - t shortest path problem (in contrast to the all node problem above) it is not clear what form the path change takes. The problems with method I for the s - t version are like the problems of degeneracy in the parametric simplex method. It is interesting to note that the parametric simplex algorithm is the same as method I for the all nodes shortest path problem. In general this is true for all non-degenerate linear programming problems, but method I works also for problems which are not linear programming problems. For example, this method could be used to find breakpoints for parametric sorting problems, or scheaduling problems where only local changes occur to the optimal scheadule at the breakpoints.

3.2 Method II

A second general parametric programming method is to focus on the change in behavior of the optimizing algorithm instead of the change in the solution. This is a weaker method than method I, but can sometimes be used when method I is unworkable. We will illustrate this method with the parametric minimum spanning tree problem.

Let A be a minimum spanning tree algorithm, and suppose tree T is optimal for the value of λ equal to λ'. The branching action of A while computing T implies a set of inequalities in λ which hold at λ'. Then if λ^* is a value of λ for which the inequalities hold, T will be optimal for λ^* also. For example, suppose that a branch in A depends on the comparison of $P(i,\lambda)$ and $P(j,\lambda)$, the parametric costs of edges i and j respectively. Then the branch action taken for λ equal to λ' implies say that $P(i,\lambda) > P(j,\lambda)$. The method to find the next breakpoint is to first locate the smallest value of λ such that one of the inequalities is violated. Computation is then backed up to the associated branch point, certain inequalities are removed, A is rerun forward from that branch point, and additional new inequalities are implied. By repeating this operation, the desired breakpoint is eventually discovered.

Implementation of this method is very simple. For inequality i, let λ_i be the minimum value of $\lambda > \lambda'$ such that inequality i is violated. When branch point i is encountered in the execution of A, λ_i is computed and stacked on SK along with a pointer to the smallest λ value among λ_i and the values already on SK. When λ_i is stacked it is compared to the λ value pointed to by the top of the stack, and its pointer is then set to the smallest of the two values. Hence, the top of the stack always points to the smallest λ value in the stack. Figure 3.1 shows a snapshot of SK.

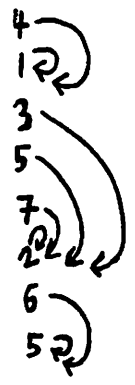


Figure 3.1

When algorithm A is backed up to the branch point j associated with λ_j, the minimum λ in SK, the portion of SK above λ_j is deleted, λ is set greater than λ_j, and A is run forward from branch point j.

The problem with this method is that a change in the behavior of the algorithm does not alway imply a change in the optimal solution. If A is the greedy algorithm, then the behavior of the algorithm changes whenever the order of the edge costs change, but this does not always correspond to a change in the minimum spanning tree. Further, there may be many minimum spanning trees for a given value of λ, and there is no mechanism in this method to avoid generating them all.

It is interesting to note that in the case of the s to all nodes shortest path problem, methods I, II, and the parametric simplex method are essentially the same. This is not true in general.

3.3 Megiddo Based Methods

In this section we show how to determine successive contiguous breakpoints in polynomial time per breakpoint for a large class of combinatorial problems. The method is based on the key idea of the Megiddo technique for rational optimization discussed in section 1.3.10.

Given parametric problem $P(X,\lambda)$, let x^1 be a feasible solution of cost $S(x^1) + \lambda T(x^1)$ which is optimal for a non-zero length interval of λ beginning at λ^1. Let $\lambda^2 > \lambda^1$ be the first breakpoint larger than λ^1, i.e. the right endpoint of the interval of optimality of x^1 in this case.

Lemma: Given x^1 and λ^1 above, λ^2 is the solution to the following problem:

P': Minimize λ

such that for some $x \in X$,

$S(x^1) + \lambda T(x^1) = S(x) + \lambda T(x)$

$S(x^1) < S(x)$

Proof: Consider figure 3.2. λ^2 is the first point to the right of λ^1 where the cost of another $x \in X$, $x \neq x^1$ intersects the cost of x^1. By the optimality of x^1 at

λ^1, $S(x) > S(x^1)$ and the lemma is proved. ▪

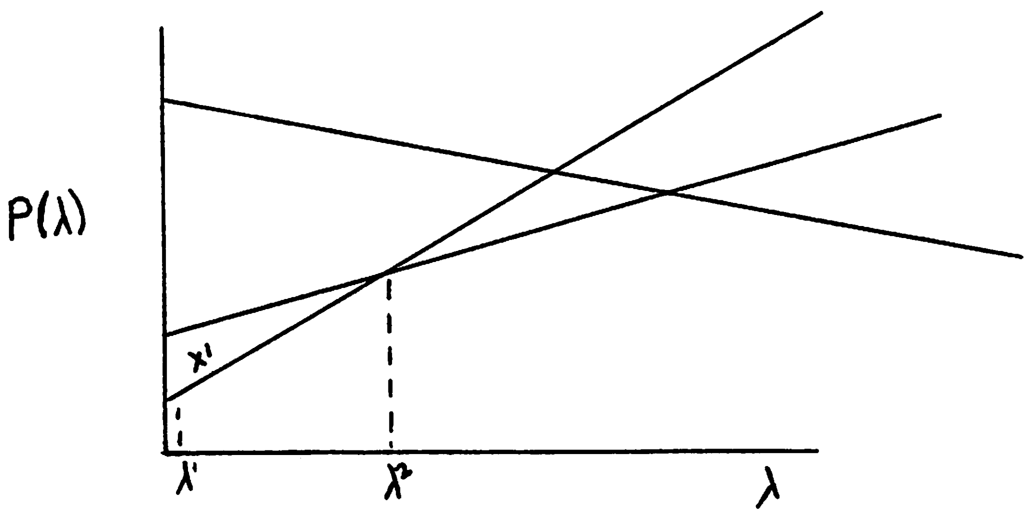


Figure 3.2

Hence, by successively solving P', a few contiguous breakpoints can be determined, or $P(\lambda)$ can be determined for an entire range of λ.

Solving P'

For many combinatorial problems, P' can be solved by a method similar to the Megiddo technique for rational optimization (section 1.3.10). Given the parametric problem $P(X,\lambda)$, let A be a suitable algorithm which solves $P(\lambda^*)$ for λ^* a fixed value of λ. By "suitable" we mean one which works for the Megiddo method of section 1.3.10. For a complete introduction to the notation and terminology refer to section 1.3.10. P' is solved as follows:

Algorithm PM(A)

Input: Breakpoint λ^1 and solution x^1 optimal at λ^1 and some $\lambda > \lambda^1$.

Output: Program variables p^2 and s^2. p^2 is a value of λ, generally λ^2, and s^2 is a solution that is optimal at p^2, generally x^2.

1. Denote interval I as [L,R]. Set L to λ^1 and R to ∞. Set N to λ^1. Set pc to 1. Note that $\lambda^2 \in I - \{N\}$. Let $g(\lambda)$ be a linear form representing the value of program variable g as a function of λ.

2. Simulate the straight line code of A, starting at point pc, until a branch statement is reached. If the end of A is reached, then go to step 6.

3. At a branch statement, two linear forms representing the values of two program variables, say g and h, are compared. Let $g(\lambda)$ and $h(\lambda)$ be linear forms representing the value of program variables g and h as a function of λ. Solve for λ^*, the unique value of λ such that $g(\lambda^*) = h(\lambda^*)$.

4. If $\lambda^* \in I - \{N\}$ go to step 5. If $\lambda^* \notin I - \{N\}$ then either $g(\lambda) > h(\lambda)$ or $g(\lambda) < h(\lambda)$ for all $\lambda \in I - \{N\}$. This determines whether $g(\lambda^2)$ is greater than or less than $h(\lambda^2)$, and hence the correct branching of A is also determined (i.e. what branch A would take on computing $P(\lambda^2)$). Branch to the appropriate code in A; set pc to the start of that code, and return to step 2.

5. If $\lambda^* \in I - \{N\}$, then for all $\lambda \in I - \{N\}$, $g(\lambda) > h(\lambda)$ on one side of λ^*, and $g(\lambda) < h(\lambda)$ on the other. To determine the proper branching of A at this branch statement, compute $P(\lambda^*)$ in a separate running of algorithm A' (as in section 1.3.10, A' is a copy of A). Let x be the resulting optimal solution to $P(\lambda^*)$. Then there are three cases:

5i. If $S(x) + \lambda^* T(x) = S(x^1) + \lambda^* T(x^1)$ and $S(x) > S(x^1)$ then set L and N to λ^*, and go to step 4.

5ii. If $S(x) + \lambda T(x) = S(x^1) + \lambda^* T(x^1)$ and $S(x) = S(x^1)$ then $\lambda^2 \geq \lambda^*$. Set interval I to $[\lambda^*,R]$, and determine the correct branching for A. Branch to the appropriate code in A and set pc to the start of that code. Return to step 2. See figure 3.3.

5iii. If $S(x) + \lambda^*T(x) < S(x^1) + \lambda^*T(x^1)$ then $\lambda^2 < \lambda^*$ and R, the right endpoint of I, should be reduced. Set the value of R to r such that $S(x) + rT(x) = S(x^1) + rT(x^1)$, and determine the correct branching for A; branch to the appropriate code in A, set pc to the start of that code, and return to step 2. Note that $r < \lambda^*$. See figure 3.3.

6. Let $y \in X$ be the solution computed by the action of algorithm A above, and let [L,R] be the final endpoints of interval I. If $S(y) > S(x^1)$ then set program variable p^2 to L. If $S(y) = S(x^1)$ then set p^2 to R. If $R = \lambda^2 < \infty$, then set program variable s^2 to x, else to y. Terminate the algorithm.

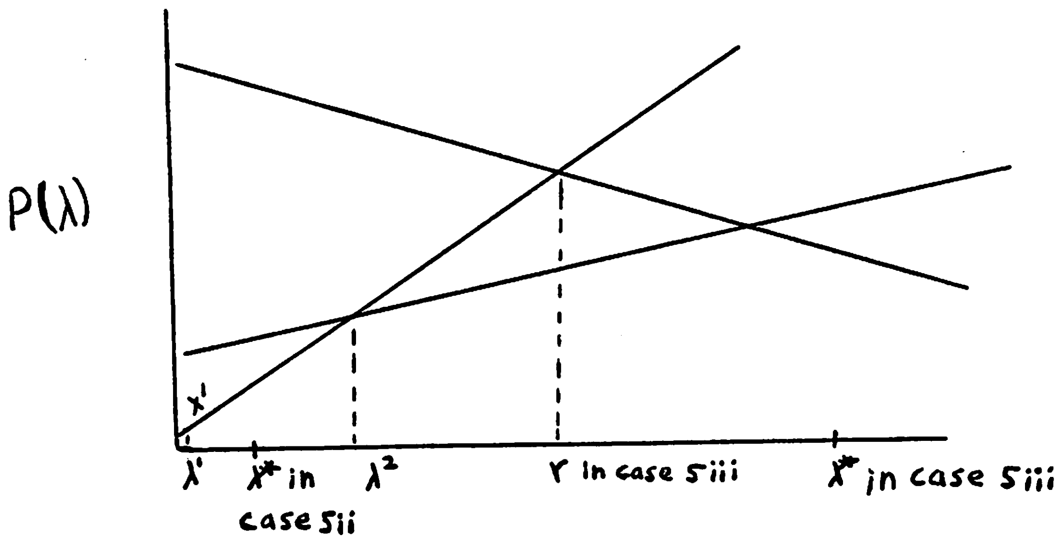


Figure 3.3

Theorem 3.1: Algorithm PM(A) solves problem P'; The value of p^2 equals λ^2, and the value of s^2 is x^2, a solution optimal at λ^2 and some $\lambda > \lambda^2$.

Proof: Let [L,R] be the final endpoints of I, and y be as in step 6. Then $L \leq \lambda^2 \leq R$, and y is optimal for all values of λ in [L,R].

To see the first claim, note that the left point of I began at λ^1, where x^1 is optimal, and moved left only to values of λ where x^1 was discoved to be optimal. Hence x^1 is optimal at L. Conversely, if $R < \infty$, then PM(A) discovered a solution better than x^1 for $\lambda > R$, and so $L \leq \lambda^2 \leq R$. If $R = \infty$ then $\lambda^2 \leq R$ trivially.

To prove that y is optimal for [L,R], we claim that the actions of algorithm A which produced the solution y are correct actions for any value of λ from L to R. Essentially, PM(A) provides a proof that y is optimal for all values in [L,R]. This is very clear if step 5i is never executed. If step 5i is executed, then the actions taken by A are consistent with the actions of A in computing $P(\lambda^2 + \varepsilon)$ for $\lambda^2 + \varepsilon < \lambda^3$, the next breakpoint to the right of λ^2. Hence y is optimal for all values of λ between L and R.

Now both x^1 and y are optimal at L. If $S(x^1) = S(y)$ then x^1 is also optimal at R, and $\lambda^2 = R$. If $S(y) > S(x^1)$ then λ^2 must be at L. In either case, p^2 is set to λ^2. $S(y) < S(x^1)$ is not possible, for it implies that $L = \lambda^1 = R$, and that x^1 is optimal only at λ^1, contradicting the assumption that x^1 is optimal for some value of λ greater than λ^1. Hence p^2 is set to λ^2 as claimed.

Now consider s^2. Recall that s^2 is set either to x (x is set in step 4), or to y. If $R = \infty$ then y is optimal for all values of λ from λ^1 to ∞. If $R < \infty$ then either $R = \lambda^2$ or $R > \lambda^2 = L$. In the first case, s^2 is set to x, which is optimal at λ^2 and also at some $\lambda > \lambda^2$. In the second case, s^2 is set to y, which is optimal at both L and $R > L$. Hence s^2 is optimal at λ^2 and some $\lambda > \lambda^2$. ▪

Degeneracy

We now examine the case of degeneracy. Suppose that x^1 is optimal only for $\lambda \leq \lambda^1$.

Claim: If x^1 is degenerate (optimal only for $\lambda \leq \lambda^1$) then PM(A) ends with p^2 set to λ^1, and with s^2 an optimal solution for λ^1 and some $\lambda > \lambda^1$.

Proof: Suppose that PM(A) is executed with x^1 degenerate. Let R and L be the final endpoints of I in step 6. Note that statements 5i and 5ii are never executed, and hence L equals λ^1. Note also that y is optimal for all λ in [L,R]. If R = L then PM(A) sets p^2 to λ^1, and s^2 to x, both correct actions. If R > L then x^1 isn't optimal at R and $S(y) > S(x^1)$. Hence PM(A) sets p^2 to λ^1, and s^2 to y, again correct actions. ▪

Hence PM(A) discovers that x^1 is degenerate, and finds a solution s^2 optimal at λ^1 and some $\lambda > \lambda^1$. The next breakpoint, λ^2, can then be determined by rerunning PM(A) using s^2 in the place of x^1.

These observations support the claim that $P(\lambda)$ can be determined at a cost of at most two applications of PM(A) for the first breakpoint, and one per breakpoint thereafter.

A Refinement

In the case where more than one breakpoint must be determined, the above method of successive calls to PM(A) can be improved. The idea is to remember information about $P(\lambda)$ discovered while locating one breakpoint, for use in locating successive breakpoints. The conclusion is that, in practice, not all of PM(A) needs to be executed for every breakpoint.

Suppose that breakpoint λ^2 has been located along with solution x^2 which is optimal at λ^2 and some $\lambda > \lambda^2$. Let s^2 be as in step 5, hence $s^2 = x^2$.

While locating λ^2, PM(A) called $P(\lambda*)$ for perhaps several $\lambda* > \lambda^2$. These calls determined feasible solutions and hence determined an upper bound U on $P(\lambda)$, for $\lambda > \lambda^2$. That is, U is the lower cost envelope of the solutions for which step 5iii is executed. Let U(L) be the leftmost line segment of U excluding s^2, and let x be the solution associated with U(L). Let R_x be the intersection point of the cost functions of x^2 and x. See figure 3.4.

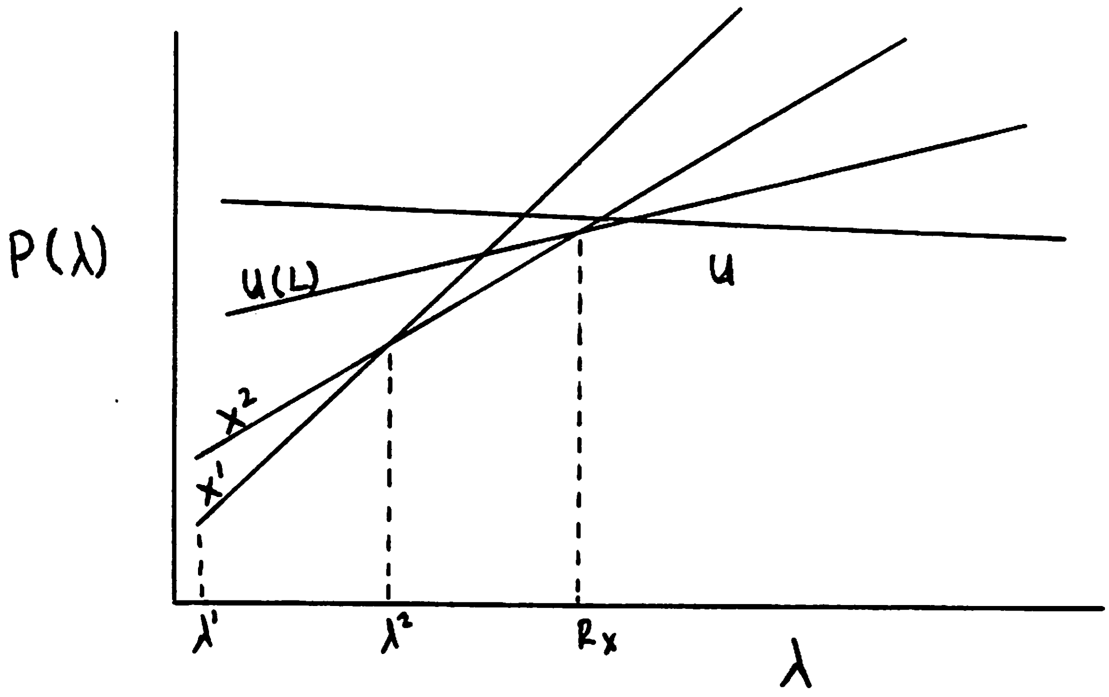


Figure 3.4

Note that every solution x of step 5iii is in U, for each such x is an optimal solution of $P(\lambda^*)$ for some λ^*. Note also that U is built up at its left end, for the values of λ^* used in step 5iii are strictly decreasing. Therefore, keeping track of additions to U is a simple matter, implementable by stacking the line segments of U as they are found. We will later exploit this.

Lemma: When locating λ^3 (the first breakpoint to the right of λ^2) R can be initialized to R_x. More importantly the computation to locate λ^3 need not begin at the start of PM(A), but can start after the point in PM(A) where line segment U(L) is generated.

Proof: Suppose U(L) is generated at time t in the search for λ^2, and let R_t be the right end of I at time t. By construction, x is optimal for some $\lambda \leq R_t$. However, x^2 is better than x to the left of R_x, hence $R_t \geq R_x$. The right end of I moves only to the left, hence in the search for λ^2, the right end of I is to the right of R_x until time t. Note also that $\lambda^3 \leq R_x$.

Now suppose that x^2 is used to locate λ^3, and that PM(A) starts at its beginning, with $L = \lambda^2$ and $R = \infty$. Since $\lambda^3 \leq R_x$, and the right end of I is to the right of R_x until time t, the action of A in locating λ^3 will be identical to the action of A in locating λ^2, until the point where U(L) is generated. When U(L) is generated in the former case, R is set to R_x, while in the latter case R is set to the intersection of U(L) and x^1, which is to the left of R_x. Hence after U(L) is generated, the two executions of A may differ, but until that point, they are the same and need not be duplicated. ▪

We have shown that when λ^2 has been located, PM(A) should be returned to the state it was in just after U(L) was generated, except that R should be set to R_x. In general, we define U^i to be the upper bound of $P(\lambda)$ for $\lambda > \lambda^i$ at the point when λ^i and associated x^i are discovered. $U^i(L)$ is defined as the leftmost line segment of U^i excluding x^i. Then to find x^{i+1}, R is set to the intersection of x^i and $U^i(L)$, and PM(A) is reset to its state when $U^i(L)$ was generated.

The mechanics of retrieving $U^i(L)$ and proper program state are very simple. Whenever step 5iii. is executed, the solution x is pushed onto a stack SK along with the associated program state. When breakpoint λ^i is located in step 6, $U^i(L)$ and associated program state are either on the top of SK or one location below the top. If $\lambda^i < R$ then $d^i = y$, and $U^i(L)$ is on the top of SK. If $\lambda^i = R$ then $d^i = x^i = x$ which is on the top of SK, hence $U^i(L)$ is on the next location in SK.

Complexity and Advantages of PM(A)

If algorithm A is polynomial, then PM(A) is also, since the complexity of PM(A) is at most the square of the complexity of A. Again, tricks such as those discussed in 1.3.10 can reduce the complexity of PM(A) and speed it up in practice.

In summary, we have proved:

Theorem 3.2: Given a suitable algorithm A for the unparameterized problem P, the function P(λ) can be determined at a cost of at most two applications of PM(A) for the initial breakpoint, and one application of PM(A) per breakpoint thereafter. If A is polynomially bounded, then the cost per breakpoint to determine P(λ) is also polynomially bounded.

The advantages of the PM(A) algorithm are the following: First, unlike the Eisner - Severance method, it finds breakpoints successively from left to right. Therefore, for many combinatorial problems, it can be used to find just one breakpoint in polynomial time, or to find several breakpoints on-line in polynomial time. Further, PM(A) is easily generalized to higher dimensional problems, unlike the Eisner-Severance method. Second, unlike the parametric simplex method which suffers from two types of degeneracy, PM(A) finds at most two optimal solutions per breakpoint, and exactly one optimal solution per line segment between successive breakpoints. Further, PM(A) works for problems that are not linear programming problems, and can provide polynomial bounds, whereas the parametric simplex method is not polynomially bounded.

The s to t Shortest Path Problem

With algorithm PM(A) we can now solve the single source - single destination parametric shortest path problem on-line in polynomial time per breakpoint. The time needed per breakpoint by the PM(A) algorithm is $O(n^3 \log n)$. Recall that this on-line polynomial bound could not be obtained with either the parametric simplex or the Eisner - Severance methods.

Minimum Cost Flow

If the optimal solution is known for one given value of λ, then often P(λ) can be determined in polynomial time per breakpoint even without a suitable polynomial time algorithm A for the unparameterized problem P. The key is that the problem P' of determining the next breakpoint can often be expressed as a specially structured subproblem for which a suitable A algorithm exists. For example, no suitable polynomial time algorithm is known for the minimum cost flow problem [LAW]. However, if the optimal flow F is known for a value λ^1 in the parametric minimum cost flow problem, then the breakpoint $\lambda^2 \geq \lambda^1$ can be found as follows:

1. Given F, construct the augmentation network F^A.
2. λ^2 is the smallest value of $\lambda \geq \lambda^1$ such that there exists a negative cost cycle in F^A.
3. λ^2 can be found by M(A) of section 1.3.10 since the basic algorithm A is now any suitable polynomial time algorithm (such as Floyd's method) to find the smallest cycle in F^A. λ^2 is the value of λ where the smallest cycle has cost 0.

3.4 An Application to Program Module Distribution

We use algorithm PM(A) to solve a problem in two dimensional parametric programming introduced by H. Stone [STO]. The unparameterized problem is as follows:

In a distributed computing system, n modules of a program must be distributed between two processors, say P_1 and P_2. For each module i, a cost of R_{i1} is incurred if i is run on P_1, and a cost of R_{i2} is incurred if i is run on P_2. In addition, for each pair of modules (i,j), a communication cost C_{ij} is incurred if modules i and j are not allocated to the same processor. The objective is to distribute the program modules to minimize the total cost of running the program.

The above problem can be modelled and solved as a maximum flow - minimum cut problem in an s-t network G. Let node s represent P_1 and node t represent P_2, and let each node i represent module i. Each node i is adjacent to s with an edge of capacity R_{i2}, and adjacent to t with an edge of capacity R_{i1}. Every node pair (i,j) is connected by an edge of capacity C_{ij}. Then the minimum s-t cut in the network defines the optimal distribution of modules to processors. The nodes on the s side of the cut are assigned to processor 1, and the others are assigned to processor 2. See figure 3.5.

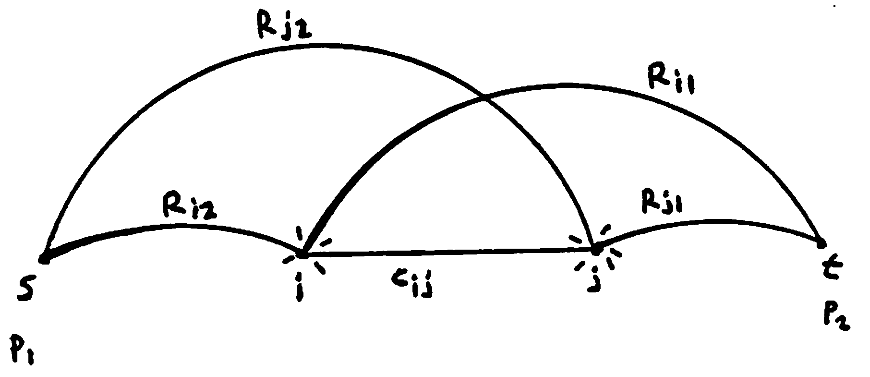


Figure 3.5

Stone [STO] discusses F(α), a one parameter version of this problem in which the costs R_{i1} are multiplied by a common parameter α, representing the congestion on P_1. As α increases the congestion and running times on P1 increase, the minimum cut changes and more modules are placed on P_2. Stone shows that as α varies from 0 to ∞, at most n minimum cuts are generated. These cuts are found by the Eisner-Severance technique of 1.3.7.

Two Parameter Problem F(α,β)

The two parameter problem, F(α,β), models variable congestion on both processors. The problem F(α,β) is to multiply R_{i1} by α, and multiply R_{i2} by an independent parameter β, solving the minimum cut problem as a function of α and β over a bounded range of the α, β space. That is, partition the α, β space ($\alpha_{min} \leq \alpha \leq \alpha_{max}$ and $\beta_{min} \leq \beta \leq \beta_{max}$) into regions such that in each region a single s-t cut is minimum. Note that the number of regions must be finite, since the number of s-t cuts in G is finite.

We present an algorithm to solve $F(\alpha,\beta)$ which is polynomial in the number of vertices and edges, and hence the number of regions, in the plane decomposition. We first present several definitions and facts.

Definition: Let K_i be an s-t cut in G, and let $R(K_i)$ be the set of (α,β) pairs for which K_i is the minimum cut in G.

Fact: $R(K_i)$ is a convex polygon.

Fact: For two distinct s-t cuts K_i and K_j, if $R(K_i) \cap R(K_j) \neq \phi$, then either $R(K_i) = R(K_j)$ or $R(K_i) \cap R(K_j)$ is an entire face (possibly a vertex) of both $R(K_i)$ and $R(K_j)$. Hence the solution of $F(\alpha,\beta)$ is unique.

Finding $R(k_i)$

To find $R(k_i)$ we will need to solve the following problem:

$D(K_i,p,L)$: Given cut K_i and a point p in (α,β) space for which K_i is a minimum cut, and given a half line L starting at p, determine the line segment LS contained in L where K_i is the minimum cut.

Lemma: $D(K_i,p,L)$ can be solved in time $O(n^5 \log n)$.

Proof: First, for each module i, express the cost of R_{i1} along L as an affine function of variable λ, $S_i + \lambda T_i$, where S_i is the cost of module i at p, and $T_i = R_{i1}$. Express the cost of R_{i2} along L as $S_i + \lambda T_{i2}$ where $T_{i2} = (\text{slope of L}) \times R_{i2}$. Essentially, express β in terms of α, and then normalize to make p the origin. Now $D(K_i,p,L)$ is equivalent to finding the extreme value (maximum or minimum depending on the direction of L) of λ for which K_i is the minimum cut. The point p is defined by $\lambda = 0$, hence finding the maximum value of $\lambda > 0$ where K_i is minimum is the problem of finding the first breakpoint along L for $\lambda > 0$. This can be solved by PM(A). The problem of finding the first breakpoint for $\lambda < 0$ can be solved by a modification of PM(A) to handle the negative direction. The A algorithm for PM(A) is any $O(n^3)$ maximum flow algorithm, and hence $D(K_i,p,L)$ can be solved in $O(n^6)$ time. This can be reduced to $O(n^5 \log n)$ using the binary search idea of section 1.3.10 with the Three Indians' flow algorithm. ▪

We assume that $R(K_i)$ has an interior point, and describe an algorithm $AR(K_i)$ to find $R(K_i)$ given a point p where K_i is minimum, and given a half-line L starting at p and intersecting some interior point of $R(k_i)$. If p is in the interior of $R(K_i)$ then any half-line starting at p will intersect the interior of $R(K_i)$. In the solution of $F(\alpha,\beta)$, p will generaly not be an interior point, but the direction from p through the interior of $R(K_i)$ will be known.

$AR(K_i)$: Algorithm to locate $R(K_i)$

1. Choose an half line L starting at p and intersecting the interior of $R(K_i)$. Parameterize L using λ, and assume that L is such that $\lambda > 0$.

2. Solve $D(K_i,p,L)$ producing $LS = [0,\lambda^1 > 0]$, and K_j, the minimum cut for $\lambda > \lambda^1$. K_j is determined by PM(A) in the search for λ^1. The capacities of K_i and K_j are both expressed as linear functions of α and β, and so the line L_{ij} of equal capacity is determined by equating the capacities and simplifying.

3. $R(K_i) \cap R(K_j)$ is contained in L_{ij}, and can be determined by solving $D(K_i,\lambda^1,L*)$ where L* is first the half line starting at λ^1 and running along L_{ij} in one direction, and then L* is the half line in the other direction. In this way, a face of of the polygon $R(K_i)$ is located. Note that if λ^1 is an end point of $R(K_i) \cap R(K_j)$, or if $\lambda^1 = R(K_i) \cap R(K_j)$, then PM(A) will recognize this by detecting degeneracy.

4. Pick a direction and a half line L starting at p and intersecting the interior of $R(K_i)$ so that L does not intersect any of the located faces of $R(K_i)$. If no such direction exists, then all faces of $R(K_i)$ have been located, and the algorithm halts. Else go to 2.

Step 4 can be implemented as follows: Consider a unit circle O around p, and the two half lines extending from p to the endpoints of a face of $R(K_i)$. See figure 3.6. The intersection points of O and these half lines are easily computed and kept in sorted order in a set of linked lists. Two points are linked together if they are associated with the endpoints of a face in $R(K_i)$, so that at a general step each linked list represents a sequence of adjacent faces that have been located so far. Then to find a half line L in step 4, find a direction so that L intersects O between the endpoint of one linked list and the starting point of the next one. When all the points form one circular list, $R(K_i)$ is fully determined.

Note that the time needed to find $R(K_i)$ is at most $O(n^5 \log n)$(number of vertices and edges of $R(K_i)$).

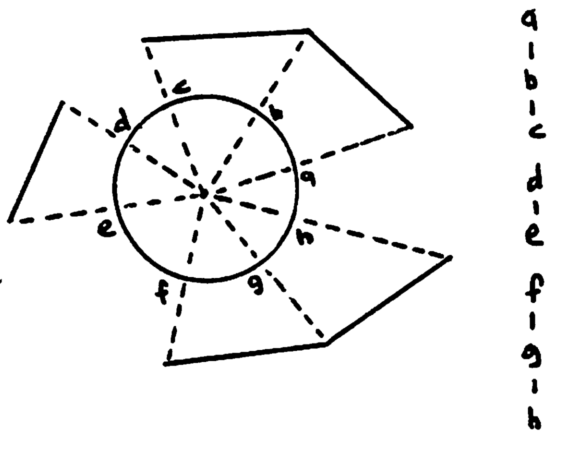


Figure 3.6

Solving $F(\alpha,\beta)$

The method to solve $F(\alpha,\beta)$ is to successively locate regions $R(K_i)$ for different i, without rediscovering any regions, until the α, β space is covered.

The regions are located in a roughly left to right manner, so that at any point in the computation the located regions establish a right-most frontier F consisting of a contiguous sequence of adjacent faces. See figure 3.7.

To extend a partial covering, pick a point p on F, and let K_i be the minimum cut at p which is optimal on the uncovered (right) side of F. Then locate $R(K_i)$. By the facts stated earlier, $R(K_i)$ will intersect F along a contiguous sequence of F, hence the new frontier is $F \oplus K(R_i)$. The frontier is easily maintained as a linked list of faces (points and edges).

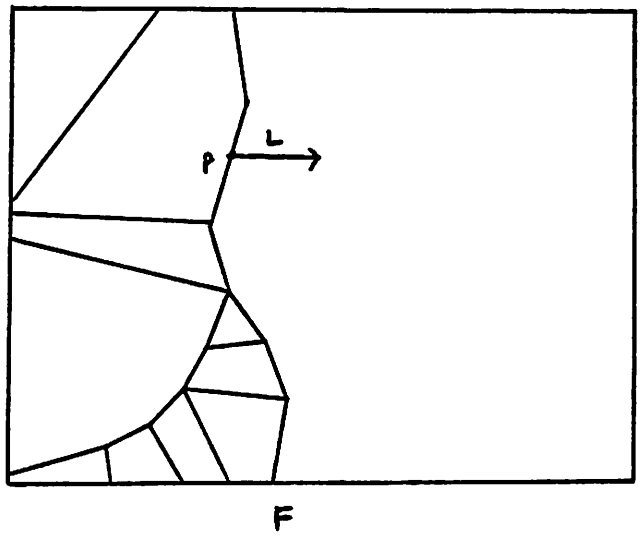


Figure 3.7

Refinements

We describe two refinements which may speed the solution of $F(\alpha,\beta)$ in practice.

1. When extending a partial covering, it may not be necessary to discover all of $R(K_i)$, since part of it has already been discovered. To eliminate the rediscovery of $F \cap R(K_i)$ walk along F in both directions from p, locating the farthest points where K_i is optimal. This is very cheap since the costs of the

edges in F are known, and K_i is optimal for a contiguous range of F.

2. Whenever L has either negative, infinite or zero slope, $D(K_i,p,L)$ can be solved in time $O(n^4)$ instead of $O(n^5 \log n)$. When L has infinite slope, α is held constant while β changes along L. When L has zero slope, α changes and β is held constant. When L has negative slope, α and β are changing in different directions along L, i.e. one is increasing while the other decreases. Stone [STO] proved that for the first two cases, there are only n breakpoints along L. Close examination of that proof shows that this holds also for L with negative slope. In these cases, $D(K_i,p,L)$ can be solved with n maximum flow computations by the fractional linear programming method of section 1.3.10. The time to solve $D(K_i,p,L)$ is then $O(n^4)$.

When L has positive slope, α and β increase or decrease together along L, and Stone's proof does not imply only n breakpoints along L. PM(A) must then be used to solve $D(K_i,p,L)$. The actual number of breakpoints along L of positive slope is, however, an open question.

Chapter IV: COUNTING BREAKPOINTS FOR PARAMETRIC PROGRAMMING PROBLEMS

In this chapter we examine the number of breakpoints possible for several parametric programming problems and cost models. We look first at a restricted parametric cost function for integer programming problems, and then look at linear parametric cost functions for minimum spanning tree, matroid, and shortest path problems. Finally we mention some implications of the counting results for the generalized Lagrange multiplier method, and the problem of finding efficient solutions in bi-criterion problems.

4.1 A Special Case

We first examine a special class of parametric cost functions $P(i) \equiv S(i) + \lambda T(i)$, where either $T(i) = 0$ or $T(i) = 1$, for every decision variable i. That is, the cost of each variable is either held constant or increased additively by λ.

For P an integer programming problem, let X be the set of all feasible solutions to P, and let K be the maximum sum of all decision variables in any solution $x \in X$. That is

$$K = \underset{x \in X}{Max} \sum_i x_i \ .$$

For example, for an n node graph, $K = n - 1$ for the minimum spanning tree problem, and $K = n(n - 1)/2$ for the problem of shortest paths from a single node s to all other nodes. In the former problem, $x_i = 1$ if and only if edge i is in the minimum spanning tree, and in the latter problem $x_i = d$ if and only if d shortest paths from s pass though edge i.

Theorem 4.1: Let P(X,λ) be a parametric integer programming problem and let K and the parametric costs P(i) be as above. Then the number of breakpoints of P(λ) is at most K.

Proof: For a given solution x ∈ X, the slope of the parametric cost of x is

$$\sum_i T(i)x_i \leq \sum_i x_i \leq K$$

since T(i) = 0 or 1. P(λ) is piecewise linear and convex, and since all variables are integer valued, the slopes of the line segments of P(λ) must decrease by integral amounts. Therefore, P(λ) can have at most K breakpoints. ▪

For example, the two node s to t path problem can have at most n - 1 breakpoints, and the s to all node shortest path problem can have at most n(n - 1)/2 breakpoints.

4.2 The Parametric Spanning Tree Problem

Let G = (N,E) be an undirected graph with n nodes and e edges, and costs S(i) and T(i) associated with each edge i in E. The cost of each edge i is given as S(i) + λT(i), a linear function of the variable λ. Let X be the set of all spanning trees in G, and recall that for a tree T ∈ X

$$S(T) \equiv \sum_i S(i)x_i \quad \text{and} \quad T(T) \equiv \sum_i T(i)x_i$$

where x_i = 1 if element i is in tree T and 0 if not. As before, the function P(λ) is defined as:

For a given value of λ

$$P(\lambda) = \underset{T \in X}{Min}\, S(T) + \lambda T(T).$$

Recall that P(λ) is piecewise linear and convex, and that points of discontinuity are called breakpoints.

Theorem 4.2: For a graph with n nodes and e edges, the number of breakpoints of $P(\lambda)$ is bounded from above by $2e\mathrm{Min}[\sqrt{n}, \sqrt{e-n}]$

To begin the proof of the theorem, assume for any two edges i and j, that $T(i) = T(j)$ implies $i = j$, and that the edges are numbered from 1 to e so that $i < j$ implies $T(i) < T(j)$. For T a spanning tree containing edge j and omitting edge i, define an i:j edge exchange as the entry of edge i into T, and the deletion of edge j from T. The distance of such an i:j edge exchange is defined as $j - i$.

The following facts will be needed to prove the theorem.

1. Given a graph with constant edge costs, the minimum spanning tree is determined by the order of the edge costs alone [LAW]. Therefore in the parametric problem, the breakpoints can only occur at values of λ where two edge cost functions intersect. This implies an upper bound of $O(e^2)$.

2. We may assume without loss of generality that the parametric cost functions are in general position, i.e. that no three cost functions of edges in G intersect at the same value of λ.

Proof of fact 2: Let L be a cost line which intersects two other cost lines at the same value of λ. Define δ to be the minimum distance from L to the intersection of any two lines, not including L. Shifting L by $\varepsilon < \delta$ guarantees that L intersects only one line at a time. Further, shifting L by such an ε produces cost orderings which contain all the original cost orderings and possibly some new ones. Hence shifting L by ε only increases the number of breakpoints of $P(\lambda)$. ▪

3. Let T and T' be two adjacent minimum spanning trees such that T is minimum for a range of λ up to λ^*, and T' is minimum for a range of λ beginning at λ^*. Then T and T' differ in exactly one i:j edge exchange, which is of positive distance, i.e. i enters and j exits, and $T(i) < T(j)$. Further, λ^* is the point of intersection of the i and j cost functions.

Proof of Fact 3: By facts 1 and 2, λ^* must be the intersection point of the cost functions of two edges of G, say i and j. By fact 2, the relative cost orders of all pairs of the edges, except (i,j), are unchanged at λ^*. Then every edge in T - {i,j} will be in T' since each is guaranteed to be the minimum cost edge in some cut set in the range $\lambda^* \pm \varepsilon$, for small enough $\varepsilon > 0$. Further, no edge not in T $\cup$ {i,j} will be in T', since each such edge is guaranteed to be the maximum cost edge in some cycle of G in the range $\lambda^* \pm \varepsilon$, for small enough $\varepsilon > 0$. These are necessary and sufficient conditions for containment or exclusion from T' (see [LAW]). By assumption $T \neq T'$ hence $T \oplus T' = \{i,j\}$. Say $j \in T$, then $j \notin T'$, $i \notin T$, and $i \in T'$. Since $P(\lambda)$ is convex, $T(T') < T(T)$, and hence $T(i) < T(j)$, and $i < j$. ▪

Proof of Theorem 4.2: Increasing λ from $-\infty$ to $+\infty$ induces a sequence of minimum spanning trees and, by fact 3, a sequence of edge exchanges, each of positive distance. There are n - 1 edges in each tree, and e edges in G, so the sum of the distances of all the exchanges cannot exceed (n - 1)(e - n + 1). To see this, consider a zero - one vector V of length e, containing exactly n - 1 ones. The position of the ones in V indicates the edges in a minimum spanning tree; as λ increases, the ones move monotonically from the right end of V to the left. At most (n - 1)(e - n + 1) moves can be made before all the ones are stacked at the left of V.

To sharpen the O(ne) bound above, recall from fact 3 that each breakpoint λ^* is associated with one i:j edge exchange, and λ^* is the point of intersection of the i, j cost functions. Therefore, no given edge exchange can occur more than once. For the vector V, this implies that a move of an indicator one between two given positions can occur at most once. Then to bound the number of edge exchanges, we establish a bound on the number of moves of ones in V by solving the following problem G1: What is the maximum number of moves of the n - 1 ones in V, each move from a higher index position in V to a lower index position, and no move repeated, such that the total distance of the moves does not exceed (n - 1)(e - n + 1)?

To maximize the number of right to left moves of ones in V, without exceeding the distance limit, and repeating no move, the optimal strategy is to make all the possible moves of smallest distance. That is, make all the possible moves of distance one, all the possible moves of distance 2 etc. until the total distance of (n - 1)(e - n + 1) is exceeded. There are at most e moves of any distance q, and so the total distance traversed by all the possible moves of distance q is at most eq. Then the maximum number of moves is bounded by e×m, where m is the first integer such that

$$e \sum_{q=1}^{m} q \geq (n - 1)(e - n + 1).$$

Now m < m' where m' is the first integer such that

$$e \sum_{q=1}^{m'} q \geq ne.$$

Therefore, m < $\sqrt{2n}$, and so the number of edge exchanges, and breakpoints is bounded by e$\sqrt{2n}$.

To establish an upper bound of e$\sqrt{2(e-n)}$, note that in V there are e - n + 1 zeros which move from left to right as the ones move from right to left. This proves the theorem. ▪

Matroids and Selection Problems

Note that in theorem 4.2 very few properties of the minimum spanning tree problem are used. In particular, the above bound uses only the facts that successive minimum spanning trees differ by one edge exchange of positive distance, and that no given edge exchange can occur twice. These properties also hold for the parametric version of any matroid optimization (minimum cost base) problem.

Corollary: Let M be a matroid with e elements, rank r(M) and dual rank $r(M_d)$. Then for any choice of costs S(i), T(i) for each element i in M, the maximum number of breakpoints of the parametric minimum cost base problem is bounded by e×Min[$\sqrt{r(M)}$, $\sqrt{r(M_d)}$].

Theorem 4.2 can also be used for selection problems which are not matroid problems. For example, consider an ordered list of e elements and the problem of selecting the n < 2e objects in positions 2k - 1, for k = 1 through n. If the cost of each object is given by a linear parametric function of λ, then over the entire range of λ there are at most e×Min[$\sqrt{n}$, $\sqrt{e-n}$] different selections possible.

Lower Bound

We now establish a lower bound on the number of breakpoints for the parametric minimum spanning tree problem.

Theorem 4.3: There exist graphs with e edges and cost assignments S(i), T(i) for each edge i, such that P(λ) has asymptotically 2e breakpoints.

Proof: Let C_0 be a cycle on n nodes, and let the cost functions of the edges of C_0 be such that each edge is the maximum cost edge in C_0 for some value of λ (see figure 4.1).

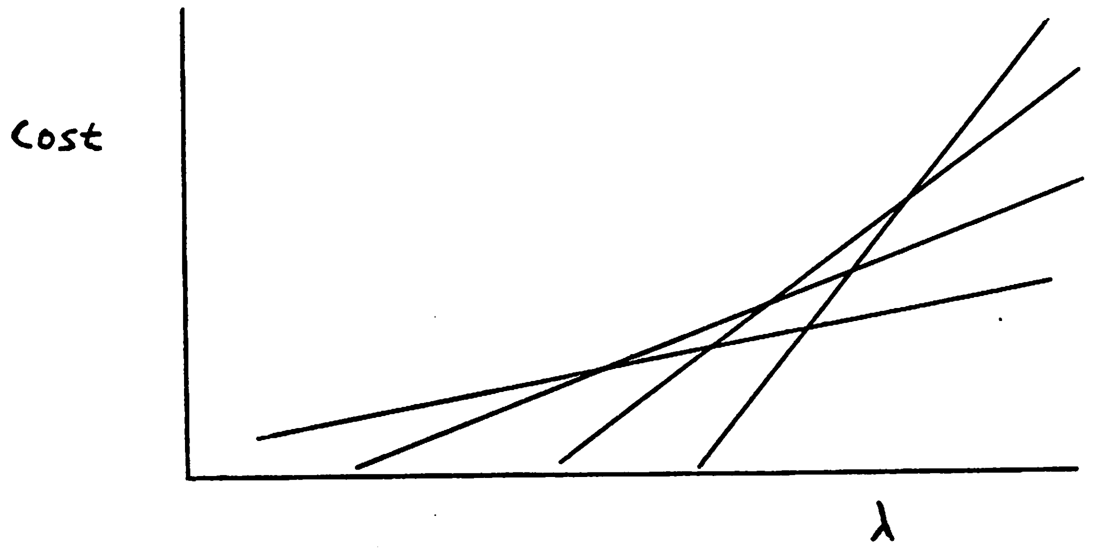


Figure 4.1: Costs of edges in cycle C_0

Then for any fixed λ, the minimum spanning tree of C_0 consists of all the edges except the maximum cost edge in C_0, and so for the full range of λ, there are n minimum spanning tree of C_0. Let C_1 be another cycle on the same n nodes, and edge disjoint from C_0. The costs of C_1 are like those of C_0; every edge of $C_1$is the maximum cost of the C_1 edges, for some λ. However, the cost functions of edges in C_1 are given lower slopes and larger intercepts than the edges in C_0, so that any edge in C_1 costs more than any edge in C_0 until the value of λ corresponding to the last minimum spanning tree of C_0. Figure 4.2 shows the intersections of the C_0 costs with the C_1 costs; the intersections of the costs inside the cycles (shown in figure 4.1) are represented by two circles.

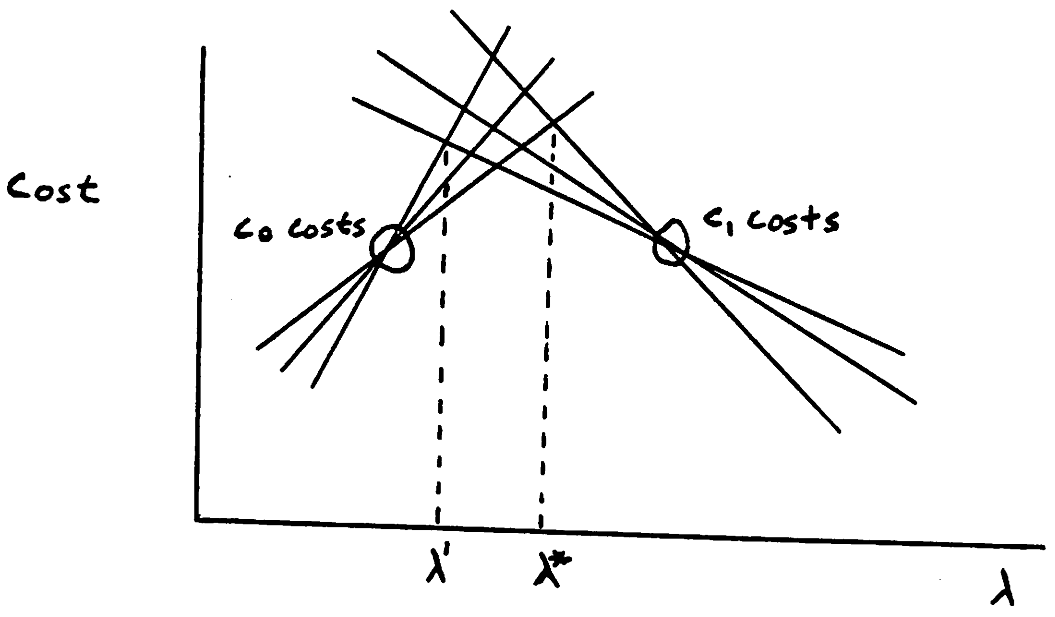


Figure 4.2

Let G = $C_0 \cup C_1$ and let λ' be the first value of λ where the costs of an edge from C_0 and an edge from C_1 intersect, and let λ* be the last value of λ where costs of edges from C_0 and C_1 intersect. By the choice of edge costs, there are n different minimum spanning trees of G for λ between 0 and λ', and these trees consist entirely of edges from C_0. There are also n different minimum spanning trees of G for λ between λ* and ∞, each consisting entirely of edges from C_1. For the range of λ strictly between λ' and λ* there are at least n - 2 different minimum spanning trees of G, each consisting partly of edges from C_0 and partly of edges from C_1. To see this, note that the minimum spanning trees before λ' and after λ* are edge disjoint, and that as λ increases the trees change by one edge exchange at a time. Hence for the graph G consisting of the cycle C_0 and C_1, there are 3n - 2 minimum spanning trees.

We can continue in this way, adding additional edge disjoint cycles $C_2,\ldots,C_K$ such that the spanning trees of cycle C_i become the minimum spanning trees of G only after all the spanning trees of C_{i-1} have been minimum. If K cycles have been added, then there are $Kn + (K - 1)(n - 2)$ minimum spanning trees on Kn edges. For a graph on n nodes there are up to $(n - 1)/2$ edge disjoint spanning cycles. By increasing n, the number of breakpoints is asymptotically 2e. ▪

4.3 Lowering the upper bound

In this section we discuss some efforts to lower the upper bound given in theorem 4.2. That is, efforts to lower the $O(e\sqrt{n})$ bound on the number of breakpoints in the parametric minimum spanning tree problem. We first discuss some additional constraints and conditions that were not used in the proof of theorem 4.2, and then show that even with these additional constraints we can't obtain an upper bound which is linear in e, the number of edges of the graph.

Additional Constraints

Recall that the $O(e\sqrt{n})$ bound was established by considering the motion of n - 1 ones in the vector V, subject only to the constraint that the moves are right to left and no move is ever made twice. These constraints were abstracted from the minimum spanning tree problem, but there are additional such constraints which were omitted. By including more constraints, we hope that a smaller upper bound can be obtained. Unfortunately, no such better bounds have yet been obtained. In this section we detail our efforts in this direction.

One seemingly powerful constraint not used for the above bound is on the order that the moves in V may be made. Consider edges $i < j < k$ and the three possible edge exchanges i:j, j:k, and i:k. In any sequence of spanning trees involving all three exchanges, the order of the exchanges must be such that i:k occurs between the other two exchanges. To see this, recall that the i:k exchanges occurs at the intersection of the i,k cost functions, and figure 4.3 shows that this intersection is always between the intersections of i,j and j,k. Then in V, the move sequence of k to j and j to i, followed later by a move of k to i, should be forbidden.

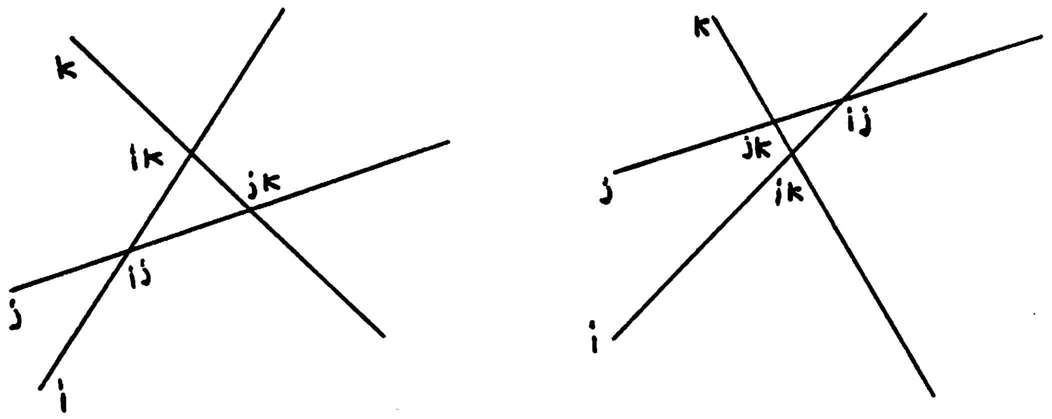


Figure 4.3

We can generalize this observation to get more constraints on the order of the edge exchanges. Let i < j < k be three edges in G, and consider the three edge exchanges i:j, i:k, j:k. What conditions on the S and T costs of the i, j and k edges determine the order in which the edge exchanges can occur? Let λ_{ij} and λ_{jk} be the intersection points of the cost functions of the j,k edges, and the i,j edges. Then

$$\lambda_{ij} = \frac{S(i) - S(j)}{T(j) - T(i)}$$

$$\lambda_{jk} = \frac{S(j) - S(k)}{T(k) - T(j)}$$

Lemma: $\lambda_{jk} < \lambda_{ij}$ if and only if

$$\frac{T(k) - T(j)}{S(k) - S(j)} < \frac{T(j) - T(i)}{S(j) - S(i)}$$

To see the consequences of the lemma, we plot the S, T costs of the i, j, and k edges on a plot P1 with the S costs on the horizontal axis, and the T costs on the vertical axis. We know that point k is to the upper left of the point j, which is to the upper left of point i. That is, if i < j < k and j:k, i:j are both edge exchanges, then S(i) > S(j) > S(k) and T(i) < T(j) < T(k). What the lemma then says is that the j:k edge exchange occurs before the i:j edge exchange if and only if on plot P1, the slope of the line connecting the k,j points is less than the slope of the line connecting the j,i points (see figure 4.4) i.e that on plot P1, the point j lies below the line between the k and i points. Conversely, the i:j edge exchange occurs before the j:k edge exchange if and only if j lies above the k to i line (see figure 4.4).

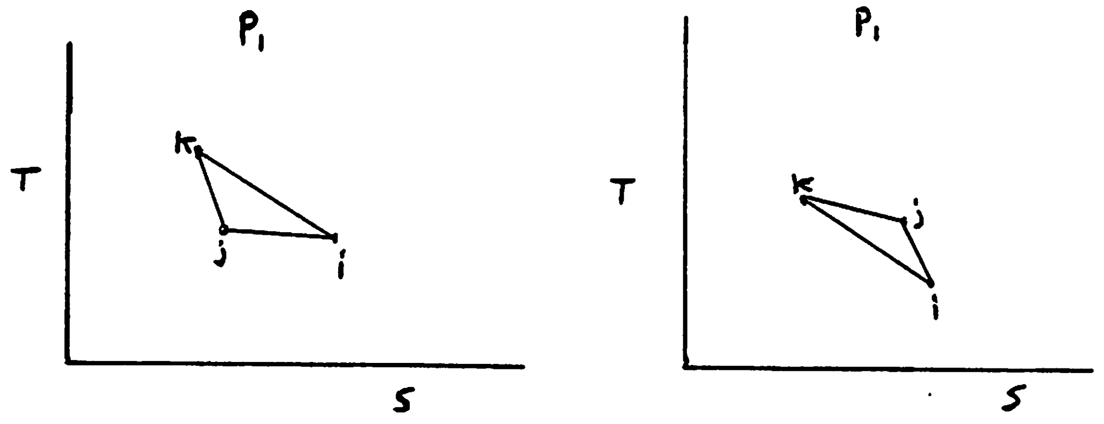


Figure 4.4

Now consider a sequence of successive edge exchanges j:k, i:j, h:i, g:h etc. where g < h < i < j < k. Then, in the S,T plot P1, the edge exchanges in the above order correspond to a piecewise linear concave decreasing curve through the k to g points (see figure 4.5). We call such a sequence of edge exchanges, and the corresponding curve, a *trajectory.* Define the *length* of a trajectory as the number of edge exchanges, or the number of distinct line segments on the associated concave curve. Two trajectories are *disjoint* if they contain no common edge exchange, i.e. no common line in the associated curve.

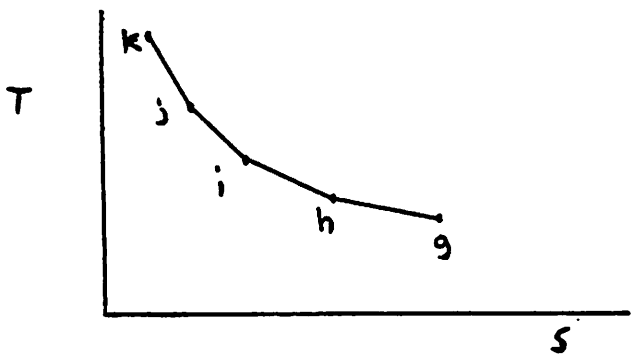


Figure 4.5

Lemma: Let G be an n node graph with edge cost parameterized by λ, and let EX be the set of all edge exchanges which occur as λ varies from λmin to λmax. Then EX can be decomposed into n-1 or fewer trajectories.

Proof: Let T be the minimum spanning tree of G at λmin, and let every edge in T be marked with a special symbol, say O. In each of the successive edge exchanges, the mark O is transferred from the exiting edge to the entering edge. If we follow the motion of a single mark O, then the ordered set of edges that it marks defines a trajectory of edge exchanges. There are only n-1 marks, hence EX can be decomposed into n-1 or fewer trajectories. The trajectories are disjoint because no edge exchange occurs twice. ▪

Hence, in the plot P1 of the edge costs, the edges exchanges determine n - 1 disjoint piecewise linear concave decreasing curves through points in P1. Further, if two trajectories, A and B, pass through the same point i in the plot, then the order of the entering and exiting slopes of the two trajectories are related. If the slope of A entering i is more negative than the slope of B entering i, then the exiting slope of A is also more negative than the exiting slope of B (see figure 4.6). This models the fact that every tree of G has exactly n - 1 edges and hence no edge has more than one mark at the same time.

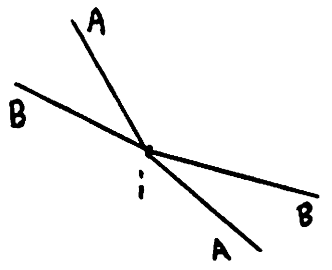


Figure 4.6

We can now bound the number of breakpoints in the parametric minimum spanning tree problem by solving the following problem G2:

Problem G2:

> Over all possible placements of e points on the plane, what is the maximum total length D of n - 1 line disjoint piecewise linear concave decreasing curves passing through points on the plane, such that when two curves pass through the same point, the order of their entering slopes is the same as the order of their exiting slopes?

This problem can be viewed as a two dimensional version of problem G1 which led to the $O(e\sqrt{n})$ bound. If each point in P1 is projected onto the x axis, then each concave curve in P1 describes a sequence of left to right moves between points on the x axis. These moves can be interpreted as moves of indicators in the vector V in problem G1. Thus problem G1 is a relaxation of problem G2. Unfortunately, we have no better bound for this problem than $O(e\sqrt{n})$ obtained before the inclusion of the timing constraints. In fact, we will show in the next section that no bound which is linear in e is possible for this problem.

We first state some limited results which follow from looking at G2.

1. If the e points are placed in the plane so that they can be decomposed into k disjoint convex curves, then D is bounded by (n-1)(2k-1).

   This follows because every trajectory is concave, and hence can intersect a convex curve at most twice. We can apply this to the lower bound construction G of theorem 4.3. Recall that G consists of K edge disjoint cycles on n nodes, and that a lower bound of $Kn + (K-1)(n-2) - 1 = (n-1)(2K-1)$ breakpoints was achieved. The costs of the edges in each cycle form a convex curve in the S,T plot P1, and so the points representing the edge costs of G can be decomposed into K disjoint convex curves. This says that the analysis of the lower bound is tight, and further that no matter how the costs of edges from different cycles are related, if the relative cost assignments inside each cycle are not changed, no larger lower bound is possible.

2. Two special cases of 1 are of interest. If the e points lie on one convex curve, then D is at most n - 1. If the e points lie on one concave curve, then D is at most e - n + 1.

3. Any two trajectories, A and B, share at most one half of all the points on either A or B. To prove this, suppose that i, j, and k are three contiguous points on trajectory A. If trajectory B also passes through points i and j, then it must pass through a point m between i and j, and it cannot pass through k (see figure 4.7). Hence out of every three contiguous points on A, B can pass through at most two, and for every three contiguous points on B, A can pass through at most two. It follows then that A and B share at most

one half of all the points on either A or B.

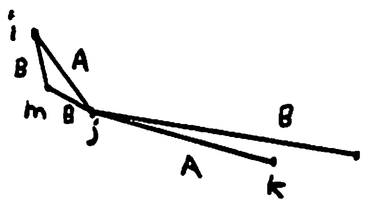


Figure 4.7

We had hoped that observations such as the ones above would lead to tighter upper bounds for the number of breakpoints, but no such bounds were found. In the next section we show that D cannot be bounded by any function linear in e.

Lower bounds for problem G2

In this section we give a lower bound construction for problem G2. showing that D cannot be upper bounded by any function linear in e. We first express problem G2 differently by representing each of the e points as straight lines on a plot P2 with λ on the horizontal axis, and cost on the vertical axis. For example, if point i = (5,3) in plot P1, then i is plotted as a straight line with y intercept of

5 and slope 3 in plot P2 (see figure 4.8).

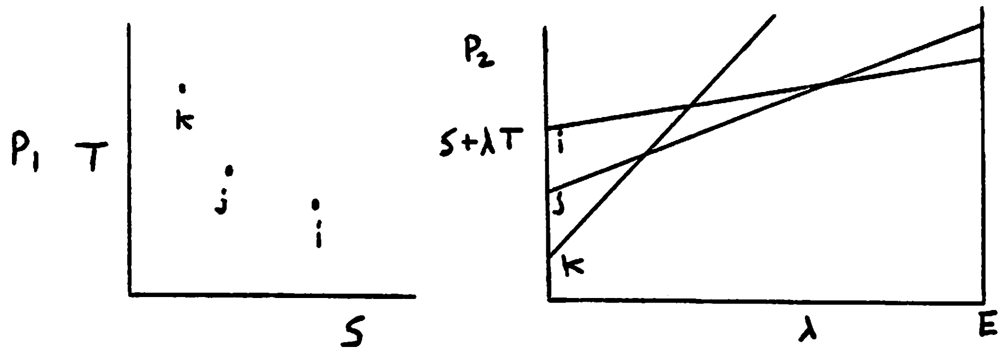


Figure 4.8

In plot P2 we place a vertical line E so that all the e(e - 1)/2 intersections of the e lines occur to the left of E. Then in plot P2, a trajectory from plot P1 is a piecewise linear convex curve starting at the y axis and ending at E, running along line segments of plot P2 and turning only at intersection points of the e lines (see figure 4.9).

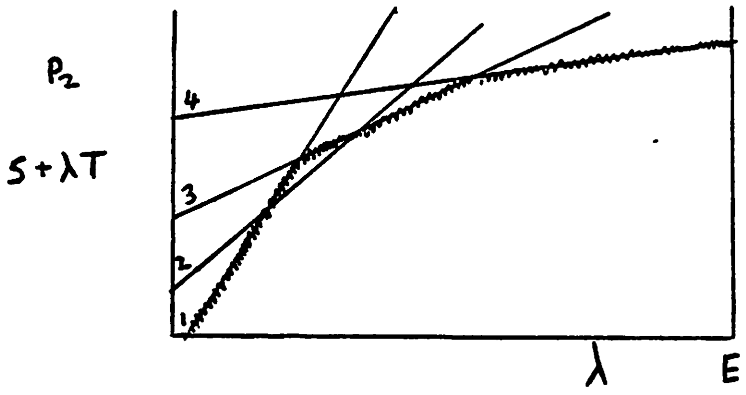


Figure 4.9: Trajectory runs along lines 1,3 and 4.

Two trajectories are disjoint if they do not both occupy any common line segment, and the length of a trajectory is the number of turns it makes.

In these terms, problem G2 is restated as follows:

Problem G2':

Over all possible placements of e straight lines on the plane, what is the maximum total length D of n - 1 disjoint piecewise linear convex curves starting at the y axis and ending at E, running along the line segments and turning only at intersections of the lines?

We will give a construction in which D can be made as large as c×e for any c, but first we note the following:

1. If in P1 the e points lie on one convex curve, then in P2 every one of the e lines lies on the upper envelope of all the lines, and if the e points lie on one concave curve in P1, then in P2 every one of the e lines lies on the lower envelope of all the lines (see figure 4.10). In the first case, D ≤ n - 1, and in the latter case D ≤ e - n + 1.

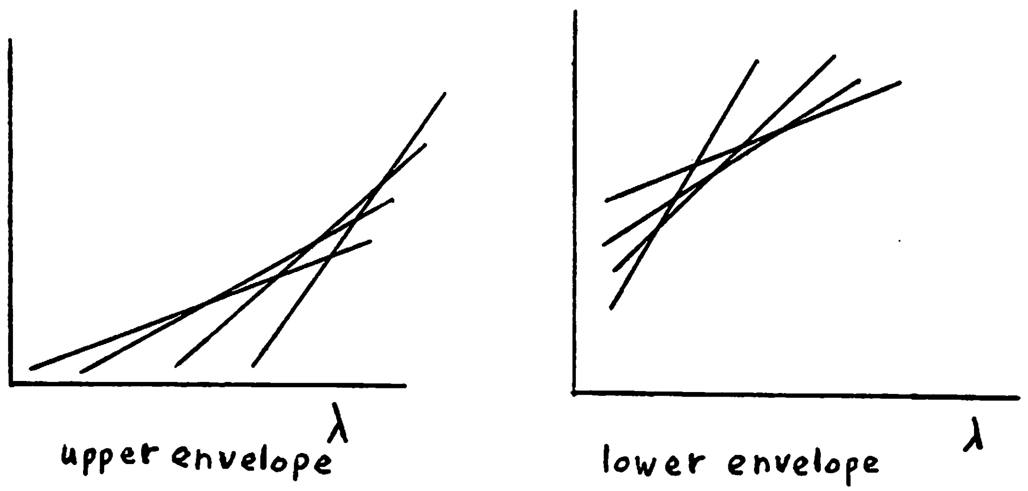


Figure 4.10

2. Given a fixed placement of the e lines, problem G2 can be solved algorithmically in polynomial time. This is a digression that is useful for running simulations of problem G2, and may lead to other ways to establish bounds on D.

The optimal set of n - 1 disjoint convex curves can be found in time $O(e^6)$ by solving an assignment problem on $O(e^2)$ nodes. We first express the problem as a maximum profit flow problem on a capacitated network G = (N,A). N contains nodes s and t and one node each for the $O(e^2)$ line segments between intersections of the e lines. The edge set A contains 2e + e(e - 1)/2 directed edges, each with capacity 1 and profit either 1 or 0. There are e directed edges from node s to the e nodes representing the line segments on the extreme left of each line in P2, and e edges directed into t from the e nodes representing the right ends of each line in P2. These edges each have capacity 1 and profit 0. Now consider an intersection of two lines in P2, and the four incident line segments. G contains one node for each of these four line segments (see figure 4.11), and these nodes are connected in G as in figure 4.12.

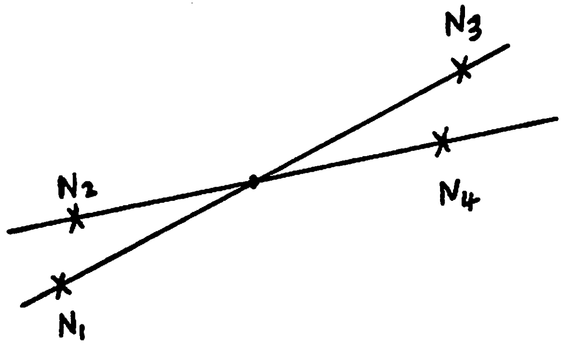


Figure 4.11

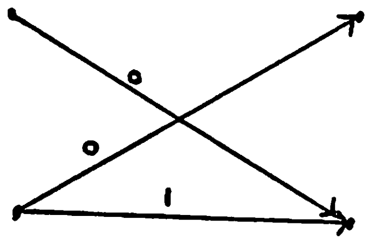


Figure 4.12

All capacities on these edges are 1, and the profits are written on the edges. Essentially, a profit of 1 is obtained for a convex turn, a profit of 0 for continuing straight, and concave turns are not permitted.

The maximum profit flow of quantity n - 1 from s to t determines the optimal n - 1 convex curves through the lines in P2. This maximum profit flow can also be written as an assignment problem, yielding an $O(e^6)$ algorithm.

Lower bound constructions

We now give a construction for $D \simeq 2e$, and then generalize this to $c \times e$ for any c. These constructions are due to Tim Winkler.

For any k, the construction C_k consists of $e = 4k - 1 + 2k^2$ lines, and achieves $D = 4k^2$. Then D is asymptotically 2e as k goes to infinity. C_k contains 2k lines forming a k×k rotated grid (see figure 4.13).

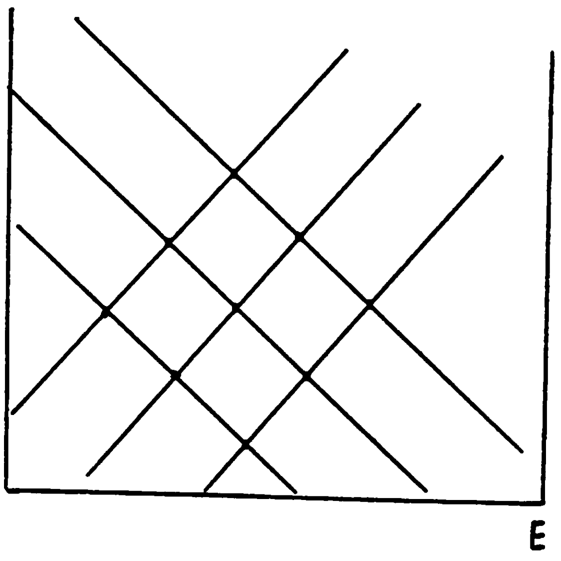


Figure 4.13: k = 3

To this grid, 2k - 1 lines are added so that a triangle is formed below every intersection point of the 2k lines forming the grid (see figure 4.14).

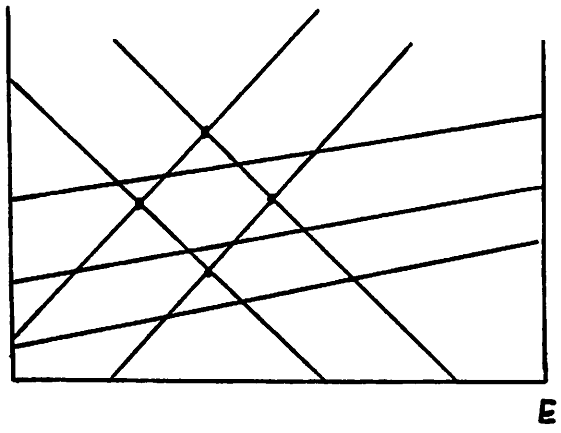


Figure 4.14: k = 2

Now for each of the k^2 triangles, two "feeder" lines are added as shown in figure 4.15. The left feeder line has positive slope, and intersects the line AC just below point A. The right feeder line has negative slope, and intersects the line BC just below point B.

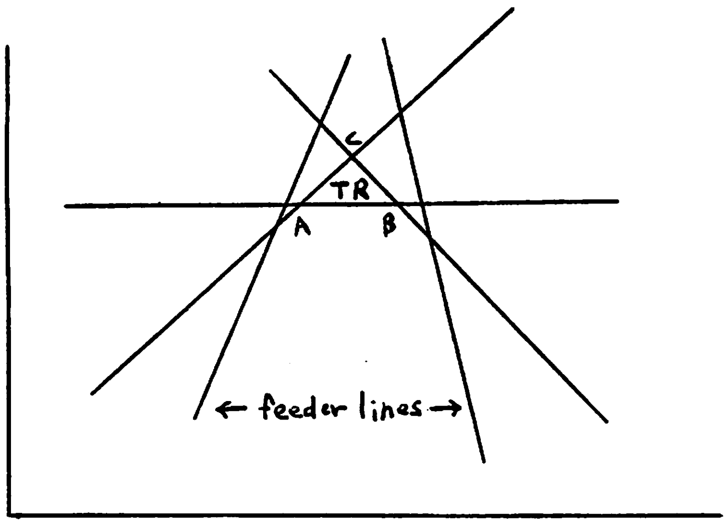


Figure 4.15: Feeder lines added to triangle TR.

Figure 4.16 shows a global picture of this construction for k = 2, but with many of the lines truncated for clarity.

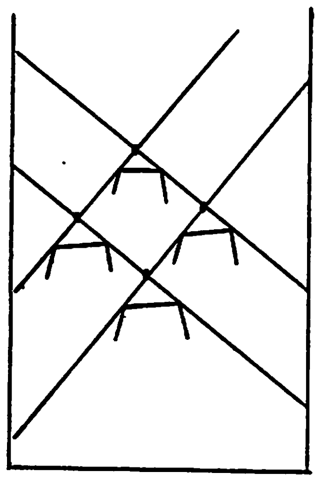


Figure 4.16

Four turns are achievable for each of the k^2 triangles by choosing for each triangle a convex curve as shown in figure 4.17. Hence $e = 4k - 1 + 2k^2$, and $D = 4k^2 \approx 2e$, and k^2 curves are used.

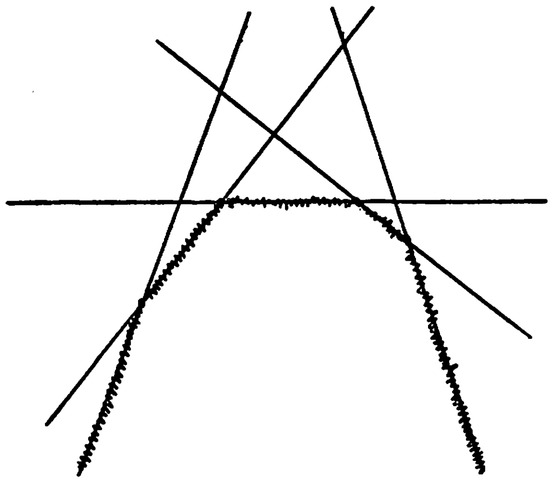


Figure 4.17

Generalization

We now generalize this construction to achieve $D \simeq (c\times e)/2$ for any given c. The construction, denoted $C_{c,k}$, again begins with a k×k grid, and will again contain $2k^2$ feeder lines, one pair for each of the grid points. However, instead of one line passing below every grid point, a *fan* of c lines will pass below each grid point. A *fan* is defined as a configuration of lines such that each one lies on the lower envelope of the lines (see figure 4.18). We will show below how to construct all the k^2 fans using only $(2k+1)c^2$ lines. We will also show that c + 2 turns are achievable from every fan plus two feeder lines. Then $e = 2k + 2k^2 + (2k+1)c^2$, and D is at least $(c + 2)k^2$. As k goes to infinity, $D \simeq (c\times e)/2$.

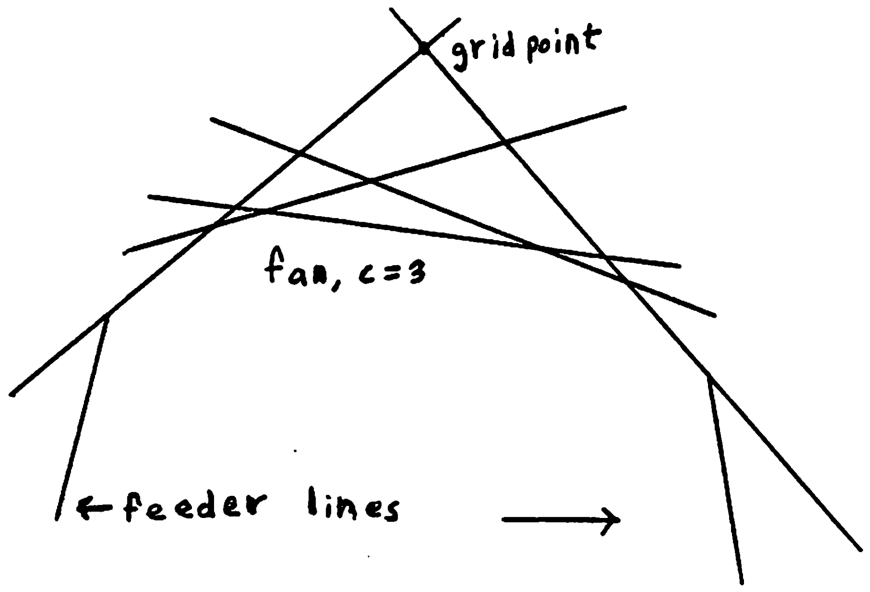


Figure 4.18

Construction $C_{c,k}$

We start $C_{c,k}$ with a k×k non-rotated grid (we will rotate later) of k horizontal and k vertical lines with distance exactly one between every neighboring pair of vertical or horizontal lines. We next add a set L of lines to the grid so that exactly c distinct new lines pass through each of the k^2 grid points. Further, each line in set L must have a slope equal to 1/d, where d is an integer from 1 to c. Then one line each of the c different slopes hits every point in the grid. Let $G_{c,k}$ denote the construction to this point.

Lemma: L contains at most $(2k+1)c^2$ lines.

Proof: Consider a (k+c)×(k+c) grid of vertical and horizontal lines with each neighboring pair at distance one. Now consider the upper right- hand k×k sub-grid, and add the set L to create $G_{c,k}$ on the upper sub-grid.

If the lines in L are extended so that they pass through the entire (k+c)×(k+c) grid, then each line will hit some point outside of the upper sub-grid. This is because each line in set L has one a slope equal to 1/d for d an integer from 1 to c, and the distance between two neighboring grid lines is one (see figure 4.19).

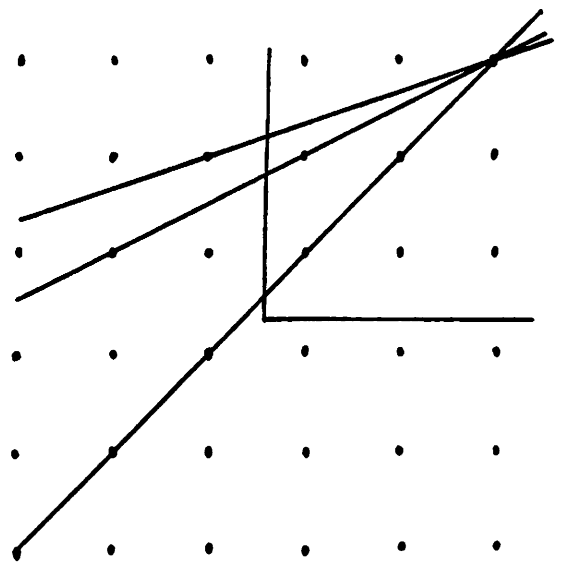


Figure 4.19: k = 3, c = 3. Three lines are shown.

There are $(2k + 1)c$ points outside the upper sub-grid, and each can be hit by at most c distinct lines of L, hence L contains at most $(2k + 1)c^2$ lines. ▪

We now shift the lines of L to create a fan below each of the grid points in $G_{c,k}$. For clarity, this is done in two steps. First, translate all the lines of L down and to the right, so that each intersection of c lines of L moves $1/(c+1)$ unit down and $1/(c+1)$ unit to the right. Figure 4.20 shows such a translation from grid point A, for $c = 3$. Note that with a translation of $1/(c+1)$, all the lines of the fan intersect the interior of the grid line segments AC and AB.

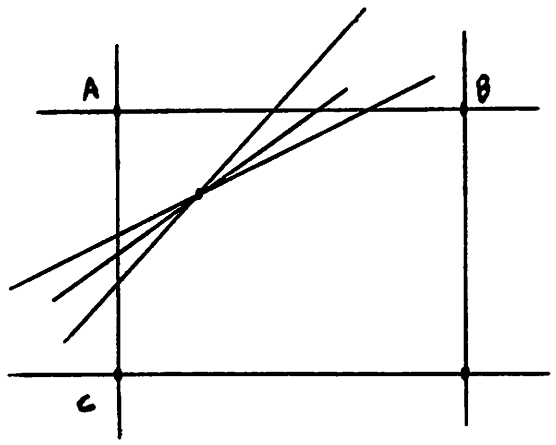


Figure 4.20

Now consider a single set of c lines that intersect at one point. By successively increasing the y - intercepts of the lines of slope 1/3 through 1/c, a fan can be created as shown in figure 4.21.

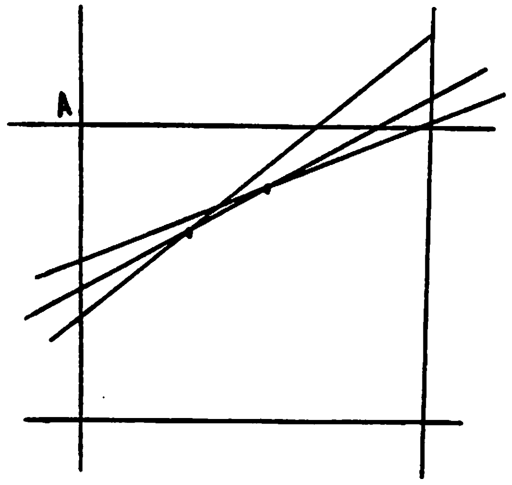


Figure 4.21

In creating a single fan, the y - intercepts of two lines of different slopes were changed differently. If all the lines in L of the same slope are translated by the same amount, then k^2 fans will be simultaneously created.

After the creation of the fans, the entire construction is rotated, $2k^2$ feeder lines are added, and k^2 convex curves are chosen, one for each fan, so that $D = (c + 2)k^2$. This construction shows that the constraints represented in problem G2 are not sufficient to establish a linear bound on the number of breakpoints in the parametric minimum spanning tree problem. Linear bounds are then only possible if we incorporate additional constraints such as those arising from the cycle cut-set structure of graphs.

Optimizing $\Gamma_{c,k}$

The construction $G_{c,k}$ can be optimized to improve the lower bound to $D \simeq e^{1.25}$. To do this, set $c = \sqrt{k}$ in $G_{c,k}$. It can be shown that this is the optimal value for c in $G_{c,k}$.

4.4 The Shortest Path Problem

Given a graph G and two distinguished nodes s and t, let P be the shortest path problem between nodes s and t, and let $P(X,\lambda)$ be the associated parametric problem. That is, associated with each edge i in G are two numbers S(i) and T(i) and the cost of edge i is then $S(i) + \lambda T(i)$, a linear function of the variable λ. For a given constant λ^*, $P(\lambda^*)$ is the cost of the shortest path from s to t with λ set to λ^*. $P(\lambda)$ is then a piecewise linear convex function of λ, and points of discontinuity are called breakpoints. We will show that the number of breakpoints of $P(\lambda)$ cannot be exponential in the number of nodes of G. This result is in contrast to the example of Murty [MU2] showing that for the parametric linear programming problem, the number of breakpoints can be exponential.

We begin with the parametric shortest path problem on a special graph $G_{n\times c}$.

Definition: $G_{n\times c}$ is a graph on $n\times c+2$ nodes consisting of two distinguished nodes s and t, and n columns of c nodes each. Edges of $G_{n\times c}$ extend from s to column 1, from column i to column i+1 ($1\leq i \leq n$), and from column n to t (see figure 4.22).

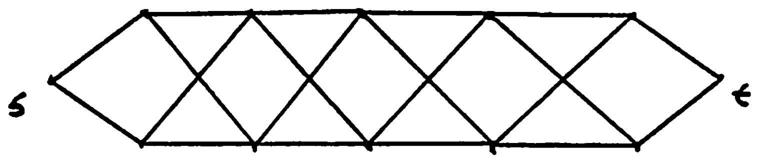


Figure 4.22: $G_{5\times 2}$

Theorem 4.4: For any choices of S and T costs, the number of breakpoints of $P(\lambda)$ for $G_{n\times c}$ is bounded from above by

$$\left\lceil \frac{2n+1}{2} \right\rceil c^{\log(2n+1)} = \frac{(2n+1)^{(\log c)+1}}{2}$$

where the log is base 2.

Proof: We prove first the claim that for n = 2^k - 1, k an integer, the number of breakpoints is bounded by

$$\frac{n+1}{2} c^{\log(n+1)} = \frac{n+1}{2}(n+1)^{\log c}$$

The proof of the claim is by induction on k.

For k = 1, the number of breakpoints of P(λ) for $G_{n \times c}$ cannot exceed c, since there are only c s to t paths, and each path is minimum for at most one contiguous range of λ. For k = 1 = n,

$$c = \frac{n+1}{2} c^{\log(n+1)}$$

and the basis is established.

Now suppose the claim is true for n = 2^k - 1, and consider $G_{n \times c}$ for n = 2^{k+1} - 1. We define G' as the concatenation of two copies of $G_{m \times c}$, where m ≡ 2^k - 1 (see figure 4.23).

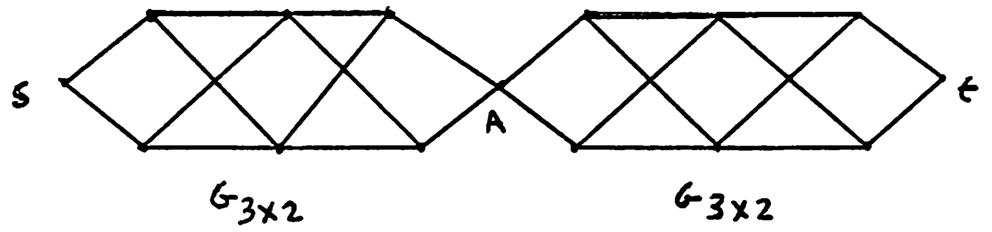


Figure 4.23: G'

The subgraph of G' consisting of all nodes from s to A is $G_{m \times c}$, as is the subgraph consisting of all nodes from A to t.

All paths from s to t pass through node A, and hence for any value of λ, the shortest path from s to t consists of the shortest path from s to A, followed by the shortest path from A to t. As λ changes, the shortest path from s to A changes independently of the shortest path from A to t. Therefore, the number of breakpoints of $P(\lambda)$ in G' is at most twice the number in $G_{m \times c}$. By the induction hypothesis, the number of breakpoints of $P(\lambda)$ in G' is at most

$$2\left[\frac{m+1}{2}\right]c^{\log(m+1)}$$

where $m \equiv 2^k - 1$. Now consider $G_{n \times c}$ for $n = 2^{k+1} - 1$. Let M be the middle column of $G_{n \times c}$, and let A be any arbitrary node in column M. By the analysis of graph G', the number of shortest paths from s to t in $G_{n \times c}$ which pass through node A is bounded by

$$2\left[\frac{m+1}{2}\right]c^{\log(m+1)}$$

Now, there are c nodes in column M, and hence the number of breakpoints of $P(\lambda)$ in $G_{n \times c}$ is a most

$$2c\left[\frac{M+1}{2}\right]c^{\log(m+1)} = \frac{n+1}{2}c^{\log(n+1)} = \frac{n+1}{2}(n+1)^{\log c}$$

for $n = 2^{k+1} - 1$, which completes the induction proof of the claim.

We can now finish the proof of the theorem for n an arbitrary integer. Let $k = \lceil \log_2 n \rceil$. Then $G_{n \times c}$ can be embedded into $G_{m \times c}$ for $m = 2^k - 1$, so that the shortest paths in $G_{m \times c}$ determine the shortest paths in $G_{n \times c}$ (see figure 4.24)

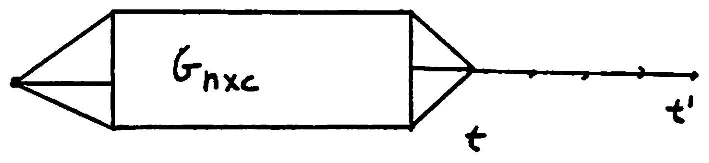


Figure 4.24:

Now $m \leq 2n$ and so the number of breakpoints of $G_{n \times c}$ is bounded by

$$\frac{(2n+1)(2n+1)^{\log c}}{2} = \frac{(2n+1)^{\log c + 1}}{2} .$$

Theorem 4.5: Let G be any arbitrary graph with n nodes and two distinguished nodes s and t. Then for any parametric edge costs, the number of breakpoints of $P(\lambda)$ for G is bounded from above by

$$\frac{(2n+1)^{(\log n)+1}}{2}$$

Proof: Define graph $G_{n \times n}$ as follows: $G_{n \times n}$ contains two distinguished nodes s and t, and n columns containing n nodes each. Node s is adjacent to node i in column 1 of $G_{n \times n}$ if and only if s is adjacent to node i in G. Similarly, the nodes in column n which are adjacent to node t in $G_{n \times n}$ are exactly those adjacent to t in G. Each node i in any column k is adjacent to node $j \neq i$ in column k+1, if and only if i is adjacent to j in G. The parametric costs of the above arcs are the parametric costs of the associated arcs in G. In addition, each node i in column k is adjacent by a zero cost arc to node i in column k+1.

For any value of λ, the shortest s to t path in $G_{n \times n}$ defines the shortest s to t path in G. To see this, see the discussion of the Bellman - Ford shortest path method in [LAW]. Therefore, the breakpoints of $P(\lambda)$ for $G_{n \times n}$ and G are identical. Then theorem 4.1 implies that the maximum number of breakpoints of P(λ) in G is bounded by

$$\left\lceil \frac{(2n+1)}{2} \right\rceil (2n+1)^{\log n}$$

which grows slower than any exponential function of n. ▪

Note that the sub-exponential bound holds for any cost function with the property that no path is minimum for more than one contiguous range of λ. Note also that the two theorems 4.4 and 4.5 hold for many problem other than shortest path problems. In particular:

Corollary: Let P be any Dynamic Programming problem solvable by a formulation with n stages, each with at most c stages. Then the associated parametric function P(λ) has at most $O(n^{\log c})$ breakpoints.

Corollary: Let G be an s-t planar graph [LAW] with edge capacities given as linear functions of Then the number of breakpoints for the parametric maximum-flow minimum-cut problem is bounded by $O(n^{\log n})$.

4.5 Discovery of Efficient Solutions and the Generalized Lagrange Multiplier Method

Both theorems 4.2 and 4.5 have implications for the discovery of efficient solutions in multi-criteria problems, and to the generalized Lagrange multiplier method, implying a large number of duality gaps for some problems. We will illustrate these implications on several multi- criteria problems involving costs and distances.

Let G be an undirected graph with two distinguished nodes s and t, and with two weights w_{i1} and w_{i2} associated with every edge i. Let W be a fixed "target" and let Γ be the set of spanning trees T of G such that

$$\sum_i w_{i2} x_i \le W$$

where $x_i = 1$ if edge $i \in T$, and $x_i = 0$ if not. Similarly, let Φ be the set of all s to t paths P in G such that

$$\sum_i w_{i2} x_i \leq W$$

where $x_i = 1$ if edge $i \in P$, and $x_i = 0$ if not. Vector x is called the *characteristic vector* of T or P respectively.

Then we define two problems:

Problem 1: Given G, find the spanning tree $T \in \Gamma$ with characteristic vector x, such that $\sum_i w_{i1} x_i$ is minimized over all trees is Φ. That is, find the tree minimizing the first criterion, provided that it does not exceed the threshold W for the second criterion.

Problem 2: Find the s to t path $P \in \Phi$ with characteristic vector x, such that $\sum_i w_{i2} x_i \leq W$ is minimized over all paths in Φ.

Before discussing the generalized Lagrange multiplier method and its limitations, we note that it is unlikely that either problems 1 or 2 can be solved exactly by fast algorithms.

Lemma: Both problems 1 and 2 are NP - hard.

Proof: We show that the decision version of problem 1 is NP - complete. The proof for problem 2 is identical. The decision version of problem 1 is:

Given constants W_1 and W_2 does there exist a spanning tree T such that $\sum_i w_{i1} x_i \leq W_1$ and $\sum_i w_{i2} x_i \leq W_2$?

We reduce the subset-sum problem to problem 1. Given a set N of numbers $\{a_1, \ldots, a_n\}$ and a target B, the subset sum problem is to determine if there is a subset of N which sums exactly to B. It is reduced to problem 1 as follows: Let G be a chain consisting of n parallel edges as in figure 4.25. The first weight on each edge i is w_{i1}, and the second is w_{i2}. W_1 is set to $\sum_i a_i - B$ and W_2 is set to $\sum_i a_i + B$. Then the answer to problem 1 is yes if and only if the answer to the subset sum problem is yes. ▪

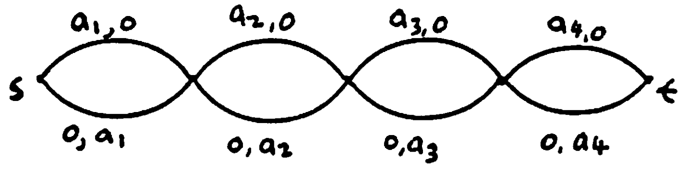


Figure 4.25

The Generalized Lagrange Multiplier Method

Given that problems 1 and 2 are both NP-hard, heuristic approaches such as the **Generalized Lagrange Multiplier Method** are often used to tackle such problems. The G.L.M. method or "Penalty Function Method" for problem 1 is the following:

1 For each edge i of G, associate the linear cost function $w_{i1} + \lambda w_{i2}$.

2. Pick a value λ^* for λ.

3. Substitute λ^* into all cost functions, and solve the induced minimum spanning tree problem $P(\lambda^*)$. Let T be the minimum spanning tree and let $WT_2 \equiv \sum_i w_{i2} x_i$.

4. If $WT_2 > W$ then increase λ^* and go to 3. If $WT_2 < W$ then decrease λ^* and go to 3. If $WT_2 = W$ or WT_2 is "sufficiently close" to W, or time runs out, then terminate.

Essentially, the G.L.M. method tries to penalize or encourage the use of the second resource in order to home in on a good solution to the two criteria spanning tree problem.

Limitations of the G.L.M. method

Recall from section 1.4 the definition of an *efficient solution.* Tree T is *efficient* if and only if no other spanning tree of G has smaller total weight for both weight criteria 1 and 2. The set of all efficient solutions is determined by G and its edge weights.

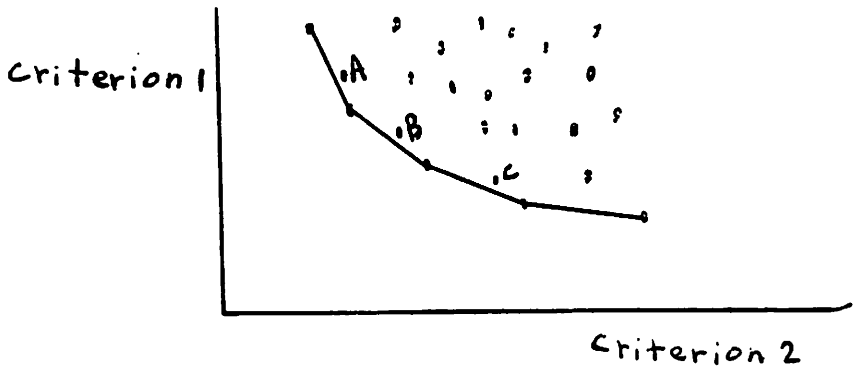
It is a simple fact that the solution to problem 1 is an efficient tree, and further, for any efficient tree T there exists a target W such that T is the solution to problem 1. It is also clear that step 3. of the generalized Lagrange multiplier method finds only efficient trees. One measure of the usefulness of the G.L.M. is the difference between the number of efficient trees in G, and the number of efficient trees discoverable by the G.L.M. method. Any efficient tree not discoverable by the G.L.M. method is said to lie in a *duality gap.*

Theorem 4.6: For any graph G with n nodes and e edges, there are edge weights w_{i1} and w_{i2} for each edge i such that every spanning tree of G is efficient. However, for no assignment of edge weights will the G.L.M. method discover more than $O(e\sqrt{n})$ efficient trees.

Proof: Given a graph G, pick edge weights w_{i1} arbitrarily and set w_{i2} to M - w_{i1}, where $M = \sum_i w_{i1}$. Then all the spanning trees are efficient because their weight order for the first criteria is the opposite of their order for the second criteria.

To see that the G.L.M. method can discover at most $O(e\sqrt{n})$ efficient points, note that every tree T discovered in step 3. is a tree associated with a breakpoint of $P(\lambda)$ for the parametric minimum spanning tree problem. That is, the G.L.M. method discovers only trees with costs on the lower envelope of all efficient trees. Then theorem 4.2 essentially says that there are at most

$O(e\sqrt{n})$ efficient trees on the lower envelope of all efficient trees (see figure 4.26). Note that in figure 4.26 points A, B and C are efficient but aren't on the lower envelope. ▪



Cost plot of all feasible solutions.

Figure 4.26

It is often useful to locate all the efficient points of a multi- criteria problem P. In general this is difficult to do. One approach that is often used for the bi-criteria problem is to find all the breakpoints of the related parametric problem P(λ), the sum of the first criterion plus λ times the second criterion. Every breakpoint of P(λ) is an efficient point, but the converse is not true and theorem 4.2 says that for problem 1 the parametric method may miss most of the efficient points.

A similar theorem for shortest paths will be stated without proof:

Theorem 4.7: For any n, there exist graphs with n nodes, and edge weights such that all s to t paths are efficient. However, at most $O(n^{\log n})$ paths are discoverable by the generalized Lagrange multiplier method.

Chapter V: SIMPLE CONSTRUCTIONS FOR MULTI - TERMINAL NETWORK FLOW SYNTHESIS

In this chapter we discuss the multi - terminal network flow synthesis problem. We present simple constructions which permit very rapid solutions to certain sensitivity analysis questions, and which have several other desirable properties.

5.1 Introduction

The multi-terminal network flow synthesis problem is one of the few nicely solved problems in the area of network design. It is used widely in courses and texts [LAW],[FOR],[FRA],[HU] on network flows and combinatorial optimization, as an example of an elegantly solved combinatorial optimization problem. The solution used in these texts is due to Gomory and Hu [GOM], and is also cited as an example of a non-direct application of maximum spanning trees. For examples where this problem arises, see also Chien [CHIE]. For NP - hard network design problems see Wong [WON], or Garey and Johnson [GAR].

We present simpler algorithms, improving the Gomory-Hu method in speed, simplicity of needed data structures, and most important, in the simplicity of the networks constructed. The networks constructed are uniformly optimal (defined in the next section), planar, have low node degrees, and have as few edges as any produced by the Gomory - Hu method. Routing algorithms for the networks are simple and can be implemented as rules applied locally at each

3. No node in G* has degree greater than four.

4. G* has as few edges as any uniformly optimal network produced by the Gomory - Hu method.

5. The structure of G* is easily expressed in terms of R.

6. Routing decisions in G* can be made locally at each node. This is a very desirable property for communications and computer network applications, and hierarchial data base applications.

Further, the algorithm shows that the use of the maximum spanning tree, and the revising of the original requirements by the Gomory - Hu method is unnecessary and undesirable.

5.3 Algorithms and Constructions

We first present a simple algorithm which constructs a planar uniformly optimal network with one node of high degree, and all other nodes of degree three of less.

Algorithm A

```
1)  For each node i, compute u(i) = Max[r(i,k)], and define u(n+1) = 0.
2)  Sort the u(i) values.  Assume u(i) > u(i+1) for i = 1,n.
3)  For i = 2 through n repeat the following:
        a. Create edge i,i-1 with capacity u(i)/2.
        b. Create edge i,1 with capacity [u(i) - u(i+1)]/2,
           provided that the capacity is non-zero.
```

Note that the algorithm uses only the u(i) values. All other data from R is ignored. Figure 5.1 shows the result of algorithm A. The requirements graph R is not displayed, but the u(i) values from R are written below each node.

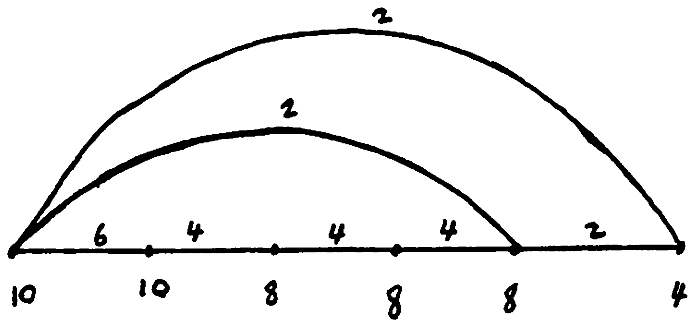


Figure 5.1:

Algorithm A requires time O(e) to find the u(i) in step 1), O(nlogn) to sort in step 2), and O(n) time to construct G* in step 3). Note that the network produced is always planar.

We now show the correctness of algorithm A. Given R, let G* be the network produced by algorithm A, and let f* be the flow function of G*.

Lemma 5.1: f*(i,j) = Min[u(i),u(j)] for all node pairs i,j.

Proof: Let i,j be two arbitrary nodes, and u(i) > u(j). Consider the path $P_{i,j}$ from i to j along the edges (k,k+1) for k = i through j-1. The edge with least capacity on $P_{i,j}$ is (j-1,j), with capacity u(j)/2. Therefore, a flow of u(j)/2 is possible along the $P_{i,j}$ path.

Now consider $P_{1,i}$, the path with edges (k,k+1) for k = 1 through i - 1. The edge with least capacity on $P_{1,i}$ is (i-1,i), with capacity u(i)/2 > u(j)/2. G* is undirected, and so a flow of u(j)/2 from node i to node 1 is possible along the reverse of path $P_{1,i}$.

To complete the proof, we claim that a flow of u(j)/2 between nodes 1 and j is achievable without using any edge of $P_{1,j}$, the union of $P_{1,i}$ and $P_{i,j}$. The proof is by backward induction on the index j. For j = n, the claim is true, since the edge (1,n) has capacity u(n)/2. Suppose the claim is true for $j = k+1 < n$, and consider node k. Let F(k+1) be the flow of u(k+1)/2 from 1 to k+1 which avoids edges of $P_{1,k+1}$. By definition, F(k+1) doesn't use edges (k+1,k) or (k,k-1), and so F(k+1) also doesn't use edge (1,k). Edge (1,k) has capacity u(k)/2 - u(k+1)/2, and edge (k+1,k) has capacity u(k+1)/2, for a total capacity of u(k)/2. Then to send u(k)/2 from 1 to k avoiding $P_{1,k}$, send u(k+1)/2 from k+1 to k along edge (k+1,k), and send the rest along edge (1,k). The flow of u(k+1)/2 to node k+1 is sent via F(k+1), and the proof is complete. ▪

Theorem 5.1: G* is uniformly optimal for R.

Proof: By lemma 5.1, G* is clearly feasible for R. To show optimality, note that the total capacity of the edges incident to any node i is u(i), which is the minimum capacity possible in any feasible network. Now suppose there is an optimal network G with flow function f, such that f(i,j) > f*(i,j), for some node pair i,j. Then f(i,j) > Min[u(i),u(j)], and if u(i) > u(j), node j must be incident in G to edges with total capacity exceeding u(j). Therefore, G can't be optimal, and G* is uniformly optimal. ▪

A low degree construction A'

The network G* produced by algorithm A has the undesirable property that node 1 has high degree. For applications involving ports into a computer, or wires wrapped on pins, small node degrees are desired. Algorithm A can be modified to produce, at equal speed, a planar uniformly optimal network G' with the same number of edges as G*, and with the property that no node of G' has degree greater than four.

We will not write the modifications formally, but give a generic description of G'. Let u(i) be defined as before, and call it the *node weight* of node i. Let t be the number of distinct node weights of R. We first give a generic description of G' for a special case of R.

Suppose R has 2k nodes with t = k distinct node weights $w_1 < w_2 < ... < w_k$, and with two nodes of each node weight. Then G' is a *ladder graph* with k rungs. The two nodes on the lowest rung are labelled w_1, the next two nodes are labelled w_2 etc. up to the top two nodes which are labelled w_k. Edges between nodes w_i and w_{i+1} are given capacity $w_i/2$, for i from 1 to k-1. All other edge capacities are forced by the rule that the total incident capacity at each node i is w_i, for i from 1 to k. Figure 5.2 shows network G' with eight nodes, four distinct node weights, and two nodes of each node weight. The circled numbers are node weights, and the other numbers are edge capacities.

Now we show how to remove the special case assumption, modifying the above construction G' for the case when R has more than two nodes of a distinct node weight, and for the case of exactly one node of a distinct node weight. If R has more than two nodes of a given weight $w_i \neq w_k$, then insert the additional w_i nodes into either of the w_i to w_{i+1} edges in G', creating a new edge with capacity $w_i/2$ for each insertion (see figure 5.3). These additional nodes then each have incident capacity of w_i, and have the same flow functions of the first two w_i nodes. If R has more than two nodes of weight w_k, then split the rung between the two w_k nodes into two parallel edges, one with capacity $w_k/2$, and the other with capacity $(w_k - w_{k-1})/2$, and then insert the additional nodes as above into the edge with capacity w_k. If R has only one node of a given weight w_i, then merge either one of the w_i nodes in G' with the unique w_{i+1} node that it is incident with, creating a w_{i+1} node of degree four. Figure 5.4 shows the network of figure 5.2 with unique node weights for all nodes except w_4.

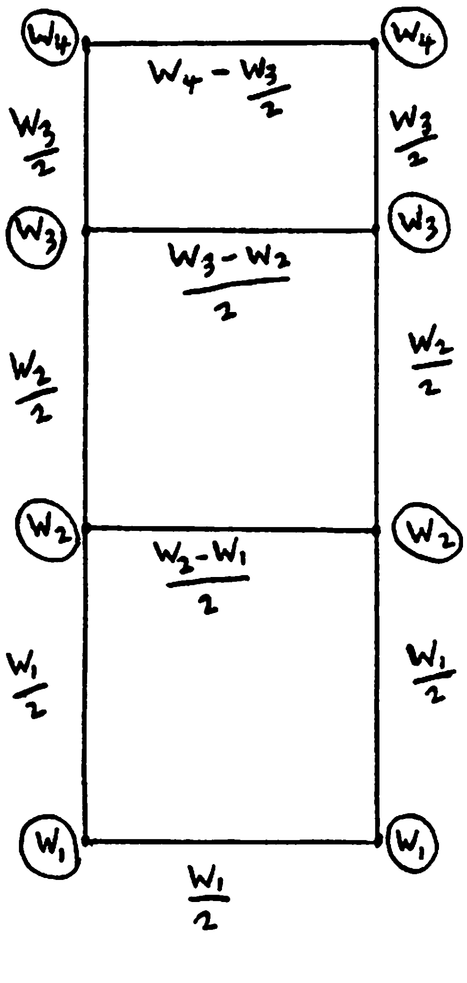


Figure 5.2

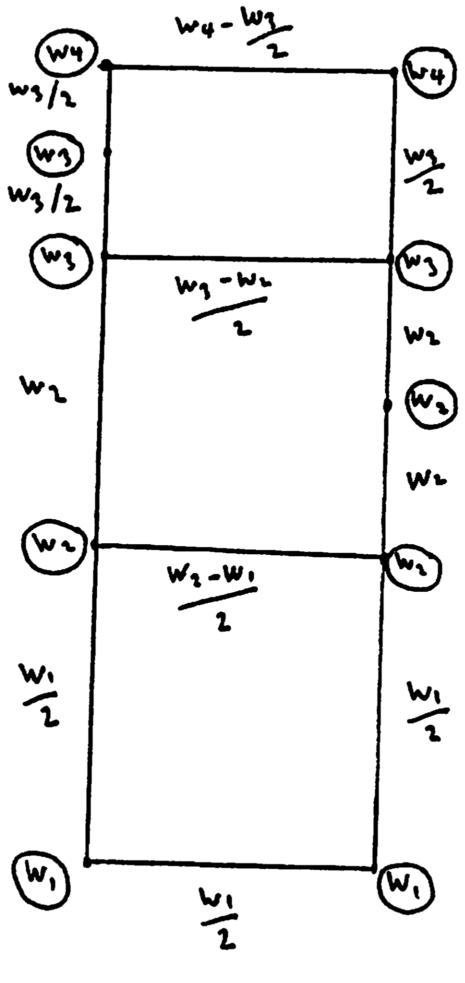


Figure 5.3

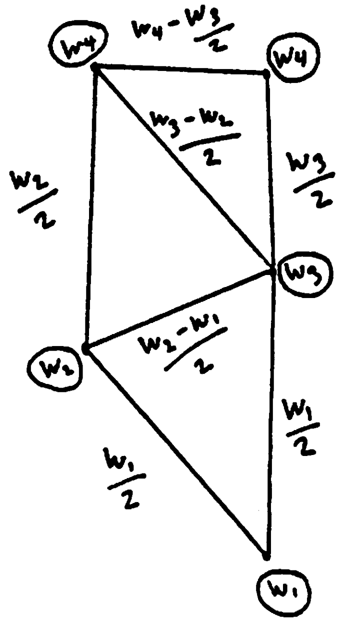


Figure 5.4

It is easily proven that G' is uniformly optimal, and has the same number of edges as G*. G' is clearly planar, and no node has degree greater than four. Further, the number of nodes of degree four equals the number of unique node weights (node weights assigned to only one node). Hence if every node weight is given to at least two nodes, no node will have degree greater than three.

Routing in G'

Routing algorithms for G' are extremely simple, and can be implemented by rules applied locally at each node. Assume that no node weight is unique, so that no node has more than three neighbors. For flow from node i to a node j of higher node weight, the routing rule at each node k is the following: If the flow into k is from a node of lower weight, then send as much as possible to k's neighbor of higher weight, and the rest to its neighbor of equal weight. If the flow into k is from a node of higher weight, send as much as possible to k's neighbor of lower weight, and the rest to it's equal weight neighbor. If the flow is from a node of weight equal to k, send it to a different neighbor of equal weight, if any, else to it's neighbor of higher weight. Flow from node i to a node of higher weight has equally simple rules.

Transparency and sensitivity analysis

Note that the edge capacities of G' are all simple functions of the node weights in R: either a node weight divided by 2, or the difference of two node weights divided by 2. Note also that the underlying graph of G' depends only on the number of distinct node weights, and the number of nodes of each distinct weight. This is the *transparency* property of G'.

G' is extremely nice for purposes of sensitivity analysis; small modifications of the requirements graph R induce simple local modifications in G'. If the requirements are modified, but no node weight in R is changed, then G' remains unchanged. If node weights are changed, but the order of the node weights is unchanged, then the capacities of G' change in a simple manner, but the graph reamains the same. If the order of two node weights in R are interchanged, then their corresponding rungs are interchanged in G'. If nodes or edges are added or deleted from R, then nodes and edges are added or deleted from G' in a simple manner with a few local changes in G'. Further, as modifications are made to G', the routing in G' is also updated by local modifications.

5.4 Number of edges

We now consider the number of edges in the construction G'. We show that there are uniformly optimal networks with fewer edges, but that none of these networks are producible by the Gomory - Hu method. In order to prove this, we must describe in detail the Gomory - Hu method.

The Gomory - Hu method

1) Given the requirements graph R, compute a maximum weight spanning tree T of R.

2) Decompose T into a sum of subtrees, each having edges of equal weight. To do this, define the decomposition of a tree T_i recursively. If w_i is the smallest edge weight in T_i then T_i is decomposed into one copy of T_i with weight w_i on each edge, plus the decomposition of each subtree of T_i resulting from deleting all edges of weight w_i from T_i, and subtracting w_i from the weights of all the remaining edges.

3) For every tree T_i in the decomposition, create a cycle C_i containing all the nodes of T_i. Set the capacity of every edge in C_i to $w_i/2$, where w_i is the weight of each edge in T_i. Superimpose all of the cycles, merging common edges and summing the capacities. The resulting network is optimal for R.

The above method produces an optimal network G for R, but in general G will not be uniformly optimal. To find a uniformly optimal network, Gomory and Hu suggest adding the following step at the beginning of the algorithm:

0) For each node i in R, compute $u(i) = \mathrm{Max}[r(i,k)]$. For every pair i,j change $r(i,j)$ to $\mathrm{Min}[u(i),u(j)]$, making R a complete graph on n nodes.

We will show that the Gomory - Hu method never produces a uniformly optimal network with fewer edges than G' produced by construction A'. Let R be the requirements graph modified by step 0) of the Gomory - Hu method; let T be any maximum spanning tree of R, and let G be the resulting uniformly optimal network produced by the method.

Lemma 5.2: Let x be any edge in T. If x has weight w_x, then the removal of x from T creates two connected components, at most one of which contains an edge weight greater than w_x.

Proof: The lemma is trivially true if one of the endpoints of x is a leaf of T, so suppose this is not the case. Let G_y and G_z be the two components of T - x, with edge y of weight $w_y > w_x$ in G_y, and edge z of weight $w_z > w_x$ in G_x. By the definition of R, $w_y = u(i)$ for some node i in G_y, and $w_z = u(j)$ for some node j in G_z. Then R contains the edge (i,j) across the G_y, G_z cut, and (i,j) has weight greater than w_x, hence T can't be a maximum weight spanning tree of R. ▪

Theorem 5.2: G contains at least as many edges as G'.

Proof: We examine first the number of edges in G. Let $w_1 < w_2 < \dots < w_t$ be the t distinct node weights of R, i.e. the t distinct values of u(i) for i = 1 through n. Because of the modification of R in step 0), the only edge weights in T are w_1 through w_t, and all edges incident with any node i in T have weight less than or equal to u(i). Further, since T is a maximum spanning tree, every node i is incident in T with at least one edge of weight u(i). It follows then from Lemma 5.2, that for any h < t, the deletion from T of all edges of weight w_h or less leaves one connected subtree containing all nodes with node weights greater than w_h, and no nodes with weight w_h or less.

We now examine the edges generated by the synthesis step 3), ignoring the capacities assigned. We claim that G is the superposition of t cycles, C_1 through C_t. C_1 connects all the n nodes of T, and for $h > 1$, C_h contains all nodes of weight w_h or more, and no nodes of weight w_{h-1} or less. To see this, recall that the decomposition step 2) of the Gomory - Hu method generates a sequence of subtrees of T, by beginning with T itself, and successively deleting all edges of weight w_1 to w_t. Step 3) creates a cycle through the nodes of every new subtree generated in this way, and the claim follows from the structure of these subtrees, which was established above.

We can now count the number of edges of G. For h from 1 through t, let N_h be the number of nodes of weight w_h. For $h < t$, cycle C_h contains all nodes of weight w_h, and at least one node of greater weight. Therefore, at least $N_h + 1$ edges of C_h are incident with nodes of weight w_h. None of these $N_h + 1$ edges can appear in any other cycle C_j, for $j > h$, and so the cycles C_1 through C_{t-1} must contain at least $(n - N_1) + (t - 1)$ distinct edges. Cycle C_t contains N_t edges, and so G contains at least $n - 1 + t$ edges if $N_t > 2$, and $n - 2 + t$ edges if $N_t = 2$.

Now we show that G' contains no more than this many edges. If t is the number of distinct node weights in R, then G' contains $t - 1$ rungs between the nodes of weight $w_h \neq w_t$. G' contains one rung between the w_t nodes if there are only two nodes of that weight, and two rungs if there are more. Note that there can never by just one node of the largest node weight. In addition, G' contains $n - 2$ edges which form the two sides of the ladder. Hence G' contains $n + t - 1$, or $n + t - 2$ edges depending on the number of nodes of largest node weight, and the theorem is proved. ▪

Corollary: G contains the same number of edges as G' if and only if the nodes of weight w_h form a single subpath in C_h for all h from 1 to t.

Note that without step 0), lemma 5.2 does not hold, and the Gomory - Hu method may produce a non - uniformly optimal network G with fewer edges than G'. Note also that there are uniformly optimal networks with fewer edges than G', but such networks are, of course, not produced by either constructions A or A', or by the Gomory - Hu method. It remains an open question whether there exist fast algorithms to minimize the number of edges in a uniformly optimal network.

Chapter VI: PROBLEMS FOR FUTURE RESEARCH

We list several promising questions and areas for additional research.

1. Reduce the upper bound on the number of breakpoints for the parametric minimum spanning tree and shortest path problems. The $O(n^{\log n})$ bound for the shortest path problem, in particular, seems ripe for improvement.

2. Generalize the Eisner and Severance parametric programming method for multi-dimensional problems. i.e. find a method that works for multidimensional linear parametric cost functions regardless of the underlying problem. A good generalization must have the property that the number of optimizations is related by a small function to the number of vertices and edges in the parametric decomposition of the parameter space. For example, in two dimensions, the parameter space decomposes into convex polygons (see section 3.4.), and a method in which the number of optimizations is linear in the number of edges and vertices of the polygons seems possible.

3. Determine the number of breakpoints possible for the parametric minimum cost flow problem [LAW]. The Zadeh and Murty examples (see section 1.3.5) force an exponential number of breakpoints for general linear programming problems. Theorem 4.5 shows that for shortest paths the number of breakpoints is less than exponential. Minimum cost flow is a generalization of shortest paths, and a specialization of general linear programming. It is natural, then, to ask how many breakpoints are possible for the minimum cost flow problem. Note that Zadeh's example does not resolve this, since in his example the capacities,

not the costs, are parameterized. A parametric cost problem is obtained by taking the dual, but the dual problem is not a network flow problem.

There is another strong motivation for solving this problem: If the number of breakpoints is polynomial, then the minimum cost flow problem can be solved in polynomial time in the number of nodes and edges of the underlying graph (in contrast to the polynomial time bound given by the ellipsoid or scalling methods). This would be an important result, for the minimum cost flow problem is probably the most natural and useful of all the standard flow models.

Let c(i) be the cost of flowing one unit along edge i, and let x_i denote the flow assigned to edge i. Then the cost of the flow is $\sum_i c(i)x_i$. The idea of the polynomial algorithm is the following:

A. Find a maximum *quantity* flow disregarding edge costs. This can be done in polynomial time by a number of different methods. Let V be the maximum flow quantity, and let x be the vector giving the values of the edge flows.

B. Find a cost vector c* such that x is a minimum cost flow of quantity V relative to c*. For example, let c*(i) = 0 if $x_i > 0$, and c*(i) > 0 otherwise.

C. For each edge i, assign the linear parametric cost function c*(i) + λ[c(i) - c*(i)] Then x is the optimal for $\lambda = 0$, and the optimal flow for $\lambda = 1$ is the minimum cost flow of the original problem.

D. For $\lambda = 0$ to 1, find the breakpoints and the associated minimum cost flows. This can be done in polynomial time per breakpoint as noted in section 3.3.

Hence the question of the number of breakpoints in the minimum cost flow problem is a natural one, yielding either an exponential example of the above method, or an algorithm that is polynomial in the number of nodes and edges. The first case would be in keeping with all other simplex like methods for minimum cost flow. Note that a polynomial number of breakpoints for the minimum cost flow problem implies a polynomial number for the shortest path problem. Hence reducing the $O(n^{\log n})$ bound for shortest paths seems like the correct first step for showing a polynomial bound for minimum cost flow.

4. Use NP - completeness theory to indicate that an exponential number of breakpoints exist in certain parametric programming problems. Consider the following problem P(c,d): G = (N,E) is a graph with distinguished node r and weights c(e) and d(e) on each edge e. For a spanning tree T, p(i,r) is the unique path in T from node i to node r. Then the distance of p(i,r) is $\sum_{e \in p(i,r)} d(e)$, and the total cost for T is $\sum_{e \in T} c(e) + \sum_{i \in N} d(i,r)$.

P(c,d) is a problem in combining initial fixed costs with variable usage costs. To make the model more useful, the distances are multiplied by λ, reflecting the level of traffic in the network. Then λ is varied to study the sensitivity of the solution to the traffic level.

One open question is how many breakpoints are possible in the function P(λ) for problem P(c,d), but a more important goal is to find a methodology for tackling such questions. If there are an exponential number of breakpoints, then we might be able to find a construction yielding that many breakpoints. Such constructions are, however, generally hard to find and don't help in finding exponential constructions for other problems. Instead we would like the following result: "P(λ) has an exponential number of breakpoints, unless P = NP". We might hope for such a result for the following reasons:

1. For arbitrary λ, evaluating P(λ) for P(c,d) is NP - hard, i.e. P(c,d) is NP - hard [GAR1].
2. Evaluating P(λ) for λ "large" is polynomial. That is, the following problem is polynomial: Find a tree T which minimizes $\sum_{i \in N} d(i,r)$, and among all such

trees, find one which minimizes the $\sum_{e \in T} c(e)$. Further, it is easy to determine a value λ* of λ which is "large" enough.

Now suppose we could use the solution at P(λ*) to find the solution at the first breakpoint before λ*, and in general that we could move from one breakpoint to the adjacent breakpoints. (We saw in chapters I and III examples of such methods for certain problems.) Then P(λ) could be determined for any arbitrary λ. This implies that for P(λ) either:

1. Moving from an arbitrary breakpoint to its neighboring breakpoints is NP - hard.

   or

2. There are an exponential number of breakpoints for P(λ).

   or

3. P = NP.

A polynomial method to move from one breakpoint to the next would then leave only the last two possibilities. Note that there are problems where moving from one breakpoint to the other is easier than the problem of finding solutions at an arbitrary λ. An example is the minimum cost flow problem (see chapter III).

Bibliography

[BEN] J.L. Bently, Decomposable Searching Problems, Information Processing Letters 8, 1979.

[BAL] V. Balachandran and G.L. Thompson, An Operator Theory of Parametric Programming for the Generalized Transportation Problem, Management Sciences Research Report No. 293, 1972, Carnegie-Mellon University, Graduate School of Industrial Administration.

[CHA1] R. Chandrasekaran, Minimal Ratio Spanning Trees, Networks 7 (1977)

[CHA2] R. Chandrasekaran and A. Daughety, Problems of Location on Trees, Discussion Paper No. 357, Northwestern University, Graduate School of Management, 1978.

[CHE] G.A. Cheston, Incremental Algorithms in Graph Theory, Technical Report No. 91, University of Toronto, Department of Computer Science, 1976.

[CHI] F. Chin and D. Houck, Algorithms for Updating Minimal Spanning Trees, J. of Comp. and Sys. Sciences 16, 333-344, 1978.

[CHIE] R.T. Chien, Synthesis of a Communication net, I.B.M. Journal, July 1960.

[ES] M.J. Eisner and D.G. Severance, Mathematical Techniques for Efficient Record Segmentation in Large Shared Databases, J. of the Assoc. for Comp. Mach. 23, No 4, 619-635, 1976.

[EVE] S. Even and Y. Shiloach, An On-Line Edge-Deletion Problem, Preprint

[FRA] H. Frank, and I. Frisch, Communication, Transportation and Flow Networks, Addison - Wesley, 1972.

[FOR] L.R. Ford, and D.R. Fulkerson, Flows in Networks, Princeton University Press, 1962.

[GT] H.N. Gabow and R.E Tarjan, Efficient Algorithms for Simple Matriod Intersection Problems, Twentieth Annual Symp. on Foundations of Comp. Sci., 1979.

[GAR] M.R. Garey, and D.S. Johnson, Computers and Intractability, A Guide to the Theory of NP - Completeness, W.H. Freeman and Co., 1979.

[GAR1] M.R. Garey, Personal communication.

[GOM] R.E. Gomory, and T.C. HU, Multi - Terminal Network Flows, SIAM Journal on Applied Mathematics 9, 1961.

[GOT] S. Goto and A Sangiovanni-Vincentelli, A New Shortest Path Updating Algorithm, Technical report, Electronics Research Lab. University of California, Berkeley, 1978. To appear in Networks.

[HU] Integer Programming and Network Flows, Addison Wesley, 1969.

[HAN] W.K. Klein Haneveld, C.L.J. van der Meer, and R.J. Peters, A Construction Method in Parametric Programming, Math. Programming 16, 21-26, 1979.

[JER] R.G. Jeroslow, Bracketing discrete problems by two problems of linear optimization, Proc. First Symp. on Operations Research, Verlag Anton Hain, Meisenheim, 1976.

[JOH] D.B Johnson, and S.D. Kashdan, Lower Bounds for Selection in X + Y and Other Multisets, JACM. Vol. 25, No. 4, 1978.

[LAW] E.L. Lawler, Combinatorial Optimization, Networks and Matroids, Holt, Rinehart and Winston, 1976.

[MEG] N. Megiddo, Combinatorial Optimization with Rational Objective Functions, Tenth Annual ACM Symposium on the Theory of Computing, 1978.

[MU] K.G. Murty, Linear and Combinatorial Programming, John Wiley and sons, 1976.

[MU2] K.G. Murty, Computational Complexity of Parametric Linear Programming, Technical Report No 78-6, University of Michigan, Ann Arbor, Department of Industrial and Operations Engineering, 1978.

[NEM] G.L. Nemhauser, G.M Weber, Optimal Set Partitioning, Matchings and Lagrangian Duality, Naval Research Logistics Quarterly, Vol. 26 No. 4, Dec. 1979.

[NAU] R.M Nauss, Parametric Integer Programming, Ph.D. dissertation, Working Paper # 226, Western Management Science Institute, UCLA, 1975.

[ORL] J.B. Orlin, J.J. Bartholdi, III, and H.D. Ratliff, Circular Ones and Cyclic Staffing, Technical Report No. 21, Stanford Univerity Department of Operations Research, 1977.

[PIC1] J. Picard and M. Queyranne, A Network Flow Solution of Some Non-Linear 0-1 Programming Problems and Applications to Graph Theory, Rapport Technique No. EP79-R-14, Ecole Polytechnique de Montreal, Department de Genie Industriel, 1979.

[PIC2] J. Picard and M. Queyranne, Selected Applications of Maximum Flows and Minimum Cuts in Networks, Rapport Technique No. EP79-R-35, Ecole Polytechnique de Montreal, Department de Genie Industriel, 1979.

[WON] A survey of network design problems, Operations research center working paper OR 080-78, August 1978, MIT.

[RAD] M.A. Radke, Sensitivity Analysis in Discrete Optimization, Ph.D. Thesis, Working Paper No. 240, University of California, Los Angelas, Western Managment Science Institute, 1975.

[ROS1] A. Rosenthal, Optimal Algorithms for Sensitivity Analysis in Associative Multiplication Problems, To appear, Theoretical Computer Science.

[ROS2] A. Rosenthal, An Optimal Algorithm for Sensitivity Analysis in Nonserial Optimization Problems, preprint.

[SCI] Scientific American, May 1980.

[SHA] J.F. Shapiro, A Survey of Lagrangean Techniques for Discrete Optimization, Preprint.

[SHE] Shen Lin, Heuristic Programming as an Aid to Network Design, Networks 5.

[SHI] D.R. Shier and C. Witzgall, Arc Tolerances in Shortest Path and Network Flow Problems, Preprint

[SOM] J.E. Somers, Parametric Network Flow Problems, Ph.D. Thesis, University of California, Berkeley, Mathematics Department, 1976.

[SPI] P.M. Spira and A. Pan, On Finding and Updating Spanning Trees and Shortest Paths, SIAM J. Comput. 4 no.3, 375-380, 1975.

[SRI1] V. Srinivasan and G.L. Thompson, Cost Operator Algorithms for the Transportation Problem, Math. Programming 12, 372-391, 1977.

[SRI2] V. Srinivasan and G.L. Thompson, An Operator Theory of Parametric Programming for the Transportation-I,

[STO] H.S. Stone, Multiprocessor Scheduling with the aid of Network Flow Algorithms, IEEE Trans. on Software Engrg. 3, 1977.

[THU] D.J. Thuente, Two Algorithms for Shortest Paths Through Multiple Criteria Networks, Preprint

[TOM] J.A. Tomlin, A Parametric Bounding Method for Finding a Minimum L-infinity-norm Solution to A System of Equations, Technical Report No 75-12, Stanford University, Systems Optimization Laboratory, 1975.

[VAN1] J. van Leeuwen, and M.H. Overmars Some Principles for Dynamizing Decomposable Searching Problems, Technical Report RUU-CS-80-1, University of Utrecht, Department of Computer Science, 1980.

[VAN2] J. van Leeuwen, and M.H. Overmars Dynamic Multi-Dimensional Data Structures Based on Quad- and K-D Trees, Technical Report RUU-CS-80-2, University of Utrecht, Department of Computer Science, 1980.

[VAN3] J. van Leeuwen, and M.H. Overmars Two General Methods for Dynamizing Decomposable Searching Problems, Technical Report RUU-CS-79-10, University of Utrecht, Department of Computer Science, 1979.

[VAN4] J. van Leeuwen and D. Wood, Dynamization of Decomposable Searching Problems, Inform. Processing Letters 10, no 2, 51-55, 1980.

[VW] L.N. van Wassenhove and L.F. Gelders, Solving a Bicriterion Scheduling Problem, European Jour. of Operational Research 4, 42-48, 1980.

[WAL] D.H. Walters, Multi-Parametric Mathematical Programming Problems, Ph.D. Thesis, Department of IEOR, University of California, Berkeley, 1976.

[WEB] G.M. Weber, Sensitivity Analysis of Optimal Matchings, Technical Report No. 427, Cornell University, School of Operations Research and Industrial Engineering, 1979.

[VAL] L.G. Valiant, The Complexity of Enumeration and Reliability Problems, SIAM J. Comput. 8, no 3, 410-421, 1979.

[YAM] Y. Yamamoto, The Held-Karp Algorithm and Degree-Constrained Minimum 1-Trees, Math. Programming, Sept. 1978.

[ZAD] N. Zadeh, A Bad Network Problem for the Simplex Method and Other Minimum Cost Flow Algorithms, Math. Programming 5, 1973.