Symbolic Layout Evaluator for Floor Plans

Amy Shih-Chun Hsu

Master's Project
under the direction of Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

March 30, 1999

Abstract

The design of an institutional building may have to satisfy a large number of requirements given by the client. Some of these constraints include the type, number, and area of the rooms that the building should contain, and the proximity relationships among those rooms. Although it is important to maintain consistency between the building specifications and layout design, it can be a tedious and difficult task for the architect to verify these constraints by hand repeatedly. We have developed an architectural CAD tool that can perform various evaluations to make certain that the user requirements are preserved during design evolution. We have also developed a new representation for describing proximity relationships between pairs of room types concisely and unambiguously. This new representation provides more flexibility and expressibility in defining the different ways that spaces should be grouped together than the traditional adjacency matrix.

Contents

1 Introduction

1.1 Motivation

The design of an institutional building can be a complicated and tedious process. Typically, such a building has to satisfy many requirements and constraints. Some of these requirements concern the type, number, and area of the rooms that the building should contain, and the proximity relationships between the different types of rooms. Room type, number, and area specifications give a general outline of how the building space should be divided and organized. For example, an educational building might need several different types of rooms, e.g. classrooms, labs, and offices. Each room type requires a certain number of instances, and a particular room type has certain requirements on area, aspect ratio, location, and accessibility to best serve the intended usages and occupancies. Proximity constraints characterize the desired distances between pairs of room types and are important where proximities are necessary for accessibility and work efficiency, or where separations are desirable to avoid noise transmission. For example, proper proximity between rooms in a hospital may enhance the ability of medical staff to move around the building quickly and save lives.

Although it is important to maintain consistency between the specifications and emerging floor plan designs, it can be a tedious and difficult task for the architect to constantly verify these constraints by hand. Computers, on the other hand, can detect these kinds of discrepancies with little computational effort. Our goal is to develop a symbolic floor plan evaluator that can perform various analyses to make certain that the user requirements are preserved in the evolving design. If some constraints are violated, the architect can try to modify the proposed design or come up with a new design as he/she attempts to converge on a design solution. If some constraints are impossible to maintain, the architect can then communicate with the client to try to reach a compromise on the specified requirements. It is our hope that this evaluation tool will help to take away the burden of specification analysis from the architect while at the same time, remain unobtrusive during the design process. We do not want this tool to stifle the architects' creativity, which is his/her unique and most valuable contribution. Thus the tool could run in the background or could get invoked periodically, when the design process has reached a phase for review.

1.2 Background

The idea of proximity relationships between spaces within a building in not new; it has existed for decades under several different terms such as adjacencies, interactions, and associations. While many works have concentrated on the notion of "direct" connection, i.e. two rooms share a common portal, there is also work that looked at a more general notion of the desirability for some rooms to be close to one and another. Many publications used the word "adjacency" to mean two spaces are right next to each other. In this report, however, we use the word "adjacency" and "proximity" in a looser sense to mean two spaces are nearby each other in term of walking distance.

Many researches have focused on using computers to generate building layouts or spatial configurations given some constraints such as adjacency requirements. MDS (multi-dimensional scaling), for example, is a technique for converting numerical information in an adjacency matrix into a spatial configuration that shows the clustering of different elements [11]. The Whitehead and Eldars program creates an "optimum" layout for a single story building from an association matrix by minimizing the cost of journeys between every pair of elements in the building [19]. Hashimshony, Shaviv, and Wachman presented a method for transforming an adjacency matrix into a planar graph from which a schematic can then be generated [8]. The model of Shaviv and Gali generates a quasi-optimal layout by minimizing an objective function based on the circulation between building elements [17].

But instead of a generative approach, which is the approach taken by all the work mentioned above, we chose an evaluative approach for our computer-aided architectural design tool. The use of the computer in helping the evaluation process has been less controversial; many people believe that the role of the computer lies more appropriately in analysis and appraisal [4, 14], rather than layout generation.

The evaluative approach has also been taken by many researchers. PHASE (Package for Hospital Architecture Simulation and Evaluation) appraises hospital designs against an association matrix and capital and energy costs requirements [12]. SPACES is a suite of three programs that aids the design of school buildings [18]. SPACES 3, in particular, provides evaluations such as association performance, environmental performance, and capital and costs estimation. A method for evaluating interactions between elements based on mutual spatial relationships in a dwelling unit was developed by Kalay and Shaviv [10]. All of these three systems aim to provide evaluations for a specific design.

We are going to present an architectural CAD tool which takes on the evaluative approach. Instead of automatically generating a floor plan based on the given specifications, which can be difficult when the building design must contain large number of rooms, the tool will verify whether or not the floor plan, created by the architect, has satisfied the specifications given by the client. One of the evaluations the tool is capable of doing is proximity verification. Proximity relationships are hard to visualize by human when given only the building layout, and therefore it is even harder to detect any violations manually. Because the main application of this tool will be for the design of large institutional building, the proximity criterion we are focusing is based on the weighted walking distance between the two entities. For large buildings where many rooms are separated by large distance, walking distance is an applicable metric. However, walking distance would not be a suitable metric for small private house where rooms are located closely together within a small perimeter. The weighted walking distance is an estimate of the distance a typical occupant would have to travel from one space to another, taking into consideration the penalty factors and path preferences.

1.3 Overview

The aim of this project is to design and implement an architectural CAD tool that will help to verify a tentative floor plan design against a set of building specifications. Before such an evaluation tool can be developed, the issue of how to express the building specifications concisely and unambiguously has to be addressed. In particular, we have to be able to describe proximity relationships between different types of room within the building. A representation with clear semantics to express the specifications is critical for capturing the original design intents and also for accurate communication between client and architect.

A suitable proximity relationships representation is introduced. Our representation divides proximity relationships into different grouping types to allow adequate descriptions of the different situations that need to be covered. All proximity types include a "strength" parameter that specifies the degree to which the stated proximity is desired. Furthermore, we define a proximity verification metric that is used in the evaluation program for checking proximity constraints on building layouts.

The proximity representation is to be captured in an ASCII file format called SYLIF [6], which stands for SYmbolic Layout Interchange Format. Although the file is human-readable, humans are not expected to type this language into the computer by hand. An entry form is implemented to assist the client in the task of formulating building specifications and producing a corresponding

SYLIF file. The entry form will remove the burden of learning the SYLIF syntax and writing an error free text file from the client.

An architectural CAD tool has been built to compare a proposed building layout against a set of building specifications. The tool is capable of evaluating the existence of the required rooms, their areas, and the proximity relationships between them. This evaluation program will help the architect and the client to detect problems on the floor plan that are not easily identifiable by casual inspector.

The remainder of this report is organized as follows. Section 2 presents the semantics and representation of proximity relationships. Section 3 describes the internal formats used for capturing building specifications and layout. The entry form for inputing building specifications is described in Section 4, and the evaluation tool for verifying building specifications is discussed in Section 5. Section 6 provides the results from running this framework on a complete building design. Section 7 discusses possible extensions and other issues, and Section 8, concludes this report.

2 Proximity Relationships

One of the design specifications that a building layout may have to satisfy is the proximity relationships between different types of rooms. Proximity relationships characterize the desired optimal distances between pairs of spaces. We define the "optimal distances" as the walking distances between the pairs. It is especially important for large institutional building where appropriate walking distances between the spaces are necessary for accessibility and work efficiency. In order to verify building layouts against proximity specifications, we need a representation with unambiguous semantics for capturing and describing the proximity relationships precisely, as well as a proper metric of evaluation.

2.1 Adjacency Matrix

The adjacency matrix is a possible simple format for storing proximity information. Other names such as interaction matrix, relationship matrix, and association matrix have also appeared in the literature. The basic principle, however, is the same: a value in each matrix entry to indicate the desire connectivity or proximity for a pair of spaces. To characterize direct adjacency, i.e. whether two rooms are connected by a portal, only *0* and *1* entries are needed in the matrix to specify whether or not two rooms are adjacent. A slight variant is to use more than one number to indicate the desirability of such adjacency, where a higher number indicates a stronger requirement [4, 8, 9, 17, 18]. A more statistical model uses the entries of the matrix to reflect the expected frequency of trips or journeys between a pair of spaces over some time period, with the data gathered from observing and monitoring an existing building of the same type that is being designed [12, 17, 19]. In particular, we are going to look more closely at a format that uses multiple values to indicate the degree of desired proximity. This format was employed in a set of collaborative courses: CS 294-5, Architectural CAD, and Arch 101, Computer-Aided Design Methods, taught by Professor Carlo Séquin and Professor Yehuda Kalay, respectively, in 1995, and again in 1996 [16].

As the name implies, the proximity relationships are captured in the form of a matrix in which every column and row represents a different room type. But instead of a full matrix, only the lower triangular matrix is used because the symmetry of the path length is assumed and no asymmetrical relationship is allowed. Each field inside the matrix is filled with a number ranging from *-3* to *+3* to indicate the degree of proximity desired by the corresponding pair of room types. The measure of proximity used in this case is the walking distance between the two entities. The diagonal entries express the desired proximity relationship among rooms of the same type (Figure 1). A positive number indicates that the rooms are to be located close together while a negative number means that proximity is to be avoided. A *0* denotes that no explicit proximity relationship exists between the pair. Furthermore, a larger absolute number indicates a stronger requirement and a smaller number indicates a weaker requirement. Thus spaces having a *+3* as the constraint between them should have a smaller walking distance between them than spaces having a *+1* constraint. Similar, spaces having a *-3* constraint should be farther away from each other as compared to spaces having a *-1* constraint.

Room Type	Fac.	Sem.	Res.	Print	Aud.
Faculty Office	2				
Seminar Room	2	0			
Research Lab	1	0	1		
Print/Copy Room	1	-2	1	-1	
Auditorium	-2	0	-2	0	0

Figure 1: An example of adjacency matrix

Although the adjacency matrix is easy to understand and simple to specify, there are a few drawbacks. When there is only one element for each room type, this simple adjacency matrix is certainly adequate. But when there are multiple elements for each room types, practical clustering constraints are more difficult to express. For instance, a constraint of *+2* between secretary offices and copier rooms may be interpreted to imply that all secretary offices and all copier offices should be located closely together. But in a more realistic situation, it is usually sufficient for each secretary office to have just one of the copier rooms nearby. In this case, each copier room can have a different number of secretary offices close to it, and some of the copier rooms may not even have any secretary office within the neighborhood. Hence, not all proximity relationships are entirely symmetrical, and there must be more sophisticated and differentiated ways of specifying a proximity requirement between room types besides being either “close” or “far”.

Another problem is the difficulty of determining which of the two room types “cares” about the proximity constraint that is being placed upon the pair in the matrix representation. For example, although it is important for secretary offices to have copier rooms nearby, copier rooms need not have secretary offices nearby. The copier rooms may have to serve other faculty or student offices. The ordering of the relationship must be clarified for such asymmetrical relationships. It is also desirable to capture the user's true intention in the representation in order to allow better future references and understanding. A possible solution to this problem would be to use the entire matrix (Figure 2) instead of just the lower triangular matrix entries, and then either the columns or the rows can be designated as the room types that “care” about the proximity relationship. So only one of the two entries would have the proximity value, while the other entry would have a *0*. For example, in Figure 2, room types on the rows are being designated as the ones that “care” about the proximity relationship. The entry (faculty office, seminar room) has a value *2* while the entry (seminar room, faculty office) has a value *0* because ”faculty office” is the entity who needs this proximity relationship. On the other hand, because of noise and traffic, the seminar room should be kept away from any printer rooms, which is expressed by the entry *-2* in the row of seminar room. But even a full matrix cannot solve all semantic problems. There are more complicate proximity relationships that need more than a simple number to specify. This will become clear in the following section.

Room Type	Fac.	Sem.	Res.	Print	Aud.
Faculty Office	2	2	1	1	-2
Seminar Room	0	0	0	-2	0
Research Lab	0	0	1	1	-2
Print/Copy Room	0	0	0	-1	0
Auditorium	0	0	0	0	0

Figure 2: Full adjacency matrix

2.2 A Richer Representation

To overcome the shortcomings of the simple adjacency matrix, a new representation with richer semantics was developed to allow more sophisticated proximity specifications. This new proximity representation is also part of the SYLIF building program specification which will be described in more detail in section 3.1.

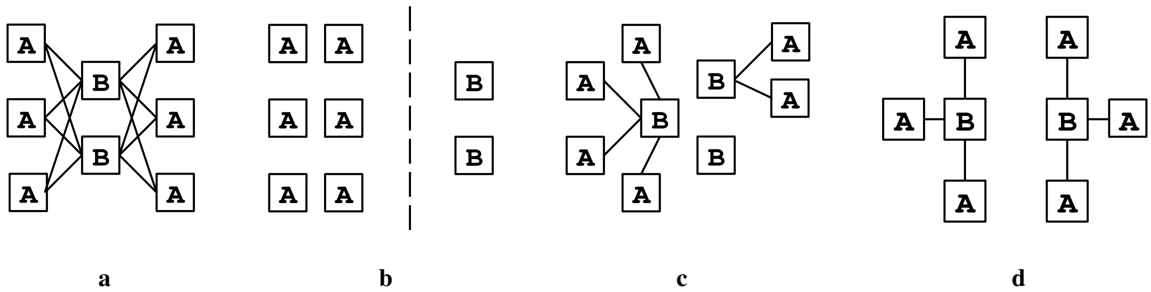


Figure 3: Proximity grouping types: a) *clustered* b) *separated* c) *at least one* d) *distributed*

2.2.1 Semantics and Syntax

The new representation divides proximity relationships into four different grouping types: *clustered, separated, at least one*, and *distributed. Clustered* and *separated* have the equivalent meaning as in the adjacency matrix; the *clustered* relationship is similar to a positive value in the matrix while the *separated* relationship is similar to a negative entry. *At least one* and *distributed*, on the other hand, are two additional relationships that give more flexibility in characterizing the relationships between spaces. The semantics of the four proximity grouping types are as follow:

- ***clustered***
  Under the *clustered* constraint, every room of type **A** requires all of the rooms of type **B** to be nearby. In order to satisfy this relationship, all distances between every room of type **A** and every room of type must be less than or equal to a certain threshold value. Basically, rooms from both types are "clustered" together in some region within the building. For example, if all the faculty offices **A** and all the visitor offices **B** want to be located near each other, then there should be a *clustered* relationship between faculty and visitor offices. (Figure 3.a)

- ***separated***
  The *separated* relationship requires all rooms of type **B** to be distant from every room of type **A**. Similar to the *clustered* relationship, all distances between the two types of rooms must be greater than or equal to the specified distance threshold. The two types of spaces would most likely occupy two different regions in the building that are some distance apart. An example would be the desired separation between the noisy classrooms **B** and the faculty offices **A**. (Figure 3.b)

- ***at least one***
  The *at least one* constraint is a variant of the *clustered* relationship except that the mandatory closeness requirement is more selectively enforced. Instead of having all rooms of type **B** be located closely to every room of type **A**, the constraint is satisfied as long as each room of type **A** has at least one of the rooms of type **B** nearby. An example would be to have a restroom **B** easily accessible from each office **A**. (Figure 3.c)

- ***distributed***
  The *distributed* relationship is an even more differentiated constraint than the *at least one* constraint. Under this constraint, rooms of type **A** require rooms of type **B** to be roughly evenly distributed among them. Not only must each room of type **A** have at least one room of type **B** nearby, but rooms of type **B** would be placed in an arrangement such that every room of type **B** would have approximately the same amount of usage. This constraint would generally be used when type **B** is a contested resource. It is to prevent a situation where some rooms would be highly congested while other rooms of the same type would be underutilized. For example, it would be practical to have storage area **B** distributed evenly among rooms **A** that need such an access. (Figure 3.d)

The new representation includes asymmetrical relationships as well as symmetrical relationships between different room types. *Clustered* and *separated* are symmetrical relationships, while *at least one* and *distributed* are asymmetrical relationships. For symmetrical relationships, the constraint is applied to the two spaces in both directions. For example, if offices want a *separated* relationship with classrooms, this implies that classrooms should also have a *separated* relationship with offices. While the user may not be inclined to express such relationships in a fully symmetrical manner, a computer assisted entry form can help to enforce fully consistent specifications. This bidirectional constraint does not apply to asymmetrical relationships. For example, if offices require a *at least one* relationship with restrooms, the constraint is satisfied as long as every office has one or more restrooms close to it, but there is no further constraint on how offices should be positioned according to the locations of restrooms.

The strength of each type of proximity specification can still be specified by a number ranging from *1* to *3*, and this strength of proximity is measured by distance in our case. But unlike the adjacency matrix, only positive values are used, because the sign of the number is now implied by the type of the relationship. For the *clustered, at least one*, and *distributed*, relationships that demand certain closeness between the spaces, a lager number indicates a stronger requirement and therefore a shorter allowable maximum distance between the spaces. A smaller number, on the other hand, indicates a weaker constraint and thus longer allowable distances. For the *separated* relationship, a larger number also indicates a stronger requirement, but it corresponds to a longer minimum distance between the spaces in this case. Similarly, a smaller number permits shorter minimum distance since it signifies a weaker requirement. The actual distances associated with values *1, 2*, and *3* can be adjusted by the user, since the distances will vary according to the actual size and style of the building.

The new representation is entered as a list of binary relationships rather than as a matrix form. To specify a proximity constraint, the pair of the spaces has to be indicated along with the desired grouping type and strength requirement, with the room type requiring the constraint being placed first. By explicitly naming the room types in order, there is no longer any ambiguity as to which of the two room types "cares" about the proximity relationship. Because each constraint has to be defined specifically, only the pairs that require a certain proximity constraint will be included in the representation, as opposed to requiring an entry in every field in the adjacency matrix regardless of whether a proximity relationship exists between the pair.

2.2.2 Consistency Issue

When multiple proximity constraints are specified, the issue of consistency arises. Conflicting proximity constraints could lead to a set of specifications that would be impossible to realize in a building layout. Even constraints that do not produce explicit contradictions can be ambiguous in meaning and may create confusion about what is really required in the building layout. For instance, a room type could have an *at least one* or a *distributed* relationship with itself, but such a constraint is not really meaningful even though it is not an impossible specification.

Most of the time, asymmetric constraints should only be specified in one direction for a pair of spaces because it is often the case in which one of the room types "cares" about the requirement while the other room type is a more passive resource which the first room type needs access to. Still, there may be situations where two parties have opposite desires and thus introduce two conflicting constraints. For example, the secretaries might want to be far away from their supervisors while the supervisors want to be close to the secretaries. With distance being the proximity criterion, this contradictory constraints cannot be satisfied. Therefore, we will not allow this kind of inconsistency in our specification. Whenever constraints are specified for both directions for a pair of spaces, we have to make sure that no contradictions have been generated.

Because walking distance is the measurement for the proximity verification, no other type of relationship is allowed in the reverse direction once a *separated* relationship is specified. This is due to the fact that all other three grouping types are "positive" relationships while the *separated* grouping type is a "negative" relationship. Furthermore *separated* relationship is a symmetrical relationship according to our definition, and therefore will be implied in both directions.

Similarly, whenever a *clustered, at least one*, or *distributed* relationship is specified, a *separated* relationship in the reverse direction will lead to a conflict. The *clustered* is also a symmetrical relationship; thus the constraint is implied in both directions. If an *at least one* or *distributed* relationship is specified for one direction and a *clustered* relationship is specified for the other, the *clustered* relationship will become the dominant constraint since the *clustered* relationship has a stronger closeness requirement.

2.3 Verification Metric

Proximity specifications are added to other layout constraints to enable people within the building to move around their environment more efficiently. Since the main use of our evaluator will be for the design of large institutional building where long distance travels may be common, it is practical to verify proximity constraints based on the walking distance for a person to travel from one space to another through some feasible paths. Travel time could also be another possible evaluating metric, but, to first order, travel time is proportional to travel distance and will scale with the walking speed of each individual.

The proximity metric we are using for verifying the proximity constraints is not solely based on Euclidean distance. Instead, weighted distances are used to represent obstacles that one might encounter along the path in order to give a more appropriate estimation of traveling distance. Some of the possible obstacles include stairs, elevators, and doors. Obviously, if a stair is encountered, the effort that is needed to get to the destination is greater than traveling an equivalent distance on the same floor. Similarly, when closed doors or elevators are along the path, some extra delay is incurred before the destination can be reached. So the weighted distance is the walking distance from one space to another plus any "extra effort" that might be required to overcome the obstacles along the path.

When stairs are involved, the traveling path seems longer than a path with equivalent distance on a level corridor not only in physical sense but also in psychological sense. The physical effort stems from the energy required to climb the stairs, and the psychological barrier results from the two spaces lying on different floors. Thus the psychological distance as well as the physical effort should be taken into account for the calculation of the weighted metric distance, since most people have the tendency to avoid a path that is psychologically farther. We generate a weighted distance for stairs with a multiplication factor on the overall vertical distance traveled, ignoring the physical property of the staircase such as whether or not they are spiral or diagonal. It is likely that the actual property of the staircases may not yet be known during the early verification phases, for proximity checking should be done as early as possible, before more detailed floor plans are drawn. Although climbing up stairs is more tiresome than going down, only one multiplication factor is currently used for proximity checking. The meaning of proximity can become ambiguous if a pair of rooms on different floors have two different weighted distances depending on which one of the two is the starting point.

As in the case of stairs, using an elevator involves the psychological sense of floor separation as well as some average waiting time. Although traveling by elevator takes less physical effort than traveling by stairs, waiting for an elevator to arrive to the desired floor could take a considerable amount of time. In our metric we add a fixed time delay cost to the weighted distance for having to wait for the elevator. Since we are using traveling distance for verifying proximity constraints, this time delay cost will be equivalent to a certain distance. A multiplication factor on the overall vertical distance is also included. This multiplication factor will be smaller than the factor for stair penalty because less effort is required for traveling by elevator. For an even more accurate measurement, this delay cost might be adjusted to take into account the number of elevators in a given location, since the time spent waiting for an elevator is generally shorter if more elevators are available. A travel time proportional to the number of floors traveled takes into consideration the possibility of the elevator stopping at other floors before reaching the destination level.

Delays also result from having one or more doors along the traveled path. Opening and closing doors can delay people from getting to their destination, due to the physical effort and the time delay involved in opening doors; unlocking locked doors can further increase this delay. The delay cost can thus vary depending on the type of door. For example, a closed fire door is generally heavier than a regular office door and would require more effort to open it. Similarly, a locked door would take more time to open than an unlocked door. If some locked doors are not accessible to the general public, the path through these doors must be eliminated from the search tree.

The following table gives a summary of all the penalty factors discussed above.

obstacle	**weight**
stair	multiplication factor on vertical distance
elevator	one time delay cost added to overall distance & multiplication factor on vertical distance
closed door	one time delay cost added to overall distance
open door	no delay cost

Table 1: Penalty factors

3 Internal Representations

In order for our specification checking program to perform all the necessary verifications on a proposed floor plan, building specifications and layout must be captured in a format that provides semantic clarity and can easily be processed by computers. This section describes the building specifications representation and the floor plan representation that are currently under development in our research group.

3.1 SYLIF Building Program

SYLIF, SYmbolic Layout Interchange Format, is a simple Lisp-like ASCII format for representing both the general requirements of a large institutional building and symbolic layouts for possible floor plans that might satisfy these requirements [6]. The language and its parser are currently under development by Laura Downs under the supervision of professor Carlo Séquin.

```
(program
    (roomlist
        (class FacultyOffice
            (info
                (defvalue numberofrooms 40)
                (defvalue area 180)
            )
        )
        (class SeminarRoom
            (info
                (defvalue numberofrooms 4)
                (defvalue area 200)
            )
        )
        (class ResearchLab
            (info
                (defvalue numberofrooms 6)
                (defvalue area 600)
            )
        )
        (class PrintCopyRoom
            (info
                (defvalue numberofrooms 4)
                (defvalue area 150)
            )
        )
        (class Auditorium
            (info
                (defvalue numberofrooms 1)
                (defvalue area 1500)
            )
        )
    )
    (proxlist
        (prox FacultyOffice FacultyOffice (clustered 2))
        (prox FacultyOffice SeminarRoom (dist 2))
        (prox FacultyOffice ResearchLab (dist 1))
        (prox FacultyOffice PrintCopyRoom (dist 1))
        (prox FacultyOffice Auditorium (sep 2))
        (prox SeminarRoom PrintCopyRoom (sep 2))
        (prox ResearchLab ResearchLab (clustered 1))
        (prox ResearchLab PrintCopyRoom (dist 1))
        (prox ResearchLab Auditorium (sep 2))
        (prox PrintCopyRoom PrintCopyRoom (sep 1))
    )
)
```

Figure 4: A sample fragment from a SYLIF building program

In order to automatically verify with the help of a CAD tool that a given building layout satisfies the user's requirements, we must be able to represent those requirements in an unambiguous computer-readable format. The building program, one of the three sections of the SYLIF file format, provides the syntax and semantics for representing such information. It allows to specify the number of each type of rooms that the building should contain and the desirable areas for each room type. It also contains the syntax for specifying the proximity relationships between room types as described in the previous section. Figure 4 shows an example of a SYLIF building program which corresponds to the example of the adjacency matrix in Figure 1. In addition, a complete SYLIF building program would also contain the number of rooms and area specifications.

In the SYLIF building program, room types are described in a class hierarchy in which the children inherit basic requirements from their parents. This hierarchical representation allows the user to define one set of specifications for a group of similar room types while giving the user the flexibility to override specifications inherited from the parent class by defining more refined specifications in the child class when necessary. The parent class can be seen as a generic type, while the child class can represent a more specific room type. Consider the following statements:

```
(roomlist
    (class Office
        (info
            (defvalue area 180)
            (defvalue numberofrooms 50)
        )
    )
    (class FacultyOffice Office)
        (info
            (defvalue numberofrooms 30)
        )
    (class SecretaryOffice Office
        (info
            (defvalue area 120)
        )
    )
    (class Classroom)
)
(proxlist
    (prox Office Classroom (sep 1))
    (prox FacultyOffice Classroom (sep 3))
)
```

In the above example, FacultyOffice and SecretaryOffice are children classes of Office, and therefore they inherit all properties of Office. But due to the explicit definition of the area requirement 120 in SecretaryOffice, the default area of rooms of type SecretaryOffice is overridden while rooms of type FacultyOffice assume the inherited area value of 180. Similarly, rooms of type FacultyOffice have an overriding *separated 3* relationship with rooms of type Classroom while rooms of type SecretaryOffice have an inherited *separated 1* relationship with rooms of type Classroom. The *number of rooms* field, on the other hand, is one exception to the inheritance rule. The children do not inherit the value from the parent class. Rather, the total number of rooms of the children and the parent combined should be at least equal to the parent's specification for the *number of instances* field. The instances number in the parent allows the user to control the room number at a higher level, and the generic rooms can be put to different use later on. As in our example, the number 50 in the instances field for Office indicates there should be at least 50 offices available in the entire building. Out of these 50 or more offices, at least 30 of them must be of type FacultyOffice. Since there is no instances specification for SecretaryOffice, there can be any number of rooms for SecretaryOffice. If there are 30 FacultyOffices and 10 SecretaryOffices, then there must be at least 10 more Offices. These 10 Offices can be seen as generic offices and may be assigned as either FacultyOffice or SecretaryOffice in the future, depending on the growth of faculty and secretary members. Having the *number of instances* to be a lower bound puts less pressure on users and architects to come up with the exact number, and the ultimate upper bound on the total number of rooms will be constrained by building size and room area.

3.2 Symbolic Floor Plan Layout

Unlike an actual floor plan, symbolic floor plans don't contain all the building layout information such as wall thickness and portal dimensions. Rather, a symbolic floor plan is a simpler representation that captures the basic space layout with single-line walls and dimensionless portals (Figure 5). The symbolic floor plan stores general room geometry and portal locations, but maintains very little detail information. This representation corresponds to the early design phase and is intended for quick manipulation of the rooms that will be placed within the building.

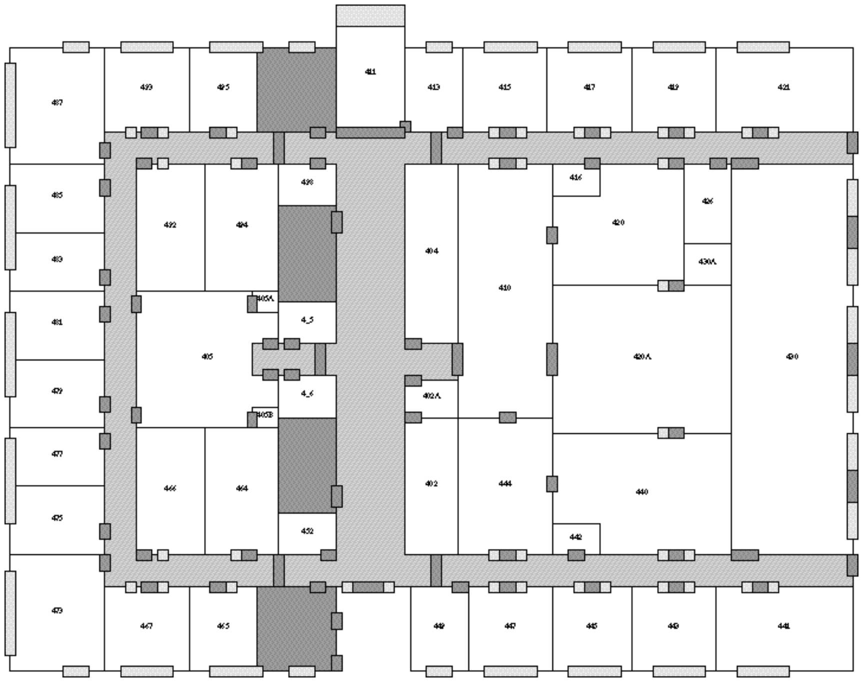


Figure 5: Symbolic floor plan of Soda Hall 4th floor

Although a symbolic floor plan lacks some of the details that an actual floor plan has, it reflects proximity and geometric information quite accurately and is sufficient for comparing the emerging design against specified building requirements. Because the symbolic floor plan is not clogged with other unnecessary information, it is easier to process for verification purposes. The proximity verification process should confirm that no significant violations of the proximity relations are present in the proposed building layout. It should take place as early as possible, before a large amount of time has been spent on making detailed floor plans. The symbolic floor plan plays its most important role in the early design phase where it provides an efficient, computer-readable way to represent initial sketches of various floor plans which can be checked against the building specifications. Once a symbolic floor plan shows a satisfactory organization of the building, more time can be spent on refining the symbolic layout into the more detailed traditional floor plans.

Currently, the Berkeley UniGrafix (UG) file format [3] is used as the geometrical representation of the symbolic floor plan. Spaces within the building layout are described as contiguous sets of contours that are defined with shared vertices. The dimensions of the simplified contours are close matches to the the actual building structures. All key elements such as rooms, corridors, stairs, and portals are labeled with symbolic identifiers. Symbolic floor plans can be obtained by converting the AutoCAD DXF file format into UG file format. A tool called Building Model Generator (BMG) developed by Rick Lewis [13] is capable of performing such a task. However, most CAD file formats, including DXF, do not contain room usage information, i.e., which rooms are of which types. In addition, UG file format is not capable of supporting such information. Thus a separate roster file is used to specify the room type of each room.

SYLIF, currently under development, also has a specific symbolic building layout representation. Our goal is to use the SYLIF symbolic layout as soon as it becomes available. In addition, a building layout editor is being developed to provide the functionality of creating symbolic floor plans and saving them in the SYLIF symbolic layout format. The SYLIF symbolic layout format will contain the relevant room usage information, and therefore eliminate the need of a separate roster file.

4 Building Specification Entry Form

Even though the SYLIF building program is in a human-readable ASCII format, it still would be a tedious and error prone process for the user to write the corresponding text file that contains all the required specifications for the building to be created. All syntactical details should be hidden from the general users who should only need to know the semantics of the different specifications. Therefore, we have added an user interface that can handle the building specifications input for the user without forcing him/her to learn any unnecessary details. This section describes such a specification entry form.

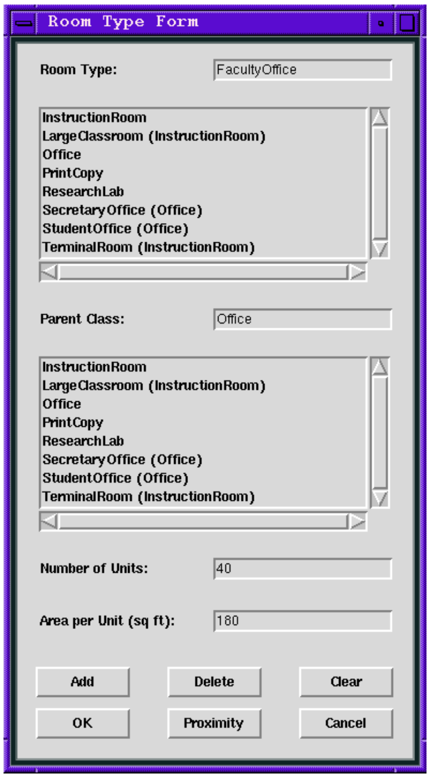


Figure 6: Room definition form

4.1 Functionalities

The building specification entry forms provide a simple interface to input building requirements such as the type, number, and area of the desired rooms, and the proximity relationships between certain types of rooms. There are two basic forms: one for defining room types and their necessary requirements and another one for specifying proximity relationships.

The form for defining room types allows the user to specify a new room type and to make modifications to any existing room types (Figure 6). The form contains entry boxes for specifying the parent class, the number and desired area of the new room class, besides the entry box for defining or choosing the room class itself. Selection boxes showing a listing of all existing room types in alphabetical order make it easier to pick from already existing classes. To give the user some idea of the hierarchical relationships between the room classes, ancestor classes are shown in parentheses if defined. When the parent field is filled out, any specifications inherited from the parent are immediately shown in red, as oppose to black, to differentiate themselves from the explicitly defined specifications. If no new values are specified for any of the fields, the room type would inherit its requirements for all unspecified fields from the parent class. Erasing the inherited values from the entry boxes does not prevent the room type from inheriting the requirements, unless new overriding requirements are specified for the room type.

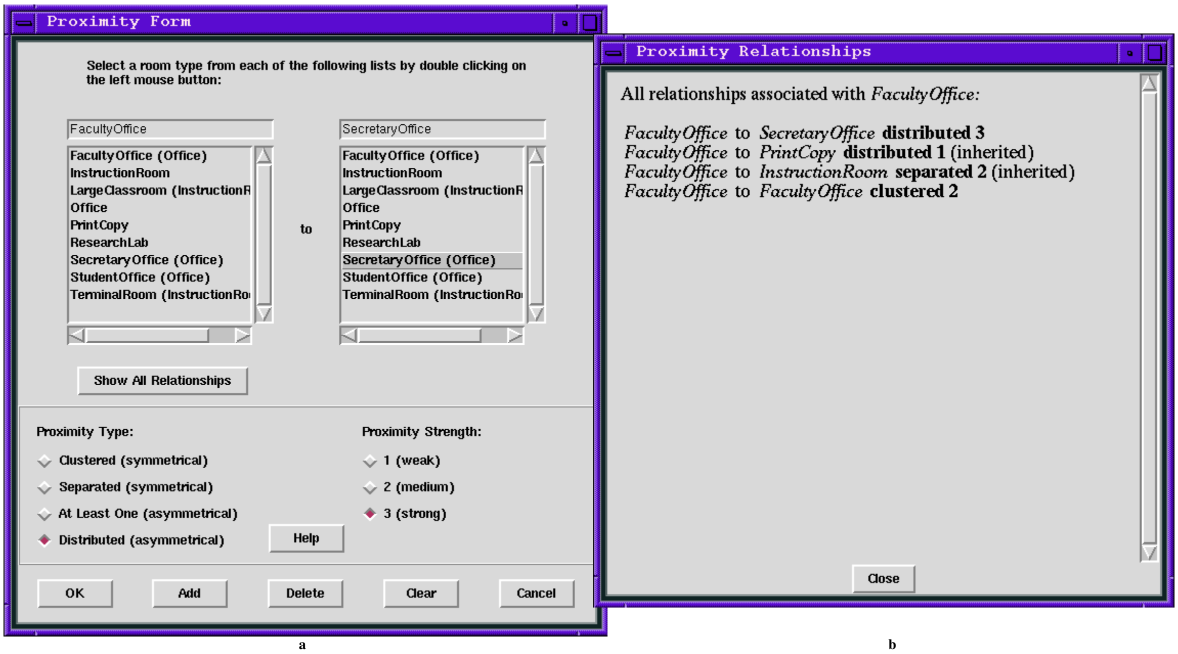


Figure 7: a) Proximity relationship form b) Window showing all existing relationships for the selected room type

The proximity form lets the user define and modify proximity relationships between any two room types (Figure 7.a). As in the room definition form, there are also selection boxes that list all the existing room types and their corresponding parents. Inherited proximity relationships, if they exist, appear after a pair of room types has been entered into the entry boxes. Proximity grouping types and strengths are listed as radio buttons for the user to click on the desired proximity relationship for the pair of rooms. Understanding that most users will not be familiar with the semantics of the various proximity relationships, a simple "Help" window can be opened, which describes the different grouping types and strengths that can be specified. Another window showing all the existing proximity relationships for the selected primary room type can also be opened to give the user an overview of the existing relationships with other room types, and thus facilitating the definition of new meaningful relationships (Figure 7.b). This "Show All Relationships" window is updated interactively whenever a change is made in the proximity form. The program also checks for inconsistencies between proximity constraints, as mentioned in section 2.2.2, and prohibits the user from specifying conflicting constraints.

Previously defined building specifications can be read into memory through SYLIF building programs, and the information entered through the entry form can be output as a new SYLIF file. This functionality allows the user to modify an existing SYLIF file and save a, possibly only partially finished, building program. The purpose of the entry form is to help the user input building requirements information with minimum effort and with no need to know the syntax of the SYLIF language.

4.2 Implementation

The building specification entry form is written in Tcl/Tk. Tcl/Tk has been chosen for its portability to different platforms and the simplicity of its commands for creating user interface widgets.

Several assumptions and implementation decisions were made to keep the entry form conceptually simple and to ease the implementation task. One of the decision was to disallow forward referencing when defining a new type of room. That is, a room type has to be defined before it can be referenced as a parent class for a new room type. This decision not only simplifies the implementation, but it also simplifies error checking for references to undefined parents. Allowing the user to reference an undeclared room class is likely to cause confusion and error, since the user may easily forget to give a definition for the undeclared reference later on.

All children classes inherit room requirements from their parents or ancestors, unless explicit values are specified at the children level to override the inherited values. When any fields are left empty, the child would inherit the associated values if one of the ancestors has default values specified for those fields. When a value being entered into the entry is the same as the parent's, the program would ask the user whether or not to decouple the child's field from its parent. If a child's field is decoupled from its parent, any future changes made to parent's field no longer affect that particular field of the child. This extra level of questioning to confirm user's true intention has been added so that there won't be any ambiguity. The children don't actually store the inherited values with their own records. Instead, each child stores the associated parent index, and the inherited values are retrieved by indexing into the parent's record. Because there may be many levels of ancestors, this indexing process is performed until a value is retrieved, or until the top level is reached.

Retrieving an inherited proximity relationship is a little more complicated. In order to find the inherited proximity relationship, the hierarchies above both sides of the pair have to be checked. First the source room type would be checked bottom up against each ancestor of the destination room type. Then each ancestor of the source type would again be checked against the destination type and its ancestors. This process is repeated until an inherited proximity relationship is found or the very top level is reached on both sides. Because of the symmetrical relationships such as *clustered* and *separated*, the checking also has to be done on the inverse pair to determine whether or not a proximity relationship has been defined for that pair of room types.

Each room class also keeps a counter of how many descendants it has. Whenever the user tries to delete a room type, the program first checks to see if there are any children associated with this room type and alerts the user if the number of descendants is greater than zero. If the user still wants to delete the entry, all the descendants of the room type to be deleted will be linked to the grandparent, i.e., the parent of the deleted parent. If the deleted room class has no parent, all the descendants will now sit at the top level with no more inherited specifications other than the ones defined explicitly in the current class. Since only the parent index, not the individual inherited values, is stored within each child, only the parent index of each descendant of the deleted room type needs to be updated to reflect the change.

When a SYLIF building program is read in and processed by the program, the assumption is made that the SYLIF file loaded by the user has been generated previously by this specification entry program. This should ensure that the files are consistent with all the assumptions and decisions discussed above, i.e., that the definition of the parent class appears before all definitions of other classes that reference this parent. Also it should guarantee that the files are in an expected format. We did not want to spend much time on writing the file parsing code when a more complete and "official" SYLIF parser is nearing completion. The SYLIF parser, when it is ready, will be incorporated into the program so that more general SYLIF files can be loaded as well.

5 Building Specification Evaluator

Given a set of building requirements, a program is needed to help the architect and the client to automatically compare a tentative building layout against these specifications. The following section describes the appraisal program that performs the verification of the symbolic floor plans against the formal room specifications and proximity relationships.

5.1 Functionalities

5.1.1 File Loading

In order to perform any verification, the user must first provide a floor plan and a file with all the building requirements. Currently, the floor plan is in the Berkeley UniGrafix (UG) file format and the building requirements should be in the SYLIF file format, as discussed in section 3. The floor plan and the SYLIF file are being loaded independently and can be input in any order. The program also allows the user to reload different floor plans and/or SYLIF building program files multiple times without quitting the program. This is helpful when there are more than one building layouts for the same set of building specifications, or when there are several versions of requirements for a single building. With multiple designs for the same specification, the user can load the SYLIF file just once and then load a different floor plan for each verification while leaving the specifications unchanged. This also provides a convenient way for the user to compare a modified building layout against the previous version of the floor plan. Similar to the example of modified floor plans, it often can be the case that after seeing the results of the initial verification process, the user realizes that some constraints are impossible to satisfy and thus the building specifications have to be modified. In this case, the SYLIF file can be reloaded without the need to also reload the floor plan.

5.1.2 Floor Plan Display

The program provides two views of the floor plan on two separate display windows: a 3D view and a 2D view. The 3D window lets the user look at the building as a whole, with the floor plans for all floors stacked above each other with some appropriate height offset. The 3D window also provides a crystal ball interface to allow the user to rotate the building and to look at it from different angles. But with floors stacked on top of one and another, it is hard to view one complete floor in detail other than the one that is at the top of the building. Thus a 2D window is also included to give a flat view of the floor plan of a chosen single floor. The crystal ball interface is not added to the 2D display so that the floor plan remains flat on the screen at all times. Instead, the user can cycle through all the floors of the building, with floor levels being displayed one at a time. Both 2D and 3D windows provide a zoom function to allow the user to look at the floor plan in detail. With these two display windows, the user can view the building as a whole as well as look at one of the floors in more detail.

Floor plans can be displayed in either outline mode or color mode. The outline mode provides the look of a traditional floor plan where each space is being drawn as a contour. The color mode differentiates among the various types of space (e.g. corridor, room, stair, elevator, etc.) by coloring the polygons in different colors. It helps the user to identify the major space types on each floor.

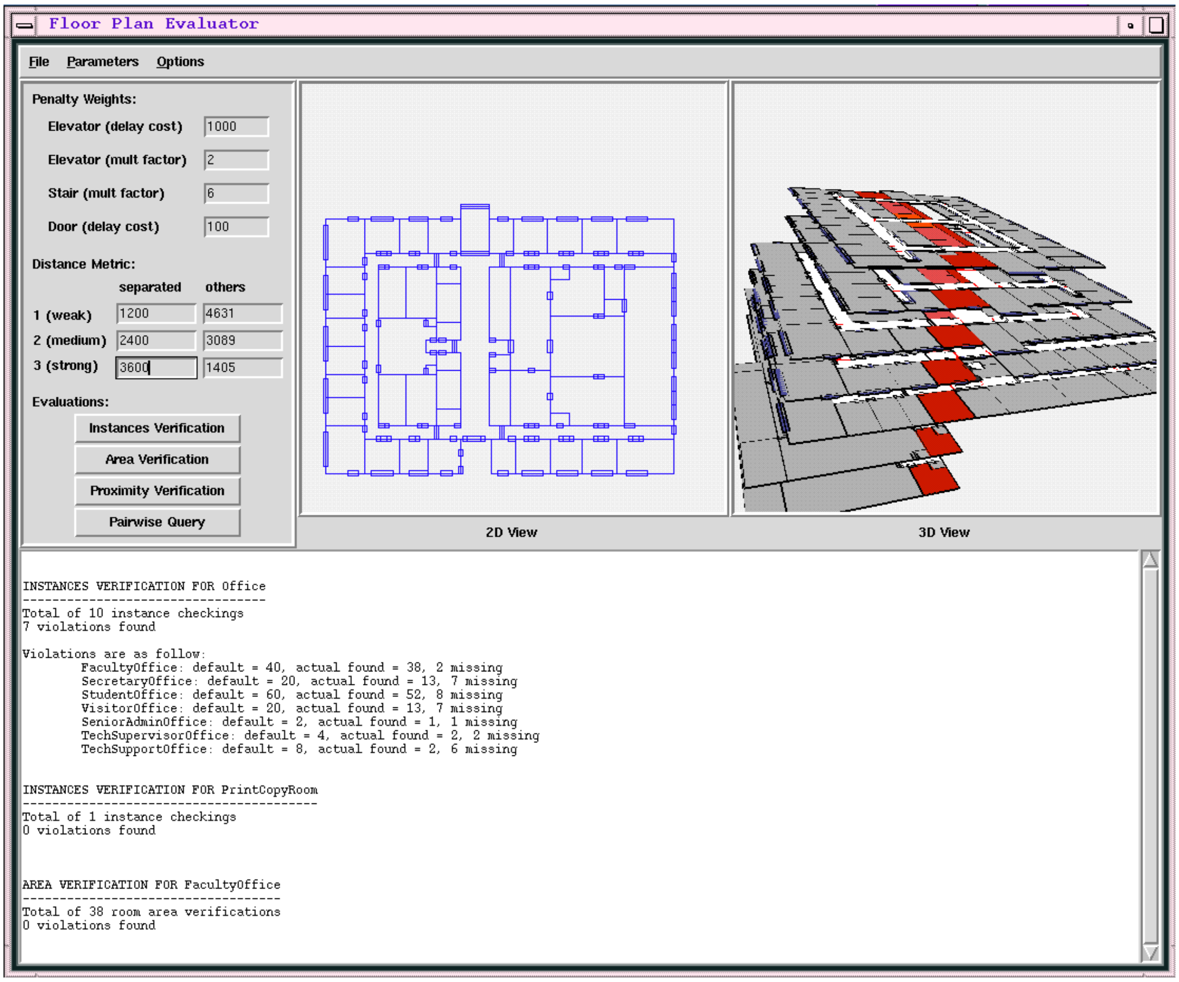


Figure 8: Building specification evaluator

5.1.3 Parameter Settings

Several different parameters can be set during the building verification process to allow different interpretations of what an easy to travel path is and to give user some flexibility in carrying out the analysis. The program allows the user to set the following parameters: *path option, penalty option, penalty factors*, and *distance metric*.

Path option lets the user select whether or not a certain path should be allowed during the evaluation. For example, the user might decide to exclude paths that use an elevator, or paths that travel through some other rooms before reaching the final destination. The case of excluding the use of an elevator can be particularly useful for the evaluation of egress patterns under emergency situations such as fire, earthquake, or power outage. The three different options that are currently in the program are the use of elevators, stairs, and pass-through rooms. The default is to include all path options.

Penalty option allows the user to choose what penalties should be taking into consideration in the distance calculation. Penalty weights can currently be assigned to elevators, stairs, and doors. Some penalties might not be necessary under certain situations and thus should be excluded from the calculation. This option can also be used for comparing results obtained with various sets of penalties to see how much of a difference each option makes. As default, all the penalties are turned on.

The user is also allowed to change the weights of the *penalty factors* that are used in the distance calculation. The stair penalty has a multiplication factor that is applied to the traveled height. The elevator and door penalties have delay costs that are added to the traveling distance whenever an elevator or a door is encountered. The elevator penalty also have a multiplication factor on the vertical traveled height in addition to the one time delay. Different users might have a different notion as to which obstacle is easier to overcome and thus how much of a weighting factor should be associated with each of the penalties. In addition, weighting factors might have to vary from one type of building to another due to different concerns and situations. Nevertheless, we still tried to provide reasonable penalty factor settings at the startup of the program. The stair factor is initialized to 6. The elevator penalty is given an 1000 inches (25.4 m) delay cost and a multiplication factor of 2. The door delay cost is initialized to 100 inches (2.54 m). Although it is possible to use the weighting factors to control which penalties are to be taken into account in the distance calculation (e.g. setting stair multiplication factor to 1 disables the stair penalty option and setting door delay to 0 disables door penalty option), separate on-off switches for each penalty are still provided for convenience.

In addition, the *distance metric* can be adjusted. User can tune the distance thresholds associated with each of the proximity strength values *1, 2*, and *3* according to the size and style of the building, to personal preferences, and/or to the results of previous evaluations. There are two distance metric settings in the interface for each of the proximity strength: one for the *separated* relationship and another for the *clustered, at least one*, and *distributed* relationships. Two separate metric tunings are included to provide the flexibility of allowing a different set of threshold values for the *separated* constraint, which is something the user might want.

5.1.4 Specification Verifications

Given a floor plan and a SYLIF file, the program can perform several different evaluations to see whether the floor plan conforms to the building specifications. The types of evaluation include *instances checking, area verification*, and *proximity checking*.

Instances checking confirms that the correct number of room instances in the building for each room type is in accordance with the building program. The user can choose to process the entire specification list, or just one specific room type. A list of all the defined room types is given to let the user choose the one particular room type that needs to be evaluated.

Area verification determines whether or not the room areas for each type agree with the default areas specified in the SYLIF file. As in *instance checking*, the verification can be performed either on all the room types that have a default area value, or on one particular room type.

Proximity checking verifies whether or not the floor plan satisfies the proximity relationship constraints specified in the SYLIF file. The evaluation can be performed on the complete list of proximity definitions for all the room types in the SYLIF file, or it can be done on a pair of room types chosen by the user. If one of the room types is part of a more complex relationship involving several types, the analysis will be done for all the participating rooms; see the *distributed* relationship description under section 5.2.4. All defined room types are listed in the interface so the user can easily click on the two desired room types to perform the check. The user can also query the shortest distance between two room instances by selecting two contours on either the 2D or the 3D display window; the shortest path from door to door will then be drawn on top of the floor plan in a red dashed line.

Results of verifications are displayed in a text window at the bottom of the display. If any violations are found, the rooms that fail the verification process are listed in this message. The violated spaces are highlighted on the floor plan to give the user an idea of where in the building the problems occur.

5.2 Implementation

The building specification evaluation program is written in C++ and Togl (a Tk widget for OpenGL rendering). Togl provides an effortless way for implementing the user interface for the program.

5.2.1 Room Instances Checking

Room instances checking confirms that the building layout has a sufficient number of rooms of a particular type as required by the building specification. For room types without any children, i.e. room types at the bottom level of the hierarchy, the checking process is simply a comparison between the actual number of rooms found on the floor plan against the given default value. For room types with one or more child classes, the verification process becomes a little more complicated, since the default room number includes the room count of the parent plus the room counts of all its descendants. To verify the instance requirements for a parent type, we sum up the actual room counts for each descendant recursively and add this sum to the number of rooms found for the parent, to see if the total equals the parent's default value. While counting the number of rooms for each descendant, we also verify whether or not the room count corresponds to the default value specified for the child. Any violation found at the children level indicates that the ultimate constraint is not satisfied, even though the final room count might correspond to the parent's room number requirement.

5.2.2 Room Area Calculation

To verify whether the layout is compatible with the area specifications in the SYLIF file, we must calculate the area of each room and compare the results to the default values. If a child does not have a default area, it inherits its default value from the closest ancestor who has a non-empty area field. If there is no default area for either the child or its ancestors, no room area verification will be done for that room type. In this implementation, we use the formula in Green's theorem [1] which yields the area of simple (non-self-intersecting), planar polygons. For a polygon given by the vertices $(x_i, y_i), i = 0, ..., n$, with $x_0 = x_n$ and $y_0 = y_n$, the formula for calculating the area is as follow:

$$A = \frac{1}{2}\sum_{i=0}^{n-1} a_i \quad where\ a_i = x_i y_{i+1} - x_{i+1} y_i$$

Since rooms are described as contours defined by a sequence of vertices in the UG file, the area of the rooms can be calculated easily by feeding the vertices into the equation.

5.2.3 Path and Distance Calculation

In order to perform proximity constraints evaluation, we have to be able to find some shortest, feasible path on the building layout between two spaces. The resulting path depends not only on the structure of the building, but also on the parameter settings of path option, penalty option, and penalty factors as specified by the user, since all these settings will contribute to the computed traveling distances. Because adjacencies existing in a floor plan can be described best in the form of a graph, we first construct an adjacency graph that maintains all connectivity information from the input building layout. Then Dijkstra's algorithm [2] is run on the adjacency graph to find the shortest path and the weighted distance between any two spaces of interest.

After the floor plan is loaded, the program constructs an adjacency graph from the geometric data given in the UG file. Before an adjacency graph can be built, the corridor contours, which are specified as one large, possibly concave, polygon, have to be broken up into convex polygons. At every generated joint between the new pieces, an "open" portal is inserted to serve as the connection between these pieces. After the corridors have been "rectified", straight paths can now be generated from piece to piece without concerns of intersection with other spaces, as illustrated in figure 9. Convex rooms are also processed in this manner.

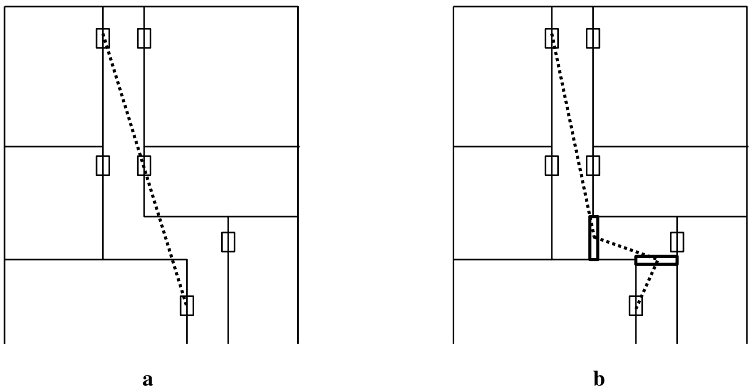


Figure 9: a) Path intersects with room interiors b) Desired path after splitting up the corridor into convex pieces

The structure of the adjacency graph is such that the nodes are portals and the links connecting the two nodes represent straight path segments through the space that joins the two portals on the floor plan. Portals are doors present in the floor plan as well as the "open" portals generated during the rectification process. Spaces connecting the portals can be corridors, rooms, elevators, or stairs. Portals on vertical connector spaces, such as elevators and stairs, are linked to all other portals that are connected to the same elevator or stair, but on different floors. Figure 10 illustrates a simple example of a 3-story floor plan and its corresponding adjacency graph. The weights of the links correspond to either the Euclidean distances between portals on the same level or the vertical heights of portals on different floors. Penalty factors are not being taken into account at this point because the adjacency graph is built only once for each floor plan while penalty settings can change many times for the same floor plan during the evaluation process. Instead, penalty factors will be considered later during the search for "shortest" paths.

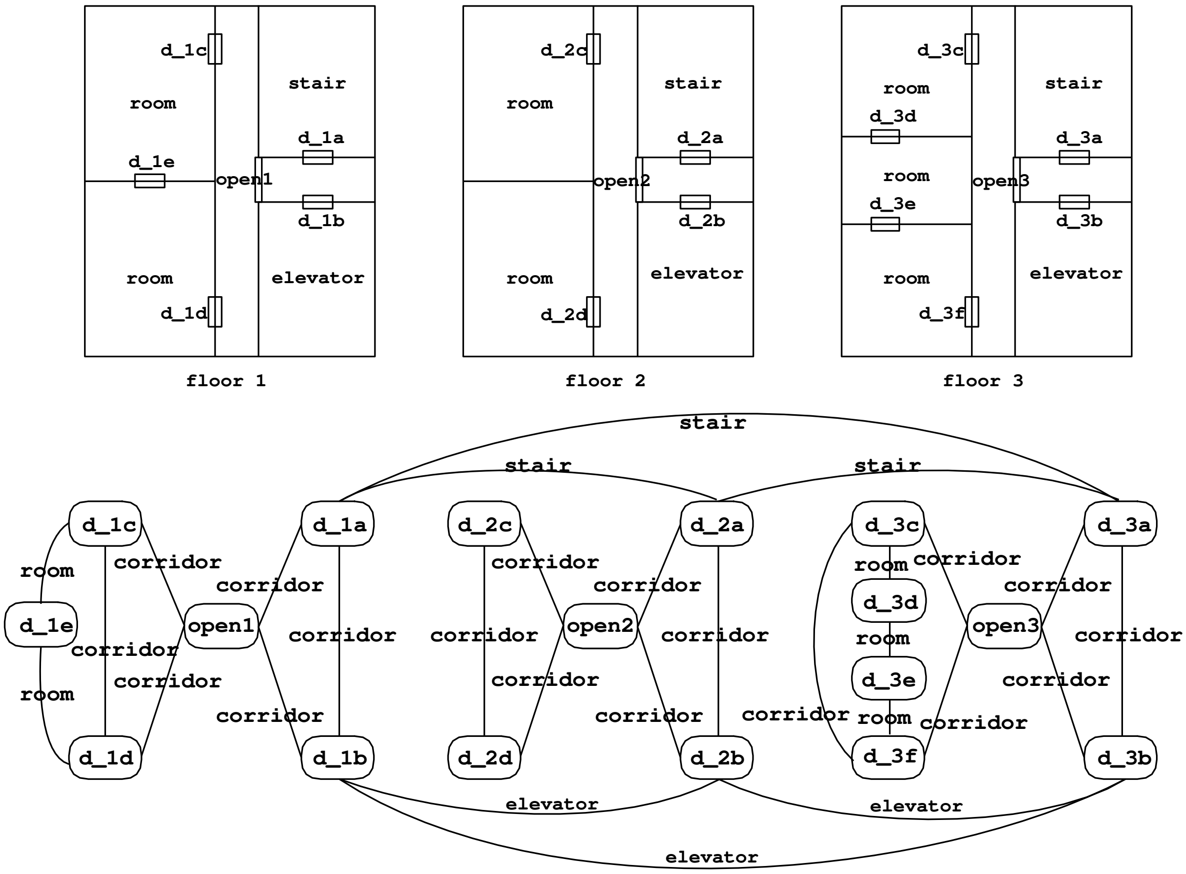


Figure 10: An example of an adjacency graph

Under the current implementation, the adjacency graph is bidirectional; i.e., if there exists some path from point A to point B, the same path can be traveled from B to A. Although this assumption might be true most of time, there could be cases where the direction of travel makes a difference. For example, it is possible that some locked doors can be opened from one side but not from the other. The adjacency graph could easily be changed into a directed graph that reflects such special cases, if such information is given with the floor plan.

After the adjacency graph is built, Dijkstra's algorithm [2] is used to determine shortest paths between spaces. The shortest path found will be from door to door because of the way the adjacency graph is built. Dijkstra's algorithm solves the single-source shortest paths problem on a weighted graph that contains only edges with nonnegative weights, which are indeed the properties of our adjacency graph. For a given source, the algorithm computes a shortest-path tree in which the source is the root and all nodes in the tree are the portals reachable from the source by some shortest path. For every node, a corresponding distance from the source to that node is also maintained. Path and distance calculations under Dijkstra's algorithm take into account the penalty factors and path options. Thus, no disallowed path elements are included in any of the resulting paths, and appropriate penalty factors are weighted into the calculated distance for each path, as elevators, stairs, and doors are encountered.

5.2.4 Proximity Relationship Verification

Proximity verification is performed by finding shortest weighted paths and distances for room pairs belonging to the two space types that are being evaluated and comparing the results to the constraint specified in the SYLIF file. Since the adjacency graph used in Dijkstra's algorithm is built with portals being the nodes, all doors that are attached to a particular room have to be determined first. For every room, a list of all attached doors is maintained during the adjacency graph construction step. Whenever a portal finds a link in the adjacency graph that is of type room, the room adds that portal to its list of doors. To find the shortest path and its distance between a pair of rooms, all paths between two sets of doors have to be determined first, and then the path with minimum weighted distance is returned. Because most rooms have only one attached door, this process can be done in a short amount of time. Once the distances between room pairs have been determined, we can verify the proximity constraint for every pair of room types. The four different proximity grouping types, *clustered, separated, at least one,* and *distributed*, are processed somewhat differently.

Clustered Relationship

The *clustered* relationship requires all rooms of type B to be close to every room of type A. In order to verify whether or not this requirement is satisfied, shortest paths from every room of type A to all rooms of type B need to be determined. All distances of the paths found must be less than or equal to the corresponding threshold set in the distance metric parameter for the specified proximity strength. Comparison to the threshold value is done for every distance to insure every pair has satisfied the relationship. Every time a distance exceeds the threshold limit, the *clustered* constraint is violated. The violation and the excess amount is written to the message display window. Alternatively, the verification process for that particular pair of room types can be terminated, to return other results to the user more quickly.

Separated Relationship

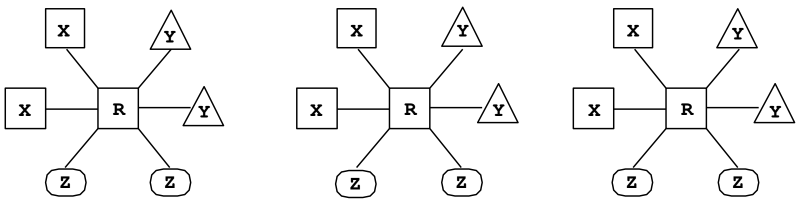
Under the *separated* constraint, every room of type A requires all rooms of type B to be distant from it. As in the case of the *clustered* relationship, shortest paths between all rooms of type A and all rooms of type B have to be determined. But instead of requiring all distances to be less than or equal to some threshold, the distances now have to be greater than or equal to some value. Again, violations encountered will be written to the message window along with the difference between the actual distance and the threshold value. The checking process for this room type pair can also be terminated after the first violation has been found.

At Least One Relationship

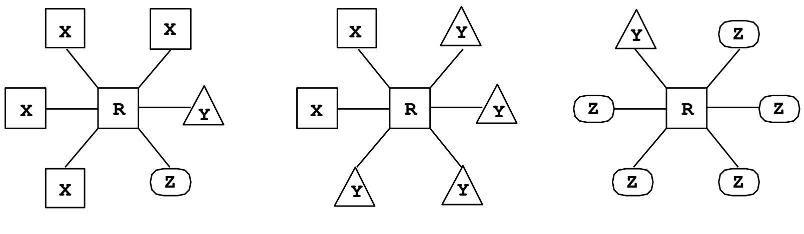
The *at least one* relationship is a variant of the *clustered* relationship. This requirement is satisfied as long as each room of type A has at least one room of type B nearby. The verification process of the *at least one* relationship is very similar to and even simpler than the *clustered* verification. Instead of finding all shortest paths from a room of type A to all rooms of type B, the calculation for a room of type A can stop once a room of type B is found to satisfy the threshold value. The number of paths that need to be determined for the *at least one* evaluation is less than in the case of *clustered* and *separated* verifications.

Distributed Relationship

Verification of the *distributed* relationship is more complicate than for the other three proximity relationships discussed above. Even the interpretation of the meaning of the *distributed* relationship can be somewhat ambiguous. Intuitively, the *distributed* constraint means rooms of type A require rooms of type B to be roughly evenly distributed among them. But what should the distribution look like when there are two or more room types that require rooms of type B to be evenly distributed among them? There are two possible interpretations. First, let's call rooms of type B the "resource" rooms and rooms of type A the "consumer" rooms. It is common for one resource room type to serve more than one consumer room type. One interpretation is to have the resource rooms evenly distributed among the rooms of each of the consumer space types, as in figure 11.a. Another interpretation is to have the resource rooms evenly distributed among all rooms of all the consumer space types that require a *distributed* relationship with the resource rooms, as in figure 11.b. Figure 11.b shows a looser interpretation than the one shown in figure 11.a, i.e., if figure 11.b is chosen as the interpretation, both configurations in figure 11 satisfy the *distributed* relationship. On the other hand, if figure 11.a is chosen as the interpretation, the configuration in figure 11 would not be acceptable. It is more likely that the different consumer room types that need *distributed* constraint with the resource rooms are in their own cluster. For example, both faculty offices and student offices have a *distributed* relationship with printer rooms, but faculty offices are likely to be in one cluster while student offices are in another cluster. Therefore, it is probably more logical to say the constraint is satisfied as long as the printer rooms are evenly distributed among all offices, not that the printer rooms have to be evenly distributed among faculty offices AND among student offices. Hence, we will be using the second interpretation, as illustrated by figure 11.b, for our *distributed* constraint evaluation. The actual number of rooms of each consumer type that a resource room would serve depends greatly on the location of all rooms having a *distributed* relationship with the resource rooms.



a



b

Figure 11: Two interpretations of the *distributed* relationship

From the interpretation we have chosen above, the *distributed* relationship verification cannot be performed on a pair of room types as in the other three proximity relationship cases. Rather, the evaluation has to be performed on a resource type against all other room types who have a *distributed* relationship with that particular resource type together. If there is only one resource room, the *distributed* relationship degenerates into *clustered* relationships between the resource room and each of the consumer rooms. If there are r resource rooms and n consumer rooms, another way of looking at "distributed evenly" is that each resource room would have to serve approximately $\lceil \frac{n}{r} \rceil$ rooms. We can then address the distribution problem by trying to assign each consumer room to one of the resource rooms in a way such that each resource room would serve no more than $\lceil \frac{n}{r} \rceil$ consumer rooms, and the distance between each pair of resource room and consumer room is within the threshold. Thus "distributed evenly" is similar to an assignment and matching problem.

Although the distributed problem can be viewed as bipartite matching problem, the regular bipartite weighted matching algorithms cannot be used in this case, because those algorithms cannot solve the matching problem when upper and lower bounds are imposed on the number of assignments. Dondeti and Emmons [5] presented a method for solving weighted matching problem with bounded assignments. The basic idea of the method, referred to as the "node-splitting" method, is to transform the given problem into an assignment problem solvable by the regular weighted matching algorithm.

The first step of Dondeti and Emmons's algorithm is to construct a new bipartite graph from the input. "Processor nodes", or resource room nodes in our case, will be placed on one side of the bipartite graph, and "job nodes", or consumer room nodes, will be placed on the other side of the graph. For each processor i, a_i nodes are created where a_i is the lower limit on the number of jobs that must be assigned to processor i. In our distribution problem, a_i is the same for all resource rooms. Then $g_i - a_i$ additional nodes are created for each processor i, where g_i is the upper bound on the number of jobs that can be assigned to processor i. Again, g_i is the same for all resource rooms. Thus there are a total of $q = r * g$ processor nodes. Similarly, n nodes are created to represent each job. If $n < q$, $q - n$ additional nodes are created to represent dummy jobs. The strategy is to construct a bipartite graph with equal number of nodes on both sides. Dummy processor nodes can also be created if $n > q$. But we won't have to deal with this case because the *distributed* relationship is already violated under this situation.

Next, weights have to be labeled for edges connecting the nodes in the new bipartite graph. Let w_{kj} denote the weight of the arc (k, j). If c_{ij} is the distance between consumer room j and resource room i corresponding to node k, then

$$w_{kj} = c_{ij}, \qquad \text{for } j = 1, ..., n$$

and

$$w_{kj} = \infty, \qquad \text{for } j = n + 1, ..., q \text{ (dummy nodes)}.$$

Now we can find an one-to-one assignment for the $q \times q$ bipartite graph constructed by using the Hungarian method [15] with minimization objective. For our distribution problem, we try to minimize the distances between assigned resource rooms and consumer rooms. The Hungarian method solves the weighted bipartite matching problem with the following formulation:

$$Minimize \quad \sum_{k=1}^{q} \sum_{j=1}^{q} w_{kj} y_{kj},$$

such that

$$\sum_{j=1}^{q} y_{kj} = 1 \qquad \text{for } k = 1, ..., q,$$

$$\sum_{k=1}^{q} y_{kj} = 1 \qquad \text{for } j = 1, ..., q,$$

$$y_{kj} \geq 0, \qquad \text{for } k = 1, ..., q \text{ and } j = 1, ..., q.$$

The Hungarian method solves the matching problem by first finding a set of "permissible" edges in the bipartite graph with minimum weights and a matching from the permissible edges such that no two edges share the same node. If the result is not a perfect matching, i.e. all nodes in the graph are matched to one other node, the algorithm successively increases the set of permissible edges by finding the next set of minimum edges until a perfect matching can be assigned.

With the solution obtained from the Hungarian method, the final assignment of which job is assigned to which processor can be found. If node k corresponds to processor i and node j is not a dummy node,

$$\text{set } x_{ij} = 1, \qquad \text{if } y_{kj} = 1.$$

Given this assignment, we can now verify the *distributed* constraint by checking whether or not every consumer room is assigned to a resource room within the required distance. If one or more consumer rooms are assigned to a resource room outside of the threshold limit, it means some resource rooms have to serve more consumer rooms than they can accommodate, and thus the "distributed evenly" requirement is violated. We can adjust how "evenly" the rooms need to be distributed by giving a different setting to the lower and upper bound. For a strict distribution requirement, the lower bound can be set to $\lfloor \frac{n}{r} \rfloor$ and the upper bound can be set to $\lceil \frac{n}{r} \rceil$. For a looser distribution requirement, a tolerance can be subtracted and added to the strict version of lower and upper bound, respectively. As for the other three relationship verifications, the *distributed* evaluation process can be terminated earlier whenever a consumer room is found to not have at least one resource room within the threshold limit if the user only wants some quick feedback.

6 Example: Soda Hall

We have applied the described evaluation program to the computer model of Soda Hall. In the WALKTHRU project [7], the geometry of Soda Hall had been captured in the Berkeley UniGrafix data format. We used simplified slices through this building geometry to create symbolic building layouts for all seven floors. A roster file which identifies the room type for each room was generated manually. Also, parts of the original building specifications relevant to this study were captured in a SYLIF building program by using the specification entry form. Some key proximity relationships were derived from an adjacency matrix used in the Architectural CAD classes in 1995 and 1996. The simple numerical entities were enhanced to the new format that can express more adequately the relationships between different room types. In particular, the *distributed* relationship was applied to several pairs of rooms. Some of these pairs are:

(all offices) to printer rooms,
(faculty offices & visitor offices & student offices) to secretary offices,
(faculty offices & visitor offices & student offices) to research labs.

The complete SYLIF building program of Soda Hall contains 28 different room classes plus two super-classes. These two parent classes are **office** and **instructional room**. The class **office** has 10 subclasses to differentiate between different types of offices such as **faculty office**, **secretary office**, **student office**, and **staff office**. Class **instructional room** has 5 subclasses to describe types of rooms that are used to provide instruction to the students. A couple of these examples are **large class room** and **instructional lab**. Also, there are a total of 114 proximity constraints defined in the SYLIF file. But due to the inheritance property of SYLIF language, the real number of proximity constraints is more than the total defined in the file.

Most of the room count requirements were proven to be satisfactory in the Soda Hall building layout. The main discrepancies were found for several office types; for 7 out of the 10 office types the actual count fell short of the ideal number desired. Although about half of these office types have a shortage of only 1 or 2 rooms, **secretary office**, **student office**, **visitor office**, and **technical support office** have a shortage of either 6 or 7 rooms, which sum up to a total shortage of 27 rooms. The other violations occurred in the shortage of **small class room**, **machine room**, and **student consulting room**, with differences of only 1 or 2 rooms for each of these room types.

For the room area verification, the most glaring discrepancy was found in the requirement for the **receiving area**. The actual receiving area is smaller than the specification by almost 800 sq ft. The areas of **storage rooms** are also smaller than the required area. But on the other hand, there is a larger number of **storage room** than specified. In addition to the room shortage violation, half of the **student offices** also suffered from an area deficiency. **Student office** is possibly one of the sacrifices made by the client and the architect during the design evolution in order to satisfy the requirements of **faculty office**.

In our proximity verification, all three penalty factors (stairs, elevators, and doors) were included in the distance calculation, and all three path options (stairs, elevators, and rooms) were allowed in the path selection. We then made some a-priori assumptions about the distance thresholds. For the *clustered, at least one*, and *distributed* relationships, rooms having a proximity strength of *3* should probably be separated by no more than 3 or 4 rooms in between them. For a strength of *2*, the rooms should be on the same floor, and for a strength of *1*, the rooms should be at most one floor apart. For the *separated* relationship, rooms having an anti-proximity strength of *3* should ideally be in separate buildings, since they want to be as far away as possible. For a *separated 2* constraint, the rooms should not be located on the same floor. For a *separated 1* constraint, the rooms should be on the opposite ends if they are on the same floor. By looking at the building layout and building dimensions of Soda Hall, we came up with a set of threshold values for the proximity strength that we believed to be suitable and realizable. The interpretations and their associated distance thresholds are shown in table 2 and table 3.

proximity strength	**interpretation**	**distance**
1	"adjacent floor"	$\leq$ 3400 in (86.4 m)
2	"on the same floor"	$\leq$ 2200 in (55.9 m)
3	"half of corridor"	$\leq$ 800 in (20.3 m)

Table 2: Strength interpretations for the *clustered, at least one*, and *distributed* relationships

proximity strength	**interpretation**	**distance**
1	"opposite side of the same floor"	$\geq$ 1400 in (35.6 m)
2	"different floors"	$\geq$ 2600 in (66 m)
3	"ideally in separate buildings"	$\geq$ 6000 in (152.4 m)

Table 3: Strength interpretations for the *separated* relationships

In the total of 195 proximity verifications performed, 93 violations were found by the evaluator. Out of these 93 violations, there were 22 *clustered* constraint violations, 61 *separated* constraint violations, and 10 *distributed* constraint violations. In a real design process, the architect will try to modify the building layout to minimize the amount of violations and/or consult with the client to change some of the specifications that are too strict or impossible to maintain until a final design solution is found. In the case of Soda Hall example, we only want to show that major violations can be detected by the evaluator, since the building has already been built.

The majority of the violations found for the *clustered* relationships were due to the different types of offices that should have been close to one another, but are actually several floors apart. Examples are **faculty office** to **chair office** and to **division staff office**. All faculty offices are on the 6th and 7th floors, while the chair office and the division staff offices are on the 3rd floor, and thus are too far away from each other to satisfy the original specification. Many separated violations were found because Soda Hall is a compact building, and therefore it's hard to place rooms far away from each other within the building. Most of the *separated* relationship violations were due to some offices and some instructional rooms being on the same floor, and thus are too close in proximity while the specification had them widely separated. Another *separated* constraint violation of interest is the one placed between **storage room** and **storage room**. The idea was by separating the storage rooms from one and another, the storage rooms would more likely to be distributed evenly among their consumers. With this constraint being violated, the *distributed* constraint for **storage room** was also violated because too many storage rooms lie on the 3rd floor and thus are too far away from the offices on the 6th and 7th. One other *distributed* violation found is for **faculty office** and **secretary office** to **supervisor office**. With all supervisor offices on the 6th and 7th floors, it is no wonder that secretary offices on the 3rd floor cannot be assigned to a supervisor office within the specified limit. But this difficulty is also due to the fact that in the program there is no clear distinction between the administration secretary offices and the research secretary offices. All administration secretary offices are on the 3rd floor while all research secretary offices are on the 6th and 7th floors. If the secretary offices were further classified, different constraints could be placed on the different secretary offices to reduce the violations found. No violation for the *at least one* constraint was found, which is understandable since the *at least one* relationship is the easiest to satisfy out of the four grouping proximity types.

On a SGI O2 workstation with a 200 MHz processor, the whole proximity evaluation for Soda Hall took approximately 65 seconds. The whole proximity evaluation consists of 195 proximity constraints that were checked against the Soda Hall building layout, which contains approximately 250 rooms among its seven floors. Of the 65 seconds, the majority of the time was spent on room assignments for the *distributed* relationship verifications, since the running time for the Hungarian method, the main computation for the assignment problem, is $O(n^3)$. The major bottleneck was found for the room assignments where there are many consumers but a relative small number of resources. Examples are the **printer room** and **kitchen** *distributed* constraints. For **printer room**, there are 147 consumers but only 5 resources, and for **kitchen**, there are also 147 consumers but only 4 resources. Time spent for these two room assignments were about 45 seconds, which accounted for 70% of the total time. Verification times for instances checking and area evaluation were less than one second each, and thus were insignificant when compared to the time spent for proximity evaluation.

7 Discussion and Future Work

Currently, the criterion of our proximity verification is based solely on walking distance. This places the limitation on the system that it can only verify relationships depending on the distances between the rooms. Other possible criteria that are useful to include in the evaluator are acoustic and visual connection. An example of the usage of acoustic evaluation would be a group of offices that need to be close in distance to a discussion room, but at the same time, they want to be separated from the discussion room acoustically. This relationship creates a contradiction when distance is the only measure. The SYLIF building program provides syntax to specify visual and acoustic connection. But in order to verify these two kinds of connections, additional informations such as the material of the walls and doors must be included in the building layout.

Although the current proximity evaluation does not allow to specify or to verify direct adjacency, i.e. two rooms must be connected by a portal, it is not difficult to incorporate direct adjacency checking into the program. The SYLIF building program already contains the syntax for specifying when a pair of rooms are to be physically connected by a portal by using the keyword *conn*, which stands for connected, instead of one of the three proximity strength. Then, to verify whether or not the constraint is met, we simply check to see if the two rooms have a portal in common.

Under the current implementation, only one distance value is associated with each proximity strength value that distance tests are done by strict greater than or less than comparisons. For example, a *clustered 3* relationship is verified by making certain the minimal distance between the two parties is strictly less than or equal to the corresponding threshold value in the distance metric. But it is also possible to have a distance range associated with each proximity strength. An example where this can be usefulness is the *separated 1* situation, where the two spaces don't want to be close together and yet they should not be too far apart either, as in the case of office and restroom. If a range is associated with each strength value, the *separated* relationship could be eliminated from the representation since each proximity strength now represents the "proper" distance the two parties should maintain. The use of range would create a stronger distinction between the different level of proximity strength.

The adjacency graph is built for the whole building, and hence the Dijkstra's algorithm has to search the entire graph in order to determine shortest paths between spaces. This can become time consuming if the building is large and contains thousands of rooms and portals. A solution to speed up the computation would be to break up the adjacency graph into several different subgraphs. For a multiple-story building, a logical division would be to have an adjacency graph for each floor, and then another adjacency graph for all the floors to describe the connectivities between different floors through stairs and elevators. So for spaces on the same floor, the shortest paths search can be done much more quickly on the smaller adjacency graph. For even finer granularity, an adjacency graph can also be built for each zone on the same floor.

Coming up with a good distance metric default values can be a difficult task if the client does not have a basic idea of what are the appropriate threshold distances for the building to be designed. A possible solution would be to use case studies from post-occupancy evaluation of similar types of buildings. Those statistics would be helpful for generating a set of threshold values that could use in the initial evaluation. For the shorter distance thresholds associated with the *clustered 3, at least one 3, distributed 3*, and *separated 1* relationships, the values should more or less the same for all buildings. But for the larger distance thresholds, the values might be more related to the size and the structure of the buildings.

Once the SYLIF parser is ready, this evaluation tool will incorporate the parser and use SYLIF's building description to obtain the full power of the SYLIF language. Even though the Berkeley UniGrafix file format is adequate for capturing the geometric representation of a symbolic floor plan, it does not provide some important building information such as floor, fire zone, vertical connection, and identification of room type. The SYLIF language, on the other hand, is designed to capture all relevant information for a building design, and therefore is much more suitable for our purpose. Furthermore, a symbolic floor plan editor for the SYLIF layout format is under development. An even more ideal case would be to integrate the evaluation program with the floor plan editor, but the evaluation would only be performed when the architect makes the request. The integration would allow the architect to verify the current design without going back and forth between different programs, and thus would facilitate the design process.

8 Conclusion

We have developed an architectural CAD tool that performs the task of evaluating tentative floor plans proposed by the architect against a set of building requirements given by the client. The types of specification that the tool is able to evaluate include type and number of instances, room areas, and proximity relationships. A specification entry form is being provided to allow the user to enter these building specifications with minimal effort. In order to perform the evaluation of proximity constraints reasonably and efficiently, we have introduced a new representation to describe proximity relationships between a pair of spaces by specifying one of the four possible groupings: *clustered, separated, at least one*, and *distributed*. This new proximity representation allows more flexibility in defining the desirable spatial relationships between different spaces than a simple adjacency matrix. We have included penalty factors in the distance calculations of paths leading through closed doors or up and down through staircases and elevators. The users are allowed to adjust these weights according to their special needs or preferences.

Although the evaluation program can accept building layouts for any type of building, it is most useful for the design of large institutional buildings with dozens of rooms of the same type, and with hundreds of rooms overall spreading across a large region, and where many requirements have to be satisfied that are hard to keep track of by visual inspection. Our goal is to provide the architect with means to verify effortlessly the client's specifications during design evolution, and thus have more time and energy left to focus on aesthetic design issues.

References

[1] G. Bashein and P.R. Detmer. Centroid of a polygon. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 3–6. AP PROFESSIONAL, 1994.

[2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, pages 514–531. McGraw-Hill, 1990.

[3] G.S. Couch. Berkeley UNIGRAFIX 3.1- Data Structure and Language. Technical Report UCB/CSD-94-830, University of California, Berkeley, September 1994.

[4] N. Cross. *The Automated Architect*. Pion, 1977.

[5] V.R. Dondeti and H. Emmons. Max-min matching problems with multiple assignments. *Journal of Optimization Theory and Applications*, 91(2):491–511, 1996.

[6] L. Downs. Interchange format for symbolic building design. Research in progress at University of California, Berkeley.

[7] T.A. Funkhouser, S.J. Teller, C.H. Séquin, and D. Khorramabadi. UCB system for interactive visualization of large architectural models. *Presence: Special Issue on Teleoperators and Virtual Environments*, 5(1):13–44, Winter 1995.

[8] R. Hashimshony, E. Shaviv, and A. Wachman. Transforming an adjacency matrix into a planar graph. *Building and Environment*, 15:205–217, 1980.

[9] J.C. Jones. *Design Methods*, pages 300–303. Van Nostrand Reinhold, 1992.

[10] Y. Kalay and E. Shaviv. A method for evaluating activities layout in dwelling units. *Building and Environment*, 14(4):227–234, 1979.

[11] D. Kernohan, G. Rankin, G. Wallace, and R. Walters. Relationship models: analytical techniques for design problem solving. *Architectural Design*, 43:275–278, May 1973.

[12] D. Kernohan, G.D. Rankin, G.D. Wallace, and R.J. Walters. PHASE: an interactive appraisal package for whole hospital design. *Computer Aided Design*, 5(2):81–89, 1973.

[13] R. Lewis and C. Séquin. Generation of 3D building models from 2D architectural plans. *Computer-Aided Design*, 30(10):765–779, 1998.

[14] T.W. Maver. A theory of architectural design in which the role of the computer is identified. *Building Science*, 4(4):199–207, 1970.

[15] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization : Algorithms and Complexity*, pages 248–255. Prentice Hall, 1982.

[16] C.H. Séquin and Y. Kalay. A suite of prototype CAD tools to support early phases of architectural design. *Automation in Construction*, 7(6):449–464, 1998.

[17] E. Shaviv and D. Gali. A model for space allocation in complex buildings: A computer graphics approach. *Build International*, 7(6):493–518, 1974.

[18] R. Th'ng and M. Davies. SPACES: an integrated suite of computer programs for accommodation scheduling, layout generation and appraisal of schools. *Computer Aided Design*, 7(2):112–118, 1975.

[19] B. Whitehead and M.Z. Eldars. An approach to the optimum layout of single-storey buildings. *The Architects' Journal*, 139(25):1373–1380, June 1964.