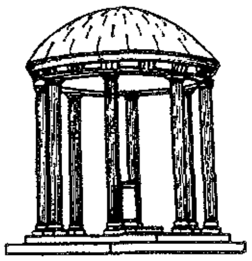
Constraint-Based Design by Cost Function Optimization

TR91-025

May, 1991

Eric Grant



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Constraint-Based Design by Cost Function Optimization

by

Eric Grant

A dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1991

Approved by:

[illegible]

Advisor: Turner Whitted

[illegible]

Reader: James M. Coggins

[illegible]

Reader: Hiroyuki Watanabe

ERIC DOUGLAS GRANT.
Constraint-Based Design by Cost Function Optimization
(Under the direction of Turner Whitted).

Abstract

Constraint-based design is the process of selecting among alternatives to best satisfy a set of potentially conflicting goals. A key problem in constraint-based design is finding globally optimal solutions to problems without limiting the complexity of constraints.

In this work, constraints are encoded as cost functions that express how well the constraints are satisfied. A geometric modeling problem is defined by specifying a collection of constraints on the desired model. A composite cost function representing all constraints is formed by combining the component cost functions.

The optimal solution to a constraint problem can be found by minimizing the value of the composite cost function. A standard probabilistic optimization technique, simulated annealing, is used to seek the global minimum value and the corresponding optimal solution. In practice, global optima cannot be guaranteed, but often near-globally optimal results are satisfactory.

The cost function representation for design problems is not new; VLSI researchers have used annealing-based optimization methods to minimize chip area and wire length. The cost functions for general constraint-based modeling problems are not as well defined as the simple VLSI cost functions. A contribution of this research is a systematic method of encoding different classes of constraints as cost functions.

The validity of this approach is demonstrated by applying the methodology to two problems: product design (specifically, opaque projector design), and site planning.

Acknowledgements

I thank my advisor, Turner Whitted, for his guidance of this research. I admire his broad knowledge of computer science and engineering.

I thank Frederick P. Brooks, Jr., James M. Coggins, Henry Fuchs, and Hiroyuki Watanabe for serving on my committee and providing advice and encouragement throughout my studies at UNC.

I thank David Chapin and Ken Pittman of North Carolina State University for providing excellent test applications for this research, and for informative discussions about design.

I thank my fellow students at UNC, and colleagues elsewhere, for their assistance with this work. Their comments, reviews, and discussions have contributed considerably to the dissertation.

I thank Schlumberger-Doll Research and Apple Computer for grants supporting this work.

Finally, I thank my family for support and encouragement throughout my graduate career. I dedicate this dissertation to my mother and father.

Table of Contents

Chapter	Page
I. Introduction	1
1.1. What is a constraint?	2
1.2. How does this work differ from other work?	3
1.3. Description of problem representation	4
1.4. Optimizing cost functions	7
1.5. Building cost functions	8
1.6. Characteristics of applications	9
1.7. Chapter overview	11
1.8. Summary	11
II. Constraint Satisfaction Techniques	13
2.1. Constraint-based modeling	13
2.1.1. Problem complexity	13
2.1.2. Modeling vs. rendering	14
2.2. Taxonomy of techniques	15
2.3. Analytical techniques	15
2.3.1. Constraint graphs	16
2.3.2. Local propagation	16
2.3.3. Propagation of degrees of freedom	17
2.3.4. Graph transformation / term rewriting	18
2.3.5. Equation solving	18
2.4. Modeling systems using analytical techniques	19
2.4.1. Van Wyk (1980)	19
2.4.2. Gosling (1983)	20
2.4.3. Brüderlin (1986)	20
2.4.4. Rossignac (1986)	20
2.4.5. Leler (1987)	21
2.4.6. Wolfram (1988)	21
2.4.7. Ervin (1990)	21
2.5. Summary of analytical techniques	22
2.6. Optimization methods	23
2.6.1. Linear programming	23
2.6.2. Nonlinear programming	23
2.7. Summary of optimization methods	24

2.8. Heuristic search 24
2.8.1. Pfefferkorn (1971) 25
2.8.2. McDermott (1982) 25
2.8.3. Brown (1986) 25
2.9. Summary of heuristic search 25
2.10. Improvement procedures 26
2.10.1. Variation with selective retention 26
2.10.2. Greatest improvement 27
2.10.3. Numerical methods 27
2.11. Systems using improvement procedures 28
2.11.1. Sutherland (1963) 28
2.11.2. Weinzapfel and Handel (1975) 28
2.11.2.1. Automated constraint satisfaction 29
2.11.2.2. Interactive constraint satisfaction 29
2.11.3. Cinar (1975) 29
2.11.4. Borning (1979) 30
2.11.5. Nelson (1985) 30
2.11.6. VLSI Layout: Kravitz (1986), Pincus (1986), etc. 31
2.11.7. Witkin, Fleischer, Barr (1987) 31
2.11.8. Barzel and Barr (1988) 32
2.12. Summary of improvement procedures 32
2.13. Generate-and-test 32
2.13.1. Exhaustive generate-and-test 33
2.13.2. Random generate-and-test 33
2.14. Summary of generate-and-test 34
2.15. Applicability of constraint satisfaction techniques 34

III. Constraint-based Design 35
3.1. The role of the computer 35
3.2. Why are design problems difficult? 37
3.3. Specific objectives of this research 37
3.3.1. Geometric and non-geometric constraints 38
3.3.2. Arbitrarily complex constraints 38
3.3.3. Separation of specification from solution 39
3.3.4. Dealing with overconstrained and underconstrained systems . 39
3.3.5. Suitability as a design aid 39
3.4. Exploration of solution methods 39
3.5. The cost function representation 42
3.6. Characteristics of applications 43

3.6.1. Near-optimal solutions acceptable 43
3.6.2. Many optimal solutions 44
3.6.3. Non-intuitive solutions 44
3.6.4. Randomness acceptable of desirable 45
3.7. Summary 45

IV. Function Optimization 46
4.1. Formulation as an optimization problem 46
4.2. Characterization of the optimization problem 46
4.3. The limitations of simple hill-climbing techniques 48
4.4. Description of probabilistic optimization techniques 49
4.4.1. Probabilistic hill-climbing heuristics 49
4.4.2. Simulated annealing 50
4.5. Applying annealing to constraint-based modeling 53
4.5.1. Solution representation 54
4.5.2. Objective function 54
4.5.3. Perturbation function 54
4.5.4. Acceptance function 56
4.5.5. Annealing schedule 56
4.5.5.1. Simple schedule 57
4.5.5.2. Sophisticated schedule 58
4.5.6. Evaluating the solution 60
4.6. Summary 61

V. Cost Functions 62
5.1. Examples of constraints 62
5.1.1. Geometric constraints 63
5.1.1.1. Position constraints 63
5.1.1.2. Distance constraints 63
5.1.1.3. Orientation constraints 64
5.1.1.4. Size constraints 64
5.1.1.5. Intersection constraints 64
5.1.2. Indirectly geometric constraints 64
5.1.3. Non-geometric constraints 65
5.2. Theory and use of cost functions 65
5.3. General cost function guidelines 66
5.3.1. Units of measure for cost functions 66
5.3.1.1. Choose a single unit of measure 66
5.3.1.2. Choose dollars as the unit of measure in product design 66
5.3.1.3. Select a standard range of values 67

5.3.2. Shape guidelines for cost functions 67
5.3.2.1. Define the cost of compromise configurations . . . 67
5.3.2.2. Shape should be accurate and easy to specify . . . 69
5.3.2.3. Attempt to separate representation from specification 69
5.3.3. Constructing functions to aid the optimization process 69
5.3.3.1. Functions should encourage hill-climbing 69
5.3.3.2. Always reward movement toward the goal 71
5.3.3.3. Functions should be easy to evaluate 71
5.4. Cost functions for objective constraints 71
5.4.1. Selecting the controlling parameter 72
5.4.2. The six basic constraint types 74
5.4.2.1. Equal to 75
5.4.2.2. Not equal to 75
5.4.2.3. Less than 76
5.4.2.4. Greater than 76
5.4.2.5. Minimize 77
5.4.2.5.1. Unbounded minimize 77
5.4.2.5.2. Bounded minimize 77
5.4.2.6. Maximize 78
5.4.2.6.1. Unbounded maximize 78
5.4.2.6.2. Bounded maximize 78
5.4.3. Qualities of cost functions 79
5.4.3.1. Areas of interest 79
5.4.3.2. Bound on function range 80
5.3.3.3. Monotonically increasing or decreasing cost 80
5.3.3.4. User control over shape 81
5.4.4. Examination of candidate cost functions for =, ≠, <, > . . . 81
5.4.4.1. Gaussian: $f(x) = e^{-x^2/2\sigma^2}$ 82
5.4.4.2. $f(x) = e^{-nx^b}$ 83
5.4.4.3. Piecewise cubic: $f(x) = ax^3 + bx^2 + cx + d$ 84
5.4.4.4. $f(x) = \frac{1}{nx^b+1}$ 88
5.4.4.5. Abstraction of user control 89
5.4.4.6. Cost functions: =, ≠, <, >, bounded min. and max. 90
5.4.5. Cost functions for unbounded minimize and maximize 91
5.4.5.1. The minimize cost function 92
5.4.5.2. The maximize cost function 93

5.4.6. Summary of cost function design for objective constraints . . 94
5.4.7. Analysis of cost functions of objective constraints 95
5.4.7.1. Functions should be easy to evaluate 95
5.4.7.2. Shape should be accurate: theoretical considerations . 96
5.4.7.3. Shape should be accurate: practical considerations . 96
5.4.7.4. Shape should be easy to specify 97
5.4.7.5. Specification should be separate from representation . 97
5.4.7.6. Functions should encourage hill-climbing 98
5.4.7.7. Summary of analysis of cost functions 99
5.5. Cost functions for subjective constraints 100
5.5.1. Divide and conquer to refine the problem 100
5.5.2. Base judgements on a simple scale 101
5.5.3. Match empirical data if constraints cannot be quantified . . . 101
5.6. Cost functions for search constraints 102
5.7. Operations on constraints 103
5.7.1. A and B 103
5.7.2. A or B 103
5.7.3. Not A 104
5.8. Summary 104

VI. Opaque Projector Project 106
6.1. Problem overview 106
6.1.1. Constraint conflicts 108
6.1.2. Projector classes 108
6.2. Parts inventory 109
6.3. Constraint implementation 111
6.3.1. Image brightness 112
6.3.2. Component cost 112
6.3.3. Light escaping 112
6.3.4. Sound escaping 112
6.3.5. Cord length 113
6.3.6. Energy use 113
6.3.7. Image focus 113
6.3.8. Projector geometry 114
6.3.9. Projector height 115
6.3.10. Temperature constraints 115
6.3.11. Chassis cost 116
6.3.12. Ergonomics 116
6.3.13. Weight 117

6.3.14. Durability 117
6.3.15. Full view 117
6.4. Results 118
6.5. Evaluation 126
6.6. Summary 130

VII. Site Planning Project 131
7.1. Problem overview 131
7.1.1. Objects 131
7.1.2. Constraints 134
7.1.3. Constraint conflicts 135
7.2. Constraint implementation 135
7.2.1. Object non-intersection 135
7.2.2. Road setbacks 136
7.2.3. Place buildings away from roads 136
7.2.4. Interbuilding spacing 137
7.2.5. Place buildings near parking lots 137
7.2.6. Place buildings on hills 137
7.2.7. Avoid unpleasant views from buildings 138
7.2.9. Discussion of cost function weighting factors 138
7.3. Results 138
7.4. Statistics 157
7.5. Evaluation 159
7.6. Convergence to final solution 160
7.7. Summary 165

VIII. Discussion 166
8.1. Comparisons with other systems 166
8.1.1. ThingLab 166
8.1.2. Ideal 167
8.1.3. Brüderlin's system 167
8.1.4. Brown's system 167
8.1.5. Barzel and Barr's system 167
8.1.6. CONTEST 167
8.2. Contributions 174
8.2.1. Specific contributions 174
8.2.2. Advancement of the state-of-the-art 175
8.3. Implementation 176
8.4. Summary 177

IX. Conclusions / Future work / Summary 178
9.1. Conclusions 178
9.2. Future work 179
9.2.1. Shape grammars 180
9.2.2. Combining grammars and cost functions 180
9.3. Summary 181

References 182

List of Figures

Figure 1.1: black box function evaluation 5
Figure 1.2: improvement search technique 6
Figure 1.3: functions with single maximum and multiple local optima 7
Figure 2.1: constraint-based layout 14
Figure 2.2: a simple constraint graph 16
Figure 2.3: a constraint graph with a cycle 17
Figure 2.4: graph transformation 18
Figure 2.5: variation with selective retention 26
Figure 2.6: greatest improvement procedures 27
Figure 2.7: generate-and-test 33
Figure 3.1: semi-automatic design of a single model 36
Figure 3.2: progression of investigation 41
Figure 3.3: global vs. local optima in a packing problem 44
Figure 4.1: black box function evaluation 47
Figure 5.1: a simplistic cost function 68
Figure 5.2: a more useful cost function 68
Figure 5.3: a wider region of accepted values 70
Figure 5.4: guiding search to optimal value 70
Figure 5.5: constraints and associated parameters 73
Figure 5.6: equal to 75
Figure 5.7: not equal to 75
Figure 5.8: less than 76
Figure 5.9: greater than 76
Figure 5.10: minimize 77
Figure 5.11: lower bounded minimize 77
Figure 5.12: upper bounded minimize 77
Figure 5.13: maximize 78
Figure 5.14: lower bounded maximize 78
Figure 5.15: upper bounded maximize 78
Figure 5.16: cost functions with different tolerances 79
Figure 5.17: user shape specification 81
Figure 5.18: piecewise construction 84
Figure 5.19: piecewise function with finite extent 85
Figure 5.20: f(x) 86
Figure 5.21: g(x) = f(x-v) 86
Figure 5.22: general form of g(x) given f(x) 90

Figure 5.23: restricted vs. unrestricted hill-climbing 98
Figure 5.24: divide-and-conquer approach 101
Figure 6.1: opaque projector operation 108
Figure 6.2: thin lens imaging model 114
Figure 6.3: standard projector 120
Figure 6.4: budget projector 121
Figure 6.5: deluxe projector 122
Figure 6.6: quiet projector 123
Figure 6.7: bright projector 124
Figure 6.8: portable projector 125
Figure 6.9: comparison of projector attributes 127
Figure 6.10: comparison of projector weights 128
Figure 7.1: road network with road widths 132
Figure 7.2: terrain elevation 133
Figure 7.3: road setbacks and building non-intersection constraints 139
Figure 7.4: add maximization of road/building distance 141
Figure 7.5: add interbuilding spacing 143
Figure 7.6: add parking lots 145
Figure 7.7: minimize parking lot/building distance 147
Figure 7.8: place buildings on hilltops 149
Figure 7.9: add trees as obstacles 151
Figure 7.10: match roof heights locally 153
Figure 7.11: final site plan 155
Figure 7.12: site plans at high temperature 161
Figure 7.13: site plans as the system begins to cool 162
Figure 7.14: site plans after further cooling 163
FIgure 7.15: site plans at final stages of cooling 164
Figure 8.1: structure of ThingLab 168
Figure 8.2: structure of Ideal 169
Figure 8.3: structure of Brüderlin's system 170
Figure 8.4: structure of Brown's system 171
Figure 8.5: structure of Barzel&Barr's system 172
Figure 8.6: structure of CONTEST 173
Figure 8.7: implementation of CONTEST 176

List of Tables

Table 5.1: summary of cost function analysis 99
Table 6.1: cost function weights for projector instances 119
Table 6.2: comparison of part and parameter selection 126
Table 6.3: CPU times for opaque projector problems 129
Table 7.1: constraint statistics for site planning application 157
Table 7.2: CPU times for site planning problems 158

Chapter 1

Introduction

The research presented in this dissertation began with the following goal: to devise an automatic design method capable of applying diverse constraints to many different types of geometric objects.

Specifically, the question we sought to answer was: *can a design problem involving diverse geometric and non-geometric constraints be expressed in such a way that the problem may be solved automatically by a computer?* This dissertation will show that the answer to this question is yes — design problems can be represented by a single scalar cost function in which minimum function values occur when optimal design parameters are presented as input to the function. Scalar functions can be minimized by probabilistic optimization techniques, such as simulated annealing, to yield near-optimal solutions. This research combines a systematic way of encoding constraints as cost functions with standard annealing-based optimization techniques to form a flexible constraint-based modeling system.

CONTEST (a CONstraint TESTbed) is a software testbed built to explore constraint-based modeling problems. The style of design promoted by this work is called result-oriented design. In result-oriented design, models are created automatically from a set of properties that define the model, rather than by manually specifying data points, or by describing a series of steps for constructing the model from its parts. A difference between this research and other work is that rather than attempting to always solve a small set of predefined constraint types (e.g., only constraints expressible as linear equations), a variety of potentially conflicting constraints may be presented. While the user is responsible for specifying the importance of each constraint, the system is responsible for finding a solution that best satisfies the set of constraints as a whole. The design methodology is geared toward solving design problems, rather than using constraints solely as a means of communicating a preconceived design.

Although the original purpose of this research was to explore constraint-based three-dimensional geometric modeling problems, the techniques developed are applicable to other fields, such as VLSI design. The generic term *configuration* is used to refer to potential solutions to a problem. A configuration is a collection of parameters that defines a model.

A key aspect of design problems is that they involve the evaluation of tradeoffs. Thus the cost function which encodes a constraint must not only be able to identify configurations which satisfy the constraint, but must also define the relative quality of any other configurations, so that an optimum compromise may be determined should constraints conflict. The heart of this research is a systematic method for defining cost functions to properly represent costs over all configurations.

This chapter provides an introduction to the remainder of the dissertation. First, the term *constraint* is defined, and an explanation of how this research differs from related work is presented. The problem representation used by CONTEST is then described: constraints are encoded as cost functions which specify how well the constraint is satisfied. Next, techniques for finding the optimal solution to the cost functions are introduced. Construction of cost functions is then discussed, followed by a description of the types of applications well-suited to CONTEST. The chapter concludes with an overview of remaining chapters and a summary.

1.1. What is a constraint?

Since the term *constraint-based modeling* encompasses such a large body of work, it is important to distinguish the goals of this research from related work. We begin by examining the various interpretations of the term *constraint*, and define its meaning in this research.

In its simplest form, a constraint is simply a relation that must hold in the object we are designing. Many applications use constraints in this way as an efficient expression of a design. The designer creates an object by selecting a set of constraints that completely describes the object. When used in this manner, the user demands that all constraints be completely satisfied by the constraint-satisfaction system.

For example, to define a square, we can define the length of any side, constrain the other sides to be of equal length, and constrain any angle to be 90 degrees. In this style of design, we usually know exactly what our desired object should look like (e.g., we might be creating a figure for a paper). Thus the issue is not one of finding the best (or at least an acceptable) solution from among many possibilities, but rather of finding *the* correct solution.

Because the types of constraints in such applications are generally quite simple, and because of the rigorous definition of what constitutes an acceptable solution, algorithmic techniques are commonly used to find the solution. Moreover, given a proper problem formulation, these algorithmic techniques are often *guaranteed* to find the solution.

In other modeling problems the constraints are more complex. In addition, we may specify a problem for which it is impossible to satisfy all constraints. For example, an architect might specify that parking spaces for a building must not be visible from offices in the building. On the other hand, local building codes might require handicapped parking access within 50 yards of the building. In this case, we should expect the building code to override the aesthetic constraint.

Artificial intelligence researchers have dealt with such problems by distinguishing between satisfying *strong* constraints and satisfying *weak* constraints. Strong constraints are constraints that must be satisfied for the problem to be solved. Weak constraints specify guidelines for an acceptable solution, but it is not mandatory that they be satisfied. In the example above, the constraint that offices should not look out on a parking lot is a weak constraint. The constraint that handicapped parking access should be provided is a strong constraint.

In addition to constraints which theoretically can be satisfied, it may be useful to specify goal constraints which can never be satisfied. For example, an automobile designer usually wishes to minimize the drag coefficient in a car design. Such a desire constrains the design, but there is no situation where this constraint can be considered completely satisfied, since it impossible to have a drag coefficient of zero. This type of unsatisfiable constraint is called a goal constraint, or simply a *goal*.

In CONTEST, a constraint is any expressed guideline that influences a design. Because of the problem formulation, it is not necessary to distinguish weak vs. strong constraints, or regular vs. goal constraints. All of these constraints are represented in the same manner, and the term constraint will henceforth be used to refer to any type of design guideline.

1.2. How does this work differ from other work?

CONTEST differs from other geometric modeling systems in that it provides greater flexibility in constraint specification. It removes the following limitations of other systems:

- limitations in constraint complexity (e.g., only linear or quadratic equations)
- limitations in constraint type (e.g., only geometric constraints)
- the need for human problem solving knowledge (e.g., expert systems)
- limitations in the ability of the constraint solver to handle new constraints without changes to the constraint solver

Consider a constraint to minimize the drag of an aircraft wing. Many systems (e.g., those which represent constraints as linear equations) cannot represent a minimization constraint. Other approaches (e.g., expert systems) require an understanding of what determines drag and rules for minimizing it. CONTEST requires only that one be able to measure drag given a particular model. The solution method is automatic given this measurement.

The problem of solving systems of sophisticated constraints is, of course, extremely difficult. To attack this problem, CONTEST takes a step back from the goal of finding the *best* solution to a set of constraints, and instead seeks to find *near-optimal* solutions using a simpler solution method. The basic strategy is to recast the constraint-based modeling problem as a function optimization problem. Constraints are represented using scalar cost (or error) functions, which return a measure of how badly a particular data set violates the given constraint. The goal of the optimization is to minimize the sum of the error functions. This representation is described in the following section.

1.3. Description of problem representation

Choosing a proper problem representation is an important aspect of solving a problem. A concise problem statement can often suggest promising solution techniques. Other times, successful solution methods for related problems can lead to the problem formulation. In this research, the idea of representing constraints using scalar cost functions was examined and initially rejected, because of the lack of a suitable optimization method. Later, when results in other fields demonstrated successful global function optimization techniques, the idea of combining constraints into a single function was reexamined and forms the basis of this work.

The basic problem is to find the best solution to a collection of constraints. In other words, we seek the set of design variables which best satisfy some evaluation measure for the constraints. The design variables are the parameters that define a geometric model. These parameters usually will be geometric variables (coordinates of points, radii of spheres, etc.), though other non-geometric parameters (e.g., color) may be used as design variables.

The usual means of defining a problem is to specify a collection of constraints that should be considered simultaneously. This composition (*anding*) of constraints can be represented by summing individual cost functions so that each contributes to the total cost function. Section 5.7 shows that other logical operations (such as *or*) can be used to combine constraints. In all problems encountered thus far, additional operations have not been required, so the summation of individual cost functions is presented here as a simplification of the formation of the total cost function.

Each cost function, c_i, returns a real number which expresses how well that constraint is satisfied by a particular configuration. The cost functions for most constraints have the following form:

- zero, if the constraint is met
- a measure of how well the constraint is satisfied, otherwise (larger costs indicate poorer satisfaction)

Moreover, for each cost function, there exists a weighting factor, w_i, that specifies the importance of the constraint relative to other constraints.

In other words,

$$c_i = f_i(\textit{design parameters}),$$
$$w_i = \textit{weighting coefficient for constraint } i,$$
and
$$c_{total} = \sum w_i\, c_i = f_{total}(\textit{design parameters}).$$

The cost or quality of any design is determined by supplying the input (design parameters) to f_{total}, and observing the output, c_{total}. By expressing quality as a scalar value, the constraint satisfaction process is converted to an optimization problem. The goal is to find the design parameters which minimize c_{total}.

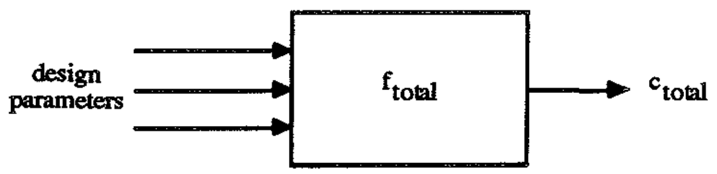


Figure 1.1: black box function evaluation

There are many strategies that might be applied in the search for an acceptable solution. One obvious solution is to generate and test all potential solutions. Unfortunately, this is infeasible because of the tremendous growth of the search space as the number of dimensions increases. Another alternative is to use *a priori* information about the function to concentrate the search in promising regions of the space. The difficulty with this approach is that the solution method must be adjusted as new constraints are added.

Instead, CONTEST treats the total cost function as a *black box*. With the black box assumption, information about the function must be learned by formulating inputs and observing the resulting costs. New potential solutions are created by modifying previous candidate solutions. Figure 1.2 shows the search method used in CONTEST.

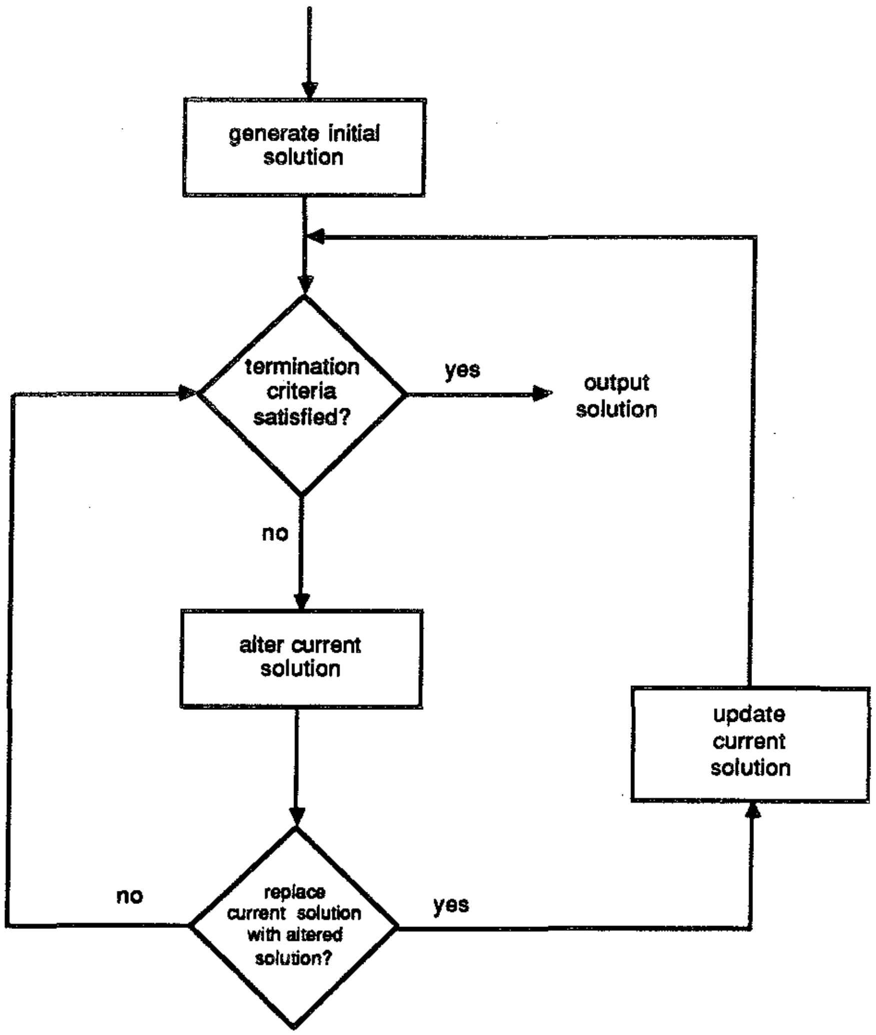
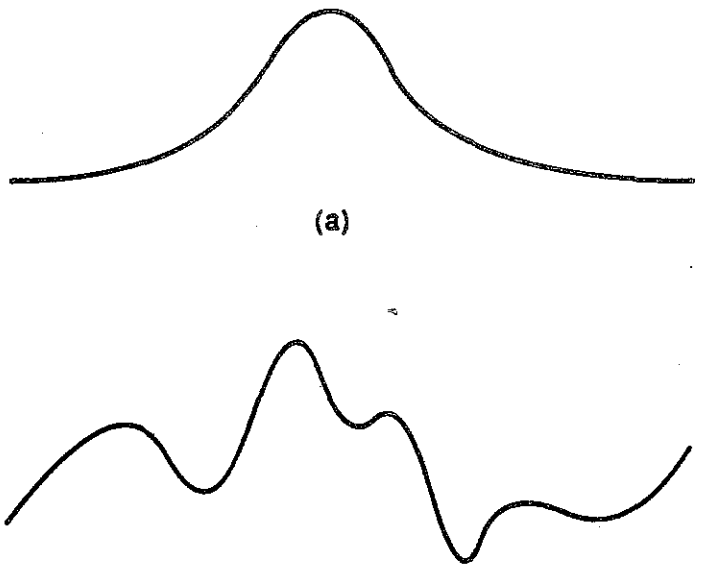

generate initial solution
termination criteria satisfied?
yes
output solution
no
alter current solution
update current solution
no
replace current solution with altered solution?
yes

Figure 1.2: Improvement search technique

1.4. Optimizing cost functions

The main difficulty in performing a black box optimization is finding a global, rather than local, optimum. Local optima are easy to find: simply choose a starting point and perform hill-climbing toward the goal[1]. Hill-climbing is suitable for unimodal functions, but unsuitable for functions with many peaks and valleys. Figure 1.3a shows a simple unimodal function which can be optimized with hill-climbing. Figure 1.3b shows a function with multiple local optima. These are simple one-dimensional functions. The total cost functions that CONTEST seeks to optimize are, of course, of much higher dimension. Chapter seven, for example, presents an application where cost is a function of 100 variables (x and y coordinates for fifty buildings).


(a)

(b)

Figure 1.3: functions with single maximum and multiple local optima

Simulated annealing is an alternate search technique for finding global optima. As with simple hill-climbing, simulated annealing generates new configurations by modifying the current configuration. It differs from simple hill-climbing in that it sometimes accepts new configurations which are worse than the current configuration. The acceptance of these bad configurations is necessary to explore the entire search space rather than getting stuck in local optima. Both the choice of a new configuration and whether to accept a bad

[1]The term hill-climbing will be used although CONTEST seeks to minimize the total cost function. The minimum of a function can be found by maximizing its negative.

one of six categories: constraining a parameter to be *equal to* a value, constraining a parameter to be *not equal to* a value, constraining a parameter to be *less than* a value, constraining a parameter to be *greater than* a value, *minimize* a parameter, and *maximize* a parameter.

A subjective constraint is a constraint involving a value judgement. Since these are not usually expressed in a computable form, such constraints must be broken down into simpler constraints that a computer can evaluate. A divide-and-conquer approach can be used to reduce value judgements to a series of objective evaluations, composed of numeric (e.g., 0.0 to 1.0) rankings, yes/no questions, and standard objective constraints.

A search constraint guides the constraint satisfaction search by providing levels of confidence to initial positions, or in general by imposing penalties for undesirable configurations. While all constraints penalize undesirable configurations, search constraints do so at the explicit direction of the user. For example, a designer can assign varying levels of confidence to the positions of model parts. When the constraint system is solved, the objects whose positions are assigned lower confidence values will be repositioned to satisfy the constraints, whereas other objects will tend to remain near their initial positions. The confidence level defines the freedom of the variable to solve the constraints, where low confidence implies high freedom.

1.6. Characteristics of applications

The benefits and limitations of the cost function formulation and simulated annealing optimization procedure define the class of problems for which CONTEST can be a useful tool. These benefits and limitations are introduced in this section and are discussed in greater detail in chapter three.

The cost function approach has several advantages. It can handle straightforward geometric constraints (e.g., distance between two objects), non-geometric constraints (e.g., constraints on color), and constraints which are indirectly specified as a function of geometry (e.g., drag coefficient of a car). These constraints can be arbitrarily complex; the only requirement is that the cost function must be capable of evaluating any potential solution. Underconstrained, exactly constrained, and overconstrained problems are all properly represented by the total cost function. The optimization technique used by CONTEST is independent of the problem specification. The constraint designer is therefore responsible only for constructing a cost function that measures constraint satisfaction; he is not responsible for devising a solution method for the constraints. The user can thus explore design problems even when he has little intuition about possible solutions.

The cost function approach used by CONTEST also has several limitations. First, it cannot guarantee that an optimal solution will be found. In fact, CONTEST cannot even determine whether a potential solution is optimal. Second, the solution technique is compute intensive because it involves iterative search of a high-dimensional space. Third, the probabilistic nature of the solution technique results in random positioning of underconstrained components (this can also be considered a feature). Finally, the process of devising cost functions can be time-consuming for new applications.

CONTEST is therefore well-suited for the following types of problems:

- applications with diverse, complex constraints
- applications with conflicting constraints
- applications where near-optimal solutions are acceptable
- applications with many optimal solutions
- applications with non-intuitive solutions
- applications where the human design process is not understood
- applications where some randomness is desirable or acceptable
- problems in which algorithmic methods are infeasible

Applications with some of these properties include computer-aided product design, modeling of natural phenomena, building layout, and architectural site planning.

Two problem areas, product design (a family of opaque projectors) and site planning, were explored as sample applications for the techniques developed in this dissertation. Satisfactory results were achieved in both of these diverse cases, suggesting a broad applicability of the techniques.

1.7. Chapter overview

Chapter two summarizes previous work in constraint-based graphics. A taxonomy of constraint satisfaction techniques exhibits the wide range of solution techniques used in constraint-based modeling. Chapter three describes constraint-based design issues relative to CONTEST. It details the features and limitations of the problem representation in this research. Chapter four discusses function optimization, presents the general form of probabilistic optimization techniques, and shows how constraint-based modeling can be performed by simulated annealing. Chapter five provides guidelines for creating cost functions for objective, subjective, and search constraints. Chapters six and seven demonstrate the usefulness of the technique. In chapter six, product design constraints are used to build an application for exploring opaque projector designs. In chapter seven, constraints for architectural site planning are encoded to generate designs for campus-like site plans. Chapter eight compares CONTEST with several systems using alternative solution techniques. Chapter nine summarizes the dissertation and provides conclusions and directions for future research.

1.8. Summary

The central thesis of this dissertation is the following:

> *Design problems involving diverse, complex geometric and non-geometric constraints can be solved by converting the problem formulation into a scalar cost function.*

The cost function expresses how well the constraint is satisfied by a particular model configuration. Typically, if the constraint is satisfied, the cost function is zero. If the constraint is not satisfied, then the value of the cost function is non-zero, and increases in magnitude as constraint violation increases.

A composite cost function, called the total cost function, is usually formed by taking a weighted sum of all individual cost functions. The total cost function expresses how well a particular model satisfies all constraints. The inputs to the cost function are the variables that define the model. The optimal configuration (the best solution to the constraints) is the configuration that yields the minimum total cost.

A probabilistic optimization technique, simulated annealing, is used to search for the optimal configuration. In theory, simulated annealing can find the global optimum of any function. In practice, limited computing budgets prevent the optimum from always being found. Fortunately, near-optimal solutions are often suitable in many applications.

Constraints have been categorized into three classes, based on the methods for constructing their corresponding cost functions. Objective constraints use a simple evaluation of a model property as input to a function which reflects the shape of six basic constraint types: less than, greater than, equal to, not equal to, minimize, and maximize. Subjective constraints require a divide-and-conquer approach to break subjective issues into a collection of objective questions. Search constraints help to concentrate the search in user-specified portions of the search space.

The solution technique is not applicable to all modeling problems. Because it cannot guarantee optimal results or tell when an optimal solution has been reached, it is best suited to applications in which near-optimal solutions are acceptable. The technique can easily handle underconstrained and overconstrained problems. Since the solution technique is totally automated, CONTEST is able to tackle problems where no human solution methods are known, or where an expert is unavailable.

Chapter 2

Constraint Satisfaction Techniques

This chapter describes previous constraint-based systems, and the constraint-satisfaction techniques they use. The systems described are drawn from various fields and organized in a taxonomy. The chapter concludes with a discussion of the applicability of these techniques to this research.

2.1. Constraint-based modeling

Constraints have been used in a variety of computer graphics applications, beginning with Sutherland's Sketchpad system [Sutherland, 1963]. Constraint-based modeling techniques have also been used in other fields, such as architecture, mechanical, electrical, and civil engineering.

In fact, the previous research most relevant to this dissertation comes from the engineering fields. Few references to constraint-based modeling systems in other disciplines can be found in the computer graphics literature. Two reasons for this oversight exist: the first deals with problem complexity and the second with an artificial requirement of rendering techniques in computer graphics.

2.1.1. Problem complexity

One reason for the lack of references to constraint-based modeling systems from other fields in computer graphics is the difference in complexity of design problems in different fields. Constraints have been used in computer graphics to make it easier for a designer to describe a shape to the computer. For example, constraints may be applied to a rough sketch of a design to yield precise geometry. The design is already in the designer's mind; the problem is to communicate the design to the computer. For instance, to simplify the task of drawing an equilateral pentagon, the user might first sketch a rough five-sided polygon, and then apply constraints to adjust the geometry, as depicted by figure 2.1.

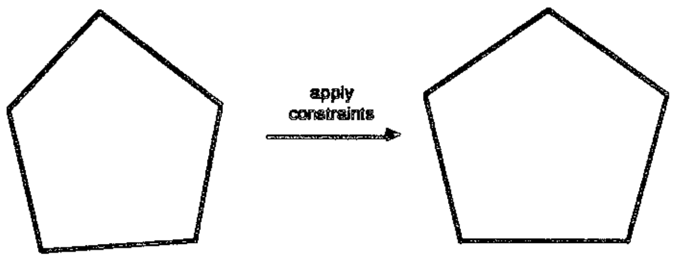


Figure 2.1: constraint-based layout

Engineering problems, however, are more complex than these computer graphics problems. The diversity and number of constraints on a particular model can make it difficult for a human to find the optimal solution to a design problem. For example, the design of an airplane wing might be dependent on many factors: lift constraints, stress constraints, material cost, safety considerations, and so on. Computer-based constraint-satisfaction techniques have been forced to explore large search spaces, often without a good initial approximation to the final solution. The computing time necessary to apply these constraint-satisfaction techniques has made them unsuitable for interactive graphics applications.

As geometric models for computer graphics continue to become more intricate, techniques for automating the design process must become more sophisticated. For many applications it is no longer reasonable to expect interactive constraint satisfaction. Thus it is wise to examine not only previous work in interactive computer graphics but also the literature on constraint-based engineering design for possible insights to constraint-based geometric modeling.

2.1.2. Modeling vs. rendering

A second reason for the lack of references to the engineering work in the computer graphics literature is that many of the constraint-based architectural and engineering systems did not produce images on a display device. Instead, output consisted of tables of numeric data. The emphasis of the work was on the creation of models and not on computer-based rendering.

In contrast, most previous research in computer graphics has concentrated on drawing models rather than creating them. In recent years, however, interest in the modeling problem has increased, and computer scientists have turned to other fields for insight. At the same time, architects and engineers have increasingly adopted computer graphics techniques for visualizing their models. The distinction between what is computer graphics and what is engineering has become blurred as researchers from both fields share and expand upon their knowledge.

This dissertation does not distinguish between computer graphics modeling applications and engineering modeling applications. A modeling system will be considered a system that generates geometric descriptions of objects, regardless of whether images of these objects are rendered.

2.2. Taxonomy of techniques

The complexity of any constraint-satisfaction problem depends on several factors: the existence and accuracy of an initial guess, the size of the search space, and the complexity and number of constraints. The simplest problems have constraints that are directly satisfiable using analytical techniques. The most complex problems may require exhaustive exploration of all potential solutions. Between these two extremes are a variety of solution techniques tailored to the specifics of individual problems. The remainder of this chapter discusses previous work based on a taxonomy developed by Mitchell [Mitchell, 1977].

Solution methods can be categorized as being either strong or weak. Strong methods require specific information or impose requirements on the problem being solved. In exchange for these limitations, they can generate solutions very quickly. Weak methods place few restrictions on the problem formulation, and thus can be applied to a larger class of problems. Weak methods, of course, tend to be time consuming compared to strong methods.

The solution categories below are organized roughly in order from strongest to weakest. The categories discussed are: analytical techniques, optimization methods, heuristic search, improvement procedures, and generate-and-test. These categories provide only an approximate organization; many techniques could be classified in multiple categories, and many applications combine techniques from two or more categories.

2.3. Analytical techniques

Analytical techniques directly and efficiently solve constraint-satisfaction problems to yield optimal results. While most complex design problems cannot be solved analytically, such procedures can be used to find solutions to trivial problems or restricted versions of more general problems. A particular representation of constraint problems, the constraint graph, is useful in explaining analytical solution methods. It is described below, followed by a discussion of four analytical techniques: local propagation, propagation of degrees of freedom, graph transformation, and equation solving.

2.3.1. Constraint graphs

Constraint problems may be represented using graphs. Nodes in the graph represent objects and operators, while arcs represent bidirectional data flow paths. The operators define the relationships (constraints) between objects. The goal is to find values for undefined objects (variables). Constraint graphs are frequently used to represent constraints based on simple algebra; all constraints are implemented using the operators sum, product, and equality.

For example, the constraint graph below expresses the relationship between *x* and *y* coordinates of points on the line: *y=4x+3*.

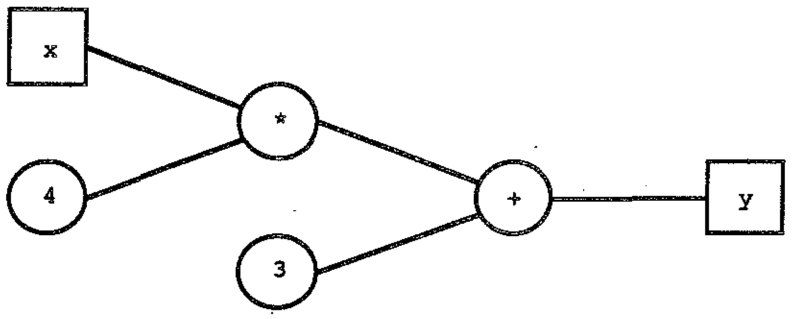


Figure 2.2: a simple constraint graph

Since information flow is bidirectional, a constraint solver can compute the value of *x* given *y*, or the value of *y* given *x*.

2.3.2. Local propagation

Local propagation, or propagation of known states, is a very simple technique for solving constraint systems. Because the technique is so simple (and fast), it should be used whenever applicable.

The basic strategy is to deduce locally any values that are computable, and propagate this information along the arcs of the graph so that additional values may be computed. This is equivalent to finding an ordering for the solution of individual constraints.

When an operator node has enough information to deduce unknown values, it can *fire,* and thus propagate computed values along its arcs. In the example above, suppose it is known that *y=11*. The addition node (+) can then fire since it knows the values of two of its arcs. The value *8* would then be propagated to the multiplication node (*), which would then fire to yield *x=2*.

This technique cannot solve for all values if cycles exist in the graph. Note that local propagation is insufficient to solve for x in the simple relation: *x+x=8*.

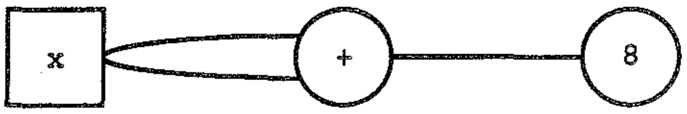


Figure 2.3: a constraint graph with a cycle

The problem is that the addition node (+) can never fire since it needs values from two arcs to produce output on a third. Local propagation can solve the equation: 2*x=8, but local propagation alone is not smart enough to recognize that the two equations are equivalent.

2.3.3. Propagation of degrees of freedom

A similar technique, propagation of degrees of freedom, examines the constraint graph for a variable with enough degrees of freedom so that it can be set to satisfy its constraints. When such a variable is identified (e.g. if the variable is controlled by a single constraint), the variable and the constraints associated with it may be removed from the constraint graph and saved for later evaluation.

When the remaining variables in the constraint graph are solved, the values of the saved variables may be deduced. This technique is efficient and can be used to reduce the size of the constraint graph should more complex techniques (e.g., relaxation [Sutherland, 1963]) be required to resolve cycles.

2.3.4. Graph transformation / term rewriting

As shown above, local propagation and propagation of degrees of freedom cannot solve all constraint systems represented by graphs. These techniques cannot break cycles in the graph because they examine only the arcs local to a node in deciding whether to propagate information from that node.

Graph transformation techniques attempt to solve constraint problems by examining regions of the graph and reducing these regions to simpler but equivalent graphs. One could define a rule to convert equations of the form $x+x=y$ to $2*x=y$. Such a rule would transform the example above, $x+x=8$, into a graph that is solvable using local propagation: $2*x=8$.

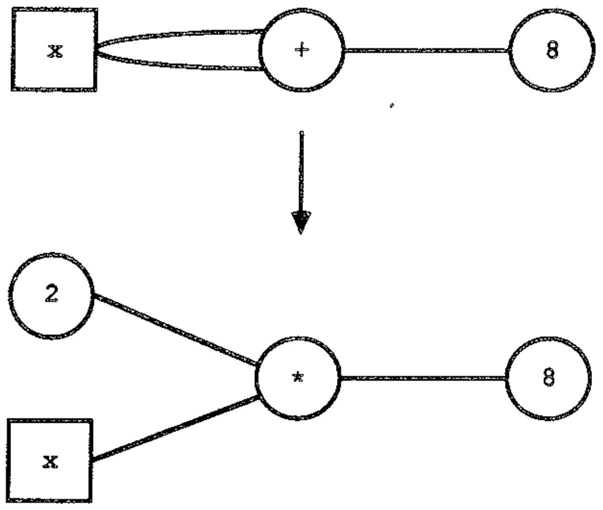


Figure 2.4: graph transformation

Although graph transformation techniques can break simple cycles in constraint graphs, complex graphs require even more powerful techniques, such as equation solving.

2.3.5. Equation solving

Constraints define equations relating the variables of the constraint system. If the constraints cannot be solved serially (because cycles in the constraint graph cannot be broken), then the problem can be treated as finding the solution to a set of simultaneous equations.

The general problem of solving sets of simultaneous equations is complex. Iterative numerical techniques (described in a later section) and symbolic techniques (e.g. Mathematica [Wolfram, 1988]) can be used but are time-consuming and therefore have not been used where interaction is a concern. When interaction has been a concern, the approach has been to restrict problem complexity by allowing only simple constraints.

For instance, one alternative is to impose the restriction that constraints must be linear. If constraints are linear then efficient equation solving techniques may be applied [Derman, 1984]. Derman's technique is similar to Gaussian elimination, though it can be extended to solve a combination of linear and nonlinear equations if the nonlinear equations reduce to linear equations after substituting variables computed by solving the original linear equations.

2.4. Modeling systems using analytical techniques

Sketchpad [Sutherland, 1963], Thinglab [Borning, 1979], and Magritte [Gosling, 1983] use local propagation and/or propagation of degrees of freedom in combination with slower, more powerful techniques. Ideal [Van Wyk, 1980] uses equation solving exclusively. Brüderlin [Brüderlin, 1986] uses a hybrid symbolic/numerical approach. Rossignac [Rossignac86] implements a user-defined constraint ordering to derive an analytical constraint solution. Bertrand [Leler, 1987] uses augmented term rewriting in a general purpose constraint language builder. Mathematica [Wolfram, 1988] is, among other things, a very general equation solving system. CBD [Ervin, 1990] is a knowledge-based approach similar to Brüderlin's system. Sketchpad and Thinglab are discussed in a later section.

2.4.1. Van Wyk (1980)

Ideal is a language for defining graphical layouts. Ideal permits hierarchical object definitions, with simple constraints defining the relations of parts of the objects. When an object is instantiated, the caller must provide enough information to solve the constraints in the object definition.

Constraints in Ideal must be reducible to linear equations. Ideal uses a fast equation solver to solve the constraints, although it was not designed for interactive use.

2.4.2. Gosling (1983)

Magritte is an interactive graphical layout system. Magritte uses local propagation in combination with graph transformation techniques to solve constraint graphs. Gosling notes that while these techniques are sufficient in Magritte, a more general system might still have to resort to relaxation to resolve cycles. Magritte is not as general as Thinglab (see below), but it is more efficient at solving problems in its limited domain.

2.4.3. Brüderlin (1986)

Brüderlin describes an approach in which constraints are solved using a hybrid of declarative and procedural techniques. Constraints are first solved symbolically in a portion of the system written in Prolog and then evaluated numerically in another portion of the system written in Modula-2. The system contains a set of geometric rules which are used to generate a symbolic solution for the object being evaluated.

Prolog permits backtracking, hence Brüderlin's system has some characteristics of a heuristic search procedure. The rules, however, are not general rules of thumb, but are instead carefully derived rules proven to terminate with a correct solution given proper input. Brüderlin's system is similar to term rewriting because rules are applied to transform the list of supplied predicates into a solution numerically solvable in Modula-2.

2.4.4. Rossignac (1986)

Rossignac's CSG system allows the user to describe models in terms of unevaluated constraints. A model is constructed by evaluating constraints sequentially in a user-specified order. A constraint is evaluated by performing a rigid body motion (i.e., translation or rotation) on an object so that an adjacency relationship between two objects is met.

Rossignac's system is not a problem-solving system that considers simultaneous constraints to reach a solution; instead, it uses constraints to simplify the description of a model. The disadvantages of the system are that the user is partly responsible for constraint-satisfaction and that cyclic constraints may not be specified or solved. In exchange for these limitations, the constraints may be solved algorithmically.

2.4.5. Leler (1987)

Leler's Bertrand is a language that can be used to build constraint satisfaction systems. Leler uses augmented term rewriting to solve constraint programs in Bertrand. Augmented term rewriting extends term rewriting by allowing binding of values to variables and the capability of defining abstract data types. He shows that augmented term rewriting can be used to implement an extended version of an equation solver similar to the one used in Ideal [Derman, 1984]. The extended equation solver can handle nonlinear equations if transformation rules (e.g. cross-multiplication) are provided for transforming the nonlinear equations to solvable linear equations.

Leler's augmented term rewriting approach has several advantages over other constraint languages. Bertrand allows new data types and constraints to be implemented. Bertrand can handle underconstrained systems by generating an expression for further processing by a human or computer. Bertrand can be used to implement both 2-D and 3-D graphical constraint languages. Finally, Bertrand is simple and efficient. Its performance makes it suitable for interactive applications.

2.4.6. Wolfram (1988)

The Mathematica system is a sophisticated tool that can be used to represent and solve constraint problems. Mathematica is many things: an equation solver, a programming language, and a knowledge representation system. Mathematica has an extensive collection of built-in transformation rules that can be used to solve equations, including systems of simultaneous equations. In addition to these built-in rules, the user may define additional rules that specify how equations may be transformed. Once these rules are defined, collections of equations representing constraints may be entered. The built-in and user defined rules are then used to find solutions to arbitrary variables. Results are expressed either numerically or symbolically. Complicated nonlinear relations between variables may be specified, although the solution method cannot always find a solution for one variable in terms of the others.

2.4.7. Ervin (1990)

CBD (Constraint Based Diagrammer) is a system that was built to explore ideas about designing with constraints and diagrams. The system consists of a rule base expressed in LISP, and a graphics module for drawing shapes. The system solves design problems by using the rule base to convert relations expressed in LISP into a graphical representation. This approach is similar to the approach used by Brüderlin.

2.5. Summary of analytical techniques

Analytical techniques can be used to solve simple constraint problems efficiently. Local propagation is a basic technique for solving constraint systems without cycles. Propagation of degrees of freedom is another technique for ordering constraint evaluation; it can be used to reduce the size of a constraint graph containing cycles. Graph transformation can eliminate simple cycles, but is not powerful enough to eliminate all cycles. Equation solving is a general technique for handling arbitrary cycles, such as those defined by sets of simultaneous equations. Leler [Leler, 1987] describes these analytical techniques in greater detail than presented here.

The simplest techniques are efficient because they solve constraints locally. Local propagation, for example, looks at only one constraint at a time during the constraint satisfaction process. Once the constraint is solved, the resulting values may be used to solve other constraints. This serial solution method is efficient in any case, but is particularly efficient when incremental changes are made to a system, since the effects of a change need be computed only for the part of the system affected.

If at all possible, a constraint graph should be reduced to a form solvable by local propagation. Propagation of degrees of freedom reduces the graph by removing subgraphs that are known to be solvable once its inputs are known. Graph transformations convert an unsolvable subgraph into an equivalent, solvable graph.

If a constraint graph representing a problem cannot be solved serially, even after reductions and transformations have been applied, then the constraints need to be solved using global solution techniques. Such techniques consider all variables and all constraints when computing a solution, and thus are more complex than local techniques, which only consider one constraint at a time. Since constraints can be expressed as equations, the global constraint satisfaction procedure requires finding the solution to a set of simultaneous equations.

If the equations in the system of constraints are linear, then known algorithmic techniques such as Gaussian elimination may be applied. If some constraints are not linear, then there are two alternatives. If the nonlinear constraints fall into a certain class of equations, then one approach is to extend the analytic constraint solver to handle this class in addition to linear constraints. The second alternative is to resort to weaker but more general techniques (e.g., heuristic search, improvement procedures, and generate-and-test) as described later in this chapter.

Analytical techniques described in this section have been used extensively in 2-D layout problems. Unfortunately, many other design problems involve constraints which are more complex than the simple constraints in 2-D layout. If the constraint system cannot be solved by the strong methods described in this section, then weaker, but more general, techniques must be used. The following sections describe some of the alternatives.

2.6. Optimization methods

The term *optimization methods* is potentially confusing, since many constraint satisfaction methods treat the constraint satisfaction problem as an optimization problem. The usual approach is to express constraints in terms of scalar error functions. The goal of the optimization is then to minimize the sum of these error functions. Iterative techniques are frequently used; in general these techniques do not guarantee an optimum, but usually they yield near-optimal results.

This section discusses two particular optimization methods which very efficiently and reliably yield optimal or near optimal results: linear and nonlinear programming. These methods are stronger than improvement methods, to be discussed later, but they can only be applied to a restricted class of problems.

2.6.1. Linear programming

Linear programming techniques can be used when the constraint problem can be expressed as the minimization (or maximization) of a linear objective function subject to linear constraints on the variables. These conditions are very restrictive; even a simple two-dimensional area constraint, such as *width*height<50*, cannot be handled by linear programming techniques. Consequently, linear programming is poorly suited for geometric modeling problems.

2.6.2. Nonlinear programming

Nonlinear programming methods have been used by several architectural floorplan layout systems (e.g., [Mitchell, 1975], [McGovern, 1976]). The main problem with nonlinear programming methods is that they cannot solve arbitrary nonlinear constraints. Suppose an algorithm can solve systems of equations consisting of linear equations and equations with terms involving the product of two variables. If an application has equations with terms involving the cube of a variable, then the solution method cannot be used, even though the constraint solver can handle some nonlinear equations.

2.7. Summary of optimization methods

Linear and nonlinear programming methods solve a narrow class of problems. Some of these techniques are analytic (e.g., the simplex method for linear programming), while others (e.g., Newton's method for nonlinear programming) are iterative numerical techniques. As a class these optimization methods can still be considered strong methods, although they may not be as efficient as the analytic techniques mentioned in the previous section. They yield excellent solutions for problems involving simple constraints and objectives. Because these techniques are reliable and efficient, it can be worthwhile to convert constraint problems by approximating more complex constraints with simpler constraints when possible.

In the search for a solution method, one should consider optimization techniques as the next alternative if simple analytic techniques cannot be found. If optimization techniques are not applicable, then weaker methods may be examined. The constraints involved in three-dimensional geometric modeling, however, are sufficiently complex that neither simple analytic techniques nor optimization techniques can solve all problems of interest. On the other hand, heuristic search, improvement methods, and generate-and-test schemes are capable of finding solutions to some problems that are not solvable analytically. These categories are explored below.

2.8. Heuristic search

Heuristic search procedures use knowledge about the design problem to guide the search for an acceptable solution. Heuristics are applicable when the problem solving process can be viewed as a tree (or graph) of states. The terminal nodes in the tree are potential solutions; internal nodes represent intermediate states leading to a solution. The goal is to find a path in the tree (a series of actions) that leads to the best solution.

The simplest forms of heuristic search rules are the general guidelines on how to search the tree (e.g., depth-first or breadth-first search). More sophisticated systems encode detailed domain-specific problem-solving techniques. Many expert systems for solving design problems have been developed, including the three representative systems discussed below.

2.8.1. Pfefferkorn (1971)

The Design Problem Solver (DPS) [Pfefferkorn, 1971] solved architectural floor plan layout problems incrementally. The system entered new components into a room serially. If a new element could not be entered without violating constraints, a special procedure was called to resolve the conflict. If the conflict could not be resolved, the system incorporated backtrack to restart the design at an earlier stage. Optimal solutions were not guaranteed, but the system had the flexibility to explore a range of design alternatives.

2.8.2. McDermott (1982)

The XCON/R1 expert system [McDermott, 1982] automatically generates VAX computer configurations. It uses a rule-based approach without backtracking. The constraints in R1 are highly domain dependent. The system is very powerful, but not well-suited to exploring a range of designs. The large number of specialized constraints in R1 makes it more difficult to maintain than DPS.

2.8.3. Brown (1986)

Brown [Brown, 1986] describes an expert system that closely matches human design problem solving techniques. Rather than using a single rule base and inference engine, Brown uses a collection of communicating design specialists. Brown organizes the specialists into a design hierarchy. Specialists at the top of the hierarchy call lower-level specialists to make detailed design decisions. When a subproblem is solved, the information is passed back to the higher levels. Each specialist maintains local design knowledge. The purpose of this information passing is to reach a globally optimal design while using specialists capable of solving local problems.

2.9. Summary of heuristic search

Heuristic search techniques can be very successful at simulating the human problem solving process, and can do so efficiently. Unfortunately, knowledge-based systems tend to be domain dependent and difficult to develop and maintain. Moreover, human design expertise is not available for solving all classes of problems.

If no analytic techniques are capable of solving a constraint system, and heuristic search methods are not applicable, then one may have to accept weaker techniques for finding a solution to the problem. Two classes of weak solution procedures, improvement procedures and generate-and-test, are discussed below.

2.10. Improvement procedures

Improvement procedures generate new potential solutions from previous configurations. Whereas heuristic methods use knowledge about the problem in seeking a solution, improvement methods typically require no specialized knowledge about the problem being solved, and thus can be applied to a wide variety of problems. Three categories of improvement procedures are discussed below: variation with selective retention, greatest improvement, and numerical methods.

2.10.1. Variation with selective retention

Variation with selective retention requires the ability to compare two configurations and determine which better meets the design criteria. New configurations are generated from existing configurations, and a decision is made regarding whether the new configuration should be retained. Traditionally the better configuration is retained (simple hill-climbing), but more sophisticated retention algorithms (e.g., simulated annealing) have been developed to prevent the method from getting trapped in local optima.

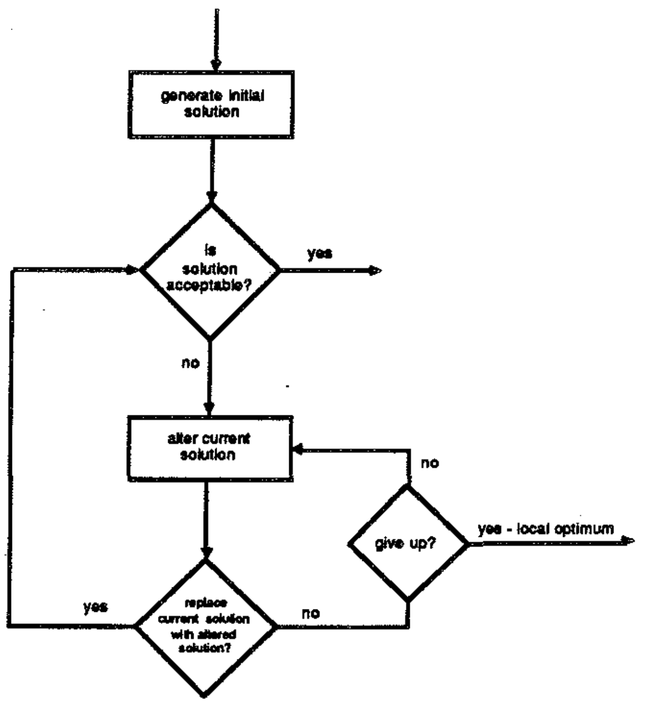
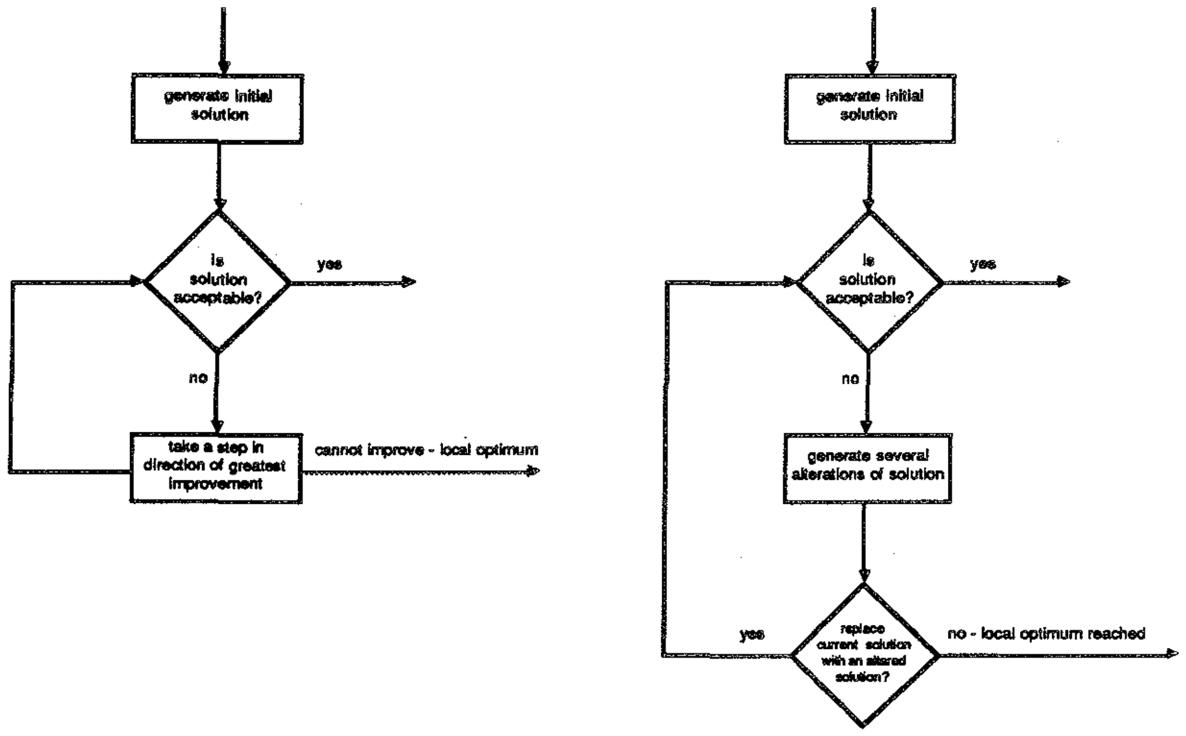


Figure 2.5: variation with selective retention

2.10.2. Greatest Improvement

Greatest improvement procedures have been used to speed convergence to a solution. The strategy is to alter the configuration in a way that yields the greatest advance towards the objective. This can be achieved by analytically computing the direction of greatest improvement, or by evaluating several possible perturbations and choosing the one which yields the greatest improvement in the objective function.



generate initial solution
is solution acceptable?
yes
no
take a step in direction of greatest improvement
cannot improve - local optimum
generate initial solution
is solution acceptable?
yes
no
generate several alterations of solution
yes
replace current solution with an altered solution?
no - local optimum reached

Figure 2.6: greatest improvement procedures

2.10.3. Numerical methods

This category encompasses a variety of numerical techniques that iteratively approach a final solution to a constraint system. Relaxation is one such technique used by several constraint solvers. Relaxation methods compute an estimate of the cost of making specific assignments to variables. The value of variables for the next iteration are chosen so as to minimize the total cost of the system. The iterative process continues until the rate of change in cost falls below some threshold. Several techniques, such as Newton's method and its variations, can be considered optimization techniques as well as numerical methods.

Since these techniques tend to be weaker than analytic optimization techniques, the systems using these methods are described below rather than in the optimization section above.

2.11. Systems using improvement procedures

Sketchpad [Sutherland, 1963] and ThingLab [Borning, 1979] use relaxation to solve constraint graphs involving cycles. Weinzapfel and Handel [Weinzapfel, 1975] use iterative methods in an architectural layout system. Cinar [Cinar, 1975] uses a form of greatest improvement in a building planning system. Nelson [Nelson, 1985] uses Newton-Raphson iteration in a 2-D imaging system. Many applications (e.g., [Kravitz, 1986], [Pincus, 1986], [Romeo, 1985], [Sechen, 1986], [Wong, 1986]) use simulated annealing to search for optimal VLSI layouts. Witkin, Fleischer, and Barr [Witkin, 1987] use iterative techniques to solve constraints expressed as energy functions. Barzel and Barr [Barzel, 1988] invoke forces on objects to satisfy constraints.

2.11.1. Sutherland (1963)

Sketchpad was an interactive 2-D system in which the user defined pictures by combining various graphic primitives (e.g., points, lines, and circles). The user sketched a rough version of a drawing to which constraints were applied. Sketchpad also provided support for instancing copies of a previously defined object.

Sketchpad used two techniques to solve constraints. It first attempted to apply propagation of degrees of freedom (see analytical techniques). If propagation failed, Sketchpad resorted to relaxation.

In Sketchpad's relaxation method, every constraint generated an error expression. At each iteration, the variables in the system were adjusted to reduce the total error. Relaxation terminated when all constraints were satisfied (zero error), or when further iterations could not reduce the error. The starting point for the relaxation procedure was a rough sketch entered by the user.

2.11.2. Weinzapfel and Handel (1975)

IMAGE is an assistant for architectural layout. It was designed to aid architects in site planning and floorplan layout problems. Image allows the architect to specify a variety of objectives including distance, area, adjacency, position, ratio, and visual access constraints. IMAGE has two methods for solving constraints: an automated constraint satisfaction procedure, and a user-guided exploration procedure.

2.11.2.1. Automated constraint satisfaction

The automated procedure starts with an initial configuration provided by the architect. It proceeds by modifying one object at a time. For each object that it is modifying, it determines which constraints are being violated, and what changes should be made to satisfy that particular constraint. Since the changes suggested by a particular constraint may conflict with those suggested by another, a least mean squares fit is applied to find the compromise that will generate the least error among all violated constraints.

After a particular object has been modified, the remaining objects are considered in succession. When all objects have been modified, the system returns to the first object and continues the improvement procedure. The procedure terminates when no further improvement can be made.

This procedure is strongly influenced by the initial configuration of objects. It performs local optimization based on the supplied configuration; it cannot search out a global optimum without a close initial guess. The procedure is also influenced by the order in which objects are moved. The architect can choose to move objects either in the order they were entered or in an order based on how seriously the objects violate constraints. Despite the somewhat ad hoc nature of the solution procedure, the system has been used to find near-optimal solutions to problems involving over 50 objects.

2.11.2.2. Interactive constraint satisfaction

IMAGE was designed to be an assistant to (rather than a replacement of) the architect. Consequently, the system allows the human to modify computer generated layouts. The architect can steer the satisfaction procedure toward an anticipated solution. In addition, IMAGE can be instructed to evaluate a library of prototype solutions to a problem. This combination of user interaction and automated evaluation allows the architect to explore a variety of potential designs.

2.11.3. Cinar (1975)

CRAFT-3D is a system for facilities planning in multi-story buildings. It attempts to minimize the transportation costs of generated facility layouts. The cost of any configuration is based on the distance between facilities and the expected flows between each pair of facilities.

CRAFT-3D uses a simple form of greatest improvement. At each stage of iteration, the system evaluates the result of all possible component swappings of two or three facilities. The swap that results in the greatest reduction in layout cost is accepted. The iteration terminates when no further reduction in cost can be found. In general, this heuristic generates suboptimal results. Moreover, it is impossible to tell how close one is to the optimal solution. Nevertheless, the CRAFT-3D system appears to be useful because of the lack of an analytical solution to the layout problem it addresses.

2.11.4. Borning (1979)

ThingLab applied and extended some of Sketchpad's ideas to a more general environment. Whereas Sketchpad was designed for the creation of 2-D geometric figures, ThingLab is designed as a simulation laboratory. Constraints can be defined involving both geometric and non-geometric objects, making ThingLab suitable for diverse applications such as simulation of electrical circuits and mechanical stress.

ThingLab uses local propagation and propagation of degrees of freedom when possible; relaxation is used to deal with cycles. ThingLab is implemented in Smalltalk, and provides the capability of defining new constraints and new object types using Smalltalk classes. When defining a new constraint, the user can specify explicit procedures which, when executed, will satisfy the constraint.

ThingLab continues to be enhanced by Borning and his colleagues. Recent enhancements include interactive constraint graph editing [Borning, 1985]. Duisberg [Duisberg, 1986] extended ThingLab to handle constraints involving time.

2.11.5. Nelson (1985)

Juno is a constraint-based system for image creation. Images are defined using a language that allows the user to specify constraints on points. Only four primitive constraints are allowed, but these are general enough to implement more complex constraints. The primitive constraints are: parallelism of pairs of lines, horizontal lines, vertical lines, and pairs of lines constrained to be of equal length. The user can modify a Juno program either implicitly by manipulating objects on the display screen, or explicitly by editing the underlying text.

Juno uses Newton-Raphson iteration to solve the systems of equations defined by Juno programs. Nelson reports that although Newton-Raphson iteration is faster than relaxation, a powerful workstation is necessary for acceptable performance. The computational demands on the system can be lessened by defining objects hierarchically and by providing close initial guesses to the final solution. An additional justification for providing a close guess is that the nonlinear equation solver may behave unpredictably without a good initial configuration.

2.11.6. VLSI Layout: Kravitz (1986), Pincus (1986), etc.

Many applications in VLSI layout (e.g., [Kravitz, 1986], [Pincus, 1986], [Romeo, 1985], [Sechen, 1986], [Wong, 1986]) have used simulated annealing, a form of iterative improvement. Constraints on a problem are represented by summing error functions so that a single scalar expresses the quality of any solution. Simulated annealing works by perturbing object parameters (usually position) to minimize the objective function.

Simulated annealing has been very successful in generating near-optimal results. It is well-suited for problems where there is not much insight into solution techniques. A disadvantage of simulated annealing is that it can be time consuming because it uses stochastic variations which may not provide much improvement at each iteration. Simulated annealing is described in greater detail in chapter four.

2.11.7. Witkin, Fleischer, Barr (1987)

Witkin, Fleischer, and Barr describe a constraint-based modeling and animation system. Constraints are defined using non-negative energy functions, which evaluate to zero when the constraint is satisfied. The system attempts to minimize the sum of the individual energy constraints.

Numerical techniques are used to follow the energy gradient to a stable configuration. In addition to rigid body motions, objects can vary internal parameters to meet constraints. Thus an object might stretch or twist itself to satisfy a constraint. A disadvantage of the system is that the energy minimization procedure may get trapped in local optima. If this occurs, user intervention is necessary to recognize the problem and bump parts of the model out of the local minimum.

2.11.8. Barzel and Barr (1988)

Barzel and Barr have pioneered research in the area of physically-based modeling. Physically-based models respond to forces and torques in accordance with the rules of Newtonian physics. Their method, called *dynamic constraints*, converts constraints into forces which act upon objects in the model, causing the objects to move into positions which satisfy the constraints on the assembly. The problem of finding the forces necessary to satisfy constraints is known as an inverse dynamics problem.

Their system solves the inverse dynamics problem iteratively. At each iteration, the forces necessary to solve a constraint are computed. This series of iterations can be rendered to form an animation of the constraint satisfaction process. Where possible, constraint forces maintain previous constraint satisfaction as new constraints or forces are applied to an existing model. For example, a pair of objects with a connectivity constraint will remain connected even as external forces are applied.

2.12. Summary of improvement procedures

Improvement procedures have been used extensively in constraint-based modeling systems. While these techniques can be time-consuming, they are more efficient than generate-and-test methods (see next section). Many improvement methods suffer from the problem of getting stuck in local optima. This is not a drawback in situations where good approximations to the final solution are provided. In applications where global or near-global optima have been desired, simulated annealing has provided satisfactory results.

Improvement procedures are so general that they can be applied to many problems. One limitation of such methods, however, is that it may be difficult to control what portions of the solution space are examined, resulting in unsatisfactory local optima. An orderly exploration of the solution space is assured by generate-and-test methods, described in the following section.

2.13. Generate-and-test

Generate-and-test procedures attempt to solve a problem by generating and testing potential solutions until a satisfactory solution is discovered. The methods described in this section are the weakest methods presented in this chapter, but are generally applicable, and, given enough time, can yield globally optimal results. Two common forms of this method are exhaustive generate-and-test and random generate-and-test.

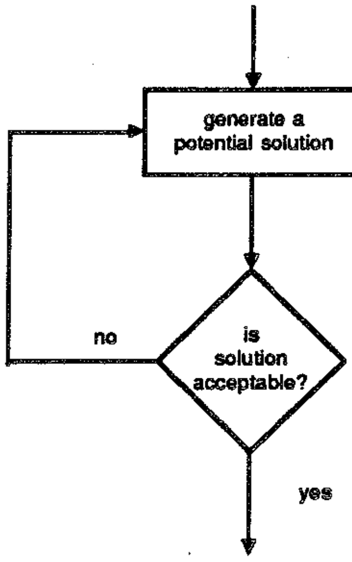


Figure 2.7: generate-and-test

2.13.1. Exhaustive generate-and-test

Exhaustive generate-and-test simply enumerates and tests all potential solutions to a problem. This technique can be used when the solution space is very small, or when there are many feasible solutions to the problem and the solutions are evenly spaced throughout the search space. It is not frequently used for design problems because of the large number of potential configurations in a typical problem. It has been used for some simple architectural floor plan problems [Mitchell, 1976].

2.13.2. Random generate-and-test

Random generate-and-test proceeds by sampling potential solutions from the solution space. It can be efficient when there are many feasible solutions, or when near-optimal results are acceptable. The ALDEP architectural floor plan layout system [Seehof, 1967] generated random layouts using simple rules; layouts that met most design constraints were selected for further processing by an improvement procedure.

2.14. Summary of generate-and-test

Both exhaustive and random generate-and-test procedures are weak but general methods for finding solutions that meet design constraints. These brute force techniques may be applicable in specialized problems where the design space is small. Generate-and-test does not appear to be well suited for 3-D (as opposed to 2-D) design involving many variables because the inclusion of the third dimension increases the size of the search space dramatically.

2.15. Applicability of constraint satisfaction techniques

This research is concerned with constraint-based three-dimensional modeling. The constraint solver must deal with diverse and complex constraints. Analytical techniques cannot be applied exclusively because the problem formulation is too complex; no analytical solutions are known. The constraints are not linear, nor do they fall into the class of nonlinear constraints solvable by nonlinear programming. Conventional optimization methods therefore are not applicable. Generate-and-test methods are infeasible because of the tremendously large solution spaces involved. This leaves two candidate solution categories: improvement procedures and heuristic search.

Both of these categories are worthy of investigation. This dissertation describes my research into improvement procedures; a fellow Ph.D. student [Amburn, 1991] is investigating the application of expert systems to geometric modeling. It is likely that future modeling systems will combine aspects of several approaches. For example, a reasonable strategy would be to solve trivial tasks analytically, then use human problem solving heuristics to further simplify the problem, and finally use improvement procedures to complete the constraint satisfaction problem.

Chapter 3

Constraint-Based Design

Design is the process of selecting among alternatives so as to best satisfy a set of potentially conflicting goals. Constraint-based modeling involves constructing geometric descriptions with the aid of constraints; constraints are used as an efficient means of expressing a human's preconceived model. Constraint-based design, however, involves more than just using constraints as a communication tool. It involves using constraints to specify unsolved design problems — problems which the human designer may not be able to solve.

Traditionally, the assumption has been that the human (as the design expert) should be responsible for solving hard design problems, and that constraints should be used to help solve trivial problems so the human may concentrate on the important issues and hence make productive use of his time. In other words, people have been responsible for making global modeling decisions, and constraints have allowed local decisions to be solved automatically.

This research expands the role of the computer by using it to make global modeling decisions. The goal is not to replace the human, but rather to automate more of the design problem. This chapter begins by expanding upon the role of the computer in this work. Second, an explanation of what makes design problems hard is presented. Third, the specific objectives of this work are presented to distinguish it from other constraint-based systems. Next, the solution methods explored in the course of this research are presented. The logical progression of this investigation toward the cost function method is described. The cost function approach is then detailed. The chapter concludes with characteristics of applications with which CONTEST is compatible.

3.1. The role of the computer

Although our objective is to enable the computer to make global modeling decisions, the purpose of this work is not to replace the human designer. The years of training a designer receives define thousands of constraints, opinions, prejudices, and objectives that contribute to the decision-making process. Quantifying these constraints is theoretically possible but impractical. While objective constraints are readily quantifiable, subjective constraints are by definition more open to interpretation and therefore difficult to quantify.

Even in problems that can be completely described by quantifiable constraints, the designer may still need to be involved in the design process. Design is a learning process, and constraints may need to be modified as a designer learns of their effect. The computer is thus best viewed as a design aid.

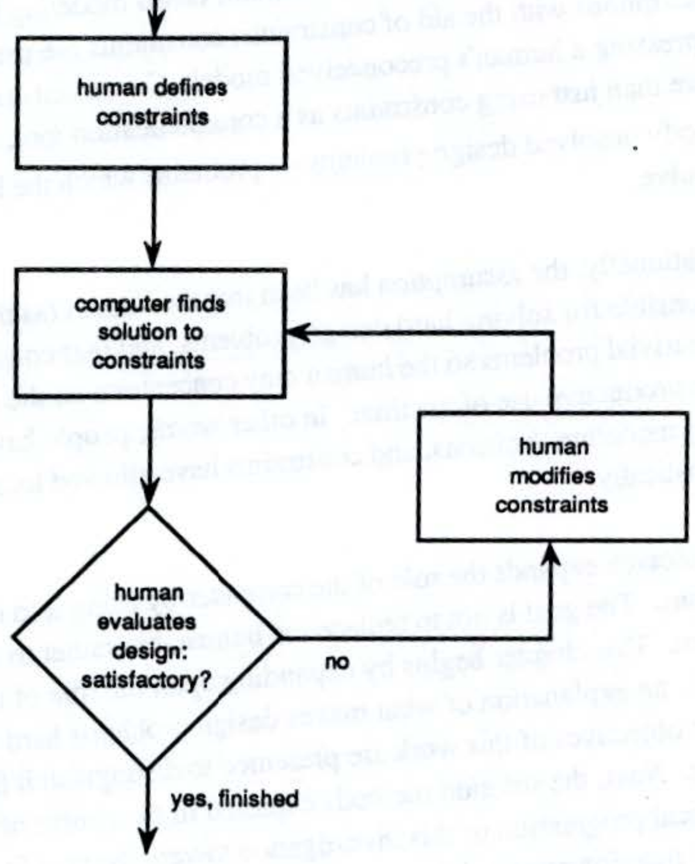

human defines constraints
computer finds solution to constraints
human modifies constraints
human evaluates design: satisfactory?
no
yes, finished

Figure 3.1: semi-automatic design of a single model

The computer can be used as a design aid in three main ways. First, designs generated by the computer can be used to stimulate the designer's imagination. He can examine computer-generated solutions as starting points for non-intuitive solutions. Second, the computer can evaluate human-generated designs in addition to computer-generated designs. One can thus compare computer-generated designs with manual designs, or make changes to a model and see how constraint satisfaction is affected. Third, and most frequently, the designer can use the modeling system to refine a problem specification. By examining the effect of different constraints and weights on generated designs, the designer can gain insight to the problem. He still participates in the design process by creating and tuning new constraints, and subjectively evaluating resulting solutions. This is not totally automatic design because the designer is involved in the *evaluation* portion of the design loop. This style of design is called *semi-automatic* design. For completely automatic design, one would have to specify all subjective constraints as cost functions and permit the computer to perform all design evaluation.

3.2. Why are design problems difficult?

Design problems present both practical and theoretical challenges. For humans, it is difficult to evaluate properly the interaction of variables. Since constraints can involve more than one design variable, in general it is not possible to find the optimal value for parameters sequentially. Instead, a large system of constrained variables must be considered as a whole. Patrick Winston [Winston, 1984] describes the optimization problem using a television set analogy. The goal is to maximize overall picture quality. If a single parameter (e.g., the tuner) controls the picture, then we can usually find the best picture. However, if multiple parameters (e.g., tint, color, brightness, tuning, contrast) define picture quality then, as Winston points out, there is likely to be more cursing than entertainment.

One solution to complex interactions of variables is to avoid understanding these interactions at all, and simply enumerate all solutions. The tedium of generating and testing solutions makes it impractical for people to use this technique for problems involving more than two or three variables. Even with the use of a computer, the exponential growth in number of configurations makes it difficult to enumerate solutions in problems of more than four or five dimensions. Thus, for both humans and computers, design problems are difficult because they involve large solution spaces.

Finally, at a theoretical level, many optimization problems are NP-complete. No polynomial-time algorithms exist for such problems, and no transformation of the problem representation will yield such an algorithm.

3.3. Specific objectives of this research

The main objective of this research was to devise a semi-automatic method of design that allows human modelers to describe assemblies using a diverse set of constraints. The primary task is to take a collection of model parameters, along with a set of constraints on their values, and use this information to construct a model satisfying the constraints. CONTEST differs from other geometric modeling systems in that it provides greater flexibility in constraint specification. The system was designed to fulfill the following requirements:

- definition of geometric and non-geometric constraints
- definition of arbitrarily complex constraints
- separation of problem specification and problem solution
- capability of handling underconstrained and overconstrained problems
- suitable for use as a design aid

These requirements affect the class of problems that may be specified and the potential solution techniques that may be applied.

3.3.1. Geometric and non-geometric constraints

Most constraint-based modeling systems deal exclusively with geometric constraints. In contrast, CONTEST can also handle constraints that relate only indirectly to geometry, as well as completely non-geometric constraints.

A constraint on the distance between two objects is a simple geometric constraint. A constraint to minimize the internal operating temperature of a part influences the shape of that part, yet is indirectly specified in terms of geometry. Finally, a constraint on the color of an object is strictly non-geometric.

3.3.2. Arbitrarily complex constraints

As noted in chapter two, some systems place restrictions on the complexity of constraints that may be specified. For instance, a system might require that design variables meet simple linear relationships. CONTEST does not place such restrictions on constraints; any constraint that may be evaluated by a procedure call can be used.

A system without restrictions on constraint complexity can be used to solve a wide variety of modeling problems, assuming an appropriate solution method exists. Unfortunately, permitting arbitrary constraints limits the range and success of potential solution techniques. It becomes trivial to define optimization problems that are NP-complete. The traveling salesman problem, for example, is a geometric problem involving a simple goal.

3.3.3. Separation of specification from solution

If the solution method is dependent on details of the particular constraints being used, then extending the system to incorporate new constraints also may require that the solution method be revised. The constraint specification method in CONTEST is independent from the solution method. Thus, the user may define new constraints without worrying about the way they are solved. Moreover, the solution method can be improved without affecting constraint representation.

3.3.4. Dealing with overconstrained and underconstrained systems

Some systems require that a constraint problem be exactly constrained. CONTEST allows both overconstrained and underconstrained problems. If a problem is overconstrained, CONTEST seeks the best compromise to satisfy conflicting constraints. If a problem is underconstrained, CONTEST simply picks a plausible solution that meets the given constraints. Most design problems include overconstrained design variables; the difficulty in a design problem is finding a design that best satisfies a set of conflicting criteria. Note that some design variables in a problem may be overconstrained, while others are underconstrained (and still others may be exactly constrained).

3.3.5. Suitability as a design aid

As explained above, CONTEST was not constructed to replace the human designer, though in some cases designs may be generated completely automatically from an initial problem specification. In many other cases, however, design will remain an iterative process, with the computer and human interacting to explore the design space. CONTEST allows a designer to add and modify constraints to evaluate prototype designs.

3.4. Exploration of solution methods

When we began this project, we had some vague ideas of how we might solve problems involving collections of complex constraints. We examined four approaches that were in some way unsatisfactory. One of our early ideas was to build intelligent objects, each of which was capable of determining its correct position and shape, given the list of constraints imposed on it. The problem with this approach was that each object needed to know about other objects and needed to know about the constraints on themselves. In essence a separate constraint solver was needed by each object. Apart from the obvious problem of implementing constraint solvers for each object, the approach suffered from the additional problem of solutions being local in nature. I.e., while each object could attempt to solve the constraints on itself locally, it did not have information about all constraints on other objects, and hence could not in general reach a globally optimal solution to the constraints.

Despite these limitations, we implemented a system which solved mutual constraints by allowing communication between neighboring objects [Amburn, 1986]. This *ad hoc* approach was successful in a special purpose terrain modeling application, but broke down in the general case because it required a global constraint solver, which we did not have and were trying avoid in the first place.

The second idea was to find a way to reduce the problem to a simpler problem solvable using analytical techniques. We tried to find a small set of geometric constraints that could be used to implement many complex constraints. We hoped that a large class of geometric constraints could be expressed using simple constraints involving distances and orientations between points, lines, and polygons. Barzel and Barr [Barzel, 1988] and Brüderlin [Brüderlin, 1987] have shown that sophisticated modeling problems can be solved using small sets of such constraints, but our goal was not to see what could be done with a given set of constraints, but to see whether arbitrary constraints could be converted into simpler constraints. This approach was rejected because no general solvable set of primitive constraints could be found. While many constraints could be expressed as a collection of simpler constraints, the resulting set of simpler constraints still was not solvable using analytical techniques.

A third approach considered was approximating all constraints by simpler constraints solvable using analytical techniques. If all constraints could be approximated by linear functions, for example, then a technique such as linear programming could be applied. This approach was rejected because basic three-dimensional properties such as area, volume, and distance are non-linear in terms of the defining coordinates. We may have proceeded if only a small number of exceptions needed to be approximated, but it did not seem wise to proceed with such a distinct clash between problem definition and potential solution methods.

A fourth idea was to evaluate constraints using a scalar quality measure. This was eventually adopted, though it was temporarily rejected because there seemed to be no practical way to use the information to find a solution. The feasibility of this approach was reconsidered after examining constraint-based specification in a related field, VLSI design.

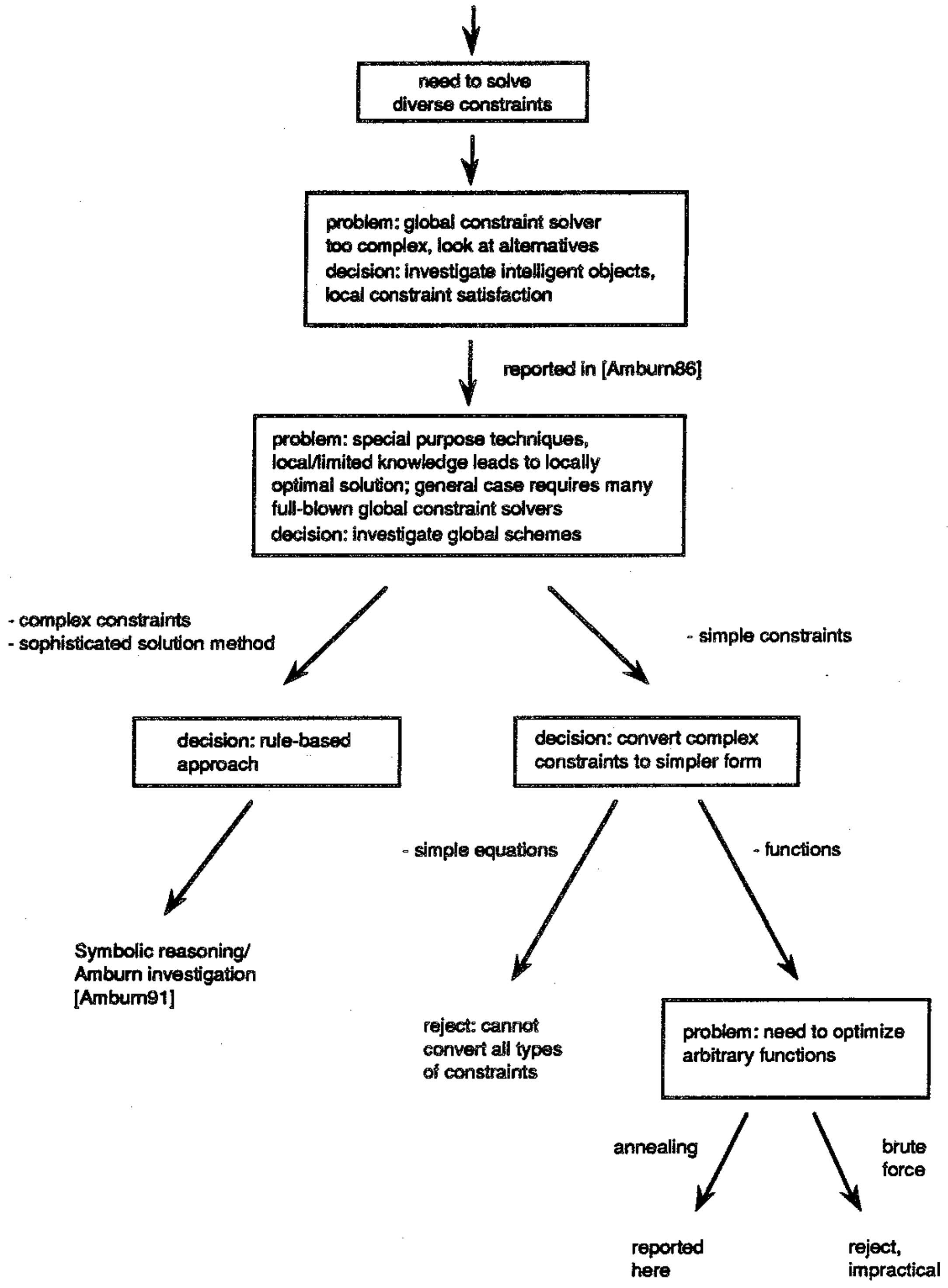

need to solve
diverse constraints
problem: global constraint solver too complex, look at alternatives
decision: investigate intelligent objects, local constraint satisfaction
reported in [Amburn86]
problem: special purpose techniques, local/limited knowledge leads to locally optimal solution; general case requires many full-blown global constraint solvers
decision: investigate global schemes
- complex constraints
- sophisticated solution method
- simple constraints
decision: rule-based approach
decision: convert complex constraints to simpler form
- simple equations
- functions
Symbolic reasoning/ Amburn investigation [Amburn91]
reject: cannot convert all types of constraints
problem: need to optimize arbitrary functions
annealing
brute force
reported here
reject, impractical

Figure 3.2: progression of investigation

3.5. The cost function representation

The problem representation used by CONTEST was chosen after noting the success of constraint-based VLSI systems. VLSI designers must constantly deal with constraints: basic constraints defining the function of the circuit, design rule constraints imposed by the manufacturing process, and cost and speed constraints involving wire length and chip area. One of the fundamental problems in VLSI design is floorplan layout; the goal is to place a number of cells so as to minimize chip area. By explicitly casting the problem in terms of optimizing the area function, VLSI designers were able to apply iterative improvement techniques to generate near-optimal floorplans (e.g., [Wong, 1986]). The design variables were the positions of the cells, and the objective function was chosen to minimize the area of the resulting floorplan, while also meeting design rules.

CONTEST uses a similar approach in which the design variables are the geometric parameters that define a model. Constraints are expressed using cost or error functions that measure how well the constraints are satisfied. Normally, if a constraint is completely satisfied, the value of its cost function is zero. Otherwise, the cost function returns a value that indicates how severely the constraint is violated.

For example, suppose we wish to constrain two objects to be five units apart from one another. If the objects in a given model are indeed five units apart, then the cost function should return a value of zero. If not, then the cost function should return a non-zero value, with the magnitude of the value increasing with the difference between the desired distance (five) and the actual distance. One cost function to accomplish this is:

cost = abs(5-distance(object1,object2))

This function provides a cost that increases with distance from the goal. In chapter 5, we will see that there are additional guidelines to cost function design which make this function a starting point for a more sophisticated cost function. In addition, for each cost function, c_i, the modeler may specify a weighting factor, w_i, to reflect the relative importance of the constraint. The weighted cost values are then summed to yield a total cost function that expresses how well an entire model matches all constraints.

Formally,

$$c_i = f_i(\textit{design parameters})$$
$$w_i = \textit{weighting coefficient for constraint } i$$

$$c_{total} = \sum w_i \, c_i = f_{total}(\textit{design parameters})$$

As was discussed in chapter one, this summation representation represents the composition of constraints corresponding to the *and* operation. A more complex expression is necessary to represent additional operations, such as *or*. The simplified form is used here since it corresponds to the current implementation, and support for additional operations has not been needed.

The cost of any particular design can be determined by supplying the inputs (design parameters) to f_{total}, and examining the output, c_{total}. The optimization process involves developing and testing different inputs in search of an acceptable solution.

This approach quantifies constraints to yield a single scalar value which reflects the quality of any particular design. The optimal design is the configuration which minimizes the cost function. This problem formulation has therefore transformed the constraint satisfaction problem into an optimization problem.

3.6. Characteristics of applications

The problem representation used by CONTEST allows flexibility in constraint specification, but because of this flexibility, CONTEST cannot guarantee optimal results. In addition, the solution method uses probabilistic techniques; this further affects the class of problems to which CONTEST it well-suited. In general, CONTEST will perform well on constraint applications with the following characteristics:

- applications where near-optimal solutions are acceptable
- applications with many optimal solutions
- applications with non-intuitive solutions or where the design process is unknown
- applications where some randomness is acceptable or desirable

3.6.1. Near-optimal solutions acceptable

CONTEST cannot guarantee that it will find the optimal solution to a constraint problem. Even when the system reaches an optimal solution, it cannot detect that the solution is optimal. CONTEST is therefore suited to applications that require good solutions, rather than the best solution. When CONTEST does not find an optimal solution, the solution it returns will tend to be a local optimum. The resulting model will thus appear reasonable in that no small changes can be made to better satisfy the constraints.

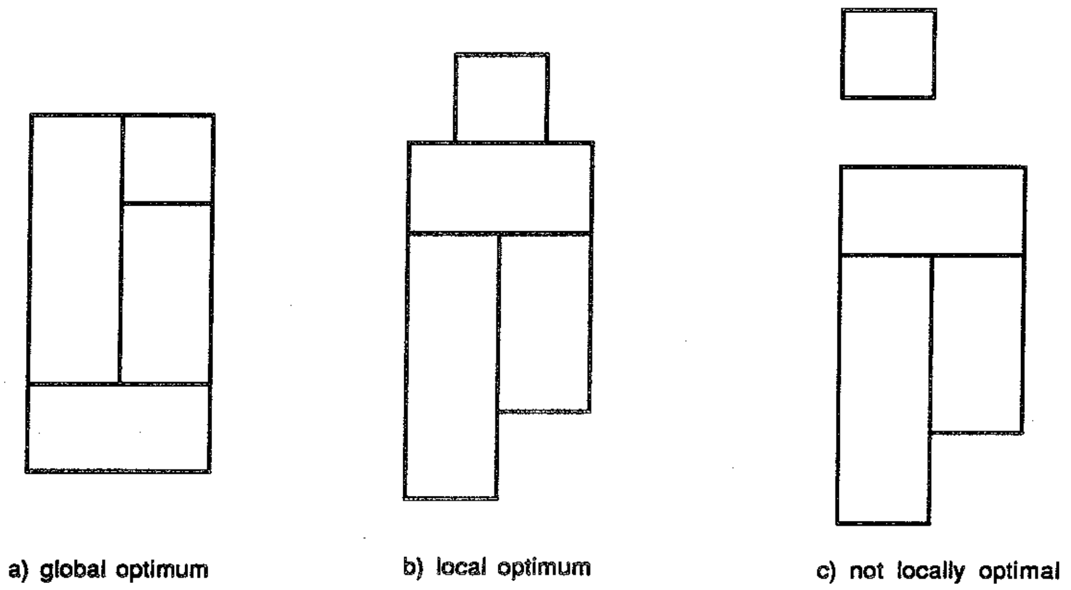


Figure 3.3: global vs. local optima in a packing problem

Suppose we use CONTEST to pack objects together so that the area of their bounding box is minimized. Figure 3.3a shows a global optimum. Figure 3.3b shows a local optimum that might be acceptable. Figure 3.3c shows a solution that is not locally optimal.

3.6.2. Many optimal solutions

CONTEST explores the space of all possible solutions when searching for an optimal solution. The probability of finding an optimal solution is greater if there are many optimal solutions as opposed to a single configuration.

3.6.3. Non-intuitive solutions

CONTEST requires no initial approximation to the final solution. Its solution method does not attempt to emulate the human design process. Consequently, it explores potential solutions that are non-intuitive and might not be found by a rule-based approach. It is suited to solving problems where a systematic human design process does not exist, or where design experts are unavailable.

3.6.4. Randomness acceptable or desirable

CONTEST searches for a solution to a design problem by making random perturbations to the design variables. As a result, underconstrained variables can be set to unpredictable values. In some cases this is undesirable; in other cases it can add complexity and naturalness to models that otherwise would appear computer-generated. By generating several potential solutions, the designer can evaluate different near-optimal solutions and potentially gain insight to the problem.

3.7. Summary

Geometric design is a complex process involving many thousands of rules and guidelines learned through experience. Totally automating the design process is nearly impossible for several reasons. The enormous task of both acquiring and encoding design knowledge is a limiting factor. Subjective constraints are particularly difficult to quantify. In addition, design is usually an evolutionary process in which the problem specification is modified as the designer explores the problem. Entirely automated design requires a complete understanding of the problem at specification time.

Despite these limitations, non-trivial design tasks can be solved with the aid of a computer. Many constraints *are* easily quantified, and when a problem can be properly encoded the computer can be used to find and evaluate solutions. An objective of this work is to expand the role of the computer from solving only trivial local constraints to solving very general global constraints. The designer can then better utilize his time in achieving a final design.

CONTEST pursues this objective by using a very broad definition of the term constraint. Essentially, any design guideline that can be quantified by a single scalar cost function can be used. This representation can gracefully represent underconstrained and overconstrained systems. Individual cost functions are summed to yield a total cost function which is then optimized using black box optimization techniques.

Because of the black box formulation, it is impossible to tell when an optimal solution has been attained. Moreover, because of the computational complexity of many optimization problems, it may be infeasible to expect an optimal solution from this or any method in a reasonable amount of time. Therefore CONTEST is best suited to problems where near-optimal results are satisfactory. The optimization method used by CONTEST is described in the following chapter.

Chapter 4

Function Optimization

This chapter discusses optimization of the global cost function. It begins by describing the goal of optimization problems in general, and characterizes the optimization problem for the geometric modeling application. The limitations of simple hill-climbing techniques are examined, and the need for more sophisticated search techniques is justified. Probabilistic optimization techniques are introduced as a way of finding global, rather than local, optima. The chapter concludes with an explanation of how a particular probabilistic optimization technique, simulated annealing, can be applied to the constraint-based modeling problem.

4.1. Formulation as an optimization problem

The goal of an optimization problem is to find the configuration that minimizes or maximizes the value of an objective function. The objective function in this case is the global cost function which incorporates the costs of all constraints. The design variables (the function input) are usually geometric variables (lengths, positions, etc.), though other model parameters (e.g., color) can be used as design variables. Parameters whose value cannot change are not design variables; instead, they are simply data that help to define the model.

Although the design variables are free to take on any values, in practice most variables have a restricted range. If we are placing furniture in a room, for instance, the set of feasible bookshelf positions should be determined by the bounds of the room. Such a restriction can be encoded as a constraint, but because it is such a hard constraint, and because cost function design is simplified by advance knowledge of such restrictions, range information is made available to the optimization procedure.

4.2. Characterization of the optimization problem

This section discusses some of the issues that led to the decision to use simulated annealing techniques for solving the optimization problem in geometric model design.

CONTEST was designed to be capable of solving a wide variety of modeling problems. Each of these modeling problems is defined by the particular constraints chosen or defined by the human modeler. The modeler should *not* be responsible for devising a solution method. Instead, a method capable of solving any specified optimization problem is needed.

Without restrictions on the form of the constraints, it is not possible to devise a general purpose analytic solution to the constraint satisfaction problem short of trying all possible solutions. Any analytic solution method would be computationally intractable since many simple optimization problems are known to be NP-complete (e.g., the traveling salesman problem).

Not only do we not have an analytic solution to the general constraint satisfaction problem, but we also have little insight about efficient problem-specific heuristics that could be used to reach a near-optimal solution. Again this is because of the arbitrary way that constraints are specified, and because the problem is subject to interactive change: the modeler may continually add new constraints or refine existing constraints. In addition, the constraints may be arbitrarily complex (e.g. non-linear in terms of the variables, with costs oscillating as a design variable is swept through its range of values).

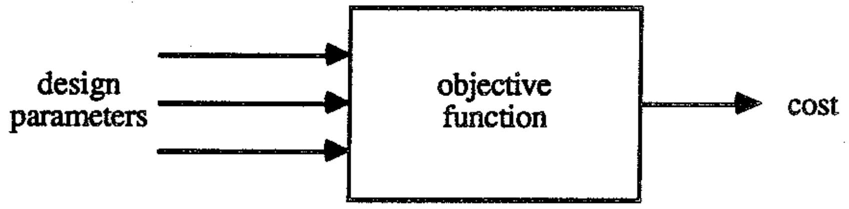


Figure 4.1: black box function evaluation

Instead of using information about the form of the function to be optimized, the solution method treats the function as a *black box*. In other words, it has no information about the function at the start of the optimization procedure, and all information is gained by supplying inputs to the black box and examining the resulting output.

Any such problem formulation requires some heuristics to efficiently explore the solution space. Although the process of solving a black box optimization problem requires heuristic search, the heuristics must be generally applicable to all functions, rather than custom-tailored based on assumptions about the form of the function. Only then can the solution method be built entirely separate from the constraint specification.

Treating this problem as a black box optimization has several important implications. Since the solution method has no knowledge of the constraints (and their associated cost functions), it cannot determine the minimum value it seeks. Even if it finds the optimal configuration, it has no way of knowing that it has done so. The search method cannot guarantee finding the optimal solution in any finite amount of time.

Fortunately, in many applications the global optimum is not needed. Instead, local optima near the global optimum may be satisfactory. This work is geared toward those applications. Thus, while the optimization method seeks the absolute minimum, the overall problem is to find a satisfactory solution.

4.3. The limitations of simple hill-climbing techniques

Although we seek the minimum of the function, this discussion refers to hill climbing, a maximization technique. To find the minimum of a function we need only find the maximum of its negative.

The black box problem formulation assumes that we have no information about the function. At times, however, the system designer does have some information about the general form of the function and can use this information to select an appropriate search procedure. In particular, if the function is unimodal, then a simple hill-climbing strategy can be used to find the optimum. Unfortunately, any sophisticated constraint-based design problem will generally yield a multimodal cost function.

Simple hill climbing works as follows: we start at some initial point (random or user chosen) on the function, and examine configurations in the neighborhood of the current configuration. If a configuration gives a higher function value than the current configuration, we accept that move and hence climb the function towards its maximum. When we reach the maximum, all moves lead downward, and the search or climb is complete.

This technique assumes that the function is unimodal: i.e., there exists a single extremum in the function. It is not successful at finding optima of multimodal functions, where each function can have many local minima and maxima. Imagine standing halfway up a foothill near a large mountain. Your goal is to reach the peak of the mountain, but with the restriction that you can only climb upwards. Unfortunately, the best that you can do in that situation is to reach the top of the foothill. To reach a global optimum, one must sometimes climb downward before resuming the ascent toward the maximum. This is the goal of probabilistic optimization techniques; they allow steps away from direction of interest in the hope of finding a path to the global optimum.

4.4. Description of probabilistic optimization techniques

This section describes the general form of a class of probabilistic optimization heuristics. The specific subclass of simulated annealing techniques is then described. The terminology and pseudo-code for these techniques is adapted from [Nahar, 1986].

4.4.1. Probabilistic hill-climbing heuristics

The general form of an adaptive probabilistic heuristic to minimize the value of an objective function *f()* is presented below. This heuristic is called adaptive because the parameters of the heuristic may be changed as the procedure is executed.

```
procedure ProbabilisticHillClimbing;
S := $S_0$;
Initialize heuristic parameters;
repeat
   repeat
      $S_{new}$ := perturb(S);
      if accept($S_{new}$,S,heuristic parameters) then S := $S_{new}$;
   until "time to adapt parameters"
   Adapt parameters;
until "termination criteria";
end;
```

S:	The current solution to the problem. S_0 is usually a random solution, though it may be a generated approximation believed to be near the optimal solution.
perturb:	A function that generates a new solution from the current solution.
S_{new}:	A perturbed version of *S*. I.e., a version of *S* with the values of one or more of the design variables changed.

`accept`: A boolean function that determines whether the perturbed solution S_{new} should be accepted as the current solution. This function has the form:

```
accept := f(S_new)<f(S) or
          random<g(f(S),f(S_new),heuristic parameters)
```

where:

random is a random number in the range [0,1].
g(cost1,cost2,heuristic parameters) is a function that determines the probability of accepting a perturbation that increases the cost function.

`Adapt parameters`:
a procedure that updates any parameters that are used anywhere in the heuristic.

This heuristic is called probabilistic because of the form of the accept function. Conventional heuristics accept a perturbation only if it decreases the value of the objective function *f()*. The probabilistic acceptance function above always accepts a perturbation that decreases the value of the objective function; however, it also may accept a perturbation that increases the value of the objective function. These "bad" perturbations are accepted to prevent the heuristic from always converging to a local, rather than global, optimum. In general, the probability of accepting a bad perturbation is set highest during initial iterations, and is decreased (possibly reaching zero) as the algorithm proceeds. In intuitive terms, the heuristic initially searches the solution space to find the optimal hill, and then climbs that hill in later stages of the optimization process.

4.4.2. Simulated annealing

A particular probabilistic hill-climbing technique, simulated annealing, has been successfully applied to a variety of engineering problems, especially VLSI optimization problems [Kravitz, 1986], [Pincus, 1986], [Romeo, 1985], [Sechen, 1986], [Wong, 1986]. Simulated annealing is based on an analogy between minimizing the cost of a function and the careful cooling of a solid so that it reaches its minimum energy state [Kirkpatrick, 1983].

In physics, annealing is a process for reaching low energy states in a solid. The solid is heated until it melts, and then slowly cooled until the particles of the solid are arranged into their minimum energy (ground) state. When heated to a high temperature, the particles in the liquid are arranged randomly, and the liquid has an equal probability of being in each possible configuration. If properly cooled, however, the particles of the solid are carefully structured into the minimum energy configuration with probability one.

Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller [Metropolis, 1953] devised an algorithm for simulating the annealing process. In the algorithm, each configuration has an associated energy. The algorithm starts with a random initial state and high temperature, and generates new states by making small perturbations to the current state. Let the energy of the current state i be E_i and the new state j be E_j. If E_j is less than E_i, then the new state is accepted as the current state. If E_j is greater than E_i, however, then state j is accepted as the new state with probability

$$exp\left(\frac{E_i - E_j}{k_B\, t}\right), \tag{4.1}$$

where k_B is a physical constant called the Boltzmann constant, and t is the temperature of the system. If the cooling of the system is performed slowly, then the system is able to reach *thermal equilibrium* at each temperature. When the system is in thermal equilibrium, the probability of it being in state i with energy E_i is

$$q_i(t) = \frac{exp\left(\frac{-E_i}{k_B\, t}\right)}{\sum_j exp\left(\frac{-E_j}{k_B\, t}\right)} . \tag{4.2}$$

This is known as the Boltzmann distribution, and is important because it defines probability distributions which must be maintained by any simulated cooling if optimal results are to be achieved. We use $q(t)$ to refer to this distribution for all states.

The simulated annealing algorithm may be applied to general optimization problems by drawing the following analogies:

Physics	**Function optimization**
State	*Configuration/solution: S*
Particles	*Variables*
Energy	*Value of function: f(S)*
Ground state	*Optimal solution*

For general optimization problems, the $k_B t$ expression can be replaced by a general control parameter T, which acts as a temperature but has no physical meaning outside the solution method. The general form of the simulated annealing heuristic (again adapted from [Nahar, 1986]) is as follows:

```
procedure SimulatedAnnealing;
S := $S_0$; k := 0;
T := $T_0$; iterations := $i_0$;
repeat
   repeat
      $S_{new}$ := perturb(S,T);
      if accept($S_{new}$,S,T) then S := $S_{new}$;
   until inner loop has been repeated iterations times;
   k := k + 1;
   decrease(T); increase(iterations);
until "termination criteria";
end;
```

where:

decrease(T) decreases the temperature,
increase(iterations) increases the number of loop iterations,

and the acceptance function has the form:

$$\text{accept} := \text{if } f(S_{new}) < f(S) \text{ or random} < \exp\left(\frac{f(S_{new}) - f(S)}{T}\right).$$

The values T_0 and i_0, and the procedures *decrease()* and *increase()*, define the annealing schedule for the simulated cooling of the system. There are many possible annealing schedules. In the following section, two possible schedules are discussed.

A number of termination criteria are possible. The simplest is to stop when the system has cooled to a certain temperature. Another possibility is to terminate when the marginal improvement in the objective function is so small that it appears that the process has converged to an optimum. A third possibility is to display intermediate results and quit based on user intervention.

Simulated annealing is of interest as an optimization technique because theoretical results exist which specify sufficient conditions for convergence to the optimum (e.g., [Geman, 1984], [Gidas, 1985], [Aarts, 1989]). The conditions for convergence require either infinitely slow cooling or an infinite number of iterations at each step. In practice this is not possible, so simulated annealing cannot be used to guarantee an optimal solution. However, these results provide a stronger foundation for the success of simulated annealing compared to *ad hoc* probabilistic optimization techniques.

Consider a method that randomly chooses a finite number of starting points and performs simple hill-climbing on those points. Even with infinitely small hill-climbing steps, such a method is not guaranteed to find the optimum. With simulated annealing, however, if a particular annealing schedule converges to a non-optimum solution, we know that there exists a more careful annealing schedule that will perform better.

The simulated annealing procedure is simple and very general. The specifics of the problem are not essential to the execution of the algorithm. It may be possible to use problem-specific information to adjust the heuristic parameters or make intelligent perturbations, but the improvements would tend to be in execution time rather than in optimality of the solution. The tradeoff for this simplicity and generality is computational expense. Many hundreds of thousands of iterations may be needed before termination criteria are met.

This section presented the general form of the simulated annealing procedure. The following section describes how this general structure may be applied to constraint-based modeling.

4.5. Applying annealing to constraint-based modeling

This section discusses design and implementation issues of the following components of the annealing method:

- solution representation
- cost (objective) function
- perturbation functions
- acceptance function
- annealing schedule

The form of some of these components (e.g., perturbation function) is dependent on this application, while the form of other components (e.g., the acceptance function) does not differ from previous simulated annealing applications. All components are described here for completeness.

4.5.1. Solution representation

A solution is the body of information that defines a model. It may include:

- passive geometric data
- procedural geometric models and their parameters
- any other parameters associated the model (e.g. color and other reflectivity information) or parameters that control the construction of the model (e.g. light source position, camera position, other viewing parameters)

Any of the above information may be treated as design variables (i.e., variables that may be changed during the annealing process), though care must be taken that reasonable perturbation functions are associated with these variables. In keeping with an object-oriented design philosophy, the particular effect of changing any given design variable is hidden from the annealing algorithm. Thus a procedural geometric model is responsible for making the necessary changes to its geometry in response to the perturbation of one of its parameters.

4.5.2. Objective function

The objective function is simply the sum of the cost functions for each of the constraints. These individual cost functions are adjusted by weighting factors that reflect their strength relative to the other constraints. Cost function design is covered extensively in chapter 5.

4.5.3. Perturbation function

The perturbation function transforms the current configuration into a neighboring configuration. For example, if the object has three degrees of freedom, the perturbation might be a random translation in x, y, or z. If rotation is permitted, the perturbation could be either a translation in x, y, or z, or a rotation (relative to the center of the object) about the x, y, or z axes. Similarly, scaling, stretching, squashing, or other arbitrary transformations may be applied as perturbation functions. The only requirement is that the transformation be reversible, either by an inverse transformation on the perturbed object, or simply by saving a copy of the original object.

The standard geometric object implementation in CONTEST supports the following transformations: rotate x, rotate y, rotate z, translate x, translate y, translate z. At object initialization time, an initial range for each of these parameters is defined. For translation, this range is normally related to the world size, so that an arbitrary translation within that range can place the object randomly within the world. The initial rotation range is usually chosen to be greater than or equal to 360 degrees, so that initially, a rotation perturbation will result in a random orientation.

CONTEST implements the perturbation function by maintaining a range parameter, *rangeFactor*, which specifies the percentage of the initial range that should be used in determining a perturbation. At the start of the annealing process, when the temperature is high, *rangeFactor=1.0*, so perturbations yield essentially random positions and orientations. For each successive temperature decrease *k*, *rangeFactor* is decreased based on an adjustment factor. In addition, a minimum range is chosen so that even after the system has cooled to a temperature near zero, reasonably large hill-climbing steps will still be performed (for fast convergence to a local optimum). Thus,

$$rangeFactor_{k+1} = max\left(rangeFactor_k \cdot rangeAdjust, rangeMin\right) \quad . \tag{4.3}$$

The perturbation function for geometric objects can be summarized as follows:

1. select an object from list of perturbable objects
2. select a transformation from the list of available transformations for that object
3. compute the range of the transformation by multiplying *rangeFactor* by the maximum range for that transformation
4. select a random perturbation in that range
5. perform the transformation

All objects are responsible for providing a perturbation function. Geometric objects are able to share the common function described above, whereas non-geometric objects require separate perturbation functions (which may or may not be shared among classes of non-geometric objects).

The randomness of the perturbation function may at first seem needlessly inefficient, since it may be possible to make more intelligent perturbations, such as evaluating the gradient and taking a step in that direction, or perhaps trying a predefined sequence of perturbations (e.g., always try to move down first, then try translation in other directions, then try various rotations, etc.).

The difficulty in designing intelligent perturbation functions is that while such functions can increase the speed of convergence to an optimum, they also increase the probability of that optimum being local rather than global. Care must be taken to ensure that the perturbation function samples a range of configurations in the solution space. Provable convergence depends on the randomness of the perturbations.

4.5.4. Acceptance function

CONTEST uses the standard simulated annealing acceptance function: always accept a good perturbation, and accept a bad perturbation with a probability that decreases as the system cools:

$$Pr.\,(accepting\ S_{new}) = \begin{cases} 1, & if\ f(S_{new}) \le f(S) \\ exp\left(\dfrac{f(S) - f(S_{new})}{T}\right), & if\ f(S_{new}) > f(S) \end{cases} \tag{4.4}$$

4.5.5. Annealing schedule

The annealing schedule consists of the definition of initial temperature, initial number of iterations in the inner loop, and functions that update the temperature and number of inner loop iterations during each execution of the outer loop. These parameters are difficult to define because the notions of temperature and cooling are artificial for this application. This section presents two structures for annealing schedules. The first is a simple schedule proposed by Kirkpatrick, Gelatt, and Vecchi [Kirkpatrick, 1983]. The second, developed by Aarts and Van Laarhoven [Aarts, 1985], is more sophisticated and is described in detail in [Aarts, 1989].

Forms of both of these schedules were implemented in CONTEST. The first schedule was initially implemented. The non-intuitiveness of parameter selection forced the implementation of the second schedule, which automates more of the parameter selection. In the discussion below, let $\chi(T)$ represent the percentage of total transitions that are accepted at temperature T.

4.5.5.1. Simple schedule

This section describes the annealing schedule developed by Kirkpatrick, Gelatt, and Vecchi.

Initial temperature: The initial temperature should be high enough that almost all perturbations are accepted. It can be set by starting at a low temperature and rapidly heating the system (increasing the temperature) until the acceptance ratio $\chi(T)$ is near one. This heating can take place in large increments, since overheating a system has no detremental effect other than increasing the time of the cooling process.

Initial inner loop iterations: At each temperature, we seek to reestablish thermal equilibrium. Exact thermal equilibrium can be maintained at all times by cooling the system infinitely slowly, or by performing an infinite number of iterations at each temperature. Since we are approximating the cooling with a finite number of steps, we can only get close to thermal equilibrium, as expressed by:

$$\| a(iterations,T) - q(T) \| < \varepsilon, \tag{4.5}$$

where $a(iterations,T)$ represents the probability distribution after a fixed number of iterations at temperature T, and $q(T)$ represents the equilibrium distribution. In practice, the initial number of iterations depends on the size of the problem, and is chosen so that a substantial percentage of neighbors of the original state are examined. A typical initial number of iterations for a small problem involving ten variables is fifty iterations, but this is provided only as a rule-of-thumb, since the actual number required depends on the complexity of the problem and the rate at which the temperature is decreased. A more complex problem requires more iterations, while slower cooling steps permit fewer iterations.

Decrease of temperature: In the simple schedule, the temperature is updated by a constant scaling factor α, so that:

$$T_{k+1} = \alpha \cdot T_k, \qquad 0.0 < \alpha < 1.0. \tag{4.6}$$

A typical value for α is .95.

Increase of inner loop iterations: At each temperature, quasi equilibrium is achieved after acceptance of some finite number of transitions. Since the acceptance rate decreases with temperature, it follows that the number of iterations at each temperature should increase. This is achieved by a multiplying by a scaling factor β, so that:

$$iterations_{k+1} = \beta \cdot iterations_k, \quad \beta \geq 1.0 . \tag{4.7}$$

A typical value for β is 1.01. An upper bound can be used to keep the number of iterations from approaching infinity as the temperature approaches zero.

Termination: A simple termination method is to quit when there is no change in final cost after a fixed number of decreasing temperatures.

4.5.5.2. Sophisticated schedule

This section describes the annealing schedule developed by Aarts and Van Laarhoven.

Initial temperature: The goal is to set the initial temperature so that the acceptance ratio, $\chi(T_0)$, is equal to a predetermined value close to one. Aarts and Van Laarhoven noted that a sequence of trials can be used to compute T_0. Suppose we run a series of m_0 trials, with the result that m_1 transitions decrease the cost of the function, and m_2 transitions increase the cost of the function, where $m_0 = m_1 + m_2$. The acceptance ratio can then be approximated by:

$$\chi(T) \approx \frac{m_1 + m_2 \cdot exp\left(\frac{-\overline{\Delta f}}{T}\right)}{m_1 + m_2}, \tag{4.8}$$

where $\overline{\Delta f}$ represents the average increase in cost for the m_2 transitions. From this equation we can derive:

$$T \approx \frac{\overline{\Delta f}}{ln\left(\frac{m_2}{m_2 \cdot \chi(T) - m_1(1-\chi(T))}\right)} . \tag{4.9}$$

To compute T_0, we set $\chi(T)$ equal to the desired ratio χ_0, and use equation 4.9 to compute successive values of T after each trial. Experience indicates that the value of T converges to T_0 rapidly.

Initial inner loop iterations: The number of iterations of the inner loop depends on the size of the problem and should be chosen so that a fraction of the neighboring configurations of the initial configuration is examined. For example, a configuration consisting of objects with (x,y) positions should evaluate at least two or three increases and decreases of both x and y.

Decrement of temperature: We assume that quasi equilibrium is attained at T_0, since we have chosen this initial temperature. To maintain quasi equilibrium, we want the difference in probability distributions for successive temperatures to be close enough so that, for the new temperature, quasi equilibrium can be restored after a fixed number of iterations. We can define *close* as meaning that for each possible configuration *i*,

$$\frac{1}{1+\delta} < \frac{q_i(T_k)}{q_i(T_{k+1})} < 1+\delta, \quad k = 0, 1, \ldots \ , \tag{4.10}$$

where δ expresses how close we need to be. In [Aarts, 1989], Aarts and Korst argue that for a given δ, the restrictions of equation 4.9 can be satisfied by choosing new temperatures as follows:

$$T_{k+1} = \frac{T_k}{1 + \frac{T_k \cdot \ln(1+\delta)}{3\sigma(T_k)}}, \quad k = 0, 1, \ldots \tag{4.11}$$

where $\sigma(T)$ represents the standard deviation of the function at a given temperature. The standard deviation can be approximated by saving and evaluating the standard deviation of the costs for the previous inner loop.

Increase of inner loop iterations: In this schedule, quasi equilibrium is maintained by adapting the temperature decrements to the statistical properties of the previous sequence of inner loop iterations. The number of inner loop iterations does not change.

Termination: We make the decision to terminate based on an approximation of nearness to the optimum. If we let $\langle f \rangle_T$ represent the expected cost of the function at temperature T, then we can define $\Delta\langle f \rangle_T$, the expected distance from the function optimum, as:

$$\Delta\langle f \rangle_T = \langle f \rangle_T - f_{optimal} \tag{4.12}$$

We can terminate if this distance is small compared to the initial expected cost, $\langle f \rangle_{T_0} \approx \langle f \rangle_{\infty}$. For small temperatures, $\Delta\langle f \rangle_T$ may be approximated by:

$$\Delta\langle f \rangle_T \approx T \frac{\partial \langle f \rangle_T}{\partial T} \tag{4.13}$$

We can terminate if, at some k,

$$\frac{\Delta\langle f \rangle_{T_k}}{\langle f \rangle_{\infty}} = \frac{\textit{expected change in cost}}{\textit{initial cost}} < \varepsilon_{stop} \tag{4.14}$$

This can be expressed as:

$$\frac{T_k}{\langle f \rangle_{\infty}} \frac{\partial \langle f \rangle_T}{\partial T}\bigg|_{T=T_k} < \varepsilon_{stop} \tag{4.15}$$

The expected cost at each outer loop iteration, k, can be approximated by saving and evaluating the average of the costs for the previous inner loop.

4.5.6. Evaluating the Solution

Convergence to a solution is affected by many components of the annealing process: the cost functions, the perturbation functions, the acceptance function, and the annealing schedule. The evaluation of whether a particular configuration is acceptable may be determined using one of the following criteria:

- all constraints are satisfied: this has been the traditional method for determining whether a solution is satisfactory. This criterion is unsuitable for overconstrained problems, or problems with goal constraints.
- comparison with previous designs: many design applications involve redesign of existing products. In these cases, there exists a basis for comparison. The new solution is acceptable if it is better than the previous version according to the defined cost functions.
- user evaluation: as a last resort, the user of the modeling system must subjectively decide whether the solution is acceptable. He or she may do this by examining the model and the values of the cost functions for the model.

If the solution has converged to an unsatisfactory local optimum, the user can explore several alternatives:

- rerun the simulation: the stochastic nature of the annealing process may yield a different solution
- modify the parameters in the annealing schedule
- add search constraints to guide the solution process

The first alternative is simplest but may fail frequently. The second alternative is relatively simple and is likely to succeed. The final alternative may be successful if the user has intuition about the general form of the final solution.

4.6. Summary

This chapter discussed several issues relating to cost function optimization. The decision was made to treat the cost function as a black box optimization problem, where knowledge about the function can only be gained by evaluating it. Probabilistic optimization techniques were discussed, and simulated annealing, a particular probabilistic technique, was selected as a method applicable to this research. Details of adapting the method to constraint-based modeling were provided. One key issue, the form of the objective function, was only briefly discussed, and is presented in detail in the following chapter.

Chapter 5

Cost Functions

A cost function for a constraint is a function that, when evaluated on a configuration, returns a measure of how well that constraint is satisfied. This measure is a scalar value that expresses the amount of constraint violation. When we compare two designs and say that one design is better than the other, we are weighing various factors to reach a judgement. The development of cost functions is an attempt to quantify these assessments.

In one sense, all cost functions are equivalent because they return a single real number; no distinction is made between classes of constraints as they are combined to form the total cost function, or as the total cost function is optimized by the solution procedure. On the other hand, for instructional purposes, three categories of constraints can be identified: objective constraints, subjective constraints, and search constraints. Objective constraints are constraints that can be objectively evaluated because they deal with material properties rather than aesthetic concepts. Subjective constraints involve an individual's value judgements. Quantifying subjective constraints is more difficult, but cost functions can be constructed by breaking down subjective constraints into a series of simpler, objective evaluations. Search constraints provide a way for the designer to direct the constraint satisfaction search to a portion of the solution space.

This chapter discusses cost function design for geometric modeling applications. It begins by presenting examples of different constraints. The theory behind cost functions and the context of their use is then discussed. Next, general guidelines for cost function construction are presented. Individual guidelines for objective, subjective, and search constraints are then described, and cost function templates are derived based on these guidelines. The chapter concludes with a brief discussion of operations on constraints.

5.1. Examples of constraints

Constraints may be applied to all types of geometric objects -- from a single coordinate of a vertex to entire assemblies consisting of thousands of primitive objects. Within this spectrum lie many object types:

• vertices	• line segments
• spheres	• cylinders
• bicubic patches	• polygons
• plane equations	• polyhedral solids
• mesh surfaces	• etc.

Later in this chapter, constraints are described as falling into three categories, based on construction of their corresponding cost functions: objective constraints, subjective constraints, and search constraints. In this section, however, a different classification is used. Constraints can be considered geometric, indirectly geometric, or non-geometric. These categories distinguish between constraints that have traditionally been used in modeling systems (geometric) and constraints that have not traditionally been available but which are supported by CONTEST (indirectly geometric and non-geometric).

5.1.1. Geometric constraints

Geometric constraints typically involve one or two simple object types. A geometric constraint might restrict the range of a particular value (e.g., "make X less than 20.0"), or specify a relation between two objects (e.g., "object A should be above object B"). Several subcategories of geometric constraints follow. These are only examples and do not form an exhaustive list.

5.1.1.1. Position constraints

- minimize the y coordinate of a point
- set z coordinate to 20.0
- maximize the height of a sphere

5.1.1.2. Distance constraints

- place two objects so they are at least 10.0 units apart
- make two objects be adjacent to one another
- maximize the distance between two objects

5.1.1.3. Orientation constraints

- make a particular side of a cube face toward a point (e.g., a light source)
- set the angle between two adjoining line segments to 45 degrees
- ensure that the base of a cylinder is flush with a particular plane

5.1.1.4. Size constraints

- minimize the surface area of a container
- maximize the volume of a cylinder
- ensure that object A can enclose object B

5.1.1.5. Intersection constraints

- ensure that two objects do not intersect
- set the volume of intersection of two spheres to 15.0
- ensure that object A is contained within object B

5.1.2. Indirectly geometric constraints

The constraints in this category ultimately affect geometry, yet are usually specified indirectly in terms of some property that is a complicated function of the geometry. Evaluation of the constraints may involve a series of computations, which makes it difficult to solve the constraints analytically. Examples of such constraints include:

- ensure object A is visible to object B
- ensure that a garden plot receives sufficient sunlight for plant growth
- minimize the drag of an airplane wing
- minimize the operating temperature of a machine
- ensure that a part meets federal safety requirements
- maximize the area of the shadow cast by object x

5.1.3. Non-geometric constraints

While the primary product of a modeling system is a geometric description, models may also have non-geometric parameters associated with model parts. Color, age, and reflectivity parameters are a few examples of parameters than can greatly influence design. Examples of constraints involving these parameters include:

- select an object color so that it matches its environment
- all else being equal, select newer parts when a choice is available
- set an object's perceived intensity from a particular viewpoint

5.2. Theory and use of cost functions

The purpose of cost function design is to provide a way to compare two or more models. Many cost functions return a value of zero if their constraints are completely satisfied, and a value greater than zero if the constraint is not satisfied. Functions can return negative values, however. For example, a property (e.g., surface area) can be maximized by defining a cost function equal to the negative of that property (though CONTEST does not use this representation). Since cost functions can take on any range of values, the value returned by a cost function has little meaning by itself. Instead, its meaning lies in the process of comparing costs between different models.

CONTEST provides a library of cost functions for use in constructing a model. The library consists of functions for objective and search constraints. Subjective constraints tend to deal with application dependent interpretation of geometry and must therefore be constructed by the designer. A typical application will apply library constraints and may or may not apply additional user-defined constraints.

A user of CONTEST can define new models in two ways. One way is to supply new constraints and geometric objects at each session. This is the method of design for a person creating a series of distinct models. The site planning examples in chapter seven are examples of models in this category. A problem defining the layout of a university campus requires a different set of constraints than the plan for a suburban neighborhood.

The second design method is to first build a single application capable of generating many instances of designs from a single class. The user then specifies a particular design by providing data or weights to constraints. The opaque projector application in chapter six demonstrates this style of design. In that example, a single set of constraints defines a class of projectors. Individual projectors are selected by setting parameters to constraints, rather than creating new constraints.

5.3. General cost function guidelines

Cost function design is guided by three concerns: choosing a unit of measure for the cost function, specifying a satisfactory shape which reflects design tradeoffs, and selecting a function which helps the annealing or optimization process. This section suggests some general guidelines for cost function design based on these concerns.

5.3.1. Units of measure for cost functions

5.3.1.1. Choose a single unit of measure

Recall the form of the global cost function for concatenated constraints:

$$c_{total} = \sum w_i \, c_i \,, \tag{5.1}$$

where w_i is the weighting factor for constraint i, and c_i is the cost function.

Suppose each constraint, c_i, has its own unit of measure. The designer must then choose each weighting factor, w_i, to reflect both a conversion to the units in which c_{total} is expressed as well as an expression of the relative importance of c_i in these units. Even if the designer does not explicitly state these individual components, the fact remains that he has converted from c_i units to c_{total} units for each constraint i. In other words, he has effectively defined a single unit of measure for each constraint. Thus, if the user does not choose a single unit of measure at cost function definition time, he will end up choosing one each time a global cost function is defined. The former choice is usually preferred.

5.3.1.2. Choose dollars as the unit of measure in product design

For any type of product design, constraints involving part costs will most naturally be expressed in real dollar values. Since this unit of measure will already exist for some constraints, it may simplify things to choose dollars as the global unit of measure. This can lead to uncomfortable decisions (e.g., putting a dollar value on items such as safety), however these decisions must be made regardless of unit of measure and choosing an artificial measure only clouds the issue.

For other applications, one can choose an arbitrary unit of measure for the first constraint, and base new constraints on their importance relative to existing constraints. The problem with this is that reuse of constraints (e.g., from libraries) requires either a common basis or a rescaling of each constraint. This issue is further explored in the next guideline.

5.3.1.3. Select a standard range of values

From an optimization standpoint, the magnitude of cost function values is not as important as the *range* of values the function assumes. This is because at the end of the optimization procedure, we are only interested in the input (i.e., modeling parameters) that yielded the minimal cost, and not the cost itself.

What are the implications of this? First, it means that any cost function can be adjusted upward or downward by any constant value and still behave equivalently. Second, it means that the value of the total cost function is meaningless in measuring quality. As mentioned above, only by comparing configurations can quality be determined. In particular, this means that unless a standard range is defined, a cost function with a value of zero does not necessarily indicate that all constraints are satisfied.

From the designer's standpoint, however, comparison and evaluation of cost functions becomes difficult if each constraint assumes a different range of values. Thus, a standard range of functions values can simplify the comparison and evaluation process. CONTEST uses an arbitrary standard of 0.0 representing maximum satisfaction and values greater than zero representing increasing constraint violation.

5.3.2. Shape guidelines for cost functions

5.3.2.1. Define the cost of compromise configurations

Consider a constraint that a sphere should have a volume of 20 units. One can simply determine whether the constraint is satisfied by computing the volume of the sphere. A possible cost function for this constraint is shown in figure 5.1. Problems with this cost function are that it is difficult to find the single acceptable point when performing a black box optimization, and no distinction is made between unsatisfactory solutions; a volume of 1.5 is considered no worse than a volume of 19.5.

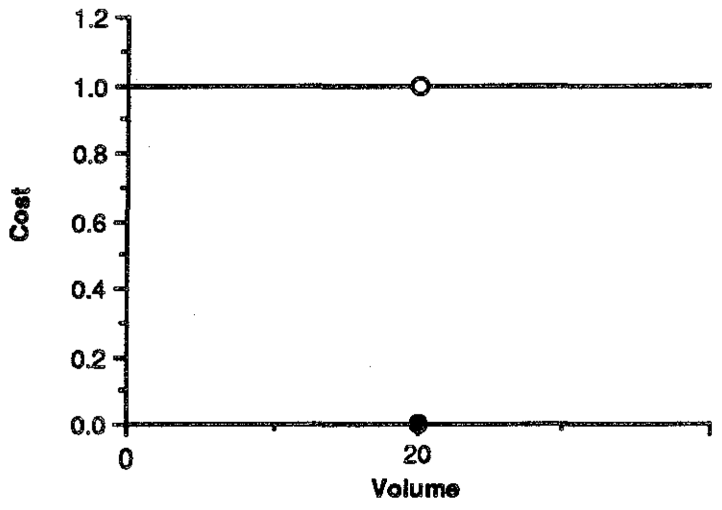


Figure 5.1: a simplistic cost function

Figure 5.2 shows a cost function in which function values gradually approach the minimum value of zero as the volume approaches the desired value of twenty. There are two reasons why cost functions should be designed with broad slopes to the optimum value. First, such a slope allows the search procedure to determine the direction of the optimum by comparing neighboring configurations, even far from the optimum value. Second, in an overconstrained problem, the cost function defines which compromise configurations are closest to the optimum.

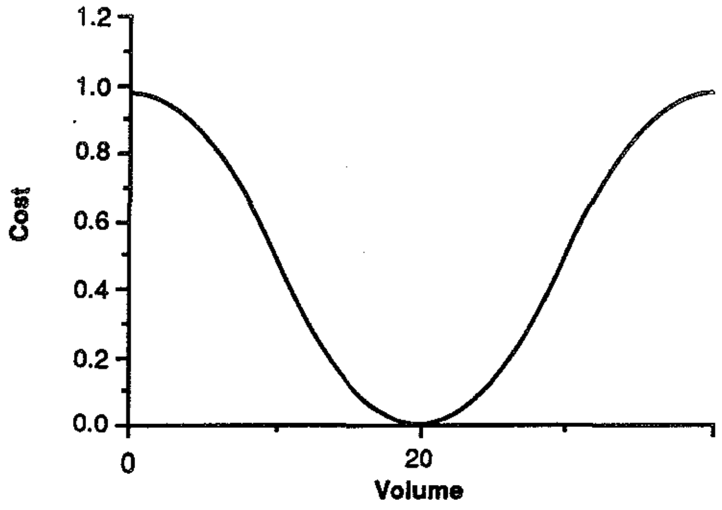


Figure 5.2: a more useful cost function

5.3.2.2. Shape should be accurate and easy to specify

A designer may specify hundreds of constraints in defining a model. To ease this constraint definition task, a minimal number of parameters should be required to specify the shape of a cost function. Modifying the shape of the function should only require changing the values of one or more of these parameters.

There is, of course, a tradeoff between ease of definition and accuracy of representation. CONTEST takes the approach of identifying the most common classes of cost functions, and allowing the user to shape standard cost function templates for each of these classes. This provides simple and accurate cost function representation.

5.3.2.3. Attempt to separate representation from specification

Ideally, a designer should not have to understand the way constraints are implemented and solved. Instead, the designer should be able to describe constraints in general terms. Consider the volume example depicted by figure 5.2. The designer should be able to express the cost dropoff in terms of general properties, such as "a broad tolerance for values close to 20.0", rather than in specific terms such as "the cost of a volume of 25.0 is 0.2". Unfortunately, terms such as *broad* and *close to* are not precise, so there is a tradeoff between precision and abstraction when specifying constraint properties.

5.3.3. Constructing functions to aid the optimization process

5.3.3.1. Functions should encourage hill-climbing

The simplistic cost function depicted in figure 5.1 is undesirable from a design point of view because it does not distinguish between unacceptable solutions. However, it is also undesirable based on the characteristics of the optimization process. Any black box optimization technique will have zero probability of finding a single point if the domain of function parameters is infinite, and the probability is near-zero even with a finite domain. Thus the cost function in figure 5.1 is incompatible with the annealing procedure.

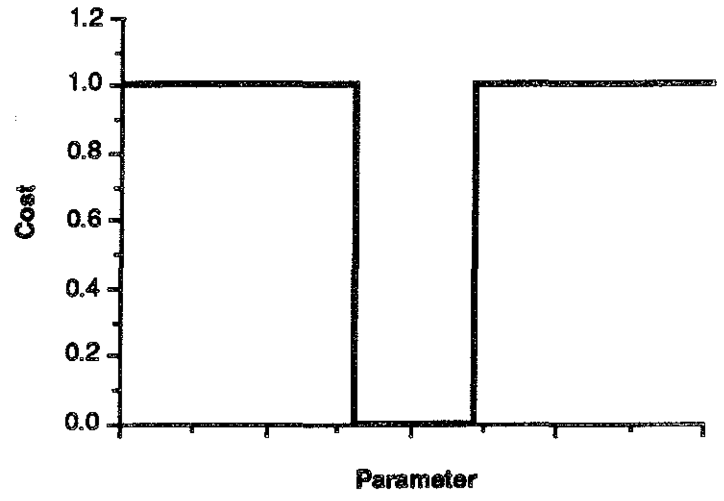


Figure 5.3: a wider region of accepted values

An alternative is to widen the range of accepted values, as shown in figure 5.3. The problem with this approach is that it accepts many values as being equally acceptable as the desired value. We can narrow the region of acceptable values to prevent this, but as the width of the accepted region approaches zero, the probability of finding that region also approaches zero. There is a tradeoff between accepting undesirable values and finding any parameter near the target.

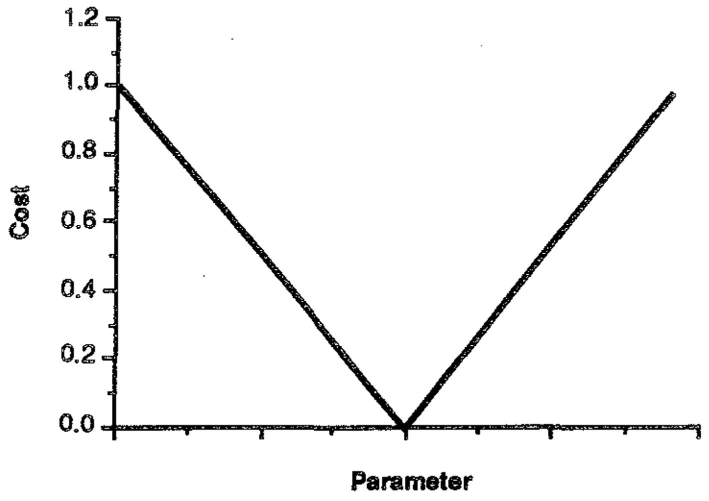


Figure 5.4: guiding search to optimal value

A third possibility is to shape the function so that the direction of the desired value can be determined throughout the parameter domain. Sloping the function as in figure 5.4 accomplishes this goal. The guidance provided by such a function shape is particularly imporant during the last stages of annealing. As the system cools, annealing reduces to hill-climbing toward a local optimum. Note that these annealing-related guidelines are compatible with the design tradeoff guidelines of section 5.3.2.

5.3.3.2. Always reward movement toward the goal

This guideline is best demonstrated by example. A possible cost function for non-intersection of solids is the volume of their intersection. Now consider a constraint that says a small sphere should not intersect a large cube. The simplistic cost function will return the same value for any configuration with the sphere entirely within the cube. A smarter cost function will reward the sphere for being closer to the edges of the cube and thus closer to a correct solution. The general principle is that cost functions should distinguish between acceptable solutions whenever possible.

5.3.3.3. Functions should be easy to evaluate

Annealing is a time-consuming process. Much of the computation time is dedicated to reevaluating cost functions after each perturbation. By selecting functions which are easy to evaluate, the total run-time to solve a problem can be reduced.

5.4. Cost functions for objective constraints

It is simple to tell when an objective constraint has been satisfied, because objective constraints deal with concrete geometric properties such as distance, volume, intersection, and area. There can be no argument as to whether two objects intersect, or whether the volume of a cylinder is more than 50 milliliters. Cost functions for objective constraints are not trivial to define, however. In addition to determining when a constraint has been satisfied, the cost function must also define the cost of compromise configurations in problems with conflicting constraints. The two main difficulties in building cost functions are: 1) defining the shape of each function, and 2) determining the importance of each cost function relative to other cost functions.

This section attacks these problems in concert. Constraints are categorized into six basic constraint types: equal (=), not equal (≠), less than (<), greater than (>), minimize, and maximize. Standard cost function templates are defined for each of these types. The particular parameters that shape the cost function are defined by the user in a natural manner and converted to the appropriate function coefficients and exponents. The balancing problem is addressed by ensuring that the functions return cost values within a bounded range.

5.4.1. Selecting the controlling parameter

The first step in defining a cost function is selecting a parameter which represents the property being constrained. For some constraints, this selection is trivial. For a constraint minimizing the distance between two points, the obvious parameter to use is the distance between the points. Similarly, for a constraint minimizing the area of a surface, the obvious parameter is the area.

For other constraints the parameter is not so obvious. With these constraints, the constraint designer must determine a parameter that provides a good representation of the constraint. Consider a constraint that two spheres not intersect. One possible parameter for measuring constraint violation is the volume of their intersection. On the other hand, the spheres will intersect only if the distance of their center points is less than the sum of their radii. Thus an alternate parameter is simply the difference between the distance of the centers and the sum of the radii. In the first case, the constraint is converted into a volume minimization constraint. In the second case, the constraint is converted into an inequality constraint. It is the responsibility of the constraint designer to determine which parameter can best be used to measure constraint satisfaction.

In general, constraints which are already expressed in terms of one of the six basic constraint types (see below) implicitly define a parameter. The parameters of other constraints, such as the sphere intersection constraint above, can be found readily once the constraint is transformed to one of the basic types.

Constraint →	Parameter
• minimize the y coordinate of a point	• the y coordinate of the point
• set z coordinate to 20.0	• the z coordinate
• maximize the height of a sphere	• the z coordinate of center of sphere
• place two objects so they are at least 10.0 units apart	• the distance between the objects
• maximize the distance between two objects	• the distance between the objects
• make a particular side of a cube face a point (e.g., a light source)	• the angle between the vector to the light source and the surface normal
• set the angle between two adjoining line segments to 45 degrees	• the angle between the line segments
• ensure that the base of a cylinder is flush with a particular plane	• the angle between the base and the plane plus the distance between them
• minimize the surface area of a container	• the surface area of the container
• maximize the volume of a cylinder	• the volume of the cylinder
• ensure that object A can enclose object B	• the volume of B which cannot be enclosed by A
• ensure that two objects do not intersect	• the volume of their intersection
• set the volume of intersection of two spheres to 15.0	• the volume of their intersection
• ensure that object A is contained within object B	• the volume of the portion of A outside B

Figure 5.5: constraints and associated parameters

Figure 5.5 gives example parameters for the geometric constraints presented in section 5.1.1. Once the parameter is chosen, the next step is to shape the cost function as a function of the parameter. The following section describes the various constraint types and the shape of their associated cost functions.

5.4.2. The six basic constraint types

In the course of this research, objective constraints have tended to fall into one of six basic categories, or be constructed of simpler constraints that fall into these categories. No claim is made that all constraints fall into these categories. On the contrary, it is easy to define an arbitrary constraint (and associated cost function) that has no relation to any of the basic categories. Nevertheless, these six categories have been sufficient for nearly all constraints encountered in the example problems described in later chapters.

The functions presented below are templates for the six categories, constructed about a target parameter, with function values chosen to range from 0.0 to 1.0. In practice, the target value is chosen by the cost function designer, a scaling factor may be applied to each function, and the shape of the function (within certain bounds) may be controlled by the designer. In a constraint such as "set area to 5.0", the parameter is the area, and the target is 5.0. Here, a function value of 0.0 represents complete satisfaction, and 1.0 indicates maximal constraint violation. Specific information about shaping these templates is provided later in the chapter.

The minimize and maximize constraints contain subcategories, resulting in the following classification:

- equal to
- not equal to
- less than
- greater than
- minimize
    - unbounded minimize
    - lower bounded minimize
    - upper bounded minimize
- maximize
    - unbounded maximize
    - lower bounded maximize
    - upper bounded maximize

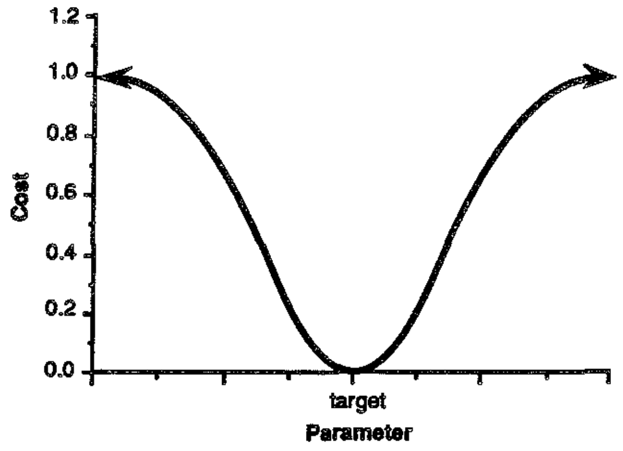


Figure 5.6: equal to

5.4.2.1. Equal to (=)

The *equal to* function has a value of zero when the parameter equals the target. The cost increases smoothly based on the distance from the target. As the distance from the target approaches infinity, the cost approaches the maximum value of one.

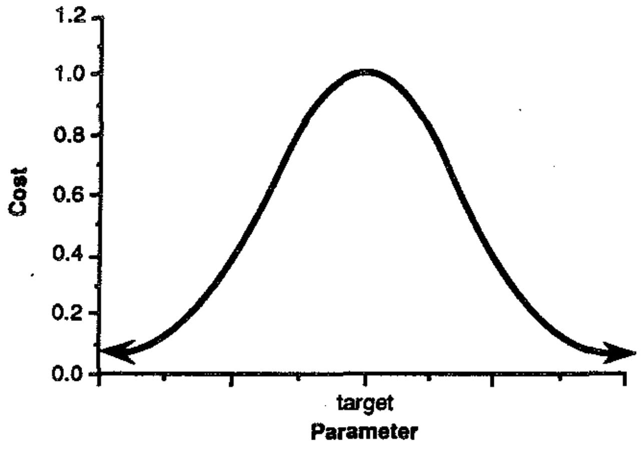


Figure 5.7: not equal to

5.4.2.2. Not equal to (≠)

The *not equal to* function has a maximum value of one when the parameter equals the target. The cost decreases smoothly based on the distance from the target. As the distance from the target approaches infinity, the cost approaches the minimum value of zero.

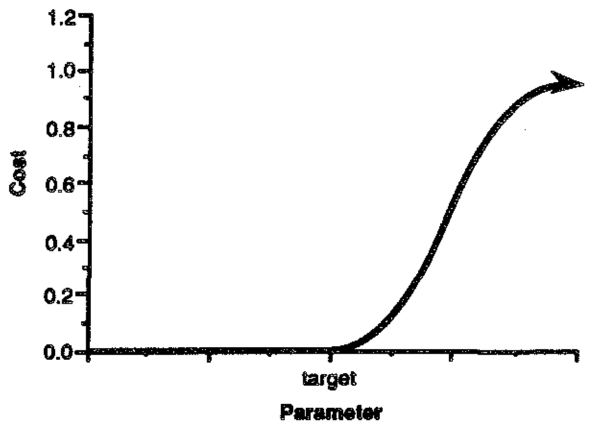


Figure 5.8: less than

5.4.2.3. Less than (<)

The *less than* constraint is completely satisfied when the parameter is less than the target, and consequently has a cost of zero for that range. For values greater than the target, the cost increases smoothly based on the distance from the target. As the distance from the target approaches infinity, the cost approaches the maximum value of one.

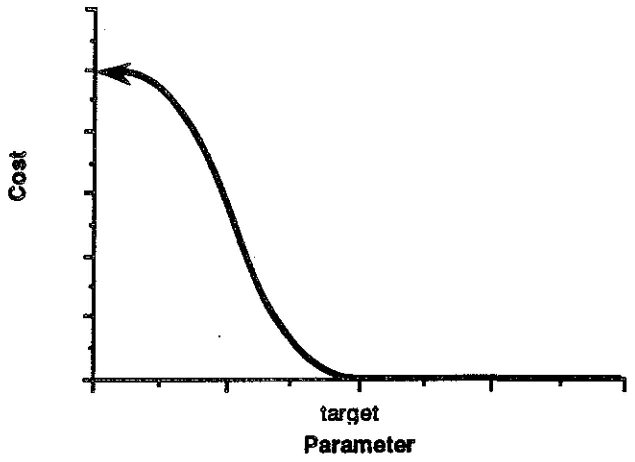


Figure 5.9: greater than

5.4.2.4. Greater than (>)

The *greater than* constraint is completely satisfied when the parameter is greater than the target, and consequently has a cost of zero for that range. For values less than the target, the cost increases smoothly based on the distance from the target. As the distance from the target approaches infinity, the cost approaches the maximum value of one.

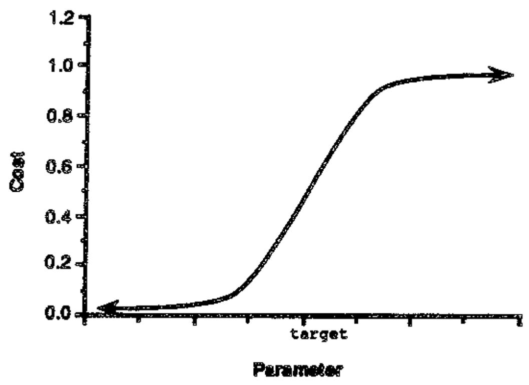


Figure 5.10: minimize

5.4.2.5. Minimize

5.4.2.5.1. Unbounded minimize

The *minimization* constraint can never be satisfied completely. It approaches a minimum value of zero as the parameter approaches negative infinity, and approaches a maximum value of one as the parameter approaches infinity. The target for this class of constraint is not the expected value of the parameter, but rather a specification of where to center the steepest portion of the curve.

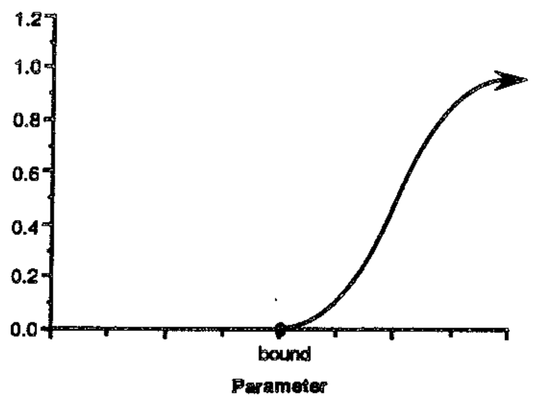


Figure 5.11: lower bounded minimize

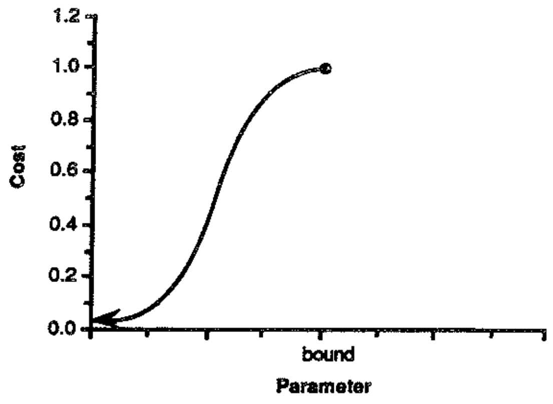


Figure 5.12: upper bounded minimize

5.4.2.5.2. Bounded minimize

There may exist lower or upper bounds on values of the parameter being minimized. For instance, a volume parameter might have a lower bound of zero, since a negative volume is meaningless. Upper bounds are less common, but can be derived from limits in the object definition or firm restrictions imposed by the world holding the object.

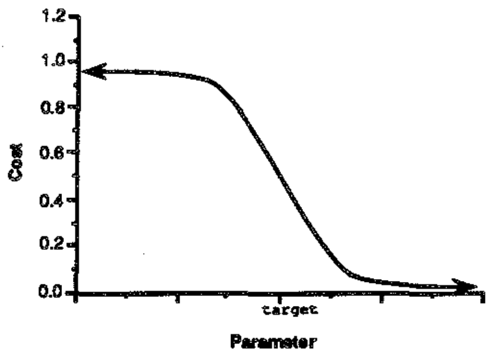


Figure 5.13: maximize

5.4.2.6. Maximize

5.4.2.6.1. Unbounded maximize

Like the minimization constraint, the *maximization* constraint can never be satisfied completely. It approaches a minimum value of zero as the parameter approaches infinity, and approaches a maximum value of one as the parameter approaches negative infinity. As with the minimization constraint, the target for this class of constraint is not the expected value of the parameter, but rather a specification of where to center the steepest portion of the curve.

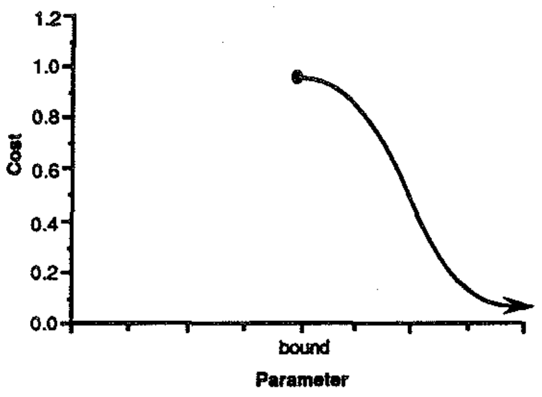


Figure 5.14: lower bounded maximize

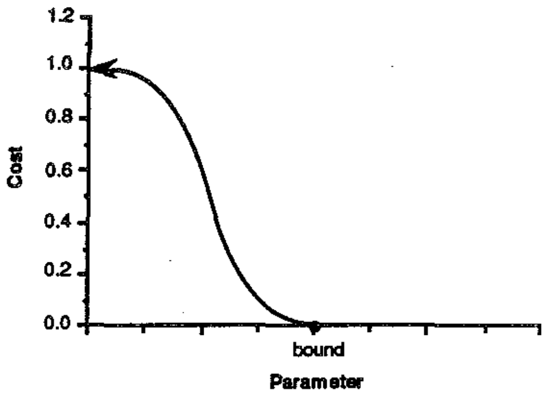


Figure 5.15: upper bounded maximize

5.4.2.6.2. Bounded maximize

As with the minimization constraint, lower and upper bounds may exist on a parameter being maximized. The lower bounded maximum has a value of one at the lower bound, and asymptotically approaches zero as the parameter increases. The upper bound maximum has a value of zero at the upper bound, and asymptotically approaches one as the parameter decreases.

5.4.3. Qualities of cost functions

This section discusses the properties of cost functions for objective constraints. These properties implicitly specify which classes of functions can be used as cost functions. A later section will show how additional properties can be used to explicitly select functions within those classes.

5.4.3.1. *Areas of interest*

Cost functions should have a broad extent as previously described and illustrated in figure 5.2. A broad extent implies a small slope, which implies little difference in cost between neighboring solutions. On the other hand, the goal of the cost function is to distinguish between various alternatives, and this can be accomplished most effectively if the slope is large. These conflicting goals can both be addressed by varying the slope so that it is large (in magnitude) in areas of interest and small in other areas.

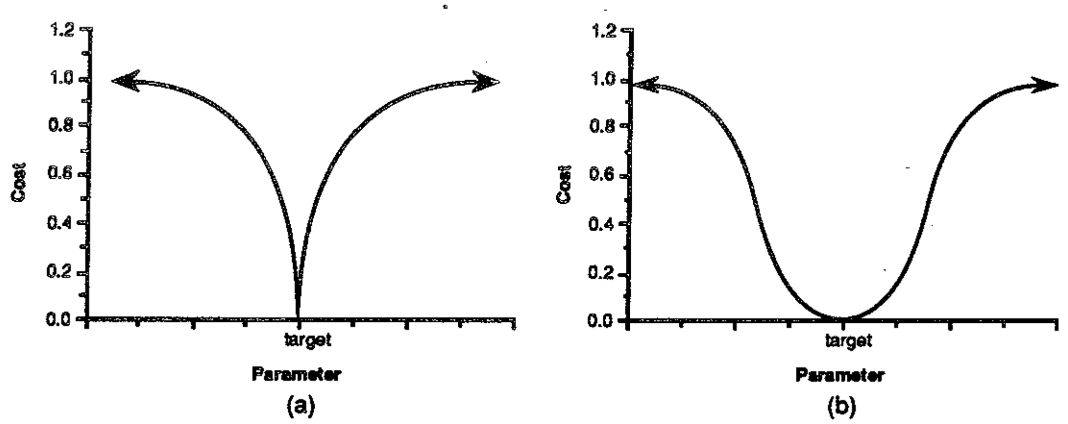


Figure 5.16: cost functions with different tolerances

Figure 5.16a shows a cost function in which primary importance is placed on distinguishing between alternatives near the target value. Because the slope is large near the target, even small deviations from the target can result in a large cost. Beyond a certain point, however, the cost tends to level out. Little distinction is made between parameters that are far from the target value. Such a function emphasizes the importance of reaching the target value exactly, and hence is appropriate in an application desiring precise specification of parameter values. An engineering problem where the parameter can affect the operation or cost of an assembly is one example of such a task. If we need a piston to be exactly two millimeters in diameter smaller than a corresponding cylinder, then any deviation is likely to result in an inefficient or non-functioning engine.

Figure 5.16b shows a cost function with a different emphasis. The slope is small near the target, so parameters near the target value have very low cost. Beyond a small neighborhood the slope increases and results in much greater costs for similar parameter increments. Eventually another threshold is reached, however, and the slope once again decreases, with a corresponding decline is cost differential. This function says, in effect, that close enough is okay. No harm is done by missing the target by a small amount, but missing by a large amount can be very costly. This function is suitable for constraints which are guidelines rather than hard-and-fast rules. A landscape guideline on placement of shrubbery is a good example. If we say that we want a bush placed 5.0 feet from a building, then it may be entirely satisfactory if it is 4.8 feet or 5.2 feet away, but entirely unsatisfactory if it is one foot away (where it might block a window), or ten feet away (where it might block a walkway).

5.4.3.2. Bound on function range

The difference between the minimum and maximum costs of a function defines the importance of the function relative to other functions. The true minimum and maximum costs are determined by the values assumed during the constraint satisfaction process. If variables are restricted (apart from constraint specification) to lie in a certain range, then the cost function range may be restricted as well. The cost functions described later in this chapter range from zero to one, providing a cap on the importance of each constraint and a standard range prior to the assignment of weighting factors.

A function with no upper bound does not present a particular problem because the goal is to minimize the cost function, and the unbounded region of the function will be eliminated from further consideration at early stages in the constraint minimization search. On the other hand, a cost function with no lower bound presents a problem. Because the search procedure will concentrate on finding the global minimum, it will attempt to satisfy this unbounded minimum cost function to the exclusion of all other constraints.

5.4.3.3. Monotonically increasing or decreasing cost

Though not a requirement for cost functions in general, this property holds for the six basic categories of constraints. The cost must be monotonically increasing or decreasing as the parameter moves away from the target. For example, given a constraint to minimize the area of a surface, a surface with area $2x$ will always have a greater cost than a surface with area x. This rules out oscillating functions which might otherwise satisfy all guidelines.

5.4.3.4. *User control over shape*

The user must have intuitive control over the shape of the function. This can be accomplished by explicitly setting coefficients of a simple function, or by setting more natural parameters which are converted to function coefficients.

5.4.4. Examination of candidate cost functions for $=, \neq, <, >$

The goal of this section is to find cost functions meeting the properties presented in the previous section. For the purpose of this discussion, only one of the basic constraint categories (*not equal to*) will be examined. The remaining categories (*equal to, less than, greater than*) are all based on simple reflections of the *not equal to* function.

The previous section introduced the notions of area of interest and natural user control of shape. The area of interest can be roughly specified by indicating a center point and width of the area. A logical center point is where the cost is halfway between the minimum and maximum cost. The width of the area of interest (i.e., how quickly the transition between low cost and high cost occurs) can be provided by specifying the slope at the center point.

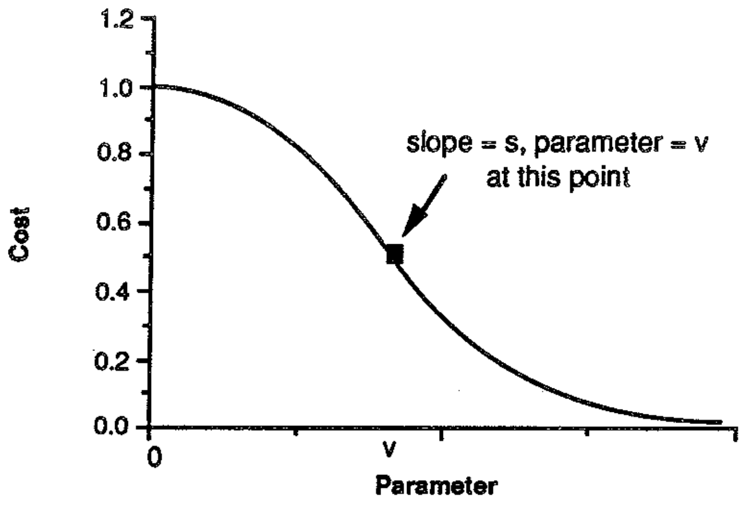


Figure 5.17: user shape specification

Figure 5.17 illustrates the user specification of shape. Let us call the center of the area of interest v, and the slope at that point s. The rules governing the shape of the cost function can then be summarized as follows:

1. $f(v) = 1/2$
2. $f'(v) = s$
3. $f(0.0) = 1.0$
4. $\lim_{x \to \infty} f(x) = 0.0$

The four candidate function categories examined were: $f(x) = e^{-x^2/2\sigma^2}$, $f(x) = e^{-nx^b}$, piecewise cubics, and $f(x) = 1/(nx^b+1)$. The first function, a Gaussian, was examined based on its ability to represent functions with the general shape of figure 5.16b. When it was found to be too specific to meet the four rules above, a more general form of the function, $f(x) = e^{-nx^b}$, was examined and found suitable. Piecewise cubics can be used to approximate many functions, so they were also examined, and a form suitable for cost function representation was derived. The final function class, $f(x) = 1/(nx^b+1)$, was examined because it is capable of providing a steep slope near $x=0$, which is necessary for cost functions with a low tolerance. It also was found suitable. The analysis is presented below.

5.4.4.1. Gaussian: $f(x) = e^{-x^2/2\sigma^2}$

The function illustrated in figure 5.16b exhibits the characteristics of an inverted normal curve. Consequently, functions of the following form were examined:

$$f(x) = e^{-x^2/2\sigma^2}. \tag{5.2}$$

Since the σ term has no meaning here other than as a constant, we can replace $1/2\sigma^2$ by the constant n, yielding

$$f(x) = e^{-nx^2}. \tag{5.3}$$

The goal is to find n, given any s and v, such that

$$f(v) = e^{-nv^2} = \frac{1}{2} \tag{5.4}$$

and

$$f'(v) = (-2nv)\, e^{-nv^2} = s. \tag{5.5}$$

The problem is that either s or v is sufficient to define n. For instance, if we solve equation 5.4 for n, we get

$$n = \frac{\ln 2}{v^2}. \tag{5.6}$$

This value of *n* completely defines the slope of the function, hence we cannot set *s* and *v* independently. This function class is unsuitable for cost function representation. A more general function is needed to allow control of both slope and value.

5.4.4.2. $f(x) = e^{-nx^b}$

This function class is a generalization of the Gaussian class. The goal now is to find *n* and *b*, given any *s* and *v*, so that

$$f(v) = e^{-nv^b} = \frac{1}{2} \tag{5.7}$$

and

$$f'(v) = e^{-nv^b}\left(-bnv^{b-1}\right) = s. \tag{5.8}$$

We can solve for *n* in the first equation, yielding

$$n = \frac{ln\, 2}{v^b}. \tag{5.9}$$

Furthermore,

$$f'(v) = f(v)\left(-bnv^{b-1}\right) = s, \tag{5.10}$$

so

$$s = \frac{1}{2}\left(-bnv^{b-1}\right). \tag{5.11}$$

Substituting for *n*, we get

$$s = \frac{-b\, ln\, 2}{2v^b}\, v^{b-1}. \tag{5.12}$$

Solving for *b* gives

$$b = \frac{-2vs}{ln\, 2}, \tag{5.13}$$

and substituting *b* in equation 5.9 gives

$$n = \frac{ln\, 2}{v^{-2vs/ln2}}\ . \tag{5.14}$$

Since we can express *n* and *b* in terms of *s* and *v*, this class of functions is a suitable candidate for cost function representation.

5.4.4.3. *Piecewise cubic:* $f(x) = ax^3 + bx^2 + cx + d$

Rather than seeking a single function to match some criteria, we can use portions of two or more simpler functions. For example, the curve presented in figure 5.17 can be split into two parts, with the following constraints defining each part:

Part 1	Part 2
1. $f(0.0) = 1.0$	1. $f(v) = 0.5$
2. $f'(0.0) = p$	2. $f'(v) = s$
3. $f(v) = 0.5$	3. $\lim_{x \to \infty} f(x) = 0.0$
4. $f'(v) = s$	4. $\lim_{x \to \infty} f'(x) = 0.0$

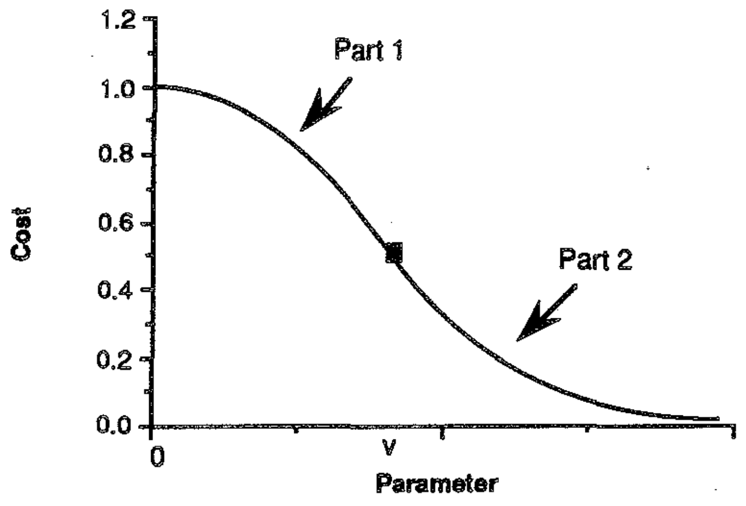


Figure 5.18: piecewise construction

The four constraints of part one define a cubic polynomial. Constraints three and four of part two define a curve which asymptotically approaches $f(x)=0.0$. Unfortunately, a cubic polynomial cannot represent such a curve. The solution is to use another function for part two, or to settle for a function without an infinite extent. Since the alternate functions for part two would be similar to the functions considered elsewhere in this section, we will only examine the latter case. As an alternative to the function asymptotically approaching $f(x)=0.0$, we can choose a value k at which $f(k)=0.0$, and a slope m at the same point as shown in figure 5.19.

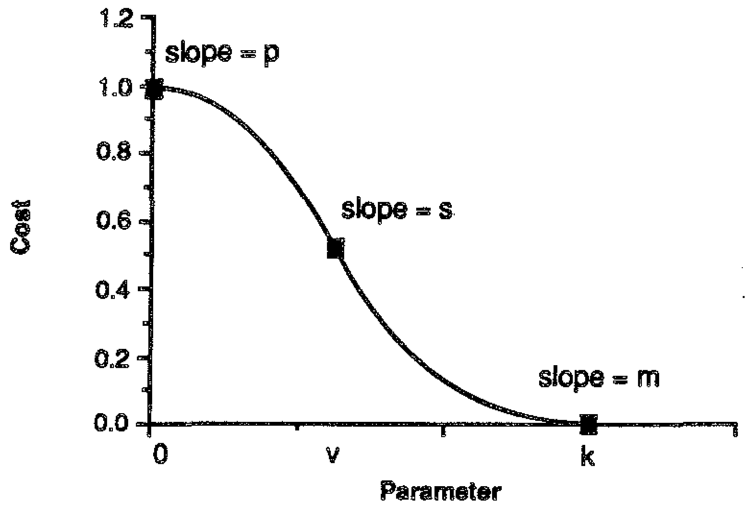


Figure 5.19: piecewise function with finite extent

The modified sets of constraints for this alternative piecewise polynomial are:

Part 1	Part 2
1. $f(0.0) = 1.0$	1. $f(v) = 0.5$
2. $f'(0.0) = p$	2. $f'(v) = s$
3. $f(v) = 0.5$	3. $f(k) = 0.0$
4. $f'(v) = s$	4. $f'(k) = m$

Since $f(x) = ax^3 + bx^2 + cx + d$, and $f'(x) = 3ax^2 + 2bx + c$, these sets of constraints define the following sets of equations:

Part 1	Part 2
1. $d = 1.0$	1. $av^3+bv^2+cv+d = 0.5$
2. $c = p$	2. $3av^2+2bv+c = s$
3. $av^3+bv^2+cv+d = 0.5$	3. $ak^3+bk^2+ck+d = 0.0$
4. $3av^2+2bv+c = s$	4. $3ak^2+2bk+c = m$

The equations of part 1 are easily solved to yield the following values for *a*, *b*, *c*, and *d:*

$$a = (1+sv+pv)/v^3$$
$$b = (-1.5-sv-2pv)/v^2$$
$$c = p$$
$$d = 1.0$$

This gives

$$f_{part1}(x) = \frac{(1+sv+pv)}{v^3}x^3 + \frac{(-1.5-sv-2pv)}{v^2}x^2 + px + 1.0. \tag{5.15}$$

The equations of part 2 are conceptually easy to solve, but application of a solution method such as Gaussian elimination results in extremely complex terms. An equivalent approach to solving these constraints is to find the coefficients for a translation of the function and then convert from these coefficients to the original equation. In this case, we can solve for $g(x) = f(x-v)$, as shown in figures 5.20 and 5.21. As a further notational simplification, let $z=k-v$.

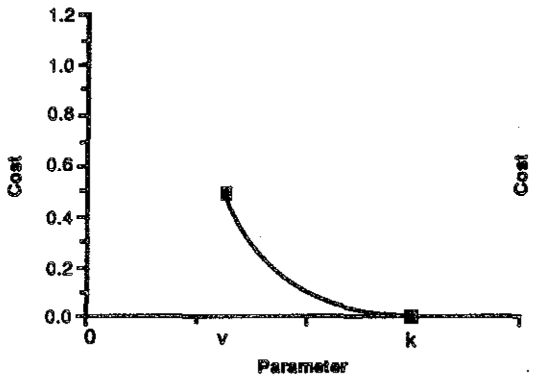


Figure 5.20: f(x)

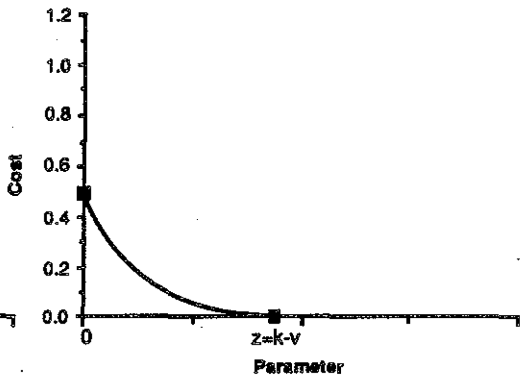


Figure 5.21: g(x) = f(x-v)

We now wish to find *A*, *B*, *C*, and *D*, where:

$$g(x) = Ax^3 + Bx^2 + Cx + D$$

and

$$g'(x) = 3Ax^2 + 2Bx + C.$$

The constraints for this polynomial are:

1. $g(0.0) = 0.5$
2. $g'(0.0) = s$
3. $g(z) = 0.0$
4. $g'(z) = m$.

This set of constraints defines the following set of equations:

1. $D = 0.5$
2. $C = s$
3. $Az^3 + Bz^2 + Cz + D = 0.0$
4. $3Az^2 + 2Bz + C = m$.

Solving for A, B, C, and D, we find:

$$A = (1+sz+zm)/z^3$$
$$B = (-1.5-2sz-zm)/z^2$$
$$C = s$$
$$D = 0.5.$$

From $f(x) = g(x-v)$, we get:

$$\begin{aligned} f(x) &= A(x-v)^3 + B(x-v)^2 + C(x-v) + D \\ &= A(x^3 - 3vx^2 + 3v^2x - v^3) + B(x^2 - 2vx + v^2) + C(x-v) + D \\ &= x^3A + x^2(-3Av + B) + x(3Av^2 - 2Bv + C) + (-Av^3 + Bv^2 - Cv + D). \end{aligned}$$

So,

$$a = A = \frac{1 + sz + zm}{z^3}$$

$$b = -3Av + B = -3\left(\frac{1+sz+zm}{z^3}\right)v + \frac{-1.5-2sz-zm}{z^2}$$

$$c = 3Av^2 - 2Bv + C = 3\left(\frac{1+sz+zm}{z^3}\right)v^2 - 2\left(\frac{-1.5-2sz-zm}{z^2}\right)v + s$$

and

$$d = -Av^3 + Bv^2 - Cv + D = -\left(\frac{1+sz+zm}{z^3}\right)v^3 + \left(\frac{-1.5-2sz-zm}{z^2}\right)v^2 - sv + 0.5.$$

Finally, we can substitute *(k-v)* for *z*, giving:

$$a = \frac{1 + (s+m)(k-v)}{(k-v)^3}$$

$$b = -3\left(\frac{1+(s+m)(k-v)}{(k-v)^3}\right)v + \frac{-1.5+(-2s-m)(k-v)}{(k-v)^2}$$

$$c = 3\left(\frac{1+(s+m)(k-v)}{(k-v)^3}\right)v^2 - 2\left(\frac{-1.5+(-2s-m)(k-v)}{(k-v)^2}\right)v + s$$

$$d = -\left(\frac{1+(s+m)(k-v)}{(k-v)^3}\right)v^3 + \left(\frac{-1.5+(-2s-m)(k-v)}{(k-v)^2}\right)v^2 - sv + 0.5\ .$$

Thus, $f_{part2}(x) = ax^3 + bx^2 + cx + d$, where *a, b, c,* and *d* are listed above. This piecewise function does not have infinite extent, but it is suitable for cost function representation because the user has significant control over its shape. Although these terms are complex, the coefficients *a, b, c,* and *d* need only be computed once for each cost function.

5.4.4.4. $f(x) = \frac{1}{nx^b+1}$

This function class can be used in situations where error tolerance is low, since it can provide a steep slope near $x = 0$, and approach a slope of zero as x approaches ∞. It was informally derived by starting with a simple function, $f(x) = 1/x$, with the desired slope properties, and generalizing so that other constraints, $f(0.0)=1.0$, $f(v)=0.5$, and $f'(v)=s$, could be satisfied. The goal is still to find *n* and *b*, given any *s* and *v*, so that

$$f(v) = \frac{1}{nv^b+1} = \frac{1}{2} \tag{5.16}$$

and

$$f'(v) = \frac{-bnv^{b-1}}{(nv^b+1)^2} = s. \tag{5.17}$$

From equation 5.16, we note that $nv^b=1$, yielding

$$n = \frac{1}{v^b}. \tag{5.18}$$

Substituting, we get

$$s = \frac{-b\left(\frac{1}{v^b}\right) v^{b-1}}{\left(\left(\frac{1}{v^b}\right) v^b + 1\right)^2}, \tag{5.19}$$

or

$$s = \frac{-b/v}{(1+1)^2}. \tag{5.20}$$

Solving for b gives

$$b = -4vs. \tag{5.21}$$

Substituting for b in equation 5.18 gives

$$n = \frac{1}{v^{-4vs}} = v^{4vs}. \tag{5.22}$$

Once again we can express n and b in terms of s and v, so this class of functions is also a suitable candidate for cost function representation. This function class is depicted by figure 5.16a.

5.4.4.5. Abstraction of user control

With the exception of the simple Gaussian, each of the above functions can be used to represent cost functions. Rather than dealing with non-intuitive function parameters, the above analysis demonstrates that intuitive properties may be converted to function parameters.

The general exponential, $f(x) = e^{-nx^b}$, and inverse polynomial, $f(x) = 1/(nx^b+1)$, provide simple control through only two parameters, *slope* and *value*, at the point where $f(x)=0.5$. While these parameters provide a certain level of abstraction, they still refer to the function explicitly, which assumes some knowledge of the problem representation. Fortunately, these parameters can be abstracted into two parameters, *sensitivity* and *tolerance*, which require no knowledge of the underlying representation.

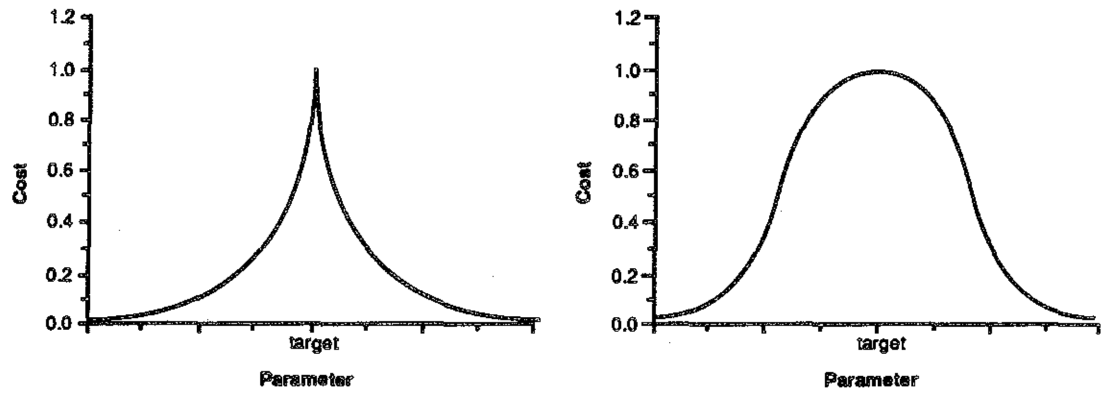
The *sensitivity* parameter controls how rapidly the transition is made from high cost values to low cost values, which conveniently is the property controlled by the *slope* parameter. Sensitivity ranked on a scale from 0 to 10 can easily be converted to slope from 0 to infinity. Similarly, the *tolerance* parameter indicates the distance (from the target) at which the cost is halfway between the maximum and minimum costs. If we express tolerance in these terms, details of the underlying function are hidden from the designer, while providing intuitively meaningful parameters for manipulation by the user.

Of the three suitable candidates, the piecewise cubic representation provides the greatest control over the shape of the function. In addition to slope and value at $f(x)=0.5$, the designer also has control over the value at which $f(x)=0.0$, and the slope at $x=target$ and $f(x)=0.0$. For this representation, a graphical function representation is probably simpler to comprehend than a list of abstract properties.

5.4.4.6. Cost functions: =, ≠, <, >, bounded minimize and maximize

The candidate cost functions were defined above for the half-plane *parameter>0* with the assumption that the target parameter was zero. These base functions must be defined for parameters less than zero, and provision must be made to center the function about an arbitrary target. For *parameter<target*, we want *f(parameter)=f(-parameter).* Given a function *f(x)*, the translation and reflection can be accomplished by:

$$g(x) = f(|x\text{-}target|) \tag{5.23}$$



$g(x)$, *where* $f(x) = 1/(nx^b+1)$ $g(x)$, *where* $f(x) = e^{-nx^b}$

Figure 5.22: general form of g(x) given f(x)

The form of *g(x)* for the bounded minimize and maximize functions is identical with the exception that the target is replaced by the bound:

$$g(x) = f(|x\text{-}bound|) \tag{5.24}$$

Moreover, given *g(x)*, we can express the cost functions for each category as follows:

$$\text{"}x = target\text{"}: cost(x) = -g(x)+1$$

$$\text{"}x \neq target\text{"}: cost(x) = g(x)$$

$$\text{"}x < target\text{"}: cost(x) = \begin{cases} 0, \text{ if } x<target \\ -g(x)+1, \text{ otherwise} \end{cases}$$

$$\text{"}x > target\text{"}: cost(x) = \begin{cases} 0, \text{ if } x>target \\ -g(x)+1, \text{ otherwise} \end{cases}$$

$$\text{"}minimize\ x, x \geq bound\text{"}: cost(x) = \begin{cases} undefined, \text{ if } x<bound \\ -g(x)+1, \text{ otherwise} \end{cases}$$

$$\text{"}minimize\ x, x \leq bound\text{"}: cost(x) = \begin{cases} undefined, \text{ if } x>bound \\ g(x), \text{ otherwise} \end{cases}$$

$$\text{"}maximize\ x, x \geq bound\text{"}: cost(x) = \begin{cases} undefined, \text{ if } x<bound \\ g(x), \text{ otherwise} \end{cases}$$

$$\text{"}maximize\ x, x \leq bound\text{"}: cost(x) = \begin{cases} undefined, \text{ if } x>bound \\ -g(x)+1, \text{ otherwise} \end{cases}$$

Note that *cost(x)* returns values between *0.0* and *1.0*, and is based on the assumption that *f(0.0)=1.0*, with *f(x)* decreasing as *x* increases.

5.4.5. Cost functions for unbounded *minimize* and *maximize*

The constraints *equal to*, *not equal to*, *less than*, and *greater than*, as well as the bounded versions of *minimize* and *maximize*, can each be completely satisfied. This is not the case for the unbounded *minimize* and *maximize* constraints. For example, the unbounded minimize constraint can never be satisfied, because for any parameter value there is always another lesser value, with a corresponding smaller cost. Similarly, greater values must have greater costs.

The goal of designing suitable cost functions for unbounded minimize and maximize is complicated by the fact that although parameter values are infinite, the range of function values should be bounded (to conform to the guideline of costs between 0.0 and 1.0). An asymptotic approach to both the minimum and maximum values of the function is therefore required.

For the minimize and maximize cost functions to be of any use, they must be capable of distinguishing between the actual values which occur during the constraint satisfaction process. This is accomplished by having the majority of the transition from *f(x)=0.0* to *f(x)=1.0* occur in the region of interest. As with previous cost functions, the region of interest may be centered about a point, with its width specified by the slope at that point.

5.4.5.1. The minimize cost function

If we temporarily consider the function centered about *x=0.0*, the following constraints define *f(x)*:

1. $\lim_{x \to \infty} f'(x) = 0.0$
2. $\lim_{x \to -\infty} f'(x) = 0.0$
3. $f'(0.0) = s$
4. $\lim_{x \to \infty} f(x) = 1.0$
5. $\lim_{x \to -\infty} f(x) = 0.0$

The slope, *s*, must be greater than zero. Constraints 1, 2, and 3 can be satisfied by a function of the form:

$$f'(x) = \frac{s}{1 + h(x)} \tag{5.25}$$

where *h(0.0)=0.0* and *h(x)* increases with *x*. If we let *h(x)=x*, then

$$f'(x) = \frac{s}{1 + x} \tag{5.26}$$

and

$$f(x) = \int \frac{s}{1 + x}\, dx = s\, ln\left|\frac{1}{1 + x}\right| + C. \tag{5.27}$$

Unfortunately, this function grows without bound as *x* approaches infinity, violating constraint 4. However, if we let $h(x)=x^2$, then

$$f(x) = \int \frac{s}{1+x^2}\,dx = s\,arctan\,(x) + C. \tag{5.28}$$

Since *arctan (x)* ranges from *-π/2* to *π/2*, this function ranges from *0.0* to *sπ*, when C=π/2. We want the function to range from *0.0* to *1.0*. We cannot simply scale the function by *1/sπ*, since that would affect the value of the slope. However, because

$$\int \frac{1}{a^2+x^2}\,dx = \frac{1}{a}\,arctan\left(\frac{x}{a}\right) + C, \tag{5.29}$$

we can use a function with derivative form

$$f'(x) = \frac{sa^2}{a^2+x^2} \tag{5.30}$$

to meet all constraints. If we integrate, we get

$$f(x) = \int \frac{sa^2}{a^2+x^2}\,dx = sa\,arctan\,\left(\frac{x}{a}\right) + C. \tag{5.31}$$

To provide the proper scaling, we need *a=1/sπ*, and *C=0.5*, so

$$f(x) = \frac{1}{\pi}\,arctan\,(s\pi x) + 0.5. \tag{5.32}$$

This form centers the function about *x=0.0*. To center the function about an arbitrary target, let

$$f(x) = \frac{1}{\pi}\,arctan\,(s\pi\,(x\text{-}target)) + 0.5. \tag{5.33}$$

If the cost function designer requires constraints that are not satisfied by this function, then an alternate approach, such as a piecewise polynomial, may be used. This function class is illustrated in figure 5.10.

5.4.5.2. The maximize cost function

The maximize function is very similar to the minimize cost function, with the exception that the cost approaches the minimum cost as the parameter gets larger. The constraints defining this function are:

1. $\lim_{x \to \infty} f'(x) = 0.0$
2. $\lim_{x \to -\infty} f'(x) = 0.0$
3. $f'(0.0) = s$
4. $\lim_{x \to \infty} f(x) = 0.0$
5. $\lim_{x \to -\infty} f(x) = 1.0$

The slope, *s*, must be negative. These constraints may be satisfied by *f(x)* as specified for the minimize function. The reflection is accomplished by setting *s* less than zero.

5.4.6. Summary of cost function design for objective constraints

The design procedure for objective functions can be viewed as a series of six steps:

> ***Step 1***: Most constraints will already be expressed as one of the six basic constraint types: (=, ≠, <, >, minimize, and maximize). If the constraint is not in this form, it should be converted to this form.
>
> ***Step 2***: Select the main parameter which represents the property being constrained.
>
> ***Step 3***: Choose the basic type of constraint being implemented. CONTEST supports the six basic types, with both the unbounded and bounded forms of minimize and maximize.
>
> ***Step 4***: Select a function template to be used to approximate the actual cost function. CONTEST supports templates based on the the type of constraint and the functions $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$.

Step 5: Select the parameters which define the shape of the particular function. CONTEST supports the specification based on the value where *cost=0.5* (*v* in the discussion above) and the slope at that point (*s* in the discussion above).

Step 6: Choose a scaling factor for the constraint. The normal range of a cost function is from 0.0 to 1.0. The user may select an arbitrary scaling factor to convert to a range of 0.0 to *scaleFactor.*

These six steps result in a cost function which may be further scaled and then summed with other objective, subjective, and search constraints to yield the global cost function.

5.4.7. Analysis of cost functions of objective constraints

Section 5.3 presented several guidelines for cost function design. This section analyzes the cost function templates, $f(x) = e^{-nx^b}$, $f(x) = 1/(nx^b+1)$, and $f(x)$=*piecewise cubic (2 pieces),* based on these guidelines. Only the guidelines which distinguish between these templates are presented below; other guidelines are satisfied equally by each template. For the first two templates, we consider the cases for which a value v and slope s are used to specify function coefficients, in addition to the cases where *sensitivity* and *tolerance* are used to specify these coefficients. We also consider the piecewise cubic made up of more than two pieces. Such a piecewise cubic might be specified using an interactive graphics program.

5.4.7.1. Functions should be easy to evaluate

For the purpose of this analysis, we define an abstract model of computation with the following costs: an arithmetic function (+, -, *, /) or comparison requires one time unit, and an exponential evaluation requires a function call at an expense of 50 time units. Furthermore, we assume a straightforward implemention of the function (no nested evaluations for polynomials, no table lookups, etc.).

With this model of computation, the function $f(x) = e^{-nx^b}$ requires 101 time units per evaluation (two exponentials and one multiplication). The function $f(x) = 1/(nx^b+1)$ requires 53 time units per evaluation (one exponential, one multiply, one add, and one divide). A piecewise cubic, for n pieces, requires $log_2 n$ tests to determine the particular cubic function to evaluate, followed by six multiplies and three additions to evaluate the cubic, for a total of $log_2 n + 9$ time units. The piecewise cubic is the clear winner here, based on the lack of a function call.

These estimates do not include the potentially expensive cost of recomputing the parameter x for each iteration.

5.4.7.2. Shape should be accurate: theoretical considerations

In this section, we estimate the flexibility of the function template: i.e., the ability of the template to match a variety of target functions. Note that this flexibility may not be necessary if all desired functions closely match available templates, even if these templates are inflexible.

All forms of the templates $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$ have a low degree of flexibility because the user has control over only two parameters: either slope s and value v for the point at which $f(v)=0.5$, or the sensitivity and tolerance parameters. The slope and value and $x=0.0$ and $x=\infty$ are predetermined.

The two-piece cubics have a medium degree of flexibility because they also allow, in addition the s and v parameters, the specification of the slope at $x=0.0$ and at $f(x)=0.0$.

Finally, general piecewise cubics have a high degree of flexibility because they allow an arbitrarily close approximation to any function.

5.4.7.3. Shape should be accurate: practical considerations

In this section, we assume that a function template is suitable for estimating a particular function, and we analyze the ease with which template parameters can be selected in order to best match that function.

The slope and value versions of the templates $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$ have a medium degree of difficulty in matching: the slope and value provide a good starting point for matching the target, but some tweaking may be necessary to find the best overall fit. The sensitivity and tolerance versions of the same templates involve an intermediate stage to convert to slope and value parameters. For a designer who understands the underlying function representation, this intermediate stage presents an additional hurdle on the path to accurate function approximation.

The two-piece cubic has a low degree of difficulty in matching. Three slopes and one value from the target function provide an accurate characterization of the intended function. The n-piece cubic has an even higher ability to match a particular function.

5.4.7.4. Shape should be easy to specify

This section examines the ease by which any particular approximation may be specified. It does not address the issue of the accuracy of this approximation.

The slope and value versions of the templates $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$ have a medium ease of specification: they require some mental or physical (pencil and paper) sketching to estimate the proper parameters. The sensitivity and tolerance versions of the same templates have a high ease of specification: they require only a general numeric ranking of two parameters.

The two-piece cubic has a low ease of specification. Physical sketching is most likely required to derive the parameters used to specify the function. The n-piece cubic has an even lower ease of specification. Some computer assistance, such as an interactive graphics program, is needed to quickly find the required parameters.

5.4.7.5. Specification should be separate from representation

A specification method is defined as having a high degree of separation from the underlying representation if knowledge of the representation is not necessary to define the function. A method is defined as having a medium degree of separation from the underlying representation if the method strongly suggests the underlying representation, but complete understanding of that representation is not necessary to successfully specify the function. A method is defined as having a low degree of separation from the underlying representation if a thorough understanding of the representation is necessary to successfully specify the function.

Based on these criteria, the slope and value versions of the templates $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$ have a medium degree of separation from the underlying representation. These versions require some information about a particular point (where $f(v)=0.5$), but a complete understanding of the form of the function is not necessary.

The sensitivity and tolerance versions of the same templates have a high degree of separation from the underlying representation. The cost function designer can specify these parameters without any knowledge of the internal representation.

All forms of piecewise cubics have a low degree of separation from the underlying representation. Information about various boundary conditions requires that the user have a firm grasp of the range of slopes and values throughout the entire cost function..

5.4.7.6. Functions should encourage hill-climbing

The term *hill-climbing* is once again used here to refer to search movement only in the direction of a local optimum (even though CONTEST seeks the minimum cost). Functions without a gradual descent to the minimum value, such as the simplistic cost function in figure 5.1, do not encourage hill-climbing.

All forms of the templates $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$ encourage hill-climbing throughout the entire parameter range. Piecewise cubics, on the other hand, cannot represent an asymptotic approach to a function value. Instead, they define a descent toward the optimum in a finite portion of the parameter range.

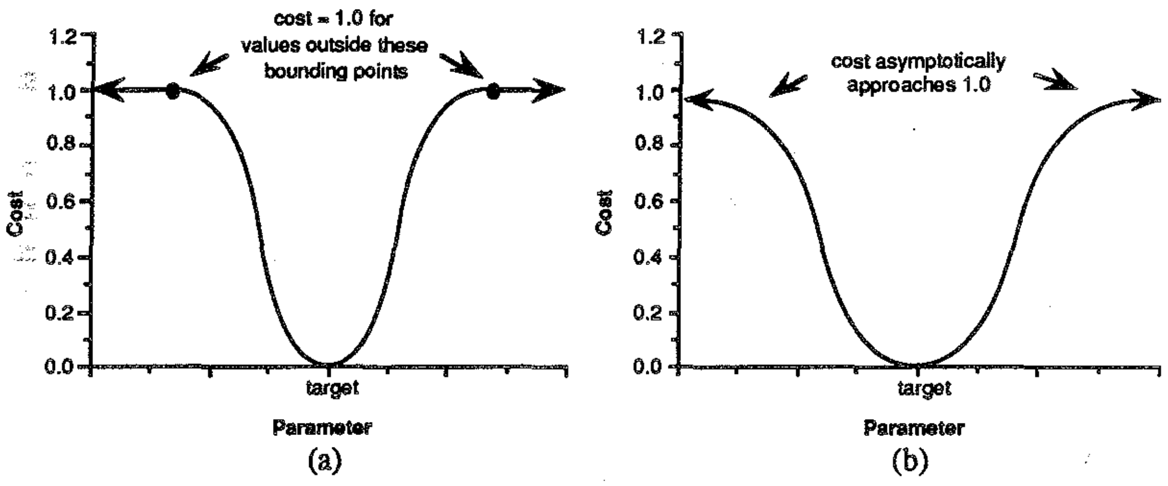


Figure 5.23: restricted vs. unrestricted hill-climbing

Figure 5.23 illustrates functions with a finite and infinite transition to the minimum. Figure 5.23a represents a function that might be represented by piecewise cubics, while 5.23b represents a function based on $f(x) = e^{-nx^b}$.

5.4.7.7. Summary of analysis of cost functions

	Time to evaluate	Ease of specification of a single function	Accuracy: theoretical representational ability	Accuracy: practical ease of getting what you want	Encourages hill-climbing	Separates specification from representation
$f(x) = e^{-nx^b}$ slope, value specification	high (101)	medium	low	medium	high	medium
$f(x) = e^{-nx^b}$ sensitivity, tolerance specification	high (101)	high	low	low	high	high
$f(x) = 1/(nx^b + 1)$ slope, value specification	medium (53)	medium	low	medium	high	medium
$f(x) = 1/(nx^b + 1)$ sensitivity, tolerance specification	medium (53)	high	low	low	high	high
f(x) = piecewise cubic (2 pieces)	low (10)	low	medium	high	medium	low
f(x) = piecewise cubic (n pieces)	low (9+log n)	very low	high	high	medium	very low

Table 5.1: summary of cost function analysis

Table 5.1 summarizes the strengths and weaknesses of each of the cost function representations. The overall goal is to approximate a cost function that exists in the designer's mind. The slope and value versions of the templates $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$ have been found satisfactory in practice. The sensitivity and tolerance versions of the same templates have not been implemented because CONTEST is still an experimental testbed, not an end-user product. The piecewise cubic form has not been used because the extra precision it provides compared to the other templates has not been necessary.

5.5. Cost functions for subjective constraints

Subjective constraints are the most difficult constraints to quantify because they involve value judgements. Computers are good at making objective evaluations, so the basic idea behind evaluating subjective constraints is to reduce them to a form that can be evaluated by a computer.

Subjective constraints involve issues such as: safety, reliability, durability, craftsmanship, simplicity of operation, ease of maintenance, and ease of manufacture. The guidelines for construction of subjective constraints are:

1. Use a divide-and-conquer approach to refine the problem
2. Base judgements on a simple scale
3. Match empirical data if constraints cannot be quantified

5.5.1. Divide and conquer to refine the problem

In a divide-and-conquer approach, problems are successively broken down into simpler problems until only trivial problems remain. In this case, subjective constraints are progressively expressed in terms of simpler constraints until the simpler constraints can be objectively evaluated. If necessary, constraints can be reduced to yes/no evaluations. The decisions on how to divide a constraint will of course be based on a particular set of values. There can be controversy about whether the criteria properly represent the constraint, but there can be no argument about the evaluation of the constraint once the criteria are established.

The divide-and-conquer procedure defines a tree structure in which terminal nodes represent basic evaluations, and internal nodes represent subjective constraints. An evaluation of every node in the tree can be created by assigning weights to the arcs, and having the terminal nodes return standard values (e.g. 0.0-1.0). Figure 5.24 provides an example in which a subjective constraint, "suitable for children", is broken down into simpler constraints. If each terminal node returns a value from 0.0 to 1.0, then the root node returns a value in the same range.

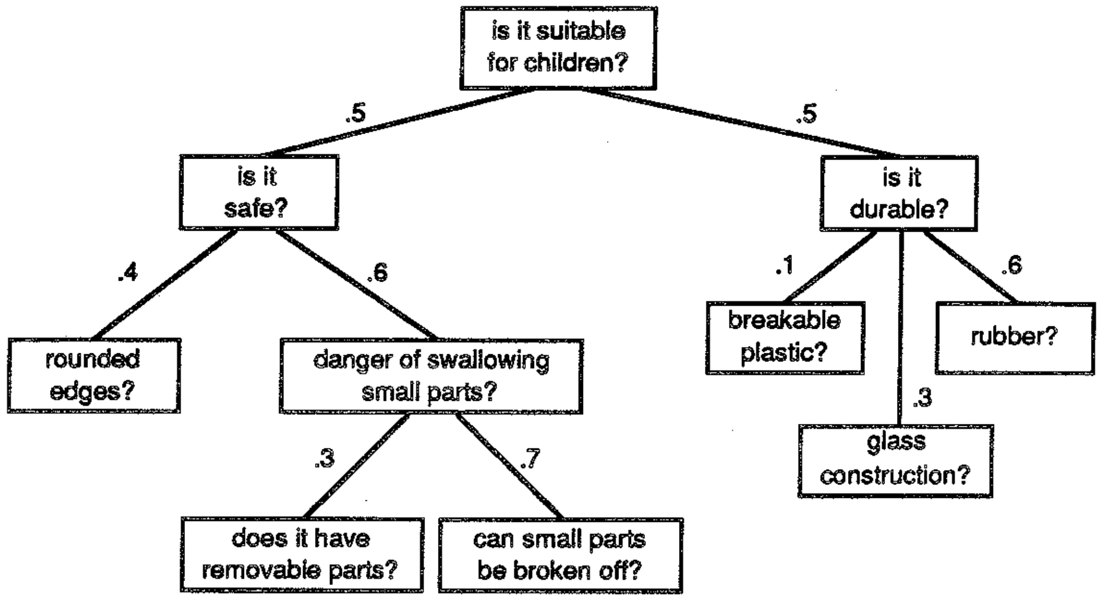

is it suitable for children?
.5
.5
is it safe?
is it durable?
.4
.6
.1
.6
rounded edges?
danger of swallowing small parts?
breakable plastic?
rubber?
.3
.3
.7
glass construction?
does it have removable parts?
can small parts be broken off?

Figure 5.24: divide-and-conquer approach

5.5.2. Base judgements on a simple scale

For value judgements, a common scale helps to reduce confusion and provides a logical way to compare pairs of constraints. Most people are familiar with ranking items on a scale from zero to ten. On such a scale, zero represents perfect satisfaction and ten represents complete failure. Any basic evaluation should be done on this scale. Its importance can then be specified by the weighting factors associated with each constraint.

5.5.3. Match empirical data if constraints cannot be quantified

If a constraint is too complex to be quantified, it may still be possible to evaluate it by comparing with various empirical data. For example, the designer of a ship hull may not have a sophisticated model of drag, but he may have relevant data for a variety of previously constructed designs. By comparing potential designs with these templates, promising configurations may be identified.

Similarly, a design may be compared to a particular design style even if it is not understood what makes that style successful. Koning and Eizenberg [Koning, 1981] were able to describe the style of Frank Lloyd Wright's prairie houses by defining a shape grammar. New prairie house designs can be parsed to see if they match the grammar. Design frequently involves drawing on previous work without necessarily understanding all of the tradeoffs involved in the previous work. Matching new designs with previous designs or design styles is a way to encapsulate knowledge without having to reduce everything to the most basic level.

5.6. Cost functions for search constraints

A search constraint guides the constraint satisfaction procedure toward certain configurations. Search constraints can be used when the designer has confidence in the position of some part, or when he wishes to avoid undesirable configurations.

In a paper discussing work related to this dissertation [Amburn, 1986], we introduced the idea of a *dominance* parameter, which specified the objects prevailing when resolving constraints. In that work, constraints were solved locally, based on object-to-object comparisons, with the less dominant object adjusting its position to satisfy constraints. Dominance was considered to be associated with pairs of objects, which allowed the possibility of circular dominance, although we did not encounter this is our application.

In this system, dominance can be implemented by applying search constraints. Instead of specifying pairwise dominance parameters, however, a general dominance parameter is associated with each object. This forces the designer to specify a global ranking of objects, but avoids the problem of circular dominance and also provides consistency between applications. After constraint satisfaction, the most dominant objects are held close to their initial positions, while less dominant objects are freer to move about to satisfy the constraints.

Search constraints differ from geometric constraints in that they are intended to provide high-level guidance to the search procedure, rather than specifically dealing with the basic constraint problem. Just as subjective constraints get reduced to objective constraints, though, search constraints are also implemented in terms of objective evaluations. The dominance constraint described above, for instance, can be implemented by defining a geometric distance constraint that attempts to hold an object to its desired position. The cost increases as the distance from the initial position increases, and the rate of that increase can be controlled by the dominance parameter (e.g., it might be used as an exponent), so that roughly $cost = distance^{dominance}$.

5.7. Operations on constraints

This section discusses possible ways that cost functions may be combined. The first method has already been introduced: cost functions are added together to represent that two or more constraints should be satisfied. In addition, a simple expression for *A or B* exists. Unfortunately, cost functions are not generally invertible by the *not* operation, which prevents a straightforward derivation of expressions for additional logic operations. Nevertheless, the section shows that constraints may be combined in ways other than by simple addition of cost functions.

The theory of fuzzy logic [Klir, 1988] provides an additional source of information regarding operations on cost functions, and is a possible source of techniques for shaping functions based on subjective modifiers such as "very" and "broad".

Operations on constraints can be implemented by using the following cost function operations:

Constraint operation	Cost function operation
A and B	*cost(A) + cost(B)*
A or B	*min(cost(A),cost(B))*
not A	*in limited cases: 1.0-cost(A)*

5.7.1. A and B

The primary way of defining a constraint problem is to *and* together a collection of constraints. Given the costs functions for two constraints, *cost(A)* and *cost(B)*, the combined cost function must represent the individual costs during minimization. This may be accomplished by summing the individual costs: *cost(A and B)=cost(A) + cost(B)*.

5.7.2. A or B

Given a choice of two constraints, a minimizing procedure should choose *cost(A)* if *cost(A)<cost(B)*, and *cost(B)* if *cost(B)<cost(A)*. The cost function reflecting this is *cost(A or B) = min(cost(A),cost(B))*. Such a combined cost function might be used to implement a constraint such as "place the lamp near electrical outlet X" *or* "place the lamp near electrical outlet Y".

5.7.3. Not A

Consider a constraint with a threshold parameter, such as the *less than* constraint (figure 5.8), *parameter<5.0*. When the constraint is satisfied, the cost is *0.0*, but when the constraint is not satisfied the cost varies based on constraint violation. If we attempt take the complement of this constraint, then we still want the resulting cost to vary based on constraint violation. However, this information must come from the region of the curve for which cost is *0.0*. Since this information cannot be derived from a region of constant cost, such a function is not complementable.

On the other hand, consider a constraint with no threshold, such as the *equal* constraint (figure 5.6), *parameter=5.0*. There is only one point at which the cost is zero; elsewhere, the function returns a value which increases based on distance from the target parameter. We can take the complement of this constraint, because the information that we need for the new function (cost based on distance from the target) can be derived from the old function throughout the entire parameter range.

In general, functions with a threshold are not complementable, while those without a threshold may be. Among the six basic constraint types, *equal to, not equal to, minimize*, and *maximize* are complementable, whereas *less than* and *greater than* are not. The complement is achieved by negating the original function. If the previous function ranges from *0.0* to *1.0*, this function range can be maintained by adding *1.0*, so that *cost(not A)=1.0-cost(A)*.

A possible way to create invertible versions of *less than* and *greater than* is to define the quality of acceptable solutions. When the function is inverted, the portion of the curve that distinguished between acceptable solutions distinguishes unacceptable solutions. The problem with this approach is that the distinction between acceptable values is usually not as large as the distinction between unacceptable values. An inverted *less than* would therefore be shaped differently than an explicitly defined *greater than* function.

5.8. Summary

Cost functions provide a way of comparing two or more models to determine which best satisfies some set of constraints. Cost functions return a single scalar value which specifies the degree of constraint satisfaction. The magnitude of a cost function is not as important as the range of values it can assume. This chapter has provided guidelines for constructing cost functions for three types of constraints: objective, subjective, and search constraints. In addition, rules for combining cost functions were defined.

Six basic types of objective constraints were identified: *equal to*, *not equal to*, *less than*, *greater than*, *minimize*, and *maximize*. Cost function templates for these constraints were derived from their properties. While the application designer may use these templates where appropriate, he is also free to design arbitrary functions for any constraint that cannot be suitably represented. Subjective constraints involve value judgements and can be difficult to quantify . One approach is to break them down into simpler objective constraints which are easily evaluated. Search constraints allow the user to specify a preference for the position or freedom of model parts.

Chapter 6

Opaque Projector Project

Opaque projector design was chosen as a test application area. This project was pursued in collaboration with Prof. David Chapin of the NC State University School of Design, and students in his senior product design studio. The term project for the course was to design an opaque projector.

My version of this project differed from the students' version in two main ways. First, while the assignment for each student was to design a single opaque projector, my goal was to design a system capable of exploring opaque projector *design alternatives*. Second, the students were more concerned with the *form* of the projector; less emphasis was placed on engineering aspects such as ventilation and optics.

The distinction between our two projects becomes clearer if one looks at the output of each of the projects. The students generated detailed specifications and mock-ups for the shape of the projector. In most cases, however, they did not make specifications for major components such as lenses, bulbs, and fans. In contrast, CONTEST generates potential designs by choosing parts which best satisfy a set of constraints. CONTEST, however, does not generate design specifications to the same level of detail as presented by the students in the class. Issues such as where to drill holes and what type of screws to use are left to the human designer.

6.1. Problem overview

The design of opaque projectors is interesting because their function is well-understood, although the choice of parts to accomplish this function is non-trivial. The function of an opaque projector is to project an enlarged image of an opaque object (such as a book or photograph) onto a screen or other viewing surface.

Many performance requirements affect the design of opaque projectors. The following constraints were encoded in this application:

- ***Image brightness***: the projected image should be bright enough to read without eye strain.
- ***Component cost***: minimize part costs to achieve highest profits.
- ***Light escaping***: the projector should not emit distracting light (e.g., through cooling vents).
- ***Sound escaping***: the projector should not generate intrusive noise.
- ***Cord length***: the projector should have a power cord long enough to conveniently reach power outlets.
- ***Energy use***: the projector should be economical to operate.
- ***Image focus***: the projected image should be crisp rather than blurred.
- ***Projector geometry***: the dimensions of the projector should conform to ratios which ensure proper operation.
- ***Projector height***: a small projector might be desirable to prevent obstruction of view.
- ***Temperature constraints***: the internal operating temperature should be low enough to prevent part failure. Normally accessible parts should not cause discomfort if touched.
- ***Chassis cost:*** the price of the chassis should be minimized to achieve highest profits. The chassis was separated from other components to provide greater flexibility in specification.
- ***Ergonomics***: conveniences such as lens caps and carrying handles should be included.
- ***Weight:*** the weight of the projector affects its portability and should be minimized.
- ***Durability***: the ability of the projector to withstand handling or abuse

should be maximized.

- ***Full view***: the distance from the base of the projector to the the mirror should be great enough to image the entire source.

6.1.1. Constraint conflicts

Unfortunately, it is not possible to completely satisfy all design goals. For example, the internal operating temperature of the unit is affected by several parameters: the wattage of the projection bulb, the airflow generated by a fan, and the number of cooling vents. Increasing the bulb wattage increases the brightness of the image, but it also increases the amount of heat generated. Using a larger fan increases airflow, but also increases cost and might increase the amount of generated noise. Adding more cooling vents helps to lower the temperature, but allows sound and ambient light to escape into the room. Designing an opaque projector for a particular application involves choosing design parameters which best satisfy the entire set of constraints.

6.1.2. Projector classes

This project assumes simple projector operation as shown in figure 6.1. The source document is illuminated by a lamp; its image is reflected off a mirror, then passed through a lens and projected on the screen. In addition to the optical components illustrated, a projector may also contain a fan and air vents to promote cooling. Within this design framework lie an infinite number of specific designs (i.e., designs where distances, wattages, focal lengths, etc. are fully specified).

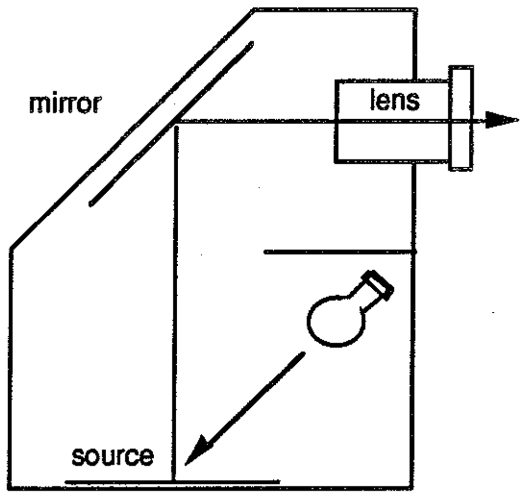


Figure 6.1: opaque projector operation

The goal of this application is to use the defined constraints and relevant catalog data to generate the specific design which best satisfies the constraints. If the weights of constraints are varied, then a new design may be generated. The human designer is thus free to explore various "what-if" design possibilities with minimal effort.

6.2. Parts inventory

This section describes the inventory of parts available when designing a projector in this application. Part attributes were estimated by consulting an Edmund Scientific catalog for typical (in some cases exact) values.

The part choices and their attributes are:

- bulb: wattage, price, weight
    - 200 watts, $2.50, 100 g
    - 400 watts, $5.00, 200 g
    - 600 watts, $6.00, 300 g
    - 800 watts, $7.00, 400 g
    - 1000 watts, $10.00, 500 g
    - 1200 watts, $11.50, 600 g
    - 1500 watts, $12.50, 750 g
    - 2000 watts, $15.00, 1000 g

- fan: wattage, price, weight, airflow (cubic feet/minute)
    - 0 watts (no fan), $0.00, 0 g, 0 cfm
    - 20 watts, $2.00, 200 g, 25 cfm
    - 25 watts, $4.00, 250 g, 40 cfm
    - 30 watts, $6.00, 300 g, 55 cfm
    - 35 watts, $8.00, 350 g, 70 cfm
    - 40 watts, $10.00, 400 g, 85 cfm
    - 45 watts, $12.00, 450 g, 100 cfm
    - 50 watts, $14.00, 500 g, 115 cfm
    - 60 watts, $16.00, 600 g, 130 cfm

- lens: diameter, price, weight, focal length (sorted by focal length)
    - 20.00 mm, $4.10, 40 g, 402 mm
    - 8.00 mm, $3.65, 6.4 g, 406 mm
    - 11.00 mm, $3.45, 12.1 g, 424 mm
    - 164.00 mm, $50.05, 2689.6 g, 436 mm
    - 8.00 mm, $3.25, 6.4 g, 460 mm
    - 112.00 mm, $15.65, 1254.4 g, 476 mm
    - 46.00 mm, $5.75, 211.6 g, 480 mm
    - 98.00 mm, $20.75, 960.4 g, 500 mm
    - 9.00 mm, $3.85, 8.1 g, 530 mm
    - 42.00 mm, $6.55, 176.4 g, 534 mm
    - 71.00 mm, $8.80, 504.1 g, 582 mm
    - 7.80 mm, $4.15, 6.08 g, 600 mm
    - 14.00 mm, $4.75, 19.6 g, 610 mm
    - 38.00 mm, $4.45, 144.4 g, 616 mm
    - 20.00 mm, $4.75, 40 g, 630 mm
    - 38.00 mm, $6.20, 144.4 g, 666 mm
    - 48.00 mm, $6.55, 230.4 g, 694 mm
    - 21.00 mm, $4.95, 44.1 g, 786 mm
    - 20.50 mm, $5.15, 42.02 g, 830 mm
    - 89.00 mm, $14.00, 792.1 g, 952 mm
    - 103.00 mm, $19.30, 1060.9 g, 1010 mm
    - 40.00 mm, $6.55, 160.0 g, 1334 mm
    - 38.00 mm, $5.25, 144.4 g, 1360 mm
    - 76.00 mm, $16.15, 577.6 g, 1370 mm
    - 38.00 mm, $4.90, 144.4 g, 1600 mm
    - 24.00 mm, $6.35, 57.6 g, 2400 mm
    - 12.00 mm, $4.60, 14.4 g, 2540 mm

- vents: number, weight
    - 0 to 8 vents
    - each vent subtracts 50 g in weight

- cord: length, price, weight
    - length between 1m and 10m
    - price: $5.00 + $1.00 / m
    - weight: 100 g + 40 g / m

- chassis geometry:
    - base-to-mirror distance: unrestricted
    - mirror-to-lens distance: unrestricted
    - external height, width, length: dependent on base-to-mirror and mirror-to-lens distances

- chassis construction:
    - type of metal: aluminum or steel
    - steel:
        - weight: 2.5 g / 500 mm^2
        - cost: \$1.00 / 50000 mm^2
    - aluminum:
        - weight: 1 g / 500 mm^2
        - cost: \$1.00 / 75000 mm^2

- optional extras:
    - carrying handle: \$3.00, 100 g
    - lens cap: \$3.00, 50 g
    - cart/stand: \$20.00

A bulb, lens, cord, and chassis must be selected. Cooling vents, a fan, a carrying handle, a lens cap, and a cart are optional.

6.3. Constraint implementation

Recall the systematic design method outlined in section 5.4.6:

1. express the constraint as one of the six basic constraint types
2. determine the main parameter which represents the property being constrained
3. select the particular form of the constraint being implemented (determine whether bounded or unbounded)
4. select the particular function template (e.g., $f(x) = e^{-nx^b}$ or $f(x) = 1/(nx^b+1)$)
5. select the parameters of the template
6. choose a scaling factor

The sections which follow document the first three steps of this method for each constraint. The final three steps involve selection of subjective parameters and hence are not documented here.

6.3.1. Image brightness

The goal of this constraint is to maximize image brightness. I assume that the bulb brightness is proportional to the wattage (since the bulb wattage is more readily available). Furthermore, I assume that the amount of light directed toward the lens is approximately the same for all projector configurations, and that the lens itself absorbs a negligible percentage of the light passing through it . In other words, the only non-constant parameters affecting image brightness are the bulb wattage and the lens diameter. I further assume that the amount of light gathered by the lens is proportional to the area of the lens.

> ***constraint:*** *maximize brightness*
> ***parameter:*** *<bulb wattage> * <lens diameter>*^{2}
> ***function:*** *lower bounded maximize, bounded by 200 watt bulb, 8 mm lens*

6.3.2. Component cost

The total part cost of the projector is simply the sum of the component prices. Because this sum is already in the global unit of measure (dollars), no further shaping of the parameter is required.

> ***constraint:*** *minimize part cost*
> ***parameter:*** *sum of individual part costs*
> ***function:*** *no further shaping necessary*

6.3.3. Light escaping

The amount of light escaping is dependent on the number of cooling vents and the brightness of the bulb. I assume a fixed percentage of the bulb's luminance escapes from each vent.

> ***constraint:*** *minimize light escaping*
> ***parameter:*** *<bulb wattage> * <number of vents>*
> ***function:*** *lower bounded minimize, bounded by 0*

6.3.4. Sound escaping

Information was not available on the noise generated by various fans. Instead, fan wattage was used as a rough estimate of the noise generated. I assume a fixed percentage of the generated sound escapes from each vent.

constraint: minimize sound escaping
*parameter: <fan wattage> * <number of vents>*
function: lower bounded minimize, bounded by 0

6.3.5. Cord length

This constraint expresses the view that it is convenient to have as long a cord as possible (up to the maximum cord length).

constraint: *maximize cord length*
parameter: *cord length*
function: *lower bounded maximize, bounded by length of 1m*

6.3.6. Energy use

Since power usage is a cost incurred by the consumer rather than the producer of the product, this constraint actually reflects an estimate of how various power consumption levels will affect the consumer's purchase behavior.

constraint: *minimize energy use*
parameter: *sum of individual part wattages*
function: *lower bounded minimum, bounded by 200 watt bulb*

6.3.7. Image focus

This is the most important constraint in the design. The projector is useless if it cannot present a focused image on the screen. We can apply the principles of thin lens theory to evaluate optical configurations. In the situation where we have a lens projecting an image onto a screen, the lensmaker's equation states that:

$$\frac{1}{\textit{object-to-lens distance}} + \frac{1}{\textit{lens-to-image distance}} = \frac{1}{\textit{focal length of lens}} \tag{6.1}$$

Equation 6.1 expresses the following imaging model:

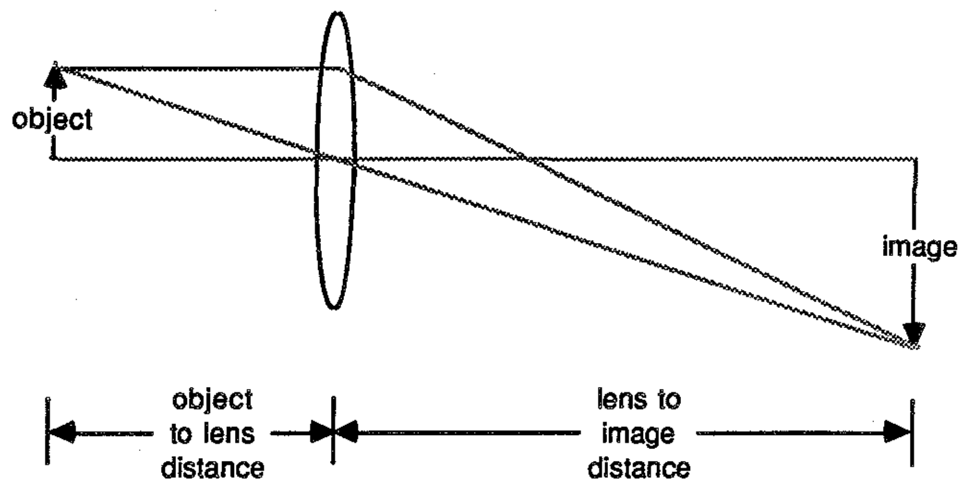


Figure 6.2: thin lens imaging model

In an opaque projector, the object-to-lens distance is the sum of the base-to-mirror distance and the mirror-to-lens distance. The lens-to-image distance for the cases in this chapter was set to a fixed distance of five meters. There is some flexibility in meeting this constraint since the lens position is slightly adjustable (focus control). The cost function uses the image and object distances to compute the focal length necessary to provide a focused image. This focal length is then compared with the actual focal length of the lens.

constraint: *equality between required focal length for perfect focus and the actual focal length of the lens*
parameter: *actual focal length of the lens*
function: *equality with required focal length*

6.3.8. Projector geometry

This constraint provides guidance about the ratios of the base-to-mirror and mirror-to-lens distances of typical opaque projectors. Its purpose is to generate projectors that have the ratios of figure 6.1, without explicitly having to encode why such a ratio is good (it leaves room for the bulb and fan, it minimizes required desk space, etc.).

constraint: *<base-to-mirror distance> equals twice <mirror-to-lens distance>*
parameter: *2 * <mirror-to-lens distance>*
function: *equality with <base-to-mirror distance>*

6.3.9. Projector height

The main parameter affecting the height of the projector is the distance from the base to the mirror. Thus to minimize the height, one needs to minimize this distance. One reason for minimizing the height is to prevent obstruction of view.

> ***constraint:*** *minimize base-to-mirror distance*
> ***parameter:*** *base-to-mirror distance*
> ***function:*** *lower bounded minimize, bounded by 0*

6.3.10. Temperature constraints

The internal temperature of the projector must be kept at a reasonable level to prevent part failure and for safety considerations. Two internal components generate heat: the bulb and the fan. A portion of the bulb's power usage is in the form of internal heat; the remainder of the energy is in the form of light which leaves the projector. Similarly, only part of the fan's power usage is used to create airflow outside the projector; the remainder generates internal heat. I have chosen percentages of the wattage of each part as an estimate of internal heat generated.

The heat generated is dissipated to the surrounding environment by three means: conduction, convection, and radiation. Conduction is the transfer of heat from direct contact of the projector with other surfaces. For example, heat might be conducted through the projector's feet to the table the projector is sitting on. Convection is the transfer of heat by motion of a hot material (e.g., air or water). Heat transfer by fan-induced airflow falls in this category, as does heat drawn away by rising currents of warm air. Radiation is the transfer of energy by electromagnetic waves. If one's hand is placed near the side of a warm projector, it absorbs radiant energy even though there is no airflow or contact with the projector. Although some heat is dissipated by conduction and radiation in an opaque projector, the vast majority occurs through convection.

A sophisticated convection model would consider many factors, including fan and vent position, the temperature difference between the inside and outside of the projector, and the shape of the chassis and vents. Rather than implementing a complex model of heat transfer, I chose to base this constraint on a cooling potential, which I define as being dependent on the the number of vents and the fan's airflow specifications. The cooling achieved by additional vents when no fan is present is represented by defining a small default airflow:

$$\textit{cooling potential} = \textit{<number of vents>} * \textit{<airflow>} \tag{6.2}$$

$$\textit{cooling requirements} = \alpha * \textit{<bulb wattage>} + \beta * \textit{<fan wattage>} \tag{6.3}$$

The cost function for the constraint is then expressed as:

constraint: *cooling requirements less than cooling potential*
parameter: *cooling requirements*
function: *less than cooling potential*

Note that there is a limit of eight on the number of cooling vents. When that limit is reached, cooling potential can only be increased by using a more powerful fan.

6.3.11. Chassis cost

This constraint minimizes the cost of the metal required to build the chassis. Because this parameter is already in the global unit of measure (dollars), no further shaping of the parameter is required.

constraint: *minimize chassis cost*
parameter: *chassis cost*
function: *no further shaping necessary*

6.3.12. Ergonomics

This constraint reflects the added convenience of extras such as a carrying handle, a lens cap, and a utility cart. Rather than having a penalty for the absence of these conveniences, the cost function equivalently returns a negative bonus when the extras are included. Cost is computed as follows:

- Start with an initial cost of 0.0.
- Does the projector have a handle? If so, subtract 12.0 from cost.
- Is there a lens cap for the lens? If so, subtract 4.0 from cost.
- Is a cart included with the projector? If so, subtract 15.0 from cost.

6.3.13. Weight

If the projector is to be shared by many people (e.g., by many instructors in a school), then a low weight is desirable since it will aid portability.

> ***constraint:*** *minimize weight*
> ***parameter:*** *sum of individual component weights*
> ***function:*** *lower bounded minimize, bounded by lightest bulb, lens, and chassis*

6.3.14. Durability

The constraint reflects the desire to minimize repair or maintenance during the lifetime of the product. Two parts are considered: the fan and the chassis. Steel is more durable than aluminum, and a projector is considered more reliable if it does not include a fan, since a fan can malfunction. Cost is computed as follows:

- Start with an initial cost of 0.0.
- Is the chassis made of steel? If so, subtract 40.0 from cost.
- Is the chassis made of aluminum? If so, subtract 2.0 from cost.
- Is there a fan? If not, subtract 2.0 from cost.

6.3.15. Full view

If the distance from the base of the projector to the mirror is too small, the projector will not be able to reproduce the entire area of the original. A distance of less than 250 mm is considered dangerously close and is penalized with a cost of 20.0. Any distances greater than 250 mm are rewarded by subtracting a bonus of 1.0 unit for every 50 mm. This function would be better represented by a *greater than* function template, but this awkward cost function was carried over from an early implementation.

6.4. Results

The particular weighting for each of the cost functions was chosen so as to specify a generic projector achieving an overall compromise among all constraints. These weightings were stored within each cost function, so that an external weight of 1.0 for any cost function refers to the same weighting as used for the standard projector.

Variations on the standard projector were then defined as follows:

budget projector: set the weighting factor for the *component cost* and *chassis cost* constraints to 5.0. This represents the situation where cost is a primary concern.

deluxe projector: set the weighting factor for *component cost* to 0.2 and for *chassis cost* to 0.1. This represents a situation where cost is not an issue, particularly for fixed parts such as the chassis, as opposed to replaceable parts, such as the fan and bulb.

quiet projector: set the weighting factor for the *sound escaping* constraint to 5.0. This represents the situation where the sound generated by the projector is a concern, such as in a small conference room with poor acoustics.

bright projector: set the weighting factor for the *image brightness* constraint to 5.0. This represents the situation where maximum image brightness is a concern, such as in a room with windows but no blinds or drapes.

portable projector: set the weighting factor for the *weight* constraint to 5.0. This represents the situation where the projector will be moved frequently, so a light projector is preferred over a heavy one.

Table 6.1 summarizes the weightings for each of the projector instances.

	standard	budget	deluxe	quiet	bright	portable
Image brightness	1.0	1.0	1.0	1.0	**5.0**	1.0
Component cost	1.0	**5.0**	**0.2**	1.0	1.0	1.0
Light escaping	1.0	1.0	1.0	1.0	1.0	1.0
Sound escaping	1.0	1.0	1.0	**5.0**	1.0	1.0
Cord length	1.0	1.0	1.0	1.0	1.0	1.0
Energy use	1.0	1.0	1.0	1.0	1.0	1.0
Image focus	1.0	1.0	1.0	1.0	1.0	1.0
Projector geometry	1.0	1.0	1.0	1.0	1.0	1.0
Projector height	1.0	1.0	1.0	1.0	1.0	1.0
Temp. constraints	1.0	1.0	1.0	1.0	1.0	1.0
Chassis cost	1.0	**5.0**	**0.1**	1.0	1.0	1.0
Ergonomics	1.0	1.0	1.0	1.0	1.0	1.0
Weight	1.0	1.0	1.0	1.0	1.0	**5.0**
Durability	1.0	1.0	1.0	1.0	1.0	1.0
Full view	1.0	1.0	1.0	1.0	1.0	1.0

Table 6.1: cost function weights for projector instances

Figures 6.3 through 6.8 show the components and dimensions chosen to best satisfy each of these constraint weightings. These designs were all generated automatically, although the figures were prepared by hand.

By using a program such as this, a manufacturer can easily develop a product line. If the inventory or pricing of parts changes, one can simply rerun the program with the new parts or new part costs. Almost all of the human time investment occurs during the initial cost function encoding process.

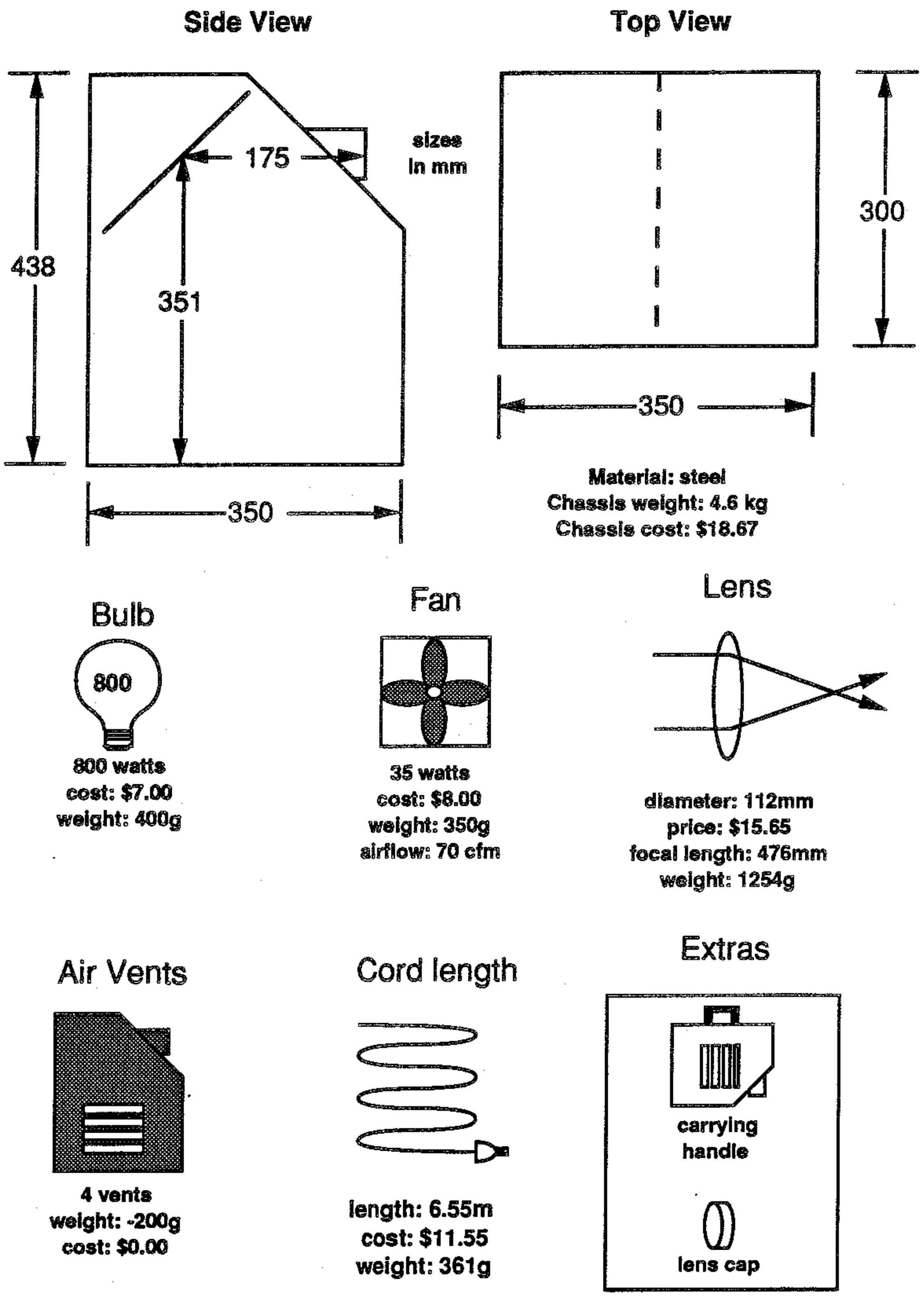


Figure 6.3: Standard Projector

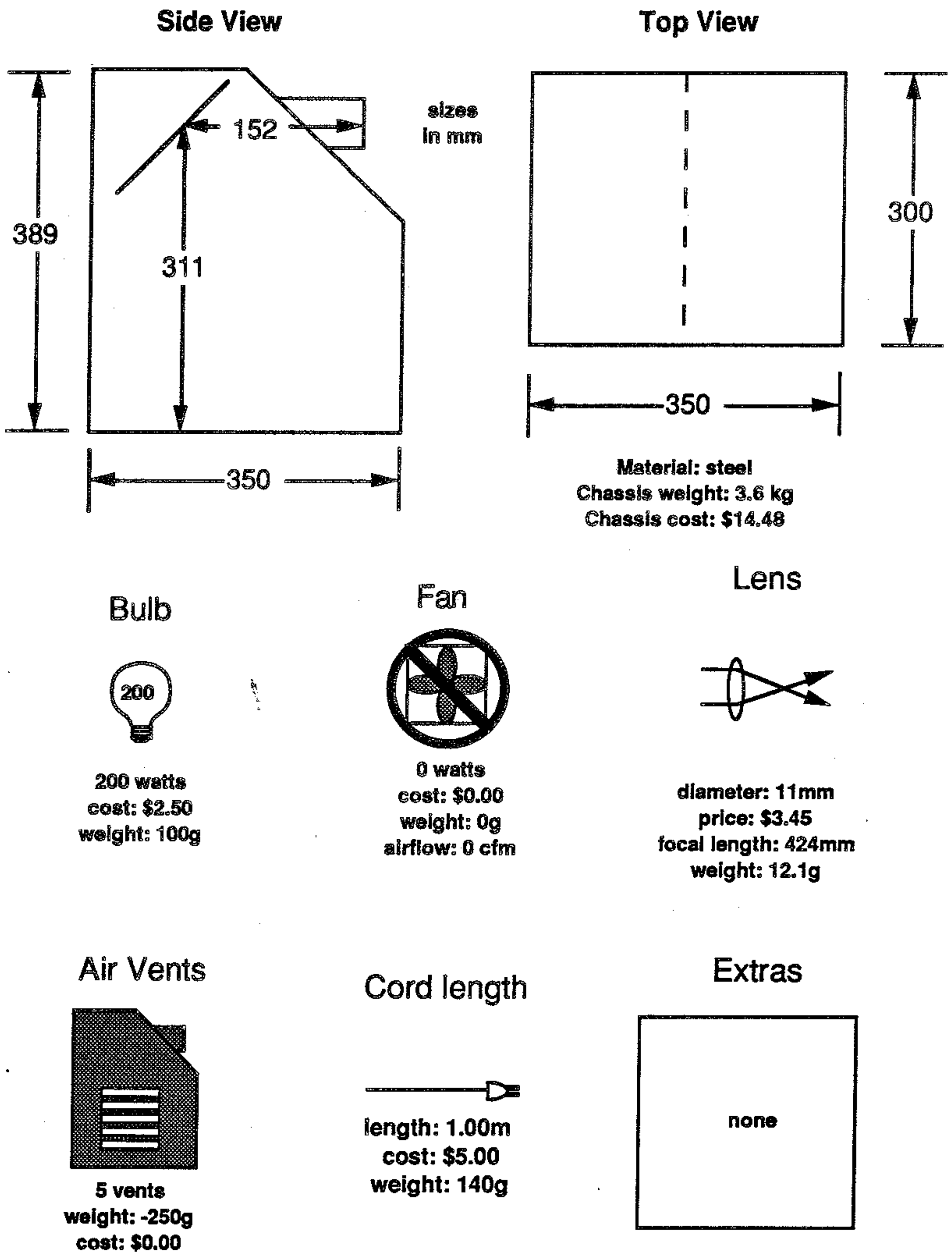


Figure 6.4: Budget Projector

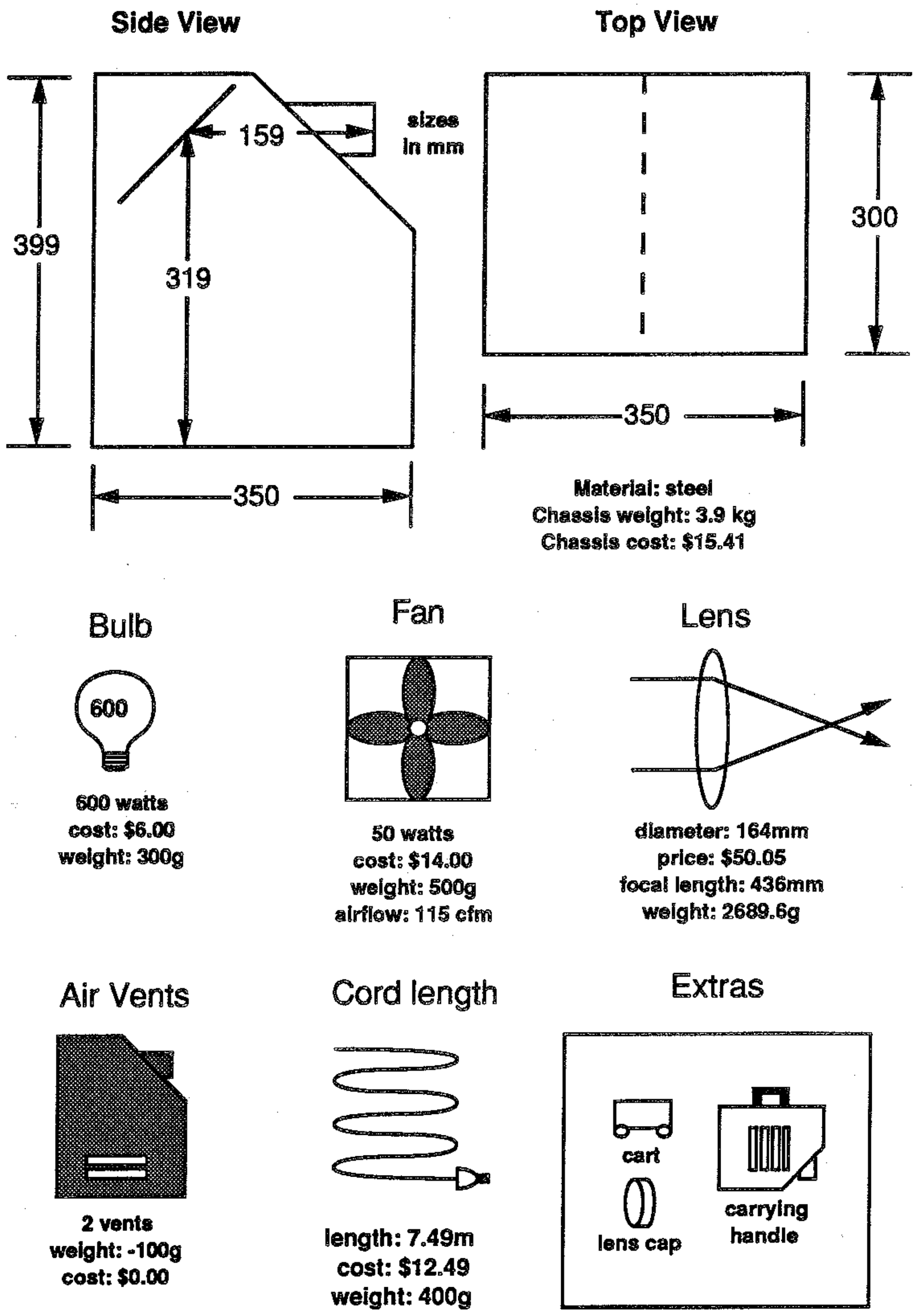


Figure 6.5: Deluxe Projector

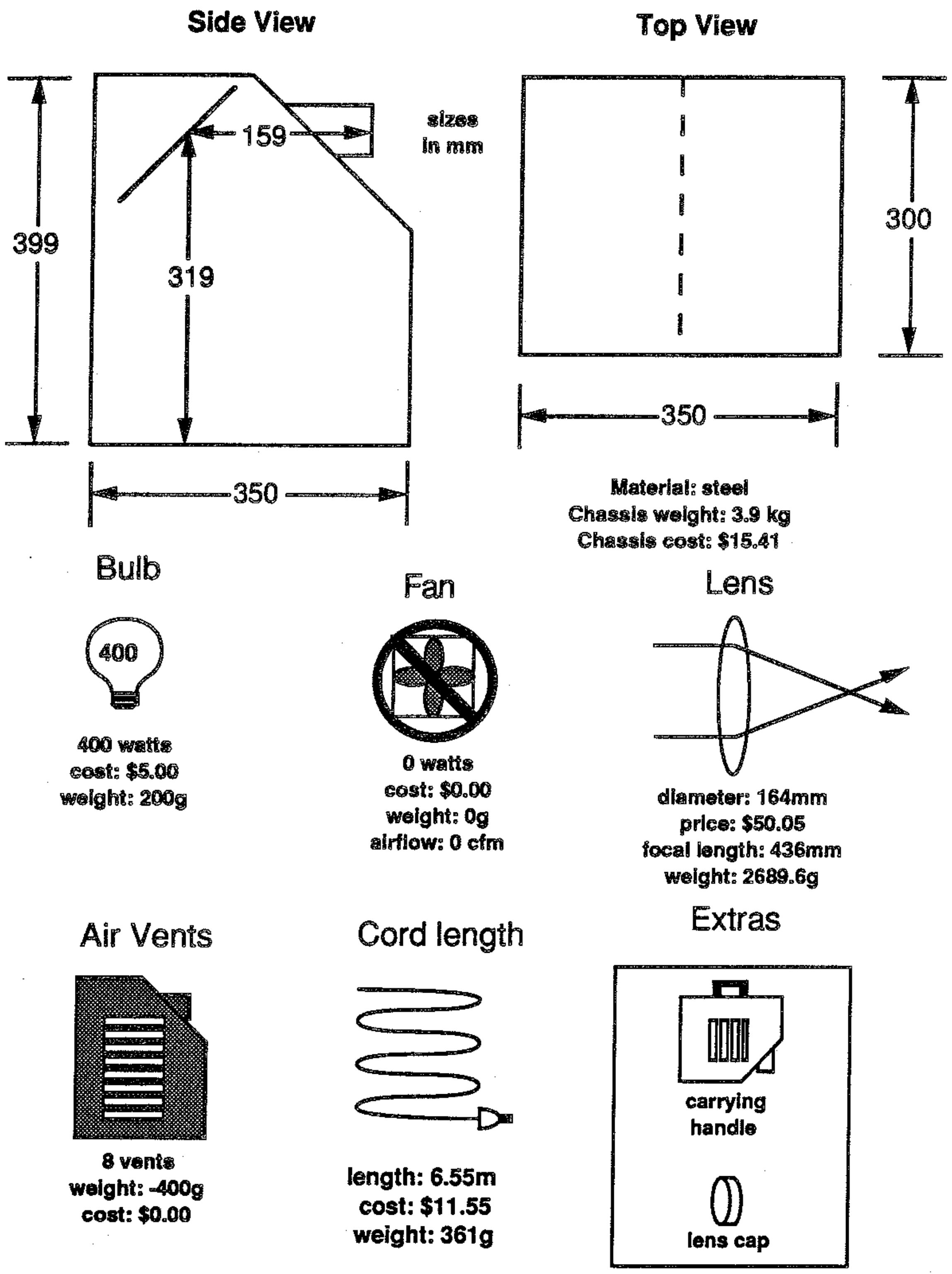


Figure 6.6: Quiet Projector

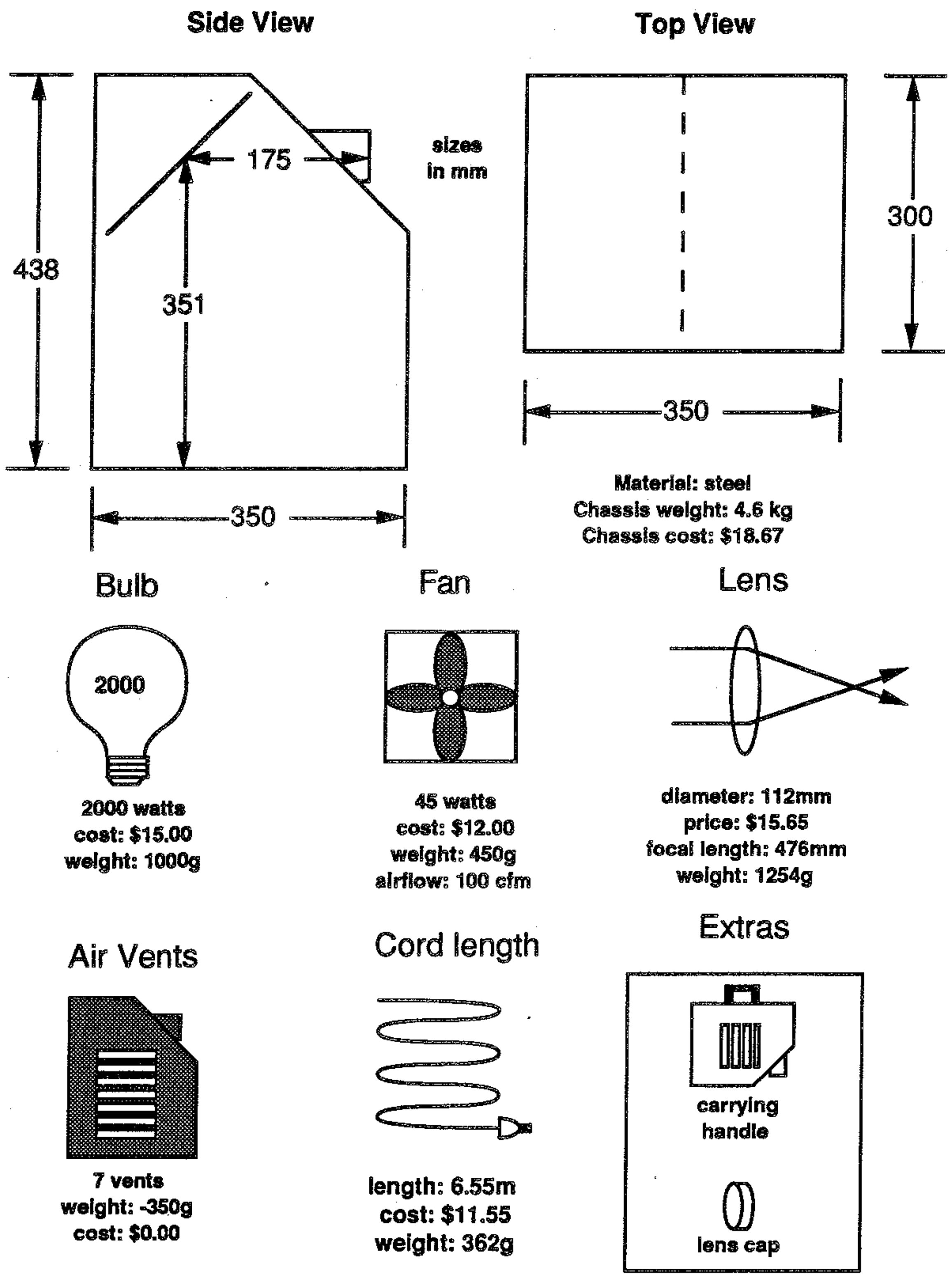


Figure 6.7: Bright Projector

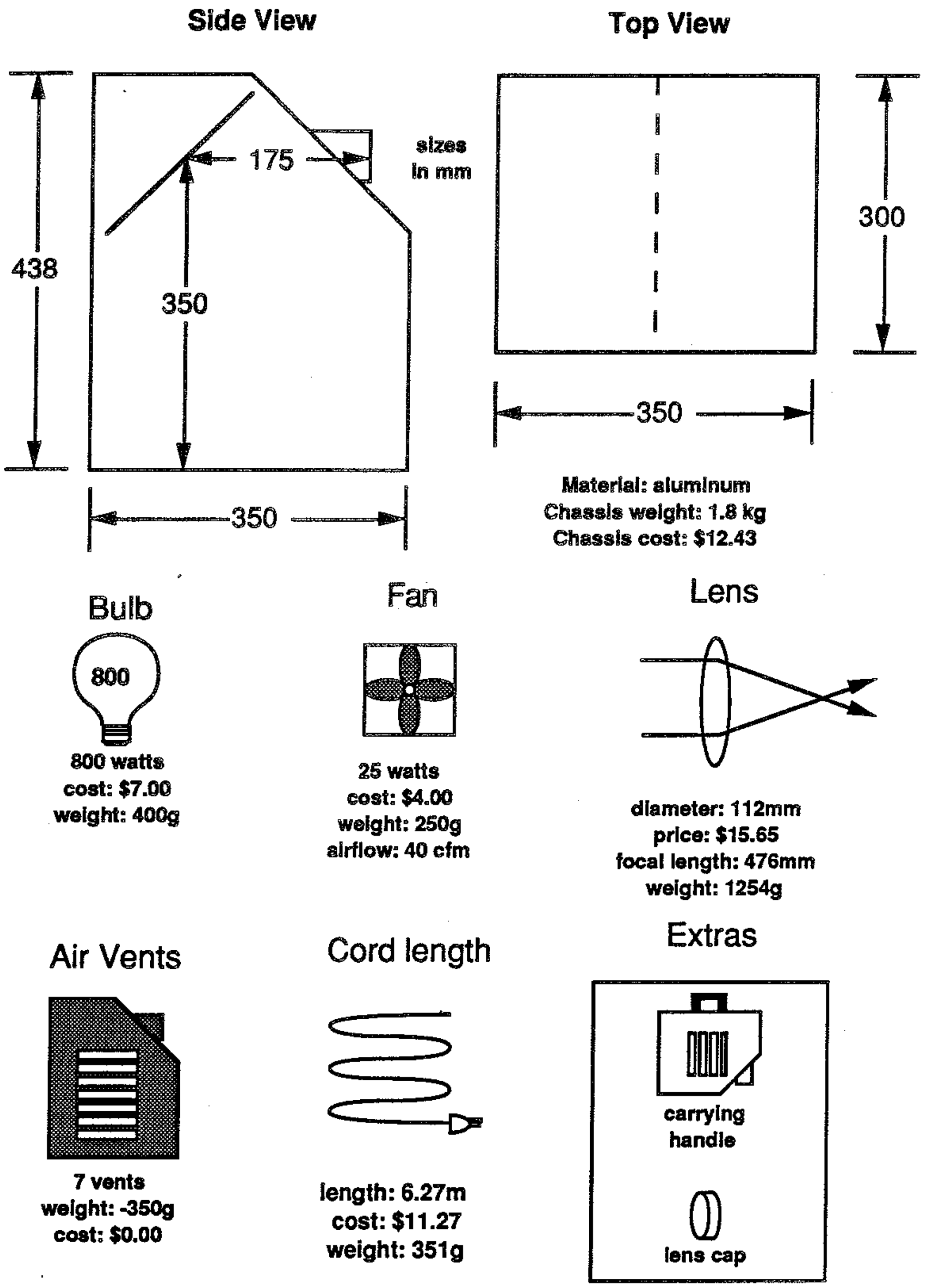


Figure 6.8. Portable Projector

Table 6.2 summarizes the information presented in figures 6.3-6.8.

	standard	budget	deluxe	quiet	bright	portable
Bulb wattage	800	200	600	400	2000	800
Fan wattage	35	none	50	none	45	25
Lens diameter	112	11	164	164	112	112
Lens focal length	476	424	436	436	476	476
Cord length	6.55	1.00	7.49	6.55	6.55	6.27
Cooling vents	4	5	2	8	7	7
Carrying handle	yes	no	yes	yes	yes	yes
Lens cap	yes	no	yes	yes	yes	yes
Utility cart	no	no	yes	no	no	no
Chassis material	steel	steel	steel	steel	steel	alum.
Chassis height	438	389	399	399	438	438
Chassis width	350	350	350	350	350	350
Chassis depth	300	300	300	300	300	300
Base-to-mirror	351	311	319	319	351	350
Mirror-to-lens	175	152	159	159	175	175

Table 6.2: comparison of part and parameter selection

6.5. Evaluation

The *image focus* constraint is satisfied in each of the six projectors generated, indicating that each projector should be capable of functioning. The choice of lenses is greatly dependent upon the distance of the lens to the screen, which was fixed at five meters. To further evaluate the results, it is useful to examine both the generated design parameters (table 6.2) as well as relevant attributes which are dependent on these parameters (figures 6.9 and 6.10).

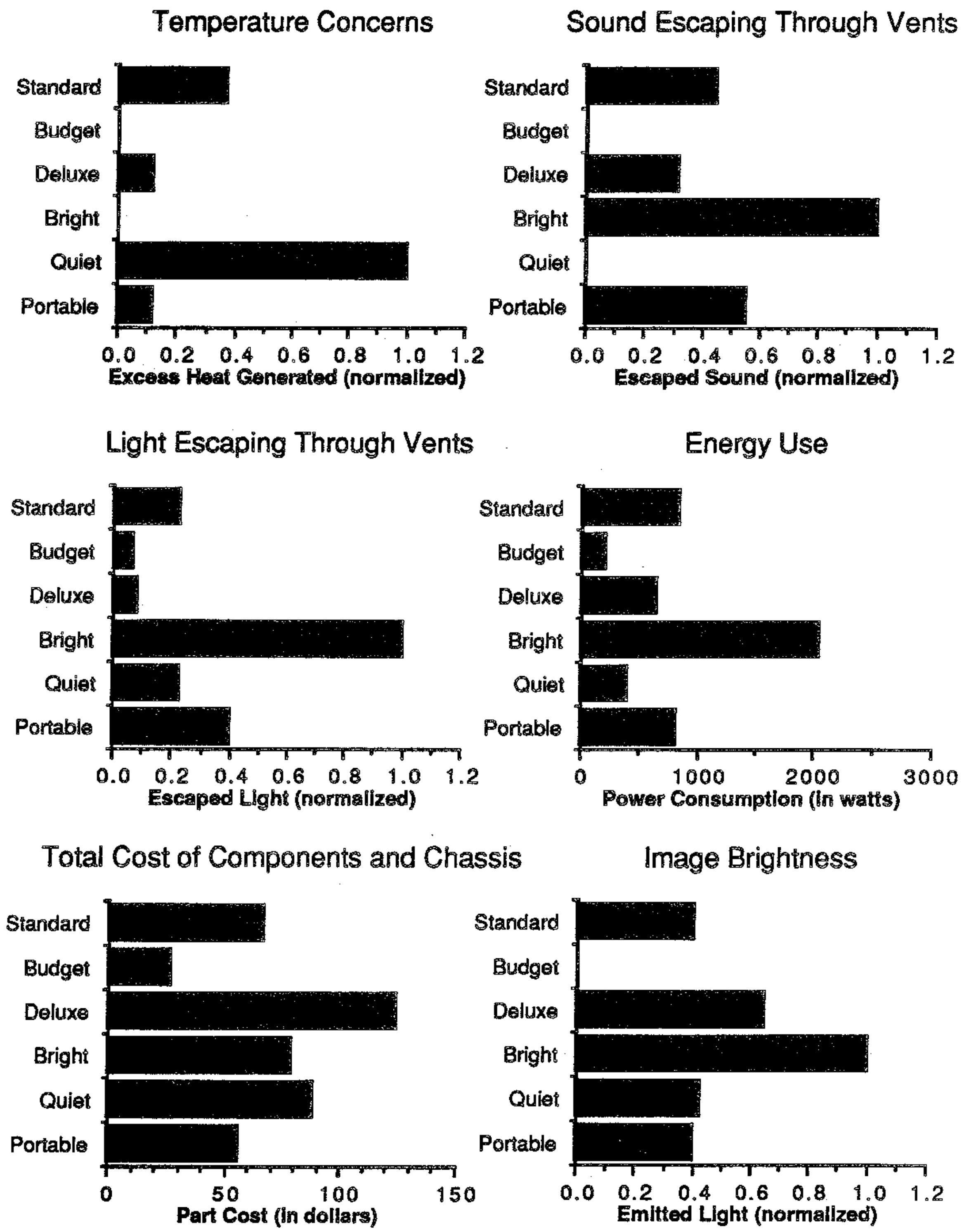


Figure 6.9: comparison of projector attributes

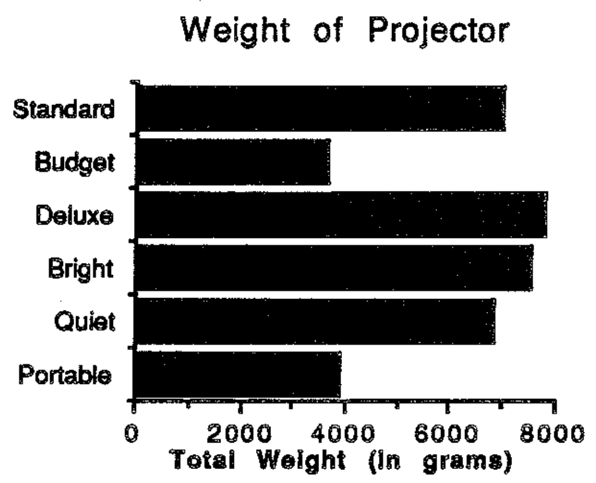


Figure 6.10: comparison of projector weights

The following is a brief evaluation of each projector design:

> ***standard projector***: This is the least interesting projector. All component values tend to be near the middle of the spectrum of legal choices.
>
> ***budget projector***: This is a barely functional projector. It contains only a 200 watt bulb, and the lens diameter is too small to pass much light. Such a projector would be useful only to demonstrate design principles, say for a course project.
>
> ***deluxe projector***: I assumed the program would generate a design with a higher wattage bulb than the standard projector, since the cost of the more expensive bulb would not be much of a concern. Instead, the program designed a projector with a lower wattage bulb. It does generate a brighter image, but it accomplishes this by selecting a larger diameter lens. A larger lens is a more reasonable alternative because it only costs extra money, whereas a brighter bulb generates extra heat which in turn can cause extra noise. The deluxe projector outperforms the standard projector in all attributes of figure 6.9 except cost, yet weights only slightly more.
>
> ***quiet projector***: Since noise is such an overriding concern, the program generates a design without a cooling fan. A projector without a fan is susceptible to overheating. The choice of a low wattage bulb, a large diameter lens, and the maximum number of cooling vents (eight) all

address this concern.

bright projector: This projector contains the highest wattage bulb (2000 watts) available. The bulb combines with a reasonable diameter lens (112 mm) to generate the brightest image. To dissipate the heat generated by this bulb, the largest number of air vents of any projector with a fan is chosen (seven). The result is a noisy, power hungry machine that generates a very bright image. It also allows the greatest amount of light to escape through the air vents, but this may not be a problem if the bright image is necessary because the projector is already in an illuminated environment.

portable projector: The portable projector is quite similar to the standard projector, with two main exceptions. First, it has an aluminum chassis, which is lighter than the steel chassis of the other projectors. Second, it cools by using a smaller, lighter fan and more air vents. In essence it allows extra noise and light to escape in exchange for a reduced weight. Figure 6.10 shows that it is light, but still heavier than the budget projector. However, it vastly outperforms the budget projector: it has a larger lens and a brighter bulb.

Projector	CPU seconds (DEC 3100)	# of constraints	# of iterations
Standard	3341.2 (55 min, 41.2 sec)	15	2991000
Budget	3638.3 (1 hr, 0 min, 38.3 sec)	15	3301000
Deluxe	3304.4 (55 min, 4.4 sec)	15	2961000
Bright	3249.1 (54 min, 9.1 sec)	15	2931000
Quiet	3267.4 (54 min, 27.4 sec)	15	2941000
Portable	3398.9 (56 min, 38.9 sec)	15	3041000

Table 6.3: CPU times for opaque projector problems

Table 6.3 shows the amount of time used in solving each of the opaque projector problems. Each simulation was run until the system cooled to a temperature threshold. Variations in run time can be attributed to the adaptive nature of the annealing schedule. The cooling rate is dependent upon the variance of cost values for a particular temperature, which in turn is dependent on stochastic perturbation of function parameters. Variation in run time therefore is to be expected.

6.6. Summary

This chapter has documented the use of CONTEST in opaque projector design. The notable feature of the resulting application is that it is capable of generating a range of projectors, rather than just a single design. Such an application can be used to design an entire product line meeting the varying needs of consumers.

To generate a new design reflecting modified importance of certain constraints, the designer only needs to change the weighting factors of the cost function associated with those constraints. Design tradeoffs can therefore be easily explored.

A straight-forward design method was followed in this application: 1) specification of the problem, 2) identification of available parts and legal parameters, 3) encoding of constraints, and 4) evaluation of results. The resulting projector designs properly reflect the tradeoffs specified by the constraints and weighting factors. The systematic design method described in chapter five was shown to be suitable for encoding the geometric (e.g., optics), indirectly geometric (temperature concerns), and non-geometric (e.g, power consumption) constraints of this problem.

Chapter 7

Site Planning Project

Site planning was chosen as a second test application area. This project was pursued with the assistance of Ken Pittman of the NC State University School of Design, and other members of the Research Triangle Park Project Team. The goal was to implement a system capable of automatically generating site plans that meet typical constraints of site plan development.

Some constraints were drawn from the manual, *Guidelines for Site Development*, prepared by the Research Triangle Park Project Team [RTP Team, 1987]. The remaining constraints were common-sense constraints (e.g., buildings should not overlap) that are obvious to humans but not computers.

7.1. Problem overview

The objective of the site planning problem considered here is to place buildings, parking lots, and access roads on a predefined tract of land so as to best satisfy a set of constraints on their location.

7.1.1. Objects

The relocatable objects and their attributes are:

- ***buildings***: A building is represented as a rectilinear solid of fixed length and width, randomly chosen for each building. Buildings are predefined to be either three, five, or seven stories tall. To satisfy constraints, the building may be repositioned to any (x,y) position. The base elevation is determined by the elevation of the terrain at that point. Any number of buildings may be created, but the simulations in this chapter were run with fifty buildings.

- ***parking lots***: Parking lots also have fixed length and width (randomly chosen) and may be repositioned to any (x,y) position. Any number of parking lots may be created, but the simulations in this chapter were run with three parking lots.

- ***access roads***: Access roads connect parking lots to main roadways. Each parking lot has an associated access road which connects it to the nearest roadway.

In addition to these relocatable objects, a few fixed objects affect the design problem. The fixed objects and their attributes are:

- ***main roadways***: A road network is predefined with roads of varying widths. The main parkways are 200 feet wide, the collector roads are 100 feet wide, and the subcollectors are 75 feet. Figure 7.1 shows the road network created for the examples presented in this chapter.

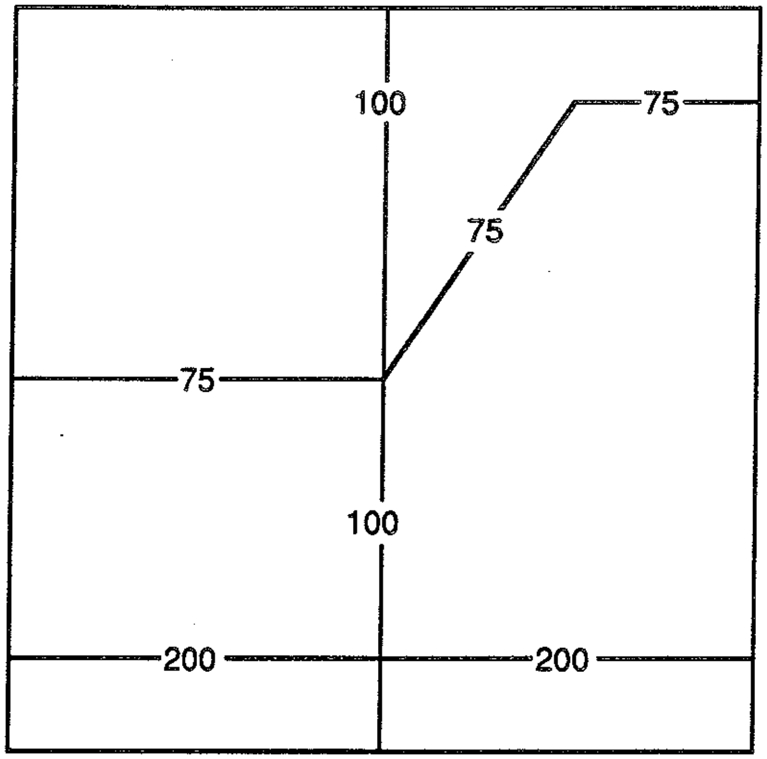


Figure 7.1: road network with road widths

- *terrain*: The tract of land is defined as a square area with hills. The elevation at each point may not be adjusted. Figure 7.2 shows the height of the terrain (lighter shades indicate higher elevations).

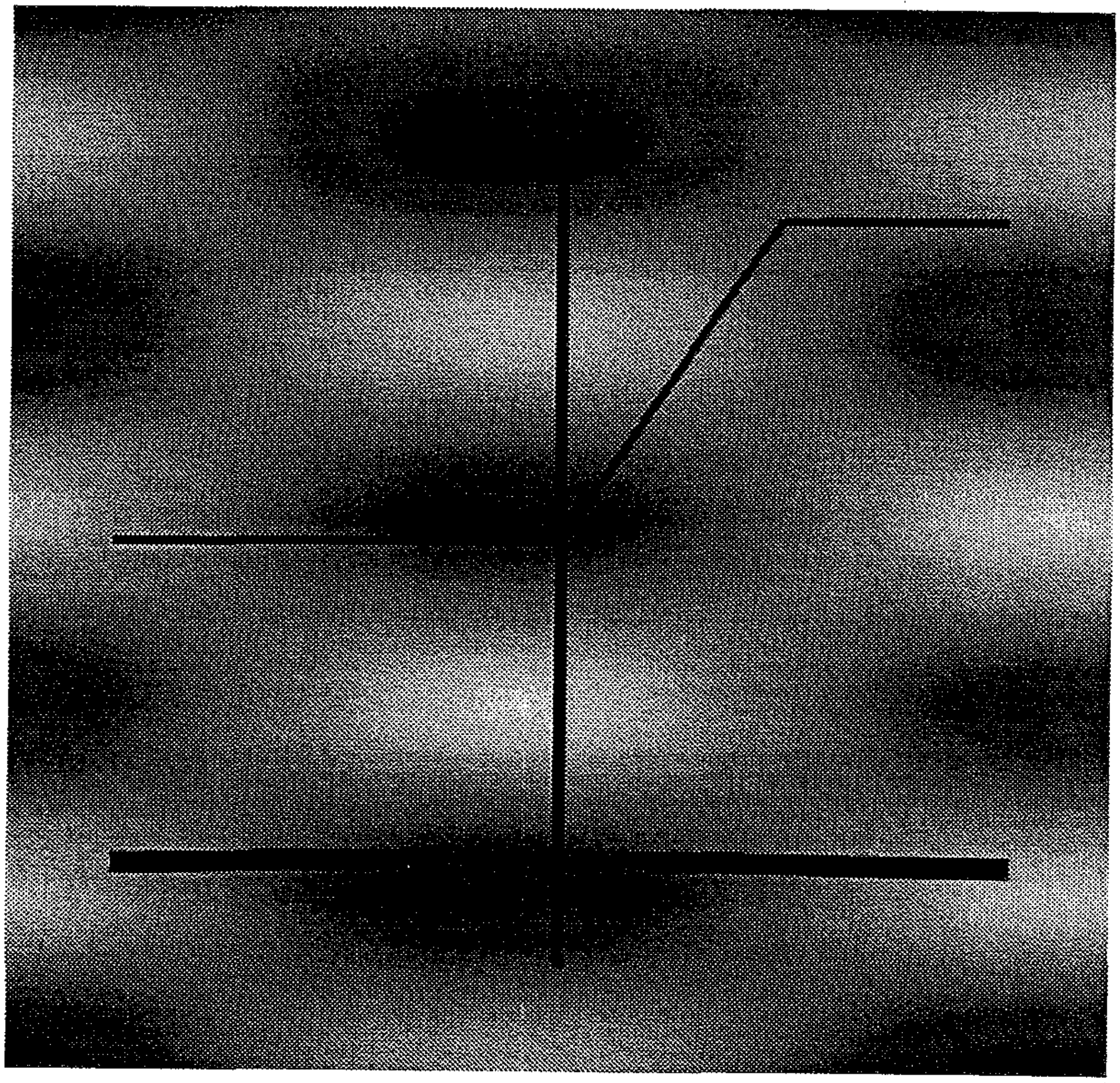


Figure 7.2: terrain elevation

- *trees*: The trees are randomly sized and positioned and are considered to be unmovable. I.e., man-made objects should be moved to avoid cutting trees. Any number of trees may be created, but the simulations in this chapter were run with fifty trees.

7.1.2. Constraints

The following constraints were encoded to guide the design of site plans:

- ***object non-intersection***: This is a common-sense constraint that buildings should not intersect other buildings, parking lots, roads, or trees. In addition, parking lots should not intersect other parking lots, roads, or trees. Finally, roads should not intersect trees.

- ***road setbacks***: This constraint defines the minimum distance between roads and any constructed facilities (buildings or parking lots), or alternately between any constructed roads (access roads in this application) and trees, buildings, or parking lots.

- ***place buildings away from main roads***: This constraint reflects the desire to maintain a pleasant, park-like setting. One way to accomplish this is to place buildings as far away from roads and traffic as possible.

- ***interbuilding spacing***: Certain minimum distances should be provided between buildings to allow walkways and maintenance access, and to provide less obstructed views from windows than might otherwise occur.

- ***place buildings near parking lots***: The distance employees or visitors must walk from their car to a building should be minimized.

- ***place buildings on hills***: Placing buildings on hills provides a pleasing external appearance, helps to manage drainage, and helps to prevent visibility of rooftops.

- ***avoid unpleasant views from buildings***: To avoid unpleasant views of nearby rooftops, neighboring buildings should be approximately the same height.

7.1.3. Constraint conflicts

As in the opaque projector project of the previous chapter, it is impossible to completely satisfy all constraints. For example, the goal of maximizing building distance from main roads conflicts with the constraint that buildings cannot overlap. In addition, placing buildings on hills conflicts with both of these constraints, since hills do not always occur away from main roads.

7.2. Constraint implementation

This application was implemented before all constraint library routines were completed, so the exact shaping functions described in chapter 5 were not used. However, the cost functions used *were* chosen to implement the six basic constraint types. This section documents the constraint implementation based on these types, even though explicit calls to library routines were not used.

7.2.1. Object non-intersection (e.g., building-building)

The goal of this constraint is to keep objects from overlapping one another. The amount of interference can be determined by the volume of the intersection of the objects. This constraint was first implemented by approximating all objects by polyhedra and using a general polyhedral intersection routine. This gave a precise measure of interference, but was also exceedingly slow. To speed up evaluation, the general polyhedral intersection test was replaced by a bounding box intersection test, which has performed satisfactorily with a much lower computational expense.

> ***constraint:*** *minimize volume of intersection*
> ***parameter:*** *volume of intersection of objects' bounding boxes*
> ***function:*** *lower bounded minimize, bounded by zero*

7.2.2. Road setbacks

The manual on site plan development states that building setbacks for the most common road widths should be observed as follows:

Right-of-way width	Minimum Setback Required
300 feet	175 feet
200 feet	150 feet
150 feet	100 feet
120 feet	100 feet
< 120 feet	100 feet

To implement this constraint, the width of each road is examined and the appropriate minimum setback is selected. A *greater than* constraint using that distance is then evaluated.

***constraint:** object distance greater than required setback distance*
***parameter:** distance between road and object*
***function:** greater than*

7.2.3. Place buildings away from roads

The manual for site plan development states that one of the objectives is "the continuation of a park-like character". A park-like character can be maintained by placing buildings away from main roads.

***constraint:** maximize distance between building and nearest road*
***parameter:** distance from building to nearest road*
***function:** lower bounded maximize, bounded by zero*

7.2.4. Interbuilding spacing

Rather than implement a separate constraint for interbuilding spacing, the building non-intersection constraint was generalized to include spacing. The bounding box used for interference computations is extended on each side by an amount equal to half of the desired spacing. The volume of the intersection of two bounding boxes is zero only if they do not intersect and the objects meet the required interbuilding spacing.

constraint: *minimize volume of intersection*
parameter: *volume of intersection of objects' extended bounding boxes*
function: *lower bounded minimize, bounded by zero*

7.2.5. Place buildings near parking lots

This constraint is implemented by minimizing the distance of each building to the nearest parking lot.

constraint: *minimize distance to parking lot*
parameter: *distance from building to nearest parking lot*
function: *lower bounded minimize, bounded by zero*

7.2.6. Place buildings on hills

The manual for site plan development states, "The location of buildings on most tracts can occur on either: 1) flat sites on ridgetops or 2) sloping sites off ridges and adjacent to natural drainageways." Since no identification is made of natural drainageways in this application, the constraint is approximated by maximizing the elevation of the base of each building.

constraint: *maximize building elevation*
parameter: *building elevation (measured at base)*
function: *lower bounded maximize, bounded by lowest terrain elevation*

7.2.7. Avoid unpleasant views from buildings

The manual for site plan development states that flat rooftops may not be overlooked by other buildings. This can occur when a tall building is adjacent to a short building. This constraint attempts to enforce this guideline by minimizing the height difference of neighboring buildings. Two buildings are considered to be in the same neighborhood if they are within 2000 feet of one another (terrain is 10000 by 10000).

> ***constraint:*** *minimize height difference of neighboring buildings*
> ***parameter:*** *magnitude of difference in height of roofline of neighboring buildings*
> ***function:*** *lower bounded minimize, bounded by zero*

7.2.8. Discussion of cost function weighting factors

In the opaque projector application of chapter 6, cost function weights played a crucial role in creating individual products within a product line. In this application, the goal was to generate a single site plan rather than multiple site plans. Constraint weights therefore are not emphasized in this chapter, although they remain important and provide utility. For example, a fundamental conflict occurs between the goal of placing buildings away from roads, and placing buildings on hills. The goal of placing buildings away from roads is largely an aesthetic constraint, while the goal of placing buildings on hills has a more practical basis involving issues such as storm water management. By changing the weightings of these two constraints, the balance between aesthetic and practical issues can be explored.

7.3. Results

This section presents the results of a site planning problem involving fifty buildings and fifty trees. Intermediate results are presented as new constraints refine the problem. While these intermediate results might not be of interest to a site planner, they are presented here to demonstrate the effect of adding each constraint.

Although the results are presented incrementally, the solution technique is not incremental. Each new version of the problem was solved independently.

Figure 7.3 shows the solution to the problem with only road setback and building non-intersection constraints. All constraints can be (and are) satisfied in this problem.

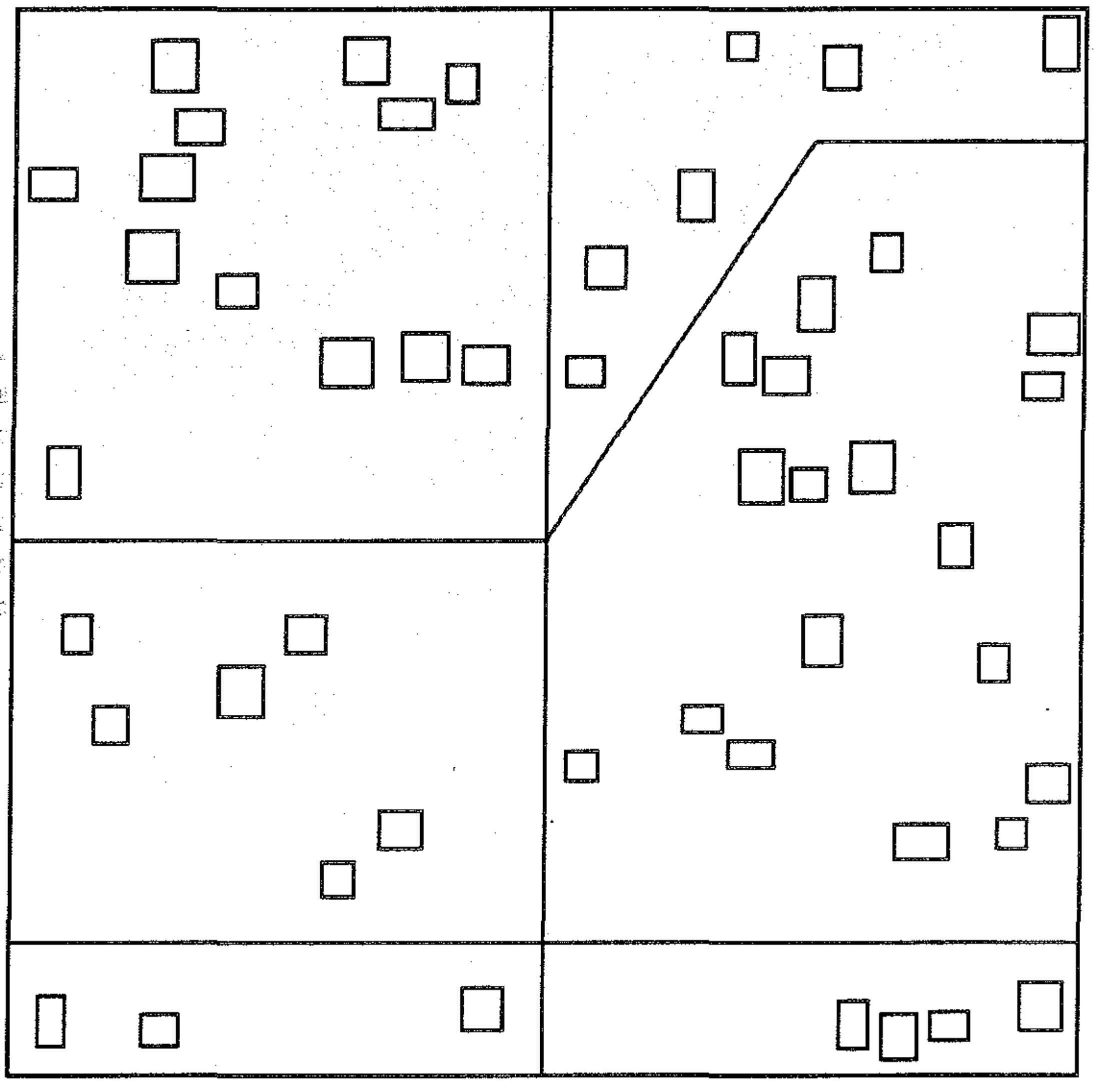


Figure 7.3a: road setbacks and building non-intersection constraints

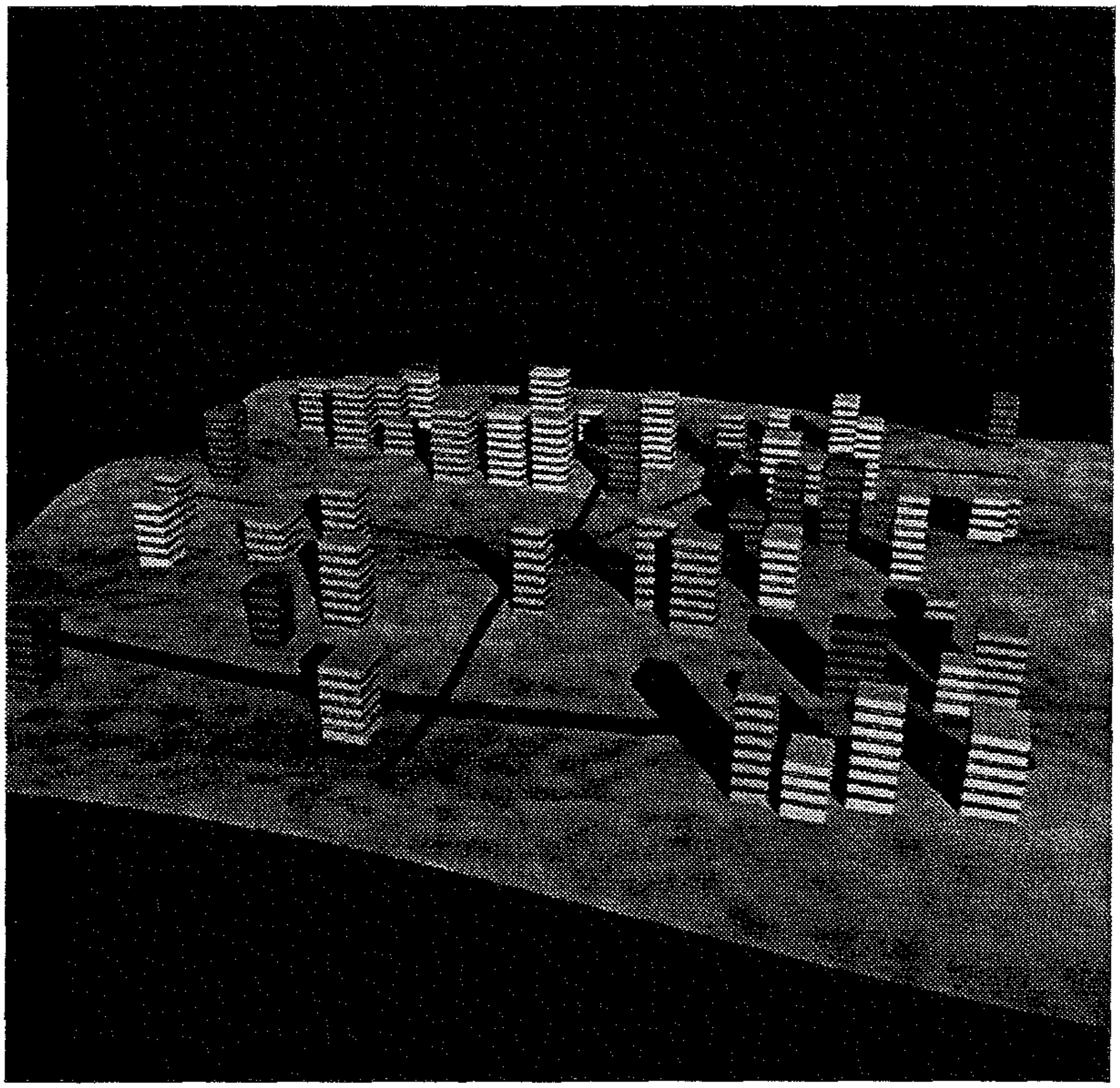


Figure 7.3b: road setbacks and building non-intersection constraints

Figure 7.4 shows the resulting of adding a constraint to maximize the distance of each building from the main roads. This results in a packing problem.

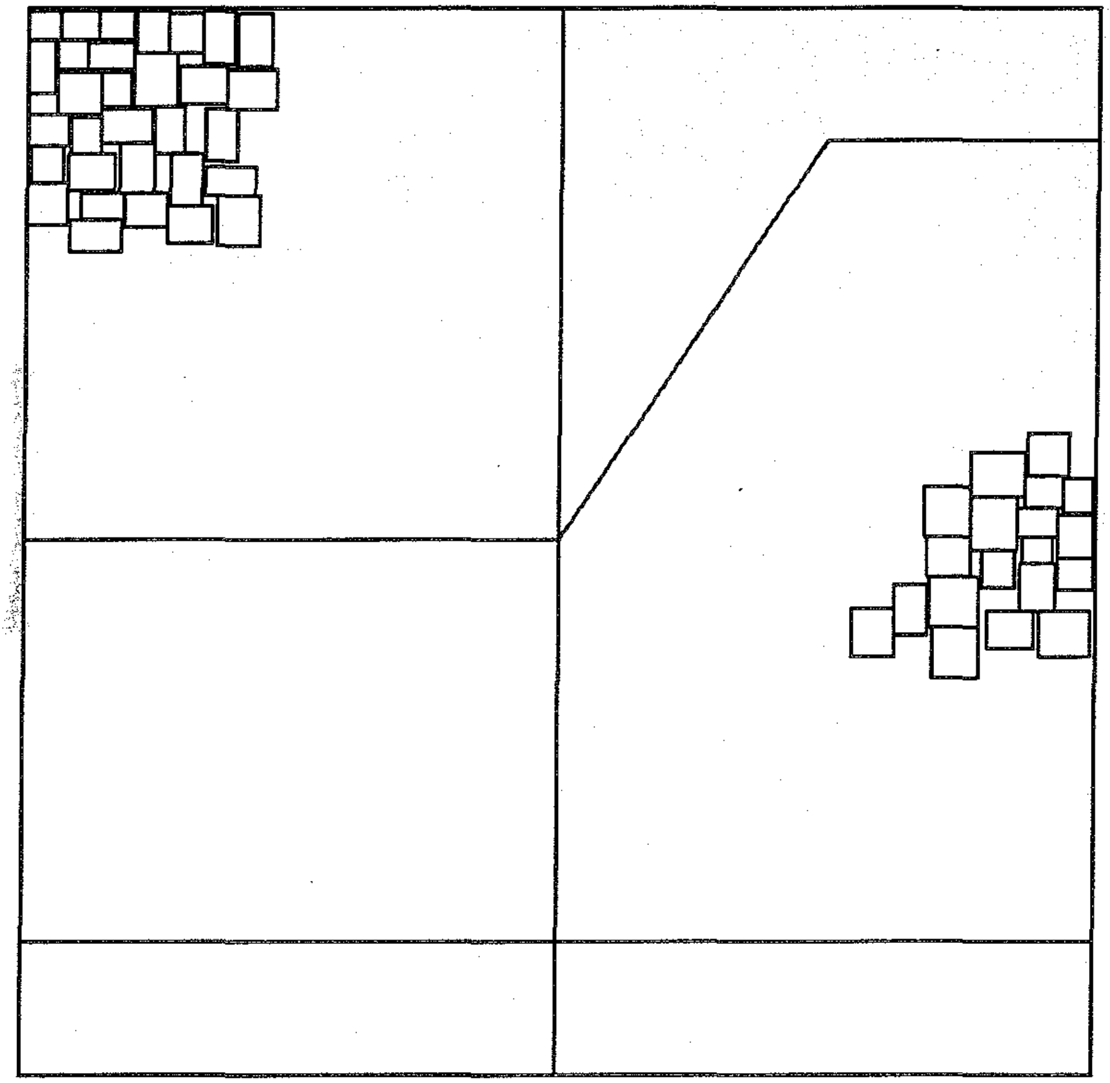


Figure 7.4a: add maximization of road/building distance

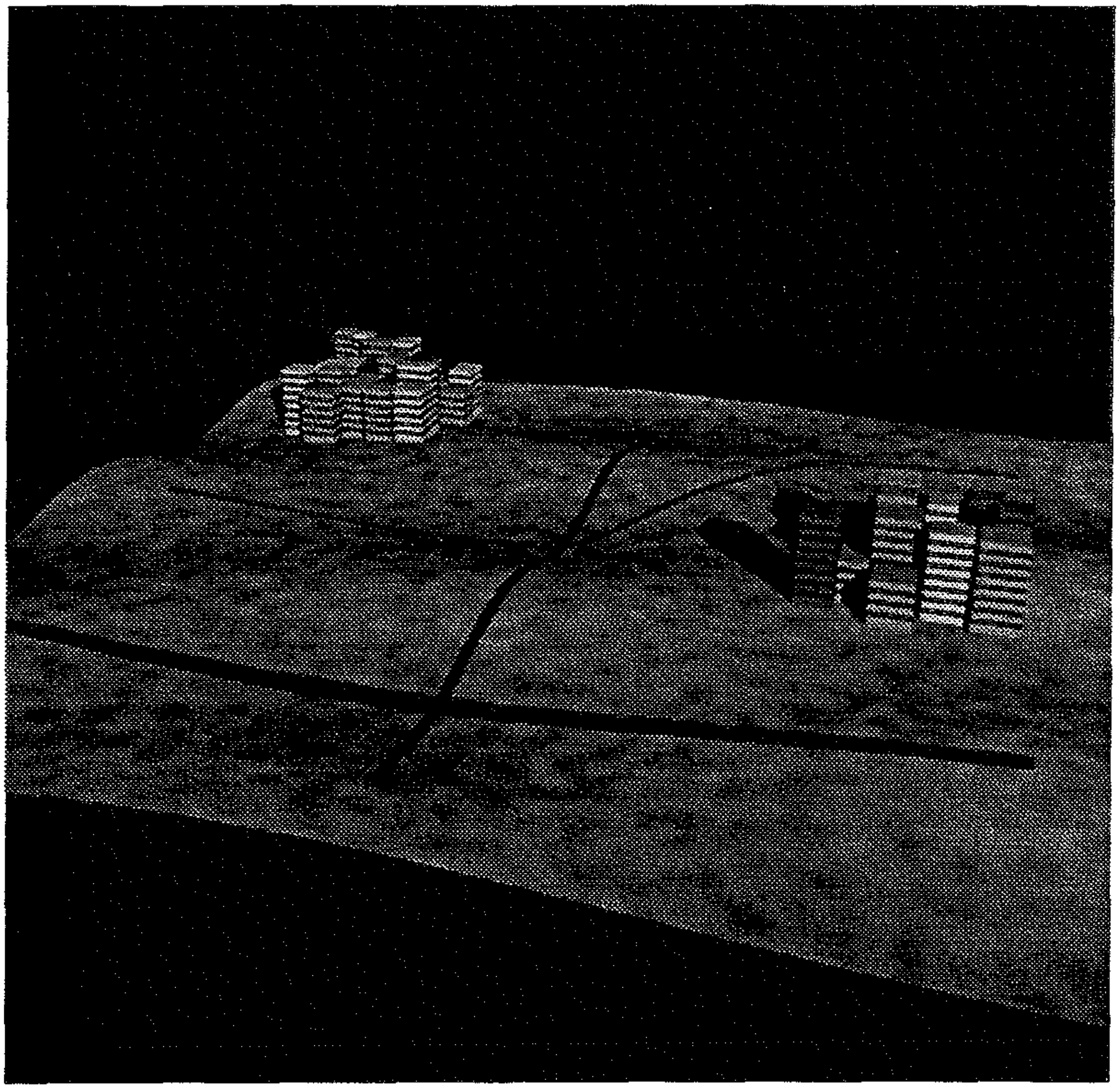


Figure 7.4b: add maximization of road/building distance

Figure 7.5 shows the result of including interbuilding spacing, since no actual site plan would have buildings as close together as in figure 7.4.

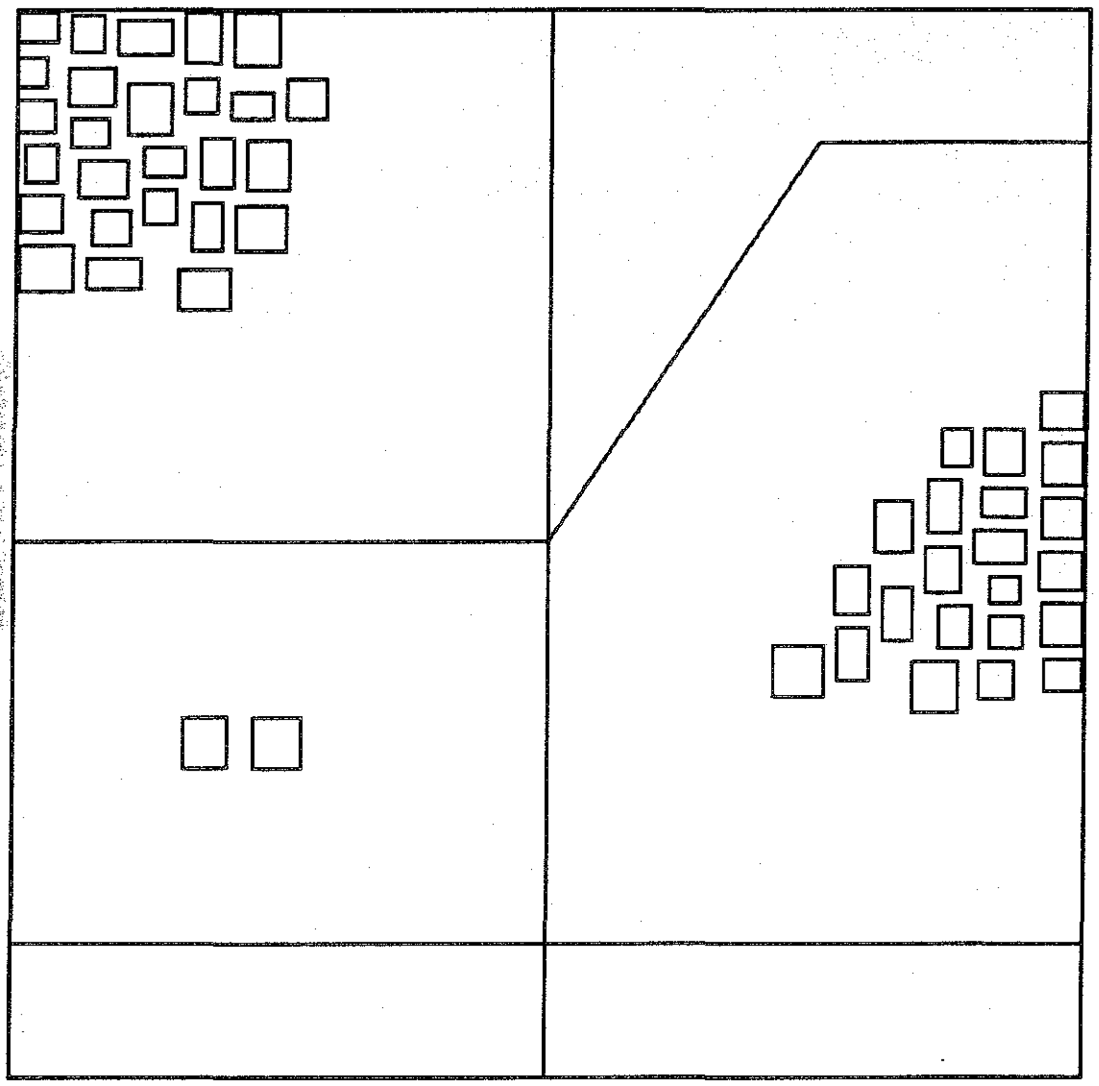


Figure 7.5a: add interbuilding spacing

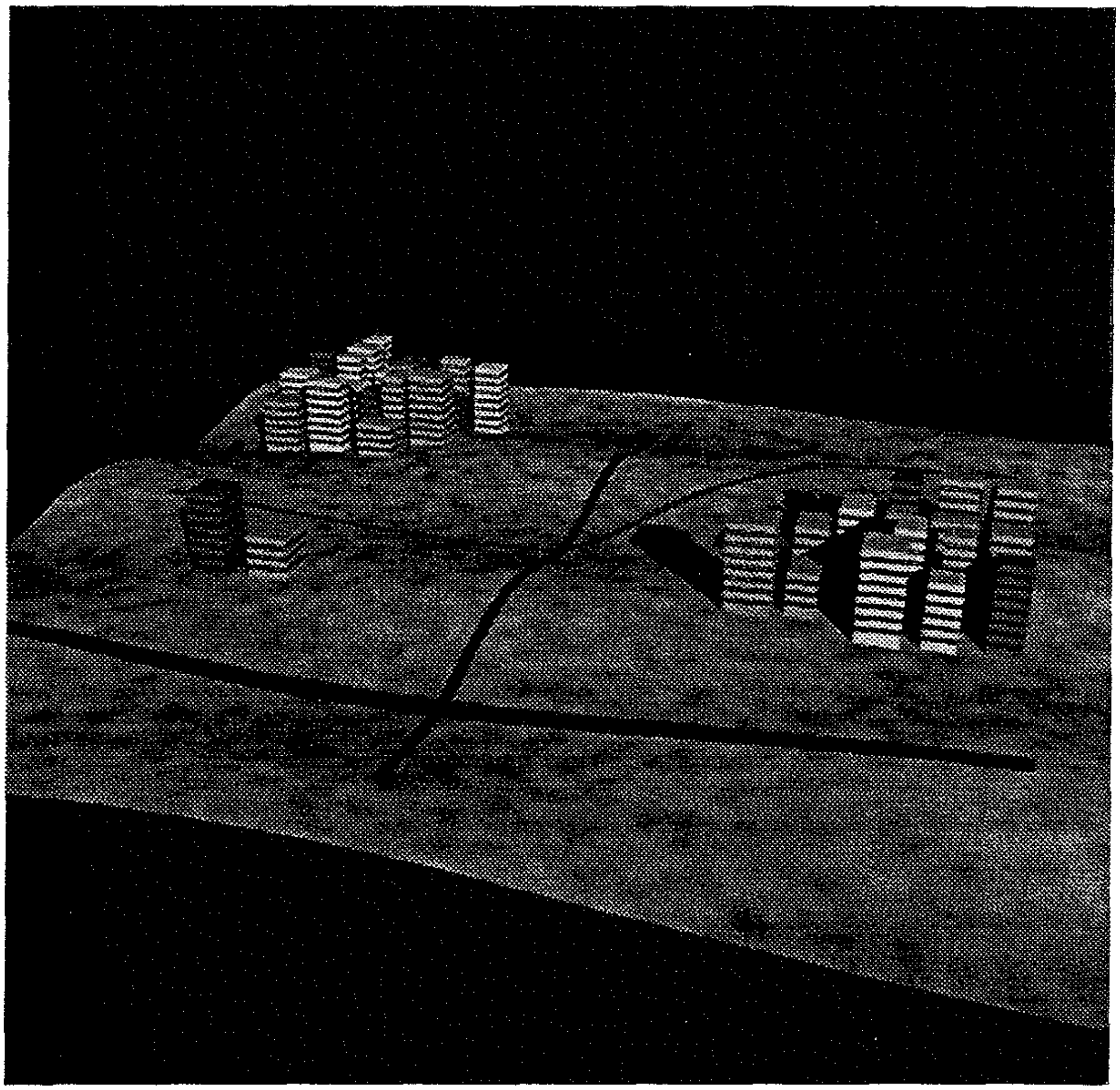


Figure 7.5b: add interbuilding spacing

Figure 7.6 adds parking lots to the problem. These parking lots act like buildings: there is no constraint yet added to minimize the distance to each parking lot.

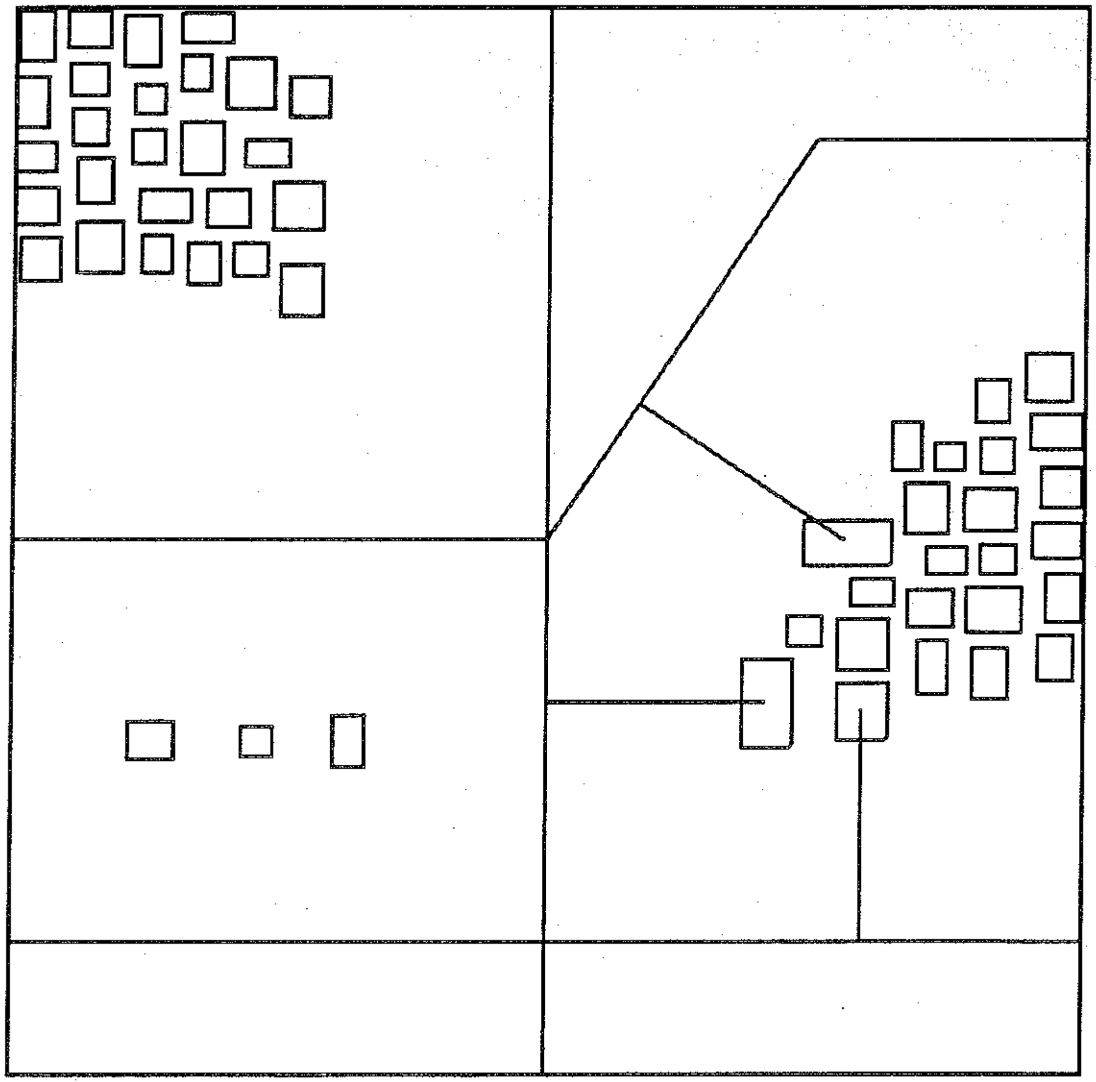


Figure 7.6a: add parking lots

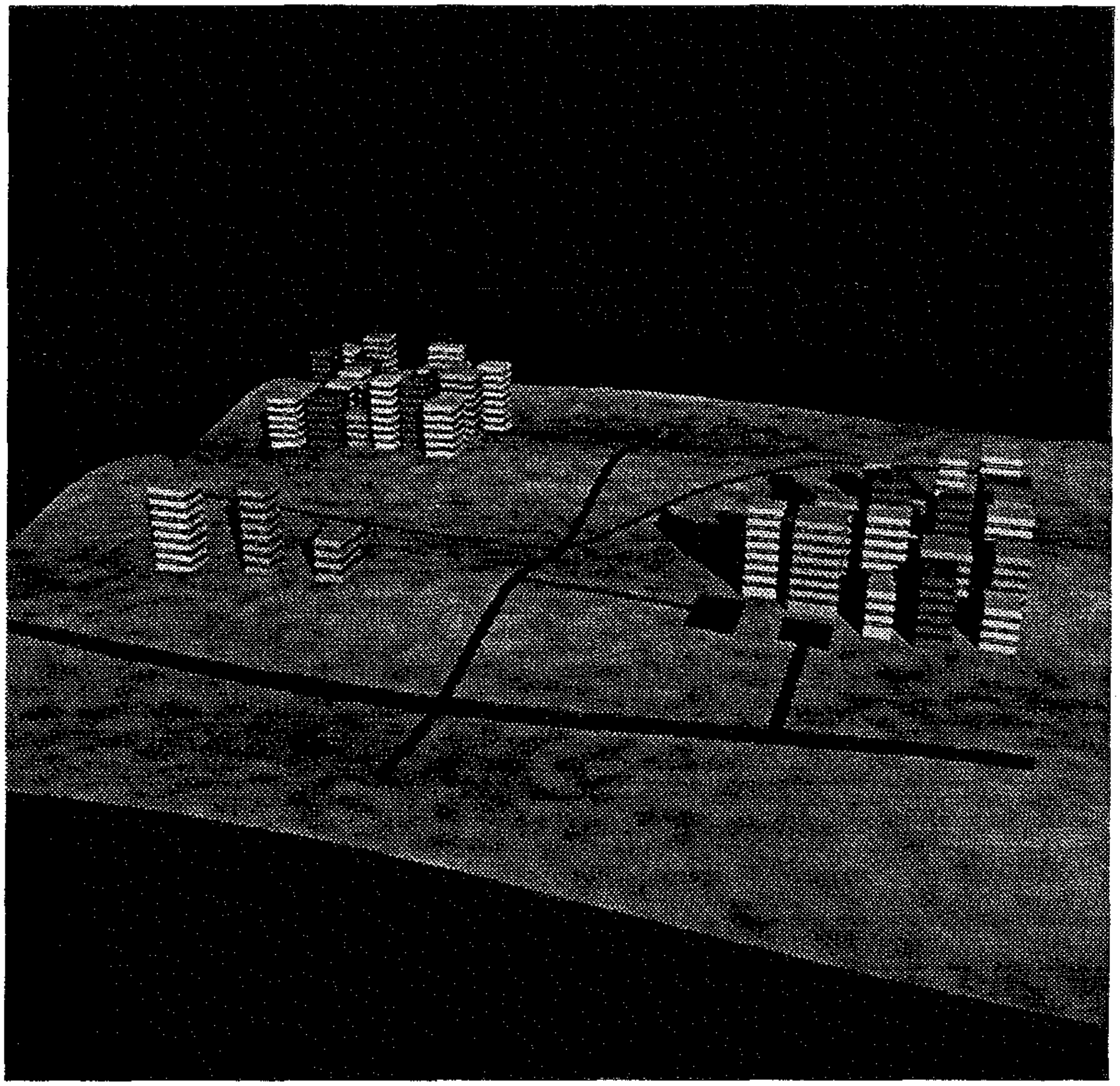


Figure 7.6b: add parking lots

Figure 7.7 adds a constraint to minimize the distance from buildings to parking lots. The distribution of one parking lot to each cluster occurs automatically.

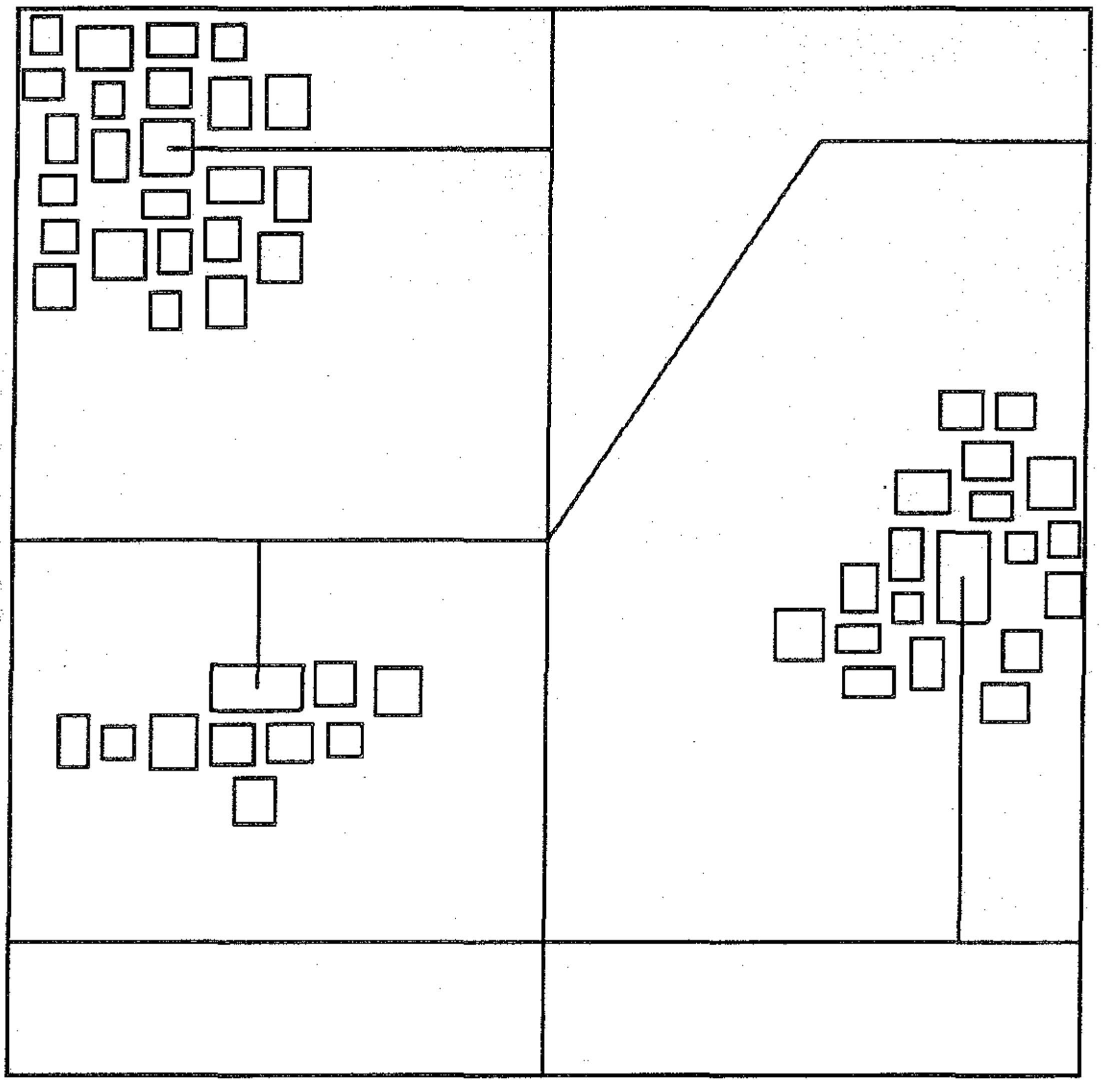


Figure 7.7a: minimize parking lot/building distance

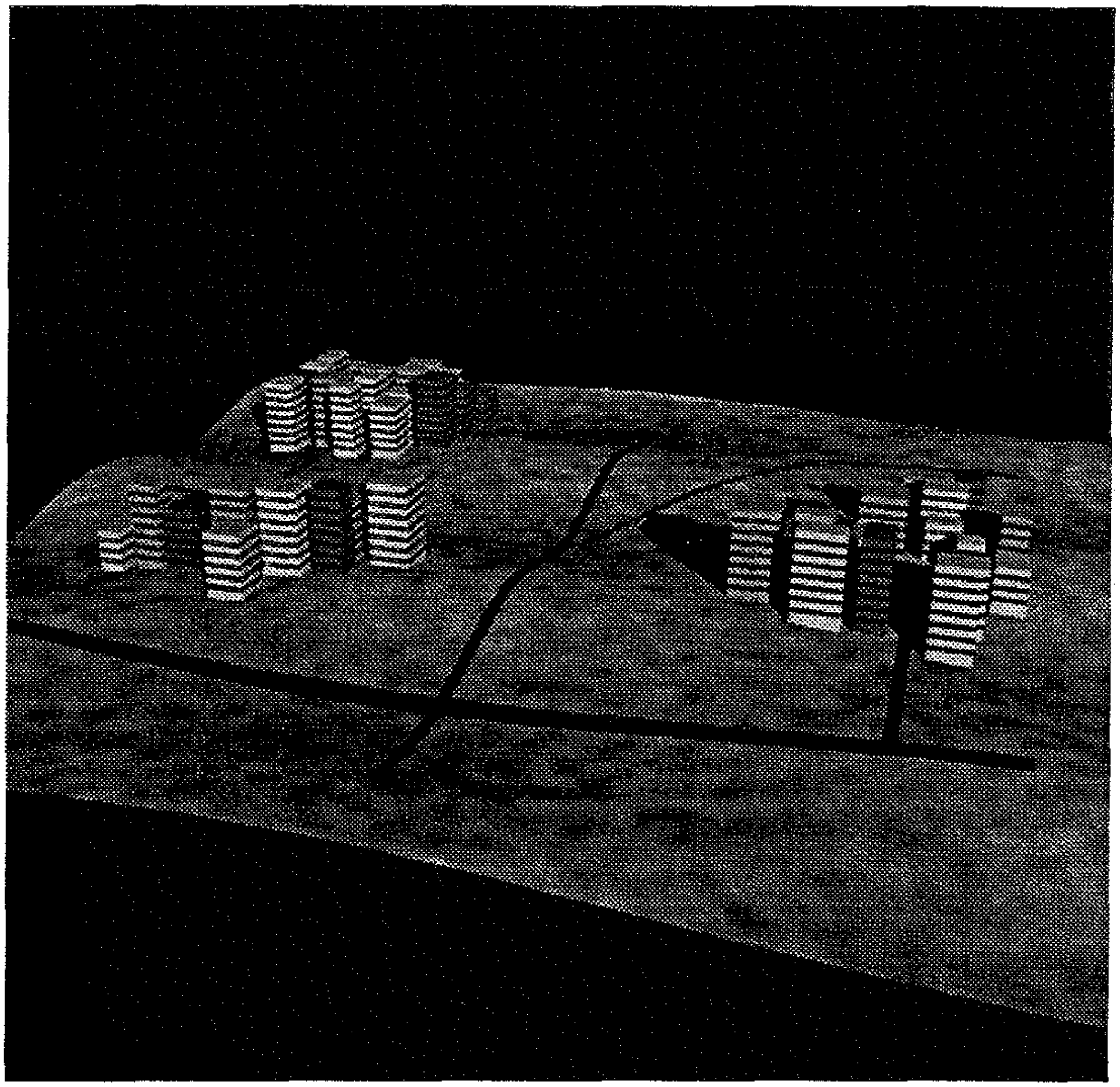


Figure 7.7b: minimize parking lot/building distance

Figure 7.8 adds a constraint to place buildings on hilltops. This pulls some buildings back toward roads in order to place them on hills.

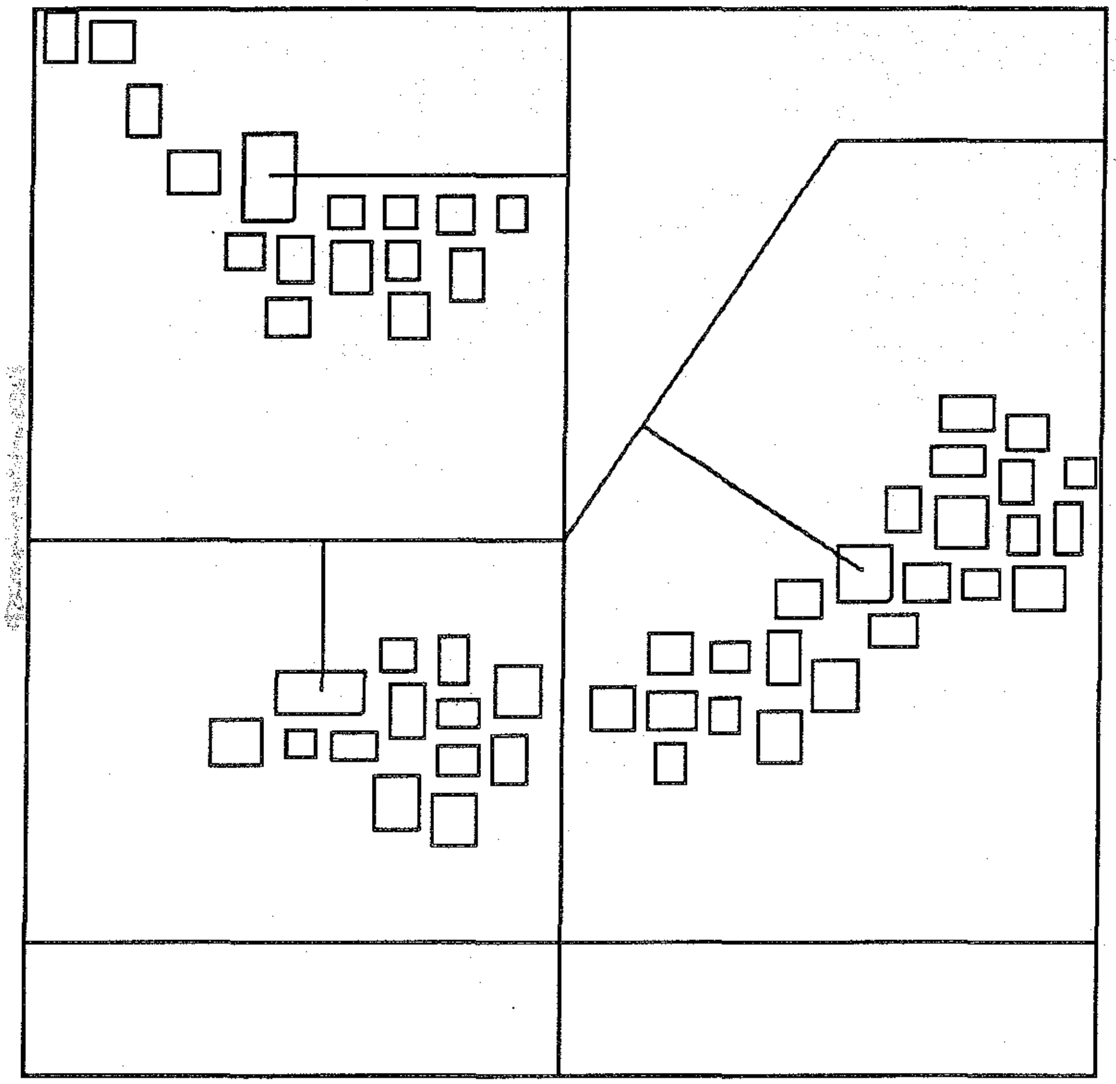


Figure 7.8a: place buildings on hilltops

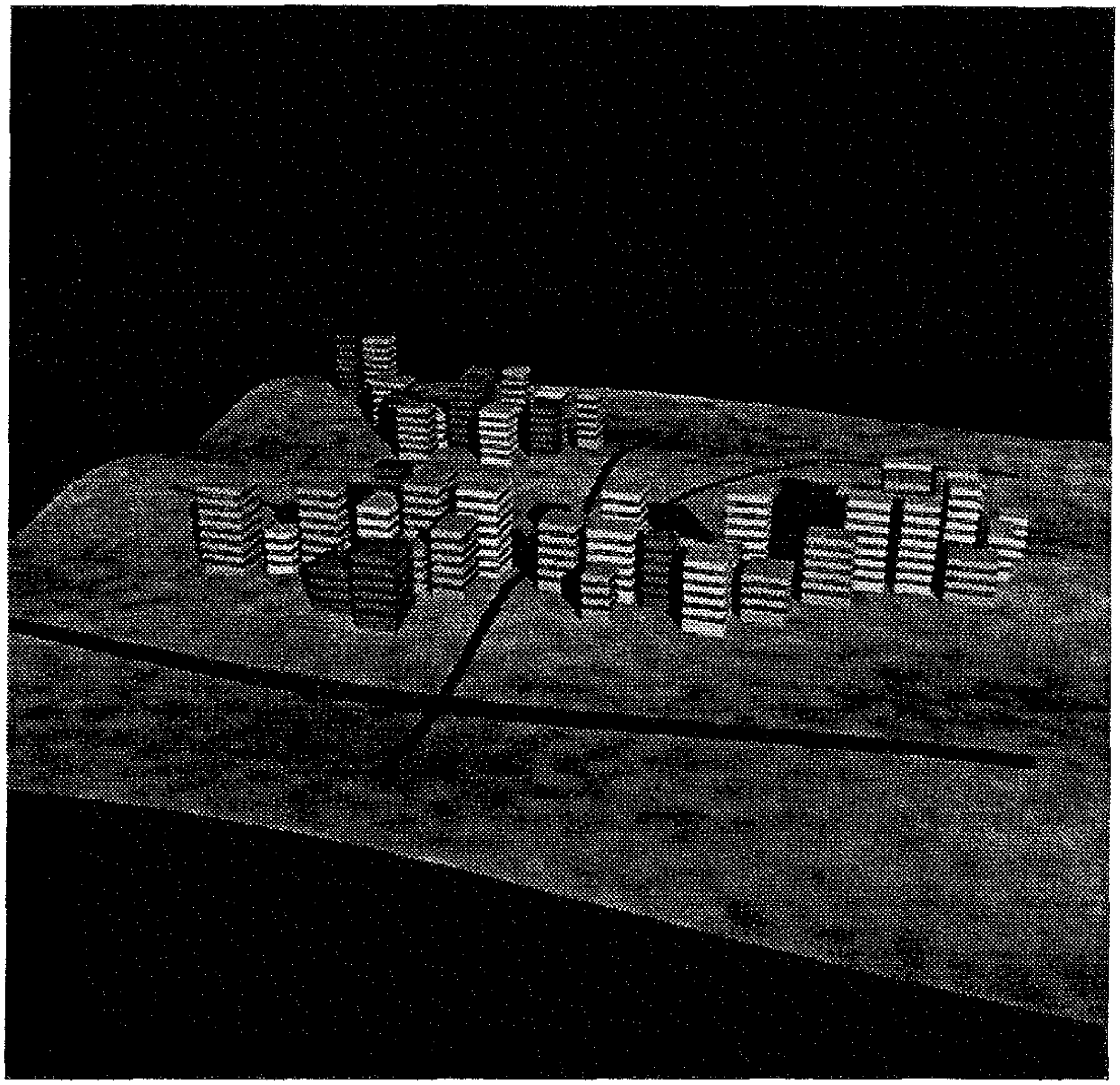


Figure 7.8b: place buildings on hilltops

Figure 7.9 adds trees to the problem. The position of the trees are fixed, so the buildings must be positioned to avoid the trees.

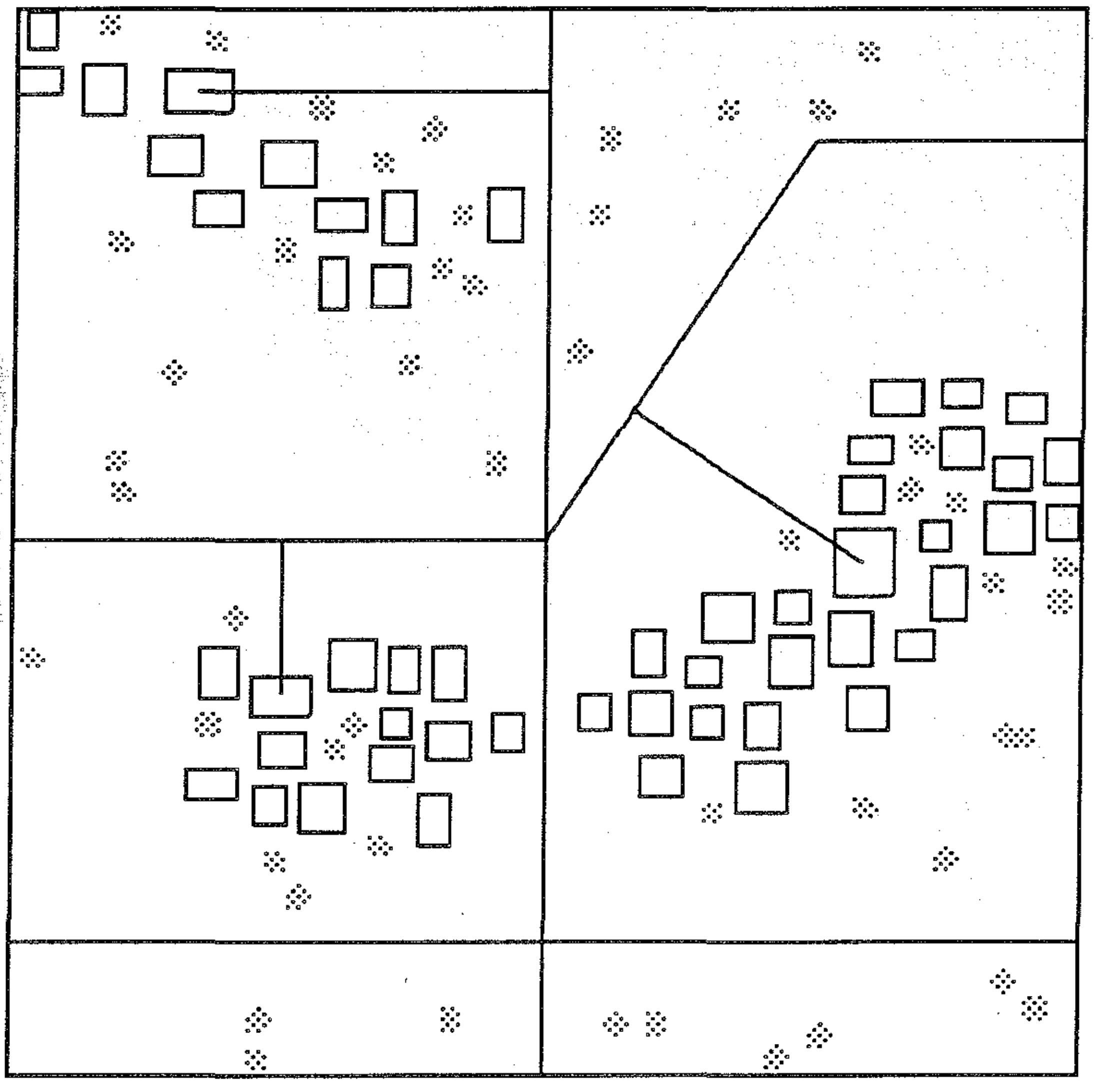


Figure 7.9a: add trees as obstacles

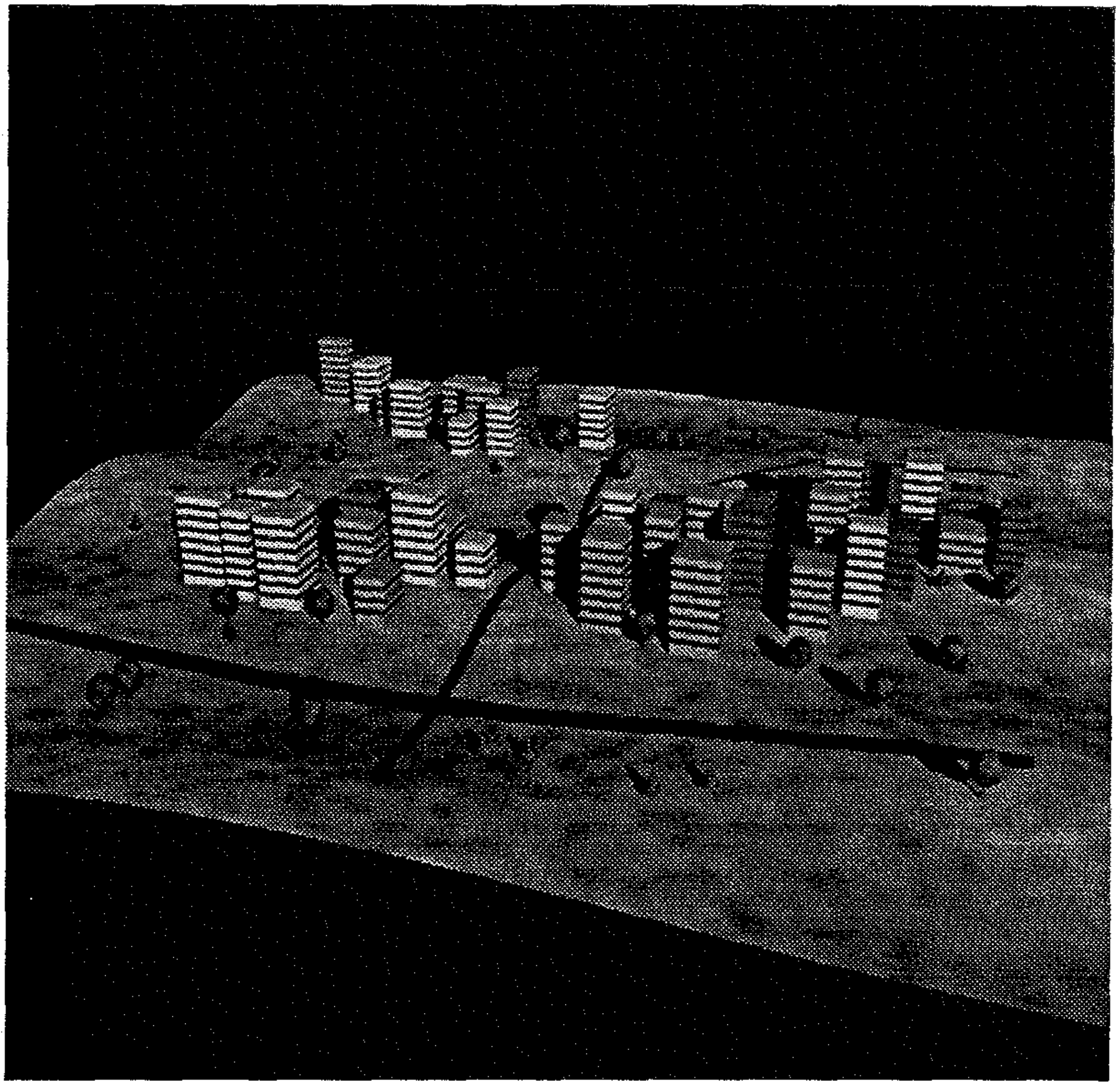


Figure 7.9b: add trees as obstacles

Figure 7.10 temporarily removes the trees and shows the result of adding a constraint to make rooflines of adjacent buildings be the same height.

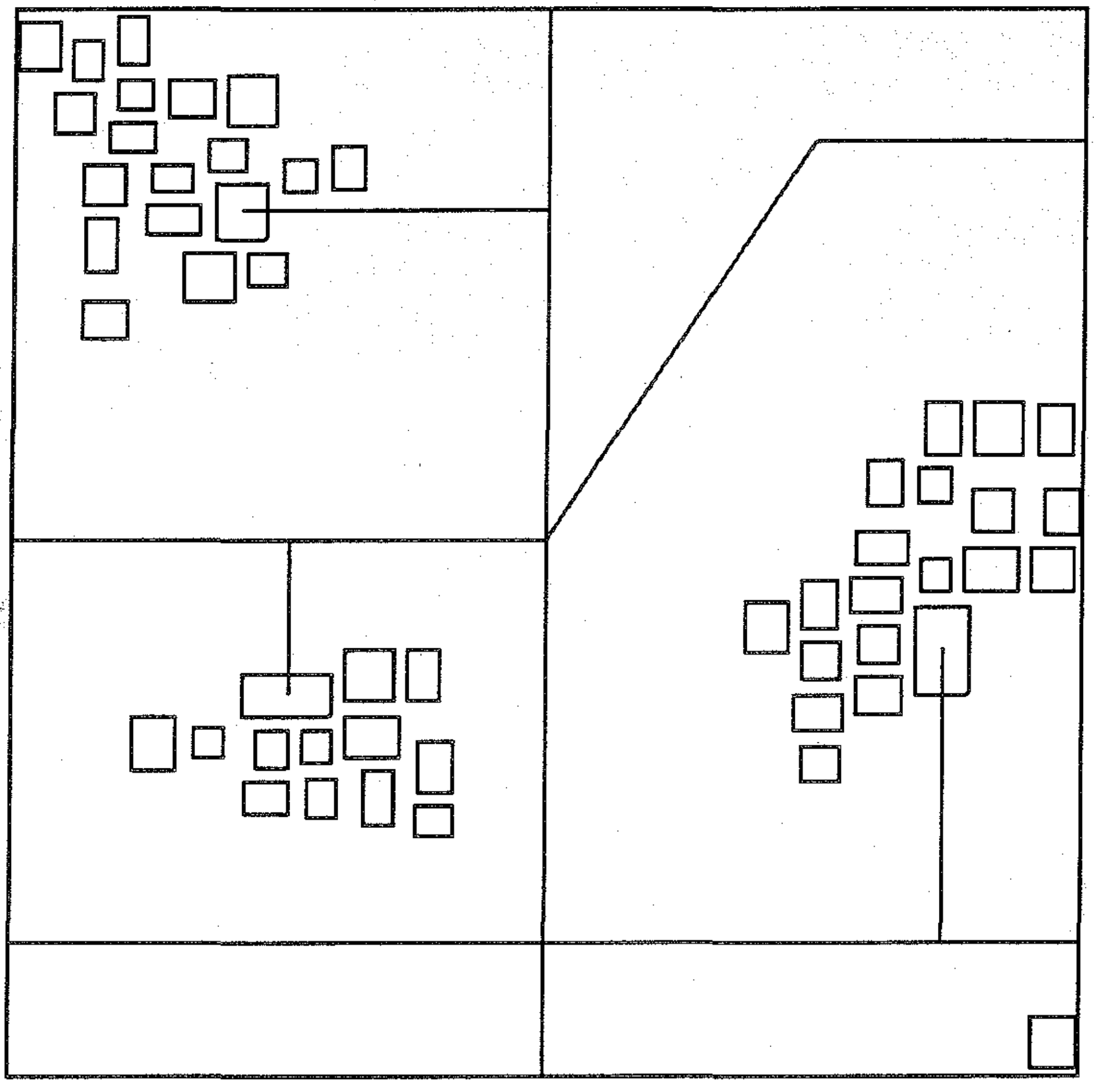


Figure 7.10a: match roof heights locally

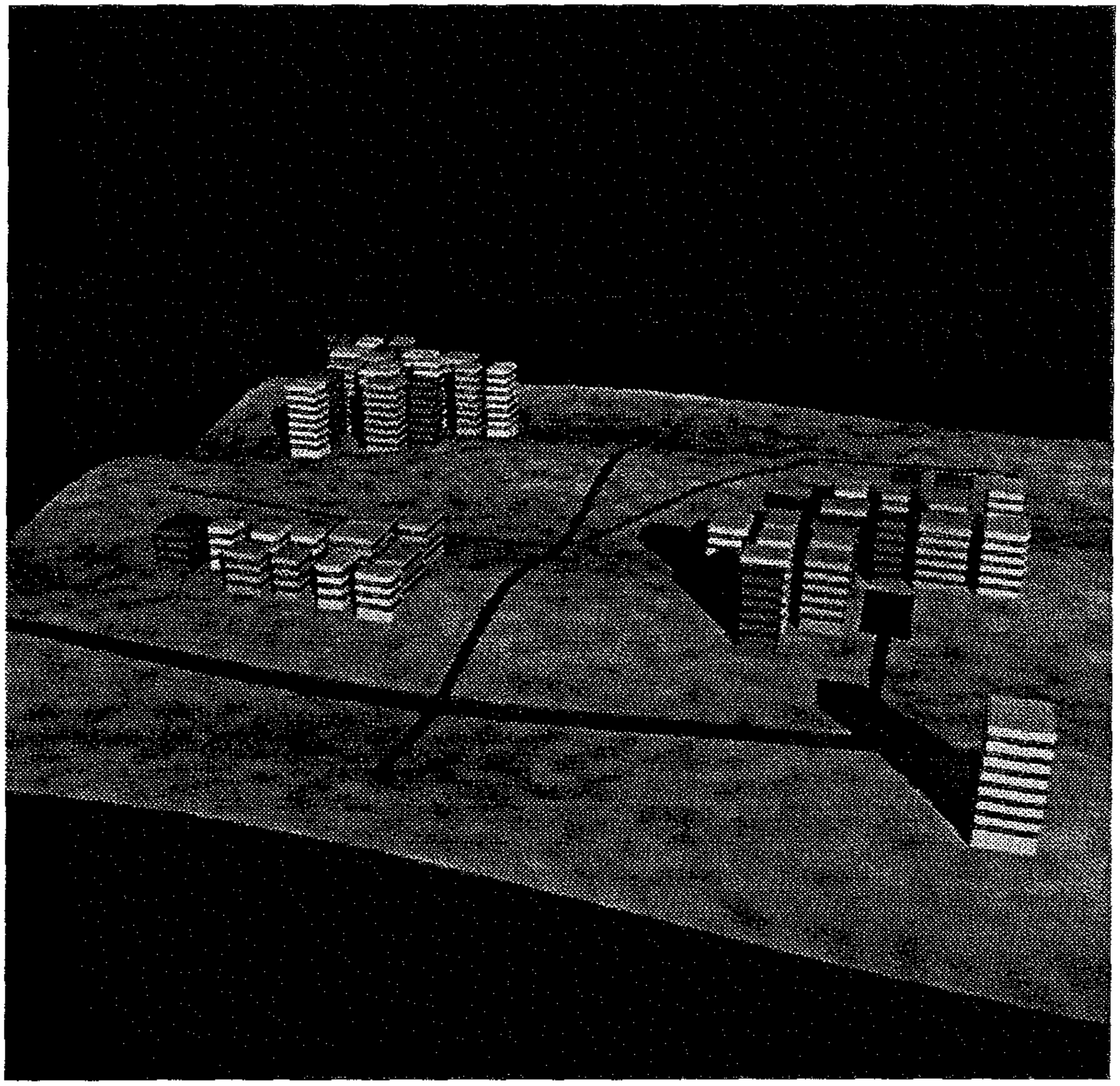


Figure 7.10b: match roof heights locally

Figure 7.11 again adds trees to show the final result of including all constraints and obstacles.

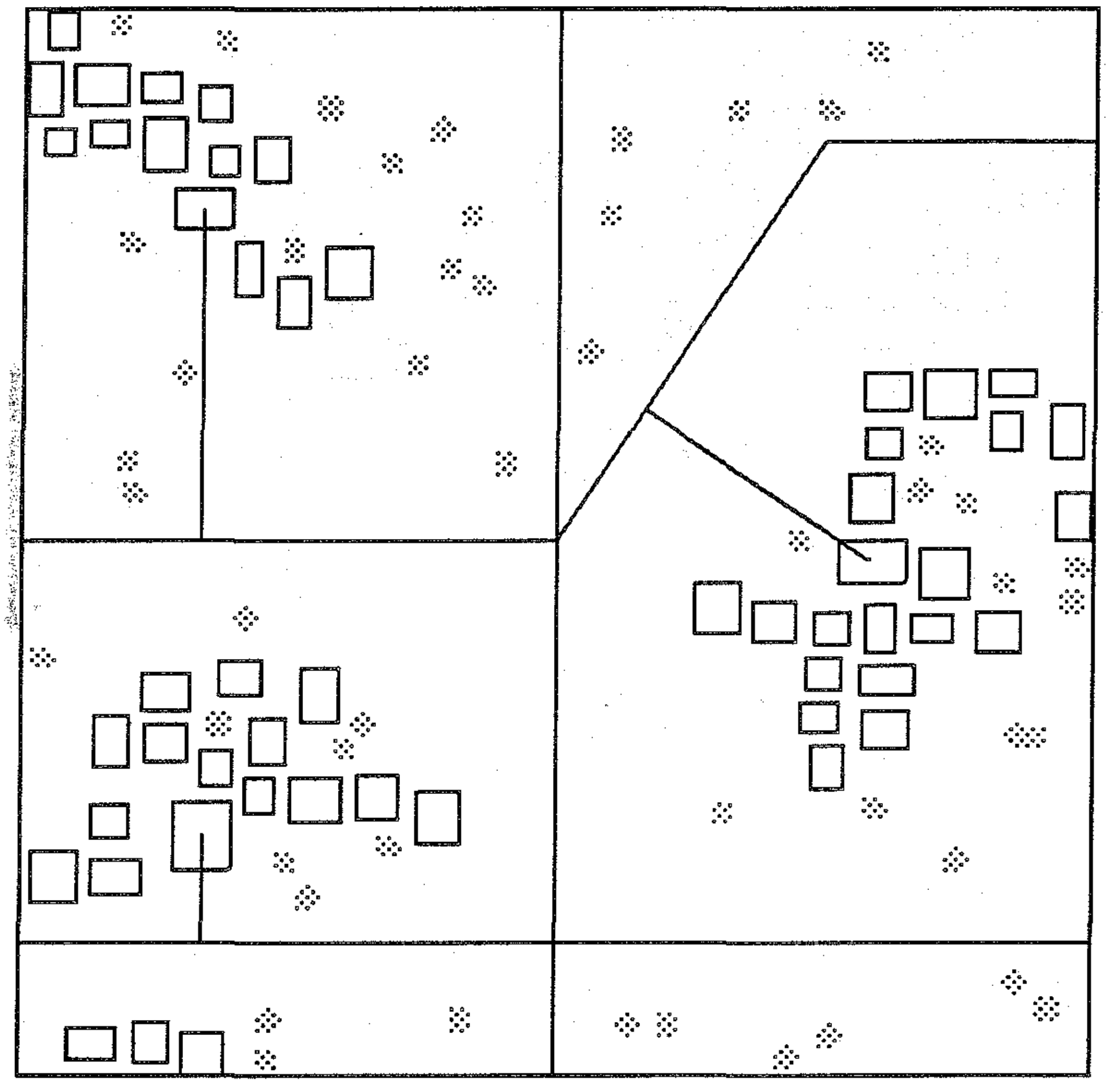


Figure 7.11a: final site plan

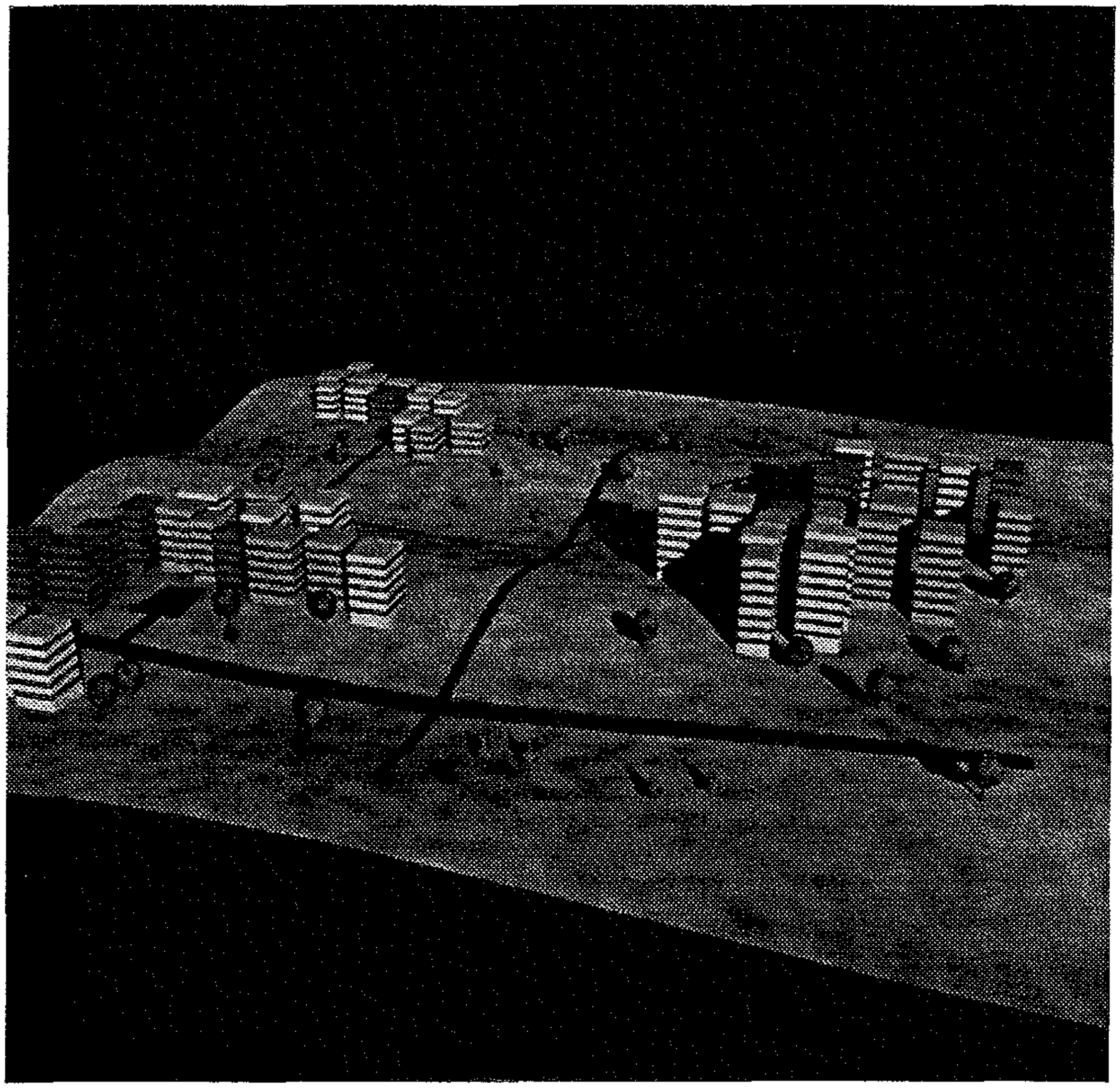


Figure 7.11b: final site plan

7.4. Statistics

This section presents statistics about the constraints and run times for the site planning problems described above.

Although section 7.2 described only seven main constraints, many instances of each of these constraints are used to compose a problem definition. Table 7.1 details the number of instances of each constraint that were used in the examples above. These are dependent on the parameters used in this application: fifty buildings, fifty trees, and three parking lots.

Constraint	# of this type	pairwise tests
Building/building non-intersection	1225	1225
Building/main road setbacks	50	250
Building/access road setbacks	150	150
Maximize distance: building/nearest road	50	250
Building/tree non-intersection	2500	2500
Building/parking lot non-intersection	150	150
Place buildings on hills	50	50
Parking lot/parking lot non-intersection	3	3
Maximize distance: parking lots/nearest road	3	3
Parking lot/main road setbacks	3	15
Parking lot/access road setbacks	6	6
Parking lot/tree non-intersection	150	150
Access road/tree setbacks	150	150
Equalize building height in neighborhood	1	1225
Place parking lots near buildings	1	150
Total	**4492**	**6277**

Table 7.1: constraint statistics for site planning application

The first column of table 7.1 shows the type of constraint. The second column shows the number of instances of constraint, as determined by explicit cost function calls within the application. Some of the constraints were implemented as composite constraints, however. For instance, the constraint to place buildings near parking lots was implemented as a special purpose constraint and therefore was counted only once in column two. Each evaluation of that constraint requires the evaluation of the distance between each parking lot and each building, a total of 150 pairwise comparisons. Furthermore, the main roads are represented internally as a single road network, and so each setback constraint requires only one call even though there are five main road segments.

The third column shows the total number of pairwise constraints for each of the main constraints. This provides a more accurate figure of the number low-level constraints being considered. Even with the expanded constraint count, however, non-intersection constraints comprise over half of the total count.

Problem	CPU seconds (DEC 3100)	# of constraints	# of iterations
Figure 7.3	235.1 (2 min, 55.1 sec)	1275	6800
Figure 7.4	5151.7 (1 hr, 25 min, 51.7 sec)	1325	150000
Figure 7.5	5035.0 (1 hr, 23 min, 55.0 sec)	1325	150000
Figure 7.6	6302.1 (1 hr, 45 min, 2.1 sec)	1640	150000
Figure 7.7	6765.9 (1 hr, 52 min, 45.9 sec)	1641	150000
Figure 7.8	6712.4 (1 hr, 51 min, 52.4 sec)	1691	150000
Figure 7.9	11374.6 (3 hr, 9 min, 34.6 sec)	4491	150500
Figure 7.10	19621.0 (5 hr, 27 min, 1.0 sec)	1692	250400
Figure 7.11	78463.6 (21 hr, 47 min, 43.6 sec)	4492	724000

Table 7.2: CPU times for site planning problems

Table 7.2 shows the amount of time used in solving each of the site plan problems. The time is related of the number of constraints, complexity of constraints, number of iterations evaluated, and the actual configurations evaluated. The program was set to terminate upon reaching a particular temperature or a fixed number of iterations.

The results indicate that the solution method is not suitable for interactive use on this particular platform. If we assume a five-hour run time for a typical site planning problem using this implementation, a thousand-fold increase in processor speed is necessary to reduce run time to a figure that might allow interactive use (eighteen seconds).

On the other hand, a five-hour run time is not much of an investment when one considers the costs and time typically involved in designing a site plan.

7.5. Evaluation

In general, the solutions presented above are non-optimal but are close to optimal. The following is a brief evaluation of figures 7.3 through 7.11:

Figure 7.3: All constraints are satisfied in this problem, so it is an optimal solution.

Figure 7.4: An optimal result would probably have the buildings packed together with smaller gaps. The packing is dense, however, in that gaps tend to be much smaller than buildings.

Figure 7.5: The interbuilding spacing resulted in two buildings being separated in the lower left portion of the terrain. With an optimal packing, it may have been possible to place these buildings further from the road (as part of the two main clusters).

Figure 7.6: The addition of parking lots has little effect on the layout of buildings. Parking lots are placed near the outside of the cluster to avoid interference involving setback constraints on the access roads.

Figure 7.7: The requirement of having buildings near parking lots forces clusters near each of the three parking lots. Buildings tend to surround parking lots, as one would expect. The cluster in the lower left portion of the terrain appears non-optimal, though relocation of a single building would correct that.

Figure 7.8: The results again appear to be reasonable. Buildings are placed on hilltops, yet still maintain proximity to parking lots.

Figure 7.9: There are no surprises here. The buildings are placed so as to avoid the trees. The results are quite similar to figure 7.8 with the exception of tree avoidance.

Figure 7.10: The goal to match building heights results in the building in the lower right portion of the terrain being placed far away from a parking lot. This is clearly a non-optimal position (though it is locally optimal). All other buildings appear to be reasonably placed.

Figure 7.11: The final solution reflects the influences of all constraints and has no obvious errors. All forms of non-intersection constraints are satisfied. Goal constraints, such as placement of buildings on hills and away from roads, are clearly reflected in the solution.

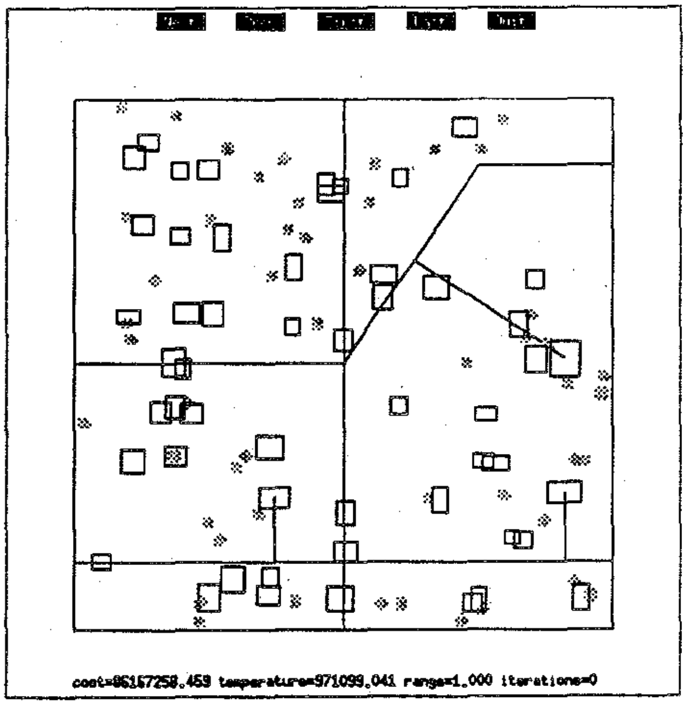
It would be interesting to compare these generated results with optimal solutions. Unfortunately, there is no simple way of generating optimal solutions for comparison. The alternative is to examine the results for compatibility with the specified constraints. This section has performed such an evaluation and has in general found the solutions to conform to the desired constraints.

7.6. Convergence to final solution

Figures 7.12 through 7.15 show the convergence of the final site plan depicted by figure 7.11. The progression shows the configuration at each of the following number of iterations:

- 0 iterations
- 88,800 iterations
- 184,800 iterations
- 280,800 iterations
- 376,800 iterations
- 472,800 iterations
- 568,800 iterations
- 724,000 iterations

At high temperatures (figure 7.12), the configurations violate many constraints and no structure is readily noticeable. As the system begins to cool (figure 7.13), the general distribution of parking lots to clusters becomes noticeable, though buildings still appear randomly distributed. Upon further cooling (figure 7.14), the organization of buildings into clusters becomes apparent, though a few stray buildings remain far from any parking lot. In the final stages of cooling (figure 7.15), the buildings settle into a packed configuration.


```
cost=86157258.459 temperature=971099.041 range=1.000 iterations=0
```

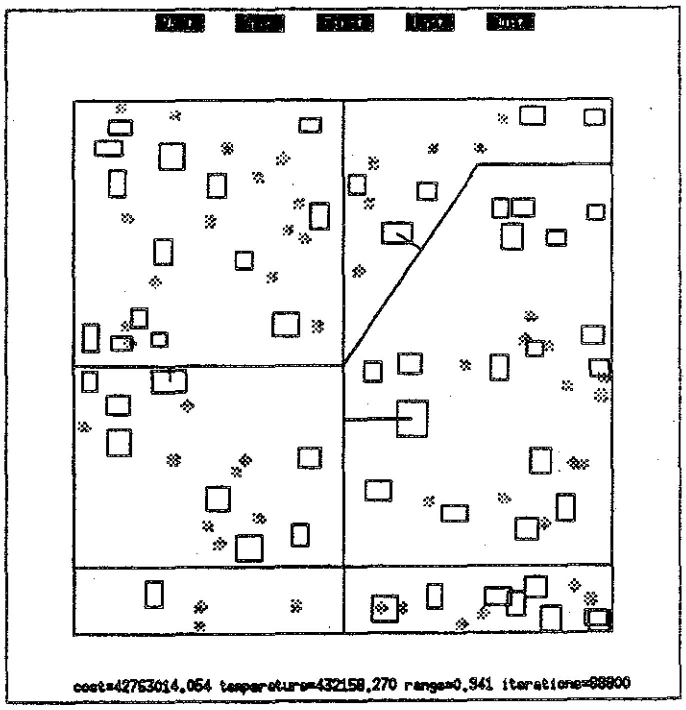
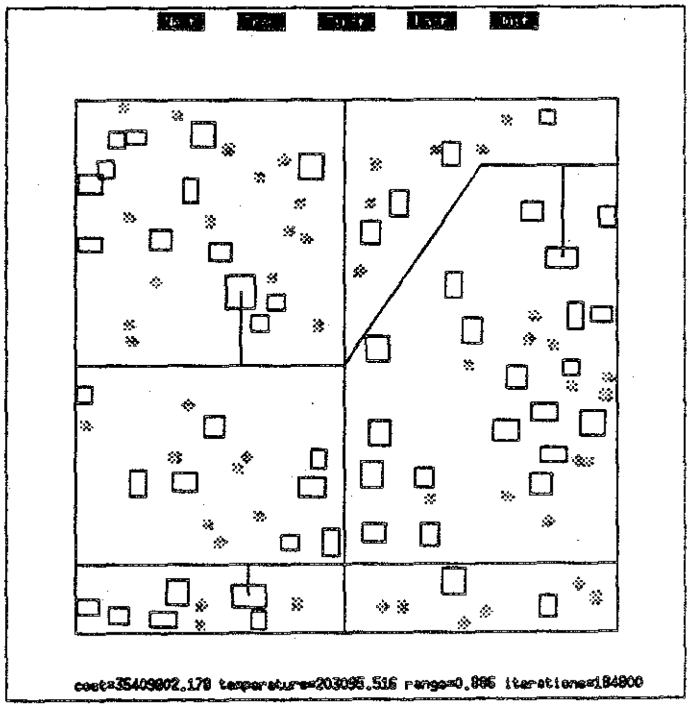
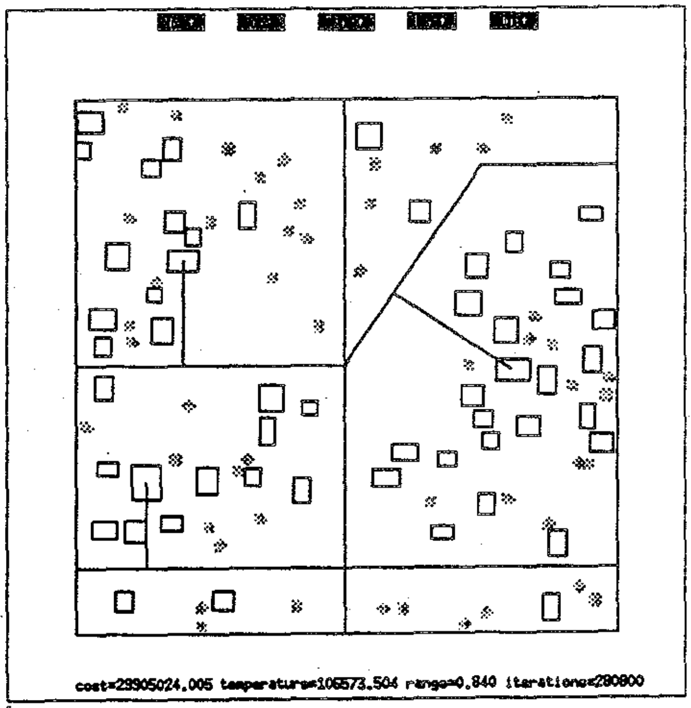

```
cost=42763014.054 temperature=432158.270 range=0.941 iterations=88800
```

Figure 7.12: site plans at high temperature

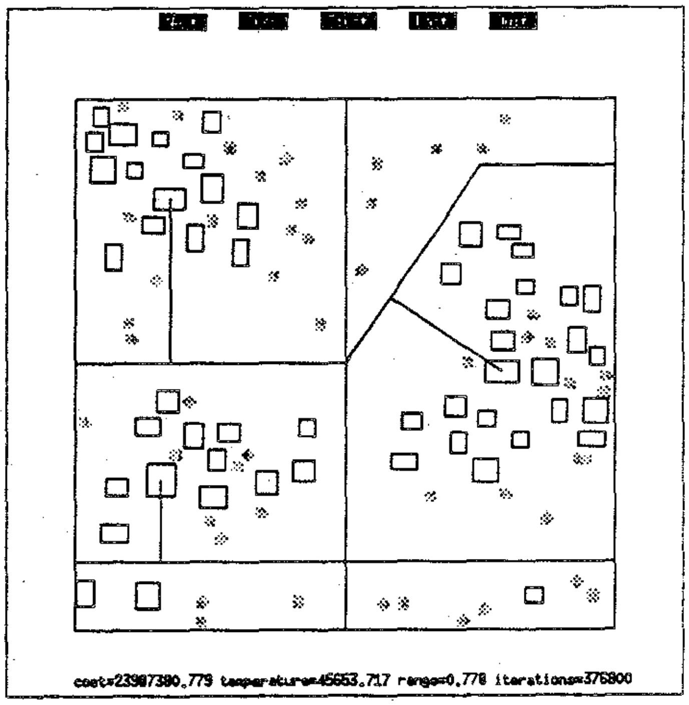


```
cost=35409802.178 temperature=203095.516 range=0.886 iterations=184800
```

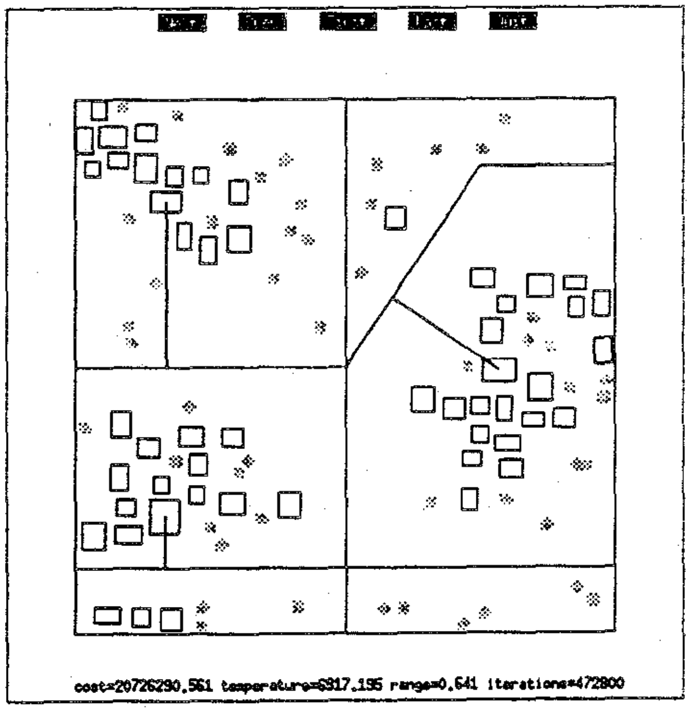


```
cost=29905024.005 temperature=106573.504 range=0.840 iterations=280800
```

Figure 7.13: site plans as system begins to cool

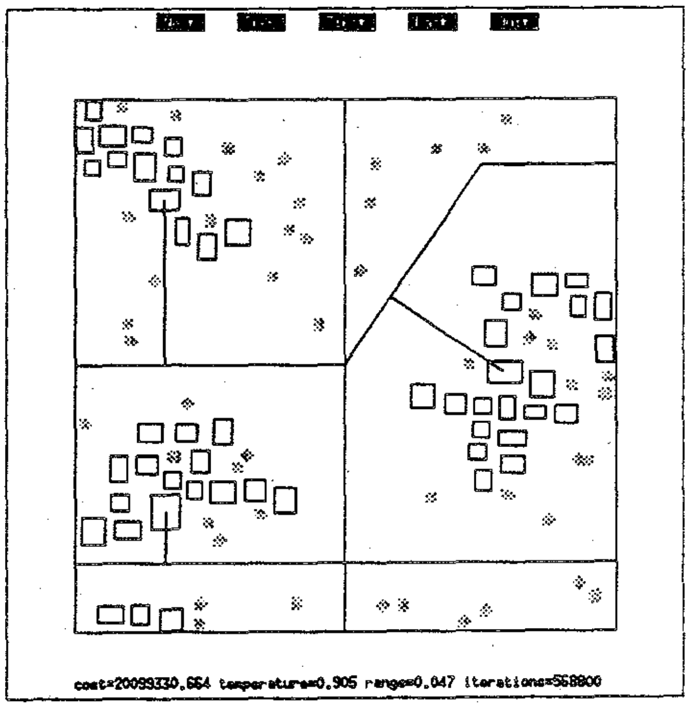


```
cost=23987380.779 temperature=45653.717 range=0.778 iterations=376800
```

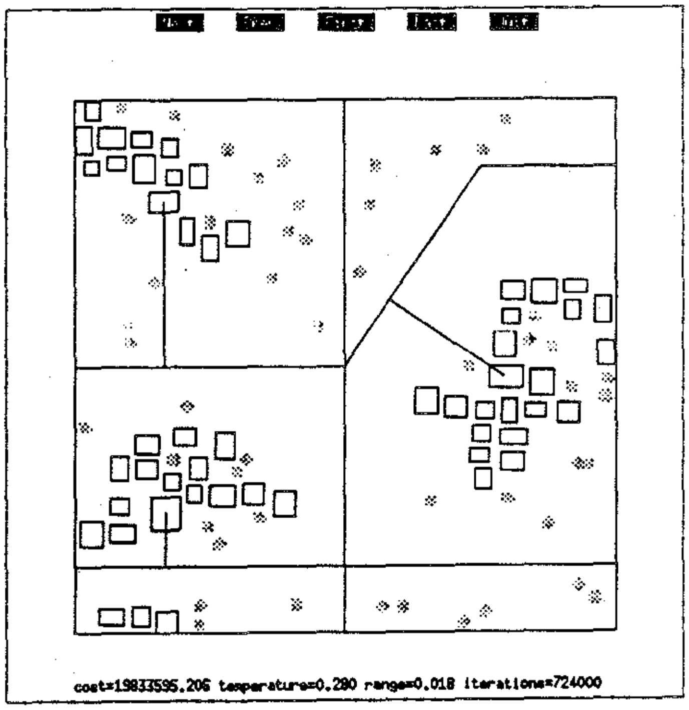


```
cost=20726290.561 temperature=6317.195 range=0.641 iterations=472800
```

Figure 7.14: site plans after further cooling



```
cost=20099330.664 temperature=0.905 range=0.047 iterations=568800
```



```
cost=19833595.206 temperature=0.280 range=0.018 iterations=724000
```

Figure 7.15: site plans at final stages of cooling

7.7. Summary

This chapter has documented the use of CONTEST in site planning. Unlike the opaque projector application, which was created to generate a family of designs, the site planning application was oriented toward the design of a single site plan.

In the opaque projector project, the part inventory was fixed and the variables were the consumers preferences, reflected by constraint weightings. In site planning, constraints tend to remain constant, while the variables are the size, number, and shape of the relevant objects: terrain, roads, buildings, trees, and parking lots.

A constructive approach was used to reach the final solution. This approach was useful in designing and presenting the constraints, because the effect of each constraint was apparent. A site planner, however, would probably be interested in only the final solution, assuming that the solution was satisfactory.

A straight-forward design method was followed in this application: 1) specification of the problem, 2) identification of objects and their parameters, 3) encoding of constraints, and 4) evaluation of results. The resulting site plan properly reflects the tradeoffs specified by the constraints.

A wide variety of constraints were encoded as cost functions in this application. The diversity of the constraints in both this application and the opaque projector application demonstrates the versatility of the cost function approach.

Chapter 8

Discussion

This chapter compares CONTEST with several other systems that are representative of particular constraint satisfaction techniques. This comparison is then used as an introduction to the contributions of this research. Finally, the implementation of CONTEST is described.

8.1. Comparisons with other systems

The systems compared are Borning's ThingLab [Borning, 1979], Van Wyk's Ideal [Van Wyk, 1990], Brüderlin's constraint system [Brüderlin, 1986], Brown's expert system [Brown, 1986], Barzel and Barr's physically-based modeling system [Barzel, 1988], and CONTEST. These systems were chosen for comparison because they each use a different solution method for solving modeling problems.

These systems are analyzed based on the following criteria: 1) solution method, 2) whether the system is capable of satisfying constraints globally or only locally, 3) restrictions on constraint complexity, 4) restrictions on type of constraint, 5) user interaction method, and 6) accuracy of results.

8.1.1. ThingLab

ThingLab uses multiple solvers: propagation of known states or degrees of freedom where possible, and relaxation to solve remaining constraints (see figure 8.1). Relaxation finds only locally optimal solutions. Constraints in ThingLab may be arbitrarily complex, but if relaxation is necessary the constraints are approximated by linear equations. ThingLab can handle non-geometric constraints in addition to geometric constraints. ThingLab is an interactive Smalltalk based environment, and produces accurate, repeatable results.

8.1.2. Ideal

Ideal uses a single constraint solver capable of solving only linear constraints (see figure 8.2). It finds exact, global solutions to properly formulated problems. It is designed for dealing with geometric constraints. Ideal is not interactive; the user creates a data file describing a picture and then presents this input to Ideal for processing.

8.1.3. Brüderlin's system

Brüderlin's system uses a fixed set of geometric constraints, and cannot be easily extended to incorporate new basic constraints. The system runs as an interactive Macintosh application. The user interacts with the system by selecting constraints from a menu and providing the parameters to those constraints. The constraints are represented as Prolog predicates, which are reduced symbolically to a numerically solvable form (see figure 8.3). The system finds exact, globally optimal solutions to properly formulated problems.

8.1.4. Brown's system

Brown's expert system approach uses a hierarchy of cooperating agents to produce globally optimal designs. Each low-level agent solves local tasks, while the higher-level specialists direct these local agents to yield a global solution (see figure 8.4). The agents model human problem solving procedures, so as long as the appropriate solution knowledge is encoded in the expert system, constraints may be arbitrarily complex. Constraints can geometric or non-geometric. The user creates a design by describing the constraint problem in DSPL, a LISP-like language. The success and accuracy of the results is dependent upon the quality of the knowledge base.

8.1.5. Barzel and Barr's system

Barzel and Barr's system works by converting geometric constraints into forces which in turn move objects to solve constraints (see figure 8.5). The constraint forces provide local convergence to a solution. This system supports constraints which are evaluated in terms of positions and orientations of geometric objects. The user specifies a problem by entering constraint definition commands in a LISP environment.

8.1.6. CONTEST

CONTEST represents constraints as cost functions which are then optimized (see figure 8.6). The optimization technique searches for a global optimum, though there exists no guarantee that the optimum will be found. Constraints may be arbitrarily complex; the only requirement is that the cost function for each constraint be capable of expressing the quality of each configuration as a single scalar value. Constraints can deal with geometric or non-geometric properties. The user specifies a design by writing a C++ program and using custom constraints or constraints drawn from a library.

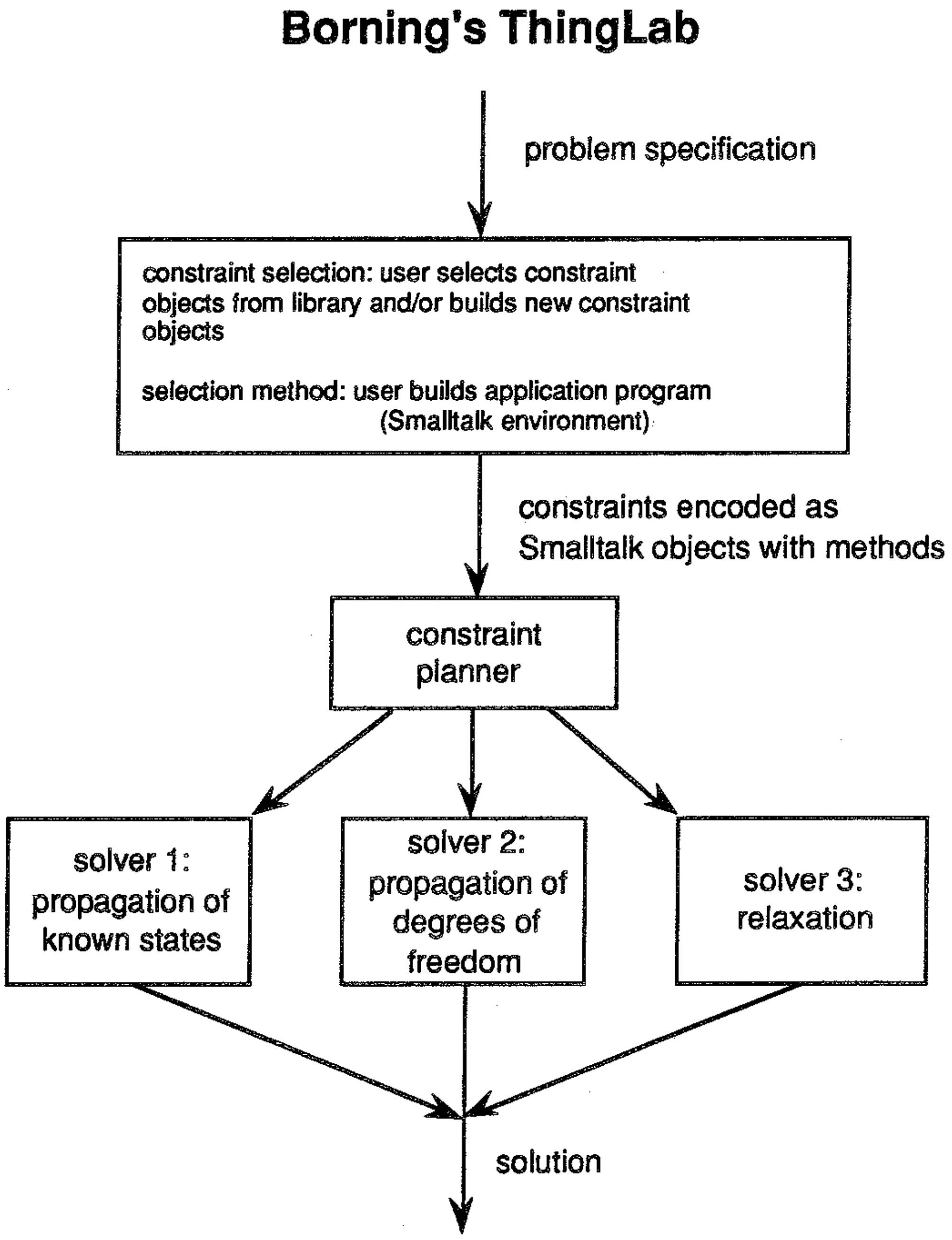

Borning's ThingLab
problem specification
constraint selection: user selects constraint objects from library and/or builds new constraint objects
selection method: user builds application program (Smalltalk environment)
constraints encoded as Smalltalk objects with methods
constraint planner
solver 1: propagation of known states
solver 2: propagation of degrees of freedom
solver 3: relaxation
solution

Figure 8.1: structure of ThingLab

Van Wyk's Ideal

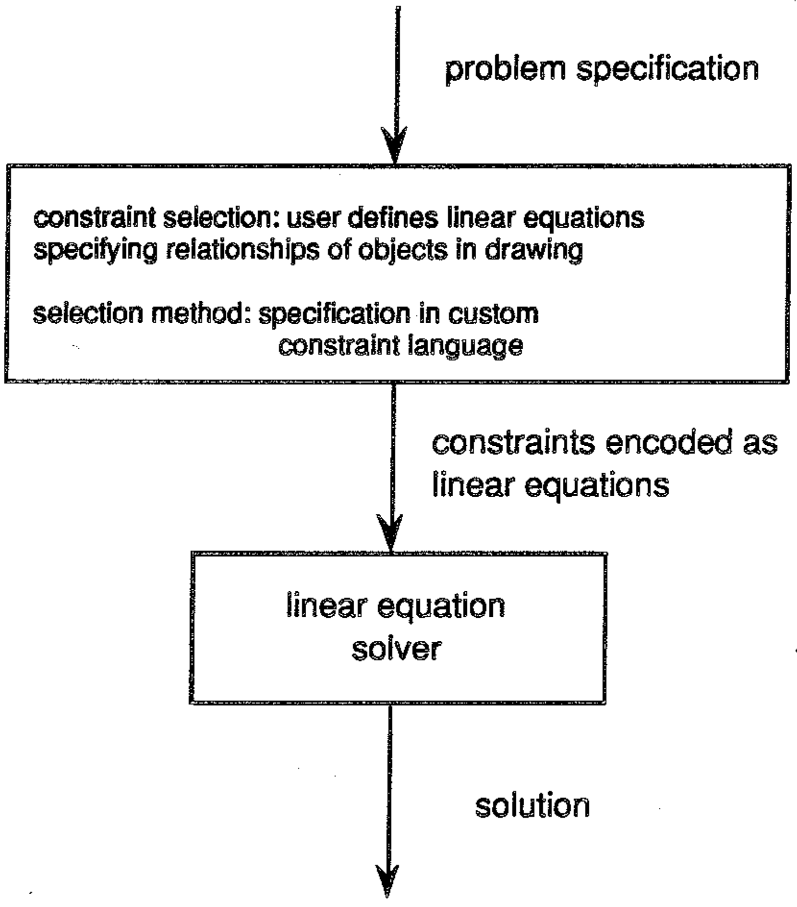

problem specification
constraint selection: user defines linear equations specifying relationships of objects in drawing
selection method: specification in custom constraint language
constraints encoded as linear equations
linear equation solver
solution

Figure 8.2: structure of Ideal

Brüderlin's System

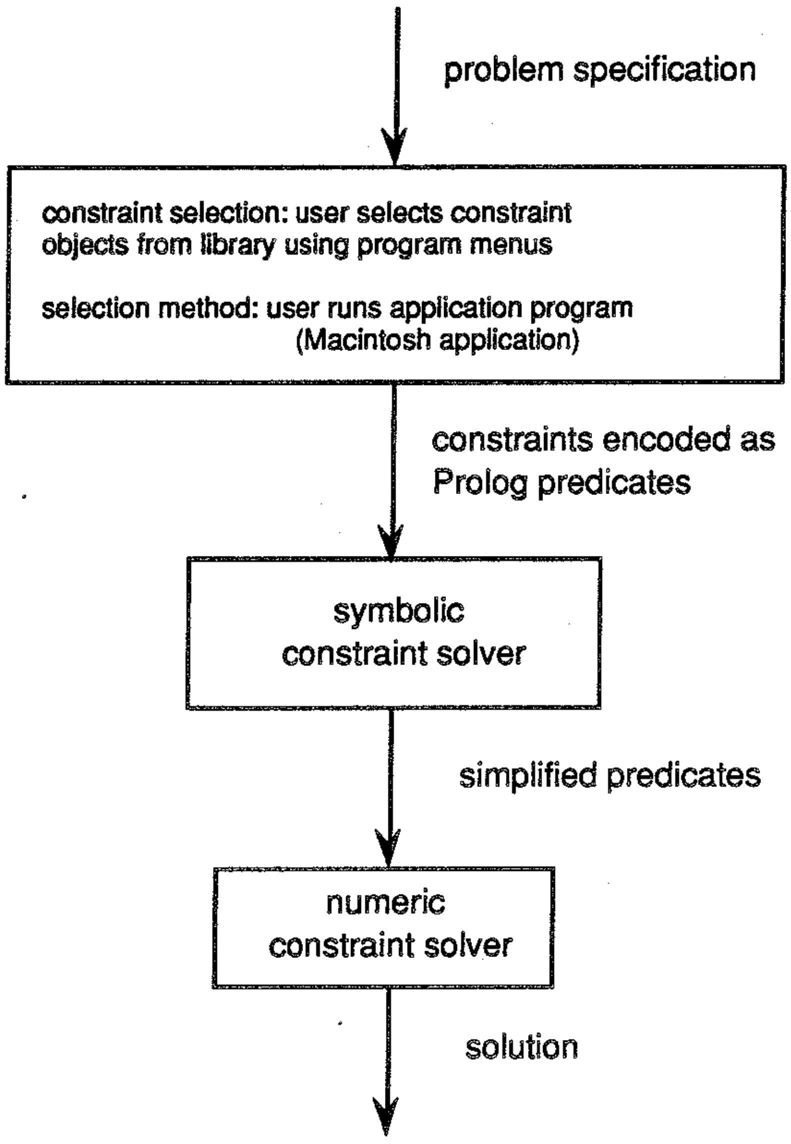

problem specification
constraint selection: user selects constraint objects from library using program menus
selection method: user runs application program (Macintosh application)
constraints encoded as Prolog predicates
symbolic constraint solver
simplified predicates
numeric constraint solver
solution

Figure 8.3: structure of Brüderlin's system

Brown's System

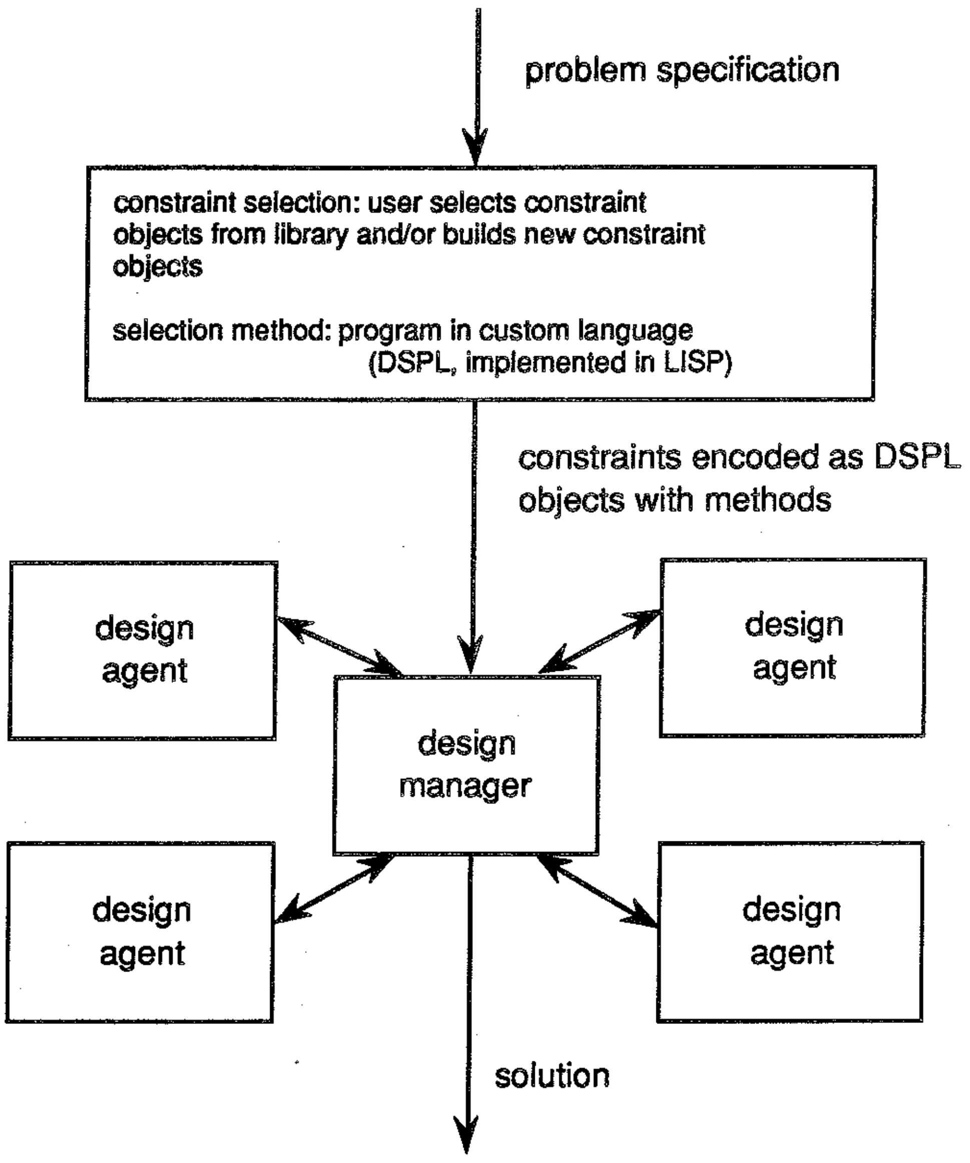

problem specification
constraint selection: user selects constraint objects from library and/or builds new constraint objects
selection method: program in custom language (DSPL, implemented in LISP)
constraints encoded as DSPL objects with methods
design agent
design agent
design manager
design agent
design agent
solution

Figure 8.4: structure of Brown's system

Barzel&Barr's System

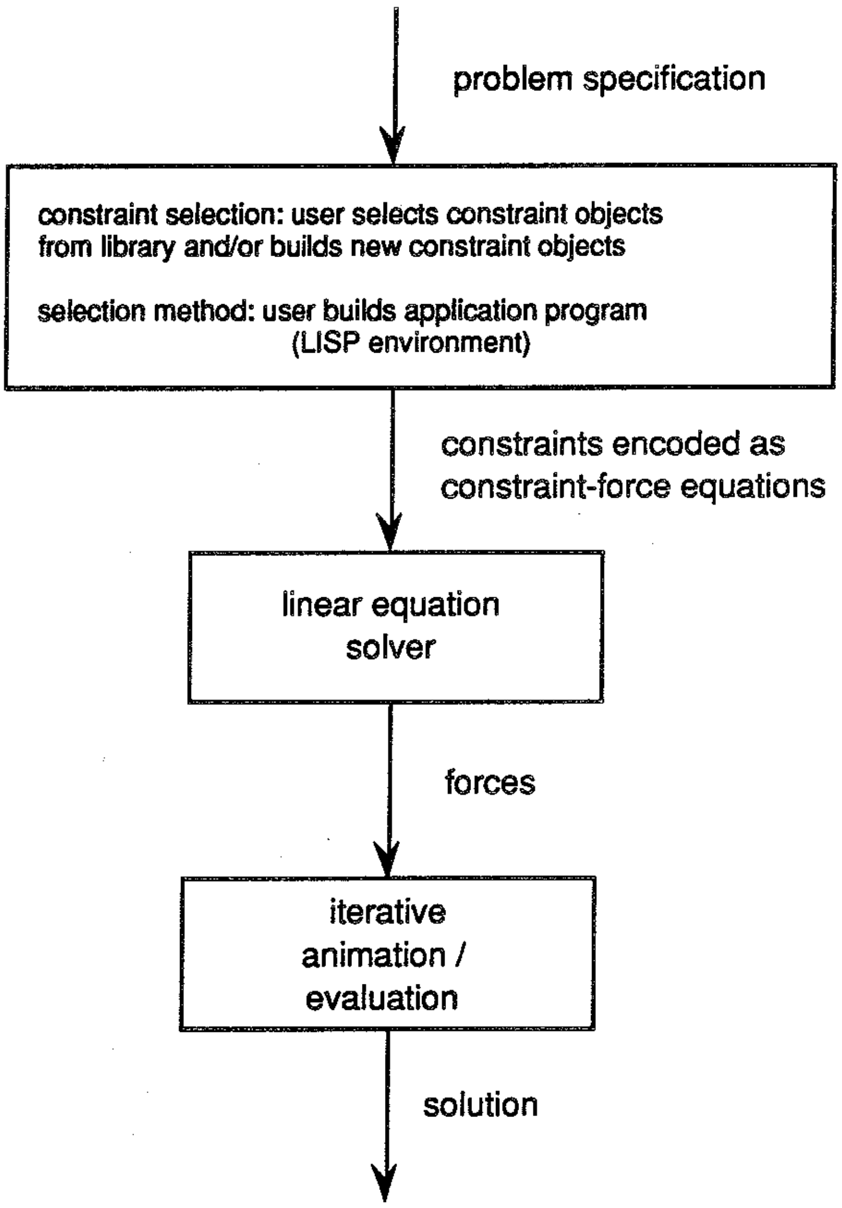

problem specification
constraint selection: user selects constraint objects from library and/or builds new constraint objects
selection method: user builds application program (LISP environment)
constraints encoded as constraint-force equations
linear equation solver
forces
iterative animation / evaluation
solution

Figure 8.5: structure of Barzel&Barr's system

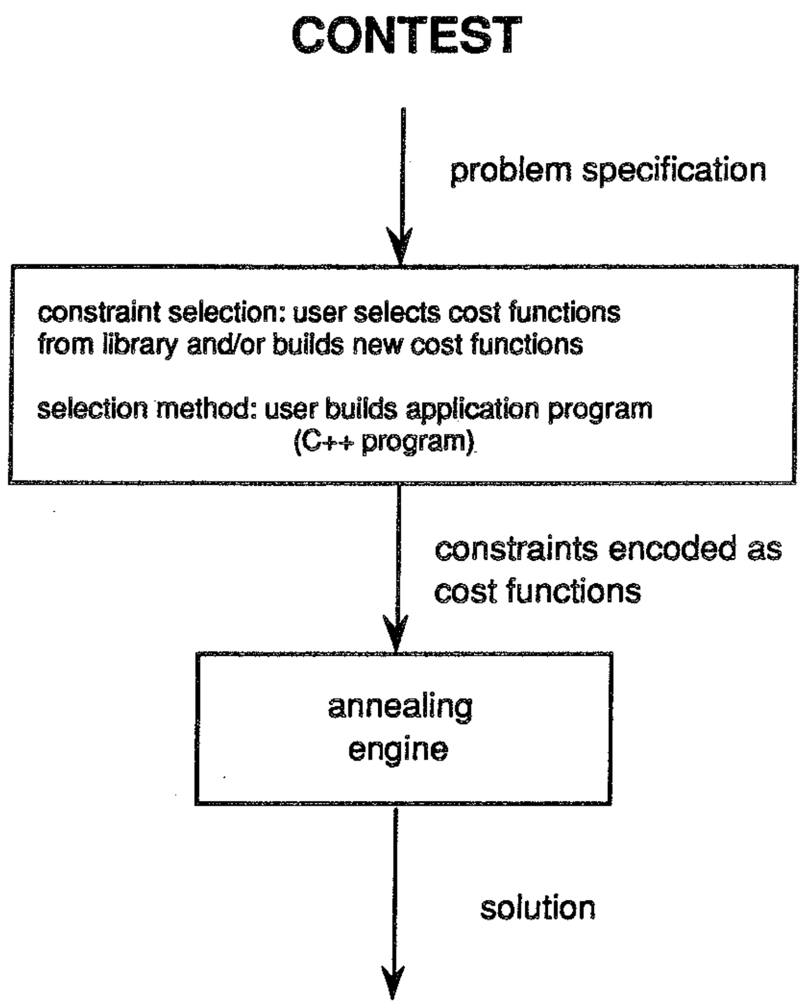

CONTEST
problem specification
constraint selection: user selects cost functions from library and/or builds new cost functions
selection method: user builds application program (C++ program)
constraints encoded as cost functions
annealing engine
solution

Figure 8.6: structure of CONTEST

8.2. Contributions

This section documents the contributions of this research. First, specific contributions are identified. Next, the advancement of the state-of-the-art by CONTEST is presented.

8.2.1. Specific contributions

This is not a dissertation about simulated annealing. Simulated annealing is used as a solution method and no new research in simulated annealing was necessary to apply the technique to constraint-based design.

Moreover, the contribution of this research is not the recognition that simulated annealing can be used to solve constraint problems. This was known prior to the start of this work.

The contribution of this research is the development of a methodology for converting general constraint-based design problems to scalar cost functions which can be optimized by simulated annealing. This dissertation has identified the various categories of constraints and described a method for converting these constraints into cost functions.

Three main contributions involve quantification of objective constraints. The first contribution is an identification of the most commonly used types of constraints. Most constraints can be expressed as a *minimize, maximize, less than, greater than, not equal to*, or *equal to* constraint. The second contribution was the derivation or identification of functions capable of representing the costs of these constraints. Several alternative function representations were presented, and three of these were implemented: the representations based on $f(x) = e^{-nx^b}$ and $f(x) = 1/(nx^b+1)$, as well as the arc tangent form for unbounded minimization and maximization. The third contribution was the development of a conversion from convenient user-specified *slope* and *value* parameters to the non-intuitive *n* and *b* parameters of these functions.

Moreover, the following categorization of constraints was identified as being useful for characterizing the cost function development procedure:

- objective constraints
- subjective constraints
- search constraints

The techniques developed for subjective and search constraints involve reducing and converting them to easily evaluated objective constraints. Thus the techniques for quantification of objective constraints also apply indirectly to all types of constraints.

The theory and development of these methods would be meaningless if they could not be used to solve real constraint problems. The validity of the theory was demonstrated by successful application of the methodology to two problems.

8.2.2. Advancement of the state-of-the-art

CONTEST differs from previous modeling systems by providing near-globally optimal results to complex, composite constraint problems, without requiring that someone supply problem solving heuristics.

The use of simulated annealing to solve design problems is not new; many VLSI systems use annealing to minimize chip area or wire length. The difference between previous systems and CONTEST is that in previous problems the objective function was well defined (e.g., minimize chip area). In CONTEST, the objective function is not well defined, and a primary contribution is the methodology for creating a composite cost function reflecting diverse constraints.

Among the systems described above, only the expert systems approach is capable of providing global constraint solutions to arbitrarily complex constraints. The difference between the expert systems approach and the cost function approach is that the expert systems approach simulates human problem solving and therefore requires an understanding of how to solve constraint problems. The cost function approach only requires that one be able to evaluate a constraint; the solution method is automatic.

CONTEST removes one or more of the following limitations on the power of previous modeling systems:

- limitations in constraint complexity (e.g., only linear or quadratic equations)
- limitations in constraint type (e.g., only geometric constraints)
- the need for human problem solving knowledge (e.g., expert systems)
- limitations in ability of the constraint solver to handle new constraints without changes to the constraint solver

8.3. Implementation

CONTEST is implemented in C++, and contains class definitions for both geometric objects and constraints. Constraint (cost function) objects request relevant information from geometric objects to evaluate constraints.

The annealing engine operates on two container classes: a geometry list and a constraint list. The geometry list is instructed to perturb or unperturb itself, while the constraint list is instructed to return the cost of the current geometry list.

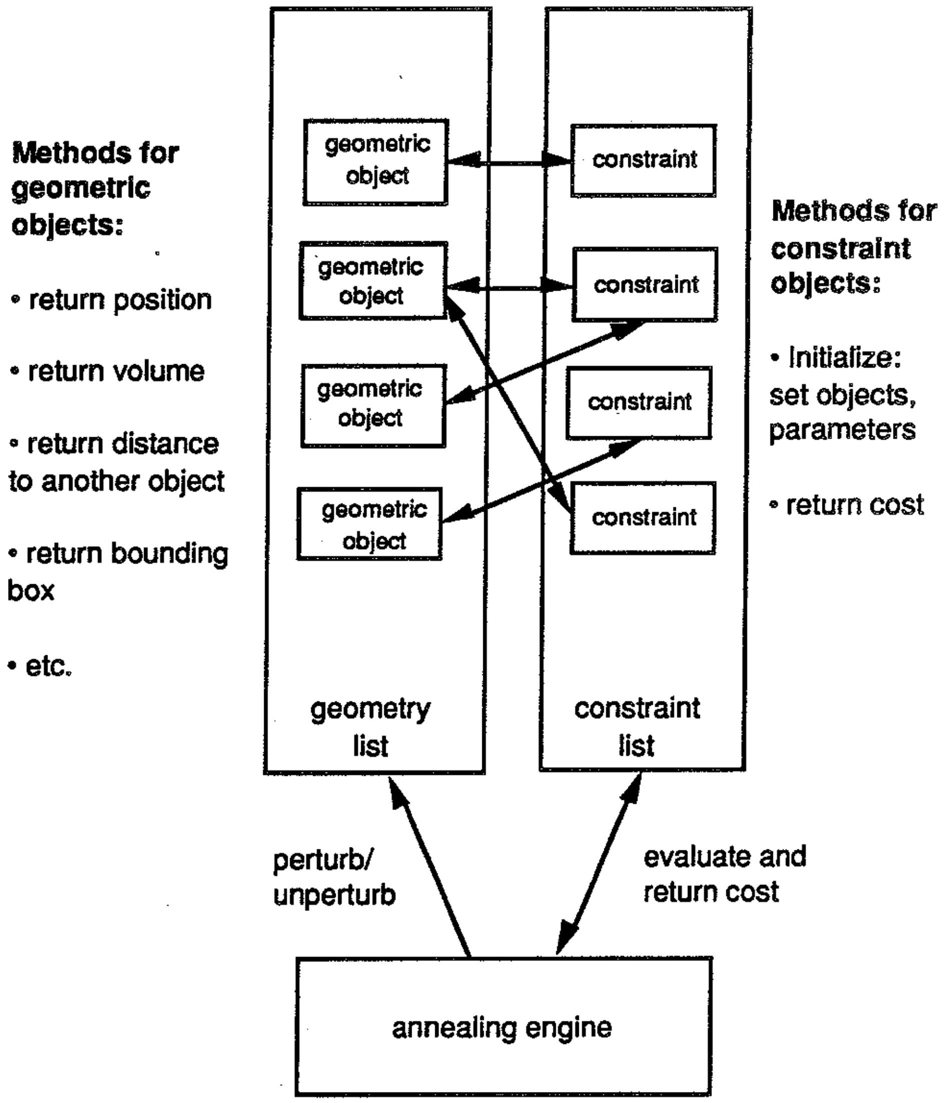


Figure 8.7: Implementation of CONTEST

The main program creates geometric objects and places them in the geometry list, and creates constraints and places them in the constraint list. It then passes the constraint list and geometry list to the annealing engine. After the annealing engine has completed, control returns to the main program. The main program then tells the geometry list to dump its geometric description. Additional information on the implementation is available in [Grant, 1986] and [Grant, 1987].

8.4. Summary

This chapter has compared CONTEST with several other systems that are representative of particular constraint satisfaction techniques: ThingLab, Ideal, and systems by Brüderlin, Barzel and Barr, and Brown. ThingLab was chosen to represent systems using multiple constraint solvers. Ideal was chosen to represent systems using equation solving. Brüderlin's system was chosen as an example of a system restricted by a fixed selection of constraints. Barzel and Barr's system was chosen to represent physically-based modeling systems. Brown's system was chosen as representative of expert systems. Each of these systems is limited in some way that CONTEST is not similarly limited. As with most advantages there are disadvantages: CONTEST takes much longer to find a solution than any of the described systems. Nevertheless, CONTEST represents an advancement of the state-of-the-art by providing capabilities not otherwise available.

This chapter has also identified the contribution of this research. The main contribution is a methodology for quantifying constraints. Specific techniques for quantifying objective constraints are combined with methods for converting subjective and search constraints to objective constraints to yield a generally applicable method for all constraints. The opaque projector and site planning problems demonstrated that the methodology may be successfully applied to real problems.

Finally the implementation of CONTEST was described. CONTEST is implemented in C++, and uses class definitions for geometric objects and constraints. The application of an object-oriented methodology has helped to manage the complexity of the software portion of the system.

Chapter 9

Conclusions / Future work / Summary

Previous chapters documented the development of a system for constraint-based modeling. The system was applied to two problems, and the results were presented. Finally, the system was compared with other modeling systems to identify the contributions of this research.

This final chapter presents some final conclusions about this work, discusses directions for future work, and summarizes the dissertation.

9.1. Conclusions

Design is an exploration process, even with automated constraint satisfaction tools. It is easy to think of the constraint-based modeling process as two simple steps: 1) the designer specifies the problem, and 2) the computer finds the solution. Unfortunately, it is not that simple. Very rarely does a designer begin the design process with a complete understanding of the problem to be solved. Design involves examining tradeoffs so as to better understand the problem. A more accurate model of design consists of four steps: 1) problem definition, 2) the computer finds a solution, 3) the designer evaluates the solution, 4) the designer modifies the problem and returns to step two.

It is important to recognize this design process in light of the run times required for the applications in chapters six and seven. A five hour turnaround for the design of a site plan is not extensive if only one design is required. However, if the designer needs to iterate toward a final design, he had better be very patient, as a one day task easily can expand to several weeks. I would recommend this solution method for applications that generate multiple designs and have a long lifetime after being created, but not for one-time design, unless a very precise problem specification is given beforehand.

One can reduce the time budget for a particular problem to get faster turnaround at the expense of the quality of the solution. While developing the opaque projector application, I was able to get reasonable results with only a ten minute run time. The results were non-optimal, but I could usually evaluate the results of the particular constraint or constraints I was trying to tune or implement. On the other hand, while developing the site planning project, there were times when I thought my cost functions were inaccurate when in fact the only problem was that I was not allocating enough time to yield near-optimal results. Once again, patience is required.

Despite the frustration of having to wait for these turnaround times, I actually consider these results quite encouraging. These were complex design problems that could not be quickly solved by a human. I did not have to use a supercomputer to get results. I look to the evolution of ray tracing for encouragement. Early implementations required hours per frame, but increased computer speed, parallelism, and algorithmic improvements have all made its use commonplace for rendering complex scenes. Cost-function-based modeling shares the same philosophy of power and generality at the expense of run time.

9.2. Future work

Since the computational expense of simulated annealing is a limiting factor in the applicability of this solution method, alternatives to the current implementation should be considered. Two possible approaches are: 1) maintain the problem representation but concentrate on reducing computation time, and 2) use a more efficient problem representation.

Methods for the first approach are more obvious. Many possible speedups exist for evaluating cost functions. Library calls to mathematical functions can be replaced by table lookup, since the cost function itself is already an approximation. Language support by C++ for inline functions can be used extensively to reduce the overhead of function calls. Computations performed during one iteration can be saved and reused whenever possible. Finally, much of the annealing procedure can be parallelized to take advantage of multiple workstations or a parallel machine architecture.

While most of these approaches are just implementation details, the parallelization of simulated annealing is a worthwhile research topic. Nevertheless, my research interests lie in the search for a more efficient problem representation rather than in methods of speeding up the current method.

The fundamental limitation of the current problem representation is that it leads to tremendously large search spaces. All possible configurations are considered as potential solutions, rather than only feasible solutions. For instance, the site planning application considers many site plans that would not be reasonable for any actual problem instance: plans with buildings intersecting one another, plans with trees growing through walls, and plans with buildings and trees in roadways. A more efficient approach is to search for the best configuration from the smaller set of legal configurations. One way to define this smaller set of legal configurations is by using shape grammars.

9.2.1. Shape grammars

Shape grammars can be used to describe legal designs. A simple shape grammar for campus layouts might have rules such as:

<campus layout> → *<dormitories> <academic buildings>*
or → *<dormitories> <academic buildings> <stores>*
<dormitories> → *<rectangular dormitory cluster> or <high rise dormitory>*
<rectangular dormitory cluster> → *<central courtyard design>*
or → *<packed building design>*

Eventually these reduce to specific geometric objects through a rule such as:

<dormitory row> → *<building> <building> <building> <building>*

Architects have used shape grammars to describe styles of design. Koning and Eizenberg, for example, defined a grammar to characterize the style of Frank Lloyd Wright's prairie houses [Koning, 1981]. Mitchell [Mitchell, 1990] provides examples of several simple grammars and the designs that can be enumerated by each grammar.

While the enumeration of all shapes is interesting, it is not particularly useful. The real power of the shape grammar approach will be realized by using constraints to select the best shape from these legal shapes. In other words, it is much more useful to select the best campus layout based on the terrain, local building codes, and the needs of the university, rather than simply generating millions of potential site plans. The following section discusses how this may be implemented.

9.2.2. Combining grammars and cost functions

The cost function methodology presented in this dissertation is not tied to simulated annealing. Cost functions may just as easily be used to evaluate shapes generated by a grammar.

If the grammar generates a small set of shapes, then the cost function can exhaustively test each shape and select the optimal design. Unfortunately, even simple grammars can still generate more shapes than may be feasibly tested. In this situation, one must search the set of potential solutions, just as simulated annealing searches the larger set of all solutions.

A grammar describes a tree structure, where internal nodes represent partially expanded designs, and leaf nodes represent possible solutions. The branches exiting a node correspond to the productions that may be applied at that node. Viewed as such, this reduces the problem to a tree search. The goal is to find the optimal solution without exploring too many paths. Initially, I plan to try standard techniques such as branch-and-bound to search the tree. If standard techniques are unsatisfactory, I will explore special-purpose search methods.

The combination of shape grammars and cost function evaluation is potentially very powerful. Merging the two techniques can lead to a system capable of producing results equal to (and perhaps better than) the results presented in this dissertation, but at a fraction of the computational expense.

9.3. Summary

Constraint-based modeling problems may be expressed as scalar cost functions. To find the optimal solution to a modeling problem, one minimizes the value of a composite cost function representing all constraints. This research has focused on the problem of encoding and combining diverse classes of constraints into a single cost function.

References

Aarts, Emile, and Jan Korst (1989), "Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing", Wiley, New York.

Aarts, E.H.L., and P.J.M. Van Laarhoven (1985), "A new polynomial time cooling schedule", In *Proceedings of IEEE International Conference on Computer-Aided Design* (pp. 206-208).

Ackley, David H. (1987), "Stochastic iterated genetic hillclimbing", Ph.D. Dissertation, Carnegie Mellon University.

Amburn, Elton P. (1991), "Using Symbolic Reasoning to Provide Progressive Truth in Geometric Modeling Systems", unpublished dissertation proposal draft, University of North Carolina at Chapel Hill.

Amburn, Phil, Eric Grant, and Turner Whitted (1986), "Managing Geometric Complexity with Enhanced Procedural Models", *Computer Graphics*, Vol. 20, No. 4, 189-195.

Barr, Alan H. (1986), "Dynamic Constraints for Modeling", In *State of the Art in Image Synthesis Tutorial Notes, SIGGRAPH '86*, Dallas, Texas, ACM SIGGRAPH.

Barzel, Ronen, and Alan H. Barr (1988), "A Modeling System Based on Dynamic Constraints", *Computer Graphics*, Vol. 22, No. 4, 179-188.

Borning, Alan (1979), "ThingLab: A Constraint-Oriented Simulation Laboratory", Ph.D. Dissertation, Stanford University.

Borning, A. (1985), "Defining Constraints Graphically" (Technical Report No. 85-09-06), University of Washington, Department of Computer Science, September, 1985.

Braid, I.C. (1985), "The Configurable Product Modeller", In *Proceedings of Eurographics '85* (pp. 143-144).

Brown, D.C. (1985), "Capturing Mechanical Design Knowledge" (Technical Report), The Ohio State University, Laboratory for Artificial Intelligence Research, August 1985.

Brown, David C., and B. Chandrasekaran (1986), "Knowledge and Control for a Mechanical Design Expert System", *Computer*, Vol. 19, No. 7, 92-100.

Brüderlin, Beat (1986), "Constructing Three-Dimensional Geometric Objects Defined By Constraints", In *1986 Workshop on Interactive 3D Graphics*, Chapel Hill, North Carolina, ACM SIGGRAPH.

Brüderlin, Beat D. (1988), "Rule-Based Geometric Modeling", Ph.D. Dissertation, Swiss Federal Institute of Technology (ETH) Zürich.

Cinar, U. (1975), "Facilities Planning: A Systems Analysis and Space Allocation Approach", In C.M. Eastman (Ed.), *Spatial Synthesis in Computer-Aided Building Design*, Wiley, New York.

Coyne, R.D., M.A. Rosenman, A.D. Radford, M. Balachandran, and J.S. Gero (1990), "Knowledge-Based Design Systems", Addison-Wesley, Reading, Mass.

Cross, Nigel (1977), "The Automated Architect", Pion, London.

Derman, E., and C.J. Van Wyk (1984), "A Simple Equation Solver and Its Application to Financial Modeling", *Software - Practice and Experience*, Vol. 14, No. 12, 1169-1181.

Duisberg, Robert A. (1986), "Constraint-Based Animation: Temporal Constraints in the Animus System", Ph.D. Dissertation, University of Washington.

Eastman, Charles M. (Ed.) (1975), "Spatial Synthesis in Computer-Aided Building Design", Wiley, New York.

Ervin, Stephen M. (1990), "Designing with Diagrams: A Role for Computing in Design Education and Exploration", In Malcolm McCullough, William J. Mitchell, and Patrick Purcell (Eds.), *The Electronic Design Studio: Architectural Knowledge and Media in the Computer Era*, MIT Press, Cambridge, Mass.

Flemming, Ulrich (1990), "Syntactic Structures in Architecture: Teaching Composition with Computer Assistance", In Malcolm McCullough, William J. Mitchell, and Patrick Purcell (Eds.), *The Electronic Design Studio: Architectural Knowledge and Media in the Computer Era*, MIT Press, Cambridge, Mass.

Franklin, Wm. Randolph, Peter Y.F. Wu, Sumitro Samaddar, and Margaret Nichols (1986), "Prolog and Geometry Projects", *IEEE Computer Graphics and Applications*, Vol. 6, No. 11, 46-55.

Geman, S., and D. Geman (1984), "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, 721-741.

Gero, John S. (Ed.) (1985), "Design Optimization", Academic Press, Orlando.

Gidas, B. (1985), "Nonstationary Markov Chains and Convergence of the Annealing Algorithm", *Journal of Statistical Physics*, Vol. 39, 73-131.

Gosling, James (1983), "Algebraic Constraints", Ph.D. Dissertation, Carnegie Mellon University.

Grant, Eric (1987), "Class Design for a Modeling Testbed", In *Object-Oriented Geometric Modeling and Rendering Tutorial Notes, SIGGRAPH '87*, Anaheim, ACM SIGGRAPH.

Grant, Eric, Phil Amburn, and Turner Whitted (1986), "Exploiting Classes in Modeling and Display Software", *IEEE Computer Graphics and Applications*, Vol. 6, No. 11, 13-20.

Gross, Mark, Stephen Ervin, James Anderson, and Aaron Fleisher (1988), "Constraints: Knowledge Representation in Design", *Design Studies*, Vol. 9, No. 3.

Gross, Mark D. (1985), "Design as the Exploration of Constraints", Ph.D. Dissertation, MIT Department of Architecture.

Gross, Mark D., Stephen M. Ervin, James Anderson, and Aaron Fleischer (1987), "Designing with Constraints", In Yehuda E. Kalay (Ed.), *Computability of Design*, Wiley, New York.

Kalay, Yehuda (1987), "Computability of Design", Wiley, New York.

Kirk, Stephen J. (1988), "Creative Design Decisions: A Systematic Approach to Problem Solving in Architecture", Van Nostrand Reinhold, New York.

Kirkpatrick, S., Jr. C.D. Gelatt, and M.P. Vecchi (1983), "Optimization by Simulated Annealing", *Science*, Vol. 220, No. 4598, 671-679.

Klir, George J., and Tina A. Folger (1988), "Fuzzy Sets, Uncertainty, and Information", Prentice Hall, Englewood Cliffs, N.J.

Koning H., and J. Eizenberg (1981), "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses", *Environment and Planning B*, Vol. 8, 295-323.

Kravitz, Saul A., and Rob A. Rutenbar (1986), "Multiprocessor-Based Placement by Simulated Annealing", In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* (pp. 567-573), IEEE Computer Society Press.

Laarhoven, P. J. M. van (1988), "Theoretical and Computational Aspects of Simulated Annealing", Centrum voor Wiskunde en Informatica, Amsterdam.

Laarhoven, P.J.M. van, and E.H.L. Aarts (1987), "Simulated Annealing: Theory and Applications", D. Reidel Publishing Company, Boston.

Lawson, Bryan (1983), "How Designers Think", Eastview Editions, Westfield, N.J.

Leler, William J. (1987), "Specification and Generation of Constraint Satisfaction Systems Using Augmented Term Rewriting", Ph.D. Dissertation, University of North Carolina at Chapel Hill.

Liggett, Robin S. (1985), "Optimal spatial arrangement as a quadratic assignment problem", In John S. Gero (Ed.), *Design Optimization*, Academic Press, Orlando.

Ligthart, Michiel M., Emile H.L. Aarts, and Frans P.M. Beenker (1986), "Design-for-testability of PLA's using Statistical Cooling", In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* (pp. 339-345), IEEE Computer Society Press.

Lin, V.C., D.C. Gossard, and R.A. Light (1981), "Variational Geometry in Computer-Aided Design", *Computer Graphics*, Vol. 15, No. 3, 171-177.

Marksjö, B. S. (1985), "Facility layout optimization using the Metropolis algorithm", *Environment and Planning B*, Vol. 12, 443-453.

MathLab (1983), "MACSYMA Reference Manual" (User Manual), MIT, Laboratory for Computer Science, January 1983.

McCullough, Malcolm, William J. Mitchell, and Patrick Purcell (Ed.) (1990), "The Electronic Design Studio: Architectural Knowledge and Media in the Computer Era", MIT Press, Cambridge, Mass.

McDermott, J. (1982), "R1 — A Rule-Based Configurer of Computer Systems", *Artificial Intelligence*, Vol. 19, No. 1, 39-88.

McGovern, I.E. (1976), "Non-Linear Optimization Theory Applied to Floor Plan Design", M.Sc. Thesis, UCLA, Graduate School of Management.

Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller (1953), "Equation of State Calculations by Fast Computing Machines", *Journal of Chemical Physics*, Vol. 21, 1087-1092.

Mitchell, William J. (1975), "Automated Generation of Minimum Energy Cost Building Designs", In B. Honikman (Ed.), *Responding to Social Change*, Halsted Press, New York.

Mitchell, William J. (1977), "Computer-Aided Architectural Design", Van Nostrand Reinhold, New York.

Mitchell, William J. (1990), "The Logic of Architecture: Design, Computation, and Cognition", MIT Press, Cambridge, Mass.

Mitchell, W.J., P. Steadman, and R.S. Liggett (1976), "Synthesis and Optimization of Small Rectangular Floor Plans", *Environment and Planning B*, Vol. 3, No. 1, 37-70.

Nahar, Surendra, Sartaj Sahni, and Eugene Shragowitz (1986), "Simulated Annealing and Combinatorial Optimization", In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* (pp. 293-299), IEEE Computer Society Press.

Nelson, Greg (1985), "Juno, a constraint-based graphics system", *Computer Graphics*, Vol. 19, No. 3, 235-243.

Osyczka, Andrzej (1984), "Multicriterion Optimization in Engineering (with FORTRAN programs)", Ellis Horwood Limited, Chichester, England.

Otten, R. H. J. M., and L.P.P.P. van Ginneken (1989), "The Annealing Algorithm", Kluwer Academic Publishers, Boston.

Peña, William (1987), "Problem Seeking: An Architectural Programming Primer" (3rd ed.), AIA Press, Washington.

Pfefferkorn, C.E. (1975), "The Design Problem Solver: A System for Designing Equipment or Furniture Layouts", In C.M. Eastman (Ed.), *Spatial Synthesis in Computer-Aided Building Design*, Wiley, New York.

Pincus, Jonathan D., and Alvin M. Despain (1986), "Delay Reduction Using Simulated Annealing", In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* (pp. 690-695), IEEE Computer Society Press.

Prusinkiewicz, Przemyslaw, and Dale Streibel (1986), "Constraint-Based Modeling of Three-Dimensional Shapes", In *Proceedings of Graphics Interface '86* (pp. 158-163).

Radford, Antony (1987), "CADD Made Easy: A Comprehensive Guide for Architects and Designers", McGraw-Hill, New York.

Radford, Antony D., and John S. Gero (1988), "Design by Optimization in Architecture, Building, and Construction", Van Nostrand Reinhold, New York.

Romeo, Fabio, and Alberto Sangiovanni-Vincentelli (1985), "Probabilistic Hill Climbing Algorithms: Properties and Applications", In Henry Fuchs (Ed.), *1985 Chapel Hill Conference on Very Large Scale Integration* (pp. 393-417), Computer Science Press, Rockville, Md.

Rossignac, Jaroslaw R. (1986), "Constraints in Constructive Solid Geometry", In *1986 Workshop on Interactive 3D Graphics* (pp. 93-110), Chapel Hill, North Carolina, ACM SIGGRAPH.

RTP Project Team (1987), "Research Triangle Park: Guidelines for Site Development" (Manual), North Carolina State University, School of Design, January 7, 1987.

Rychener, Michael D. (Ed.) (1988), "Expert Systems for Engineering Design", Academic Press, Boston.

Sanoff, Henry (1977), "Methods of Architectural Programming", Hutchinson & Ross, Stroudsburg, Pa.

Sechen, Carl, and Alberto Sangiovanni-Vincentelli (1986), "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package", In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* (pp. 432-439), IEEE Computer Society Press.

Seehof, J.M., and W.O. Evans (1967), "Automated Layout Design Program", *Journal of Industrial Engineering*, Vol. 18, No. 12, 690-695.

Shaviv, Edna (1987), "CAAD Tools for Spatial Allocation Problems", In Yehuda E. Kalay (Ed.), *Computability of Design*, Wiley, New York.

Siddall, James M. (1982), "Optimal Engineering Design: Principles and Applications", Marcel Dekker, New York.

Sriram, D., and R.A. Adley (Ed.) (1987), "Knowledge Based Expert Systems in Engineering: Planning and Design", Computational Mechanics Publications, Boston.

Stiny, George (1975), "Pictorial and Formal Aspects of Shape and Shape Grammars", Birkhauser, Basel.

Stiny, George (1990), "What Designers Do That Computers Should", In Malcolm McCullough,William J. Mitchell, and Patrick Purcell (Eds.), *The Electronic Design Studio: Architectural Knowledge and Media in the Computer Era*, MIT Press, Cambridge, Mass.

Stiny, George, and James Gips (1978), "Algorithmic Aesthetics: Computer Models for Criticism and Design in the Arts", University of California Press, Berkeley.

Stroustrup, Bjarne (1986), "The C++ Programming Language", Addison-Wesley, Reading, Mass.

Sutherland, Ivan E. (1963), "Sketchpad: A Man-Machine Graphical Interface System", Ph.D. Dissertation, Massachusetts Institute of Technology.

Thompson, George (1985), "Design Review: The Critical Analysis of the Design of Production Facilities", Mechanical Engineering Publications, London.

Van Wyk, C.J. (1980), "A Language for Typesetting Graphics", Ph.D. Dissertation, Stanford University.

Weinzapfel, G.E., and S. Handel (1975), "IMAGE: Computer Assistant for Architectural Design", In C.M. Eastman (Ed.), *Spatial Synthesis in Computer-Aided Design*, Wiley, New York.

Winston, Patrick Henry (1984), "Artificial Intelligence" (2nd ed.), Addison-Wesley, Reading, Mass.

Witkin, Andrew, Kurt Fleischer, and Alan Barr (1987), "Energy Constraints on Parameterized Models", *Computer Graphics*, Vol. 21, No. 4, 225-232.

Wolfram, Stephen (1988), "Mathematica: A System for Doing Mathematics by Computer", Addison-Wesley, Redwood City, Calif.

Wong, D.F., and C.L. Liu (1986), "A New Algorithm for Floorplan Design", In *Proceedings of the 23rd ACM/IEEE Design Automation Conference* (pp. 101-107), IEEE Computer Society Press.

Zeltzer, David (1984), "Representation and Control of Three Dimensional Computer Animated Figures", Ph.D. Dissertation, Ohio State University.