Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1997), *11*, 245–262. Printed in the USA.
Copyright © 1997 Cambridge University Press 0890-0604/97 $11.00 + .10

Spatial synthesis by disjunctive constraint satisfaction

CAN A. BAYKAN[1] AND MARK S. FOX[2]

[1]Department of Architecture, Middle East Technical University, 06531 Ankara, Turkey
[2]Department of Industrial Engineering, University of Toronto, 4 Taddle Creek Road, Toronto, Ontario M5S 1A4

(RECEIVED April 1, 1996; ACCEPTED November 18, 1996; REVISED January 15, 1997)

Abstract

The spatial synthesis problem addressed in this paper is the configuration of rectangles in 2D space, where the sides of the rectangles are parallel to an orthogonal coordinate system. Variables are the locations of the edges of the rectangles and their orientations. Algebraic constraints on these variables define a layout and constitute a constraint satisfaction problem. We give a new $O(n^2)$ algorithm for incremental path-consistency, which is applied after adding each algebraic constraint. Problem requirements are formulated as spatial relations between the rectangles, for example, adjacency, minimum distance, and nonoverlap. Spatial relations are expressed by Boolean combinations of the algebraic constraints; called disjunctive constraints. Solutions are generated by backtracking search, which selects a disjunctive constraint and instantiates its disjuncts. The selected disjuncts describe an equivalence class of configurations that is a significantly different solution. This method generates the set of significantly different solutions that satisfy all the requirements. The order of instantiating disjunctive constraints is critical for search efficiency. It is determined dynamically at each search state, using functions of heuristic measures called textures. Textures implement fail-first and prune-early strategies. Extensions to the model, that is, 3D configurations, configurations of nonrectangular shapes, constraint relaxation, optimization, and adding new rectangles during search are discussed.

Keywords: Spatial Layout; Geometric Reasoning; Disjunctive Constraint Satisfaction

1. INTRODUCTION

Spatial layout deals with the design of 2D layouts; site plans, floor plans, manufacturing facility layouts and the arrangement of equipment in rooms. Topological relations, such as adjacency, alignment, and grouping; geometric properties, such as shape, dimension, and distance; and other functions of spatial arrangement are the main concerns in spatial synthesis (Eastman, 1973). A *layout*, in this paper, is a 2D configuration of rectangles, where the sides of the rectangles are parallel to the axes of an orthogonal coordinate system. A layout problem is defined as follows: Given a set of rectangles, desired spatial relations between them, and limits on their dimensions, areas and aspect-ratios, generate the feasible alternatives. We describe a system called WRIGHT which solves this problem.

The efficiency apartment problem defined in Tables 1 and 2 is an example to the types of problems addressed: Find all configurations of an efficiency apartment consisting of living room, vestibule, kitchen and bathroom, satisfying the required adjacencies and the limits on their dimensions and areas. The apartment is adjacent to other apartments to the north and south, receives natural light from the west, and is accessed from the east. The objects in a layout, for example, the living room and vestibule, are called *design units*. The topological or geometrical relations between them, for example, adjacency, are called *spatial relations*. The requirements given in Tables 1 and 2 are called *performance* constraints. Performance constraints are based on function and may specify required and forbidden adjacencies, minimum and maximum distances between the design units and minimum and maximum lengths, widths, areas and aspect-ratios. There are also requirements implicit in a problem definition, for example, the interior spaces must be inside the apartment and should not overlap each other. These are called *realizability* constraints. The layout seen at left in Figure 1 shows a solution that satisfies the performance and realizability constraints of this problem.

There are other aspects of a configuration we may want to control, for example, whether holes are allowed, whether the corners of the design envelope must be filled, and which design units can be adjacent to the periphery or at the cor-

Reprint requests to: Can A. Baykan, Department of Architecture, Middle East Technical University, 06531 Ankara, Turkey. E-mail: cbaykan@vitruvius.arch.metu.edu.tr.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

Table 1. *List of spaces, minimum and maximum dimension and areas*

	Length (cm)		Width (cm)		Area (m^2)
List of spaces	Min	Max	Min	Max	Min
0. Apartment				700	
1. Living room	360		360		22
2. Vestibule	120	600	120	600	
3. Kitchen	180		180		4.2
4. Bathroom	180		180		

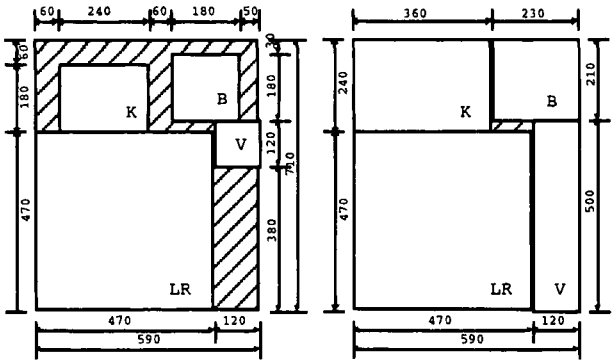


Fig. 1. Solutions to the efficiency apartment problem with and without trivial holes.

ners of the design envelope. These are called *style* constraints. Adding a constraint that every interior space must be adjacent to another space or to a side of the apartment on all four sides eliminates *trivial holes*, resulting in the layout shown at right in Figure 1. The hole that remains is a *non-trivial hole*. In WRIGHT, all aspects of a layout problem are specified uniformly and explicitly by constraints, and the spatial relations required to express the relationships that are of interest in some domain can be created declaratively.

The topology and geometry of a layout is specified by algebraic equations and inequalities, called *atomic constraints*, which determine the locations, dimensions and orientations of design units. The variables and atomic constraints constitute a constraint satisfaction problem (CSP). The performance, realizability and style constraints are expressed in terms of Boolean combinations of atomic constraints, termed disjunctive constraints, and define a *disjunctive CSP*. Solutions are constructed by backtracking search, which instantiates a disjunctive constraint by asserting one of its disjuncts in each new state. A disjunctive constraint is selected for instantiation using a function of *textures* that are measures of the topological and other features of the constraints (Fox et al., 1989). Textures reduce search by identifying disjunctive constraints that are more likely to fail and to cause other constraints to fail.

Section 3 describes how a layout is represented by atomic constraints and gives a new incremental path-consistency algorithm used when adding a new constraint during search. Sections 4 and 5 describe spatial relations and how they represent problem requirements in terms of atomic constraints. Section 6 describes the search method and constraint selection by textures. Section 7 discusses whether this formulation can express knowledge of a design domain and some extensions to the basic model.

Table 2. *Required adjacencies (minimum length of shared boundary in brackets)*

Living room—Vestibule (90)	Living room—West edge of Apartment
Living room—Kitchen (120)	Living room—South edge of Apartment
Vestibule—Bathroom (70)	Vestibule—East edge of Apartment

2. BACKGROUND

Two-dimensional configuration problems have been formulated in various ways for different purposes. There are packing or loading problems, where the configuration is of no interest other than fitting the most objects in a given area. Perfect square and incomparable rectangle problems involve placing objects as well as determining their dimensions (Aggoun & Beldiceanu, 1993). Another formulation is the quadratic assignment problem (Liggett, 1980). These are discrete problems, that is, the dimensions, when they are not fixed, and locations are selected from a discrete domain. Spatial layout deals with the geometric and topological relationships of the layout that requires dealing with continuous ranges of dimensions and locations.

The early spatial layout systems, DPS (Pfefferkorn, 1971) and GSP (Eastman, 1973), could not deal with variable sizes and were not complete; they could miss possible solutions. Systematic approaches formulated to deal with these drawbacks used a relational representation, that is, rectangular dissections (Mitchell et al., 1976) and adjacency graphs (Baybars & Eastman, 1980). DIS (Flemming, 1978) uses an algorithm that systematically generates all distinct configurations. The generation process has two steps: determining the relational structure of a layout using north-of and east-of relations; and deriving the constraints on the dimensions based on this topology. DIS and other layout programs that use a relational representation have built-in constraints that restrict the set of layouts under consideration. In DIS, the built-in constraints are that design units do not overlap and are tightly packed. LOOS (Flemming, 1986) removes the tight-packing restriction to allow layouts with holes. Relational approaches determine the *significantly different alternatives* based on their underlying representation. The disjunctive CSP formulation defines both relational and dimensional constraints at the outset and represents them uniformly. It is possible to use all constraints to reduce search. Each significantly different solution is formed by the intersection of one disjunct from every disjunctive constraint. A comparison of the approaches of LOOS and WRIGHT can be found in Flemming et al. (1992).

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

A CSP is a network that has variables with finite domains as its nodes and constraints as its edges. A constraint is a relation between some subset of the variables that identifies compatible values of the variables. The domains of variables can be discrete or interval. If the domains are discrete, constraints can be expressed as compatibility constraints, by a list of ordered pairs. They can also be expressed as procedures, mathematical relations, or disjunctive constraints. Solving a CSP involves finding an assignment to all variables satisfying all constraints. Solutions can be found by generate and test, that is, by selecting a variable and assigning a value at each step and backtracking when dead-ends are reached; by using consistency methods, for example, node, arc or path-consistency; or by some combination of the two. Consistency methods, also termed constraint propagation, remove from the domains of variables those values that do not have corresponding values in the domains of other variables (Mackworth, 1977; Mackworth & Freuder, 1985). Arc-consistency ensures that a value in the domain of a variable has a match in the domain of any other variable that is connected to the first by a constraint. Waltz propagation (Waltz, 1975) is an arc-consistency algorithm. Path-consistency algorithms impose local consistency among triplets of variables, that is, paths of length two. Path-consistency is stronger than arc-consistency and achieves global consistency, that is, every value in the domain of a variable can be part of a complete solution to the CSP. When composition of constraints is distributive over intersection, path consistency algorithm PC-1 given in Mackworth (1977) becomes equivalent to the Floyd–Warshall all-pairs, shortest-path algorithm (Dechter et al., 1991). Otherwise, PC-1 repeats a loop that is equivalent to the Floyd–Warshall algorithm until no entry is updated in one complete pass.

For bounded distance and orientation constraints used in WRIGHT composition is distributive over intersection, and we give a new incremental form of the Floyd–Warshall algorithm for these. We use Waltz propagation for area and aspect-ratio constraints. The earlier version of WRIGHT reported in Baykan and Fox (1992) and Baykan (1991) also used the disjunctive CSP formulation but used a combination of arc consistency and qualitative reasoning instead of path consistency when updating layouts. All variables including dimensions, aspect-ratios, areas, and atomic constraints formed by $+$, $\times$, $<$, $\leq$, and $=$ were represented explicitly. This formulation is more flexible as it makes it possible to add new variables and new types of constraints on them, that is, the center lines of design units as variables and the centered-on relation and Euclidean distance relation as constraints. Waltz propagation results in looser bounds, and an inconsistent constraint cannot always be detected prior to asserting it even when it is augmented by qualitative reasoning. It results in more search. The two versions are compared in Section 6.4.

The disjunctive CSP formulation has been applied to a variety of problems. In circuit analysis, it arises as a result of modelling the behavior of nonlinear devices, such as transistors, by disjuncts of linear constraints (Stallman & Sussman, 1977). In qualitative physics, a system that has several states, such as a quantity of material having solid, liquid, and gas phases where each phase has different rules, has been formulated as a disjunctive CSP. Different alternatives are given by the combinations of rules, and each alternative forms a CSP (Forbus, 1984). Scheduling problems can also be formulated as disjunctive CSPs (Dechter et al., 1991).

In algebra, a disjunctive CSP is called a Boolean satisfiability problem. It is NP-complete even if all equations are of the form $x = 0$ or $x = 1$ unless the combinations are very restricted in form (Davis, 1987). Adding an objective function to minimize or maximize some combination of variables transforms a satisfiability problem into a mixed integer programming (MIP) problem. MIP is also NP-complete. The MIP solution methods use heuristics that make assumptions on problem structure. If the method works it works fast, but when it does not work, it may not terminate in a reasonable time. And it finds a single optimal solution which does not enable the designer to evaluate the possibilities and tradeoffs implicit in a problem formulation (Dhar & Ranganathan, 1990; Smith et al., 1996).

3. REPRESENTING LAYOUTS

WRIGHT represents a layout as an interval CSP. The CSP can be elaborated by adding new constraints, and a unique layout can be created by selecting a particular value from the domain of each variable. Each CSP defines an equivalence class of layouts, considered to be a significantly different layout.

3.1. Variables

In WRIGHT, design units are rectangles parallel to the x and y axes. The coordinate system used has the x-axis pointing right and the y-axis pointing down. Each design unit is defined by four variables specifying the locations of its edges; x, X, y and Y, as shown in Figure 2. There are also variables for the area, aspect-ratio and orientation of each design unit.

The design units and variables in the efficiency apartment problem are seen in Table 3. The domains of the variables are closed intervals defined by lower and upper bounds; $v \in [v_{min}, v_{max}]$. The lower bound of a variable must be less than or equal to its upper bound; $v_{min} \leq v \leq v_{max}$. When the

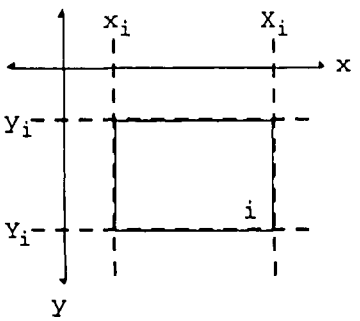


Fig. 2. Variables defining design unit i.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

Table 3. *Design units and variables of the efficiency apartment problem*

Design unit	North-edge	South-edge	West-edge	East-edge	Area
0 Apartment	y_0	Y_0	x_0	X_0	
1 Living room	y_1	Y_1	x_1	X_1	A_1
2 Vestibule	y_2	Y_2	x_2	X_2	
3 Kitchen	y_3	Y_3	x_3	X_3	A_3
4 Bath	y_4	Y_4	x_4	X_4	

lower bound is equal to the upper bound, the interval becomes an exact value. Thus, exact values are a special case of closed intervals and can be handled uniformly. In WRIGHT, the bounds are integers. Since the user is free to define the unit of length, using integers does not impose undue restrictions. For example, it is possible to have more precision by using millimeters as the unit of length instead of centimeters in the efficiency apartment problem.

Area and aspect-ratio are also interval variables. Area bounds are integer but aspect-ratio bounds are real. Orientation is a discrete variable, and its domain is {0,90,180,270}. When its orientation is 0, front of a design unit faces south, and when its orientation is 90, it faces east. Orientation is meaningful when a design unit has sides that must be treated differently, for example, a refrigerator has a front and back that must be taken into account when specifying relations between it and other design units.

3.2. Atomic constraints

An atomic constraint defines a relation between two variables. Types of atomic constraints used in WRIGHT are bounded difference constraint, area constraint, aspect-ratio constraint, and orientation constraint. A bounded difference constraint specifies minimum and maximum distances between two lines in the same direction, $x_i - x_j \in [d_{\min}, d_{\max}]$. The kinds of partial configurations that can be represented using this constraint are given below.

The dimensions of a design unit, the length of adjacency between two design units, and the horizontal or vertical distance between two design units can be represented as the difference of two lines as shown in Figure 3. Length of design unit 1, $d_1 \in [d_{1\min}, d_{1\max}]$ is expressed by $X_1 - x_1 \in [d_{1\min}, d_{1\max}]$ as seen at left. The length of overlap between design units 2 and 3 is expressed by $X_2 - x_3 \in [d_{2\min}, d_{2\max}]$ as seen in the middle, and the distance between design units 4 and 5 is represented as $x_5 - X_4 \in [d_{3\min}, d_{3\max}]$ as seen at right.

Topology and alignment can be expressed by $=$, $<$, and $\leq$ relations between the edges of rectangles. These algebraic relations can be expressed as differences of $[0,0]$, $[1,\infty]$, and $[0,\infty]$, respectively. For example, design units 1 and 2 can be made adjacent, as seen at left in Figure 4, by asserting the constraint $X_1 - x_2 \in [0,0]$. Design unit 3 is placed west of 4, as seen in the middle, by posting the constraint $x_4 - X_3 \in [0,\infty]$. East edges of design units 5 and 6 can be aligned, as shown at right, by asserting $X_5 - X_6 \in [0,0]$. Absolute location of a line is represented as the distance between it and a special line $x_0 \in [0,0]$ or $y_0 \in [0,0]$. Thus, $x_1 \in [100,150]$ is denoted as $x_1 - x_0 \in [100,150]$. The special lines x_0 and y_0 are the west and north edges of the design envelope, which is design unit number 0 by the convention used in WRIGHT.

Area constraints specify minimum and maximum areas for a design unit. The area constraints of the efficiency apartment problem are $A_1 \in [220000,\infty]$ and $A_3 \in [42000,\infty]$. Similarly, aspect-ratio constraints specify bounds on the ratio of the long side of a rectangle to its short side. For ex-

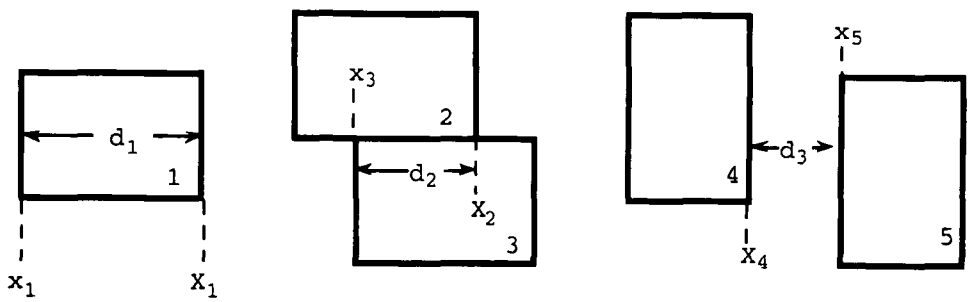


Fig. 3. Representing dimensions, length of adjacency, and distance by bounded distances.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

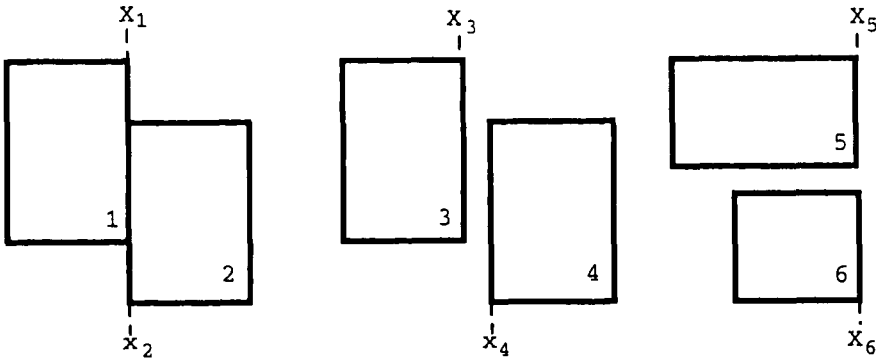


Fig. 4. Representing topological relations; adjacency, relative location, and alignment.

ample, the constraint $AS_1 \leq 1.5$ specifies that a side of the living room can not be longer than 1.5 times the other side. Orientation constraints specify the relative orientations of two design units, such as parallel, perpendicular, clockwise, counterclockwise, or opposite; the constraint $O_1{:}O_2 \in \{90, 270\}$ asserts that the living room should be perpendicular to the vestibule. There are no aspect-ratio and orientation constraints in the efficiency apartment problem, so these variables are not created.

Atomic constraints that are valid for all solutions can be asserted in the initial state, according to the singleton heuristic, and form the partial specification of all layouts generated from this state. These bounded difference constraints are given in Table 4. Constraints 1–5 define limits on the x-dimensions and constraints 6–9 limits on the y-dimensions. Constraint 10 places the living room adjacent to the west, and 11 makes it adjacent to the south. Constraint 12 places the vestibule adjacent to the east. Constraints 13–28 place all design units inside the apartment. Constraints 13, 15, and 18 are redundant given 10, 11, and 12. Forward-checking detects inconsistency and redundancy between the constraints that are asserted and the constraints that are yet to be satisfied. Forward-checking and the singleton heuristic are described in Section 6.1.

The solution at left in Figure 2 is defined by adding atomic constraints 1–5 in Table 5 to the constraints of the initial state. Adding constraint 6–11 eliminates the trivial holes and defines the solution at right. The layout defined by these constraints can be elaborated further by adding the constraint $x_2 - X_3 \in [0,0]$, which eliminates the nontrivial hole by making the kitchen and the vestibule adjacent. On the other hand, asserting $x_2 - X_3 \in [1,\infty]$ makes the east edge of the kitchen less than the west edge of the vestibule and makes the hole impossible to remove.

3.3. Maintaining consistency

The variables and atomic constraints constitute a CSP, which is formed incrementally by adding constraints to elaborate the layout. Any change in the domain of a variable due to asserting a new constraint is propagated to other variables via the existing constraints; thus, a constraint asserted in some state defines a relation that is maintained in all layouts derived from it.

The constraints and variables form groups. Orientation variables and constraints form a network by themselves. The edges in the x direction and the bounded difference constraints on them are separate from the y edges and the

Table 4. *Bounded difference constraints defining the initial state of efficiency apartment problem*

1. $X_0 - x_0 \in [0,700]$	2. $X_1 - x_1 \in [360,\infty]$	3. $X_2 - x_2 \in [120, 600]$
4. $X_3 - x_3 \in [180,\infty]$	5. $X_4 - x_4 \in [180,\infty]$	6. $Y_1 - y_1 \in [360,\infty]$
7. $Y_2 - y_2 \in [120,600]$	8. $Y_3 - y_3 \in [180,\infty]$	9. $Y_4 - y_4 \in [180,\infty]$
10. $x_1 - x_0 \in [0,0]$	11. $y_1 - y_0 \in [0,0]$	12. $X_0 - X_2 \in [0,0]$
13. $x_1 - x_0 \in [0,\infty]$	14. $X_0 - X_1 \in [0,\infty]$	15. $y_1 - y_0 \in [0,\infty]$
16. $Y_0 - Y_1 \in [0,\infty]$	17. $x_2 - x_0 \in [0,\infty]$	18. $X_0 - X_2 \in [0,\infty]$
19. $y_2 - y_0 \in [0,\infty]$	20. $Y_0 - Y_2 \in [0,\infty]$	21. $x_3 - x_0 \in [0,\infty]$
22. $X_0 - X_3 \in [0,\infty]$	23. $y_3 - y_0 \in [0,\infty]$	24. $Y_0 - Y_3 \in [0,\infty]$
25. $x_4 - x_0 \in [0,\infty]$	26. $X_0 - X_4 \in [0,\infty]$	27. $y_4 - y_0 \in [0,\infty]$
	28. $Y_0 - Y_4 \in [0,\infty]$	

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

Table 5. *Bounded difference constraints defining a solution to the efficiency apartment problem*

1. $Y_3 - y_1 \in [0,0]$	2. $Y_4 - y_1 \in [0,\infty]$	3. $X_1 - x_2 \in [0,0]$
4. $Y_4 - y_2 \in [0,0]$	5. $x_2 - X_3 \in [0,\infty]$	
6. $y_4 - y_0 \in [0,0]$	7. $y_3 - y_0 \in [0,0]$	8. $Y_2 - Y_0 \in [0,0]$
9. $x_3 - x_0 \in [0,0]$	10. $X_4 - X_0 \in [0,0]$	11. $X_3 - x_4 \in [0,0]$

bounded difference constraints on them, but area and aspect-ratio constraints connect the x and y variables. Consistency of bounded difference constraints and orientations are maintained by a path-consistency algorithm, and consistency of area and aspect-ratio constraints are maintained by Waltz propagation (Waltz, 1975).

The network of variables in the x or y direction and the bounded difference constraints on them can be represented by a complete directed weighted graph, called the distance graph where each edge from α to β is labeled by d, length of the shortest path from α to β. A bounded difference constraint, $x_i - x_j \in [d_{\min}, d_{\max}]$, is represented in canonical form as a pair of inequalities of the form $\alpha - \beta \leq d$, because $x_i - x_j \in [d_{\min}, d_{\max}] \rightarrow x_i - x_j \leq d_{\max} \wedge x_j - x_i \leq -d_{\min}$. A given layout can be completely specified by two such graphs, one for each direction. Each distance graph has n nodes, where n is the number of horizontal or vertical lines, that is, two lines per design unit. Each dimension by itself is similar to a temporal constraint network, and more on this formulation can be found in Dechter et al. (1991). But, instead of the Floyd–Warshall all-pairs, shortest-path algorithm given there to achieve path-consistency in the distance graph, we give two incremental algorithms, IPC-1 and IPC-2. These are sufficient for updating layouts during search because a few constraints are added in each state. The Floyd–Warshall algorithm has time complexity of $O(n^3)$, whereas IPC-1 and IPC-2 given below have time complexity of $O(n^2)$.

IPC-1, seen in Figure 5, takes one inequality of the form $\alpha - \beta \leq d$ as input and updates all paths of length 3 that contain the changed path in the middle. The first distance in the path is from every one of the n lines to α; the second is the changed distance, d, between $[\alpha, \beta]$, and the third is from β to all other lines, as shown in Figure 6. When length of path $[i, \alpha] + d + [\beta, j]$ is less than the current distance between $[i, j]$, $[i, j]$ is updated. When $i = \alpha$, all paths of length two that contain the new constraint as its first edge are updated. When $j = \beta$, length two paths that contain the new constraint as its second edge are updated. When both $i = \alpha$ and $j = \beta$, the new constraint itself is posted. There are n possible first and last nodes, thus n^2 paths to consider; resulting in time complexity $O(n^2)$. The order in which the distances are updated is the same as it is in the Floyd–Warshall algorithm, and this order is necessary for the algorithm to work correctly. Under these conditions, IPC-1 achieves the same result as the Floyd–Warshall algorithm.

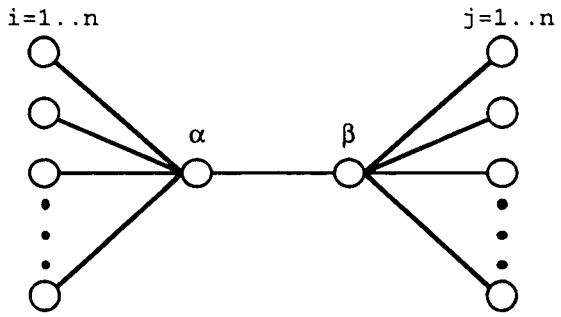


Fig. 6. Paths of length 3 considered by IPC-1 and IPC-2.

The CSP should be in a consistent state with respect to the existing constraints before calling IPC-1. This allows checking a constraint $\alpha - \beta \leq d$ before asserting it. If d is greater than or equal to the current value of the edge $[\alpha, \beta]$ then it is redundant, and if $-d$ is greater than $[\beta, \alpha]$ then it is inconsistent. These checks can be done in constant time before calling IPC-1, therefore the consistency check in the innermost loop of the Floyd–Warshall algorithm is not needed in IPC-1.

IPC-2, given in Figure 7, tries only a portion of the n^2 distances that are candidates for being updated, by first checking paths of length 2, which are contained in the paths of length 3 that have $[\alpha, \beta]$ in the middle. The length of path $[i, \alpha] + d + [\beta, j]$ can be less than the current maximum distance between $[i, j]$, if and only if $([i, \alpha] + d < [i, \beta])$ and $(d + [\beta, j] < [\alpha, j])$. In the first case, paths of length 3 starting from i are considered. In the second case, paths of length 3 ending at j are considered. Otherwise, these paths are eliminated from consideration. On problems we have solved, IPC-2 reduced propagation steps by 66–75% compared to IPC-1. Time complexity of IPC-2 is also $O(n^2)$.

The consistency of orientation constraints is also maintained by IPC-2. The domain of an orientation variable is represented by a bit-vector of length 4. Each position in the vector represents a possible orientation. An orientation re-

```
Procedure IPC-1 (α, β, d)
  for i = 1 to n do
    for j = 1 to n do
      [i, j] ← min ([i, j], [i, α] + d + [β, j])
```

Fig. 5. Incremental path-consistency algorithm IPC-1.

```
Procedure IPC-2 (α, β, d)
  for i = 1 to n do
    if [i, α] + d < [i, β] then add i to end of Q1
  for j = 1 to n do
    if d + [β, j] < [α, j] then add j to end of Q2
  for i in Q1 do
    for j in Q2 do
      [i, j] ← min ([i, j], [i, α] + d + [β, j])
```

Fig. 7. Incremental path-consistency algorithm IPC-2.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

lation is represented similarly; each position in the vector representing whether the clockwise difference in orientation between two design units is allowed. For orientations, composition of constraints adds two orientation differences and intersection of constraints is the intersection of their bit vectors. Composition is distributive over intersection.

Area and aspect-ratio constraints are handled by a different algorithm as they involve propagation of constraints between the x and y matrices. When there is an area or aspect-ratio constraint on a design unit, any change to one of its dimensions is propagated to its other dimension by Waltz propagation (Waltz, 1975). The Waltz algorithm is an arc-consistency algorithm, and it does not achieve global consistency when there are loops in the constraint graph even though it always detects inconsistencies (Davis, 1987).

4. SPATIAL RELATIONS

It is natural to define layout problems using spatial relations to indicate the location of one design unit with respect to another. Some spatial relations are purely topological, such as adjacency, inside and nonoverlap. Some, for example, distance, involve a dimension, and some are dependent on the orientations of the design units.

The x or y component of a rectangle is an interval. If we take the x component, its endpoints are the west and the east edges. There are 13 exhaustive and mutually exclusive relations between two intervals; Allen's qualitative temporal relations between two time intervals (Allen, 1983). These can be expressed as algebraic relations between the endpoints of the two intervals as shown in Malik and Binford (1983). Relations between the x and y components of rectangles are independent, therefore there can be $13 \times 13 = 169$ mutually exclusive and exhaustive relations between two rectangles. A design unit has 4 possible orientations; so there can be 16 orientation combinations between two design units. Considering the orientation of one of the design units results in $4 \times 169 = 676$ possible relations, and considering orientations of both design units results in $16 \times 169 = 2704$ relations. There are two shortcomings with defining all possible qualitative spatial relations at the outset. There are too many relations that are too specific. They do not correspond to the types of relations we want to express. For example, 192 of the 2704 relations are types of adjacency. The set of spatial relations should make it easy to express the desired features in a domain of spatial layout. This set is not fixed; it depends on the design domain. If we need to express other relationships, it should be possible to create new spatial relations easily. In WRIGHT, all spatial relations are defined declaratively by templates. The constraint compiler, described below, makes it easy to define new spatial relations and use them in disjunctive constraints.

The spatial relations defined in WRIGHT are seen in Figure 8. These are the relations that were needed to solve the problems we tried. They are grouped in two: *global* relations do not take orientations into consideration at all; and *object-centered* relations are defined with respect to the orientation of the first design unit. Configurations illustrating

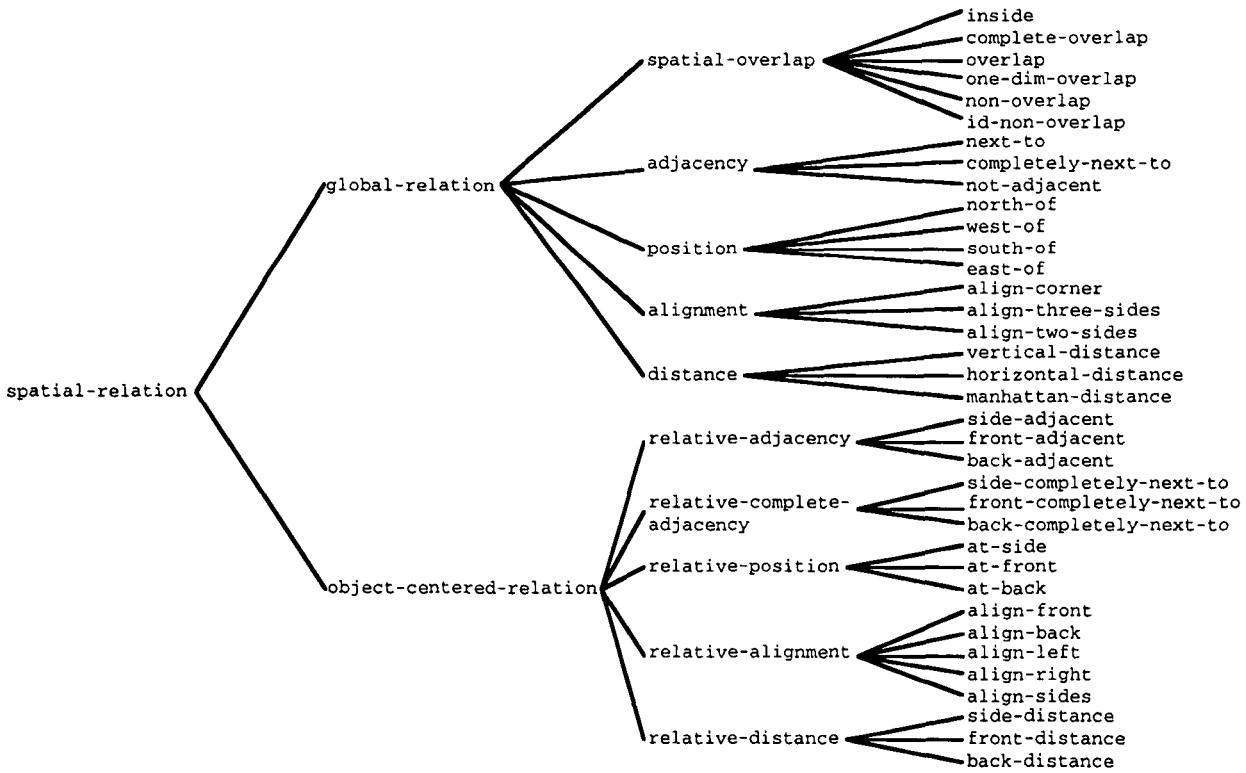


Fig. 8. The set of spatial relations defined in WRIGHT.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

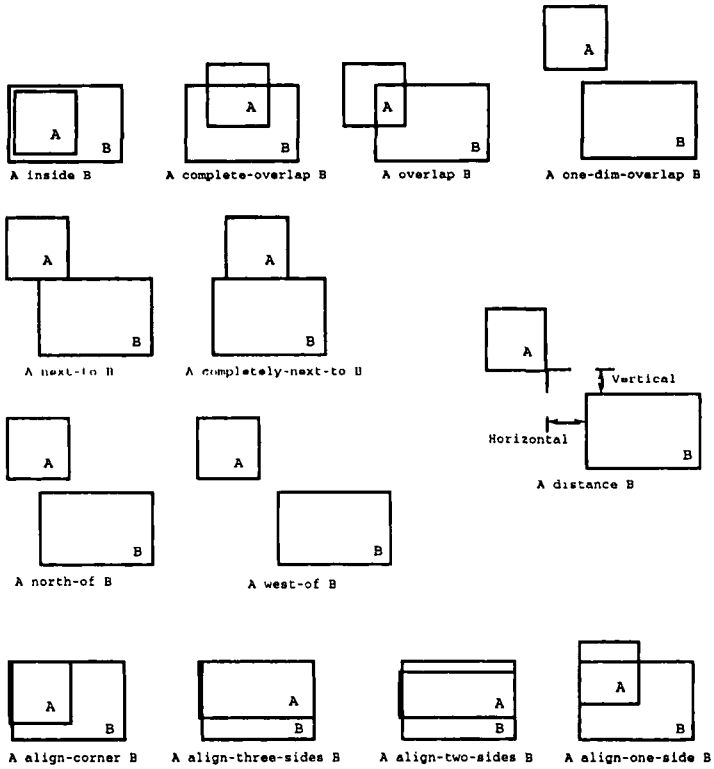


Fig. 9. Some global relations.

global relations are seen in Figure 9. Also, some spatial relations of both types require the specification of a dimension, such as, distance, relative distance, adjacency, and relative adjacency. Thus, they are not purely qualitative or topological, but geometrical.

There are object-centered relations corresponding to all global relations except spatial-overlap. Configurations illustrating some object-centered relations are shown at the bottom row of Figure 10. These are similar to their global counterparts, except they also depend on the orientation of the first design unit, that is, *A back-adjacent B* depends on the orientation of A. The parallel, perpendicular, clockwise (cw) and counterclockwise (ccw) relations shown at the top row in the same figure are orientation relations.

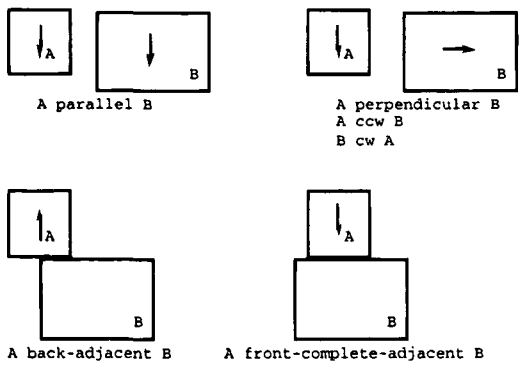


Fig. 10. Some object-centered and orientation relations.

The relative orientations of both design units are considered. Certain types of relations are difficult to express in terms of spatial relations between two design units. For example, the requirement that two design units be physically accessible from each other, or that there be a path between two design units with minimal clearance at each point, or that a design unit has an unobstructed line of sight to another. The difficulty is that an arbitrary number of additional design units, changing with the layout, may be involved in the desired relation. Currently, style constraints express general conditions affecting all or most of the design units in a layout but they are not easy to extend to handle the cases above.

5. REPRESENTING PROBLEM REQUIREMENTS

The efficiency apartment problem is defined in Figure 11 in terms of spatial relations. Performance constraints specify the adjacency requirements. Limits on areas and dimensions, which are also performance constraints, are not shown as they are not spatial relations. Realizability constraints specify that all rooms are inside the apartment and that they do not overlap. Style constraints eliminate trivial holes inside the apartment. The edges of the apartment, which is the design envelope, are indicated by N, S, E, W. Representing all requirements declaratively and explicitly rather than building them into the generator gives control to the

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

Performance Constraints
C-1 living room next-to vestibule ≥ 90
C-2 kitchen next-to living room ≥ 120
C-3 bath next-to vestibule ≥ 70
C-4 living room completely-next-to S
C-5 living room completely-next-to W
C-6 vestibule next-to E

Realizability Constraints
C-7 vestibule inside apartment
C-8 living room inside apartment
C-9 kitchen inside apartment
C-10 bath inside apartment

C-11 vestibule non-overlap living room
C-12 vestibule non-overlap kitchen
C-13 vestibule non-overlap bath
C-14 living room non-overlap kitchen
C-15 living room non-overlap bath
C-16 kitchen non-overlap bath

Style Constraints
C-17 bath north-adj (living room ∨ vestibule ∨ kitchen ∨ N)
C-18 bath south-adj (living room ∨ vestibule ∨ kitchen ∨ S)
C-19 bath east-adj (living room ∨ vestibule ∨ kitchen ∨ E)
C-20 bath west-adj (living room ∨ vestibule ∨ kitchen ∨ W)
C-21 kitchen north-adj (vestibule ∨ living room ∨ bath ∨ N)
C-22 kitchen south-adj (vestibule ∨ living room ∨ bath ∨ S)
C-23 kitchen east-adj (vestibule ∨ living room ∨ bath ∨ E)
C-24 kitchen west-adj (vestibule ∨ living room ∨ bath ∨ W)
C-25 living room north-adj (vestibule ∨ kitchen ∨ bath ∨ N)
C-26 living room south-adj (vestibule ∨ kitchen ∨ bath ∨ S)
C-27 living room east-adj (vestibule ∨ kitchen ∨ bath ∨ E)
C-28 living room west-adj (vestibule ∨ kitchen ∨ bath ∨ W)
C-29 vestibule north-adj (living room ∨ kitchen ∨ bath ∨ N)
C-30 vestibule south-adj (living room ∨ kitchen ∨ bath ∨ S)
C-31 vestibule east-adj (living room ∨ kitchen ∨ bath ∨ E)
C-32 vestibule west-adj (living room ∨ kitchen ∨ bath ∨ W)}

Fig. 11. Disjunctive constraints formulating the efficiency apartment problem.

users but results in the order of n^2 disjunctive constraints for n design units.

The performance, realizability, and style constraints that define a spatial synthesis problem are formulated as Boolean combinations of atomic constraints. The canonical form of a disjunctive constraint is defined to be disjunctive normal, that is, the top level elements, called disjuncts, are connected by an *or*, the second level elements by an *and*, and there are at most two levels. Thus, a disjunctive constraint C_i is in the form $C_i = (d_{i1} \vee d_{i2} \vee \ldots \vee d_{ik(i)})$, and each disjunct d_j is in the form $d_j = (c_{j1} \wedge c_{j2} \wedge \ldots \wedge c_{jk(j)})$, where the c are atomic constraints.

Disjunctions result from a spatial relation that can be satisfied in topologically distinct ways or a requirement that can be satisfied by more than one design unit. In the first case, each disjunct corresponds to a topologically different way of satisfying the constraint. For example, two design units can be adjacent by the first being to the north, south, east or west of the second; thus, an adjacency constraint has four disjuncts. A spatial relation needs to be formulated by a set of mutually exclusive and exhaustive disjuncts. Disjuncts that are not mutually exclusive result in duplicate solutions. A set of disjuncts that is not exhaustive results in missing possible solutions, that is, an incomplete generation process. The second cause of disjunction is if more than one design unit can satisfy a requirement. For example, the requirement that the kitchen should be adjacent to either the living room or the vestibule can be expressed by a disjunctive constraint with eight disjuncts. C-17 in Figure 9 expresses that the west edge of the bathroom should be adjacent to the living room, vestibule, kitchen, or the west edge of the apartment. Because the bathroom can be adjacent to more than one of these, the disjuncts of C-17 are not mutually exclusive.

Spatial relations are defined by templates that show the disjunctive and conjunctive combinations of algebraic constraints between the edges of the two design units. For example, north-of-relation consists of one disjunct that consists of one bounded distance constraint; *i north-of j* is defined as the south edge of i being north-of the north edge of j, $y_j - Y_i \in [0,\infty]$. The built-in spatial relations, as well as new ones, are defined in the same way, by defining its template. A constraint compiler creates the atomic constraints given the templates defining spatial relations and the disjunctive constraints on design units.

Let the edges of the bath and the vestibule be as seen in Table 3, and let C-3 be the constraint *bathroom next-to vestibule* ≥ 70 cm. C-3 is seen in Figure 12 in disjunctive normal form. Some relations, for example, inside, result in a disjunctive constraint with a single disjunct, as in C-10. Disjunctive constraints can share disjuncts and atomic constraints. The third disjunct of C-3 and the second disjunct of C-17 are very similar except they specify minimum adjacent distances of 70 cm and 1 cm.

6. SOLUTION METHOD

6.1. Search algorithm

The disjunctive CSP is solved by instantiating disjunctive constraints by backtracking search. A new state is created for each disjunct. The layout in the new state is modified by asserting the atomic constraints in the disjunct and propagating constraints. If an inconsistency is detected, then the current state is eliminated and backtracking occurs. Otherwise forward-checking evaluates the remaining disjunctive constraints and removes disjuncts and atomic constraints that are satisfied or contradicted as a result of asserting the disjunct. If a disjunctive constraint is left with a single disjunct in its domain after forward-checking, it can be instantiated in the same state. This is termed the *singleton-disjunct heuristic* or the singleton heuristic. It may cause more than one disjunctive constraint to be instantiated in a state. The cycle of forward-checking future disjunctive constraints and finding and instantiating those that have a single disjunct left is repeated until no singleton disjuncts are found in a pass.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

C-3 bathroom next-to vestibule ≥ 70
$(((x_4 - X_2 \in [0, 0]) \wedge (Y_4 - y_2 \in [70, \infty]) \wedge (Y_2 - y_4 \in [70, \infty])) \vee$
$((x_2 - X_4 \in [0, 0]) \wedge (Y_4 - y_2 \in [70, \infty]) \wedge (Y_2 - y_4 \in [70, \infty])) \vee$
$((y_4 - Y_2 \in [0, 0]) \wedge (X_4 - x_2 \in [70, \infty]) \wedge (X_2 - x_4 \in [70, \infty])) \vee$
$((y_2 - Y_4 \in [0, 0]) \wedge (X_4 - x_2 \in [70, \infty]) \wedge (X_2 - x_4 \in [70, \infty])))$

C-10 bathroom inside apartment
$((x_4 - x_0 \in [0, \infty]) \wedge (X_0 - X_4 \in [0, \infty]) \wedge (y_4 - y_0 \in [0, \infty]) \wedge (Y_0 - Y_4 \in [0, \infty]))$

C-13 vestibule non-overlap bathroom
$(((x_2 - X_4 \in [0, \infty]) \wedge (Y_2 - y_4 \in [1, \infty]) \wedge (Y_4 - y_2 \in [1, \infty])) \vee$
$((x_4 - X_2 \in [0, \infty]) \wedge (Y_2 - y_4 \in [1, \infty]) \wedge (Y_4 - y_2 \in [1, \infty])) \vee$
$(y_2 - Y_4 \in [0, \infty]) \vee$
$(y_4 - Y_2 \in [0, \infty]))$

C-17 bathroom north-adj (livingroom ∨ vestibule ∨ kitchen ∨ N)
$(((y_4 - Y_1 \in [0, 0]) \wedge (X_4 - x_1 \in [1, \infty]) \wedge (X_1 - x_4 \in [1, \infty])) \vee$
$((y_4 - Y_2 \in [0, 0]) \wedge (X_4 - x_2 \in [1, \infty]) \wedge (X_2 - x_4 \in [1, \infty])) \vee$
$((y_4 - Y_3 \in [0, 0]) \wedge (X_4 - x_3 \in [1, \infty]) \wedge (X_3 - x_4 \in [1, \infty])) \vee$
$(y_4 - y_0 \in [0, 0])))$

Fig. 12. Expressing spatial relations between design units as disjunctive constraints.

Forward-checking infers the status of an atomic constraint by checking the domains of its variables. If their domains contain no values that can satisfy the atomic constraint, as in the following example: $c_1 : v_1 - v_2 \in [0,\infty], v_1 \in [100, 200]$ and $v_2 \in [300, 400]$; forward-checking can infer that c_1 is violated. A constraint is inferred to be satisfied when all combinations of the values of its variables satisfy the constraint. If $v_1 \in [500, 600]$ and $v_2 \in [300, 400]$, then every possible combination of values for v_1 and v_2 satisfies c_1. If neither of the above two cases hold, then the status of the constraint must remain undetermined. This is the case if $v_1 \in [300, 400]$ and $v_2 \in [300, 400]$, because value combinations that satisfy or violate c_1 are both possible.

Efficiency of forward-checking depends on tightness of bounds derived by constraint propagation. Since path-consistency updates all relations and achieves global consistency, a constraint can be checked by simply comparing the value derived by path-consistency against the value specified in the constraint. Arc-consistency derives looser bounds that may contain inconsistent values when there are loops in the constraint network, thus some inconsistent combinations are not removed by forward-checking but can only be discovered by asserting the disjuncts during search. Path-consistency reduces search by detecting violated disjuncts earlier than arc-consistency does.

The search tree generated while solving problem 1 is shown in Figure 13. Nodes of the tree are the search states numbered in the order of generation. The constraint that is instantiated in each intermediate state is written under the state. States 1, 17, 32, and 39 have two lists to their left. The top list shows the disjunctive constraints that forward-checking finds to be satisfied initially. The bottom list shows the disjunctive constraints instantiated by the singleton-disjunct heuristic or found to be satisfied by forward-checking during successive applications. The configuration(s) at each solution state is drawn below the state.

Forward-checking does not find any satisfied disjunctive constraints in state 1, thus the top list for state 1 is empty. The singleton heuristic finds some constraints with only one disjunct. These are the inside constraints, C-7–C-10, and the fixed adjacencies, C-4–C-6, which are instantiated. Forward-checking and the singleton heuristic are applied again. Forward-checking detects that C-26, C-28, and C-31 are satisfied. C-2 *kitchen next-to living room* ≥ 120 cm is selected from the remaining constraints for instantiating at state 1. C-2 has two undetermined disjuncts at this point: the kitchen can be adjacent to the north or the east of the living room. The other disjuncts have been removed by forward-checking. In state 2, the kitchen is placed to the east of the living room, and in state 17, to the north of it. In all solutions derived from these states, the kitchen and the living room remain in the same position with respect to each other. Placing the kitchen adjacent to the north of the living room at state 17 satisfies C-14 *living room non-overlap kitchen*, C-22 *kitchen south-adj (vestibule ∨ living room ∨ bathroom ∨ S)*, and C-25 *living room north-adj (vestibule ∨ kitchen ∨ bathroom ∨ N)*. There are no singleton disjuncts, and search continues by selecting C-1 *living room next-to vestibule* ≥ 90 cm. In state 18, the vestibule is placed adjacent to the east of the living room and in state 32 to the north of it.

Problem 1 has 22 significantly different solutions. This method finds the solutions without looking at any dead ends because forward-checking is able to remove the disjuncts that lead to dead ends before they are instantiated. In more complicated problems, it is not possible to remove all dead ends even though the search space that is expanded is a very small portion of the total problem space.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

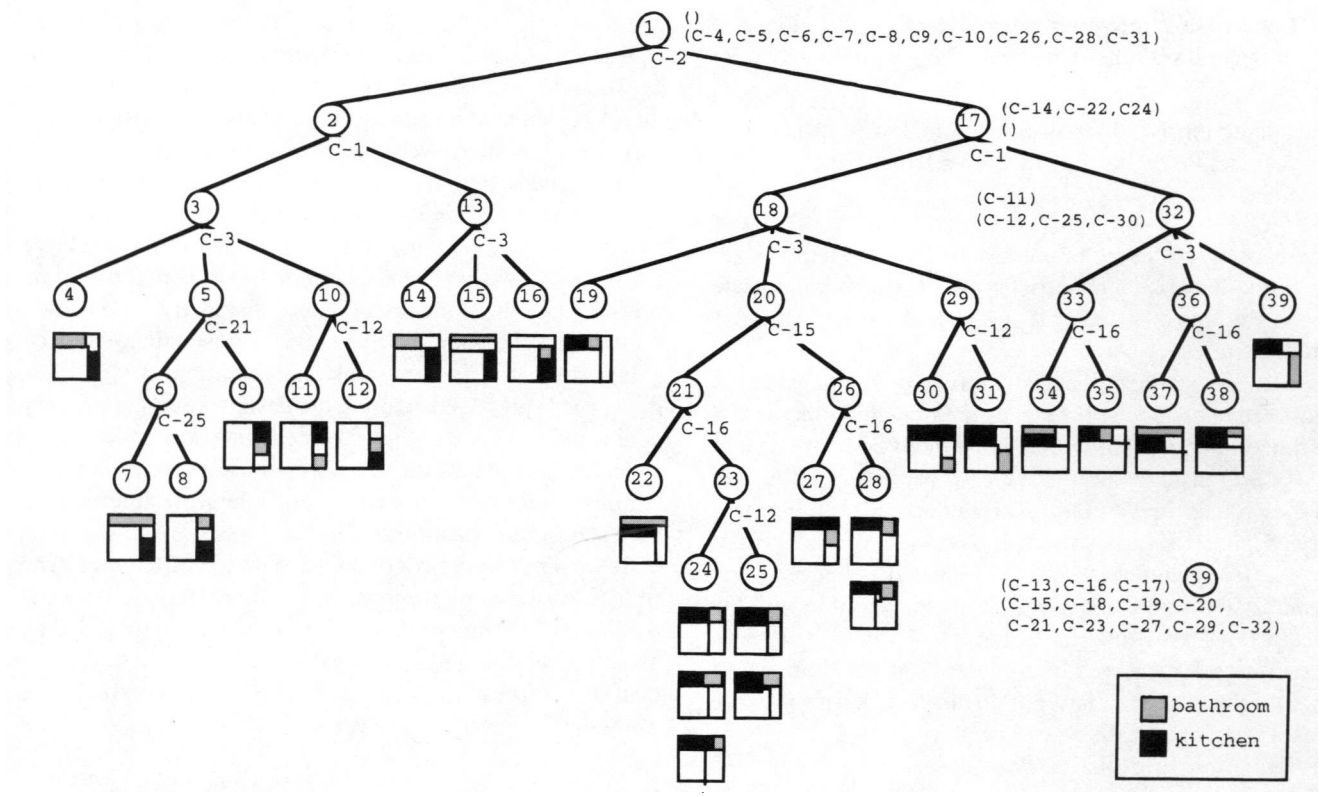


Fig. 13. Search tree for the efficiency apartment problem.

6.2. Problem space

The problem space is the Cartesian product of the domains of all disjunctive constraints. Given d disjunctive constraints each having b disjuncts, there are b^d candidate solutions, where d is the depth of the search tree and b is the branching factor. Thus, the number of candidate solutions for the efficiency apartment problem is 4^{25} without considering duplicate disjuncts. According to the analysis above, additional constraints should increase search complexity measured by the number of states. But, additional constraints on the same variables increase inconsistent combinations. Constraint propagation and forward-checking detects inconsistent and redundant disjuncts, pruning many possible search paths.

A more realistic measure of search complexity in constraint problems is *problem difficulty* as defined by Purdom (1983). Problem difficulty decreases as the ratio of constraints to variables increases and is categorized as hard, difficult, and easy. Hard problems have an exponential number of solutions, and it takes exponential time to solve them by backtracking search. Difficult problems have a polynomial number of solutions, but backtracking still takes exponential time. Easy problems have a polynomial number of solutions, and there are known procedures for solving them in polynomial time. Purdom showed that for a subset of difficult problems, backtracking takes polynomial time with dynamic order of instantiation and exponential time with a fixed order of instantiation. He also conjectured that dynamic instantiation may save exponential time throughout the difficult region even though the resulting times may still be exponential. As they are initially given, layout problems usually do not contain enough constraints to restrict the solutions to a small set. Thus, layout problems are hard. But, design process is characterized by identification of new constraints and relaxation of constraints (Akin et al., 1992). Baykan (1991) conjectured that by modifying constraints, the designer changes hard problems into difficult ones. This enables the designer to evaluate the possibilities and trade-offs implicit in a problem definition by considering all solutions.

6.3. Dynamic determination of instantiation order

In WRIGHT, a disjunctive constraint is selected for instantiation by a function of textures. Textures are measures of features of the constraint graph, topological, and otherwise (Fox et al., 1989). Textures can be on variables, atomic constraints, and disjunctive constraints. We have defined three texture measures.

1. Looseness-1 is the number of active disjuncts in a disjunctive constraint.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

2. Looseness-2 measures the reduction in the domains of interval variables due to satisfying a disjunctive constraint.
3. Interaction is a measure of the relationship between disjunctive constraints that arise from sharing design units.

Textures assign values to future disjunctive constraints. Variable and value ordering heuristics and textures are not the same; a variable or value ordering heuristic is a function of a set of textures.

Looseness-1 selects the disjunctive constraint with the fewest active disjuncts. It reduces the branching factor and maximizes the probability that a disjunctive constraint will fail. If every disjunct has the same probability of failing, P, and the probabilities of failure are independent, the probability of failure of a disjunctive constraint with n active disjuncts is P^n (Haralick & Elliott, 1980). This heuristic has been used in discrete CSPs by picking the variable with the fewest values remaining in its domain (Purdom, 1983).

Looseness-2 measures the reduction in the domains of variables due to instantiating a disjunctive constraint. The domain size of a variable v, $v \in [v_{\min}, v_{\max}]$, is $v_{\max} - v_{\min}$. The reduction in domains due to asserting a bounded difference constraint is calculated by adding the increases in the lower bounds and the decreases in the upper bounds of the two variables. The looseness-2 measure of a disjunct is the sum of the reductions due to its atomic constraints, and that of a disjunctive constraint is the minimum of its disjuncts. Looseness-2 considers the dimensions and the current locations of the design units, and the severity of the spatial relations between them in a uniform and principled way. Locating larger design units first has been found to be the best strategy when the problem involves arrangements that fill available space tightly (Pfefferkorn, 1971). Using spatial relations that project smaller areas has been shown to be a fail-first strategy when the intersection of the areas projected by two spatial relations depend only on the magnitudes of the areas (Eastman, 1973).

The third texture, interaction, is a measure of the relationships between future disjunctive constraints that arise due to sharing variables. The variables considered are design units. Each design unit is assigned a weight based on the number of constraints on it. The interaction measure of a disjunctive constraint that specifies a spatial relation between two design units is the sum of the weight of its design units. For a disjunctive constraint whose disjuncts are between different design units, the interaction of each disjunct is calculated by adding the weights of its design units, then the minimum interaction of its disjuncts is taken as the value of the disjunctive constraint. The width of the constraint graph is dynamically minimized. Selecting a disjunctive constraint that interacts strongly with future disjunctive constraints maximizes the likelihood that disjuncts will be pruned from the domains of future disjunctive constraints by forward-checking. A heuristic is to instantiate the variable participating in more constraints (Purdom, 1983). Eastman (1973) proposed entering the design unit most strongly connected to those already located. In WRIGHT, forward-checking propagates the effects of decisions to future constraints, therefore this texture considers the interactions between future constraints.

Selection is by a lexicographic function of textures. The first texture assigns ratings to all future disjunctive constraints and eliminates those with lower values. If more than one constraint remains, the next texture is applied. If more than one constraint remains after applying all textures, one of them is selected at random.

6.4. Performance of textures

To evaluate the effectiveness of textures in reducing search, we solved six problems using all possible lexicographic orderings of the textures, as well as randomly without using textures. Table 6 shows the performance of selected texture functions in solving the courtyard apartment problem. The texture functions are seen in the first column. The problem is solved 12 times using each texture function, starting the random number generator with a different seed each time. The entries in the table are the average of the 12 runs. If the texture function selects a unique constraint in every state, then all runs become identical. If the texture function is not able to pick a unique constraint at each state, and a constraint is selected randomly then each trial is different. When no textures are used, the variation between trials is larger. Tests were run on a Decstation 5000, using Allegro common lisp.

The number of states expanded given in row 2 and total search time given in row 1 are good indicators of search efficiency. Rows 3 and 4 show the average branching factor

Table 6. *Performance of textures in solving the courtyard apartment problem*

	L1/L2	L2/L1	L2	L1/Int	Int/L1	Int	L1	Random
Seconds	6.4	8.8	8.6	15.1	17.2	18.6	17.6	65.9
# States	321	337	350	523	609	652	822	2548
b	2.00	2.74	2.85	2.00	2.64	2.75	2.00	3.09
d	8.36	5.77	5.60	9.06	6.61	6.40	9.68	6.96

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

and depth of the search tree. Columns of Table 6 are arranged from left to right in decreasing search efficiency. The best texture function is looseness-1 and looseness-2, which reduces search states by 87% and total time by 90% compared to random selection of constraints. All texture functions reduce search. Even the texture function that performs worst, looseness-1 by itself, reduces search states by 68% and search time by 73% compared to random selection. In some cases, adding another texture to some combination of textures does not improve performance. Those combinations are not shown in Table 6.

Looseness-1 used first reduces the branching factor by selecting a disjunctive constraint with the fewest disjuncts. In the courtyard apartment problem, there is a constraint with two disjuncts at every state. Looseness-2 reduces the depth of the search tree by implementing prune-early and fail-first strategies. Using the two in combination results in the best performance, as seen in the first two columns of Table 6. Looseness-2 also performs well by itself. Interaction reduces search depth to a lesser extent than looseness-2, and reduces the branching factor to a lesser extent than looseness-1. Looseness-1 combines well with other textures, especially looseness-2 when used first. Interaction does not combine well with the other textures, thus it is less useful. Interaction measure is expensive to use because it is more complicated to compute. This may explain some of the increased search times when using this texture. In general, texture functions that reduce search also reduce the time per state.

Next we look at how textures perform on various problems. The complexity of a layout problem depends on the number of design units being arranged, the number and type of disjunctive constraints, and how constraining the initial state of the layout is. Problem complexity or difficulty increases with the number of design units. Given the same number of design units, a problem becomes less difficult if more constraints are available. Table 7 shows the number of design units to locate plus the number of design units fixed in initial state, the number of disjunctive constraints, the number of solutions, the number of states expanded by the best texture function, and the percent reduction in number of states by the best texture function compared to random search. The last column gives the best texture function.

Based on the number of states that must be expanded and the number of solutions, we can say that the top three problems are more difficult than the bottom three. In the easier problems, the best texture function reduces the number of search states by 34–67%, and in the more difficult problems it reduces search states by 80–90% compared to random selection. The courtyard apartment problem is solved by expanding fewer states than blocks-3, because of the number and type of constraints available. There are also adjacency, no-trivial-holes and fill-corners constraints in the courtyard apartment problem in addition to the nonoverlap and dimensional constraints in blocks-3.

Textures reduce search by an order of magnitude in difficult problems and find an instantiation order that solves the problem with minimal or no dead ends in easy problems. Based on our observations, the texture functions to use based on problem characteristics are as follows: in easy problems, looseness-1 performs better than others when used alone; otherwise, looseness-2 is best. In bin-packing problems, looseness-2/looseness-1/interaction; and in architectural layouts, looseness-1/looseness-2 combinations work better. Interaction improves performance in all but one problem when used last. The relation between the effectiveness of textures and problem characteristics need to be investigated more, but we have not considered creating random problems with desired characteristics to systematically look at this issue.

Using path-consistency results in more efficient search than arc-consistency. The courtyard apartment problem is solved in 760 states by the earlier version of WRIGHT (Baykan, 1991) compared with 321 states in the current version as shown in Table 6. There is no difference in the number of states between the two versions when solving blocks problems. Because the blocks have fixed dimensions, search is not affected by ambiguities of interval arithmetic.

6.5. Constraint relaxation and optimization

The efficiency apartment example defines the disjunctive CSP as a satisficing problem; all of the disjunctive constraints have to be satisfied by a feasible solution. It may not be possible to satisfy all constraints. It is useful to in-

Table 7. *Performance of textures on selected problems*

Problem	No. of design u.	No. of disj. constr.	No. of solutions	Best no. of states	Reduction in states (%)	Best texture fn
Blocks-3	7 + 1	25	96	1076	90	L2/L1/Int
Courtyard Apt.	9 + 1	75	45	321	87	L1/L2
Blocks-2	6 + 1	25	72	509	80	L2/L1/Int
Blocks-1	6 + 1	15	24	111	67	L1/L2/Int
Kitchen-1	7 + 8	59	3	25	64	L2/L1/Int
Kitchen-4	7 + 8	57	2	29	34	L1/L2/Int

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

dicate which constraints are relaxable, and also which should be relaxed before others. Omitting style and performance constraints results in meaningful configurations, whereas solutions violating realizability constraints may not be admissible. In WRIGHT, relaxable (soft) constraints are indicated by including a null disjunct in their domains, which is trivially satisfied. Let C_i be a disjunctive constraint, and d_{ij} the disjuncts in its domain. If C_i is soft, a null disjunct $d_\varnothing$ is added to its domain: $C_i = (d_{i1} \vee \ldots \vee d_{ik(i)} \vee d_\varnothing)$. The null disjunct is the same as omitting the constraint. When it is instantiated, it has no effect other than lowering the evaluation of that state. This formulation is used in kitchen layout problems discussed below.

This formulation can be extended by assigning a utility between 0 and 1 to every disjunct. The disjuncts with lower utilities are relaxations of those having the highest utility. An objective function that can be used to evaluate intermediate and final states is given below:

$$\text{Rating} = \sum_{i \in \text{satisfied constraint}} u_i + \sum_{j \in \text{future constraint}} \text{Max}(u_{jk}).$$

The utility of a satisfied constraint, u_i, is the utility of the instantiated disjunct. The utility of a future constraint, u_{jk}, is the maximum of the utilities of the disjuncts in its domain (Fox, 1987). The objective function can be a function of any set of variables in the problem. For example, the area of the apartment can be minimized in the efficiency apartment problem, or the weighted total distance between design units can be minimized in a computer board layout problem. It is possible to terminate search paths that cannot lead to a better solution than the current best.

The largest problem we have attempted is the layout of a computer board. There are 14 design units that are fixed on the board and 20 that are being configured. The elements on the board are connected by wires that run in the two main directions. This is an optimization problem where the weighted manhattan distance between the design units are minimized. It is an underconstrained problem with an exponential number of solutions, and it was not possible to exhaust all alternatives even when the system worked for days. It is possible to derive constraints from the objective function, such as, design units connected to each other by the most number of wires should be adjacent or close (maximum distance constraint). Adding these constraints simplifies the problem by orders of magnitude.

It is more efficient and may be also more useful to satisfice rather than to optimize. Minimizing the area of the apartment gives one solution as the best, whereas setting a limit on the area and satisficing gives all possible solutions having an area less than the limit. Then by decreasing or increasing the area iteratively it is possible to see the trade-offs. In most spatial layout problems, the designer has a pretty good idea of the limits that can be achieved.

7. REPRESENTING DOMAIN KNOWLEDGE

Are spatial relations between pairs of design units and the approach described above adequate to represent knowledge of a design domain? We have used kitchens to test this because kitchen layout is complicated enough to be challenging, and because kitchen layout knowledge has been formulated extensively. We have tried to express kitchen layout knowledge as spatial relations between prototype design units and used it to solve different kitchen layouts.

7.1. Kitchen domain knowledge

The basics of kitchen layout are as follows: There should be well-defined centers for cooking, mixing, and the sink. They should be arranged based on their sequence of use during food preparation. Enough countertop area to meet various functions and cabinet frontage to meet storage needs must be provided. The work triangle formed by the front midpoints of the sink, range, and refrigerator should have a total length of 360–660 cm. If the appliances are too close together, there is not enough work area between them. If they are too far apart, users have to walk too much. There should be no traffic or furniture interfering with the work triangle. Because most of the time spent in the kitchen is spent at the sink, placing the sink against a window provides light and view while working. For safety reasons, the range should not be next to a window or door. It is desirable to have all the work centers in a kitchen connected, with the refrigerator at one end. If it is not possible to have all work centers connected, they may be split (Drake & Pheasant, 1984; Gulliver, 1984; Jones & Kapple, 1984; Miell, 1984; Prizeman, 1984; Small Homes Council, 1950).

There is a taxonomy of prototype design units for organizing constraints. Inheritance of constraints through the taxonomy eliminates duplication. The design units in a particular problem are instances of the prototypes and inherit constraints from them. The design unit taxonomy used to formulate kitchen constraints is shown in Figure 14. The design units to be configured in the example problems are sink, range, refrigerator, sink center, mix center, range center, and circulation area. A design unit can itself be composed of other design units. The different levels of detail are called

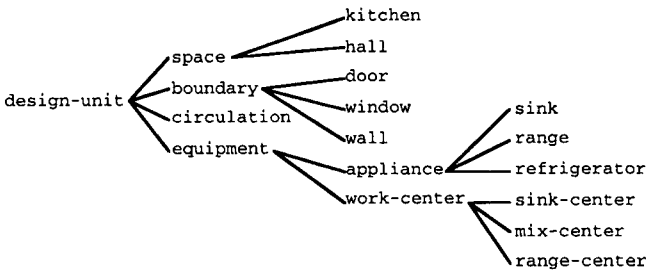


Fig. 14. Taxonomy of kitchen design units.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

levels of aggregation. In kitchens, we are dealing with design units at three levels; the kitchen, the work centers, and the appliances. WRIGHT permits designing at different levels concurrently.

Most of the requirements can be expressed in terms of desired or unwanted spatial relations between two design units. The work triangle can be split into distances between every pair of appliances. This results in looser bounds because taking the minimum in each case results in a lower bound that is less, etc. The distances are approximated by manhattan distance between the edges of the appliances. Work centers should be next to other work centers or to a boundary element at sides. The constraint that the sink center should be next to the mix center has a null relaxation, which permits splitting the counters into two groups if required. Other conditions, such as, there should not be any furniture or traffic interfering with the work triangle is very hard to express. It is a global constraint that is a function of all design units. These are the performance constraints.

The realizability constraints are that the work centers and the circulation area should be inside the kitchen and they should not overlap. The sink should be inside the sink center and the range inside the range center. There should be one circulation area, and all doors must completely overlap it on one side. The fronts of appliances should be completely next to the circulation area, and the fronts of work centers should be next to the it. In all of the 50 plus kitchen designs given in Small Homes Council (1950), the appliances and the work centers were always placed with their backs against a wall; there were no island kitchens. Thus, it is also added as a constraint. These result in 64 constraints on the prototype design units (Baykan, 1991). There are no style constraints.

7.2. Constraint compiler

The constraint compiler formulates a disjunctive CSP by applying the domain knowledge to the design unit instances in a particular problem. Domain constraints express general knowledge about the design domain in the form of desired spatial relations between prototype design units or in the form of limits on their dimensions. Because constraints expressing domain knowledge are posted to prototype design units, they must also contain the quantifiers *all* and *some* to designate how they apply to instances. The constraint compiler takes a taxonomy of prototype design units, domain constraints on them, and the templates defining spatial relations in terms of atomic constraints; and creates the disjunctive constraints on the design unit instances.

The user may decide to define a problem using disjunctive constraints rather than domain constraints. They can also be specified declaratively. In this case, the constraint compiler creates the atomic constraints using the templates defining spatial relations. A disjunctive CSP for even a small problem, such as the efficiency apartment, is quite complex and tedious to generate manually. The constraint compiler lets the user specify it at a high level, in a user-friendly manner.

The philosophy behind the representation of domain knowledge in WRIGHT is to do it explicitly and declaratively, so that the users can have direct control. The three components of the knowledge base: the taxonomy of design units, the set of spatial relations, and the domain constraints are represented declaratively and can be modified through the user interface. New spatial relations are defined the same way the built-in relations are.

7.3. Performance of spatial relations and constraints

One of the books (Small Homes Council, 1950) gives a catalogue of 50 plus small home kitchens. These are classified with respect to position of door(s) and window. There are three differently sized kitchens for each window and door configuration. The kitchens are between 7–12 square meters, rectangular in shape, and have one or two doors and one window. The best layout attainable in each kitchen is also given, together with a rating that can be used to compare different kitchens. The test cases selected from this catalogue are solved using the same domain knowledge. There are some differences between the layouts given in the handbook and those generated by WRIGHT. In WRIGHT's layouts, doors open inside, whereas in the solutions given in (Small Homes Council, 1950), it is not specified, even though the layout is such that doors can open inside if need be, and the work centers are not shown as separate areas but as a continuous counter, sometimes L-shaped, that contains the sink and range.

The first plan in each row in Figure 15 is the solution given in the *Handbook of Kitchen Design*, and the rest are the solutions generated by WRIGHT. In WRIGHT's layouts, all work centers are separate rectangles, and the mix center is indicated by diagonals inside it. Having separate rectangles instead of connected counters causes spurious alternatives to be generated. For example, in the last two solutions in the top row, the counter in the top right corner is part of the sink center in one and range center in the other solution. This is also observed in the bottom left corners of the first two solutions in the second row. In all five cases, one of the solutions found by WRIGHT is the same as the one given in terms of the sequence of work centers and the placement of the appliances. In all kitchens except the last, another sequence of work centers is also permitted by the constraints. These are left-handed and right-handed sequences and both are valid, except the last solution in the third row is not acceptable due to the relation of the door and the refrigerator. A spatial relation that permits the first solution but prevents the second needs to take into account the orientations of the refrigerator and the door.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.



Fig. 15. Solutions to five kitchen layout problems. The solution given in *Handbook of Kitchen Design* is at left, and the other layouts in each row are the solutions by WRIGHT.

In the second and fifth problems, sink center is not next to another work center. This constraint is relaxed because it cannot be satisfied. It is the only constraint that requires a relaxation. Placing the sink next to the window, mix center next to the sink, etc. finds the possible solutions without much search in the above problems. The number of states expanded when solving kitchens 1 and 4 is given in Table 7.

The constraints that solve small house kitchens are inadequate in solving the detached house kitchen shown in Figure 16. There needs to be more than one circulation area so that all doors and all appliances can be adjacent to one. The exact number depends on the configuration; and it is not possible to specify beforehand. It is possible to formulate the problem such that instead of a circulation space, every work center and appliance has its own use area in front of it. The use areas may overlap each other. An alternative formulation may be to omit the circulation area altogether and to specify minimum distances from the fronts of all appliances and work centers to everything else. Both of these formulations run into problems with global conditions, that is, the circulation paths and the work triangle should not be obstructed. As this example shows it is harder to define the realizability constraints than the performance constraints, that is, to define what is a well-formed kitchen. When the constraints are not formulated adequately, WRIGHT either fails by generating thousands of nonsense solutions or by finding no solution and giving no indication of which other constraints to relax.

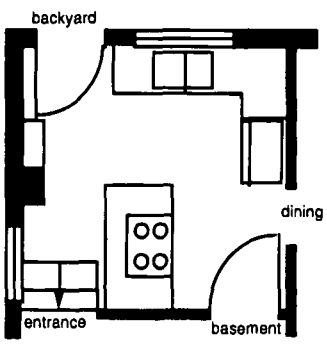


Fig. 16. A detached house kitchen and existing layout.

Two approaches are possible to deal with this issue. One is to have test-only constraints for checking global conditions in the layouts generated by disjunctive constraints. The second approach is to create new design unit instances, such as circulation areas, dynamically during search when they are needed (Mittal & Falkenhainer, 1990). The design units that are used depend on the configuration to some extent. The island counter in the kitchen in Figure 16 is wider than usual and can be used from both sides, and there is a tall cabinet next to the chimney. These are selected because they fit the layout.

Disjunctive constraints specifying spatial relations are not sufficient for expressing deep knowledge about a design do-

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

main. They are constructed on the fly based on deep knowledge about the domain and the possibilities of the problem at hand. Currently this is the role of the user. Role of the spatial layout system is to find a set of significantly different solutions. Design is more about determining the constraints and selecting one of the alternatives based on other considerations rather than finding solutions when the constraints are given. Thus, a system such as WRIGHT is best used interactively for exploring a problem as well as for finding a solution.

7.4. Extensions

Because objects required in a layout sometimes depend on the configuration, it is useful to be able to create new design units during search. An example is the creation of circulation areas when they are needed. New design units would be created when some required spatial relations cannot be formed between existing design units.

The disjunctive constraints specifying spatial relations between pairs of design units are not good for expressing global conditions. It is possible to have procedural or other types of constraints that can express arbitrary conditions and to use them just for testing. A constraint compiler that can express functional and performance requirements in terms of spatial relations between design units can be an area for future research.

WRIGHT can have design units that are at different levels of aggregation, and problems containing a large number of design units can be simplified by creating hierarchical levels. This is made use of in kitchen problems, but all design units and constraints between them are treated the same. The issue is how to make use of levels to increase efficiency and to simplify search.

This formulation can be extended naturally to three dimensions, by making each design unit a rectangular prism defined by six planes, where each plane is parallel to two axes. The same textures and atomic constraints can be used; but 3D spatial relations need to be formulated. It is also possible to extend this formulation to nonrectangular shapes. Arbitrary lines can be defined by treating the four coordinates of a rectangle as the two endpoints of a line. The constraints on them are similar in form to the bounded difference constraints, but their semantics are different. It is not clear whether this formulation is useful for any domain.

8. CONCLUSION

This study demonstrates that it is possible to formulate spatial synthesis as a disjunctive CSP. We have solved satisficing and optimization problems from various domains, requiring different types of constraints. The disjunctive constraints used in defining layout problems are based on a deeper knowledge of problem domains. Formulating functional requirements as spatial relations between design units is an issue that can be addressed in future work.

We give a new $O(n^2)$ incremental path-consistency algorithm for efficiently maintaining consistency of layouts during search. We define textures, which are heuristic measures based on the structure and other features of the constraints, and use them to dynamically select the disjunctive constraint to instantiate. Textures implement fail-first and prune-early strategies and reduce the width of the constraint graph. Our experiments show that textures can reduce search time by an order of magnitude in difficult problems and can find an instantiation order that solves the problem with minimal or no dead ends in easy problems. As they are initially given, layout problems usually do not contain enough constraints to restrict the solutions to a manageably small set, thus the problem has to be explored interactively by modifying the constraints.

ACKNOWLEDGMENTS

This research was supported in part by a grant from Digital Equipment Corporation.

REFERENCES

Akin, O., Dave, B., & Pithavadian, S. (1992). Heuristic generation of layouts (HeGeL): Based on a paradigm for problem structuring. *Environ. Plann. B 19*, 33–59.

Aggoun, A., & Beldiceanu, N. (1993). Extending Chip in order to solve complex scheduling and placement problems. *Math. Computational Modelling 17(7)*, 57–73.

Allen, J.F. (1983). Maintaining knowledge about temporal intervals. *Commun. ACM 26(11)*, 832–843.

Author. (1984). Domestic kitchen design: Conventional planning. *Architects J.*, October, 71–78.

Baybars, I., & Eastman, C.M. (1980). Enumerating architectural arrangements by generating their underlying graphs. *Environ. Plann. B 7*, 289–310.

Baykan, C.A. (1991). *Formulating spatial layout as a disjunctive constraint satisfaction problem.* PhD Thesis. Carnegie Mellon University, Pittsburgh, PA.

Baykan, C.A., & Fox, M.S. (1992). WRIGHT: A constraint based spatial layout system. In *Artificial Intelligence in Engineering Design, Volume 1* (Tong, C. and Sriram, D., Eds.), pp. 395–432. Academic Press, Inc., Boston.

Davis, E. (1987). Constraint propagation with interval labels. *AI 32(3)*, 281–331.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *AI 49*, 61–95.

Dhar, V., & Ranganathan, N. (1990). Integer programming vs. expert systems: An experimental comparison. *Commun. ACM 33(3)*, 323–336.

Drake, F., & Pheasant, S. (1984). Domestic kitchen design: The ergonomists view. *Architects J.*, October, 79–80.

Eastman, C.M. (1973). Automated space planning. *AI 4*, 41–64.

Flemming, U. (1978). Wall representations of rectangular dissections and their use in automated space allocation. *Environ. Plann. B 5*, 215–232.

Flemming, U. (1986). On the representation and generation of loosely packed arrangements of rectangles. *Environ. Plann. B 13*, 189–205.

Flemming, U., Baykan, C.A., Coyne, R.F., & Fox, M.S. (1992). Hierarchical generate-and-test vs. constraint-directed search: A comparison in the context of layout synthesis. In *Artificial Intelligence in Design '92* (Gero, J.S., Ed.), pp. 817–838. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Forbus, K. (1984). Qualitative process theory. *AI 24*, 85–168.

Fox, M.S. (1987). *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann, Los Angeles.

Fox, M.S., Sadeh, N., & Baykan, C.A. (1989). Constrained heuristic search. *Proceedings IJCAI-11*, 309–315.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.

Gulliver, W. (1984). Domestic kitchen design: Specifying built-in furniture. *Architects J.*, October, 91–93.
Haralick, R.M., & Elliott, G.L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *AI 14*, 263–313.
Jones, R.J., & Kapple, W.H. (1984). *Kitchen planning principles—equipment—appliances.* Small Homes Council—Building research Council. University of Illinois, Urbana-Champaign.
Liggett, R.S. (1980). The quadratic assignment problem: An analysis of applications and solution strategies. *Environ. Plann. B 7*, 141–162.
Mackworth, A.K. (1977). Consistency in networks of relations. *AI 8*, 99–118.
Mackworth, A.K., & Freuder, E.C. (1985). The complexity of some polynomial network consistent algorithms for constraint satisfaction problems. *AI 25*, 65–74.
Malik, J., & Binford, T.O. (1983). Reasoning in time and space. *Proc. Eighth Int. Conf. Artif. Intell.*, 343–345.
Miell, C. (1984). Domestic kitchen design: the building regulations. *Architects J.*, October, 95–97.
Mitchell, W.J., Steadman, J.P., & Liggett, R.S. (1976). Synthesis and optimization of small rectangular floor plans. *Environ. Plann. B 3*, 37–70.
Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. Eighth Nat. Conf. Artif. Intell.*, 25–32.
Pfefferkorn, C. (1971). *Computer design of equipment layouts using the design problem solver.* PhD Thesis. Carnegie Mellon University, Pittsburgh, PA.
Prizeman, J. (1984). Domestic kitchen design: Services. *Architects J.*, October, 99–103.
Purdom, P.W. (1983). Search rearrangement backtracking and polynomial average time. *AI 21*, 117–133.
Small Homes Council. (1950). *Handbook of kitchen design.* University of Illinois, Urbana-Champaign. Circular C5.32R.
Smith, B.M., Brailsford, S.C., Hubbard, P.M., & Williams, H.P. (1996). The progressive party problem: Integer linear programming and constraint programming compared. *Constraints 1(1)*, 119–138.
Stallman, M.R., & Sussman, G.J. (1977). Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *AI 9(2)*, 135–196.
Waltz, D. (1975). Understanding line drawings of scenes with shadows. In *Psychology of Computer Vision* (Winston, P.H., Ed.), pp. 19–91. McGraw Hill, New York.

Can A. Baykan is an associate professor, teaching design, computer-aided design and design methods at the Department of Architecture at Middle East Technical University. He received B.Arch. and M.Arch. from METU and a Ph.D. from Carnegie Mellon University. He worked as a research associate at the Robotics Institute and as an adjunct assistant professor at the Dept. of Arch. at Carnegie Mellon. His research interests are in AI, CAD and cognitive studies in design.

Mark S. Fox received his B.Sc. in Computer Science from the University of Toronto in 1975, and his Ph.D. in Computer Science from Carnegie Mellon University in 1983. In 1979 he joined the Robotics Institute of Carnegie Mellon University as a Research Scientist. In 1980 he started and was appointed Director of the Intelligent Systems Laboratory. He co-founded Carnegie Group Inc. in 1984, a software company which specializes in knowledge-based systems for solving engineering, manufacturing, and telecommunications problems, and was its Vice-President of Engineering and President/CEO. Carnegie Mellon University appointed him Associate Professor of Computer Science and Robotics in 1987 (with tenure in 1991). In 1988 he was appointed Director of the new Center for Integrated Manufacturing Decision Systems. In 1991, Dr. Fox returned to the University of Toronto where he received the NSERC Research Chair in Enterprise Integration and was appointed Professor of Industrial Engineering, Computer Science and Management Science. In 1992, he was appointed Director of the Collaborative Program in Integrated Manufacturing. In 1993, Dr. Fox co-founded Fox-Novator Systems Ltd., a company that provides Electronic Commerce services over the Internet.

Downloaded from https://www.cambridge.org/core. 24 Jul 2020 at 16:17:16, subject to the Cambridge Core terms of use.