LEARNING WHILE DESIGNING

GOURAB NATH
Amadeus Development SAS France

and

JOHN S GERO
University of Sydney Australia

Abstract. This paper describes how a computational system for designing can learn useful, reusable, generalized search strategy rules from its own experience of designing. It can then apply this experience to transform the design process from search-based (knowledge-lean) to knowledge-based (knowledge-rich). The domain of application is the design of spatial layouts for architectural design. The processes of designing and learning are tightly coupled.

1. Introduction

This paper describes the application of machine learning for automatically learning heuristics about a design process, gained during the process itself. The distinguishing feature of this work is that the learning and designing processes are tightly coupled with a strong interdependence on each other, which makes learning during designing a part of the design activity. The illustrative domain of application is the design of architectural layouts. Before learning, the design process is modeled as uninformed search; i.e. there is little or no strategy information, which can be used by the process to progress towards good designs and avoiding bad ones. After learning the search process becomes more informed, in the sense that the learned heuristics can be used to reject inappropriate design decisions and select appropriate ones. Rejection of inappropriate and selection of appropriate decisions ultimately lead to better design solutions. Learning takes place from both partial and complete design solutions and is seamlessly applicable as additional strategy knowledge. During the computational design process, if learned heuristics match, they apply, else more heuristics are learned. The

behaviour of the design generator thus gradually changes from search-based to knowledge-based for both immediate and future design search.

The plan of this paper is as follows. First the terminology for the rest of the paper is defined. Then the closely coupled relationship between the design and learning process is described. Following this, the particular mechanism of learning is explained through a small example, after which a simple formalization is presented. Then an architectural design problem is chosen and the ideas are applied. Experimental results with learning are then used to support the claims in this paper. The generality that may be achieved with the learning mechanism is also analysed. Next, an evaluation of the method in terms of strengths, weaknesses, scalability, extensions and distinction with past work is presented.

2. Terminology

It is useful to define some terms that will be used in the rest of this paper. A *pattern* is defined to be a collection of one or more variables representing values, attributes or compositional parts of representational entities. The possible values of variables are constrained by relationships to values of other variables, Figure 1. A pattern can be syntactically specified as a collection of object-attribute-value triplets, each of which can be a variable that can potentially match to data. A variable is identified by a symbol that is called its identifier. The value of a variable can be the identifier of another variable of the pattern. This introduces relations between the values of the variables. A pattern may match data to instantiate the values of each of the variables that define it. A *pattern match* results in an *instantiated pattern*; such a match is possible when every variable in the pattern has at least one instantiation that satisfies all the relations between its value and the values of other variables related to it. Without or before a pattern match, the pattern is an *uninstantiated pattern*. Whenever the term ‘pattern’ is used without any additional qualification, it refers to an uninstantiated pattern.

A design *feature* is an instantiated pattern that can be semantically mapped onto a common human interpretation. This semantic component is the only difference between an instantiated pattern and a feature. For example, a square internal courtyard could be a feature of a building. A *feature class* is an uninstantiated pattern that is an abstract parametric conceptual description of some feature. A feature class when instantiated is a feature. The relation between a feature class and feature is exactly the same as the relation between a class and its instances.

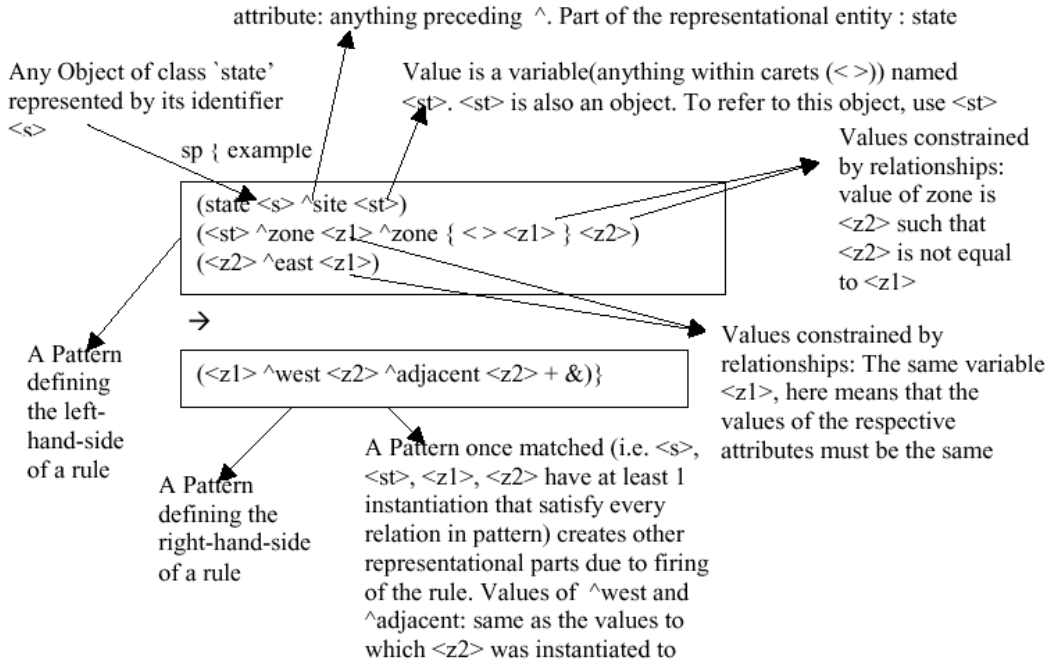


Figure 1: Example of a pattern and its match in the context of a rule in the SOAR syntax (Laird et al 1987)

The term *design context*, Figure 2(b), is the superset of all design information that includes the description of a given design alternative and all information that surrounds it from the point of its generation through its evaluation to its subsequent modification. In this paper, design context comprises a description of the design alternative, design requirements, the subsequent evaluation of the design alternative and design process information. Design process information consists of the parent design context and generative choices for further transformation (elaboration/modification/refinement) of the design alternative from that point onwards. Every design alternative thus has a design context associated with it.

Additional parts may be added to a design or existing designs may be transformed in different combinatorial ways by making generative choices. These combinatorial ways of design generation are referred to as *design decisions*, Figure 2(a). Design decisions are typically sets of rules that apply to a given design alternative and are considered as the knowledge units for design transformation. Each of these rules constitutes a pattern on its left-hand-side (rule precondition) and a pattern on its right-hand-side. The pattern on the right changes the design resulting in a new design when the pattern on the left matches the design context. The rule preconditions are the

necessary condition*s* for a design transformation. When the design is subsequently evaluated, using design evaluation knowledge a new design context is created. Design evaluation knowledge usually is a feature class (or some formula) that associates parts of one or more feature classes whose presence/absence/(or evaluation of the formula) is an indicator of design quality.

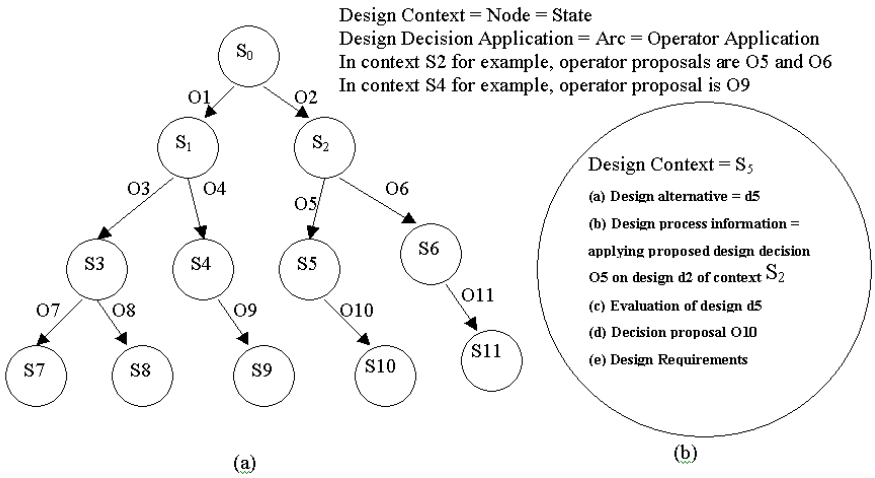


Figure 2. (a) Design computation process mapped onto the state-operator paradigm, and (b) the elements that comprise the design context

A *preference on a design decision* is a flag (+/-) that can be used to prefer (+) or reject (-) design decisions from a set of possible design decisions that may be the choices for transforming a given design alternative. Such preferences are meaningful only under certain conditions. In this work the semantic equivalent of such conditions are design situations.

Structurally, a pattern that comprises a subset of the variables representing the design context is termed a *design situation* in this paper. Typically this is a proper subset. A critical element of the learning process described in this paper is to extract a design situation from a given design context. Each design situation thus represents a class of possible patterns that could be extracted from different design contexts.

A heuristic is an association between a situation and a preference on a design decision. For a heuristic, a situation is a sufficient condition that ensures a given quality of solution (good (+) or bad (-)) resulting from the application of the given design decision. If such associative knowledge is available then the process of preferring a decision on the match of its associated situation is useful as strategy knowledge to augment the process

of design as search. A heuristic is said to match a design context when its situation matches the design context.

3. Design Computation and learning- tight coupling

3.1 DESIGN COMPUTATION PROCESS

Assume that design generation is formulated as a step-by-step combinatorial and constructive process of configuration of the parts of the design. At each step, for a given partial configuration, there are design decisions that are proposed. Transformation opportunities for a given design are combinatorial i.e. there is a tree of design alternatives, each of which will subsequently be the root to other trees emanating from it as a result of future design decisions. At each step of such a process a design decision may be selected and is applied, resulting in a new design. This new design is evaluated using design evaluation knowledge. Once a design alternative is evaluated some learning can be done. How this is done is explained in the next section.

At a lower level of abstraction, the design computation process may be mapped onto the traditional AI state-operator paradigm of search, Figure 2(a), where operators match the state information to transform the existing state into a new state. In such a mapping operators map onto design decisions and states map to design contexts. The state-operator paradigm of search is used in a number of general problem solving architectures such as SOAR (Newell 1990; Laird et al 1987) and PRODIGY (Carbonell et al 1991). The SOAR architecture was chosen to implement the ideas in this paper. SOAR is also an embodiment of a psychological theory of cognition (Newell 1990), an architecture for general intelligence (Laird et al 1987), as well a programming tool for AI. The general ideas used are best described at the knowledge level of computational designing rather than at the level of SOAR. However some two concepts decision proposal and application have been borrowed from SOAR.

Design experience is the historical trace of data that is generated during such a process. As the entire computational design process is rule-based this data is a historical collection of instantiated patterns that matched and were replaced by new instantiated patterns during the process of decision proposal, application and solution evaluation. This trace is the data that is utilized by the learning algorithm used in this paper.

3.2 INTEGRATING LEARNING IN THE DESIGN COMPUTATION PROCESS

The model of learning is tightly coupled with design processes, Figure 3. The notion of appropriateness of designs (absolute (e.g. satisfaction of constraints), relative (e.g. minimizing a global design variable)), controls the

design strategy and how learning is done. In this paper the focus of the demonstrative example is on the satisfaction of constraints. Such control is exercised due to the effects and side-effects of a chain of events that happen as a generated design is evaluated. Evaluation of designs drives positive or negative credit assignment of design decisions. The credit assignment determines whether the consequent of a heuristic is about avoiding or preferring a design decision. Because the antecedent of the heuristic is expressed in terms of the representational space of the design context (see next section for details), the design context influences what is learned. As both of these are encapsulated in the derived heuristic, when the heuristic becomes applicable, it in turn, influences subsequent design generation and hence influences the resulting design description and its quality. The learned heuristic becomes a part of the generation control strategy and in turn influences future design generations in a similar situation by eliminating search effort. This is how learning incrementally changes the design process from being more search-based to being more knowledge-based. Every new unexplored path in the design solution space, presents either an opportunity for learning or an opportunity for using what was learned.

The heuristic applies for any arbitrary design context in which the extracted situation matches. If the heuristic is available at the time of proposal of the design decisions the operator may not even be proposed if the heuristic predicts a bad solution. Therefore these automatically derived heuristics act as strategy knowledge, just like the manually defined heuristics that are often used to prune the space of design alternatives.

3.3 LEARNING MECHANISM EXPLAINED BY SIMPLE EXAMPLE

The learning mechanism used in this paper can be better understood by initially reviewing the concept of explanation-based learning (EBL) in artificial intelligence. One striking difference between EBL and other common forms of similarity-based learning is that the target concept definition already exists. The task of EBL is in reformulating it in terms of what is called *operationality criteria*. Operationality criteria define the space of representational terms that are allowed in the reformulated concept definition. Another difference is that learning is from a single example rather than a set of positive and negative examples. The learning mechanism used in this paper is the chunking method of SOAR, a variant of this method.

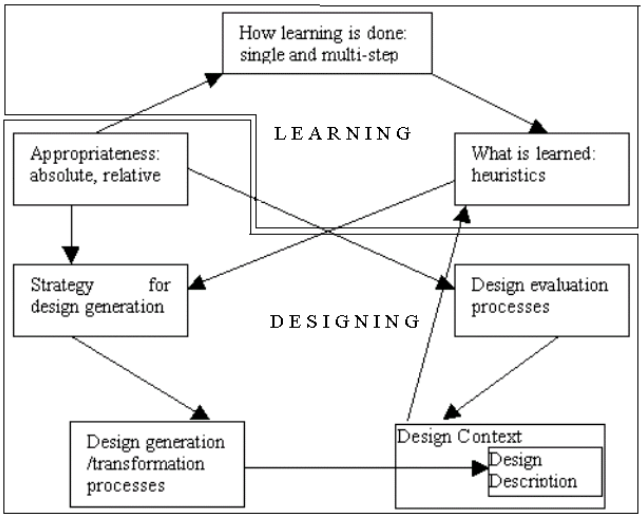


Figure 3. Tight coupling between learning and designing. Arrows signify influences on the box at the head of the arrow by the box at the tail of the arrow.

Let C1 be a *training example* for a *target concept*, here we will use a cup. Let the following features of the cup C1 be represented: it is light, is made of porcelain, has a decoration, has concavity, has a handle, and a flat bottom. Assume that the following five statements constitute a domain theory for the functional definition of a cup. A domain theory in EBL is a set of rules or facts that is used to determine the membership of an arbitrary instance to the target concept. In this context, let the domain theory statements be the following:

a) If an object is stable and enables drinking, it is a cup,
b) If an object has a bottom which is flat, it is stable
c) If an object carries liquids and is liftable, it enables drinking,
d) If an object is light and has a handle, it is liftable, and
e) If an object has a concavity, it carries liquids.

Can it be shown that C1 is a cup? In this case, C1 is indeed a cup. C1's flat bottom makes it stable, it is light weight and has a handle that ensures liftability, its concavity allows it to carry liquids, the liftability and ability to carry liquids enables drinking. The above explanation is an informal proof tree of why the example is a cup. Thus, the relevant features of C1 that determine 'cupness' are lightness, concavity, handle and flat bottom. The rest of the description of the cup are often stated in the context of a cup but are immaterial/irrelevant in diagnosing a cup. The output of EBL is a new rule: If an object is light, has a concavity, has a handle, and has a flat bottom, then it is a cup. Note that the explanation is a set of chains; in each chain one or more initial structural features (facts) match domain theory rules to

generate new intermediate assertions that ultimately imply some functionality of the cup. All these functionalities in combination represent the cup concept. In this case, the operationality criterion was the space of structurally observable features for the cup. To derive these relevant structural features, the proof tree may be also considered to be a trace of a set of inference chains. Traversing these chains backwards (backtracing) from the functional features of the cup to their originating features until all features satisfy the operationality criteria, yields the relevant elements that in the future can diagnose a cup. This is the basic task of any explanation-based or analytical learning algorithm. In practice, this is more complicated, as the left hand side of the derived rule would be composed of patterns rather than constants.

3.4 MAPPING LEARNING CONCEPTS TO THE DESIGN PROCESS

We can map the components of the explanation-based learning example to the design process. The target concept is mapped onto the concept of a good (+) or bad (-) design solution. The training example, an instance of the target concept, is mapped onto a complete or partial design solution that is generated and subsequently evaluated during the search process. The domain theory is mapped onto the design evaluation knowledge that is used to make the evaluation. Then the *proof tree* or *justification basis* for the reformulation of the concept is the part of the design experience from the time the design decision was proposed through the application of the decision to the evaluation of the design alternative. The operationality criteria are mapped onto the representation space of the design context just at the point before the design solution is about to be generated by some design decision. If the above are the mappings of EBL concepts to the parts of the design process, then analogically an EBL backtracing procedure will result in an output rule that is an association between a relevant subset of the parent design context and a preference on the design decision that constructed it.

The role of learning, again, analogically, is to change the representation of the evaluative concept of the quality of a design to "what to do, when in order to achieve a good or eliminate a bad design solution" or in other words "what to do when" so that the presence or absence of an instance of the feature class that defined goodness or badness could be achieved. The term "what to do" is characterized in terms of preference on a design decision. The term "when" is characterized as a *situation*, a potential pattern that can match a wide variety of future design contexts. When such a heuristic is available and matches the design context, the quality of the design solution that a design decision is expected to generate is now predictable directly using the features of the design context that were there before the design is

even generated. This makes the result of the concept re-representation directly applicable to the process of designing.

3.5 A MORE FORMAL DESCRIPTION

Using the previously described mapping between the proposed design computation process to the state-operator paradigm, we can now describe the designing and the learning process more formally, combining the levels of abstraction. A design generator recursively proposes *design decisions* (d_k) to transform initial partial design configuration (s_0) to generate a tree of designs, Figure 2(a).

The generation of each design, is followed by the evaluation of the design thus creating a new design context. The process as a whole, is thus a transformation of design context information ($s_i \rightarrow s_j$), Figures 2(a) and details of a single decision application are shown in Figure 4. The additional input required for learning is the description of a design *feature class* (f_j) and a flag to it identifying whether the feature class is bad (-) or good (+) (an evaluator to determine the quality of the design). Learning can take place only when instances of one of these evaluator feature classes is present in the newly generated design. The process of learning is essentially constructing the association between what are the conditions (c_i) under which the decision d_k should be preferred (+)/rejected (-) i.e. a rule $c_i \rightarrow$ *(+/-)* d_k where $c_i \subseteq s_i$.

c_i, is found by *backtracing* each pattern-matched variable of f_j from state s_j to its originating elements in state s_i through the trace of design experience, Figure 4. *Tracing* is defined as the process of maintaining a historical chain of instantiated patterns that were matched and replaced during the process of decision proposal, application and evaluation of design alternative. This output of trace is the design experience. Tracing is done when a design alternative is constructed using uninformed search i.e. without the aid of a heuristic preferring an operator. The process of *backtracing* a given instantiated pattern, f_j, is the process of traversing the historical chain of instantiated patterns from f_j in a reverse direction to its generation to find which patterns were used to generate it. In this case, it is c_i . Looking in a forward direction, c_i is one or more instantiated patterns in s_i, which were transformed by the design decision d_k to give rise to feature f_j that was subsequently evaluated in s_j to conclude about the quality of the design. The situation construction is denoted as $c_i \leftarrow f_j$, the reverse arrow signifying the backtracing operation. The process of learning from a single design generation and evaluation is termed in this paper as "single-step learning"[1].

[1] "Multi-step learning" is an extension of this method proposed by Nath (2000) where learning is along a longer, if not a complete solution path. But this is not within the scope of this paper and is briefly covered in a later section.

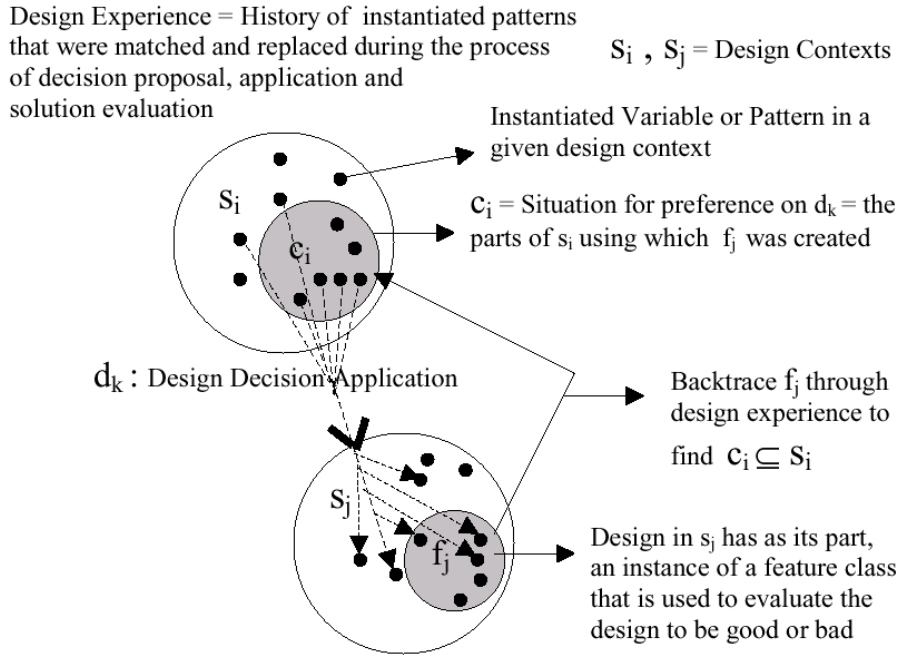


Figure 4. Learning from an instance of design generation and evaluation the knowledge "*If pattern c_i matches some context s_m then prefer(+)/avoid(-) d_k*"

Once c_i is obtained, the task is to turn it into a variable c_i (also called variablizing) so that it can pattern-match future design contexts. The process of variabilization is a direct consequence of using the chunking algorithm of SOAR (Laird et al 1986) and its implicit generalization strategy. This corresponds to the same variable identifier replaced by the same variable and different identifiers by different variables.

3.6 CONNECTION WITH SOAR

The main reason for using SOAR framework is that it allows a uniform representation space to be used for both learning and problem-solving. With other systems there may be the issue of mapping the representation space of learned knowledge to that of a representation space for problem-solving in order to use what has been learned.

SOAR uses a variant of EBL called chunking. Chunking results in rules known as chunks. The learning method described in the previous subsection is based on SOAR's chunking algorithm that has been specialized, reinterpreted and applied in the context of computational design. Learned heuristics map onto SOAR chunks. The principle of chunking in SOAR is: whenever there is problem-solving done within a subgoal, it can be bypassed

the next time because of a chunk (rule) that captures an association between the results of the subgoal and the features of the supergoal that led to the result in the subgoal. But chunking is an abstract domain independent learning mechanism, applying it in the context of generic design tasks requires commitment on several aspects that chunking does not commit to, e.g. the nature of subgoal generation, and nature of problem-solving in design subgoals. This is equivalent to the addition of one knowledge level that captures the generic tasks involved in computational design and maps them to the domain general knowledge level of SOAR. Nath (2000) describes these in more detail and proposes extensions and variations to this method. The successful integration of learning and problem solving is also a direct consequence of using the SOAR architecture, as SOAR treats the learned chunks in the same way as human encoded rules.

4. Application

The application, reported here, uses the ideas of tightly coupled learning and problem solving to generate architectural layouts on a site. The difference with other layout design applications is essentially the method used to achieve the end, i.e. the learning process automatically constructs heuristics that are profitably utilized by the search process. The architectural layout design problem is one where rooms are required to satisfy adjacency and cardinal direction constraints to fulfill their functions. For example, the "master-bedroom" must have morning sunlight, which implies "master-bedroom must face east" (requires-constraints) or a "bedroom" must not face (avoids-constraints) south because there is noise from the south. The adjacency and site constraints for the different rooms of the layout are depicted in Figure 5.

A rectilinear shape class is a compact abstraction that can be used to represent a set of possible rectilinear shapes. Each room when placed on the site is an instance of one of several allowable rectilinear shape classes, all of the same area. These shape classes for each room are shown in Figure 6. The area required for each of the rooms is specified as a part of design requirements.

4.1. SHAPE REPRESENTATION

A shape (instance) is represented using an anticlockwise ordered sequence of directed unit edges called facelets from a base point[2]. The perimeter of the shape is the number of facelets in the shape. A facelet is described by a head point and a tail point and the cardinal direction in which the head of the

[2] The shape representation bears some resemblance with the representation proposed by Rosenman (1996).

facelet points (represented in subsequent figures as arrows). A facelet also has information about which are its previous and next facelets. However, to allow for rotations of these shapes, the cardinal direction of each facelet of a shape is described by means of relations to the cardinal direction of the starting facelet of the shape. These relationships are straight (same), anticlockwise, clockwise, opposite, e.g. if the cardinal direction of the starting facelet is "north", and the relationship of a second facelet to the cardinal direction of the starting facelet is clockwise, then the second facelet will have a cardinal direction, "east".

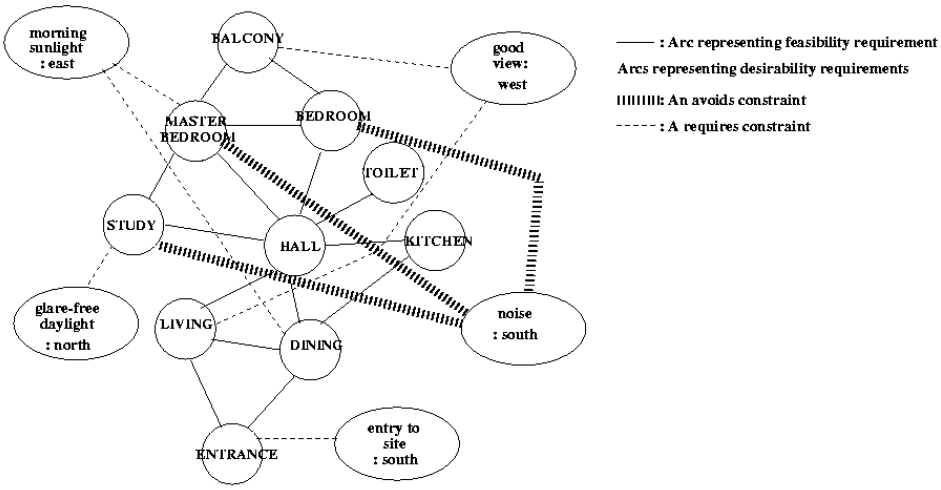


Figure 5. Adjacency and site constraints graph of a layout design problem

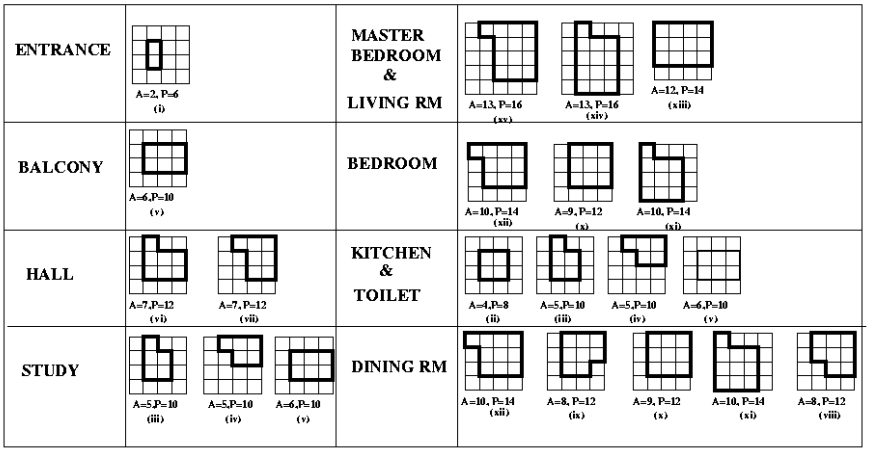
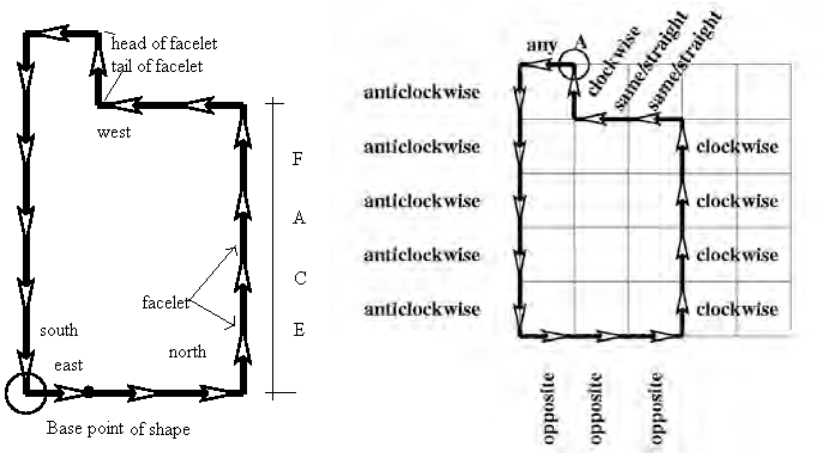


Figure 6. Set of allowable shape classes for various design parts (rooms)

Figure 7(b) is the pictorial representation of an abstract shape class {base-point, any -5*anticlockwise -3*opposite e-4*clockwise -2*straight -1*clockwise}. Its base point is circled and labeled A and the facelet that has its tail at A is the starting facelet. The directional move sequence in this representation is a shape generation plan, where each move (viz. "anticlockwise, opposite, clockwise, straight) is relative to the starting cardinal direction, "any", to which the initial facelet can be instantiated. Numerals in this representation represent the number of consecutive times of application of the same relative move. Thus the cardinal directions of non-starting facelets can be instantiated to 4 possible directions depending on the instantiation of "any" to north, south, east or west. This abstract representation with a given basepoint, suffices to encode 4 possible instantiations of the shape for a given starting basepoint. If the basepoint representation is also parametric the set of possible shapes, which belong to this class, are multiplied by the number of possible base points. Figure 7(a) show one instantiation of the shape class shown in Figure 7(b) with a given base point and the starting facelet pointing to west. Design evaluation knowledge and reasoning are on shape classes rather than their instantiations, so that the knowledge that it produces as a result of learning is more general.



(a) (b)

Figure 7. Elements of representation of (a) a shape, (b) a shape class

Adjacency between an already placed room A and B (being currently configured), is achieved by instantiating a given shape class such that B's starting facelet is equal in position (same end points as the joining facelet of A1) but opposite in the direction to any facelet of A. This is shown in Figure 8, and is the task of the design decisions. The combinatorics of the design

alternatives (and design decisions) for a given room are defined by the choices of different allowable shape classes, the different facelets of room A that can be used as the instantiation seed for the initial facelet of room B, and the different shape generation plans.

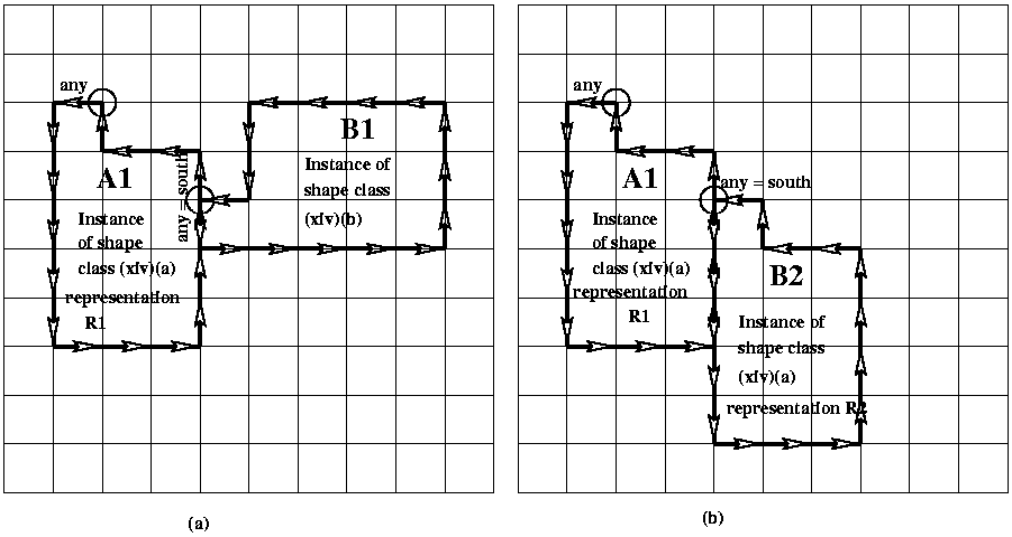


Figure 8. Adjacency of design parts: representation

4.2. DESIGN EVALUATION CRITERIA

One evaluation criterion that often governs good layout designs is “efficient usage of space”. To demonstrate the effectiveness of the ideas, let us assume that the concept of efficient usage of space is defined as the absence of three feature classes: (a) narrow deep (con)cavities (I-shaped and L-shaped) between two or more rooms, (b) unit holes between two or more rooms, (c) too much edge overlap between two given rooms. Positive and negative examples of what is implied by criteria (a) and (b) are shown in Figure 9. Two of the rules that identify typical unit holes and deep concavities are shown in Figure 10. Criterion (c) is formulated as follows: for two given rooms, the number of overlapping facelets of each of these rooms should be <= 2. The “Hall” being the one room with limited perimeter, which is adjacent to most rooms, cannot afford to have its perimeter edges consumed by a single room to satisfy adjacency criteria, because when the last few rooms are being configured with respect to the hall, there may not be enough perimeter length of the “Hall” left to satisfy the adjacency requirement of these later rooms. In that case, the program goes into costly backtracking operations.

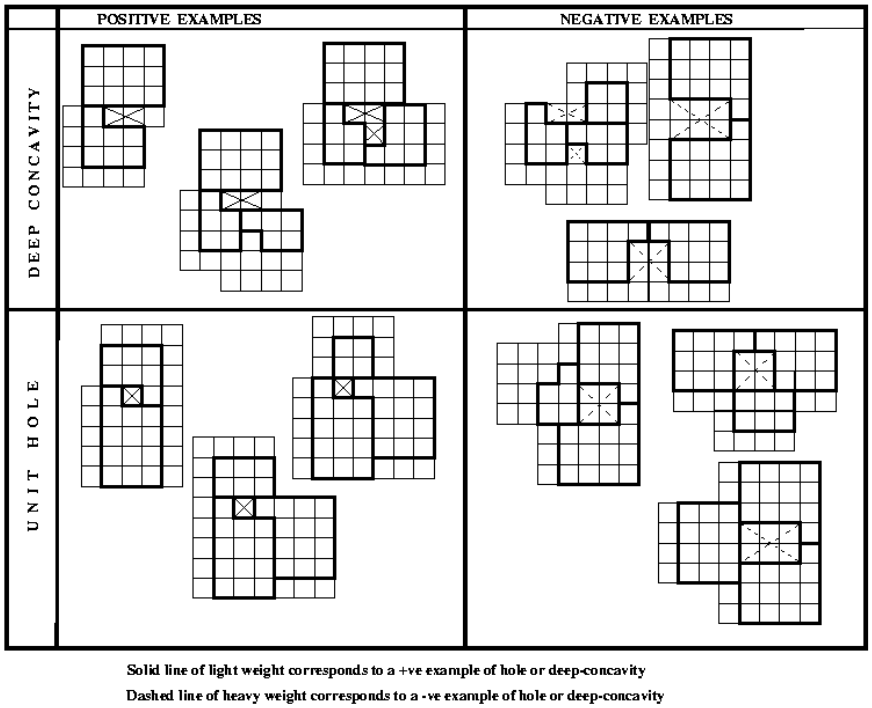
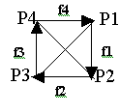
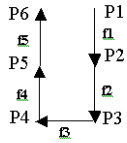


Figure 9. Features that make layouts undesirable: deep (con)cavities and holes between rooms.



IF facelet f1 has head P2 and tail P1
AND facelet f2 has head P3 and tail P2
AND facelet f3 has head P4 and tail P3
AND facelet f4 has head P1 and tail P4
AND f1 ≠ f2 ≠ f3 ≠ f4
AND P2 is in a direction D with respect to P1
AND P3 is in a direction clockwise(D) with respect to P2
AND P4 is in a direction opposite(D) with respect to P3
AND P1 is in a direction anticlockwise(D) with respect to P4
THEN
Unit Hole is defined by points P1, P2, P3, P4



IF facelet f1 has head P2 and tail P1
AND facelet f2 has head P3 and tail P2
AND facelet f3 has head P3 and tail P4
AND facelet f4 has head P4 and tail P5
AND facelet f5 has head P6 and tail P5
AND f1 ≠ f2 ≠ f3 ≠ f4 ≠ f5
AND P2 is in a direction D with respect to P1
AND P3 is in a direction D with respect to P2
AND P4 is in a direction clockwise(D) with respect to P3
AND P5 is in a direction opposite(D) with respect to P4
AND P6 is in a direction opposite(D) with respect to P5
AND P6 is in a direction clockwise(D) with respect to P1
THEN
Deep concavity is defined by points P1, P3, P4, P6

Figure 10. Typical rules that define one case of deep (con)cavities and unit holes.

These design evaluator features (labeled as bad) are described by reasoning about the facelets between rooms, Figure 10. In this example, the

task of learning is to re-represent the concept of unit holes, deep concavities and excess edge overlap in terms of a subset of patterns in the previous design context from which these features were generated, so that for the future those patterns could be used as a predictor of these types of bad features and thus be used to avoid decisions that will produce such features.

5. Experimentation Strategy

The following strategy was adopted to evaluate the ability of the proposed model to learn knowledge in order to avoid infeasible designs.

(a) Generate two-roomed layout configurations (set S1) with learning turned off (assuming a given position of the first room to be configured), say we have *n1* solutions
(b) Inspect and mark the infeasible solutions (set S2), with holes and deep concavities, say these are *n2* in number
(c) Then for the same position of the first room, we turn learning on and generate all possible configurations of the second room. If learning is successful then the new set of designs (S3= S1-S2) should not have any holes or deep concavities and the number of designs produced should be n1- n2. The check n1- n2 ensures that learned knowledge does not misclassify any feasible designs as infeasible.

Next, we add an extra evaluator feature: the excess edge overlap criterion. Steps (a) and (b) are performed again on the set S3 and the infeasible solutions (S4) based on the new features were identified. This should result in additional learned knowledge for the new evaluator feature, while restricting the new solution set to S5=S3-S4.

Next the generality of learned knowledge and hence its transferability is examined at both an intra-problem level and at an inter-problem level. At the intra-problem level, is the knowledge learned from laying out two rooms sufficiently abstract so that it can be transferred to other parts of the search tree? To determine this the task of configuring the third room from the configurations in S4 was executed. The objective is to test, whether knowledge learned from the experience of configuring two-roomed layout configurations would be transferable between two-roomed configurations, where the shape classes were different. At the inter-problem level, a different starting design configuration was chosen and examined as to whether the learned knowledge applied.

6. Results

The solution set of Step (a) in the previous section with learning turned off for the features “deep (con)cavities and holes” is shown in Figure 11. There are 81 solutions, which comprise the set S1. The 9 boxed solutions that identify the designs, which contain deep concavities and holes, is the set S2. Figure 12 shows the set S3 (=S1-S2) that comprises exactly 81-9 = 72

solutions that do not contain deep concavities or holes. Thus the learned heuristics do not seem to misclassify designs. Next, the additional feature of minimizing perimeter edge overlap was introduced. Applying the same experimental step, the designs in set S3 with this feature are box marked; there are 31 boxed solutions and they represent set S4. Figure 13 illustrates the solution set S5 (=S3-S4), where not a single solution was produced that had “too-much-edge overlap” or “holes and deep concavities. This set also contains exactly 72-31 = 41 solutions, which again shows that the learned heuristics correctly identify bad designs.

Figure 14 shows that what was learned from the configuration of 2 rooms could be applied to layouts with 3 rooms and the same principles that were learned were applied successfully. This is explained in detail in a later section. Figures 15 and 16 show that what was learned was transferable to other design problems that use the same representations, but where the initial configuration of the “Hall” was different. No deep concavities or holes, no excess edge overlap between the “Hall” and other rooms exist. Figure 17 shows how using the learned design knowledge, complete layouts produced demonstrate efficient packing of the rooms considering the protrusions and indentations in other rooms.

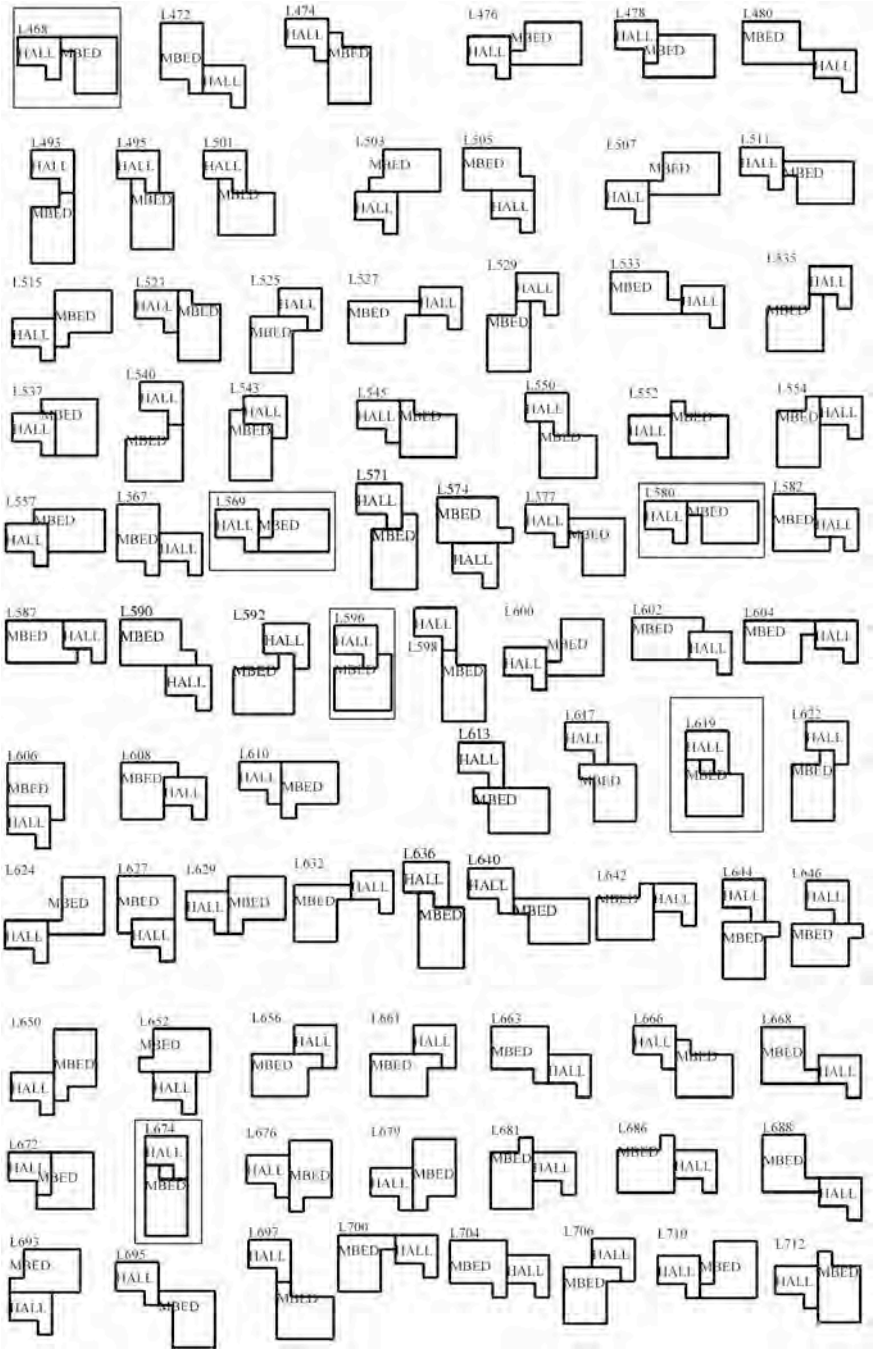


Figure 11. Solution space of 81, 2-roomed layouts with learning turned off. The 9 boxed solutions show deep concavities and holes. Correct learning should not misclassify designs; this implies producing exactly 72 solutions, at the next iteration.

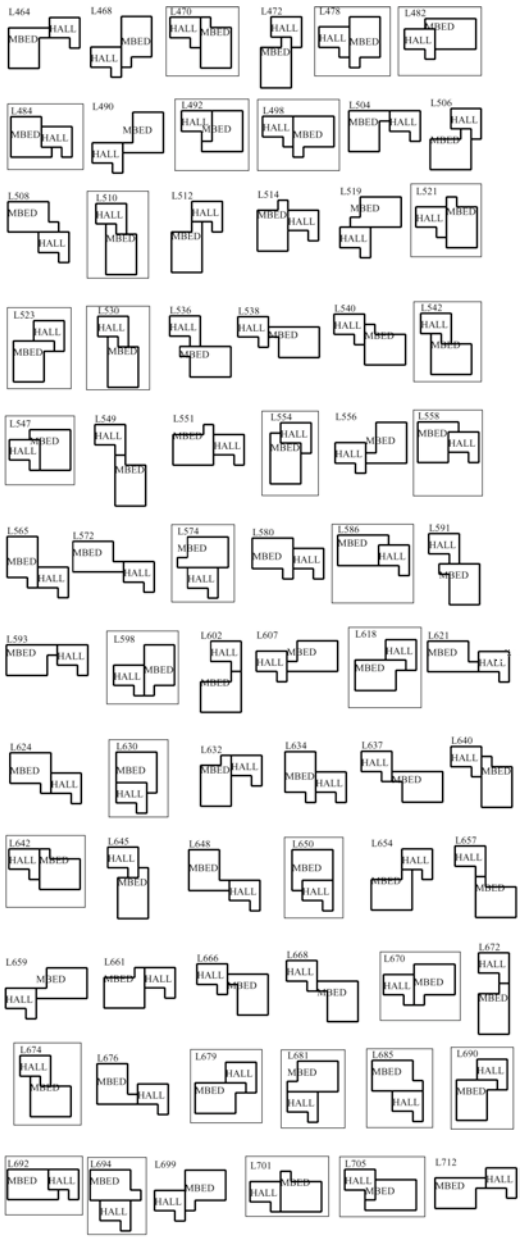


Figure 12. Solution space of exactly 81-9=72 layouts after learning heuristics about how to avoid deep concavities and unit holes. Not a single solution contains holes or deep concavities. The 31 boxed solutions show the "excess-edge-overlap" feature with the HALL. Correct learning implies producing exactly 72-31=41 solutions in the next iteration.

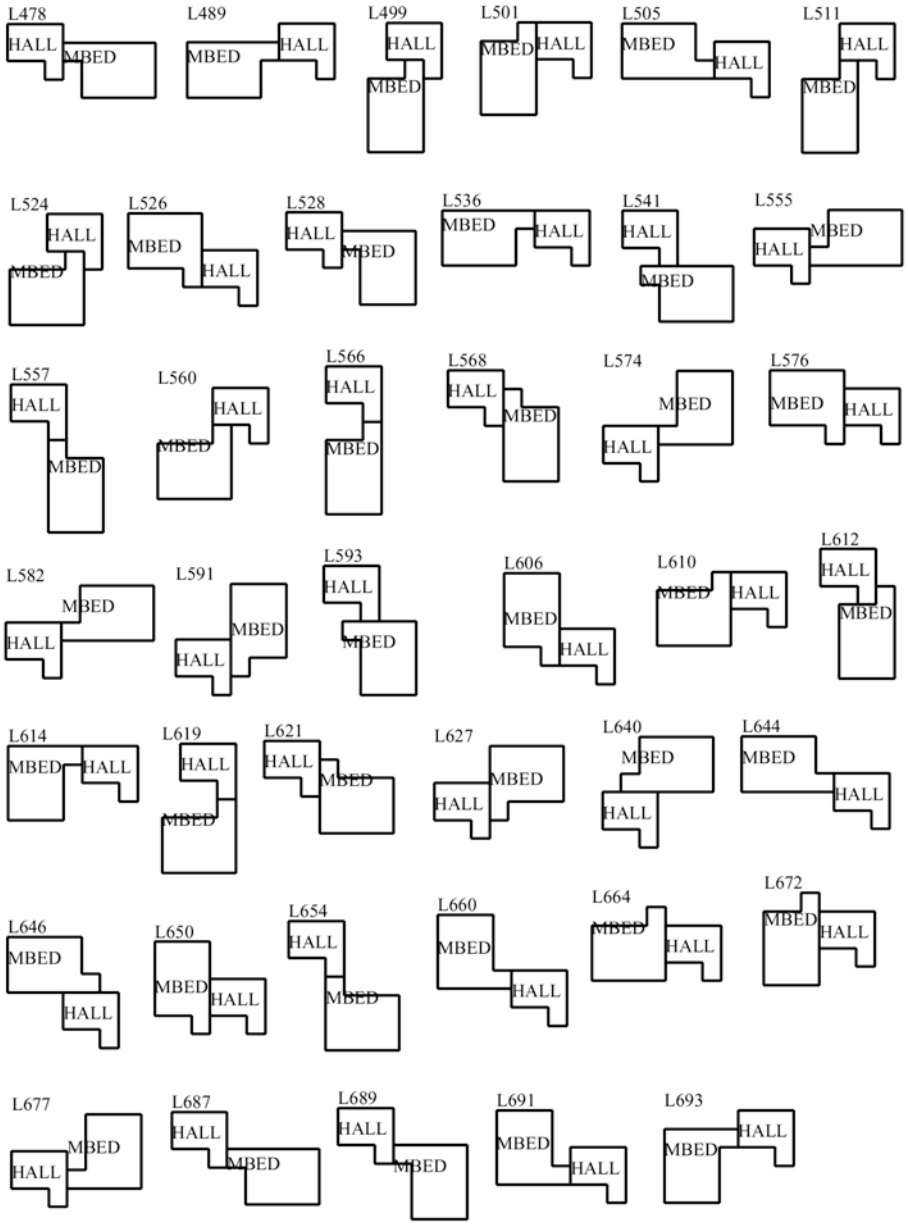


Figure 13. Solution space of exactly 72-31=41 layouts after using learned knowledge about how to avoid unit holes, deep concavities and excess edge overlaps with the HALL. Note not a single solution produced contains these features.

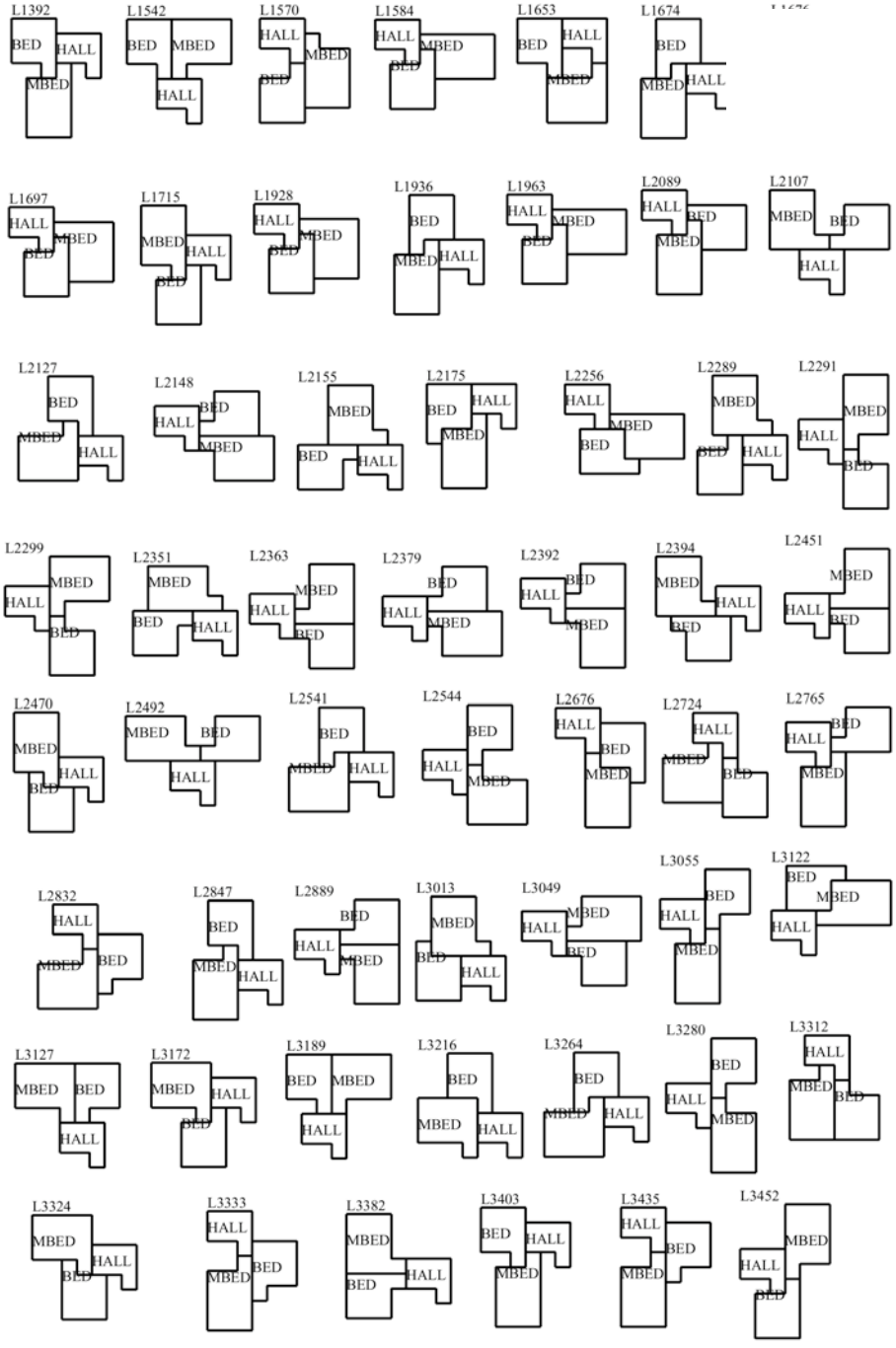


Figure 14. Heuristics learned while generating solutions shown in Figures 11, 12 and 13 were general enough to eliminate such features even for 3-roomed designs.

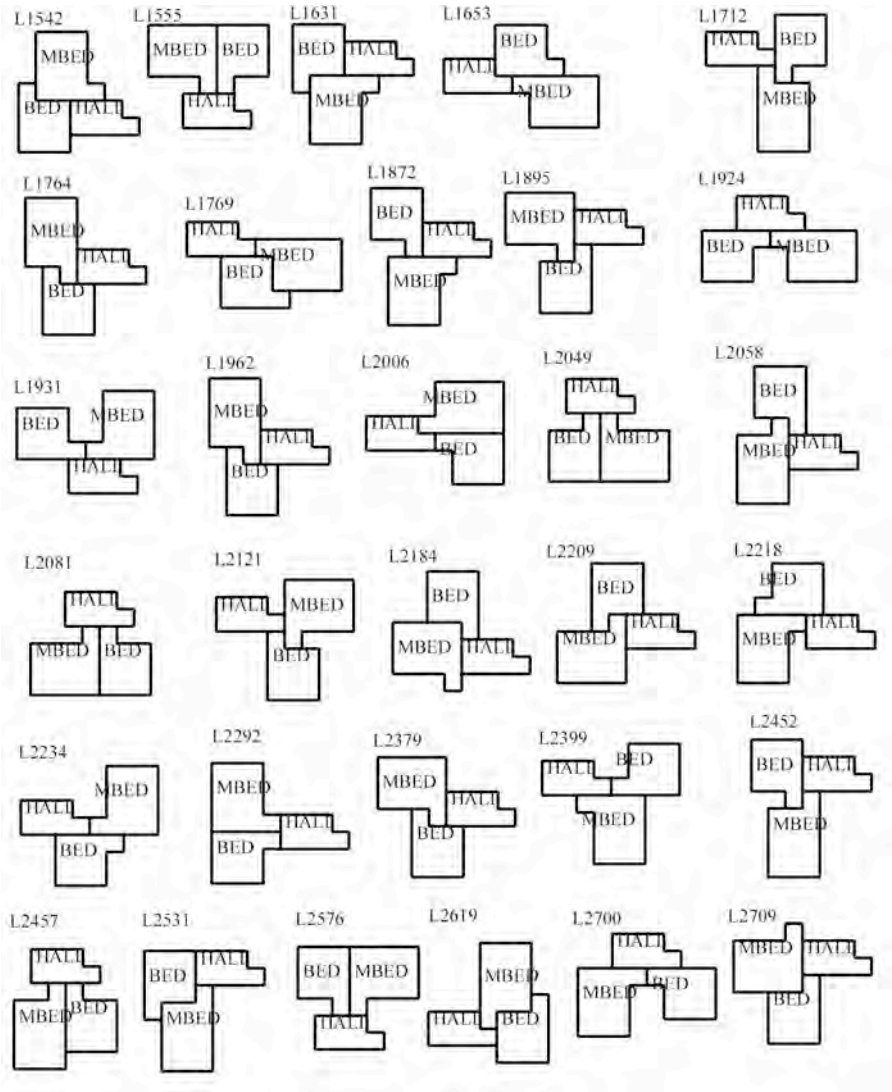


Figure 15. Heuristics learned while generating solutions shown in Figures 11, 12 and 13 were general enough to eliminate undesirable features in 3-roomed designs, even when the initial configuration of the first room was different.

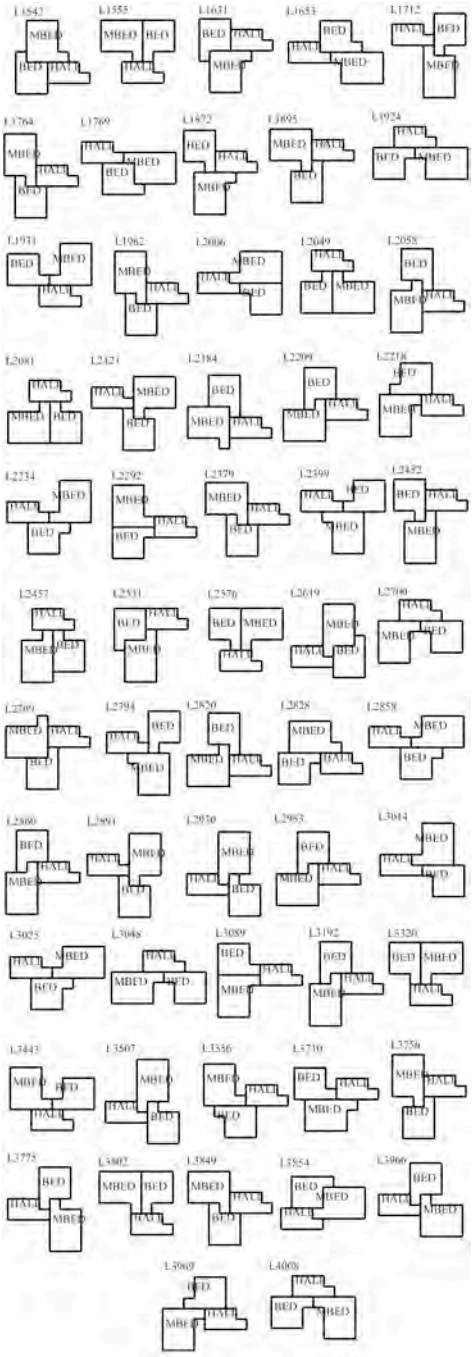


Figure 16. Another instance of the application of learned knowledge to generate successful 3-roomed solutions with a different initial shape of the HALL. No undesirable features are present.

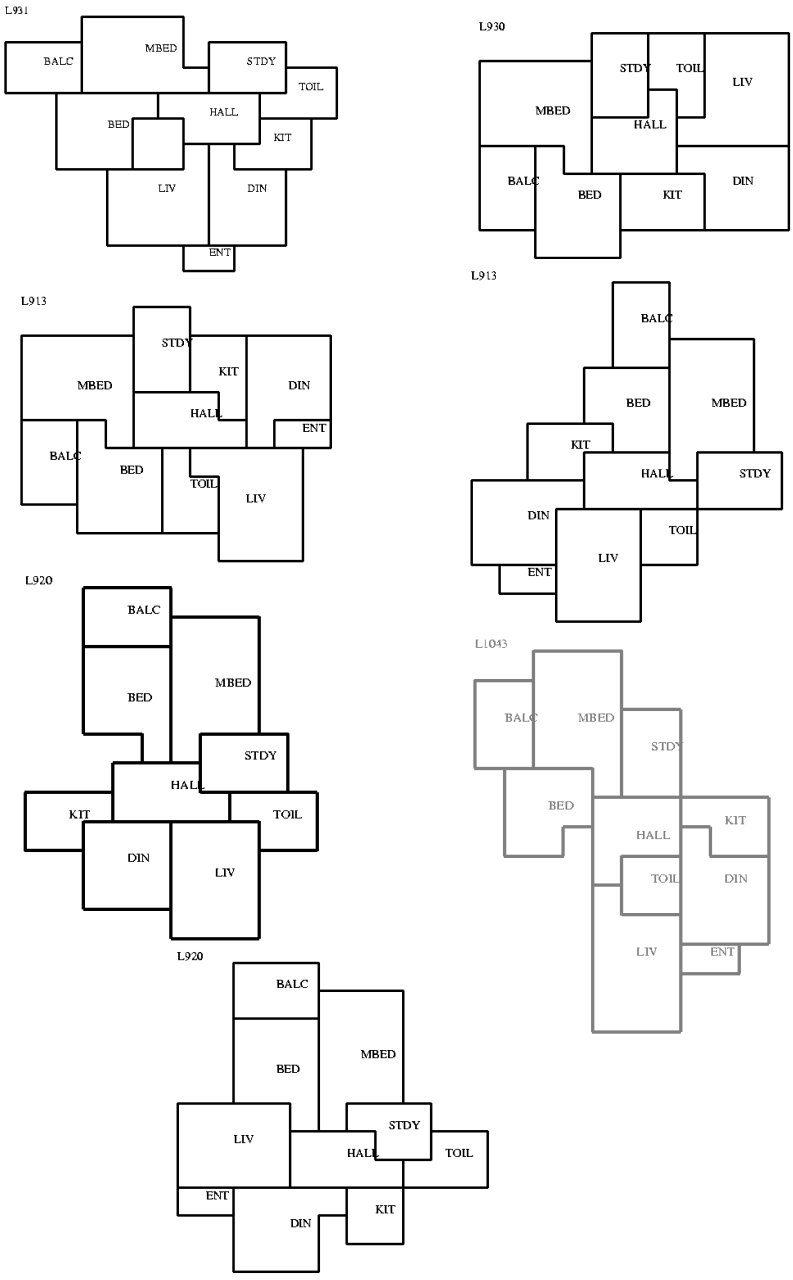


Figure 17. Complete layout configurations generated with learned knowledge. Note that the top leftmost solution has an internal hole but it is not a unit hole .The learned heuristics should only prevent unit hole generation, not larger holes. Thus the heuristics have allowed the desired generation successfully.

7. Learned knowledge and generality

The main reason for effective knowledge transfer from a 2-room situation to a 3-room situation, was that the situation part of the heuristics reasoned about the relations between partial shape generation plans of the room being configured and the facelets of other rooms and not the individual cardinal directions. The rooms to which shape generation plans or existing facelets were variabilized. The abstract representation of the cardinal directions captured relationships between the directions instead of a commitment on the actual directions. Even though the global layout in which the learning was applied consisted of 3 rooms, often 2 rooms were involved whose facelets formed a unit hole or a deep concavity. In such cases, the learned knowledge could immediately apply. Figure 18 shows one of the many pieces of learned knowledge that avoids the generation of layouts with deep concavities. Figure 19 shows the chunk in SOAR for this. Similarly, Figure 20 shows one typical rule that avoids generation of unit holes. Figure 21 shows one of the chunks in SOAR for this.

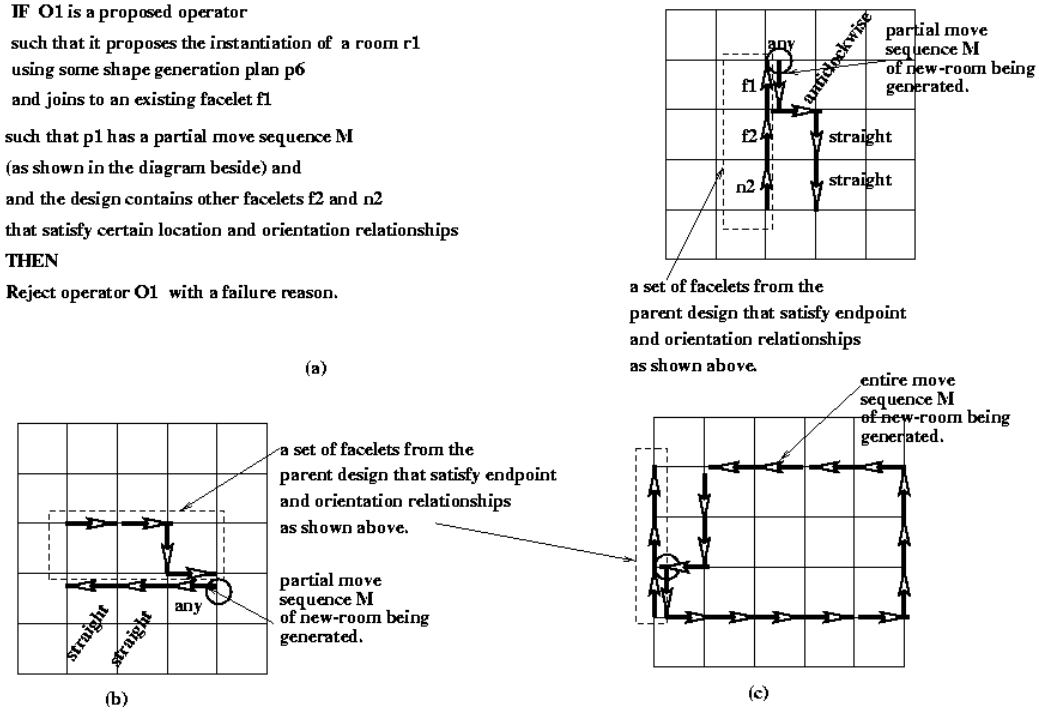


Figure 18. Three learned rules that prevent deep concavities in layouts

```
sp {chunk-2
:chunk
(state <s1> ^operator <o1> ^directions <d1> ^old-layout <l1>)
(<o1> ^type joining-facelet ^new-facelet-direction <n1> ^facelet <f1>
      ^alt <a2> ^plan <p6> ^adjacency-room <a7> ^room <r1>)
(<n1> ^straight <n1> ^anticlockwise <a1>)
(<d1> ^direction <n1>)
(<f1> ^points <p1> ^pt1 <p2> ^pt2 <p3>)
(<p1> ^clockwise <a1> ^opposite <n1>)
(<l1> ^room <a2>)
(<a2> ^facelet <f2>)
(<f2> ^points <p1> ^next <n2> ^pt2 <p4> ^pt1 <p5>)
(<n2> ^points <p1> ^pt2 <p2>)
(<p2> ^adjacent <a3>)
(<a3> ^direction <a1> ^location <l2>)
(<p5> ^adjacent <a4>)
(<a4> ^direction <a1> ^location <l4>)
(<l2> ^adjacent <a5>)
(<a5> ^direction <n1> ^location <l3>)
(<l3> ^adjacent <a6>)
(<a6> ^direction <n1> ^location <l4>)
(<p6> ^move <m1>)
(<m1> ^previous nil ^next <n3>)
(<n3> ^direction anticlockwise ^next <n4>)
(<n4> ^direction straight ^next <n5>)
(<n5> ^direction straight)
-->
(<s1> ^operator <o1> -)}
```

Figure 19. One of the learned rules in SOAR syntax that prevent deep concavity generation

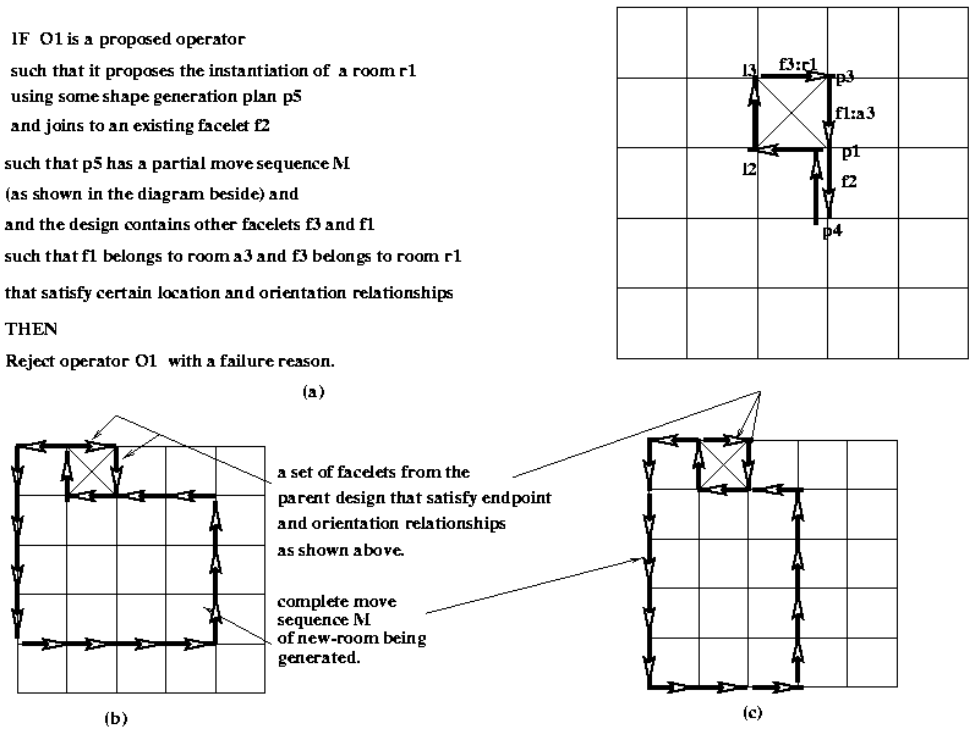


Figure 20. Three learned rules that prevent unit holes in layouts

```
sp {chunk-51
:chunk
(state <s1> ^operator <o1> ^directions <d1> ^layout <l1>)
(<o1> ^type joining-facelet ^new-facelet-direction <n1>
      ^adjacency-room <a2> ^alt <a3> ^facelet <f2> ^plan <p5>
      ^room { <r3> <> <a2> <> <r2> })
(<n1> ^straight <n1> ^anticlockwise <a1> ^clockwise <c1>)
(<a1> ^clockwise <n1>)
(<d1> ^direction <n1>)
(<a3> ^room-id <a2> ^facelet <f1>)
(<l1> ^room <a3> ^room <r1>)
(<f1> ^pt2 <p1> ^pt1 <p3>)
(<f2> ^pt1 <p1> ^points <p2> ^pt2 <p4>)
(<p1> ^adjacent <a4>)
(<a4> ^direction <a1> ^location <l2>)
(<p2> ^opposite <n1> ^clockwise <a1>)
(<l2> ^adjacent <a5>)
(<a5> ^direction <n1> ^location <l3>)
(<l3> ^adjacent <a6>)
(<a6> ^direction <c1> ^location <p3>)
(<p3> ^adjacent <a7>)
(<a7> ^location <p1> ^direction <p2>)
(<p5> ^move <m1>)
(<m1> ^previous nil ^next <n2>)
(<n2> ^direction anticlockwise ^next <n3>)
(<n3> ^direction straight)
(<r1> ^facelet <f3> ^room-id { <r2> <> <a2>
})
(<f3> ^pt1 <l3> ^pt2 <p3>)
-->
(<s1> ^operator <o1> -)}
```

Figure 21. One of the learned rules in SOAR syntax that prevent unit hole generation

Figure 22 shows a richer knowledge set: learned heuristics that avoid the design generation to overstep site boundaries.

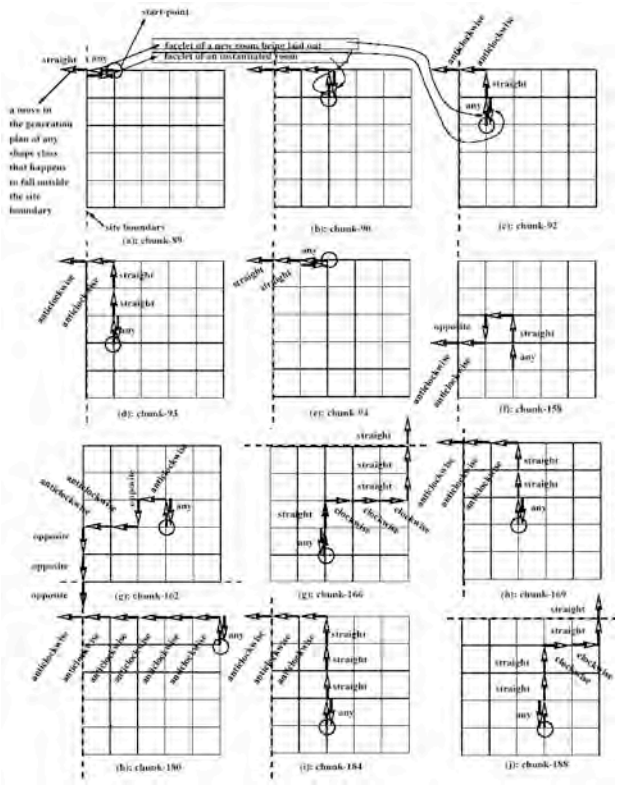


Figure 22. Learned heuristics that avoid the design generation that oversteps site boundaries.

Figures 23 and 24 show how these patterns can match in different design contexts (different orientations and different shape generation plans) and allow reuse of the knowledge learned. When learned heuristics like these apply it eliminates infeasible design solutions. For example Figure 20 shows how the same situation learned using the feature of "deep concavity" recurs for different design contexts. These are some layouts picked for explanation from Figure 11 for illustration. A similar recurrence of situations can be found in the marked solutions in Figure 12. Some of the heuristics learned were also more specific than the ones described above. Here the entire shape generation plans formed part of the situations of these rules. Hence, these rules could only apply to preventing deep concavities or holes with specific shape classes; the reason is that it is the later facelets in the shape generation plans that define the deep concavity or the hole.

If more than 2 rooms were involved in forming a deep concavity or a hole then the situation part of the learned heuristic from the 2 room situation may not match and hence learned knowledge may not be applicable. Every evaluated design solution is an opportunity to either learn knowledge or use learned knowledge. In such cases instead of using previously learned heuristics, new heuristics would be learned and would the heuristics would cover holes or deep concavities that involved the facelets of more than 2 rooms. Similarly these heuristics would be useful in eliminating potentially space-wasting design decisions. There is a wide variety of ways in which the concepts of space wastage occur in layout design, especially with the non-rectangular nature of the rooms.

In total around 560 rules of decision preference in different design contexts were learned although the space of design solutions was not completely explored. If it was completely explored then adding these heuristics would have resulted in no search, resulting in complete knowledge based activity guided by situation-directed prediction of the quality of a decision.

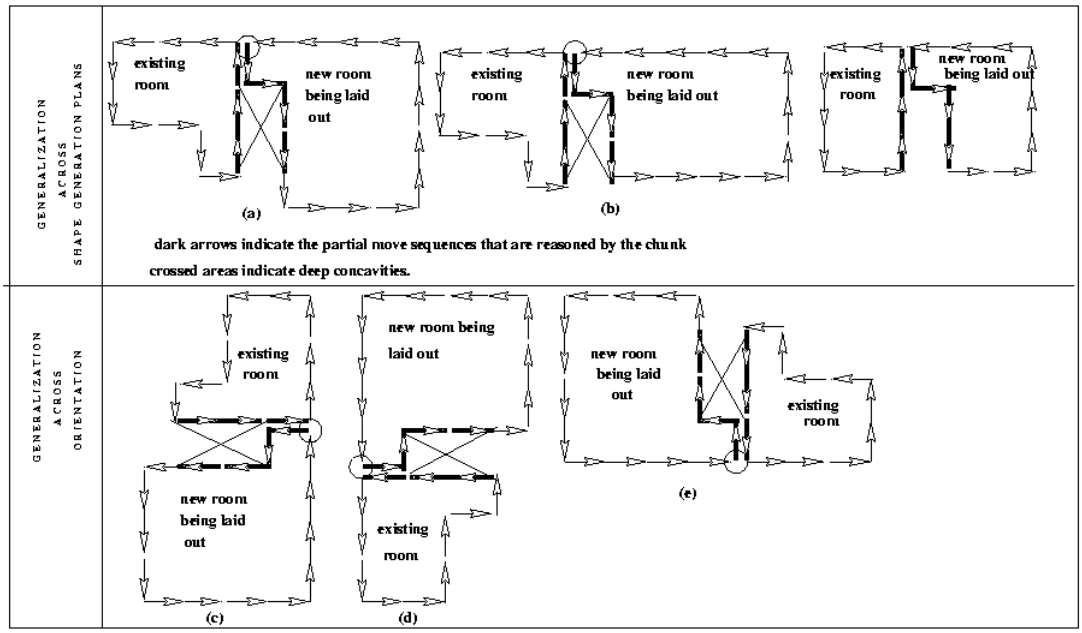


Figure 23. How the learned knowledge as in Figure 15 matches various designs

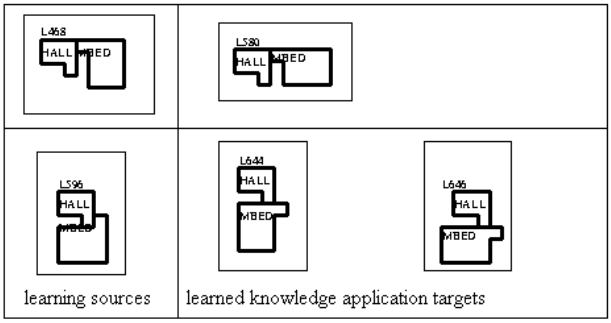


Figure 24. Left column shows two instances of partial layout (with deep concavities) from Figure 10 using which learning was done. The right column shows how learned situations match in other instances of layouts with deep concavities in Figure 12.

9. Evaluation

9.1 STRENGTHS OF THE METHOD

There are mnay advantages in using this method. First, learning is dynamically driven by the requirements of the designing process to produce

better solutions rather being than a passive process of classification of designs. This is the basis for a highly coupled interaction between learning and designing processes that results in a self-improving design search procedure. Second, the uniform representation space for learning and designing borrowed from the SOAR framework allows seamless transfer in both directions. Third, the learning method and the knowledge learned aid the design computation process in reasoning about regions of the search space rather than specific points. This is possible because the situations embedded as antecedents of learned heuristics encode in their patterns complex conditions that correspond to regions of the solution space. In addition the knowledge engineer can control the level of generality desired in these learned heuristics. The more abstract, the evaluation reasoning and the representation of the problem, the more general and applicable the learned knowledge will be. Fourth, sets of inappropriate solutions can be eliminated using a single learning session thus enabling rapid exploration of the solution space. This of course has a bearing on the level of abstraction employed in representation and reasoning of the domain. Fifth, costly processes involved in domain-oriented knowledge engineering like elicitation of tacit knowledge and subsequent encoding of heuristics are eliminated. Only those heuristics are constructed by the program that are relevant to goal of satisfying the design requirements under consideration.

The experimentation strategy and the results showed that the heuristics did not fail, the situations embedded in the heuristics did not estimate a bad design solution as good, neither did it classify a good design solution as bad. The process uses an approach similar to explanation-based learning and performs a deductive compilation of the domain theory (design evaluation rules) based on operational criteria. This implies that the learned heuristics could only fail if the knowledge engineer incorrectly encodes the domain theories employed in evaluative reasoning. It is therefore important that the user of this method has precise definitions of what exactly constitutes a bad design and what is a good design. Because of this advantage, the embedded situations in the heuristics do not need any statistical evaluation to determine their success or failure. At the knowledge level, a correctly encoded evaluative domain theory will result in a correct concept re-representation i.e. correct heuristics.

Finally the unlike many other works on learning, the learned knowledge can be inspected to have an understanding of the kind of learned knowledge generated and thus have greater control over it. The need for such inspection could be not only for the human designer or the researcher to experiment with their representations to best utilize a tool such as this to suit their goals, but also to understand the nature of tacit hard-to-explain knowledge that is captured as a result of learning. The tacit nature of this learned knowledge was explored in another work (Nath, 2003).

It is these advantages that should encourage computer aided design system developers or researchers to incorporate such an approach in the design of their tools for designing. Practicing designers, who would be users of such a tool, would also gain these advantages.

9.2 DISTINCTION WITH PAST WORK

The work that is presented here is distinct from learning from designs, on which there is a plethora of research. This work demonstrates learning and designing co-occurring in a coupled design process. Manfaat et al (1996) have reported generalizations from a set of spatial layout designs. The form of learning reported here can also be viewed as one form of knowledge compilation (Mostow, 1990). Knowledge compilation has been applied by Brown and his colleagues (Brown, 1996; Chabot and Brown, 1994: Liu and Brown, 1991; Sloan and Brown, 1988; Brown and Spillane, 1991) to learn design process knowledge, but these works were not in the spirit of the tight coupling between a learning and a designing process. Braudaway and Tong (1989) introduced a knowledge compilation based system called RICK which describes how an inefficient generate and test problem-solver for a parametric building layout design problem, can adapt itself to the problem specific constraints. No heuristics are learned, here, instead a more efficient LISP program (constrained generators) for solving the particular problem is synthesized. Voigt and Tong (1989) describe an extension of RICK called MENDER, which reformulates the global design constraints as an evaluation function and derives ``hill climbing patchers'' through knowledge compilation.

If one views this work as adaptive search, i.e. tailoring a generic generative mechanism to produce desirable designs, there is research by Cagan and his colleagues (Cagan and Mitchell, 1993; Reddy and Cagan, 1995; Schmidt and Cagan, 1998; Shea and Cagan, 1997), who apply simulated annealing to optimize grammatical design generation. In a similar stochastic spirit, there has been some recent applications by Vale and Shea (2003a; 2003b), which attempt at distinguishing good sequences of design generation operators from the bad using search experience. The intention is to use scores on these sequences as strategies that allow good sequences to be preferred over the bad in design generation. Machine learning here is of the incremental parameter tuning variety (Samuel, 1959), where the parameter is a score that is calculated on the basis of changes in objective function over grammar rule sequences weighted by the probabilities assigned to sequences based on past experience and future utility. However, the relation between the problem conditions and when a sequence could be useful is not explicitly modeled and it is not clear whether the probabilistic weights capture such a relationship implicitly.

Approaches to learning heuristics in design using the chunking method of SOAR has been reported by Modi et al. (1995a, 1995b) while using SOAR as a design problem-solver in the design of chemical engineering distillation sequences. However, the focus of that work was not on learning heuristics for preferring good solutions to inferior ones, but on heuristics for choosing evaluation functions from a predefined set. The research presented in the current paper treats learning design process knowledge more generally.

9.3 WEAKNESSES OF THE METHOD

The first weakness of the method is the inability to handle negated reasoning in its evaluative domain theory. This is more a consequence of the use of chunking algorithm of SOAR than the approach itself. The primary difference between chunking and EBL is that the backtracing of the concept in EBL takes place using the symbolic structure of the rules, rather than the instantiated rules. When instantiated patterns are used as the source for backtracing, the design experience as a result of tracing can account for the presence of instantiated patterns but cannot account for the absence of instantiated patterns. Hence the resulting heuristics are likely to be overgeneral. It is in such situations that the heuristics may fail, inspite of a correct domain theory. But this problem can be overcome by using another analytical learning algorithm like the traditional EBL, which uses backtracing through uninstantiated patterns.

As the evaluative reasoning tends to get more complex with many cases of applicability and increasing number of reasoning chains, it can often seem like new strategy knowledge is generated. However, at the theoretical level, there no new knowledge (outside the bounds of what was given) is generated as a result of learning. All heuristic knowledge is within the deductive closure of the knowledge that was encoded a priori, although to the human it may seem like new knowledge.

9.4 SCALABILITY ISSUES

Analytical learning from experience is often designed to incrementally increase the performance of a search mechanism. So the learning mechanism in itself is a step towards future scalability in terms of efficiency of exploration of a space. The time taken to learn is not really the issue for scalability, whereas the ability to match learned knowledge to a future design context in order to apply it to some future design context is. In other words, scalability is more of an issue at the applicability time of the learned knowledge rather than at learning time.

At the knowledge level, the scalability of a heuristic at application time, in isolation, is related to the complexity of the situation that is embedded in the heuristic. The complexity of the situation depends on the number of unit

representational entities (object-attribute-value triplets) and shared constraining interrelationships that have to be matched to a given design context. Sometimes such a heuristic with a large set of conditionals and many shared relationships could be very expensive to match with a large data set encapsulated in a design context and as a consequence that the system suffers a slowdown (Minton, 1988) after learning, contributing to what is called the *utility problem*. In general the utility problem occurs for all systems that learn rules from experience. The main reason for such a slowdown, especially when production systems like SOAR are used, can be understood at the internal algorithmic level rather than at the design knowledge level. For SOAR, the greatest amount of computational resources are used in matching rules and the knowledge search (Tambe, 1991) for which rules apply to the given context. This is significant even though SOAR uses an efficient production match algorithm like RETE (Forgy, 1982) and rule firing is simulated to be in parallel. SOAR's chunking algorithm is well known to reduce the number of steps to solve a subproblem the next time but when the utility problem occurs, the cost of matching the learned chunks becomes expensive as a result of the production match algorithm that offsets the benefit gained from the reduction in the number of problem solving steps. Ideally, the cost of using the learned rules should always be bounded by the cost of the problem-solving episode from which they are learned (Kim and Rosenbloom, 1996). The current solution is to design representations for data and reasoning in such a way that a larger number of cheaper chunks with smaller patterns is generated. A related but different reason that could also contribute to the utility problem is what is known as the average growth effect (Doorenbros, Tambe and Newell, 1992) of the knowledge base due to heuristics being learned continuously. Interaction across heuristics slow down the system even though the heuristics are not themselves expensive. Solutions to the average growth effect have already been proposed (Doorenbros, Tambe and Newell, 1992; Doorenbros, 1993).

11. Conclusions

This paper has presented a method of learning while designing that utilises design experience and design situations (Gero, 1998. It uses SOAR as its underlying framework. The method has been applied on an architectural design problem where learning and designing are tightly coupled. The main strength of this method is "intelligent control" which is manifested as the automatic extraction, from experience, of design problem conditions under which certain generation strategies are useful and the use of these heuristics to drive future design generation towards requirements. In the past, various design computing researchers (Akin, 1990; Coyne, 1988; Mitchell, 1996;

Muller and Pasman, 1996; Smithers et.al., 1990; Vancza, 1991) expressed control mechanisms as an essential need in design search processes, be it in a computational context or otherwise; but there has been a paucity of design research in this area.

The learning principle could be extended over a series of design transformations. In that case, if we have the following sequence of states representing a solution path, $s_1 \rightarrow s_2 \rightarrow s_3 \ldots. \rightarrow s_{n-1} \rightarrow s_n$ caused by application of decisions, $d_1 \rightarrow d_2 \rightarrow d_3 \ldots. \rightarrow d_{n-1} \rightarrow d_n$, and s_n containing feature $f_n(+)$, then we have the following conditions constructed from f_n as a result of backtracing: $c_1 \leftarrow c_2 \leftarrow c_3 \leftarrow \ldots\ldots c_{n-1} \leftarrow f_n$, where $c_i \subseteq s_i$. The process of learning, then, constructs the heuristic $c_i \rightarrow (+)d_k$ from a single design description in s_n. such that $c_i \subseteq s_i$. A series of rules can thus be learned from a single example. This forms a multi-step learning process and is reported in more detail in Nath (2000).

Acknowledgements

The research reported here was carried out at the Key Centre of Design Computing and Cognition, Sydney, Australia as a part of the first author's Ph.D. thesis that was supported by an AUSAID fellowship. The authors would also like to acknowledge the comments made by the reviewers of this AIEDAM special issue that improved both the content and presentation of the ideas in this paper.

References

Anderson, J.: 1983, *The Architecture of Cognition*. Harvard University Press, Cambridge, MA.

Braudaway, W and Tong, C: 1989, Automated synthesis of constrained generators, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA, pp. 583-589.

Brown, D: 1996, Knowledge compilation in routine design problem solving systems: Research abstract, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **10**(2): 137-138.

Brown, D and Spillane, M.: 1991, An experimental evaluation of some design knowledge compilation mechanisms, *in* J. S. Gero (ed.), *Artificial Intelligence in Design'91*, Butterworth Heinemann, Oxford, pp. 323-326.

Cagan, J, and Mitchell, WJ: 1993, Optimally directed shape generation by shape annealing. *Environment and Planning B: Planning and Design* **20**: 5-12.

Chabot, R and Brown, D: 1994, Knowledge compilation using constraint inheritance, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **8**(2): 125-142.

Carbonell, J. G., Knoblock, C. A., Minton, S.: 1991, PRODIGY: An integrated architecture for Prodigy, *in* K. VanLehn (ed.), *Architectures for Intelligence*, Lawrence Erlbaum Associates, Hillsdale, N.J, pp. 241-278.

Doorenbos, B., Tambe, M. and Newell, A.: 1992. Learning 10,000 chunks: What's it like out there? *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp.830–836.

Doorenbos, B. 1993. Matching 100,000 learned rules. *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp 290-296.

Forgy, C. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence*, **19**: 17-37.

Gero, J. S. (1998). Towards a model of designing which includes its situatedness, *in* H. Grabowski, S. Rude and G. Green (eds), *Universal Design Theory,* Shaker Verlag, Aachen, pp. 47-56

Kim, J. and Rosenbloom, P.: 1996. Learning efficient rules by maintaining the explanation structure. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp 763-770.

Laird, J, Newell, A and Rosenbloom, P: 1986, Chunking in SOAR: The anatomy of a general learning mechanism, *Machine learning* **1**: 11-46.

Laird, J, Newell, A and Rosenbloom, P: 1987, SOAR: An architecture for general intelligence, *Artificial Intelligence* **33**: 1-64.

Liu, J and Brown, D: 1991, Generating design decomposition knowledge for parametric design problems, *in* J. S. Gero (ed.), *Artificial Intelligence in Design'91*, Kluwer, Dodrecht, pp. 661-678.

Manfaat, D, Duffy, AHB and Lee, BS: Generalization of spatial layouts*, Workshop on Machine Learning in Design, Artificial Intelligence in Design'96*, Stanford University. CA.

Minton, S.: 1988, Quantitative results concerning the utility of explanation-based learning, *Proceedings of the Seventh National Conference on Artificial Intelligenc*e, pp. 564–569.

Modi, A, Newell, A, Steier, D and Westerberg, A: 1995(a), Building a chemical engineering process design system with SOAR-2: Learning issues, *Computers and Chemical Engineering* **19**(3): 345-361.

Modi, A., Newell, A, Steier, D and Westerberg, A: 1995(b), Building a chemical process design system with SOAR-1: Design issues, *Computers and Chemical Engineering* **19**(1), 75-89.

Mostow, J: 1990, Towards automated development of specialized algorithms for design synthesis: Knowledge compilation as an approach to computer aided design, *Research in Engineering Design,* **1**: 167-186.

Nath, G: 2000, A model of situation learning in design, *Ph.D. Thesis,* Department of Architectural and Design Science, University of Sydney.

Nath, G.: 2003, A computer program automatically acquiring some skills for a simple design problem, *in* Cross, N. and Edmonds, E.(eds.)*, Expertise in Design,* Creativity and Cognition Studios Press, Sydney*,* pp. 323-339.

Newell, A: 1990, *Unified Theories of Cognition*, Harvard University Press, Cambridge, MA.

Rosenbloom, P and Laird, J: 1986, Mapping explanation-based generalization onto SOAR, *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI Press, Los Altos, CA, pp. 561-567.

Reddy, G and Cagan, J: 1995, An improved shape annealing algorithm for truss topology generation, *ASME Journal of Mechanical Design,* **117**(2): 315-321.

Rosenman, M : 1996, The generation of form using an evolutionary approach, *in* J. S. Gero and F. Sudweeks (eds), *Artificial Intelligence in Design'96*, Kluwer Academic, Dodrecht, pp. 643-662.

Schmidt, LC, and Cagan, J: 1998, Optimal configuration design: An integrated approach using grammars, *ASME Journal of Mechanical Design,* **120**(1): 2-9.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers, *IBM Journal of Research and Development,* **3**: 210-229.

Shea, K, and Cagan, J: 1997, Innovative dome design: Applying geodesic patterns with shape annealing*, Artificial Intelligence in Engineering Design, Analysis and Manufacturing,* **11**: 379-394.

Sloan, W and Brown, D: 1988, Adjusting constraints in routine design knowledge, *Workshop on AI in design: Proceedings of the Seventh National Conference on Artificial Intelligence*, AAAI press, Menlo Park, CA.

Tambe, M. and Newell, A.: 1988, Some chunks are expensive, In J. E. Laird(ed), *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufmann, pp. 451-458.

Tambe, M. and Rosenbloom, PS.: 1989, Eliminating Expensive Chunks by Restricting Expressiveness, *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pp. 731-737.

Tambe, M.: 1991, Eliminating conbinatorics from production match. *PhD thesis*, Carnegie-Mellon University.

Vale, CAW and Shea, K.: 2003a, A machine learning-based approach to accelerating computational design synthesis, *Proceedings of the Fourteenth International Conference on Engineering Design(ICED 03)*, Stockholm, pp ??-??.

Vale, CAW and Shea, K.: 2003b, Learning Intelligent Modification Strategies in Design Synthesis, *Proceedings of the AAAI Spring Symposium on Computational Synthesis*, Palo Alto, CA., pp. 247-254.

Voigt, K and Tong, C: 1989, Automating the construction of patchers that satisfy global constraints, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA, pp. 1446-1452.

This is a copy of the paper: Nath, G and Gero, JS (2004) Learning while designing, *AIEDAM* **18** (4): 315-341.