COMPARATIVE ANALYSIS OF ROOM GENERATION METHODS USING RULE LANGUAGE-BASED EVALUATION IN BIM

Christoph Sydora and Eleni Stroulia
Department of Computing Science, University of Alberta, Canada
{csydora, stroulia}@ualberta.ca

Abstract

Automated layout generation can improve interior designs by suggesting constraint-compliant design alternatives. Persistent issues relate to intuitive and explainable constraint formulation and efficiency of layout search that result in highly compliant, diverse alternatives. This paper proposes a domain-specific language for rule representation and evaluation, along with a continuous layout-configuration search, Jump Search. The proposed method is compared against a previous grid-based layout search method and a Simulated Annealing approach, under rule-based evaluation budgets for different room-type scenarios. Our experimental results demonstrate that Jump Search is able to generate higher-quality layouts more efficiently, while also exploring a larger variety of layouts.

Introduction

Automation offers the potential for significant improvements in building design and construction, from eliminating repetitive routine tasks, such as stud placement, to automatically generating high-quality building designs. The last stage in design is the arrangement of furnishings in the available interior spaces of the building, a task that arguably has the largest impact on the occupants, as a poor arrangement can lead to inaccessible objects, have negative ergonomic effects on occupants, and generally make a space unpleasant that could otherwise be better utilized. Because of the variety of furnishing types and the complexity of the relevant ergonomic constraints and aesthetic preferences, the potential placement options and arrangement configurations are vast. Therefore, automatic layout design could greatly benefit individuals who might lack the skills and know-how to design practical and functionally efficient layouts.

3D digital representations of layouts can provide visualizations of design options prior to design commitment; in fact, it is a tool often used by vendors and decorators. Building Information Modeling (BIM), an emerging tool for building digitization, integrates specifications of the building architecture with 3D object models annotated with conceptual type, property, and relationship data. In this work, we adopt a BIM-based methodology to ensure that our layout representations fit in the broader building design life cycle and evaluate our algorithms by implementing and evaluating them in the context of BIMkit (Sydora and Stroulia (2021)), a toolkit that supports a broad range of BIM management use cases, including model storage, model checking against design rules, visual representation and editing, and automated design generation.

Design rules are computational representations of the ergonomic and aesthetic constraints and guidelines as well as user preferences applicable to the placement of the objects inside a space. The evaluation of these rules supports the assessment of the quality of an existing layout, and, embedded in a design-space search algorithm, it enables the automatic generation of valid layouts. An important question then arises regarding the language in which these rules should be represented. Earlier work, Merrell et al. (2011); Yu et al. (2011), proposed rules as geometric formulas, based on shapes, angles, and distance, and many subsequent approaches since have followed suit (Akase and Okada (2013); Kán and Kaufmann (2018); Li et al. (2020); Zhang et al. (2021)). Such representations are not easy to explain to occupants who tend to think in terms of furnishings, and, for the same reason, they are also difficult for interior design experts to express. This is why, in previous work, a domain-specific language (DSL) for rules was designed, that includes furnishing types, properties, and relationships as elements.

Automated design generation is typically formulated as a systematic search through a design space, toward a high-quality design. Numerous such algorithms have been proposed (Zhang et al. (2019)), reporting different metrics on their performance; however, comparing them against each other is quite challenging. Run-time metrics are difficult to standardize since they depend on the underlying hardware, and the design-generation problem is formulated differently across different algorithms, that rely on different representations of layouts and rules. A possible metric that could serve as a standard for comparison is the number of rule evaluations during the search.

In their previous work, Sydora and Stroulia (2020) designed a simple generate-and-test design method, where the list of possible locations and orientations for object placement was pre-generated based on a grid defined relative to the walls of the space. At each iteration, one object was placed at the location and orientation producing the current highest evaluation score, then repeated with the remaining objects. Furnishing object placements were evaluated based on the rule-language evaluation score (as a proof of concept of the rule language as a viable evaluation method). From their experiments, they were able to show that the approach was able to re-create input kitchen layouts by removing all objects and placing them back in following the input design rules for kitchens (although the input kitchens were not implicitly created with the same rules). Then, interpreting the rules from Merrell et al. (2011) into their rule language and running living room experiments, they demonstrated that their algorithm could successfully create functional (relative to the design rules) room designs.

In this paper, we propose a continuous space search method, the Jump Search method. At a high level, the new method places one object at a time, starting with objects subject to more dependency based on the rules. The furnishing object potential next placements are sampled from an incrementally decreasing probability distribution, each evaluated against the rule language evaluator, then greedily selects the highest scoring location and orientation for the movement and repeats until the number of moves is exhausted.

We have evaluated the two methods and a simple simulated-annealing method in terms of, first, how efficiently they find layouts of desired quality, and second, how good the layouts they produce are, given the same "budget" of rule checks. Furthermore, we have evaluated their relative performance under different room types, which have different layout rules and furnishing types.

The contribution of this paper is a new continuous space search algorithm, Jump Search (JS). We have demonstrated the good performance of this novel algorithm, and its effectiveness in delivering high-quality layouts in many different problem scenarios, by comparing it against a previous grid-based method, Grid Search (GS), and a basic continuous space search algorithm based on Simulated Annealing (SA).

The remainder of the paper is organized as follows. First, we review relevant research on interior design and furnishing layout and summarize previous work, including the rule language, layout evaluation, and Grid Search generative design algorithm. Next, we describe our new Jump Search generative design algorithm and report on our comparative evaluation of this algorithm against previous work and a Simulated Annealing generative design method. Finally, we conclude with a summary of our contributions and our future plans.

Related Work

In this section, we will first describe rule constraint formulation methods, that result in evaluation scores. Then, we will outline layout optimization methods, specifically the layout modification and action selection methods. A more in-depth survey of automated layout generation (or sometimes Scene Synthesis) can be found in Zhang et al. (2019).

Configuration Evaluation Methods A common approach is to define rules in terms of geometrical formulas as in Merrell et al. (2011); Yu et al. (2011); Akase and Okada (2013); Kán and Kaufmann (2018); Li et al. (2020); Zhang et al. (2021). The most prominent criticism of this rule definition approach is the difficulty in defining and testing new rules, specifically for domain experts lacking the skills to formulate the rules. These approaches are also not able to be data-driven, i.e. cannot be directly derived from examples.

Relational graphs have been proposed in Yeh et al. (2012); Xu et al. (2015); Liang et al. (2018); Wang et al. (2019); Zhou et al. (2019); Li et al. (2019); Keshavarzi et al. (2020). Typical graphs will pre-define a set of possible relational occurrences, such as "next to" or "facing", with nodes in the graph representing the objects placed in the layout and edges as the relations. A similar approach is based on statistical likelihoods of relational occurrences as in Fisher et al. (2012); Chang et al. (2014); Zhang et al. (2021). Both graph and probabilistic models are typically example-dataset dependent.

To make the rule creation more accessible to domain experts, some methods initially use text-based input (Liang et al. (2018)), before being converted to graphical relation models as rules. Other approaches have defined the rules in terms of actions or activities, rather than classical geometric rules such as in Fisher et al. (2015); Ma et al. (2016); Qi et al. (2018); Fu et al. (2020). Thus, the evaluation scores are determined by simulation or path planning.

Finally, Machine Learning (ML) methods using Neural Networks have been proposed that determine location or location probabilistic mapping based on scene feature (Li et al. (2019); Wang et al. (2019); Zhou et al. (2019); Yang et al. (2021)).

The rule language described in this paper supports a user-friendly process for specifying design rules, in a representation format that reminds textual descriptions rather than geometrical formulas and with support for more complex relations than graph representations. In the future, we plan to demonstrate how machine-learning methods can be used to learn design rules from examples so that users' layout knowledge and preferences can be extracted and used by our methods.

Configuration Variation and Selection Methods There are two broad categories of approaches for placing objects. Either all objects are placed and then simultaneously moved around in a single action, or a single object is moved per action.

Methods for moving all objects include Genetic Algorithms (Akase and Okada (2013)), Particle Swarm Optimization (Li et al. (2020)), or Simulated Annealing with Markov Chain Monte Carlo (MCMC) sampling such as Metropolis-Hastings (Merrell et al. (2011); Yu et al. (2011); Yeh et al. (2012); Qi et al. (2018)). The benefit of moving all objects simultaneously is that objects inherently have arrangement relations to other objects and moving more objects provides a more drastic alteration that could improve evaluation scores. On the other hand, finding more optimal evaluation scores becomes more challenging due to the number of moving pieces.

Placing objects incrementally in the scene has the benefit of finding locally optimal locations for a single object at a time. The main drawback, however, is that the earlier placements have no concept of the later object placements and their evaluations. Some methods that place objects incrementally in the scene are using probabilistic sampling (Fisher et al. (2012); Chang et al. (2014); Xu et al. (2015); Fisher et al. (2015); Ma et al. (2016); Liang et al. (2018); Zhang et al. (2021)) or procedural placement (Germer and Schwarz (2009); Kán and Kaufmann (2018); Keshavarzi et al. (2020); Kan et al. (2021)). Procedural placement generally relies on prior layout knowledge, while probabilistic sampling methods are paired with data-driven methods requiring example models.

The proposed Jump Search method explores possible placements for individual objects one at a time, which results in converging to a valid layout faster. In order to decide which object to place first, it relies on the dependencies between object types and chooses to place first these objects that depend on objects that are already part of the layout.

Background

In this Section, we briefly discuss a rule DSL structure, layout evaluation details, and furnishing placement ordering relevant to the generation methods being compared. An extensive overview of the rule language and layout evaluation (or model-checking) is available in the work by Sydora and Stroulia (2020). Furthermore, the work is placed in the context of BIMkit, a toolkit of BIM-based services for reasoning about buildings, including the domain-specific rule language for describing design rules and the model-checking algorithm for evaluating the compliance of a building design against a set of design rules, which can be found in more detail in Sydora and Stroulia (2021).

Rule Language Structure The rule language is a Domain Specific Language (DSL) for interior design rules that compiles to an executable format (in their case C#). Rules are created using one of three syntax-directed editors available in BIMkit. The structure of an individual rule can be broken down into three components: (1) The relevant filters and quantifiers that define the objects to which the rule applies (or Existential Clauses), (2) the relation checks between these objects, and (3) the logical expression that ties each of the relation checks by logical operators.

First, the Existential Clause (EC), is a filter on the relevant object types and the number of those object types that must pass the checks for the rule to be satisfied, such as, for example, *ALL chairs* or *ANY couch*. In the case of ALL, each of the objects of that type in the layout must pass the checks, while for the ANY case, a single object of the type passing the check is sufficient.

The relation checks evaluate constraints on a variety of spatial relationships between two (or more) objects. For example, a check such as $HDistance(A,B) < 10ft$ computes the $HorizontalDistance$ between objects A and B and evaluates whether it meets the constraint of being $lessthan$ $10ft$. Furthermore, the larger the distance (and closer to the $10ft$ limit) the lower the quality of the layout is. BIMkit's Application Programmable Interface (API) includes a broad range of implementations of functions that compute such relationships in the context of the space and object geometry that can be invoked when the rules are evaluated. These implementations represent various simple, intuitive relationships that are meaningful to occupants and interior designers alike.

The final element of the rule language is the composition of a set of checks through logical AND or OR operators. For example, if we want to check that the distance between object type instance A and B is less than $5ft$ and that object B is facing object A, it would be written as "$Distance(A,B) < 10ft$ AND $Facing(B,A) == TRUE$".

Rather than returning boolean True/False values, the values of the relation checks, and therefore the whole logical expression, are scaled, such that values returned are in the range $[0,1]$. This is achieved by swapping out the check operator in the relation checks and logical operators in the logical expression with corresponding numerical functions. For instance, the *OR* operator is replaced with the maximum value function of each of its contained relation check values. The result is that a final expression result of 1 is a pass and any other value is a degree of failure, with higher values being favoured over lower, implying they are closer to a pass.

Layout Evaluation Layout evaluation (or more generally model-checking) is the process of evaluating the layout design against the design rules. This process takes the rules in the form of the aforementioned rule language, compiles them to executable code, and runs the code on the layout model.

Rules have different levels of importance; they can either be principles (hard constraints), or guidelines or preferences (soft constraints). Layouts are compared first against principle-level rules. Ties are broken by guideline-level rules, and if those are again tied, the layouts are compared against preference-level rules. The final quality of a layout is represented as a percentage of the number of rules that the layout passes over the total number of applicable rules. In calculating this score, only soft constraints are considered since a design is not valid if it fails to pass the principle-level rules.

Ordering of Furnishing Type Placement Through the dependencies among the various object types as captured in the design rules, a hierarchy of object types is implied. The least constrained object type (i.e., the object type that is placed without consideration for the placement of other object types) is at the root of the hierarchy. The object types that depend on the root type through rules are children of the root, and so on, until the leaves of the hierarchy that correspond to the object types with the most dependencies to other object types that must have already been placed. This hierarchy guides the order in which the various objects are placed in the space, similar to Kan et al. (2021).

The Search Algorithms

Grid Search (GS) A generative design method that relies on a grid, defined based on the space walls. For the grid creation, lines are created outward parallel to each of the wall lines at regular intervals; the intersections between the grid lines define the possible locations where objects can be placed. This allows control of the spacing between furnishing objects and between the furnishing and wall objects, at the expense of more limited control of the number of points.

In this algorithm, for which the pseudocode is provided in Sydora and Stroulia (2020), objects can only be placed on the grid-intersection points, in four (at most) possible orientations. The object, location, and orientation combination with the highest resulting evaluation score is selected as the decision action for that iteration. For subsequent object placements, possible locations are removed if they are under an existing object. The process repeated with the remaining objects until either all objects have been placed or the highest possible evaluation score was reached. In this paper, object placement ordering is introduced based on the rules, something that was omitted in the previous work.

Jump Search (JS) The intuition behind the new layout-generation algorithm is to shift one furnishing object at a time into a position that adheres to the relevant layout design rules. Each object begins near the center of the floor and, through iteratively smaller moves, reaches its final location either when a valid location is found or when a maximum number of moves has occurred. The process repeats until each of the selected objects has been placed in the layout.

The Jump Search (JS) algorithm is shown in pseudo-code in Algorithm 1. It takes as input the empty room, the applicable ruleset, the list of to-be-placed furnishing objects, and the following parameters: t_{MAX}: the number of iterations exploring different placements, before an object is placed in its final position; N: the number of moves in a single iteration; M: the number of orientations attempted for each object placement attempt. (The orientations are in the order of π/M Radians); and s_0: the initial standard deviation for sampling movements in the first iteration.

Looking at the rule dependencies, the process determines the order of object types that must be placed in the room, given the existing types already in the room (Wall/Floors/etc) (Line 2). All possible orientations are calculated based on the input M value (Line 3). This initialization phase is the same for all three algorithms compared in this paper.

Starting with the first object in the queue, the object is placed in the center of the room, with a slight offset randomly generated from a normal distribution with a mean value of 0 and standard deviation of $1m$ (Line 5). Then, N move locations are sampled in each iteration, each with an X and Y move value sampled from a normal distribution of mean $= 0$ and iteratively decreasing standard deviation $= s$ (Line 12), initially set to s_0 (Line 9). If the locations generated are outside the room floor, re-sampling occurs.

Algorithm 1: The Jump Search Algorithm.

```
1  Function GenerateSampleLayout(Layout,Objects,Rules):
       Input: $Layout$, $Objects$, $Rules$, $t_{MAX}$, $N$, $M$, $s_0$
       Output: $Layout$
2      $Objects \leftarrow$ ReorderByRuleTypes($Objects, Layout, Rules$)
3      $Orientations \leftarrow$ GetOrientations($M$)
4      foreach $obj \in Objects$ do
5          $Moves \leftarrow$GetMoves($1, Layout.Center, 1$)
6          $xy_{Top}, o_{Top} \leftarrow Moves[0], Orientations[0]$
7          $score_{Top} \leftarrow$Eval($obj, xy_{Top}, o_{Top}, Layout, Rules$)
8          $t \leftarrow 0$
9          $s \leftarrow s_0$
10         while $t < t_{MAX}$ do
11             $xy_{Pre}, o_{Pre}, score_{Pre} \leftarrow xy_{Top}, o_{Top}, score_{Top}$
12             $Moves \leftarrow$GetMoves($N, xy_{Top}, s$)
13             foreach $xy \in Moves$ do
14                 foreach $o \in Orientations$ do
15                     $score \leftarrow$Eval($obj, xy, o, Layout, Rules$)
16                     if $score > score_{Top}$ then
17                         $xy_{Top}, o_{Top}, score_{Top} \leftarrow xy, o, score$
18                     end
19                 end
20             end
21             if $SA == TRUE\ AND\ score_{Top} > score_{Pre}$ then
22                 $P = 1 - e^{(score_{Pre} - score_{Top})/(1 - t/t_{MAX}))}$ if
                     $P < Random.Sample(0, 1)$ then
23                     $xy_{Top}, o_{Top}, score_{Top} \leftarrow$
                         $xy_{Pre}, o_{Pre}, score_{Pre}$
24                 end
25             end
26             $t \leftarrow t + 1$
27             $s \leftarrow s_0 - t * s_0 / t_{MAX}$
28             if $score_{Top} == Rules.Count$ then
29                 break
30             end
31         end
32         Layout $\leftarrow$ PutObject($Layout, obj, xy_{Top}, o_{Top}$)
33     end
```

The potential object placements in each of the N moves and M orientations and compared against each other (Lines 13-20). Because only one object moves at a time, the layout evaluation process is only required to re-evaluate the subset of rules that relate to the moving object, and not the full ruleset, which contributes to the JS algorithm's performance.

The move with the best layout score is selected for the next placement (using the layout evaluation described earlier) (Line 16-18). t is then increased which reduces the next iteration move amount variation, s (Lines 26-27).

Once the object has been placed at a location with a score equal to the maximum possible score for that object (which is equal to the number of rules relating to that object, i.e., all rules passed) or the maximum number of moves is completed (t_{MAX} has been reached), the current object is locked into its final location (Line 32). The next object in the queue is selected and the process repeats until the object queue is empty.

Simulated Annealing (SA) A simpler variant of the above algorithm, based on an implementation of the well-known Simulated Annealing (SA) algorithm (Kirkpatrick et al. (1983)), was implemented as a competitor. It uses nearly the same sequence as the above, however, rather than testing N moves (Line 12), it only attempts one move at each iteration (taking the best of the M orientations). It evaluates the move and if the move has a higher evaluation score, it uses the following formula to determine the move acceptance probability (Lines 21-25):

$$P = 1 - e^{(Q_{previous} - Q_{current})/(1-t/t_{MAX}))}$$

As the iteration count increases, $t \to t_{MAX}$, the likely hood of accepting the move increases, while earlier on, exploration is favoured if the move does not drastically improve the current evaluation score.

Experimental Design

To comparatively evaluate the three above algorithms, we focus on three questions of interest.

1. How long does it take each algorithm to deliver a layout of a desired quality?
2. How does each algorithm perform in different types of rooms, with different functionalities, different sets of layout rules, and different types of furnishings?
3. How does each algorithm perform at different levels of scenario complexity, i.e., layout density?

Performance and Quality Measures To comparatively evaluate the performance of the three algorithms, we run all our experimental scenarios, described in detail below, with the same "budget" of rule checks and we compare the quality of the solutions produced by each of the three algorithms for the same number of checks. For this work, we adopt the percentage of rules that the design complies with as a measure of layout quality or:

$$Score = \sum_{i=1}^{n} Rule[i].eval/n * 100\%$$

where n is the total number of rules (soft constraints only). The intuition is that the algorithm that produces layouts that meet all the applicable rules with the smallest number of rule checks is best. Thus, embedded in each algorithm implementation is a counter for the number of times a rule-evaluation check is invoked and all algorithms are invoked with a check budget as a parameter.

Because the number of evaluation checks is dependent on the room shape for GS, the evaluation check budget is calculated for the GS method first. The parameters that result in closely matching evaluation check counts for the JS and SA methods are then calculated. For the JS the t_{MAX} and N are both set to the square root of the number of location points, while for the SA method, t_{MAX} is set to the number of location points as N is set to 1.

Room Types Different room types present different layout challenges since each room type is associated with different types of objects, with dimensions that may vary to a greater or lesser degree, and with different rules governing their placements. This is why we evaluated the three algorithms in four room types: living rooms, kitchens, bedrooms, and bathrooms. While there are many other categories of room types (offices, game rooms, etc.), we believe these three also represent a range of features such as dependency amount on movable objects (furnishing) versus dependency amount on static objects (walls) and room layout density.

Rulesets Different publications consider different sets of rules for the above room types. In our work, the three rule sets have been developed based on our examination of numerous example layouts we collected from design websites. Multiple rules can be related to the same object types and there is no guarantee that rules do not conflict with one another. Therefore, a perfect evaluation score may not be achievable in all cases.

Layout Scenarios Intuitively, given the type of a room, two parameters influence the complexity of the layout problem: (a) the shape of the room, and (b) the number and types of objects to be placed in the room. Therefore, we experiment with two different shapes of living rooms: rectangular or L-shaped. L-shaped rooms are less likely to fit objects in the way the rules intended them as in the standard rectangular rooms, making full rule compliance more challenging.

All layout generations start with one of the empty rooms, defined by their walls, floors, doors, and windows, and a list of furnishing objects to be placed in the layout. All furnishing objects are described in terms of their 3D shapes (or mesh representations) and various associated metadata, including type and facing orientation. Without the rules, furnishing objects have no inherent restriction on how and where they can be placed in the layout model (i.e. no location "snapping"). For this study, the furnishing list for each is based on common objects tested in prior research (see Related Work Section) and commonly found in each room type.

Finally, in order to evaluate method adaptability to higher constrained layout problems, we conducted easy and hard living room and kitchen experiments. The difference between the two is the hard living room problem has additional objects, namely one extra side table and two extra armchairs; the hard kitchen consists of an additional two chairs and a table with rules for both.

Based on the above rationale, we evaluated the three algorithms in eight design scenarios: Standard Living Room Easy, Standard Living Room Hard, L-Shape Living Room Easy, L-Shape Living Room Hard, Kitchen Easy, Kitchen Hard, Bedroom, and Bathroom.

Results and Discussion

Table 1 reports the performance of the three algorithms on the eight evaluation scenarios described above. Each experiment was run 10 times for each generation method and each rule check budget. Figure 1 shows outputs from the JS method.

For a runtime benchmark: on an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz processor with 16.0 GB RAM, 10 iterations with 5 moves and 4 orientations for the rectangular living room scenario (14 object total: 6 initial objects and 8 added furnishing objects) resulted in a compliance score of 96.78%, required 1616 evaluation checks

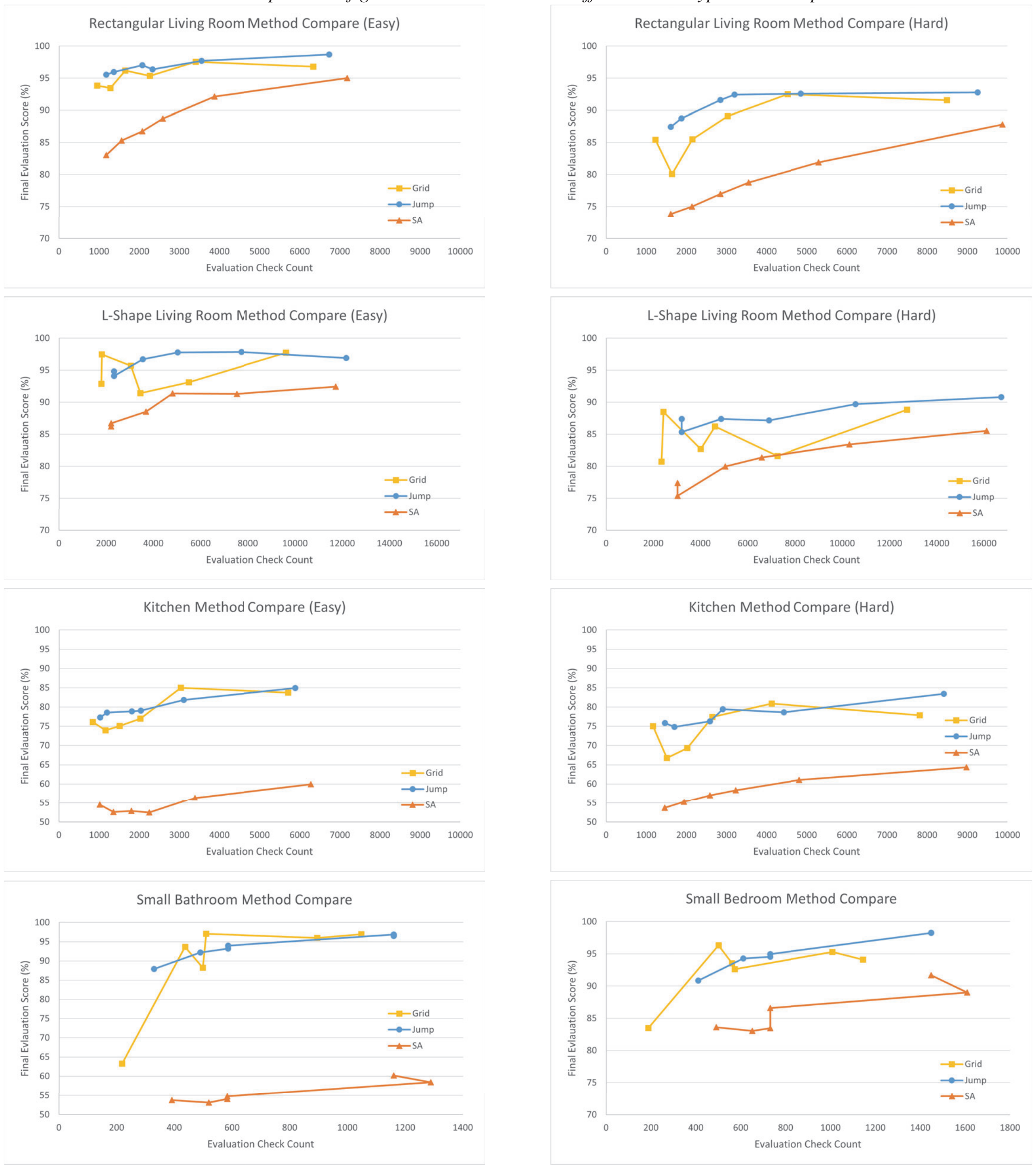

Table 1: Comparison of generation methods over different room types and complexities.

(not all rules checked each evaluation), and a runtime of 23.3083559 seconds.

Let us now revisit the three evaluation questions above, in light of the experimental results.

How long does it take each algorithm to deliver a layout of a desired quality? The answer to this is largely dependent on the room size, type, objects, and rules, however, our experiments show some general benchmarks for different room scenarios. The most prominent difference between the GS and JS methods is notable consistency in the correlation between the check budget and final evaluation score. As evident by the jagged lines, the GS method suffers from heavy reliance on the grid spacing parameter, thus making the final evaluation less predictable and more dependent on parameter tuning.

How does each algorithm perform in different types of rooms, with different functionalities, different sets of layout rules, and different types of furnishings? The strength of the GS method is the ability to place objects directly against a wall, as typically found in many rooms. Therefore, GS performed comparably well in kitchens and bathrooms where nearly all objects are against the wall but performed poorly in living rooms where objects were less wall dependent. Conversely, SA suffers due to the lack of motion of the objects because of the probability of rejecting potential better movements, making moving towards the walls of the rooms less likely.

How does each algorithm perform at different levels of scenario complexity, i.e., layout density? On the comparison of layout complexity, when more of the same objects are added to the layout, as in the living room case, all methods perform consistently poorer than their less complex counterpart, meaning complexity inherently makes the search for high-quality layouts more difficult. Evidently, adding additional objects and rules that do not overlap in the kitchen complexity scenario does not have an effect on final quality, only that more check iterations are required due to the addition of objects.

When looking at result variance, the GS method can randomly select over tie breaks, however, these are less frequent given the rule score quantification. Thus, the same result (usually scoring moderately high) is often reproduced. JS and SA, on the other hand, produce more variety in results, although some early selections result in lower local optima. When looking at strictly the maximum of all produced layouts from each of the 10 runs for each experiment, one variant of the JS always produced the highest result.

One area where the JS could have been improved, and more prominently in the SA, would have been a more intelligent sampling method. SA for instance is often paired with sampling methods such as Metropolis–Hastings (Metropolis et al. (1953)), which samples from unknown sampling distributions. In this case, sampling could be prioritized closer to higher evaluation scoring locations, making the location sampling more efficient. However, the JS does this to a minor extent by sampling nearer to the current location, which is also the highest-scoring location. Finally, adjusting the SA to more greedily accept the next location may have improved its results.

Overall, the JS method should be considered the stronger of the compared algorithms as it:

1. produces equal or better results on average, given a reasonable check budget;
2. has minimal reliance on the input room shape and size as the only impact is on the initial move amount, thus making the quality more predictable based on budgetary check requirements;
3. is more flexible when more challenging layout experiments are presented; and
4. gives more variation in the final layouts, as the locations explored are more random.

Some of the limitations of the JS algorithm include the fact that objects start near the center of the room, making movement closer to walls more challenging than the GS. Making no assumptions about any object relationships makes the solution general and fully customizable to the rules. However, intuitively some objects are usually placed relative to each other, such as for example couches relative to walls or nightstands relative to beds. In its current form, JS does not take advantage of this domain knowledge. Thus, while JS supports generality, having some procedural placement, like "snapping" would improve runtime in practice if applicable.

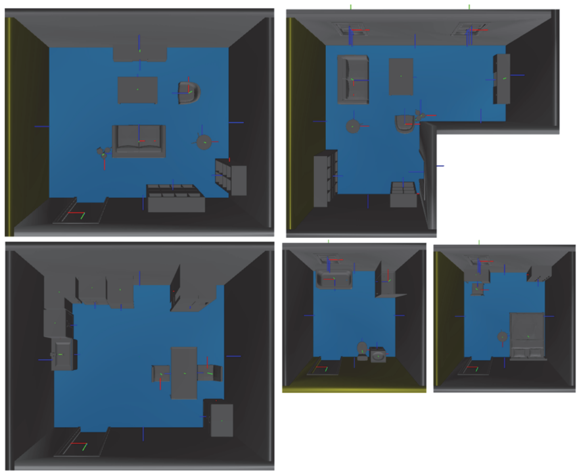

Figure 1: Rooms generated from new Jump Search method. Living room (top left), Living room (top right), Kitchen (bottom left), bathroom (bottom center), bedroom (bottom right).

Conclusion

Automation offers the potential for significant improvements in building design and construction. More specifically, configuration of the layout of furnishings and appliances in the available interior spaces of the building can have a significant impact on the comfort of the occupants, as a poor arrangement can lead to inaccessible objects, have negative ergonomic effects on occupants, and generally make a space unpleasant when it could have been better utilized otherwise. Automated interior layout generation has been a highly researched problem over the last 10 years. However, previous algorithms suffer from two important shortcomings. First, they tend to rely on non-intuitive and difficult-to-explain formulation of the rules that drive the generation of the candidate layouts, and second, they are difficult to compare against each other to analyze their relative merits and shortcomings.

In this paper, we propose a novel layout-configuration algorithm, Jump Search, that searches through a continuous design space. Jump Search relies on a domain-specific language for rule representation and a corresponding method for rule evaluation, which was first embedded in a simple grid-based search algorithm. Relying on the "number of rule evaluations" as a metric, we have conducted a comparative evaluation of Jump Search against a Grid Search algorithm and a simple Simulated Annealing search algorithm, in eight different design scenarios of varied difficulty. Our results demonstrate that Jump Search outperforms both other algorithms in terms of layout quality relative to computational complexity, while also exploring a bigger variety of layouts.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Akase, R. and Okada, Y. (2013). Automatic 3d furniture layout based on interactive evolutionary computation. In Seventh International Conference on Complex, Intelligent, and Software Intensive Systems, pages 726–731.

Chang, A., Savva, M., and Manning, C. D. (2014). Learning spatial knowledge for text to 3d scene generation. In Conference on empirical methods in natural language processing (EMNLP), pages 2028–2038.

Fisher, M., Ritchie, D., Savva, M., Funkhouser, T., and Hanrahan, P. (2012). Example-based synthesis of 3d object arrangements. ACM Transactions on Graphics (TOG), 31(6):1–11.

Fisher, M., Savva, M., Li, Y., Hanrahan, P., and Nießner, M. (2015). Activity-centric scene synthesis for functional 3d scene modeling. ACM Transactions on Graphics (TOG), 34(6):1–13.

Fu, Q., Fu, H., Yan, H., Zhou, B., Chen, X., and Li, X. (2020). Human-centric metrics for indoor scene assessment and synthesis. Graphical Models, 110:101073.

Germer, T. and Schwarz, M. (2009). Procedural arrangement of furniture for real-time walkthroughs. Computer Graphics Forum, 28(8):2068–2078.

Kán, P. and Kaufmann, H. (2018). Automatic furniture arrangement using greedy cost minimization. In IEEE Conference on Virtual Reality and 3D User Interfaces (VR), pages 491–498.

Kan, P., Kurtic, A., Radwan, M., and Rodriguez, J. M. L. (2021). Automatic interior design in augmented reality based on hierarchical tree of procedural rules. Electronics, 10(3):245.

Keshavarzi, M., Parikh, A., Zhai, X., Mao, M., Caldas, L., and Yang, A. Y. (2020). Scenegen: Generative contextual scene augmentation using scene graph priors. arXiv preprint arXiv:2009.12395.

Kirkpatrick, S., Gelatt Jr, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220(4598):671–680.

Li, M., Patil, A. G., Xu, K., Chaudhuri, S., Khan, O., Shamir, A., Tu, C., Chen, B., Cohen-Or, D., and Zhang, H. (2019). Grains: Generative recursive autoencoders for indoor scenes. ACM Transactions on Graphics (TOG), 38(2):1–16.

Li, Y., Wang, X., Wu, Z., Li, G., Liu, S., and Zhou, M. (2020). Flexible indoor scene synthesis based on multi-object particle swarm intelligence optimization and user intentions with 3d gesture. Computers & Graphics, 93:1–12.

Liang, Y., Xu, F., Zhang, S.-H., Lai, Y.-K., and Mu, T. (2018). Knowledge graph construction with structure and parameter learning for indoor scene design. Computational Visual Media, 4:123–137.

Ma, R., Li, H., Zou, C., Liao, Z., Tong, X., and Zhang, H. (2016). Action-driven 3d indoor scene evolution. ACM Trans. Graph., 35(6):173–1.

Merrell, P., Schkufza, E., Li, Z., Agrawala, M., and Koltun, V. (2011). Interactive furniture layout using interior design guidelines. ACM transactions on graphics (TOG), 30(4):1–10.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of state calculations by fast computing machines. The journal of chemical physics, 21(6):1087–1092.

Qi, S., Zhu, Y., Huang, S., Jiang, C., and Zhu, S.-C. (2018). Human-centric indoor scene synthesis using stochastic grammar. In IEEE Conference on Computer Vision and Pattern Recognition, pages 5899–5908.

Sydora, C. and Stroulia, E. (2020). Rule-based compliance checking and generative design for building interiors using bim. Automation in Construction, 120:103368.

Sydora, C. and Stroulia, E. (2021). Bim-kit: An extendible toolkit for reasoning about building information models. In European Conference on Computing in Construction (EC3), pages 107–114.

Wang, K., Lin, Y.-A., Weissmann, B., Savva, M., Chang, A. X., and Ritchie, D. (2019). Planit: Planning and instantiating indoor scenes with relation graph and spatial prior networks. ACM Transactions on Graphics (TOG), 38(4):1–15.

Xu, W., Wang, B., and Yan, D.-M. (2015). Wall grid structure for interior scene synthesis. Computers & Graphics, 46:231–243.

Yang, B., Li, L., Song, C., Jiang, Z., and Ling, Y. (2021). Automatic interior layout with user-specified furniture. Computers & Graphics, 94:124–131.

Yeh, Y.-T., Yang, L., Watson, M., Goodman, N. D., and Hanrahan, P. (2012). Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. ACM Transactions on Graphics (TOG), 31(4):1–11.

Yu, L. F., Yeung, S. K., Tang, C. K., Terzopoulos, D., Chan, T. F., and Osher, S. J. (2011). Make it home: automatic optimization of furniture arrangement. ACM Transactions on Graphics (TOG)-Proceedings of ACM SIGGRAPH, 30(4).

Zhang, S.-H., Zhang, S.-K., Liang, Y., and Hall, P. (2019). A survey of 3d indoor scene synthesis. Journal of Computer Science and Technology, 34(3):594–608.

Zhang, S.-K., Xie, W.-Y., and Zhang, S.-H. (2021). Geometry-based layout generation with hyper-relations among objects. Graphical Models, 116:101104.

Zhou, Y., While, Z., and Kalogerakis, E. (2019). Scenegraphnet: Neural message passing for 3d indoor scene augmentation. In IEEE/CVF International Conference on Computer Vision, pages 7384–7392.