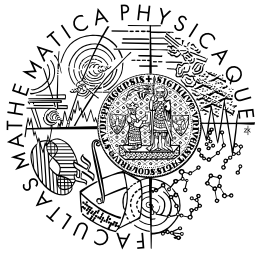


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ondřej Nepožitek

Procedural 2D Map Generation for Computer Games

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Jakub Gemrot, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank to my supervisor, Mgr. Jakub Gemrot, Ph.D., for his guidance during the writing of this thesis and for his valuable advice.

Title: Procedural 2D Map Generation for Computer Games

Author: Ondřej Nepožitek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: In some video games, levels are procedurally generated to increase game's replayability. However, such levels may often feel too random, unbalanced and lacking an overall structure. Ma et al. (2014) proposed an algorithm to solve this problem. Their method takes a set of user-defined building blocks as an input and produces layouts that all follow the structure of a specified level connectivity graph. The algorithm is based on two main concepts. The first one is that the input graph is decomposed into smaller chains and these are laid out one at a time. The second one is that configuration spaces are used to define valid relative positions of building blocks. In this thesis, we present an implementation of this method in a context of 2D tile-based maps. We enhance the algorithm with several new features, one of them being a mode to quickly add short corridors between neighbouring rooms. We also propose speed improvements, including a smarter decomposition of the input graph and tweaks of the stochastic method that is used to lay out individual chains. The resulting algorithm is able to quickly produce diverse layouts, which is demonstrated on a variety of input graphs and building blocks sets. Benchmarks of our algorithm show that it can achieve up to two orders of magnitude speedup compared to the original method.

Keywords: procedural content generation, computer games, 2D maps, rooms

Contents

Introduction

Procedural generation is a method of creating content algorithmically rather than by hand. In video games, it is often used to increase game's replayability. The classic example is the game Rogue[20] which is a dungeon crawler video game inspired by a board game Dungeons & Dragons[7]. It contains procedurally generated dungeon levels, treasures and monster encounters and all that leads to a unique experience on every playthrough. Procedural techniques are also used in newer games including Borderlands[1], Diablo[16] or Minecraft[14].

Procedural generation is often used to create game levels. One popular approach to this problem is to use binary space partitioning[17]. This algorithm starts with a rectangular area and recursively splits it until there are enough subareas. Some subareas are then chosen to represent rooms and these are connected by corridors. Another possible approach is a so-called agent-based dungeon growing[17]. The algorithm starts with an area that is completely filled with wall cells and an agent is spawned at a specified location. The agent is controlled by a predefined AI and moves through the area, digging corridors and placing rooms.

The problem with these algorithms is that a game designer often loses control over the flow of gameplay, and generated layouts may feel too random and lacking an overall structure[6, 11]. Although this approach may be appropriate in some genres, Dormans & Sanders[6] note that the gameplay these algorithms support often does not translate to action-adventure games. These games are story-driven with concepts like puzzle-solving and exploration making the majority of the gameplay. They aim to solve this problem by using generative grammars to generate both missions and spaces of a game. Ma et al.[11] propose a different approach. Their method takes a set of room shapes as an input and produces layouts that all follow the structure of a specified level connectivity graph.

The goal of this thesis is to implement an algorithm that will allow game designers to retain control over the structure of generated layouts. We will focus on generating 2D tile-based maps[1], as there is no known method that can be directly used in such a context. Our algorithm will be based on the aforementioned work of Chongyang Ma et al. because the main concepts of their method are quite universal and can be modified to handle tile-based layouts. The advantage of this method over the grammar-based approach is that it is quite simple to define its input, while construction of generative grammars can be relatively hard.

Goals

1. Implement the algorithm from Ma et al.[11] in a context of 2D tile-based maps.
2. Propose changes that will improve speed of the original method.
3. Provide a framework that will allow programmers to replace or extend individual building blocks of our method.
4. Provide a simple GUI to control the algorithm.

[1] Maps that consist of small square graphic images that are laid out in a grid.

Structure

The structure of the thesis is following: The algorithm from Ma et al. is analyzed in chapter 1. Modifications and performance improvements of the original method are described in chapter 2. The architecture of the framework is covered in chapter 3. Results, including benchmarks of our method and several generated layouts, are presented in chapter 4. And the final chapter concludes the thesis.

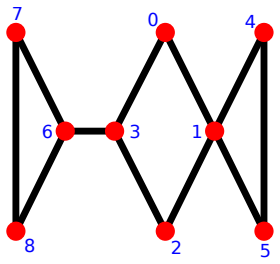
1. Analysis and related work

In this section, we will first describe the method from Ma et al. and then discuss some of its shortcomings.

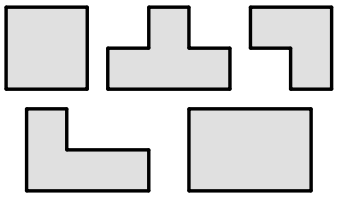
1.1 Algorithm

The algorithm takes a set of polygonal building blocks and a level connectivity graph as an input. Nodes in that graph represent rooms, and edges define connectivities between them. The goal of the algorithm is to assign a shape and a position to each node in the graph in a way that no two nodes intersect and that every pair of neighbouring nodes share a common boundary segment.

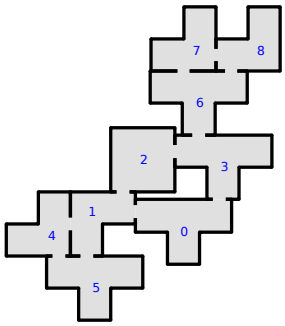
Instead of searching through all possible positions and room shapes of nodes in the input graph, we use configuration spaces to define valid relative positions of individual room shapes. The configuration space of two nodes is a set of such positions in $\mathbb{R}^2$ that if we translate one of the nodes to that position, both nodes can be connected by doors and do not intersect. However, it is not possible to formulate the whole problem as a configuration space computation because even a restricted version of such a computation was shown to be PSPACE-hard[9]. Therefore, a probabilistic optimization technique is used to efficiently explore the search space. To further speed up the optimization, we break the input problem to smaller and easier subproblems. This is done by decomposing the graph into smaller parts (called chains) and then laying them out one at a time.



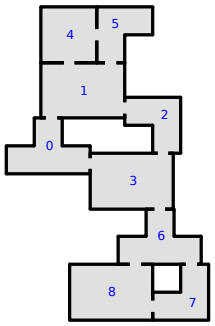
(a) Level connectivity graph



(b) Building blocks



(c) Generated layout

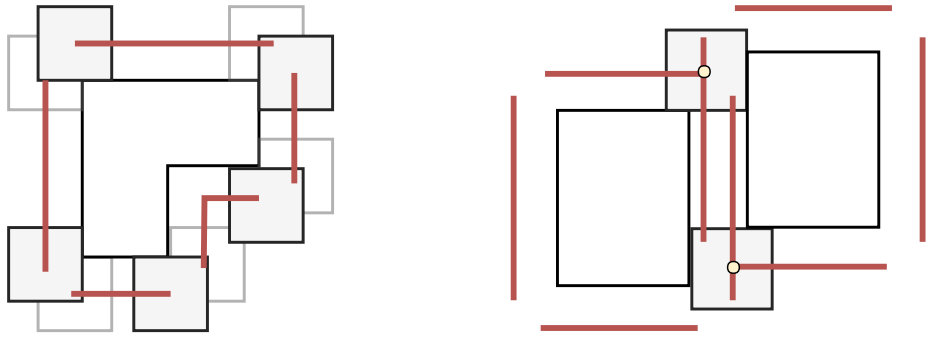


(d) Generated layout

Figure 1: Output of the original algorithm from Ma et al. (c) and (d) demonstrate layouts that were generated from the level connectivity graph in (a) and building blocks shown in (b).

1.1.1 Configuration spaces

For a pair of polygons, one fixed and one free, a configuration space is a set of such positions of the free polygon that the two polygons do not overlap and can be connected by doors. When working with polygons, each configuration space can be represented by a possibly empty set of lines (Figure 2a) and can easily be computed with basic geometric tools. By leveraging these valid position sets, the size of the space we have to search is dramatically reduced.



(a) (b)

Figure 2: Configuration spaces. (a) shows the configuration space (red lines) of the free square with respect to the fixed l-shaped polygon. It defines all the locations of the center of the square such that the two blocks do not intersect and are in contact. (b) shows the intersection (yellow dots) of configuration spaces of the moving square with respect to the two fixed rectangles.

Following algorithm is used to compute the configuration space of two blocks, one being fixed and the other one being allowed to move. We pick a reference point on the moving block and consider all locations in $\mathbb{R}^2$ such that, if the polygon is moved in a way that the reference point is placed at that location, both the moving block and the fixed block contact each other but do not intersect. The set of all these points forms the configuration space of the two blocks (Figure 2a). To get the configuration space of a moving block with respect to two or more fixed blocks, the intersection of the individual configuration spaces is computed (Figure 2b).

Because the block geometry is fixed during optimization, configuration spaces of all pairs of block shapes are precomputed to speed up the process.

1.1.2 Incremental layout

Authors of the method note that chains, or graphs where each node has at most two neighbours, are relatively easy to lay out. Therefore, the input graph is decomposed into chains and these are later laid out one at a time. The strategy of decomposing a graph into chains is based on computing a planar embedding of the graph and then using faces of the embedding to form the basis of the decomposition.

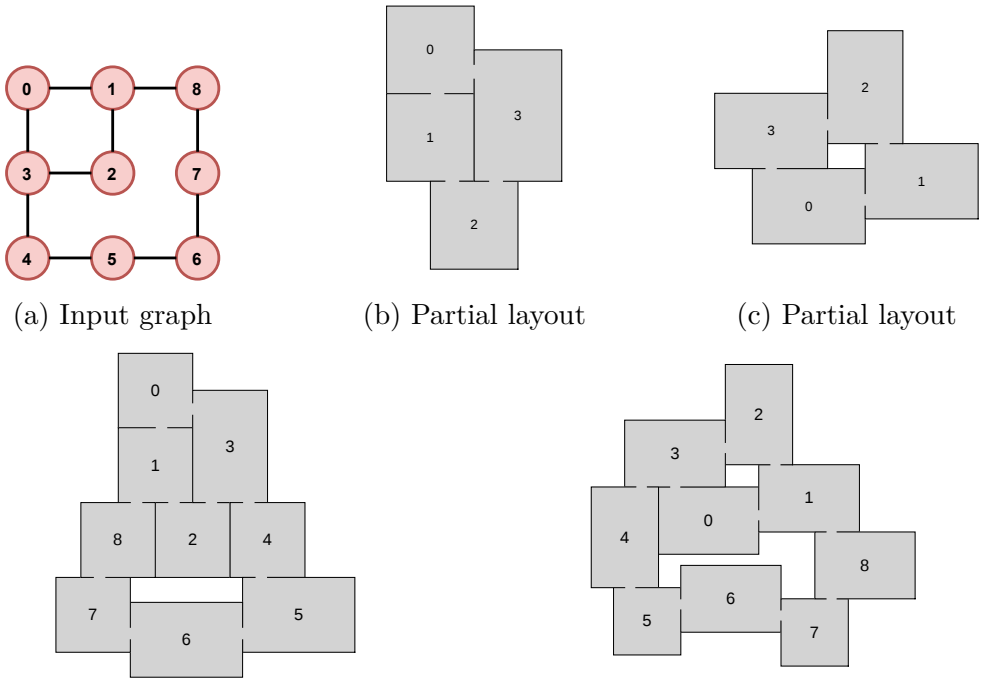
Our final output layout is always a single connected component, hence there is no benefit in laying out separate components and then trying to join them, as the joining process can be quite difficult. Instead, after laying out a chain, the next chain to connect is always one that is connected to already laid out vertices.

```
1  Input: planar graph G, building blocks B, layout stack S
2
3  procedure IncrementalLayout(c,s)
4    Push empty layout into S
5
6    repeat
7      s ← S.pop()
8      Get the next chain c to add to s
9      AddChain(c, s)                                  // extend the layout to contain c
10
11     if extended partial layouts were generated then
12       Push new partial layouts into S
13     end if
14
15   until target # of full layouts is generated or S is empty
16 end procedure
```

Algorithm 1: Incremental layout.

Algorithm 1 shows the implementation of incremental layout. In each iteration of the algorithm (lines 6 - 15), we first take the last layout from the stack (line 7) and compute which chain should be added next (line 8). This can be simply done by storing the number of the last chain that was added to each partial layout. The following step is to add the next chain to the layout (line 9), generating multiple extended layouts and storing them (lines 11 - 13). If the extension step fails, no new partial layouts are added to the stack and the algorithm has to continue with the last stored partial layout. Throughout the paper, we refer to this situation as *backtracking* because the algorithm cannot extend the current partial layout and has to go back and continue with a different stored layout. It is usually needed when there is not enough space to connect additional chains to already laid out vertices (Figure 5). Backtracking is also the reason why we always try to generate multiple extended layouts (line 9). Otherwise, we would have nothing to backtrack to. The process terminates when enough full layouts are generated or if no more distinct layouts can be computed.

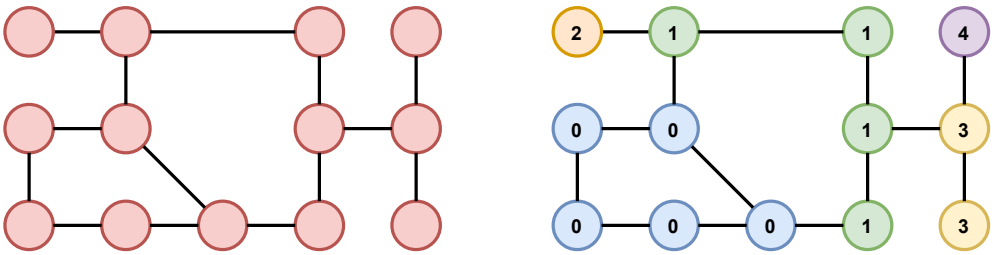
To decompose a graph into chains, a classic algorithm[5] to find a planar embedding is applied to the graph. That embedding is then used to get all the faces of the graph. The basic idea of the decomposition is that cycles are harder to lay out because there are more constraints on the nodes. Therefore, it is attempted to put cycles to the beginning of the decomposition, thus making sure they are processed as soon as possible and the chance of backtracking in later phases of the algorithm is decreased. The first chain in the decomposition is formed by the smallest face of the embedding and following faces are then added in a breadth-first search ordering. If there are more faces to choose from, the smallest one is used first. When there are no faces left, remaining acyclical components are added. In Figure 4 we see can an example of a chain decomposition that is obtained by following these steps.



(a) Input graph (b) Partial layout (c) Partial layout

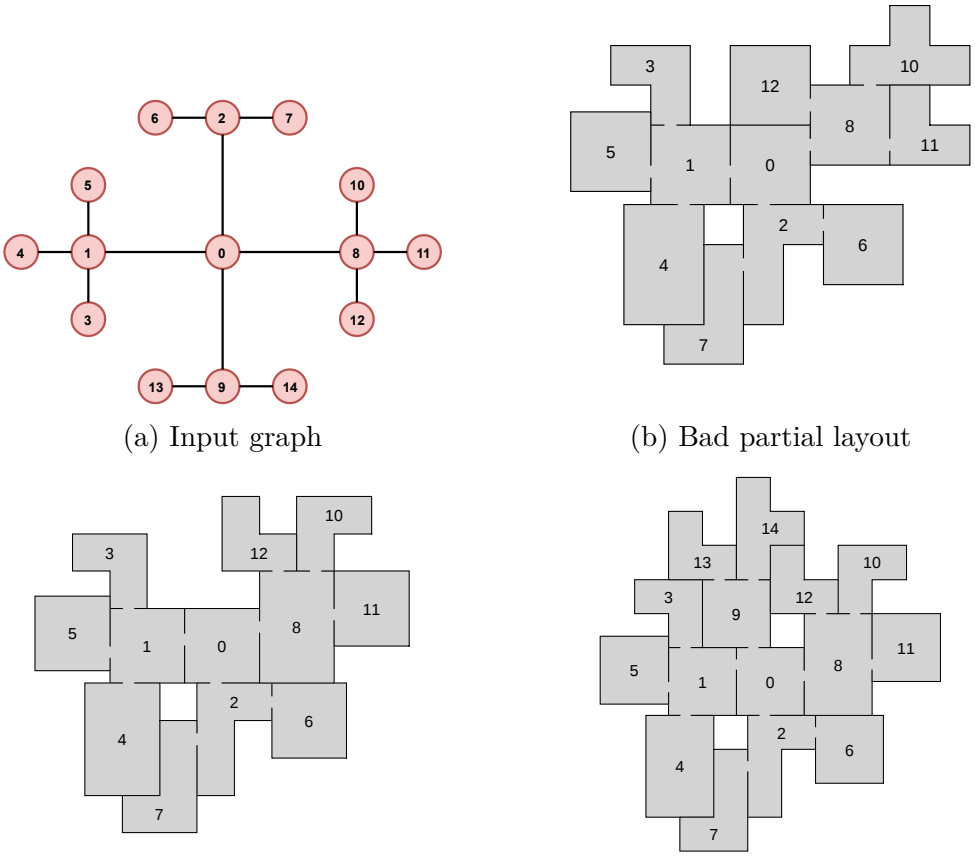
(d) Full layout (e) Full layout

Figure 3: Incremental layout. (b) and (c) show two partial layouts after laying out the first chain. (d) shows a full layout after extending (b) with the second chain. (e) shows a full layout after extending (c) with the second chain.



(a) Input graph (b) Chain decomposition

Figure 4: Chain decomposition. (b) shows an example of how can (a) be decomposed into chains. Each color represents one chain. Numbers show in what order were the chains created.



(a) Input graph

(b) Bad partial layout

(c) Good partial layout

(d) Full layout

Figure 5: Backtracking. (b) shows an example of a bad partial layout because there is not enough space to connect nodes 0 and 9. Backtracking to a different partial layout (c) is needed to generate a full layout (d).

1.1.3 Simulated annealing

The authors chose simulated annealing framework to explore the space of possible layouts for individual chains. The reason for choosing simulated annealing is that it is able to produce multiple partial layouts in a single run. This is useful in two situations. First, it allows us to backtrack if we are unable to lay out a chain. And second, we are able to quickly generate subsequent full layouts. Instead of starting the generation process all over again from an empty layout, we start with an already computed partial layout that was produced by simulated annealing in the process of generating the first layout.

The goal of simulated annealing is to assign a position and a room shape to every node in the current chain, such that all constraints are satisfied.

Simulated annealing operates by iteratively considering local perturbations to the current configuration, or layout. That means that we create a new configuration by randomly picking one node and changing its position or shape. If the new configuration improves the energy function, it is always accepted. Otherwise, there is a small probability of accepting the configuration anyway. The probability of accepting worse solutions decreases as the temperature of simulated annealing tends to zero.

The energy function is constructed in a way that it heavily penalizes nodes that intersect and neighbouring nodes that do not touch.

$$E = e^{\frac{A}{\omega}} e^{\frac{D}{\omega}} - 1$$

A is the total area of intersection between all pairs of blocks in the layout. D is the sum of squared distances between the centers of pairs of blocks that are neighbours in the input graph, but which are not in contact. The value of ω affects how often is simulated annealing allowed to move to a worse configuration, and was empirically derived to be one hundred times the average area of building blocks.

To speed up the process, they try to find an initial configuration with a low energy. To do that, a breadth-first search ordering of nodes from the current chain is constructed, starting from the ones that are adjacent to already laid out nodes. Ordered nodes are placed one at a time, sampling the configuration space with respect to already laid out neighbours, choosing the configuration with the lowest energy.

```
1  Input: chain c, initial layout s
2
3  procedure AddChain(c,s)
4    generatedLayouts ← Empty collection of generated layouts
5    t ← t0                                                  // Initial temperature
6
7    for i ← 1,n do                                          // n: # of cycles in total
8      for j ← 1,m do                                        // m: # of trials per cycle
9        s' ← PerturbLayout(s, c)
10
11       if s' is valid then
12
13         if s' ∪ c is full layout then output it
14         else if s' passes variability test
15
16           Add s' into generatedLayouts
17           Return generatedLayouts if enough extended layouts computed
18         end if
19       end if
20
21       if ΔE < 0 then                                      // ΔE = E(s') - E(s)
22         s ← s'
23       else if rand() < e^(−ΔE/(k∗t)) then
24         s ← s'
25       else
26         Discard s'
27       end if
28
29     end for
30
31     t ← t ∗ ratio                                         // Cool down temperature
32   end for
33 end procedure
```

Algorithm 2: Simulated annealing. This pseudocode uses $n = 50$, $m = 500$, $t_0 = 0.6$ and k is computed using ΔE averaging[8].

```
Input: layout, current chain

procedure PerturbLayout(layout, chain)
  configSpaces ← Get precomputed configuration spaces
  perturbShape ← Pick at random – 40% true, 60% false
  nodeToBePerturbed ← Get a random node from the chain

  if perturbShape then
    Pick a random shape for nodeToBePerturbed
  else
    Use configSpaces to get a random position from the interesection
    of configuration spaces of neighbours of nodeToBePerturbed
  end if

  Update position/shape of nodeToBePerturbed
  Update energy of layout

  return perturbed layout with updated energy
end procedure
```

Algorithm 3: Layout perturbation.

1.2 Shortcomings

1.2.1 Overall speed

The main problem of the algorithm is that it is just not reliable enough to be used directly in a game. In Table 1, you can see a benchmark of the algorithm when used on input graphs that are shown throughout this thesis. The algorithm clearly struggles when it has to generate a layout that is based on a complex input graph. We can see that for such inputs the success rate is below 60% and if we manage to generate a layout, it takes tens of seconds. If we want to use the algorithm in a game, we should aim to generate a layout under a few seconds and with a high success rate.

Note that the original paper contains a benchmark that shows a higher success rate than what can be seen in Table 1. However, we were not able to reproduce these results with the available implementation that can be found on Github[11]. And even with a higher success rate, the algorithm still needs tens of seconds to handle complex input graphs.

Input	**Success rate**	**Time avg/med**	**Iterations avg/med**
Figure 5 (15 vertices)	40%	38.00s/33.00s	667k/1000k
Figure 8 (13 vertices)	80%	14.70s/2.25s	62k/163k
Figure 22 (9 vertices)	100%	0.56/0.07s	16k/28k
Figure 23 (17 vertices)	40%	19.00s/46.00s	400k/199k
Figure 24 (41 vertices)	8%	50.00s/55.00s	920k/1000k
Figure 25 (21 vertices)	60%	30.00s/24.00s	500k/144k
Figure 26 (11 vertices)	100%	10.00s/20.00s	183k/199k

Table 1: Benchmark of the original implementation. Success rate shows how often was the algorithm able to generate a valid layout. We provide average and median values for both the time and the number of iterations.

1.2.2 Corridors between rooms

In the original paper, it is shown that the method can be used to generate layouts with rooms connected by corridors. To achieve that, a new node is added between every two neighbouring nodes in the input graph and all these new nodes get assigned a set of room shapes that was made for corridors. The advantage of this approach is that the algorithm does not care whether a room has any special meaning and therefore no code modifications are needed.

The problem is, however, that we now have almost twice as many nodes than before and the algorithm, therefore, needs significantly more time to generate a valid layout. When we tried this approach with corridors that were rather short, we also encountered a problem with the energy function, because it takes into account what is the area of intersection of individual pair of nodes. The corridors were so small that their contribution to the energy of a layout was negligible, causing the algorithm to not converge at all.

2. Algorithm

Our algorithm is based on the method from Ma et al. that was described in the previous section. In this section, we will first discuss the consequences of using the method in a tile-based context, then introduce new features and finally propose some performance improvements.

New features:

- Corridors between rooms
- Explicit door positions
- Custom constraints
- Different probabilities for room shapes

Speed improvements:

- Simulated annealing parameters
- Chain decomposition
- Lazy evaluation

2.1 Tile-based output

One of the goals of this thesis is to reimplement the algorithm from Ma et al. in a context of tile-based maps, i.e. maps that consist of small square graphic images that are laid out in a grid. The original method works with real coordinates and cannot be, therefore, directly used to generate such maps.

The biggest change that comes with using integer coordinates is that we can now use only rectilinear polygons instead of arbitrary polygons for room shapes. Rectilinear polygons are polygons with each side being parallel to one of the axes.

It is interesting to observe how the algorithm behaves if we scale all the rooms up to simulate, to some extent, real coordinates. From our experience, it seems that the convergence rate is slightly better when working with a discrete space where the algorithm has less options to choose from (e.g. available positions when perturbing nodes).

2.2 New features

In this section, we will discuss enhancements of the original algorithm.

2.2.1 Corridors between rooms

In the Shortcomings section, we discussed how are corridors handled in the original algorithm. We present a different approach to this problem. We use two different instances of configuration spaces. The first one is the basic one in which a position of two rooms is valid when both rooms touch and do not overlap. The second one, on the other hand, accepts only positions where the two rooms are exactly a specified distance away from each other (and also do not overlap).

Algorithm 4 shows how we perturb a layout when we want to have rooms connected by corridors[1]. By using the second type of configuration spaces (lines 14 - 16), we should converge to a state where all pairs of non-corridor nodes of the current chain have a space between them (Figure 6b). When this happens, we switch to the first type of configuration spaces (line 22) and try to greedily add all corridors rooms, i.e. for each corridor node pick the first valid position that connects corresponding non-corridor nodes. In some cases, we may not be able to lay out all corridor rooms if, for example, the only way to add a corridor is to cross another one. Such cases, however, are not very frequent and if we encounter them, we just abort the current attempt, remove already added corridors (lines 24 - 26) and return to simulated annealing.

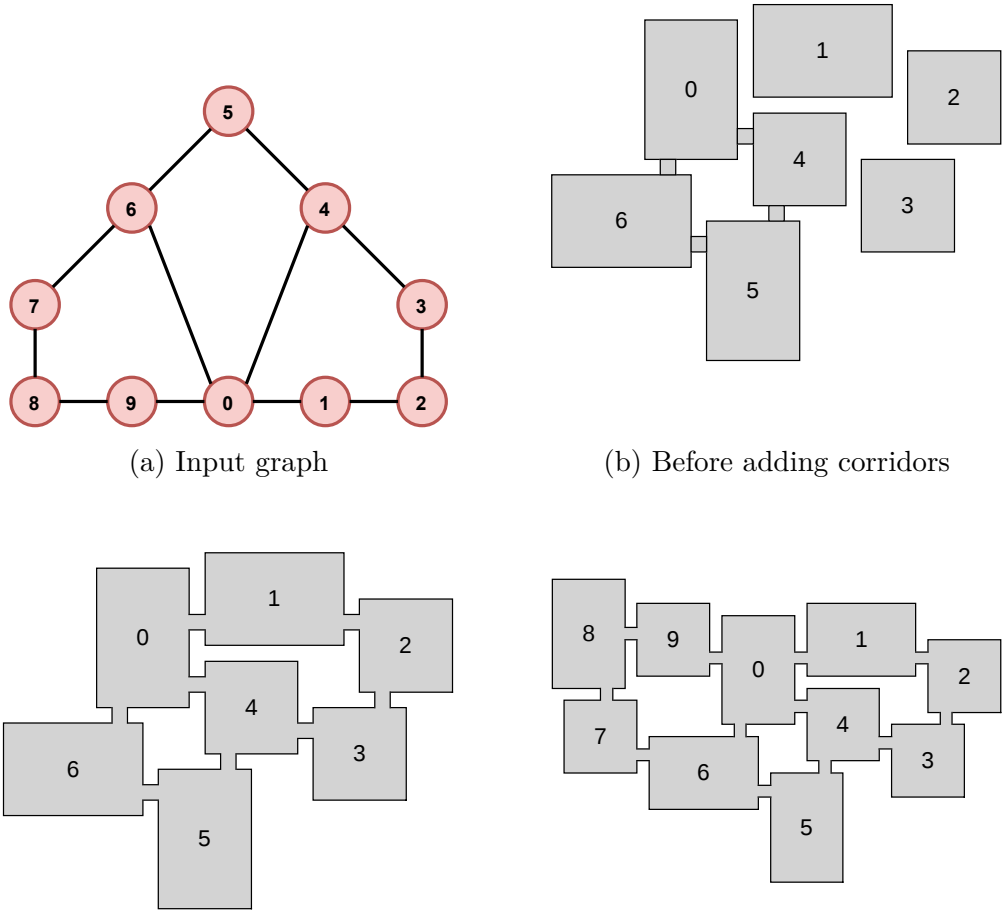
```
1  Input: layout, current chain
2
3  procedure PerturbLayoutWithCorridors(layout, chain)
4    configSpaces ← Get basic precomputed configuration spaces
5    configSpacesCorridors ← Get precomputed configuration spaces that
6      force a small space between each pair of neighbouring rooms
7
8    perturbShape ← Pick at random – 40% true, 60% false
9    nodeToBePerturbed ← Get a random non–corridor node from chain
10
11   if perturbShape then
12     Pick a random shape for nodeToBePerturbed
13   else
14     Use configSpacesCorridors to get a random position from
15     the interesection of configuration spaces of neighbours
16     of nodeToBePerturbed
17   end if
18
19   Update position/shape of nodeToBePerturbed
20
21   if layout is valid then
22     Try to greedily add corridors (from chain) to layout using configSpaces
23
24     if not all corridors were added then
25       Remove all corridors (from chain) from layout
26     end if
27   end if
28
29   return perturbed layout with updated energy
30 end procedure
```

Algorithm 4: Our approach to adding corridors. Red lines show what is different from Algorithm 3.

With this approach, we can quickly generate layouts with rooms connected by short corridors. We observed that the algorithm sometimes converges even quicker when we enable corridors (in terms of iterations count). This is probably

[1]Because of a lack of time, our current implementation only allows users to either choose to have corridors between all rooms in the original graph or to not have corridors at all. However, the proposed approach can be implemented to support choosing exactly which rooms should be connected by corridors.

caused by the fact that it may be easier to lay out non-corridor nodes if they are not required to touch, and because the process of greedily adding corridors has a high success rate.



(a) Input graph

(b) Before adding corridors

(c) After adding corridors

(d) Full layout

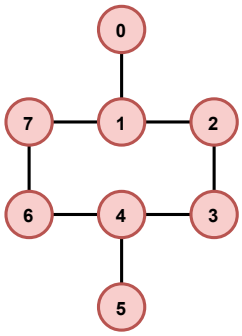
Figure 6: Corridors. (b) shows how is the second type of configuration spaces used to create space between rooms in the second chain. (c) shows how are corridors added to (b). (d) shows a full layout.

2.2.2 Explicit door positions

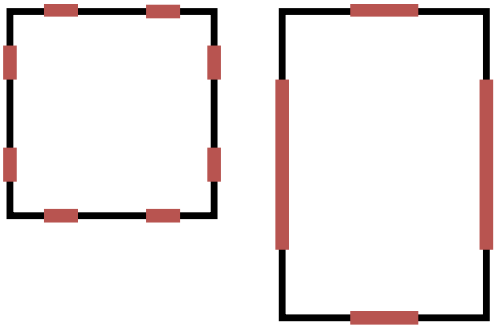
In the original algorithm, it is not easily possible to specify door positions of individual room shapes. It only allows us to configure one global length and that is used for all doors in a layout. The problem is that there are situations where it is convenient to explicitly specify door positions of a room. For example, we may have a boss room and need the player to enter the room from a specified tile. Or we may have multiple room templates and they may have some tiles reserved for walls or other obstacles.

We have implemented our own configuration spaces generator that works directly with door positions. It allows users to explicitly define door positions of

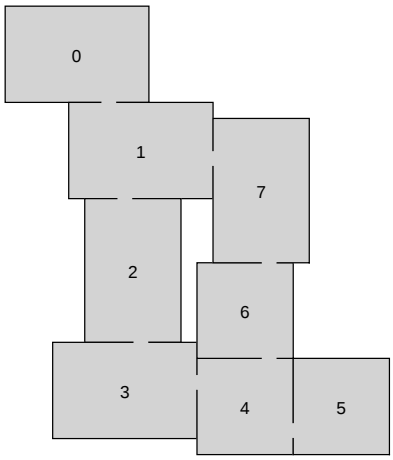
every room shape in a layout. This modification has no runtime overhead because configuration spaces are generated only once before the algorithm starts. However, note that having too few door positions, e.g. only two door positions for a node that should be connected to two neighbours, makes it significantly harder for simulated annealing to connect neighbouring rooms and will often cause the algorithm to need more iterations to generate a valid layout.



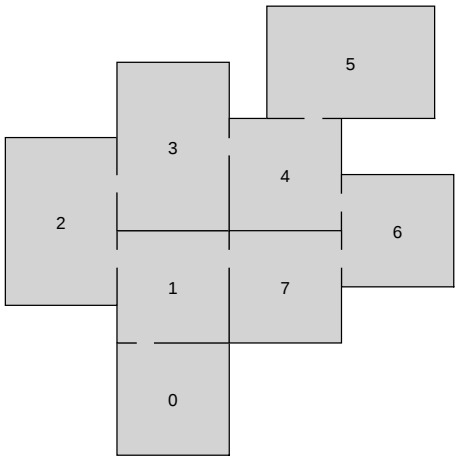
(a) Input graph



(b) Building blocks with their door positions in red



(c) Generated layout



(d) Generated layout

Figure 7: Example of explicitly defined door positions.

2.2.3 Custom constraints

The original method enforces two basic constraints on the layout - no two rooms may overlap and all neighbouring rooms must be connected by doors. We decided to make the concept of constraints more general and customizable.

Our framework allows defining two types of constraints - the first type ensures that the whole layout satisfies some conditions (total area, etc.) and the second type operates on individual nodes (no overlap, etc.). Both types of constraints can be either hard or soft. All hard constraints must be satisfied before a layout can be accepted whereas soft ones are used to control the evolution by modifying the energy, but do not invalidate the layout.

As an example of what we can do with custom constraints, we created one that does not allow non-corridor rooms to share a common wall segment. We use this constraint together with the two basic ones and even though it makes the

convergence rate slightly worse, we find such layouts more visually pleasing and use it to generate layouts with corridors.

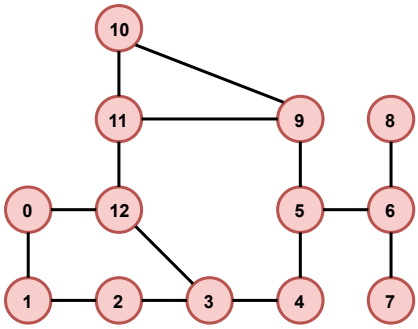
Note that even though this feature allows us to remove the basic two constraints and use completely different ones, it should not be used to do so. The idea is to add constraints that will work well with the existing ones. We can, for example, create a constraint that will make sure that the whole layout does not exceed some defined boundaries or we can create an obstacle that we have to avoid.

2.2.4 Different probabilities for room shapes

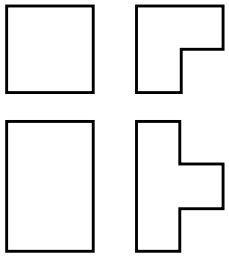
To make it easier for users to define room shapes, we added an option to automatically compute all rotations of a given room shape instead of doing it by hand. However, it introduces a problem with some room shapes being present in a layout more often than others. Suppose we want to generate a layout consisting of square and rectangle rooms and that we wish to have an approximately same number of both types of rooms. The issues is that a square has only one rotation but a rectangle has two rotations. That means that shape perturbations will choose randomly from three shapes and we will often end up with a layout that has approximately two times more rectangles than squares.

To solve this problem, we could just make the algorithm to choose from a set of four shapes where the square shape would be present twice. However, this is not really a solution but rather a workaround. Instead, we decided to add explicit probabilities to every room shape and when perturbing a shape of a node, we pick a shape according to this probability distribution. Not only does it solve the problem, but it also gives us more possibilities to control the look of generated layouts.

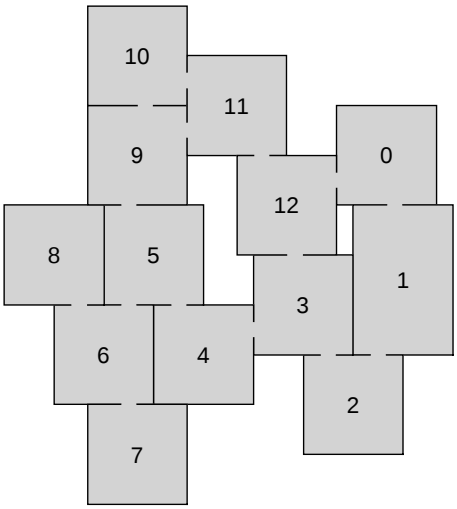
Note that this feature is meant to allow game designers to slightly change the look of produced layouts while not sacrificing the speed of the algorithm. In its current implementation, we cannot guarantee that all generated layouts will obey a defined probability distribution of room shapes. This is because the probability distribution is only a guide for simulated annealing when perturbing shapes of nodes. Another reason is that in the process of selecting the best initial configuration for simulated annealing, we greedily choose the best position and shape, and that may also go against the probability distribution. To overcome these possible biases, we would have to define the distribution of room shapes as a hard constraint and also change the strategy of selecting the initial configuration of a chain. By doing so, we would make the convergence speed significantly worse, which goes directly against the idea of keeping the feature as lightweight as possible.



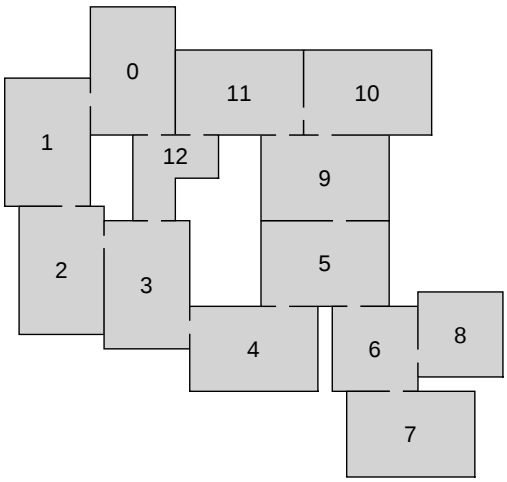
(a) Input graph



(b) Building blocks



(c) Square has greater probability



(d) Rectangle has greater probability

Figure 8: Different probabilities of individual room shapes.

2.3 Performance improvements

In this section, we will provide information about the most important speed improvements of our algorithm. For each such improvement, we provide a benchmark to show how it influences the overall speed of the algorithm. All these benchmarks demonstrate the difference between our initial implementation and the implementation with only the respective improvement enabled. A comparison of the original algorithm, our initial implementation and our final implementation with all the improvements enabled can be seen in the Results section.

2.3.1 Simulated annealing parameters

When investigating how to improve the performance of simulated annealing, we observed that most of the time is spent on runs that either fail to generate anything or do not produce enough partial layouts. Such situations happen mostly if the current chain cannot be laid out because of an unlucky positioning of previous chains (Figure 5). With this in mind, we tried to find ways to terminate non-perspective runs as soon as possible.

The original algorithm uses a mechanism of random restarts. If we do not

accept any state for too long, we quit the current run of simulated annealing. This can be seen in Algorithm 5 (purple lines with *Position c* and red lines).

The problem is that we can accept a lot of states without producing a single valid layout. Generating valid layouts, however, is our main goal. Therefore, we experimented with multiple approaches to random restarting. In Algorithm 5, we present three different positions where it is decided if the current iteration of simulated annealing is successful or not. *Position c* is the original approach where we penalize iterations that fail to accept new states. *Position a* penalizes iterations that fail to produce valid layouts. And finally *Position b* is the most strict one and penalizes iterations that fail to produce valid layouts that are different enough from already generated layouts.

We benchmarked all three possibilities and *Position c* came out as the best one. We also tried various values of parameter m (trials per cycle) and decided to set it to 100 from the original 500. Both these changes show us that, for the overall speed of the method, it is important to try to aggressively terminate non-perspective runs of simulated annealing.

```
1  Input: chain c, initial layout s
2
3  procedure AddChain(c,s)
4    generatedLayouts ← Empty collection of generated layouts
5    failedAttempts ← 0
6    t ← t0                                              // Initial temperature
7
8    for i ← 1,n do                                      // n: # of cycles in total
9      Return if enough failedAttempts
10     iterationSuccessful ← false
11
12     for j ← 1,m do                                    // m: # of trials per cycle
13       s' ← PerturbLayout(s, c)
14
15       if s' is valid then
16         iterationSuccessful ← true (Position a)
17
18         if s' ∪ c is full layout then output it
19         else if s' passes variability test
20           iterationSuccessful ← true (Position b)
21
22           Add s' into generatedLayouts
23           Return generatedLayouts if enough extended layouts computed
24         end if
25       end if
26
27       if ΔE < 0 then                                  // ΔE = E(s') - E(s)
28         s ← s'
29         iterationSuccessful ← true (Position c)
30       else if rand() < e^(−ΔE/(k∗t)) then
31         s ← s'
32         iterationSuccessful ← true (Position c)
33       else
34         Discard s'
35       end if
36
37     end for
38
39     if not iterationSuccessful then
40       failedAttempts++
41     end if
42
43     t ← t ∗ ratio                                     // Cool down temperature
44   end for
45 end procedure
```

Algorithm 5: Several approaches to random restarts.

Table 2 and Figure 9 show the speed difference between the initial implementation and the implementation containing the changes proposed in this section. We achieved a significant speedup for all the input graphs.

Input	**Success rate**	**Time avg/med**	**Iterations avg/med**
Initial implementation:			
Figure 5 (15 vertices)	86%	5.55s/2.65s	323.54k/138.29k
Figure 8 (13 vertices)	100%	0.39s/0.31s	21.29k/17.72k
Figure 22 (9 vertices)	100%	0.21s/0.08s	13.29k/4.13k
Figure 23 (17 vertices)	100%	2.77s/1.98s	137.88k/103.09k
Figure 24 (41 vertices)	44%	20.12s/23.60s	821.29k/1002.13k
Figure 25 (21 vertices)	100%	2.05s/1.75s	146.07k/126.32k
Figure 26 (11 vertices)	100%	1.40s/1.05s	108.12k/78.62k
New approach:			
Figure 5 (15 vertices)	100%	0.33s/0.20s	16.32k/9.54k
Figure 8 (13 vertices)	100%	0.05s/0.05s	1.60k/1.42k
Figure 22 (9 vertices)	100%	0.02s/0.02s	0.79k/0.74k
Figure 23 (17 vertices)	100%	0.25s/0.21s	9.24k/7.77k
Figure 24 (41 vertices)	99%	1.76s/0.97s	67.67k/32.89k
Figure 25 (21 vertices)	100%	0.13s/0.08s	8.30k/4.80k
Figure 26 (11 vertices)	100%	0.07s/0.05s	3.89k/3.21k

Table 2: Benchmark of our changes of simulated annealing. The benchmark was run 100 times for each input graph. Success rate shows how often was the algorithm able to generate a valid layout. We provide average and median values for both the time and the number of iterations.

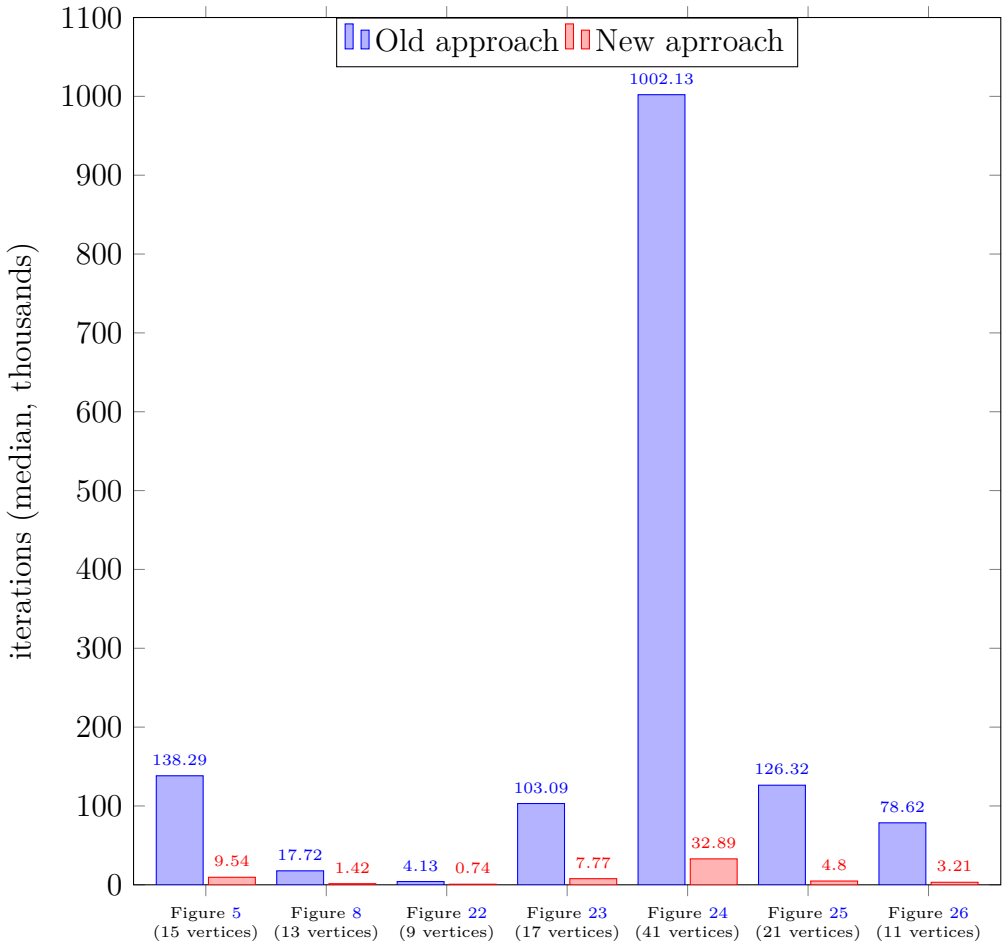


Figure 9: Benchmark of our changes of simulated annealing. The plot shows the median number of iterations from Table 2. Fewer is better.

2.3.2 Chain decomposition

One of the main ideas of the original algorithm is that the process of generating layouts is incremental. A given input graph is decomposed into smaller sets of nodes (called chains) and simulated annealing starts with an empty layout and tries to extend it with one chain at a time. The quality of the chain decomposition plays a crucial role in the speed of the whole algorithm.

Our first implementation of the decomposition simply followed the strategy from the original algorithm that we described in the Incremental layout section. However, it was not very successful.

After some experiments, we came up with Algorithm 6. The basic idea is still the same - we compute faces of an arbitrary planar embedding of the input graph and use them to form the base of the chain decomposition. We create the first chain from the smallest face and then add the remaining faces in the depth-first search ordering. The only face we exclude from this process is the largest one (line 6). The motivation behind this is that the largest face usually contains too many vertices to be considered to form a single chain. An extreme example of this situation is any graph that is a tree, where the largest faces contains all vertices. The consequence of doing that is that we will often end up with vertices that are not contained in any face. The strategy for dealing with such vertices is that

we always pick one that neighbours with an already covered vertex[2] and use it as the first vertex of a random path from which we form a chain (lines 19 - 35). We terminate the path as soon as we encounter a vertex that is contained in a not yet processed face (lines 28 - 30). By doing so, we prioritize creating chains from faces rather than from long paths, which proved to be a better approach. Throughout the pseudocode, the depth of chains is used to guide the algorithm. It leads to a more uniform distribution of chains and, in our experience, prevents some cases of backtracking in the later phases of the incremental layout. This implementation proved to be quite successful but we decided to further explore how it behaves on various types of graphs.

```
1  Input: planar graph G
2
3  procedure DecomposeGraphIntoChains(G)
4    C ← empty list of chains
5    faces ← get faces from a planar embedding of G
6    Remove the largest face from faces
7    depth ← 0
8
9    repeat
10     if C is empty then
11       face ← the smallest face in faces
12       Remove face from faces
13       Add face to C, set its depth to depth
14     else if there is a face that neighbours with any chain in C then
15       face ← face that neighbours with a chain with the smallest depth in C
16       Remove face from faces
17       Add face to C, set its depth to depth
18     else
19       v ← uncovered vertex that neighbours with a chain with
20        the smallest depth in C
21       chain ← create an empty list of vertices
22       Add v to chain
23
24       while v has uncovered neighbours do
25         v_new ← pick a random uncovered neighbour of v
26         Add v_new to chain
27
28         if v_new is contained in a not processed face then
29           break
30         end if
31
32         v ← v_new
33       end while
34
35       Add chain to C, set its depth to depth
36     end if
37
38     depth++
39   until all nodes of G are contained in some chain in C
40
41   return chain decomposition C
42 end procedure
```

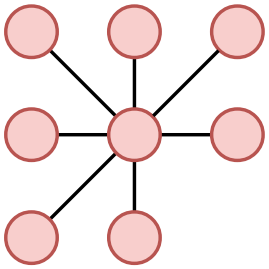
Algorithm 6: Chain decomposition.

We observed that having a lot of small chains in a decomposition is quite bad, mainly in a situation where we have to backtrack very often. The problem is that a substantial amount of time is spent when initializing the process of laying out the next chain. For example, it is quite time-consuming to find the best initial configurations for nodes in the current chain (see the Simulated annealing section).

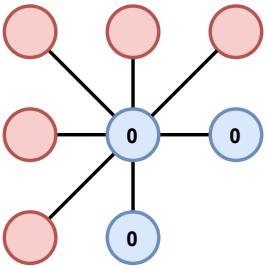
In Figure 10a we can see a prototype of a problematic graph - note that it

[2]A vertex that is already contained in the decomposition.

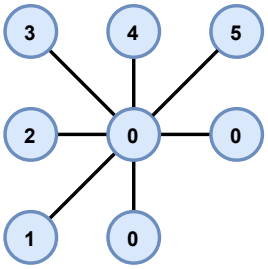
has a lot of nodes with only a single neighbour. Figure 10b shows how would the decomposition look like after the first iteration of the algorithm above. We can see that we now have a lot of small acyclical components. The problem is that the algorithm creates a new chain from every such component. And finally in Figure 10c we can see that we will end up with a lot of small chains - which is a situation we want to avoid.



(a) Input graph



(b) Partial chain decomposition



(c) Complete chain decomposition

Figure 10: Chain decomposition. Blue nodes are contained in a chain with a corresponding number. (a) shows the input graph. (b) shows a partial chain decomposition after the first iteration of Algorithm 6. (c) shows a complete chain decomposition of the graph.

Our solution is quite simple. When we want to add a node to a chain, we check if it does not create acyclical components with only one node. If it does, we add all such components to the current chain. By doing so, we often do not stick to the definition of a chain because we allow a node to have more than two neighbours. However, this approach behaves, in our experience, better that the original one.

Table 3 and Figure 11 show the speed difference between the initial implementation and the implementation containing the changes proposed in this section. The most significant speedup was achieved on the input graphs found in Figure 24 and 25. That is because both these graphs contain several occurrences of the pattern that caused the decomposition to have too many chains. The speed of other inputs either improved or remained approximately the same.

We also provide a specialized implementation that can be used when generating layouts with corridors. If corridors are enabled, we have a graph with a new node added between every two nodes that were neighbours in the original graph. The problem is that corridor nodes are less important than non-corridors nodes

and we do not want them to deform our decomposition. Therefore, our goal is to get a decomposition that has non-corridor nodes divided into the same chains as if we decomposed the original graph without corridors. To do that, we first remove all corridors from the graph, run our classic decomposition algorithm and then add all corridors back to the corresponding chains.

Input	**Success rate**	**Time avg/med**	**Iterations avg/med**
Initial implementation:			
Figure 5 (15 vertices)	86%	5.55s/2.65s	323.54k/138.29k
Figure 8 (13 vertices)	100%	0.39s/0.31s	21.29k/17.72k
Figure 22 (9 vertices)	100%	0.21s/0.08s	13.29k/4.13k
Figure 23 (17 vertices)	100%	2.77s/1.98s	137.88k/103.09k
Figure 24 (41 vertices)	44%	20.12s/23.60s	821.29k/1002.13k
Figure 25 (21 vertices)	100%	2.05s/1.75s	146.07k/126.32k
Figure 26 (11 vertices)	100%	1.40s/1.05s	108.12k/78.62k
New approach:			
Figure 5 (15 vertices)	98%	2.93s/.95s	174.00k/50.72k
Figure 8 (13 vertices)	100%	0.39s/.30s	21.45k/17.05k
Figure 22 (9 vertices)	100%	0.19s/0.08s	12.38k/4.20k
Figure 23 (17 vertices)	100%	1.72s/1.16s	83.65k/51.41k
Figure 24 (41 vertices)	98%	6.64s/5.06s	235.72k/190.77k
Figure 25 (21 vertices)	100%	0.20s/0.04s	10.21k/1.84k
Figure 26 (11 vertices)	100%	1.70s/0.77s	122.44k/54.24k

Table 3: Benchmark of our changes of chain decomposition. The benchmark was run 100 times for each input graph. Success rate shows how often was the algorithm able to generate a valid layout. We provide average and median values for both the time and the number of iterations.

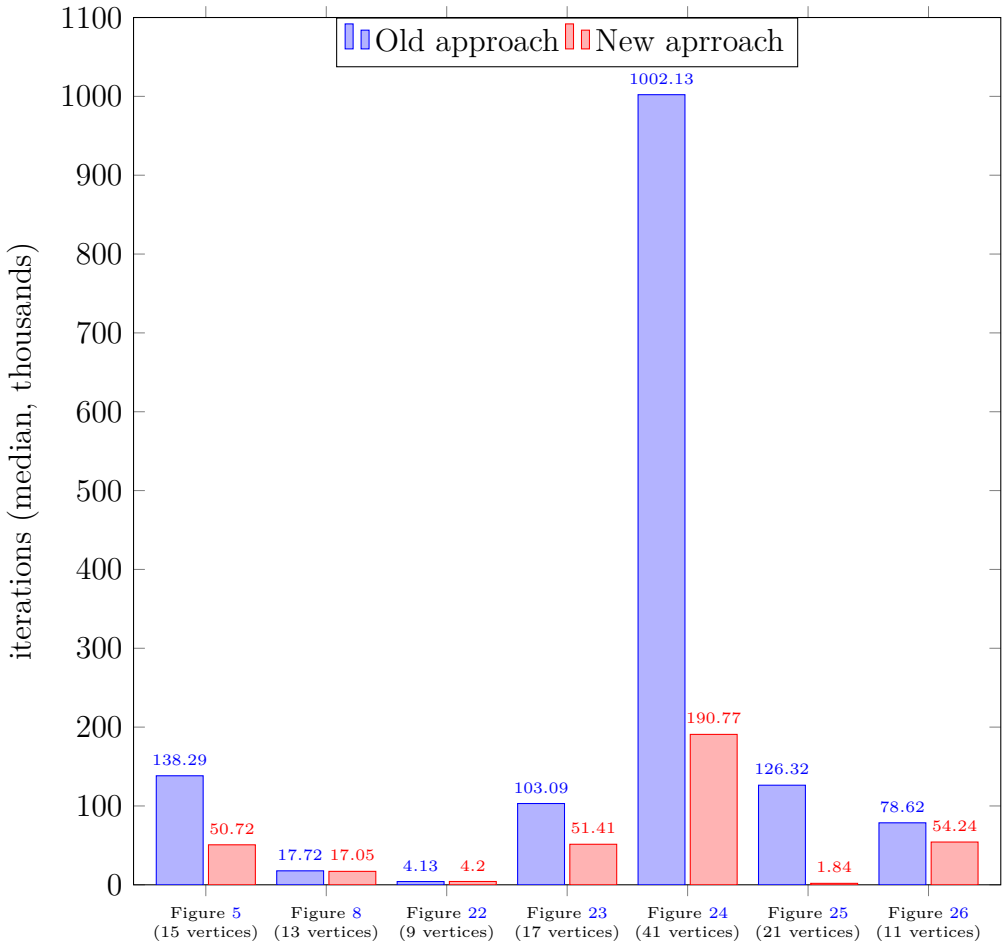


Figure 11: Benchmark of our changes of chain decomposition. The plot shows the median number of iterations from Table 3. Fewer is better.

2.3.3 Lazy evaluation

In each run of simulated annealing, we try to generate multiple layouts in case we need to backtrack later. But what if we are lucky and do not need to backtrack? In that case, we have wasted a lot of time by computing something that is not really needed.

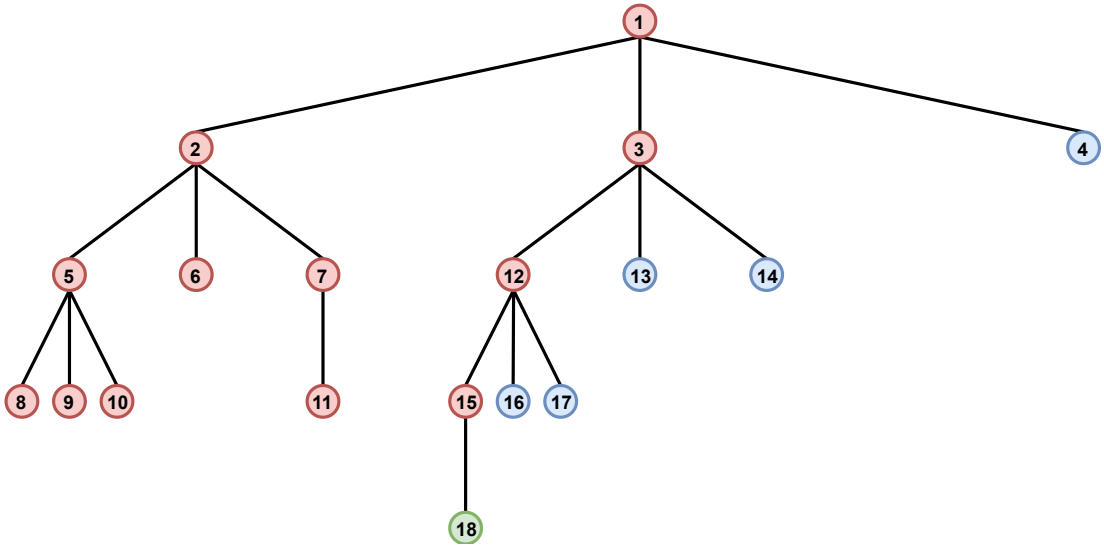
We can imagine that the algorithm builds a tree where each node represents a generated partial layout and all children of a node are layouts that were generated from the parent node. That means that the depth of a node corresponds to the number of its already laid out chains (with the root node being in the depth 0). With this representation, the goal of the algorithm is to find a node that is in a depth that equals to the total number of chains.

Figure 12a depicts a run of the original algorithm on an input graph that has 4 chains. All nodes in the tree correspond to a layout that was generated in simulated annealing. The green node is a final layout that was generated. Blue nodes are nodes that were not yet expanded. All nodes are numbered to represent the order in which they were generated.

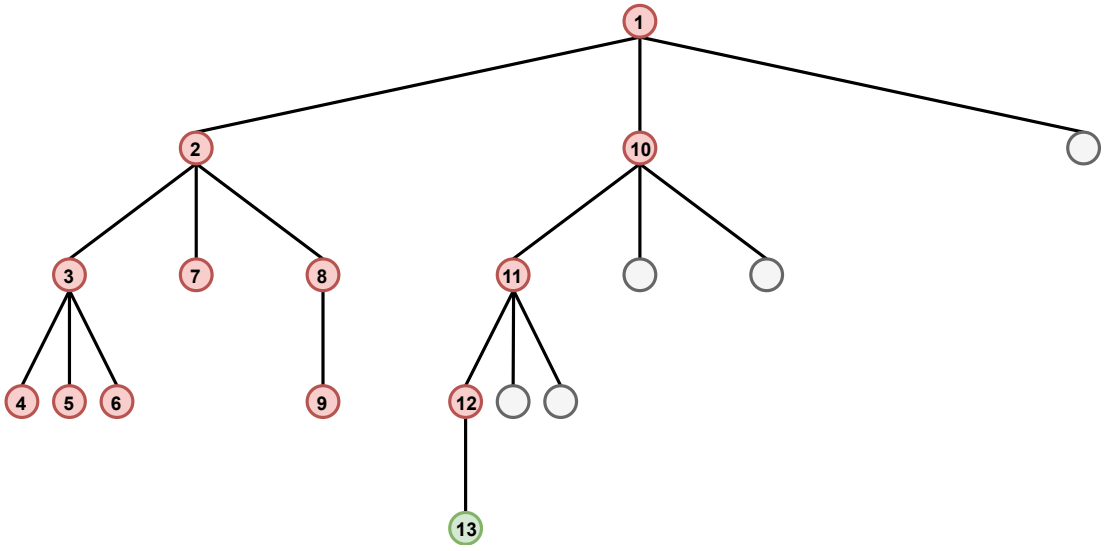
The problem is that the original algorithm always generates all children of a node before moving to another node. Figure 12a shows that in order to generate a valid full layout, we had to generate 5 partial layouts that were never used. In

fact, the original algorithm generates up to 10 layouts from every node in the tree and we would, therefore, often end up with significantly more than 5 unused layouts.

Fortunately, C# makes it quite easy to transform any algorithm to a lazy one with the `yield return` keyword. Instead of generating all children nodes at once, we save the state of the current run of the algorithm and resume it later only if it is really needed. Figure 12b shows a run of our algorithm after this modification.



(a) Without lazy evaluation.



(b) With lazy evaluation.

Figure 12: Tree representation of the generation process. Each node represents a generated layout. (a) shows the original method and (b) shows our method with lazy evaluation.

Table 4 and Figure 13 show the speed difference between the initial implementation and the implementation containing the changes proposed in this section.

The most significant speedup was achieved on quite simple input graphs. This is caused by the fact that on these inputs the algorithm almost never backtracks. Therefore, every partial layout, that is generated and not used, has a great impact on the overall speed.

Input	**Success rate**	**Time avg/med**	**Iterations avg/med**
Initial implementation:			
Figure 5 (15 vertices)	86%	5.55s/2.65s	323.54k/138.29k
Figure 8 (13 vertices)	100%	0.39s/0.31s	21.29k/17.72k
Figure 22 (9 vertices)	100%	0.21s/0.08s	13.29k/4.13k
Figure 23 (17 vertices)	100%	2.77s/1.98s	137.88k/103.09k
Figure 24 (41 vertices)	44%	20.12s/23.60s	821.29k/1002.13k
Figure 25 (21 vertices)	100%	2.05s/1.75s	146.07k/126.32k
Figure 26 (11 vertices)	100%	1.40s/1.05s	108.12k/78.62k
With lazy evaluation:			
Figure 5 (15 vertices)	88%	4.43s/2.51s	295.75k/141.64k
Figure 8 (13 vertices)	100%	0.14s/0.03s	8.49k/0.51k
Figure 22 (9 vertices)	100%	0.06s/0.02s	4.14k/0.16k
Figure 23 (17 vertices)	100%	2.21s/1.93s	109.87k/96.73k
Figure 24 (41 vertices)	40%	19.58s/23.43s	817.63k/1001.60k
Figure 25 (21 vertices)	98%	0.43s/0.05s	25.66k/0.35k
Figure 26 (11 vertices)	100%	0.73s/0.26s	56.40k/25.11k

Table 4: Benchmark of using and not using lazy evaluation. The benchmark was run 100 times for each input graph. Success rate shows how often was the algorithm able to generate a valid layout. We provide average and median values for both the time and the number of iterations.

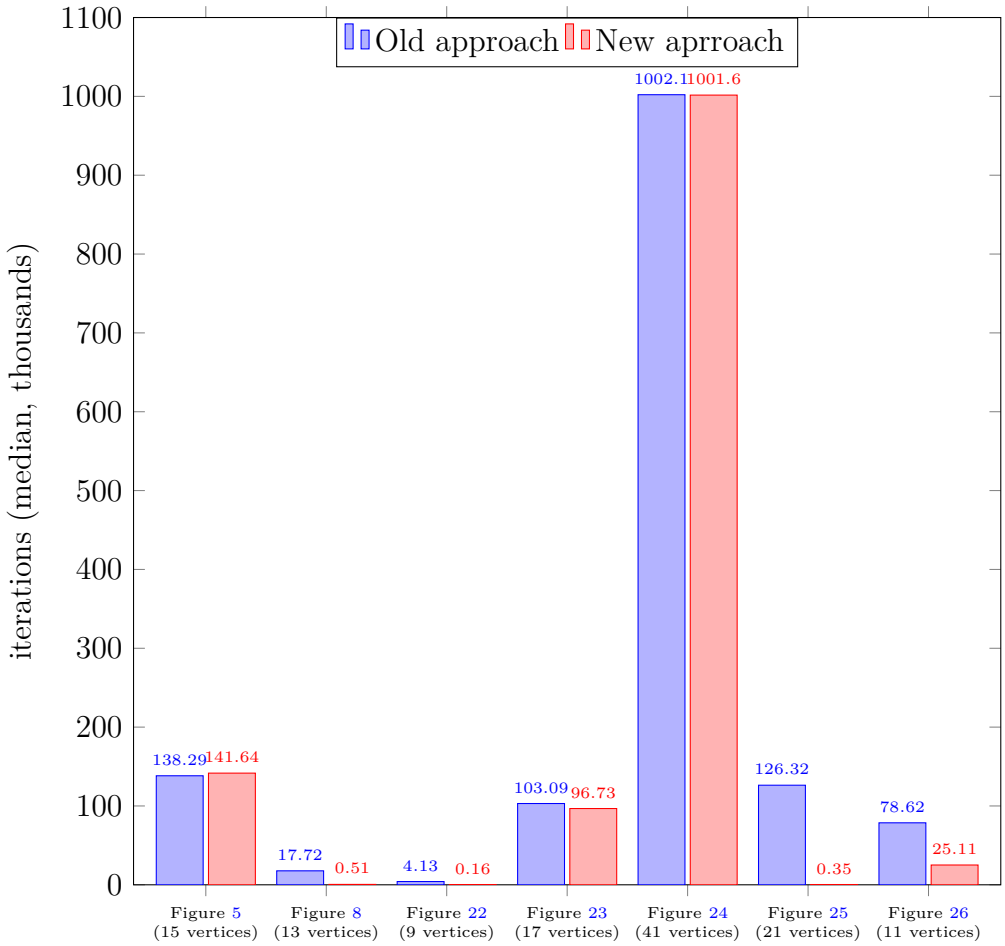


Figure 13: Benchmark of using and not using lazy evaluation. The plot shows the median number of iterations from Table 4. Fewer is better.

3. Framework

One of the goals of this thesis is to provide a framework that will allow programmers to replace or extend individual components of our layout generator. In this section, we will first analyze needs of the framework and then describe its architecture and used technologies.

3.1 Analysis

3.1.1 Extensibility

One of the main goals of the framework is to be extensible. The minimum requirement is that a programmer should be able to replace or extend individual building blocks of our method. It would be also convenient to provide an infrastructure that could be used to implement similar methods of procedural layout generation.

The framework should support:

- replacing the stochastic method that is used to lay out individual chains
- changing the strategy of chain decomposition
- changing the process of generating configuration spaces of pairs of nodes

C# makes it possible to write such extensible code, primarily by using interfaces, inheritance and generics. The problem with generics is that types with too many generic parameters are hard to be used by programmers, especially if there are multiple constraints on the parameters.

Among all the decisions of what is worth being implemented as a generic type, the hardest one was whether we should support switching between integer and real coordinates. The main goal of this thesis is to implement an algorithm that will generate tile-based layouts, which is equivalent to using integer coordinates. However, most components of the algorithm do not care about the type of coordinates and could be directly used in both contexts.

In the end, we decided to support only integer coordinates. Supporting both types of coordinates would, in our opinion, make a lot of code unnecessarily complex and unreadable.

3.1.2 Input format

One of the goals of the framework is to provide a GUI for the layout generator. We want the GUI to be controlled by config files because the GUI itself would need to be too complex if it had to support defining level connectivity graphs and creating room shapes.

There exist a lot of formats for config files, popular ones being for example JSON, YAML or XML. We want the format to be easily writable and readable by human and there must also exist a .NET library that can parse it.

We decided to use YAML format because it fulfils all mentioned requirements. The syntax is, in our opinion, quite simple but also able to represent complex

types. It is also very readable and there are several parsers implemented for .NET. We chose to use the YamlDotNet library[2] because it seems to be actively maintained and supports everything we need.

3.1.3 Output format

The framework should support both generating layouts directly in a game and pregenerating them for later use. In the former case, we will probably consume the runtime representation of generated layouts, whereas in the latter case we must be able to somehow store the output.

In contrast with the input format, we do not have to care too much about human readability of the format. Therefore, we decided to use JSON because it is quite well-known and supports all needed data structures. There is also a very popular .NET library called Json.NET[13] that supports both serialization and deserialization of JSON files.

3.1.4 Planar graphs

To decompose a graph into chains, we need to check if the graph is planar and then construct a planar embedding to get its faces. Several efficient algorithms exist for both the planarity check and constructing a planar embedding.

In our method, we do not need any custom behaviour from these algorithms and therefore finding a third-party library seems to be the right choice. Unfortunately, there is no usable .NET library that implements both algorithms. Another possibility is to use a C++ library and call it from C# using the P/Invoke mechanism.

We decided to use the C++ Boost[3] library because it is peer-reviewed, portable and provides both needed algorithms.

3.1.5 Polygon geometry

To compute configuration spaces, we must be able to do basic operations with polygons. These operations include computing whether two polygons intersect or whether they share a common part of a wall segment. For this purpose, authors of the original method use Clipper[10] library that is also available as a .NET library.

We decided to implement our own polygon operations because our method supports only rectilinear polygons which are much easier to implement. And we can also optimize it for our needs. For example, it lets us connect room shapes by doors instead of working only with sides of a polygon.

3.1.6 Benchmarks

One of the goals of this thesis is to improve speed of the original algorithm. To compare speed after various modifications, we must implement an automated benchmarking framework. Moreover, because our method uses probabilistic techniques, we must be able to run the algorithm multiple times to get meaningful results.

3.2 Used technologies

The majority of the framework is implemented in C# on .NET platform. The main advantage of this technology stack is that we can use the generator directly in Unity[19] which is one of the most popular platforms to develop games on[18]. It also allows us to quickly develop a simple GUI using Windows Forms[12] library. A small portion of the library is written in C++ to delegate some tasks to native C++ Boost library[3].

Used 3rd party libraries:

- Boost[3] - Planarity checks and planar embeddings of graphs
- Newtonsoft.JSON[13] - JSON serialization
- nUnit[15] - Unit testing
- RangeTree[4] - Range tree data structure
- YamlDotNet[2] - Yaml deserialization

3.3 Solution structure

The solution is divided into several projects. The most important one is the MapGeneration project that contains the layout generator itself.

- BoostWrapper - C++ wrapper for the Boost library[3]. Handles planar embeddings of graphs and planarity checks.
- GUI - GUI for the layout generator. Controlled by config files. Built on the Windows Forms library[12].
- GeneralAlgorithms - General purpose algorithms and data structures that are used in the generator.
- GeneralAlgorithms.Tests - Unit tests for GeneralAlgorithms.
- MapGeneration - DLL project that contains the layout generator with all its building blocks.
- MapGeneration.Interfaces - Interfaces for MapGeneration.
- MapGeneration.Tests - Unit tests for MapGeneration.
- Sandbox - Executable application that can be used to do benchmarks or play with the generator.

3.4 Data structures

In this section, we will describe the most important data structures that are used throughout the framework. We will start with the data structures that are used to describe the input for the generator, then continue with the ones that are used in the generation process itself and finish the section with the representation of the output. The framework contains implementations of all interfaces mentioned in this section.

3.4.1 IMapDescription interface

The `IMapDescription` interface represents a description of a layout that we want to generate. The interface itself contains only a method that returns the underlying level connectivity graph because that is the only general information that is needed from the interface. Other information, e.g. a list room shapes and their probabilities, are retrieved directly from implementing classes and used for example in a configuration spaces generator.

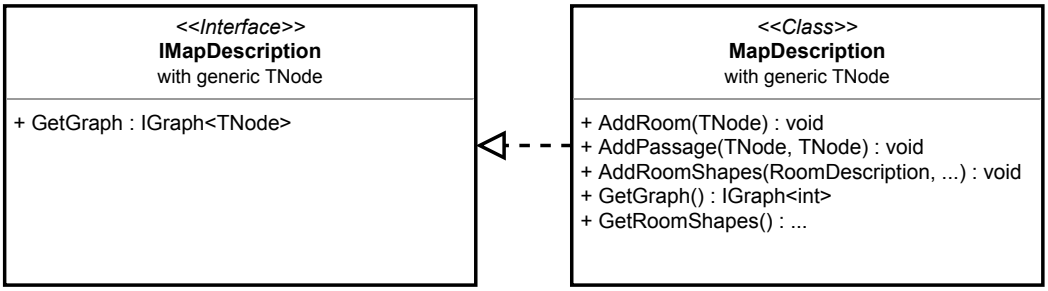


Figure 14: UML diagram of the `IMapDescription` interface and the `MapDescription` class.

3.4.2 ILayout interface

The `ILayout` interface represents a layout that is used in the generation process. The layout contains a reference to the input graph and information about configurations of all nodes in the graph. The interface can also be extended with the `IEnergyLayout` interface that adds information about the energy of the layout which is used to control the evolution process.

It is important to make sure that the methods for getting and setting configurations (Figure 15) are as fast as possible because these methods are called very frequently. Typical implementation would use a hash table with keys being nodes of the graph. Nevertheless, hash table access can be quite slow compared to an array access. Therefore, our provided implementation of the interface works only with nodes that are integers. By doing so, we can implement these getters and setters with a simple array access which is really fast. However, that does not mean that we do not support different types of nodes. The trick is to first map all nodes to a sequence of integers, evolve the layout with integer nodes and then map these integers back to the original type when returning the result.

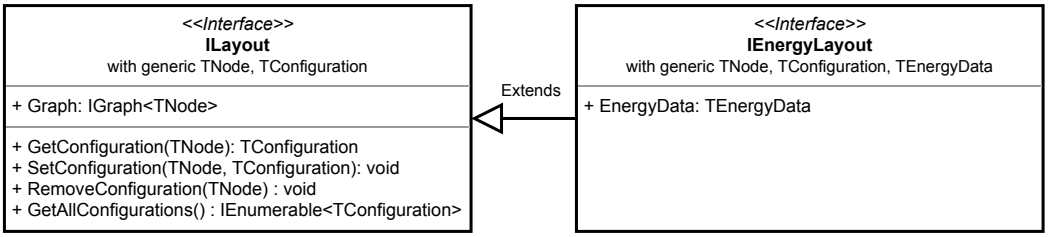


Figure 15: UML diagram of the `ILayout` interface and the `IEnergyLayout` interface.

3.4.3 IConfiguration interface

The `IConfiguration` interface represents the configuration of a node in the input graph. The configuration consists of a shape of the node, its position and a boolean information whether the configuration is valid. In its basic version, all fields of the interface are readonly. This is because several parts of our algorithm should not be able to modify any configurations. `IMutableConfiguration` interface must be used to gain access to the setters. The interface can also be extended with the `IEnergyConfiguration` interface which adds a field with data about the energy of the node. The energy is used to control the evolution process and to determine whether the configuration is valid or not.

In Figure 16 we can see that the interface contains a generic argument called `TShapeContainer`. This allows us to pass additional information together with the shape. We use it to pass an integer id of the shape which we use to avoid hash tables when working with polygons.

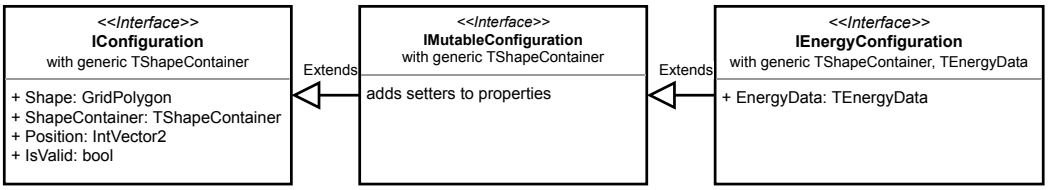


Figure 16: UML diagram of the `IConfiguration` interface, the `IMutableConfiguration` interface and the `IEnergyConfiguration` interface.

3.4.4 IMapLayout and IRoom interfaces

The `IMapLayout` interface is a representation of a layout that is returned by the layout generator. In contrary to `ILayout` interface, this representation is meant to be convenient for programmers rather than optimized to be as fast as possible. It contains a list of rooms that are represented by the `IRoom` interface. The `IRoom` interface shares the same philosophy and contains all information that would be useful when using the library in a game.

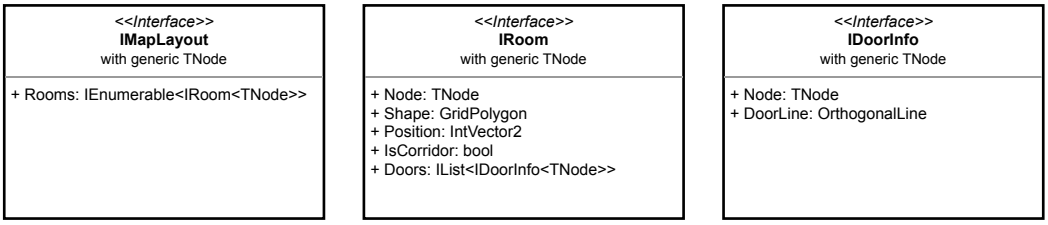


Figure 17: UML diagram of the `IMapLayout` interface, the `IRoom` interface and the `IDoorInfo` interface.

3.5 Algorithms

In this section, we will describe the most important algorithms that are used throughout the framework. We will start with a high-level description of the layout generator itself and then continue with descriptions of individual algorithms that are used in the generator.

3.5.1 ChainBasedGenerator class

The `ChainBasedGenerator` class is the layout generator itself. The class contains almost no logic - it just connects individual building blocks of the whole algorithm. For each such building block, we have a setter method that is used to inject it into the generator. By doing so, we get a very flexible architecture and can easily extend the algorithm without modifying the code of the generator itself.

In Figure 18 we can see a flowchart describing the overall flow and responsibilities of individual building blocks. The `IGeneratorPlanner` part of the figure is very simplified in order to show the flow of the algorithm - see the actual implementation in the IGeneratorPlanner interface section.

The generator is composed of components which are represented by individual interfaces. All these components are therefore easily replaceable.

Individual components

- `IChainDecomposition` - decomposes a given input graph into chains
- `ILayoutEvolver` - adds another chain to a given layout
- `ILayoutOperations` - locally perturbs a layout and updates its energy
- `IConfigurationSpaces` - implements the concept of configuration spaces
- `IGeneratorPlanner` - decides which layouts should be further expanded
- `ILayoutConverter` - converts a layout to a representation that is more convenient for users of the library

3.5.2 IChainDecomposition interface

The `IChainDecomposition` interface represents an algorithm that is used to decompose a given planar graph into chains. A description of the algorithm used in our implementation can be found in the Algorithm section.

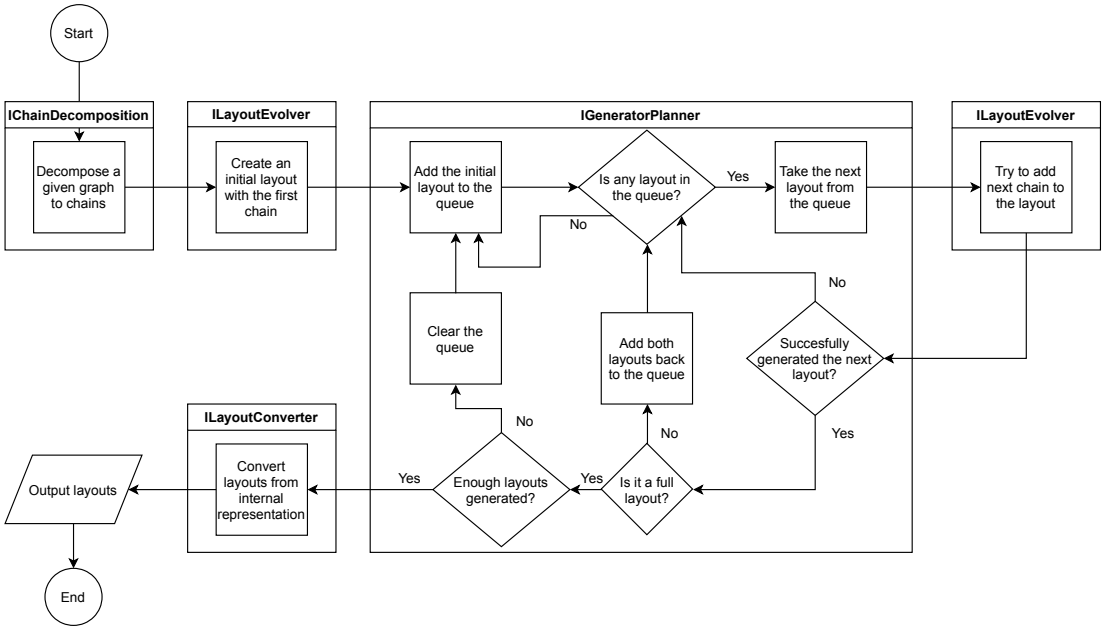


Figure 18: Flowchart of the `ChainBasedGenerator` class.

3.5.3 ILayoutEvolver interface

The `ILayoutEvolver` interface represents an algorithm that is used to evolve valid layouts from a given initial layout. Together with the initial layout, the algorithm also receives a collection of nodes that it is allowed to perturb to find a valid layout.

Layout evolvers should terminate as soon as possible if they find themselves in a situation where it is unlikely to quickly produce valid layouts. That means, that although there is a parameter that specifies how many layouts should be generated, it is possible to return less layouts or even no layouts at all. This is because the initial layout may be in a state where it is very hard to lay out the current chain without modifying nodes from already laid out chains. In this case, we want to quickly terminate the process and move to a different initial layout.

The main method of the interface is prepared to be implemented lazily. The return type is `IEnumerable<TLayout>` which can be easily combined with the `yield return` keyword to create a lazy implementation of the evolver. Lazy evaluation can have a huge impact on the overall convergence speed because generator planners (see the IGeneratorPlanner interface section) are able to make optimizations without generating more partial layouts than they really need.

Layout evolvers should not directly modify the layout or change its energy. Instead, we advise to use the `ILayoutOperations` interface. This is because we want to separate the evolution process itself from the logic of polygon manipulation and energy computing. By doing so, we can easily implement almost any evolution technique in its general form without worrying about in which context it will be used.

SimulatedAnnealingEvolver class

We provide an implementation of simulated annealing as described in Simulated annealing section and in Simulated Annealing Tutorial[8].

Extensibility

It would be interesting to compare simulated annealing to other optimization algorithms. These should be relatively easy to implement because they can use any implementation of the `ILayoutOperations` interface to handle layout perturbations and energy computing.

3.5.4 IConfigurationSpaces interface

The `IConfigurationSpaces` interface represents a data structure that holds configuration spaces as described in the Configuration spaces section. The interface provides methods to check whether a configuration of a node is in a configuration space of another node; to get a random point in an intersection of multiple configuration spaces; to get a random shape for a node and other.

ConfigurationSpaces class

We provide an implementation that can handle different probabilities for individual room shapes and different building blocks for individual nodes.

In our first implementation of this class, there were a lot of hash tables with keys being individual room shapes. The problem is that configuration spaces are used very frequently and indexing with polygon room shapes is much slower than a simple array access. Therefore, we decided to assign an integer alias to each room shape and replace all hash tables with simple arrays. By doing so, we significantly improved the speed of all operations on configuration spaces.

ConfigurationSpacesGenerator class

The `ConfigurationSpacesGenerator` class is used to create an instance of the `ConfigurationSpaces` class. It computes configuration spaces of all pairs of room shapes.

If we want two polygons to be connected by doors, we must make them touch along a common parallel edge. We pick a reference point on one of these polygons and as this polygon slides along this edge, the reference point traces a line segment. By repeating this process for each pair of parallel edges, we get a set of line segments which forms the configuration space of the two polygons (Figure 2).

If we want to support defining explicit door positions, we have to slightly change this procedure. Instead of working directly with edges of polygons, we replace them with specified door lines of each polygon. In Figure 7b we can see an example of such door lines (in red).

3.5.5 ILayoutOperations interface

The `ILayoutOperations` interface provides methods to locally perturb a given layout and update its energy. The basic idea of layout perturbation can be seen in Algorithm 3.

Energy computing

When evolving a layout, updating its energy is probably the most computationally expensive operation. It involves checking if no two nodes overlap and that a node is contained in the configuration spaces of all its neighbours. Both these checks need to use a polygon geometry and are not very cheap. As a result, the speed of the stochastic method is heavily influenced by how fast we can perturb a layout and update its energy.

The most straightforward way to compute the energy of a layout is to iterate through all pairs of nodes and check all constraints. This will, however, result in a quadratic time algorithm (with respect to the number of nodes). Instead, we should exploit the fact that we are always perturbing only one node at a time.

The first thing we must do to get a linear time algorithm is to store energies of individual nodes and compute the overall energy of a layout as a sum of these energies. If we stored only the total energy of a layout, we would probably always end up with a quadratic time algorithm as we would have to recompute everything. The second thing to do is to compute how the energy of each node changes with respect to the perturbed node, which can be done in linear time. After doing that, it is only a matter of applying the change to the stored energy of each node to get an updated energy value.

This optimization gives us a significant speedup and can be found in the `LayoutOperationsWithConstraints` class.

Constraints

We have two main constraints on a layout - no two nodes may overlap and all neighbouring nodes must be connected by doors. However, we decided not to hardcode them directly in the algorithm. Instead, we created two interfaces - `INodeConstraint` and `ILayoutConstraint` - that allow us to define any number of custom constraints without modifying the layout generator itself. The first interface defines conditions for individual nodes, e.g. not overlapping other nodes. If these conditions are not satisfied, positive energy is added to that node. The second interface defines constraints for the layout itself, e.g. not exceeding a user-defined area. And similarly, positive energy is added to the layout if such constraint is not satisfied.

It is also important to note that the energy of a layout is not a factor when deciding if the layout is valid. Instead, the layout is valid if and only if no registered constraint explicitly states that the layout is invalid. This allows us to define both hard and soft constraints. Hard constraints invalidate the layout and add positive energy whereas soft constraints only manipulate the energy, thus making it possible to further control the evolution process.

3.5.6 IGeneratorPlanner interface

In the Lazy evaluation section, we described that the layout generator implicitly builds a tree-like structure (Figure 12) with nodes being valid partial layouts and children nodes being partial layouts generated from the parent layout (by adding the next chain). The `IGeneratorPlanner` interface represents an algorithm that is used to control how is the tree built.

BasicGeneratorPlanner class

We provide a basic implementation of the `IGeneratorPlanner` interface that behaves exactly as described in the Lazy evaluation section. It always picks a node on the deepest level of the tree (the one with the maximum number of chains) and tries to expand it.

We limit the maximum number of layouts that can be generated from a single node. If a node reaches that limit, we no longer consider it when choosing nodes to expand. The reason for this is that we need to explore the space of possible layouts and not just exploit already generated layouts. The limit is now set to be 5 layouts from every parent layout.

Extensibility

This interface provides various possibilities to extend the behaviour of the generator. For example, it is possible to adaptively change the maximum number of layouts that are generated from every node. Or we can create a more sophisticated strategy of choosing which node should be extended. And this is also the place to make the algorithm multithreaded by building the tree simultaneously on multiple threads.

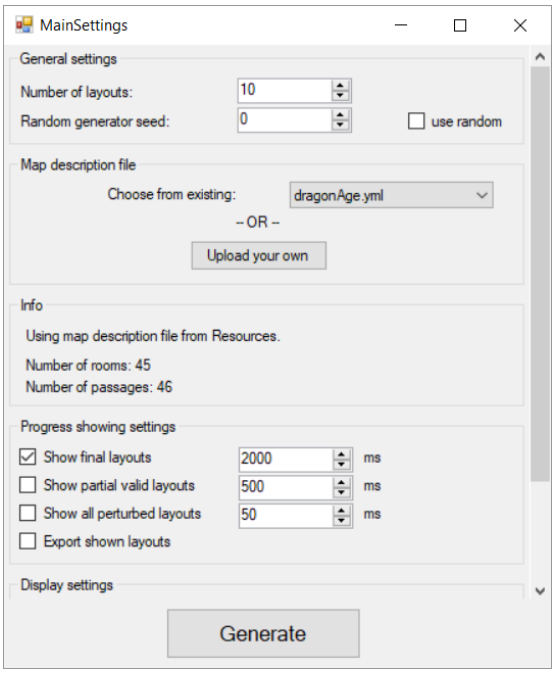
3.5.7 ILayoutConverter interface

The `ILayoutConverter` interface represents an algorithm that is primarily used to convert layouts from a representation that is used in the layout generator to a representation that is easily consumed by users of the library.

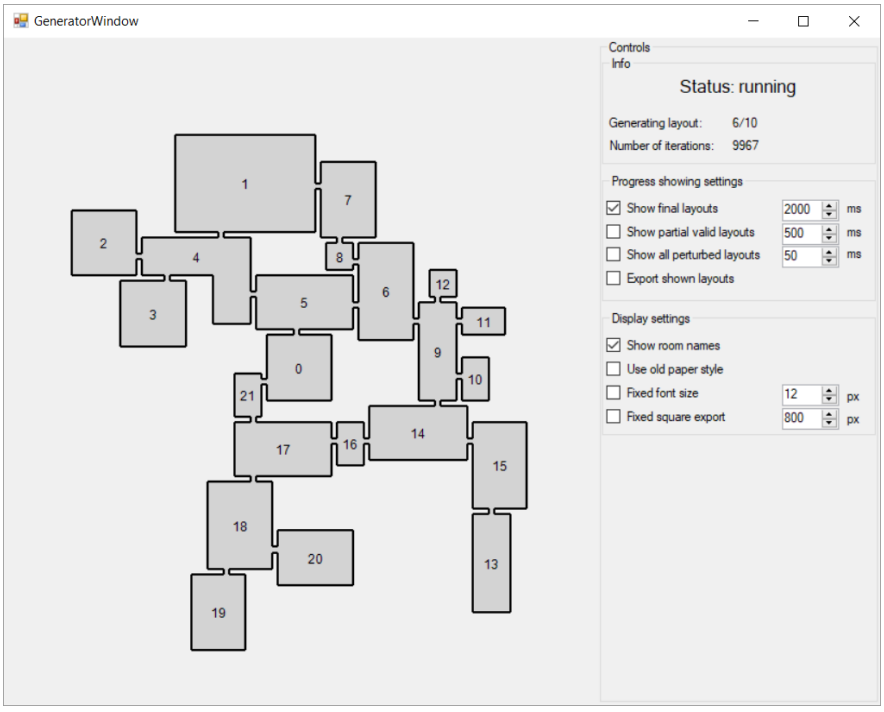
The framework contains an implementation that can convert layouts from the `ILayout` interface to the `IMapLayout` interface (both interfaces were discussed in the Data structures section). The conversion mostly consists of a computation of door positions of individual rooms because the internal representation only guarantees that valid door positions can be found but they are not stated explicitly.

3.6 GUI

The GUI is implemented as Windows Forms application. Its description can be found in the User documentation.



(a) Main settings.



(b) Progress of the generator and export of generated layouts.

Figure 19: Screenshost of the GUI. A description of its individual controls can be found in the User documentation.

4. Results

Throughout the thesis we demonstrate how our algorithm handles various input graphs and building blocks sets. Our method is able to process complex graphs with multiple interconnected cycles (Figure 23) and it also successfully tackles large graphs (Figure 24). To demonstrate that our method can deal with layouts that are usually found in video games, we chose 2 maps from popular games and used their level connectivity graphs as an input to our algorithm (Figure 25 and Figure 26). For all these inputs, our method was able to quickly produce multiple diverse layouts.

4.1 Benchmarks

Since we use a stochastic method, results are often heavily influenced by the seed of the random numbers generator that is used in the algorithm. That means that the time needed to generate a layout will usually be quite different for individual runs of the algorithm. To evaluate the speed of our method, all benchmarks in this thesis are obtained by running our algorithm 100 times on each input graph, with different randomization seeds.

We measure the time that is needed to generate a layout and the number of iterations, i.e. how many times we need to perturb a layout to generate a full layout. For both these statistics, we provide average and median values because the average can be influenced by outlier runs. We also record the success rate which is defined as the number of runs that managed to generate a layout in less than 1 million iterations. We force the generator to stop if it exceeds this limit.

In Table 5, you can see a benchmark[1] of our method when used on input graphs presented in this thesis. Note that our method was able to generate all layouts without corridors in under one second. And all layouts with corridors in under two seconds. Our algorithm is, therefore, quick enough to serve as an inspiration for game designers or to generate layouts directly in a game.

In the Performance improvements section, we described our most important performance improvements and demonstrated their impact on the overall speed of our algorithm. For comparison, Table 6 (bottom) shows a benchmark of our initial implementation without all major improvements. This implementation can be considered to be a straightforward implementation of the original method in a context of tile-based maps. And for completeness, Table 6 (top) shows results of the original method from Ma et al. These results were obtained by benchmarking their implementation of the method that can be found on Github[11]. Even though the original implementation works with real coordinates, we can use our tile-based building blocks because rectilinear polygons represent a subset of general polygons. The only difference is that the output will be real-based.

To put our changes to perspective, all mentioned implementations are shown side by side in Figure 20. You can see that with our improvements, in a context of tile-based maps, our algorithm is over 100 times faster than the original one from Ma et al.

[1] All benchmarks in this thesis were done with the building blocks from Figure 21, on a 2.7GHz CPU (the algorithm runs on a single core).

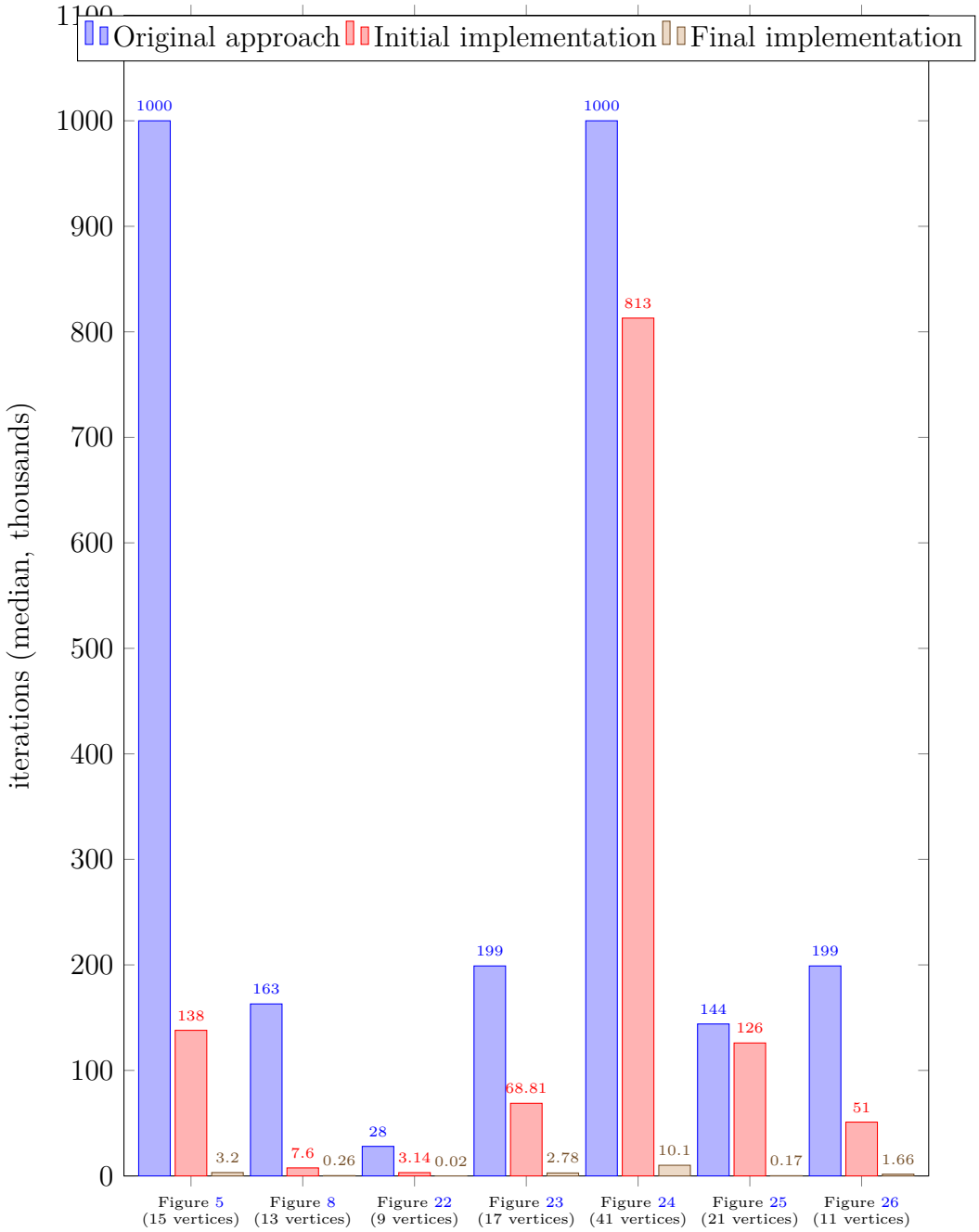


Figure 20: Comparison of the original method, our initial implementation and our final implementation. The plot shows the median number of iterations from Table 5 and Table 6. Fewer is better.

Input	Success rate	Time avg/med	Iterations avg/med
Without corridors:			
Figure 5 (15 vertices)	100%	0.18s/0.09s	5.50k/3.20k
Figure 8 (13 vertices)	100%	0.02s/0.01s	0.49k/0.26k
Figure 22 (9 vertices)	100%	0.00s/0.00s	0.12k/0.02k
Figure 23 (17 vertices)	100%	0.12s/0.08s	4.15k/2.78k
Figure 24 (41 vertices)	100%	0.62s/0.36s	15.28k/10.10k
Figure 25 (21 vertices)	100%	0.01s/0.00s	0.29k/0.17k
Figure 26 (11 vertices)	100%	0.05s/0.03s	2.83k/1.66k
With corridors:			
Figure 5 (15 vertices)	100%	0.35s/0.16s	4.57k/2.50k
Figure 8 (13 vertices)	100%	0.05s/0.04s	1.01k/0.69k
Figure 22 (9 vertices)	100%	0.01s/0.01s	0.28k/0.08k
Figure 23 (17 vertices)	100%	0.87s/0.54s	17.85k/11.55k
Figure 24 (41 vertices)	100%	1.61s/1.35s	20.16k/16.90k
Figure 25 (21 vertices)	100%	0.04s/0.02s	0.90k/0.39k
Figure 26 (11 vertices)	100%	0.12s/0.07s	3.78k/2.24k

Table 5: Benchmark of our final implementation of both our modes. The benchmark was run 100 times for each input graph. Success rate shows how often was the algorithm able to generate a valid layout. We provide average and median values for both the time and the number of iterations.

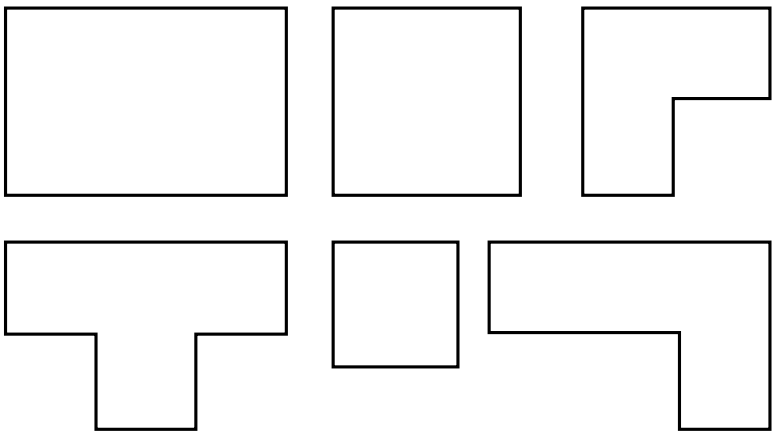


Figure 21: Building blocks used for benchmarks.

Input	Success rate	Time avg/med	Iterations avg/med
Original approach:			
Figure 5 (15 vertices)	40%	38.00s/33.00s	667k/1000k
Figure 8 (13 vertices)	80%	14.70s/2.25s	62k/163k
Figure 22 (9 vertices)	100%	0.56/0.07s	16k/28k
Figure 23 (17 vertices)	40%	19.00s/46.00s	400k/199k
Figure 24 (41 vertices)	8%	50.00s/55.00s	920k/1000k
Figure 25 (21 vertices)	60%	30.00s/24.00s	500k/144k
Figure 26 (11 vertices)	100%	10.00s/20.00s	183k/199k
Our initial implementation:			
Figure 5 (15 vertices)	86%	2.77s/1.10s	323.54k/138.29k
Figure 8 (13 vertices)	100%	0.30s/0.16s	14.90k/7.60k
Figure 22 (9 vertices)	100%	0.18s/0.06s	11.56k/3.14k
Figure 23 (17 vertices)	100%	2.03s/1.41s	96.65k/68.81k
Figure 24 (41 vertices)	58%	9.37s/10.28s	764.15k/813.83k
Figure 25 (21 vertices)	100%	1.93s/1.83s	132.33k/126.41k
Figure 26 (11 vertices)	100%	1.21s/0.73s	87.96k/51.00k

Table 6: Top table shows a benchmark of the implementation of the original method. Bottom table shows a benchmark of our initial implementation before we applied the speedups proposed in this thesis. Success rate shows how often was the algorithm able to generate a valid layout. We provide average and median values for both the time and the number of iterations.

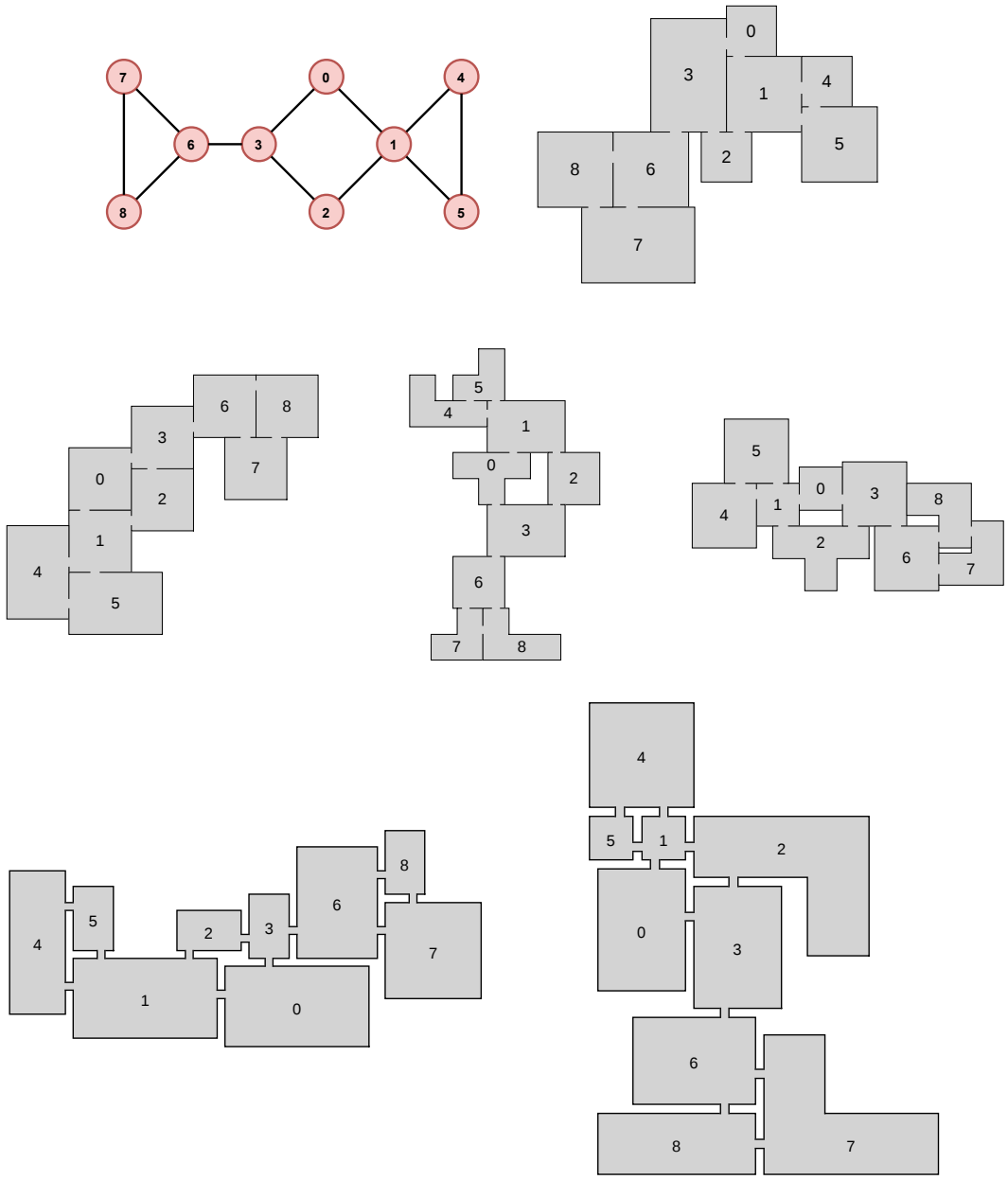


Figure 22: Layouts generated from a simple input graph with 9 vertices. Various sets of building blocks are used.

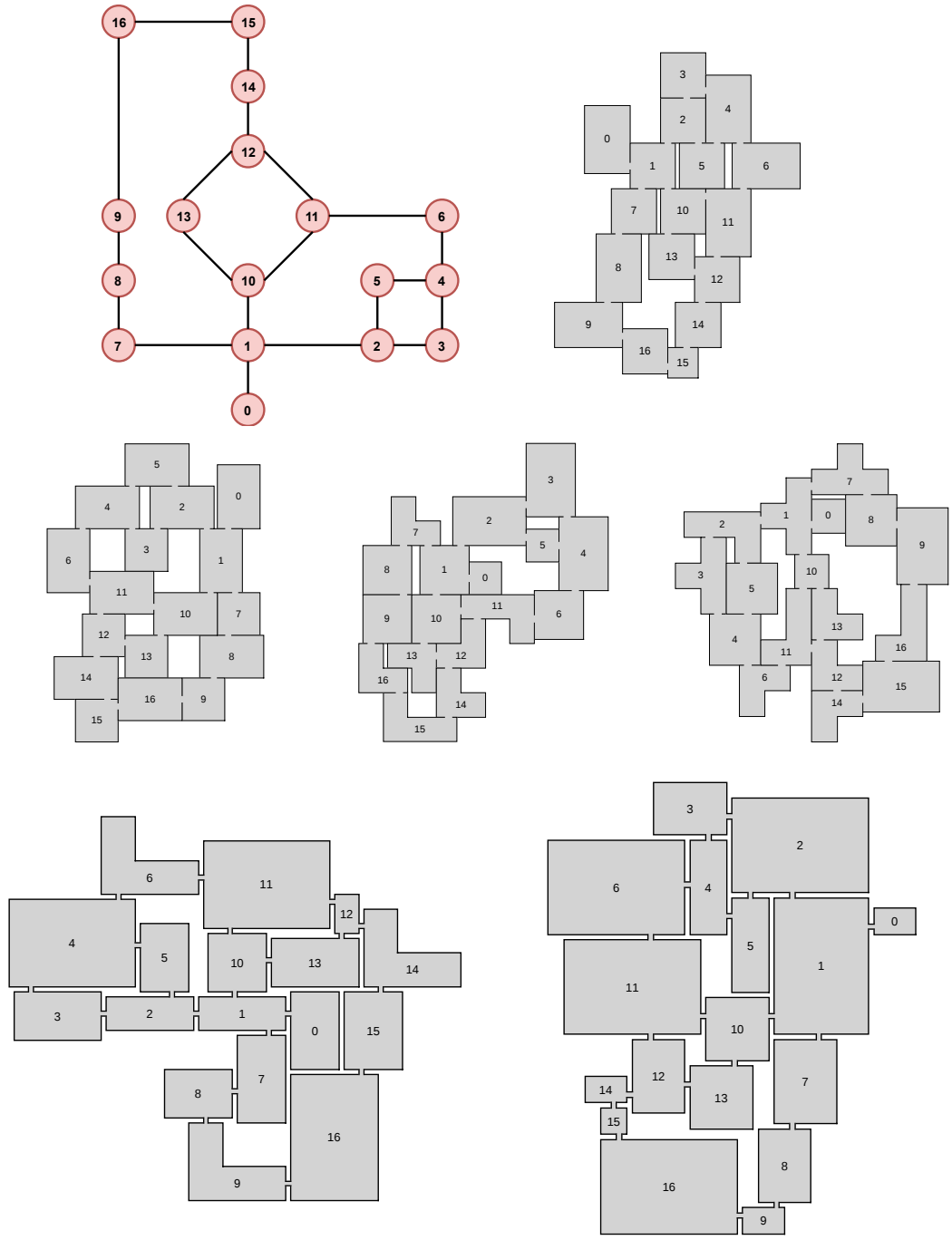


Figure 23: Layouts generated from a complex graph with multiple interconnected cycles. Various sets of building blocks are used.

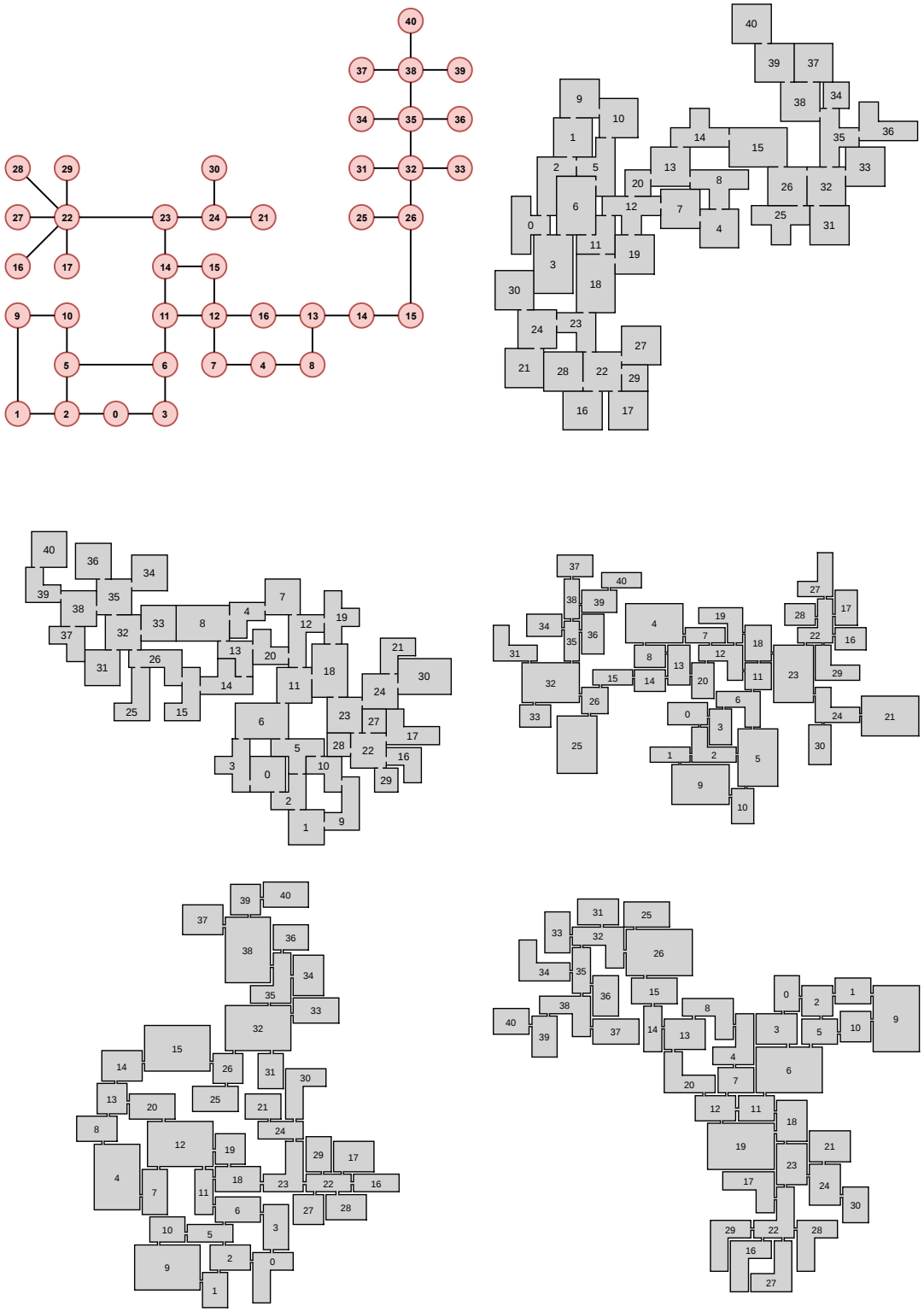


Figure 24: Layouts generated from a large graph with 41 vertices. Various sets of building blocks are used.

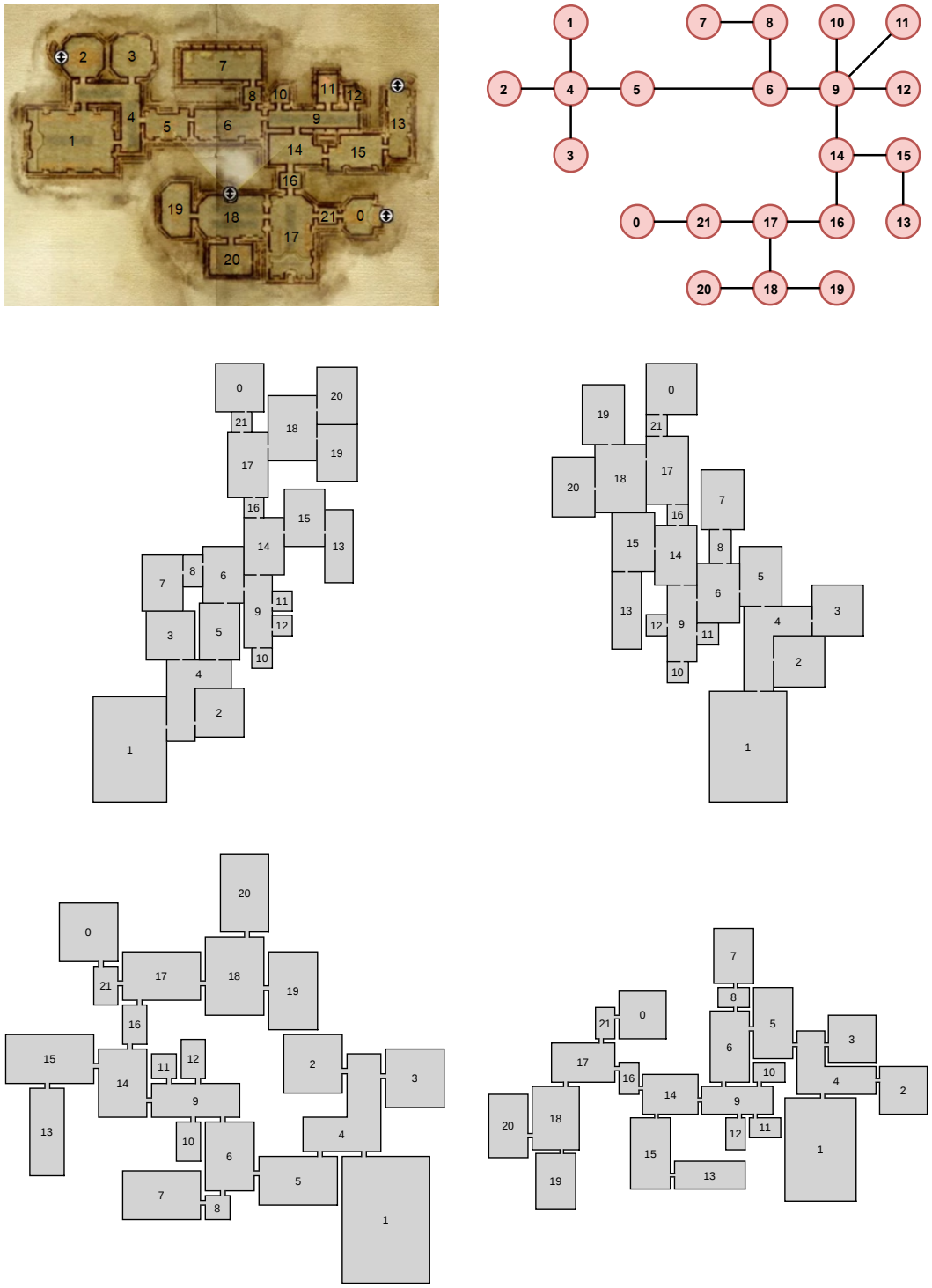


Figure 25: Layouts generated from a graph that is based on a map from Dragon Age: Origins. We tried to choose room shapes that are similar to those found in the map.

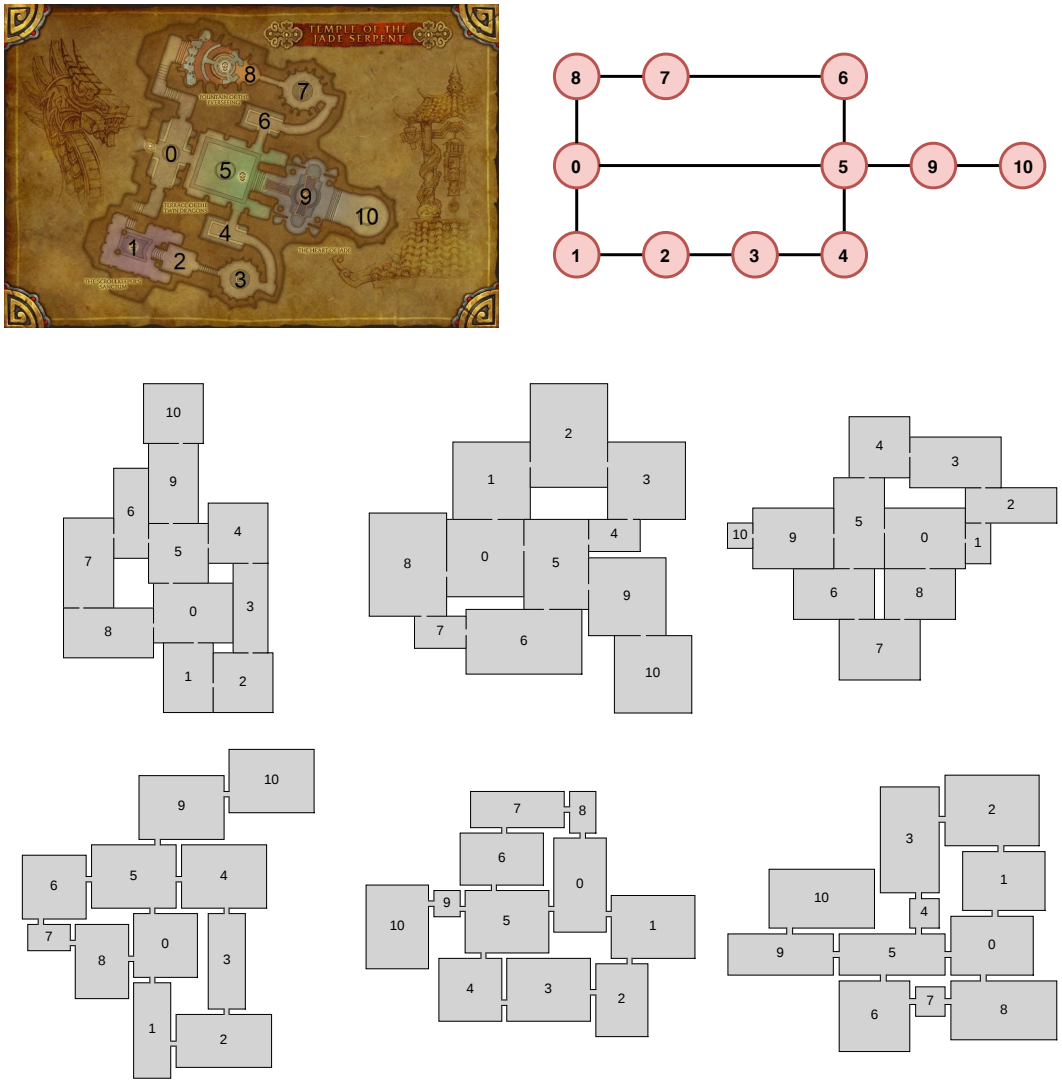


Figure 26: Layouts generated from a graph that is based on a map from World of Warcraft. A set of simple rectangular building blocks is used.

Conclusion

We presented an algorithm for procedural generation of tile-based maps from user-defined building blocks. It takes a level connectivity graph as an input and produces layouts that satisfy connectivity constraints imposed by the graph. Our method is based on the previous work of Ma et al. They first decompose the input graph to smaller subgraphs and then use simulated annealing to lay them out one at a time.

The original method was enhanced with several new features. Users can now easily specify door positions of building blocks and explicitly set the probability distribution of choosing individual room shapes. We also presented a method to quickly generate layouts with rooms connected by short corridors as usually found in dungeon levels. Moreover, we proposed several performance improvements, including tweaks of simulated annealing and smarter decomposition of the input graph.

The resulting application consists of two parts. The first part is a C# library with the layout generator itself. It is implemented as a framework that lets programmers easily extend or replace individual components of the algorithm. The second part is a simple GUI that is controlled with config files written in YAML. It allows game designers to use the generator without any prior programming knowledge.

We demonstrated that our method can handle various input graphs and building blocks sets. Because our method is stochastic, we benchmarked the speed of the generator by running it multiple times for every input. Results of these benchmarks showed that, on average, our algorithm is over 100 times faster then the original one, and able to generate a layout in under one second for all our inputs in the basic mode without corridors. This makes our algorithm fast enough to be used directly in a game or as an inspiration for game designers.

Future works

Unity plugin

Even though we tried to make the API of the generator user-friendly, the process of creating level connectivity graphs and room shapes can still be quite time-consuming. It would be, therefore, convenient to create a plugin for Unity (or any other game engine) that would connect our library directly to the game engine. It would allow users to draw input graphs in a GUI and possibly even design room shapes and assign materials to walls, floor, etc. After creating a map description, the plugin would be used to spawn the map in the game.

Designing new constraints

When implementing new features, we always tried to create a simple API and find ways to keep the convergence rate of the generator as good as possible. Authors of the original paper demonstrated that the algorithm can be used to generate layouts spanning multiple floors and layouts that must avoid intersecting user-defined obstacles. Even though we could possibly come up with a proof of concept

of such constraints, there was not enough time to think it through and implement it properly.

Speed improvements

Even though we managed to significantly speed up the original method, there is always a room for improvements. One possibility is to try a different stochastic method for evolving layouts. Another possibility is to improve the generator planner - either by making it multithreaded or by applying some heuristic to control which layouts are expanded.

Bibliography

[1] M. Armstrong, S. Hurley, and S. Palmer. Borderlands, 2009.

[2] A. Aubry. YamlDotNet. `https://github.com/aaubry/YamlDotNet`, v4.3.

[3] Boost C++ libraries. `https://www.boost.org/`, v1.66.0.

[4] M. Buchetics. RangeTree. `https://github.com/mbuchetics/RangeTree`, v1.0.

[5] M. Chrobak and T. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1989.

[6] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, 2011.

[7] G. Gygax and D. Arneson. Dungeons & Dragons, 1974.

[8] J. Hedengren. Simmulated annealing tutorial. `http://apmonitor.com/me575/index.php/Main/SimulatedAnnealing`, 2013.

[9] J. Hopcroft, J. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects; pspace- hardness of the "warehouseman's problem". *International Journal of Robotics Research*, 3(4):76–88, 1984.

[10] A. Johnson. Clipper. `http://www.angusj.com/delphi/clipper.php`, 2014.

[11] C. Ma, N. Vining, S. Lefebvre, and A. Sheffer. Game level layout from design specification. *Computer Graphics Forum*, 34(2), 2014. `https://github.com/chongyangma/LevelSyn`.

[12] Microsoft. Windows forms. `https://docs.microsoft.com/en-us/dotnet/framework/winforms/`.

[13] J. Newton-King. Newtonsoft.Json. `https://github.com/JamesNK/Newtonsoft.Json`, v11.0.

[14] M. Persson and J. Bergensten. Minecraft, 2009.

[15] C. Poole and R. Prouse. nUnit. `https://github.com/nunit/nunit`, v3.9.

[16] E. Schaefer, D. Brevik, M. Schaefer, E. Sexton, and K. William. Diablo, 1996.

[17] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[18] Unity Technologies. Company facts. `https://unity3d.com/public-relations`.

[19] Unity Technologies. Unity user manual. https://docs.unity3d.com/Manual/index.html, v2018.1.

[20] M. Toy, G. Wichman, K. Arnold, and J. Lane. Rogue, 1980.

List of Figures

List of Tables

A. Attachments

A.1 Contents of the attached CD

- `/Binaries/GUI` - binaries of the GUI application
- `/Binaries/MapGeneration` - binaries of the the layout generator
- `/Documentation` - user documentation
- `/Source` - source code
- `/Thesis/thesis.pdf` - this thesis