Fast Configurable Tile-Based Dungeon Level Generator

Ondřej Nepožitek, Jakub Gemrot
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské náměstí 25, 118 00, Prague 1, Czech Republic
E-mail: ondra@nepozitek.cz, gemrot@gamedev.cuni.cz

KEYWORDS
Procedural content generation, Dungeon levels, Stochastic method, Simulated Annealing

ABSTRACT

Procedural generation of levels is being used in many video games to increase the replayability. But generated 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 the algorithm searches only over valid relative positions of pre-defined building blocks using configuration spaces. In this paper, we present an implementation of this method in a context of 2D tile-based maps. We enhance the algorithm with several new features, for example, a mode to quickly add short corridors between neighbouring rooms, specifying doors or allowing a user to specify constrains over generated levels. We also implement speed improvements including smarter decomposition of the input graph and tweaks to the stochastic method that is used to lay out individual chains. We also show that the algorithm is able to 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. As the result, it is suitable to be used during game runtime.

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 which is a dungeon crawler video game inspired by a board game Dungeons & Dragons. It contains procedurally generated dungeon levels, treasures and monster encounters and that leads to a unique experience on every playthrough. Procedural techniques are also used in newer games including Borderlands, Diablo or Minecraft.

There are many areas in game development, where procedural generation can help with. One of them is the generation of game levels. One popular approach to this problem is to use binary space partitioning (Shaker et al. 2016). 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 (Shaker et al. 2016). 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 (Dormans and Bakkes 2011, Ma et al. 2014). Although this approach may be appropriate in some genres, Dormans & Sanders (Dormans and Bakkes 2011) note that the gameplay induced by these algorithms 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. (Ma et al. 2014) propose a different approach. Their method takes a set of room shapes and the level connectivity graph as an input and produces layouts that all follow the defined structure; a game designer thus have complete control of high-level structure of the level, for example, a control over possible sequences a player can visit respective rooms (graph nodes).

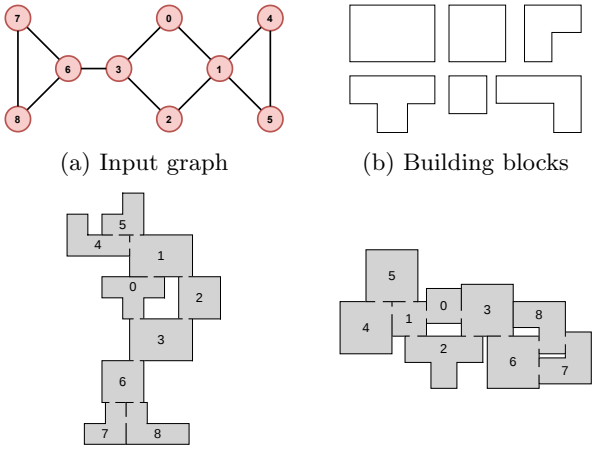
In this paper, we describe conceptual and technical improvements to the procedural generation algorithm for dungeon levels developed by Ma et al. (Ma et al. 2014)

The rest of the paper is structured as follows. We first describe the algorithm in detail, then we describe new features and speed improvements that we implemented. To illustrate how the algorithm can be useful in practice, we provide a use-case of dungeon level generation related to an existing game. Finally, we evaluate the performance of the algorithm with respect to the original one and conclude the paper.

ALGORITHM

To produce a dungeon level, the algorithm (Ma et al. 2014) takes a set of polygonal building blocks (referred to as room shapes or simply as shapes) and a level connectivity graph (the level topology) as an input. Nodes in the graph represent rooms, and edges define connectivities between them. The graph has to be planar. The goal of the algorithm is to assign a room shape and a position to each node in the graph in a way that no two room shapes intersect and every pair of neighbouring room shapes share a common boundary segment (Figure 1).

Instead of searching through all possible positions and room shape assignments of an input graph's nodes, the algorithm uses configuration spaces to define valid relative positions of individual room shape pairs. The configuration space of two shapes is a set of such positions in $\mathbb{R}^2$ that if we translate one of the shapes to that position, both shapes shares a common segment (so they can be connected, for example, by a door) and they 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 (Hopcroft et al. 1984). Therefore, a probabilistic optimization technique is used to efficiently explore the search space. To further speed up the optimization, the input problem is broken down to smaller and easier subproblems. This is done by decomposing the graph into smaller parts (called chains as detailed later) and then laying them out one at a time.



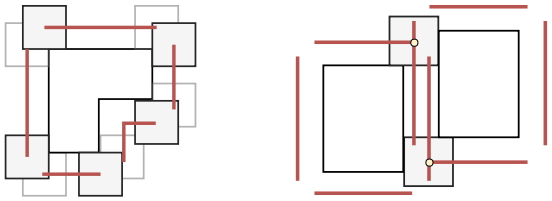
(a) Input graph (b) Building blocks

(c) Generated layout (d) Generated layout

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

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 shares common segment(s). When working with polygons, each configuration space can be represented by a (possibly empty) set of lines (Figure 2) 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.

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

Incremental layout

The algorithm assigns room shapes to graph nodes incrementally; in each step it lays out one chain. A chain is a sub-graph where each node has at most two neighbours. Chains have an advantage that they are relatively easy to lay out. The final 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 nodes.

```
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 layouting. In each iteration of the algorithm, we take the last layout from the stack and try to add the next chain, generating multiple extended layouts and storing them. If this 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. The backtracking is usually needed when there is not enough space to connect additional chains to already laid out nodes. The process terminates when enough full layouts are generated or if no more distinct layouts can be computed.

The strategy of decomposing a graph into chains is based on computing a planar embedding of the graph (Chrobak and Payne 1989) and then using faces of the embedding to form respective chains. The first chain in the decomposition is formed by the smallest face of the embedding and following faces are added in a breadth-first order. If there are more faces to choose from, we first lay out the smallest one. When there are no faces (no cycles) left, remaining acyclical components are added.

Favoring cycles is quite important as we have empirically confirmed them to be harder to layout and causing the algorithm to backtrack unnecessary (as the original paper states).

Simulated annealing

The authors of the original algorithm 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.

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.

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 (Hedengren 2013).

```
1  Input: layout, current chain
2
3  procedure PerturbLayout(layout, chain)
4   configSpaces ← Get precomputed configuration spaces
5   perturbShape ← Pick at random − 40% true, 60% false
6   nodeToBePerturbed ← Get a random node from the chain
7
8   if perturbShape then
9    Pick a random shape for nodeToBePerturbed
10  else
11   Use configSpaces to get a random position from the interesection
12   of configuration spaces of neighbours of nodeToBePerturbed
13  end if
14
15  Update position/shape of nodeToBePerturbed
16  Update energy of layout
17
18  return perturbed layout with updated energy
19 end procedure
```

Algorithm 3: Layout perturbation.

Tile-based output

We provide an implementation of the algorithm 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.

NEW FEATURES

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; therefore, the algorithm 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 not to converge at all.

We present a different approach to this problem. We use two different instances of configuration spaces. The first one is the standard 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).

When we perturb a layout, we first use the second type of configuration spaces. By doing so, we should converge to a state where all pairs of non-corridor nodes of the current chain have a space between them. When this happens, we switch to the first type of configuration spaces 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 and return to simulated annealing.

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 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.

Explicit door positions

In the original algorithm, it is not possible to specify door positions of individual room shapes easily. It only allows us to configure one global length and that is used for all doors in a layout. However, there are situations where it is convenient to explicitly specify door positions of a room. For example, we may have a boss room and we require 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, furnitures, chests or other items or 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, one must be careful as having too few door positions, for example, 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.

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.

The framework allows us to define two types of constraints - the first type ensures that the whole layout satisfies some conditions (for example, minimum and/or maximum of total area of the generated level) and the second type operates on individual nodes (for example, 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.

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.

SPEED IMPROVEMENTS

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. 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. 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. We tried three different mechanisms that decide if the current iteration of simulated annealing is successful or not. The first one is the original approach where we penalize iterations that fail to accept new states. The second one penalizes iterations that fail to produce valid layouts. And finally the third one 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 the most strict 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.

Chain decomposition

The decomposition of an input graph into chains affects the overall performance of the algorithm. We observed that having a lot of small chains in a decomposition leads to poor performance, especially in situations 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.

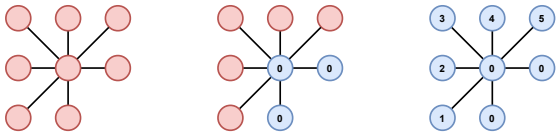
In Figure 3a we can see an example of a problematic graph - it has a lot of nodes with only a single neighbour. Figure 3b shows how the decomposition may look like after the first iteration of the basic decomposition algorithm. We can see that we have a lot of small acyclic components now. The problem is that the algorithm creates a new chain from every such component. And finally in Figure 3c, we can see that we will end up with a lot of small chains - which is a situation we want to avoid.

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 are violiting the definition of a chain because we allow a node to have more than two neighbours in a single chain. However, we found out that this approach performs better in practice.

Lazy evaluation

Another technical problem is that the algorithm is trying to generate multiple layouts in each run of simulated annealing 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 needed.

We can imagine that the algorithm builds a tree



(a) (b) (c)

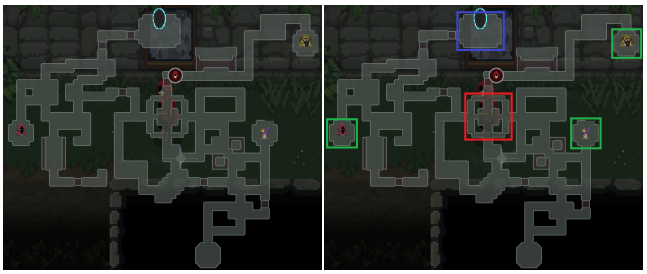
Figure 3: Problematic input for chain decomposition. Blue nodes are contained in a chain with a corresponding number. (a) shows the input graph. (b) shows a partial chain decomposition after the first iteration of the algorithm. (c) shows a complete chain decomposition of the graph (6 chains in a graph of 8 nodes).

where each node represents a generated partial layout and all children of a node are layouts that were generated from the parent node. The problem is that we always generate all children of a node before we move to another node.

The solution is to make the computation lazy. 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 truly required; as we are using C#, this is easily achievable by using *yield* statement.

USE-CASE

In the following paragraphs, we will demonstrate how to use the algorithm to generate game levels that are similar to what can be found in a roguelike game Wizard of Legend. We chose this game because its procedurally generated maps contain elements that we can use to showcase various features of our algorithm.

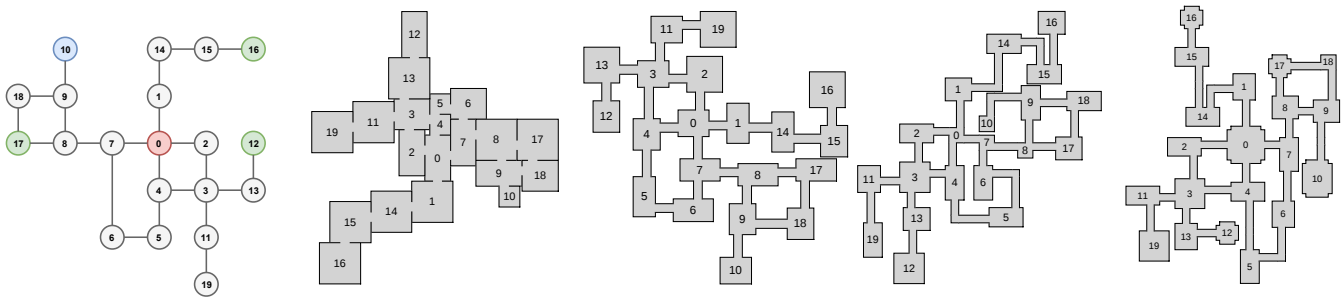


(a) (b)

Figure 5: (a) shows one of the maps from the game Wizard of Legend. (b) shows the same map but with highlighted special room types that are present on every map in the game.

Map analysis

Even though all maps within the game look different, they all share some common elements. In Figure 5, we can see an example of such a map. The player always starts in a spawn room (red) and the goal is to get to a boss room (blue) and defeat the boss. On each level, there are also 3 shop rooms (green). These three room types have always the same room shapes assigned. The rest of the rooms contains various obstacles with enemies



(a) (b) (c) (d) (e)

Figure 4: Imitating the style from Wizard of Legend. Figure (a) shows the input graph. Figures (b) - (e) demonstrate various stages of the generator setup.

and uses a common pool of room shapes.

Algorithm setup

The first step is to create the input graph. Based on the analysis from the previous section, we prepared a sample input graph that can be seen in Figure 4a. And Figure 4b shows how the output looks like if we use this graph with some basic rectangular building blocks.

The next step is to add corridors. In the majority of popular algorithms for dungeon level generation, we let the algorithm figure out how to connect individual rooms and often do not explicitly limit the look of created corridors. Our method is, however, based on configuration spaces and therefore we need a fixed geometry during the optimization process. That means that we have to define all possible shapes of corridor rooms beforehand. In Figure 4c, we can see how the output looks like after we defined several rectangular corridor shapes with various lengths. In the maps from the game, however, corridors are often l-shaped. Instead of defining all possible l-shapes, we randomly pick a pair of neighbouring rooms in the graph and insert a small square room between them. This allows us to keep our corridor shapes simple and at the same time add diversity to the output. Result of this step can be seen in Figure 4d.

In the map analysis, we described that there are three types of special rooms, each of them having a specific room shape assigned. To achieve that, we simply create individual polygon shapes and set up the algorithm to use only these shapes when deciding what shape to use for such rooms (Figure 4e). In most cases, we also want to use all the possible rotations of defined room shapes, which can be handled by the generator itself. The final step is to enhance the set of building blocks that is used for ordinary rooms.

Results

In Figure 6, we can see several layouts generated for the input graph in Figure 4a. The next step would be to apply a simple postprocess to make the corridors feel more organic or to implement an algorithm that would make small changes to the input graph.

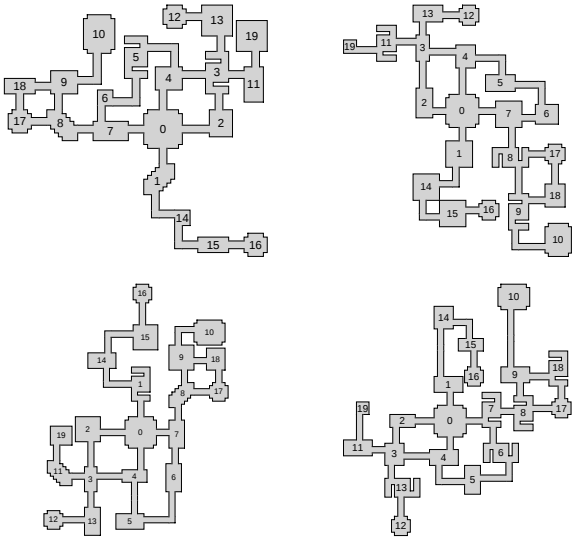


Figure 6: Layouts generated in the style of Wizard of Legend. The median time needed to generate a layout was 0.53 seconds.

EVALUATIONS

Since we use a stochastic method, the total time required to find the first full layout is influenced by the seed of the random number generator. Therefore, all benchmarks presented here 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 statistics, we provide average and median values.

In Table 1, we can see a benchmark of our method when used on input graphs presented in Figures 1 and 8. 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 generate layouts directly in a game.

Input	Time avg/med	Iterations avg/med
Without corridors:		
Fig. 1	0.00s/0.00s	0.12k/0.02k
Fig. 8, top-left	0.12s/0.08s	4.15k/2.78k
Fig. 8, top-right	0.01s/0.00s	0.29k/0.17k
Fig. 8, bottom-left	0.18s/0.09s	5.50k/3.20k
Fig. 8, bottom-right	0.62s/0.36s	15.28k/10.10k
With corridors:		
Fig. 1	0.05s/0.04s	1.01k/0.69k
Fig. 8, top-left	0.87s/0.54s	17.85k/11.55k
Fig. 8, top-right	0.04s/0.02s	0.90k/0.39k
Fig. 8, bottom-left	0.35s/0.16s	4.57k/2.50k
Fig. 8, bottom-right	1.61s/1.35s	20.16k/16.90k

Table 1: Benchmark of our final implementation of both modes. The benchmark was run 100 times for each input graph. We provide average and median values for both the time and the number of iterations. All benchmarks were done with the building blocks from Figure 1b, on a 2.7GHz CPU (the algorithm runs on a single core).

Performance comparison

In the Speed improvements section, we described our most important performance improvements. Figure 7 shows a comparison of how these changes affect the overall speed of the algorithm. The y-axis of the chart shows the relative speedup of the algorithm when compared to the implementation of the original method from Ma et al. which can be found on Github (Ma et al. 2014). 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.

The first bar shows the performance of our initial implementation of the original method in a tile-based context. We can see that this implementation is already faster than the original algorithm. This is mainly caused by the fact that with integer coordinates, we were able to significantly speedup operations with polygons and energy function computation. We can also see that this implementation converges faster in terms of iterations count. That is most likely because simulated annealing converges slightly faster in a tile-based context. We tried to scale all the rooms up to simulate, to some extent, real coordinates, and simulated annealing needed approximately 3-5 times more iterations to produce a valid layout.

The following three bars of the charts demonstrate how the algorithm behaves with only one improvement enabled. And finally, the last bar shows that with all improvements enabled, our algorithm is over 100 times faster than the original method.

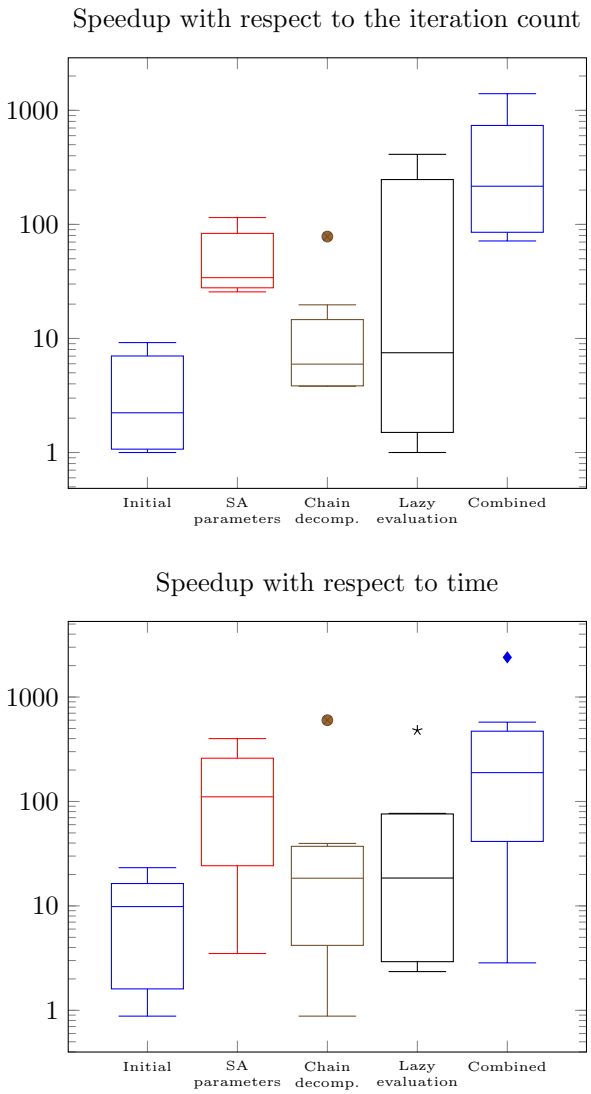


Figure 7: Relative speedup of our performance improvements when compared to the implementation of the original method. Bottom chart shows how different approaches affect the total time needed to generate a layout. Top chart shows how the iterations count is affected.

CONCLUSION

We presented an algorithm for procedural generation of tile-based maps from user-defined building blocks. It takes a planar 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.

The original method was enhanced with several new features. Users can now easily specify door positions of building blocks and add custom constraints on the layout. 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

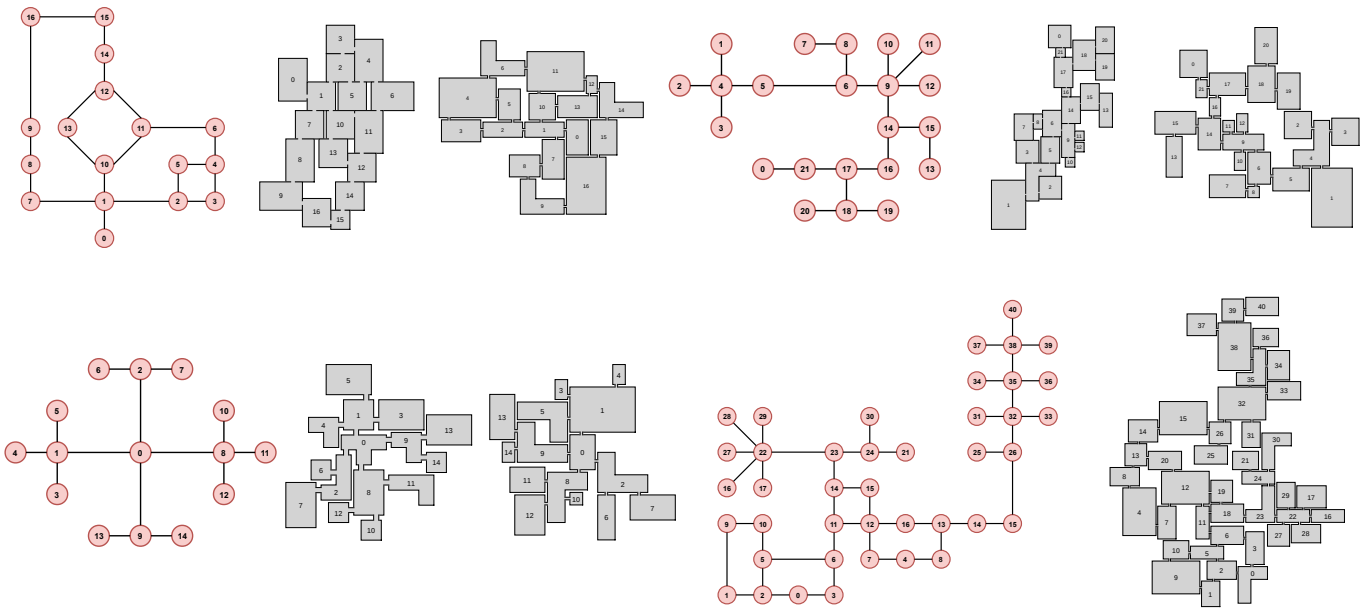


Figure 8: Layouts generated from four different input graphs, using various sets of building blocks.

simulated annealing and smarter decomposition of the input graph.

We demonstrated that our method can handle various input graphs and building blocks sets. Benchmarks of our method showed that, on average, our algorithm is over 100 times faster then the original one, and is 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 during runtime or as an inspiration for game designers during design time.

Our C# implementation of the algorithm can be downloaded from `https://github.com/OndrejNepozitek/ProceduralLevelGenerator` under the MIT License, which allows the result to be used in commercial games.

Future works

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.

It would be also interesting to investigate ways to further improve the speed of the algorithm. For example, it is possible to make the algorithm multithreaded or try a different stochastic method for evolving layouts.

ACKNOWLEDGMENTS

This research was funded by the Czech Science Foundation (project no. 17-17125Y).

REFERENCES

Chrobak M. and Payne T., 1989. "A linear-time algorithm for drawing a planar graph on a grid". *Information Processing Letters*, 54, no. 4, 241–246.

Dormans J. and Bakkes S., 2011. "Generating Missions and Spaces for Adaptable Play Experiences". *IEEE Transactions on Computational Intelligence and AI in Games*, 3, no. 3, 216–228.

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

Hopcroft J.; Schwartz J.; and Sharir M., 1984. "On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE- Hardness of the "Warehouseman's Problem"". *International Journal of Robotics Research*, 3, no. 4, 76–88.

Ma C.; Vining N.; Lefebvre S.; and Sheffer A., 2014. "Game Level Layout from Design Specification". *Computer Graphics Forum*, 34, no. 2. `https://github.com/chongyangma/LevelSyn`.

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