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 their 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. 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 and propose speed improvements. We also show that the algorithm is able to produce diverse layouts. Benchmarks 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 content generation (PCG) is a method of creating content algorithmically rather than by hand (Togelius et al. 2010). In video games, it is often used to increase game's replayability. The classic example is the game Rogue that contains procedurally generated dungeon levels, treasures and monster encounters, which lead to unique experience on every playthrough. Procedural techniques are also used in newer games including Borderlands, Diablo or Minecraft.

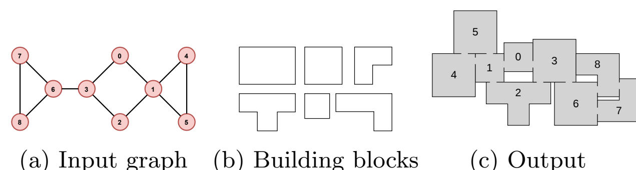
In this paper, we focus on PCG of game levels. One approach to this problem is to use binary space partitioning (Shaker et al. 2016); they start with a rectangular area and recursively split it until there are enough subareas. Some subareas are then chosen to represent rooms and these are then connected by corridors. Another possible approach is so-called agent-based dungeon growing (Shaker et al. 2016); they start 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 it is not suitable for action-adventure games and propose to generate both missions and spaces of a game using generative grammars. 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 over the high-level structure of a 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.

The rest of the paper is structured as follows. We first describe the algorithm in detail, then we show our new features and speed improvements. Finally, we evaluate algorithm's performance and conclude the paper.

ALGORITHM

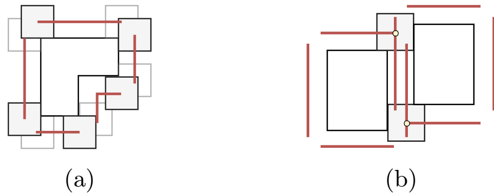
To produce a dungeon level, the algorithm (Ma et al. 2014) takes a set of polygonal building blocks (referred to as room shapes) and a planar level connectivity graph (the level topology) as an input. Nodes in the graph represent rooms, and edges define connectivities between them. The goal of the algorithm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



(a) Input graph (b) Building blocks (c) Output

Figure 1: Example output of the algorithm.

Instead of searching through all possible positions and room shape assignments of graph's nodes, the algorithm uses configuration spaces to define valid relative positions of individual room shape pairs. However,



(a) (b)

Figure 2: Configuration spaces. (a) the configuration space (red lines) of the free square with respect to the fixed l-shaped polygon. (b) the intersection (yellow dots) of configuration spaces of the moving square with respect to the two fixed rectangles.

formulating the whole problem as a configuration space 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) and laying them out one at a time.

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 share common segment(s). With polygons, a configuration space can be represented by a (possibly empty) set of lines (Figure 2).

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

Algorithm 1 assigns positions and 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 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  procedure IncrementalLayout(c,s)
3   Push empty layout into S
4
5   repeat
6    s ← S.pop()
7    Get the next chain c to add to s
8    AddChain(c, s)              // extend the layout to contain c
9
10   if extended partial layouts were generated then
11    Push new partial layouts into S
12   end if
13  until target # of full layouts is generated or S is empty
14 end procedure
```

Algorithm 1: Incremental layout.

In each iteration, 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 (referred to as *backtracking*). It is usually needed when there is not enough space to connect additional chains to already laid out nodes. The process terminates when enough number of full layouts are generated or if no more distinct layouts can be computed.

To decompose a graph into chains, we first compute a planar embedding of the graph (Chrobak and Payne 1989). The first chain is then 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 unnecessarily (as the original paper states).

Simulated annealing

The authors of the original algorithm chose simulated annealing (SA) framework to explore the space of possible layouts for individual chains. The reason is that it produces multiple partial layouts in a single run, which is useful in two situations. First, it allows us to backtrack if we are unable to lay out a chain. 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 SA previously.

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

SA operates by iteratively considering local perturbations to the current configuration, or layout. 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.

Tile-based output

We provide an implementation of the algorithm in a context of tile-based maps. Therefore we changed the representation of shape coordinates from original floats to integers and we use only rectilinear polygons instead of arbitrary polygons for room shapes.

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 into the input graph and these new nodes get assigned a set of room shapes that was made for corridors.

The problem of this approach is, that we now have almost twice as many nodes than before, which slows the generation down.

In our approach, we use two different instances of configuration spaces. The first is the standard one, in which a position of two rooms is valid when both rooms touch and do not overlap. The second, 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. Then we switch to the first type of configuration spaces and try to greedily add all corridors rooms. If we are not able to lay out all corridor rooms this way, we abort the current attempt, remove already added corridors and return to SA.

Explicit door positions

In the original algorithm, it is not easy to specify door positions within individual room shapes. But this can be useful, for example, if we have a boss room which requires the player to enter the room from a specified tile, or if we have a room template with some tiles reserved for furniture, chests, etc.

Therefore, we extended configuration spaces generator to work with door positions directly. It allows us to define door positions of every room shape in a layout explicitly. This modification has no runtime overhead as configuration spaces are generated only once before the algorithm starts. One must be careful though as having too few door positions makes it significantly harder for SA 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. We allow to define constraints over the whole layout and constraints over individual nodes.

They 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.

This can be useful, for example, if we want to make sure that the whole layout does not exceed some defined boundaries or if we create an obstacle that the layout must not intersect.

SPEED IMPROVEMENTS

Simulated annealing parameters

We observed that SA spends most of the time 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 SA. 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. We tried three different mechanisms that decide if the current iteration of SA is successful or not:

1) is the original approach where we penalize iterations that fail to accept new states; 2) penalizes iterations that fail to produce valid layouts; 3) 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

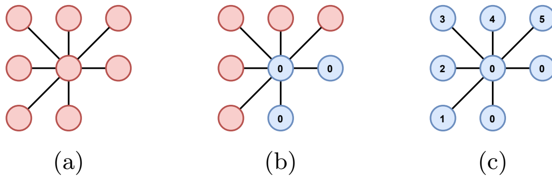


Figure 3: Problematic input for chain decomposition. Blue nodes are contained in a chain with a corresponding number. (a) shows the input graph. (b) shows a partial chain decomposition after the first iteration of the algorithm. (c) shows a complete chain decomposition of the graph (6 chains in a graph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

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. 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. This violates the definition of a chain, but we found out this approach to perform 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.

The solution is to make the computation lazy. Instead of generating all children layouts 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.

EVALUATIONS

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.

In Table 1, we can see a benchmark of our method when used on input graphs presented in Figures 1 and 5. 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
Fig. 1	0.00s/0.00s	0.12k/0.02k
Fig. 5, top-left	0.12s/0.08s	4.15k/2.78k
Fig. 5, top-right	0.01s/0.00s	0.29k/0.17k
Fig. 5, bottom-left	0.18s/0.09s	5.50k/3.20k
Fig. 5, bottom-right	0.62s/0.36s	15.28k/10.10k

Table 1: Benchmark of our final implementation of the algorithm. All benchmarks were done with the building blocks from Figure 1b, on a 2.7GHz CPU (the algorithm runs on a single core).

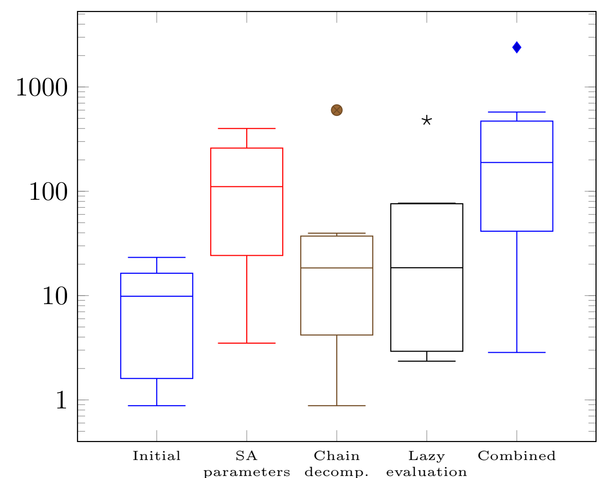


Figure 4: Relative speedup of our performance improvements when compared to the implementation of the original method, available at Github (Ma et al. 2014). The chart shows how different approaches affect the total time needed to generate a layout.

Performance comparison

In the Speed improvements section, we described our most important performance improvements. Figure 4 shows a comparison of how these changes affect the overall speed of the algorithm.

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.

The following three bars of the chart 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.

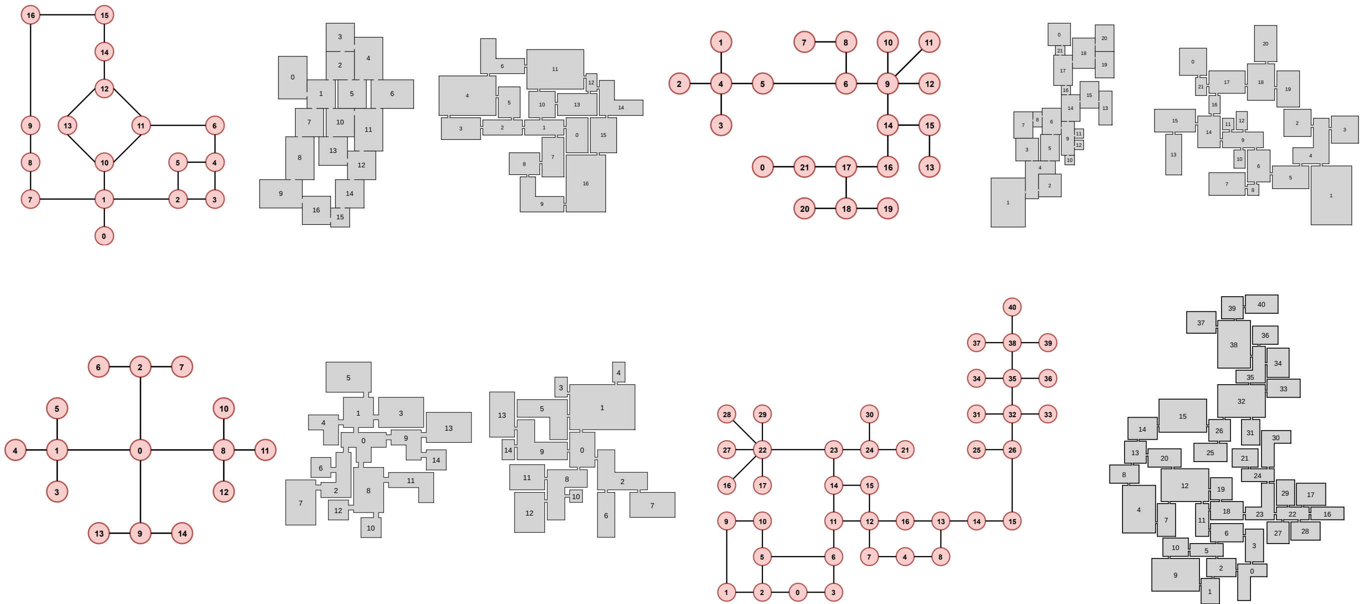


Figure 5: Layouts generated from four different input graphs, using various sets of building blocks.

CONCLUSION

We presented an algorithm for procedural generation of tile-based maps from user-defined building blocks that adapts and improves the previous work of Ma et al.

We proposed several new features and speed improvements. Users can now easily specify door positions 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.

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. The repository also contains an extended version of the paper together with an example of a practical use-case of the algorithm.

ACKNOWLEDGEMENTS

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.

Togelius J.; Yannakakis G.N.; Stanley K.O.; and Browne C., 2010. “Search-based Procedural Content Generation”. In *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I.* Springer-Verlag, Berlin, Heidelberg, EvoApplicatons’10. ISBN 3-642-12238-8, 978-3-642-12238-5, 141–150. doi:10.1007/978-3-642-12239-2_15. URL `http://dx.doi.org/10.1007/978-3-642-12239-2_15`.