

Automated Decomposition of Game Maps

Kári Halldórsson and Yngvi Björnsson

School of Computer Science, Reykjavik University
IS-101 Menntavegur 1, Reykjavik, Iceland
{kaha,yngvi}@ru.is

Abstract

Video game worlds are getting increasingly large and complex. This poses challenges to the game AI for both pathfinding and strategic decisions, not least in real-time strategy games. One way to alleviate the problem is to manually pre-label the game maps with information about regions and critical choke points, which the game AI can then take advantage of. We present a method for automatically decomposing game maps into non-uniform sized regions. The method uses a flooding algorithm at its core and has the benefit, in addition to its effectiveness, to be relatively intuitive both conceptually and in implementing. Empirical evaluation on game maps shows that the automatic decomposition results in intuitive regions of a comparable standard to human-made labeling. Furthermore, we show that our automatic decomposition, when used by a pathfinding algorithm capable of taking advantage of pre-labeled regions, significantly improves search effectiveness.

Introduction

Real-Time Strategy (RTS) games pose interesting challenges for computer-controlled (and human) players. Artificial intelligence (AI) constructed agents must in real-time ceaselessly take a wide range of non-trivial decisions pertaining to both short and long term planning issues. For example, in addition to micro-managing multiple units, an effective AI agent also needs to consider questions such as: how to effectively gather in-game resources, in which order to build units and advance technology, how to secure the home-base, and how to attack the opponents – to name a few. Such decisions are more often than not strongly influenced by geospatial attributes of the game-world terrain, thus requiring some kind of a spatial reasoning. Terrain analysis is thus a vital part of any successful RTS game AI.

Terrain analysis for RTS (and other) video games has thus received considerable research attention in the past (Pottinger 2000; Forbus, Mahoney, and Dill 2002; Brobs, Saran, and van Lent 2004; Björnsson and Halldórsson 2006; Hale, Youngblood, and Dixit 2008; Perkins 2010; Si, Pisan, and Tan 2014). Typically one of the most important steps in such analysis is the decomposition (or partitioning) of the game

map into strategic regions. This is useful for the game AI not only for spatial reasoning at a higher abstraction level than otherwise possible, but also to speed up pathfinding. Computing paths for multiple units in real-time on large game maps is computationally demanding, even for modern-day computer hardware. Furthermore, pathfinding queries are not only useful for unit navigation, but also for assisting with answering some of the aforementioned queries pertaining to strategic planning (e.g., how far to an important resource).

The main contribution of the paper is a new algorithm for decomposing game maps. One key advantage of our algorithm, in addition to its effectiveness, is how intuitive it is conceptually, thus resulting in predominantly human-like partitions. This is a valuable quality as the partitions are more likely to harmonize with the objectives and intentions of the game-map designer(s). Furthermore, using a standard test-suite of game maps from commercial RTS (and role-playing) games (Sturtevant 2012), we provide empirical evidence of the algorithm’s effectiveness by visualizing and contrasting the resulting map partitions to both computer- and human-made ones, as well as by demonstrating how the partitions improve pathfinding efficiency.

The paper is structured as follows. In the next section we introduce relevant background material and the terminology used throughout. The subsequent sections describe the automatic map decomposition algorithm and its empirical evaluation, respectively. Finally, we give an overview of related work before concluding and discussing future work.

Background

We assume grid-based maps of any width and height consisting of *tiles*. A tile can be either *traversable* or *non-traversable* (also referred to as *empty* or *wall*, respectively). A *region* (or *zone*) is a set of connected traversable tiles of any size or shape. The process of *decomposing* (or *partitioning*) a map is to cluster the tiles into meaningful regions. Although adjacent regions may initially have irregular boundaries, we refine them to be line segments (connecting walls), called *gates*. Gates are represented by their end points. The core output of the decomposition algorithm is a list of gates.

In our setting, a meaningful decomposition ideally creates zones that help the game AI make strategic and pathfinding decisions. Conversely, it should avoid creating zones influenced by irrelevant textures and aesthetic structures.

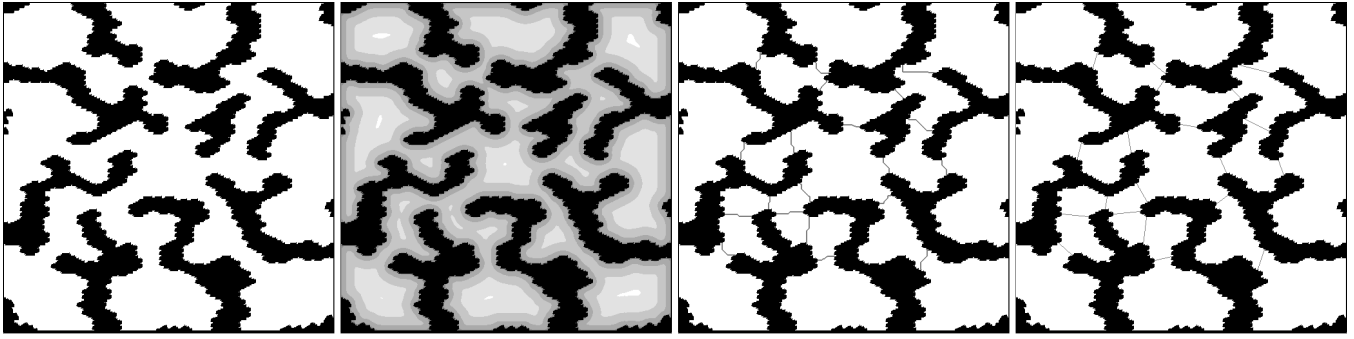


Figure 1: Here we see four images of maps during intermediate stages of the algorithm. The first image is the original map with traversable tiles in white and non-traversable wall tiles in black. The other images show the maps corresponding to the output of Algorithms 1, 2 and 3, respectively.

Decomposition

Our decomposition algorithm is conceptually easy to understand. First, we create a depthmap from the original map, where each non-traversable tile is at a ground level and each traversable tile at a sub-ground level: the further its distance to the nearest non-traversable tile the deeper its level. This depthmap forms a carved out 3D landscape where the traversable tiles form valleys of different depths, possibly separated by ridges. These ridges form candidates for boundaries between regions. Second, our algorithm locates the (most prominent) ridges, for which it uses a technique simulating a rising ground-water level. As the water level rises, lakes start to form and grow in the valleys and gradually start to unite, overflowing ridges. This amalgamation of lakes is used to identify the ridges. The contours of the identified ridges can, as in nature, have some twists and turns. The final step of the algorithm is thus to approximate the ridges by straight line segments, which are easier for the game AI to work with.

Algorithm

We start by building a depthmap where the depth of each tile is a function of its distance to the nearest wall. The further away from any wall a tile is, the deeper its level.

Algorithm 1 Depth mapping

```
for all tiles  $(x, y)$  in map do
    determine depth of tile (see Algorithms 4 & 5)
    write depth into depthmap at  $(x, y)$ 
end for
```

After building the depthmap we begin building a zone map as well as a gate cluster map. The zone map is a grid where each tile has a label; zones are composed of every tile with the same label and the gate cluster map is just used to keep track of which tiles are right on the boundaries between adjacent zones.

The water level starts at the maximum depth found during depth mapping and labels each tile at that depth; a unique label is given to tiles that stand alone, but tiles adjacent to previously labelled tiles inherit their neighbor's label. If a

tile has two or more neighbors with different labels then it is located where two zones meet (on a ridge); the tile is marked as a gate tile and becomes part of a gate cluster.

Algorithm 2 Water level decomposition

```
 $currentWaterLevel \leftarrow maxDepth$ 
while  $currentWaterLevel \geq 0$  do
    for all tiles  $(x, y)$  at depth  $currentWaterLevel$  do
        if  $(x, y)$  has  $> 1$  labelled neighbors then
            if neighbors have different labels then
                mark  $(x, y)$  as gate tile
            end if
            give  $(x, y)$  same label as any neighbor
        else if tile has 1 labelled neighbor then
            give tile same label as neighbor
        else
            give tile new label
        end if
    end for
     $currentWaterLevel \leftarrow currentWaterLevel - 1$ 
end while
```

The final step is building gates from the irregularly shaped gate clusters and adding them to the gate list. Two end points are detected for each gate cluster. Having found these end points we can rasterize straight lines between them and use these to flood-fill the zones again for our final zone mapping, if needed.

Implementation Details

The depth map structure is in fact two structures: a) the depth map; a map indexed on grid position to find the depth of each tile, and b) the depth tile list; a vector indexed on depths where each element is a list of grid positions that have the same depth. This way the depth map can be accessed in near constant time from any part of the algorithm, whether it needs the depth of particular coordinates or the set of coordinates at a particular depth.

To aid in building this depth map faster we use a temporary structure which is indexed on octile distances and has elements which list all (x, y) -grid offsets that add that partic-

Algorithm 3 Build gate list

```

for all tiles  $(x, y)$  in map do
  if  $(x, y)$  is marked as gate tile then
    FloodFill cluster of connected gate tiles
    Remove all tiles not adjacent to wall
    Select one tile from each remaining connected cluster
    Build gate from two selected tiles
    Add gate to list of gates
    Remove remaining tiles from cluster
  end if
end for

```

ular octile distance. When mapping the depth of each tile we check each octile distance, starting at zero, add each offset in that distance's list to the current tile coordinates and check if there is a wall at that location. A further optimization is to not start this offset at zero distance each time, but at one horizontal movement less than the previous depth found.

Algorithm 4 Determine depth of tile - simple version

```

 $(x, y) \leftarrow \text{tile}$ 
 $\text{currentDepth} \leftarrow \text{lastFoundDepth} - 1$  (0 first pass)
while depth not found do
  for all  $\text{offsetCoord}$  of  $\text{currentDepth}$  do
    if  $(x, y) + \text{offsetCoord}$  is wall tile then
       $\text{depth}(x, y) \leftarrow \text{currentDepth}$ 
      break while
    end if
  end for
   $\text{currentDepth} \leftarrow \text{currentDepth} + 1$ 
end while

```

Refinements

One artifact of the algorithm is that it detects gates that close off tiny spaces that have little or no effect on the search strategy, especially in noisy maps and maps with wavy or uneven walls. To reduce this noise we chose to add parameters to the algorithm for tweaking the depth-mapping. The *wall threshold* is used to average out the depths by not registering the depth of a tile as soon as the search finds a wall, but rather after finding a number of walls equal to the threshold. This adds several more iterations of the grid-offset search, but the smoothness in the output outweighs the performance concerns. To further smooth the output we group depth values together by dividing by a *distance denominator* and flooring the result. This makes each depth line wider and helps even out noise in the depth mapping.

Instead of manually setting these parameters, we opted for a fully automated approach by tuning them dynamically at runtime.

This also prevents erratic results when maps are unusually tight and crowded or wide and open. The algorithm dynamically sets the wall threshold for each tile when processing it. The value is a function of the distance to the first wall found and the map size.

The dynamic distance denominator is set to the base 2 logarithm of the distance, floored, resulting in depth areas that are progressively wider the further they are away from walls. This way the algorithm is less likely to close off little useless pockets in the map, but still retaining meaningful details of small and intricate areas within the maps.

Algorithm 5 Determine depth of tile - refined version

```

 $(x, y) \leftarrow \text{tile}$ 
 $\text{currentDepth} \leftarrow \text{lastFoundDepth} - 1$  (0 first pass)
while depth not found do
  for all  $\text{offsetCoord}$  of  $\text{currentDepth}$  do
    if  $(x, y) + \text{offsetCoord}$  is wall tile then
      if first time a wall tile is found then
         $\text{wallThreshold} \leftarrow (\text{maxDepth} - \text{currentDepth}) / 28 + 1$ 
      end if
      if wall tile has been found  $\text{wallThreshold}$  times then
         $\text{depth}(x, y) \leftarrow \lfloor \log_2(\text{currentDepth}) + 1 \rfloor$ 
        break while
      end if
    end if
  end for
   $\text{currentDepth} \leftarrow \text{currentDepth} + 1$ 
end while

```

(Björnsson and Halldórsson 2006) describes a heuristic function for A* pathfinding search that uses precalculated data derived from a decomposition of the map to quickly and closely estimate path distances between locations in the map. It uses gates to separate zones, however, the gates must be either vertical, horizontal, or 45° diagonal. On the other hand, our decomposition method can generate gates of any orientation. Thus, to make our partitioning compatible, we added a pre-processing phase where all gates are rotated so that they have either 0°, 45° or 90° orientation. In each rotation step the the algorithm selects a line-segment end to move such that there is as little change as possible needed to the length and overall position of the gate.

Empirical Evaluation

The effectiveness of our decomposition method is evaluated in three ways, as reported in the following subsections. First, we collect various logistics about the decomposition process. Second, we visualize the resulting partitions and contrast them with those generated by other computerized methods reported in the literature, as well as with those generated by skilled humans. Third, and last, we show the usefulness of the automated decomposition when used with an existing pathfinding algorithm capable of taking advantage of pre-labeled partitions.

Table 1 lists the maps we use for our experiments. They are all, with the exception of the last one, from the Grid-Based Pathfinding Benchmark Test-Suite (Sturtevant 2012). The last map is the demo map used throughout the (Perkins 2010) paper and is included to allow for a direct comparison to that work. The entire pathfinding test-suite contains

Table 1: Map decomposition statistic. The first column is the name of the map, the second column the time in seconds it takes to decompose the map, the third column is the map size, the forth column the number of gates the decomposition algorithm generates, and the last column shows the relative speedup compared to (Perkins 2010). The maps are taken from StarCraft (the first group, 17 maps), Baldur’s Gate II (the second group, 2 maps) and Starcraft II (the third group, 1 map).

Map	Time	Size	Gates	Speedup
AcrossTheCape	7.71	768x768	57	4.1
ArcticStation	10.15	768x768	82	2.8
Backwoods	3.59	768x512	79	4.2
BigGameHunt	2.26	512x512	20	3.1
BlackLotus	5.01	768x768	63	6.9
BlastFurnace	9.55	768x768	68	3.5
BrokenSteppes	12.23	768x768	86	2.8
Brushfire	1.14	512x512	29	4.9
CatwalkAlley	4.56	512x512	98	1.1
Cauldron	18.70	1024x1024	140	7.9
Crossroads	8.91	768x768	54	2.8
DarkContinent	4.92	512x768	46	2.9
Elderlands	10.54	768x768	42	2.0
Enigma	4.16	768x768	64	5.6
FireWalker	1.35	512x384	16	4.3
FloodedPlains	8.83	768x768	85	3.6
GladiatorPits	6.49	768x512	76	3.9
AR0205SR	1.31	512x512	41	-
AR0406SR	0.73	512x512	60	-
Byzantium 3.0	2.35	512x512	29	-

hundreds of maps from various games, but we use only a small subset of those to make it feasible for us to visually inspect all partition results. Most of the maps are from Starcraft, but we also included a few others maps used in other work (Björnsson and Halldórsson 2006; Perkins 2010) to allow for a more direct comparison.

All the experiments were run on a computer with a Quad Core Intel i5 CPU with 16 GB of memory (one core used). We used the offline BWTA2 module at <https://bitbucket.org/auriarte/bwta2> to run and time Perkins’ algorithm. For fairness, we timed only the map decomposition relevant parts of the BWTA analysis (that is, generation and pruning Voronoi diagrams and the subsequent detection of chokepoints and regions).

Decomposition Statistics

Table 1 shows, for each map, the run-time of the decomposition and the number of gates generated. First, we note that it typically takes only a few seconds to decompose a map, and no more than 10-20 seconds for the larger and more computationally demanding maps. This is a sizable speedup compared to (Perkins 2010), where our method runs on average more than four times faster and close to eight times faster on the largest map. Such a fast decomposition approach opens up for the possibility of using the decomposition in real-time settings for dynamically changing maps; for example, when

new regions become reachable (e.g., as in Warcraft when foresting connects new regions) or when regions become non-reachable (e.g., when a bridge collapses). Of course, it is not feasible to run the decomposition too frequently during gameplay, but such partition-altering events only occur sporadically. Second, we note that the number of gates per map is relatively small, which is preferred for these maps in terms of creating human-intuitive regions.

Decomposition Output

We contrasted the partitioning output of our algorithm to that of the partition algorithm’s introduced in (Björnsson and Halldórsson 2006) and (Perkins 2010), as well as to human-made partitioning. Figure 2 shows representative results from that comparison. Essentially, our partitioning looks more intuitive than that of (Björnsson and Halldórsson 2006) and yields very similar partitions to both (Perkins 2010) and the human-made ones.

It is not surprising that we do better than the decomposition method introduced in (Björnsson and Halldórsson 2006), because it was steered towards room-like maps. Also, the focus of that work was primarily on a pathfinding algorithm that uses the partitions, as opposed to the map decomposition algorithm itself. What is more of an interest is that our method generates almost identical partitions to those of Perkins’ state-of-the-art method, despite being both simpler and more computationally efficient. We also recruited a few avid RTS players among our students, all with some game-development background, and asked them to partition four different game maps into regions of interest for a game AI. Not only were the humans surprisingly consistent with their labeling among them selves, but the partitions were also almost identical to the ones produced by our automated decomposition method. The bottom rightmost map in Figure 2 shows one human-made partition and Figure 3 pictures another example.

Decomposition Used by Pathfinding

Finally, we implemented the pathfinding method described in (Björnsson and Halldórsson 2006), which is capable of taking advantage of map partitioning to speed up pathfinding, and ran it on the aforementioned game maps partitioned by our decomposition methods (using the pathfinding benchmark searches provided in the maps scenario files). This resulted in 43% savings in terms of node expansions and 32% saving in run-time, compared to 41% and 17%, respectively, as reported in the (Björnsson and Halldórsson 2006) paper. We report only relative improvements of each individual decomposition, as a comparison based on absolute values would not make much sense because of different implementations, hardware, and maps used. Even so, this comparison needs to be taken with some scepticism as the original paper uses smaller maps on average. However, if we look at their most favorable reported result, where they look only at the top 10% largest paths on their largest map, their decomposition yields 35% runtime speedup. This is close to our average over all paths and maps, with our best case maps and batches saving close to 50%. We can thus (conservatively) conclude that the run-time savings are at least in

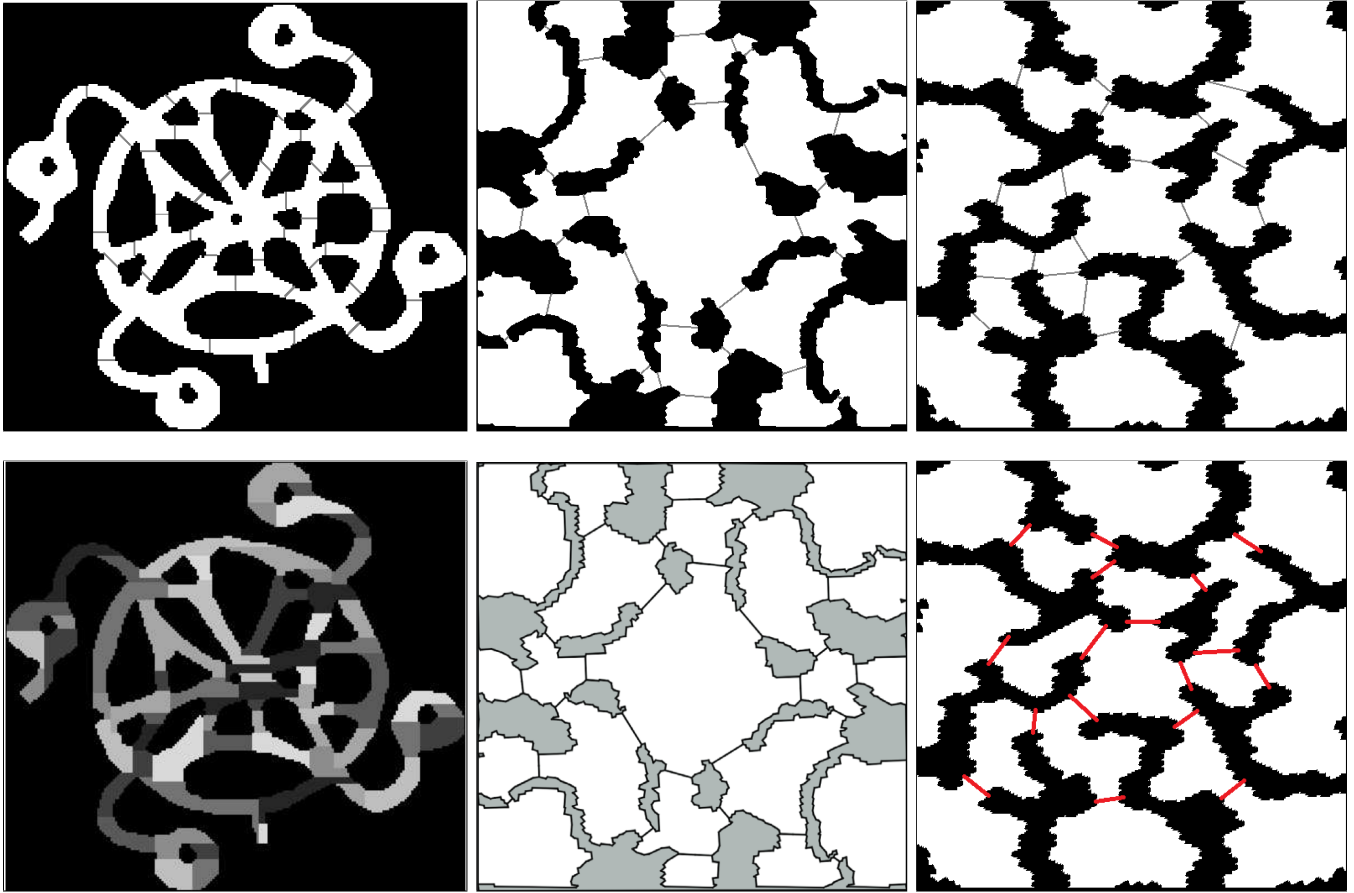


Figure 2: The decomposition of the maps in the top row are done by our method, but the maps in the bottom row are done by contrasting decomposition methods: the one on the left by (Björnsson and Halldórsson 2006), the one in the middle by (Perkins 2010), and the one to the right by humans.

the same ballpark for both decompositions, meaning that our partitioning scheme seems equally well-suited for speeding up pathfinding despite not being specifically designed for that purpose.

Discussions

The empirical evaluations clearly demonstrate the viability of our method for automated map decomposition. Not only does it generate partitions that are intuitive and human-like, but it also compares favorably with an existing state-of-the-art automated decomposition method; that is, it produces similar quality partitions, but in a more computationally efficient manner. The resulting partitioning can also be used to speed up pathfinding.

Related Work

The following two works are the most related to the work we present here. The former is focused on map decomposition for improving strategic decisions and detecting choke-points, whereas the latter introduces both a map decomposition method and a pathfinding algorithm for taking advan-

tage of the decomposition.

In (Perkins 2010) a method is presented for detecting choke points and decomposing a game map into region polygons. It starts by recognizing separate obstacle polygons, using them to build a Voronoi Diagram that is then pruned and evaluated in order to find regions and choking points in the traversible area of the map. These choking points represent the shortest distances between obstacle polygons where congestions could happen when moving great numbers of units through. This method yields a similar result to our algorithm but seems very intricate. It requires the reduction of map tile clusters into polygons before building its decomposition and also needs to prune its results and finally convert back into the original format of the map.

(Björnsson and Halldórsson 2006) describes a search heuristic that uses a decomposition of a grid-based map to better evaluate path lengths and speed up optimal pathfinding searches. By precalculating distances between gates in the decomposed map a better informed search heuristic for A* is constructed, resulting in significant speedup of the pathfinding searches. Whereas the partition-based pathfinding

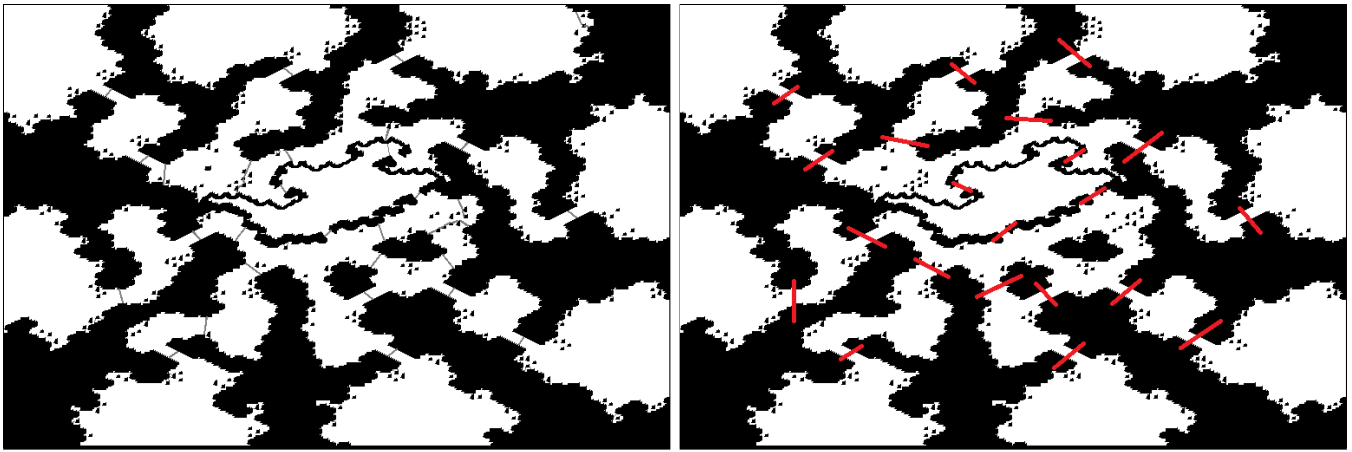


Figure 3: Side by side comparison of the decomposition done by our algorithm (left) and a manual decomposition (right).

algorithm is general and seemingly has a wide applicability, then the decomposition algorithm introduced seems overly targeted towards maps with rectangular structures, such as rooms and hallways. Consequently, it does not seem well-suited for landscape-like maps as seen by it creating an excessive number of regions and gates.

Conclusions and Future Work

We introduced a fully automated method for map decomposition that is both computationally efficient and yields intuitive partitions comparable in quality to the state-of-the-art. Also, an added appeal of the new method is its simplicity and good run-time efficiency.

The effectiveness of our decomposition method was evaluated in three ways: firstly we showed its runtime efficiency and that it typically takes only a few seconds to decompose a map; secondly we showed that the partitions it generates are of a comparable quality to both manual decompositions and that of other state-of-the-art automated methods; finally we demonstrated its usefulness for speeding up pathfinding searches.

As for future work, we plan to empirically test our method on a larger set of more disparate maps to better map the method's strengths and weaknesses. Also, an optimization of the decomposition with specific objectives in mind is an interesting avenue of further research. For example, we noticed that even though the efficiency of the partition-based pathfinding algorithm is not affected by the total number of gates in a map, it is quite sensitive to the maximum number of gates individual zones have (the reason being that the improved heuristic function using the gate information has a time complexity of $O(g^2)$, where g is the number of gates a zone has). Thus, decomposing a map with the objective of creating only zones with a small number of gates could potentially yield additional pathfinding speedups. On similar notes, by profiling usage data from in-game pathfinding and AI decision making one would get a better sense of which zones are the least and most relevant; one could then use the insights gained from the profiling to further refine the auto-

mated decomposition.

References

- Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal pathfinding on game maps. In *AIIDE'06*, 9–14.
- Brobs, P.; Saran, R.; and van Lent, M. 2004. Dynamic path planning and terrain analysis for games. In *"AAAI Workshop: Challenges in Game Artificial Intelligence*, 41–43.
- Forbus, K. D.; Mahoney, J. V.; and Dill, K. 2002. How qualitative spatial reasoning can improve strategy game ai's. *IEEE Intelligent Systems* 17(4):25–30.
- Hale, D. H.; Youngblood, G. M.; and Dixit, P. N. 2008. Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds. In Darken, C., and Mateas, M., eds., *AIIDE*. The AAAI Press.
- Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *AAAI'10*.
- Pottinger, D. C. 2000. Terrain analysis in realtime strategy games. In *Proceedings of Computer Game Developers Conference*.
- Si, C.; Pisan, Y.; and Tan, C. T. 2014. Automated terrain analysis in real-time strategy games. In *FDG'14*.
- Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.