Using Map Decomposition to Improve Pathfinding

Kári Halldórsson

Thesis of 60 ECTS credits
Master of Science (M.Sc.) in Computer Science

December 2015

Using Map Decomposition to Improve Pathfinding

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

December 2015

Thesis Committee:

Yngvi Björnsson, Supervisor
Professor, Reykjavík University, Iceland

Nathan Sturtevant,
Associate Professor, University of Denver, USA

Stephan Schiffel,
Assistant Professor, Reykjavík University, Iceland

Using Map Decomposition to Improve Pathfinding

Kári Halldórsson

December 2015

Abstract

Artificial intelligence in games performs computationally expensive searches in large state spaces, i.e. for pathfinding and strategic decisions. Breaking the state space down into regions, with clear connections, can greatly benefit these algorithms, allowing decision making on a higher level and guiding searches in a more focused way through the search space. We present an improved heuristic for pathfinding search that takes advantage of such decompositions, as well as a fully automated method for identifying meaningful strategic regions in game maps. Empirical evaluation shows that our automatic decomposition method results in intuitive regions of comparable quality to the current state of the art, when run on game maps taken from commercial video games. Its implementation also runs faster than the current standard and the approach is conceptually intuitive and readily understandable. Furthermore we show that significant improvement can be made to pathfinding search effectiveness using an algorithm that takes advantage of the map decomposition.

Svæðaskipting korta til að bæta leit að stystu leið

Kári Halldórsson

desember 2015

Útdráttur

Gervigreind í leikjum inniheldur stórar leitir í sérhæfðum leitarrýmum, m.a. til að finna stystu leið í korti og við áætlanagerð. Niðurbrot leitarrýmisins niður í svæði með skýrar tengingar getur bætt slík reiknirit verulega með því að leyfa ákvarðanatöku á hærra plani og stýra leitum á nákvæmari hátt gegnum leitarrýmið. Við kynnum bætt matsfall fyrir leit að stystu leið sem notfærir sér slík niðurbrot, og auk þess alsjálfvirka aðferð til að skipta leikjakortum upp í svæði á skilmerkilegan hátt. Tilraunir keyrðar á kortum úr tölvuleikjum sýna að sjálfvirka niðurbrotsreikniritið skilar skýrum svæðum af sambærilegum gæðum og það sem best þekkist. Útfærslan okkar keyrir líka hraðar en núverandi aðferðir og er þar að auki auðskiljanleg og einföld í útfærslu. Ennfremur sýnum við fram á að hraða leitar að stystu leið í korti má bæta umtalsvert með reikniriti sem tekur mið af slíku niðurbroti korts.

Using Map Decomposition to Improve Pathfinding

Kári Halldórsson

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

December 2015

Student:

.....
Kári Halldórsson

Thesis Committee:

.....
Yngvi Björnsson

.....
Nathan Sturtevant

.....
Stephan Schiffel

Acknowledgements

This research was supported by a grant from The Icelandic Research Fund (RANNIS). Acknowledgements also go out to Jónas Tryggvi Jóhannsson who did an early implementation of the dead-end heuristic and Gabríel Arthúr Pétursson who wrote the UI and graphics for our analysis tool. BioWare Inc. kindly provided the Baldur's Gate II maps.

Preface

The research on improved heuristics for pathfinding search described in Chapter 2 was previously published in *Improved Heuristics for Optimal Pathfinding on Game Maps* by Yngvi Björnsson and Kári Halldórsson [1].

The research on map decomposition described in Chapter 3 was previously published in *Automated Decomposition of Game Maps* by Kári Halldórsson and Yngvi Björnsson [2].

A graphics user interface to analyze and edit decompositions, and visualize pathfinding searches step-by-step was written by Gabríel Arthúr Pétursson as part of an independent study in the spring of 2015. It is referred to in the text as our *analysis tool*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-Time Strategy (RTS) games pose interesting challenges for computer-controlled (and human) players. Modern computer game worlds are getting larger and more complex every year, both in terms of the map size and the number of units existing in the world. In RTS games there can be hundreds of units navigating the world simultaneously, and *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. 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. Also, calculating paths for all these units in real-time is computationally demanding.

In RTS games the AI decisions are more often than not influenced by geospatial attributes of the game-world terrain. Terrain analysis is thus a vital part of any successful RTS game AI and an important component of such analysis is to decompose the game map into strategic regions, for example with respect to suitability for building a home base or setting up military outposts. The connections between such regions are of a special importance, for example the presence or absence of choke-points when mobilizing multiple units at once. A strategically decomposed game map thus is beneficial not only for tactical decision making, but also for making unit pathfinding and navigation more effective.

In this thesis we devise new methods for addressing the aforementioned tasks. On the one hand we introduce new pathfinding heuristics that use map decomposition information to significantly speed up the computation process, while preserving path optimality. On the other hand a new computationally effective algorithm is introduced for automatically decomposing game maps into human intuitive strategic regions.

The thesis is structured as follows. Chapter 2 describes heuristics that benefit from a good spatial abstraction of the search space, primarily the *gateway heuristic* which uses pre-calculated distances between *gates* in the map to guide and speed up an A* pathfinding search. Chapter 3 describes an automated decomposition method that uses distances between traversable and non-traversable (wall) tiles in a map to group tiles into areas in

order to build a set of gates between them, which can in turn be used by various AI systems, including the gateway heuristic A* search. In Chapter 4 we empirically evaluate the pathfinding and decomposition in conjunction on a wide selection of computer game maps. Finally we discuss related work and conclude.

Background

We assume grid-based maps of arbitrary width and height consisting of *tiles*. A tile can be either *traversable* (also *empty*) or *non-traversable* (also *wall* or *obstacle*). 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 term *decomposition* is used throughout the text, to describe an abstraction of the game map state space used in our work, or the action of building such an abstraction, either manually or automatically. 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 purely aesthetic structures.

The process of finding a valid path between two tiles, start and goal, in a map is called *pathfinding*. Each traversable tile has a set of possible *moves*, each move taking the current state to an adjacent tile by moving either horizontally or vertically, at cost 1, or diagonally, at cost $\sqrt{2}$, and the product of the search is a list of moves between adjacent traversable tiles, forming a *path* between the start and goal tiles.

The *de facto* industry standard for pathfinding in games is the A^* algorithm [3]. Whereas the state-space representation may differ from game to game (a grid or a mesh both being common), A^* search or a variant thereof is generally the algorithm of choice. Informed search methods such as A^* use a *heuristic*, a function that returns an estimate of the length of a shortest path from the current state to a goal state. Generally, the closer the heuristic value is to the true length of the shortest path, the less exploration the search will perform. A *perfect heuristic* is a heuristic that always returns the correct length of the shortest path and will result in a search method that never explores a node outside the actual shortest path.

In order to make sure an A^* search is *optimal*, that it always finds an actual shortest path, the heuristic must never overestimate. A heuristic that always returns an estimate equal or shorter than the true length of the shortest path from every state to every possible goal is called *admissible*. If, for every two adjacent tiles, the cost of going from one to the other added to the the heuristic estimate for the second one is never greater than the heuristic estimate of the first one alone, the heuristic is also termed *consistent*. An A^* search using an inconsistent heuristic must expect the possibility of re-expanding a search node, that is re-estimating a tile with a different path towards it, from the start, but when using a consistent

heuristic it is guaranteed that once a node has been expanded, the shortest path to that node has been found, and so subsequent possibilities of travelling through that tile can be ignored.

A common heuristic in grid-based pathfinding is the *octile* distance, similar to a manhattan distance but allowing diagonal moves, which is a correct estimate of the shortest path, given that there are no obstacles in the map. The octile distance is both admissible and consistent, but in maps with many obstacles or complex wall structures such a simplistic heuristic often cannot offer sufficiently targeted guidance, resulting in the search exploring almost the entire map when finding a shortest path between two distant map locations.

Chapter 2

Improved Heuristics in Pathfinding

As stated earlier, the octile distance can be an overly simplistic estimate for path lengths in large and complex grid maps. One technique used to overcome this problem is *hierarchical pathfinding*. Instead of having only a single representation of the state space, additional higher-level abstractions are used as well. Each level in the hierarchy uses an increasingly abstract view of the game map, and can subsequently be represented using a smaller state space. When answering a pathfinding query an approximate path is found in one of the higher-level layers (and then possibly refined using small local searches in the base layer). This results in much faster processing because A^* searches a smaller state space. The main drawback of this approach is that the paths returned are not necessarily optimal. This is because some of the finer details of the map typically get lost in the abstraction process. However, this is generally of a little consequence for game-play if the paths are only slightly sub-optimal. Fortunately, this is most often the case. However, with increased number of units and other dynamic obstacles on the map the risk of the paths becoming seriously sub-optimal increases. This is because a search performed in an abstract state-space usually does not (and cannot) take these dynamic obstacles into account.

Our approach reduces state-space exploration while still making it possible to account for dynamic obstacles. Instead of using state-space abstraction to create hierarchical views, we use it to provide an improved heuristic function for guiding a regular A^* search. The challenge is to devise heuristics that can be computed efficiently, yet provide improved search guidance. We introduce two such new heuristics, both of which are admissible and thus preserve optimality.

In the next section we describe the new heuristic functions and provide both detailed examples and pseudo-code. Following is a section summarizing the results of our original empirical evaluation of the heuristics using real game maps. Finally there is a summary of our work on pathfinding heuristics.

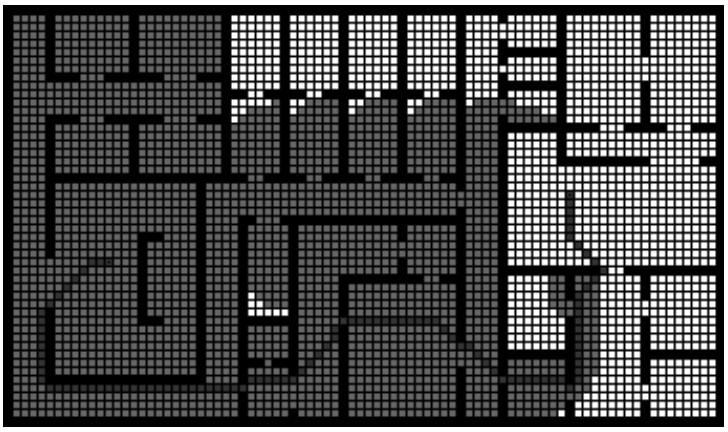

Figure 2.1: Example map: locations explored by A^* are shown in dark gray.

2.1 Improved Heuristics

The map in Figure 2.1 depicts an outdoor/cave scene typical of a role-playing game. The world consists of multiple areas that are connected via narrow passes and tunnels.

When finding a shortest path between two distant locations in this map a naive heuristic based on octile distance would explore more or less all the locations on the map. This is in part because it has no way of telling beforehand whether there exists a pathway through any given room that leads to a shortcut to the desired destination. To demonstrate this better we have marked in dark gray all the tiles in the map that A^* using the octile heuristic explores when finding an optimal path between two far apart locations. The optimal path is shown in darker gray, the start is to the left and the goal to the right. The algorithm spends a lot of effort exploring areas that — as is immediately obvious to us — cannot possibly be relevant, because they result in either dead-ends or clearly inferior paths.

The first heuristic presented here, the *dead-end heuristic*, seeks to alleviate this problem by identifying and excluding beforehand all areas (in our case rooms) that cannot possibly be on an optimal path between two given locations. It avoids areas that lead to a dead-end by excluding these entire areas from the search space.

The idea behind the second heuristic, the *gateway heuristic*, is to better guide the search by storing pre-calculated distances between certain key points, or gates, in the map and then use it to better recognize which areas are more likely to lead to optimal paths than others. Computing the heuristic is a two-phase process. In the first phase the map is preprocessed and an abstract view created. This is done by automatically decomposing the map into smaller areas and then computing path information. This calculation is done offline and only once for each map. In the second phase the abstract view from the preprocessing phase is used to derive improved heuristic estimates for the pathfinding search. The heuristic is calculated in real-time and efficiency is therefore important.

2.1.1 Dead-End Heuristic

The dead-end heuristic can immediately tell if the search enters a room which eventually leads to a dead-end, that is, there is no pathway from this room to the goal (except back out via the entrances we came in through). Clearly there is no need to explore such rooms.

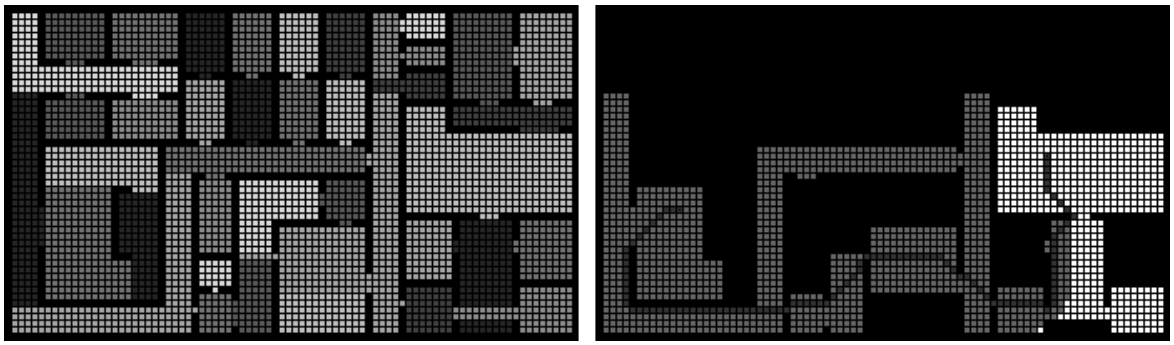

Figure 2.2: Dead-end heuristic: area decomposition (left) and relevant areas and nodes explored (right).

2.1.1.1 Preprocessing Phase

The preprocessing phase continues in two steps. In the first step the game map is decomposed into several smaller areas, representing in this case rooms and corridors. The result of running our ad-hoc decomposition algorithm (See Appendix B) on the example map is shown to the left in Figure 2.2.

The second step in this phase is to construct a high-level graph for representing the different areas and the inter-connections between them. A node in the graph represents an area and an edge between nodes represents an entrance between the two corresponding areas. Note that there are possibly more than one entrance connecting the same two rooms, resulting in more than one edge connecting a pair of nodes in the graph. The graph is therefore a so-called undirected *multi-graph*. The graph along with the area information is stored with the game map.

2.1.1.2 Runtime Phase

When the map is loaded into the game, the data from the preprocessing phase accompanies the map. This does result in some additional memory usage, but with a careful implementation this can be minimized.

When we get a pathfinding query asking for the shortest path between a start and a goal location, two searches are performed. First a search is performed in the multi-graph to identify the subset of areas in the map that are relevant for the query; other areas, the so-called dead-end areas, can be excluded from the pathfinding search altogether. Let nodes S and G in the multi-graph stand for the nodes representing the areas holding the start and

goal locations in the map, respectively. We do a search in the multi-graph to find *all* possible paths from node S to node G to identify the relevant areas. Note that during this search we need to mark all edges we have visited to prevent loops and other duplicate search effort. A simple depth-first search proved the most effective for this task, both because of how small the multi-graph is and the fact that we have to find all possible paths.

Once we have identified the subset of relevant areas a regular A^* like pathfinding search is performed. The only difference is that we use an improved heuristic function that returns a value of infinity for grid cells that are located in non-relevant areas. This can be done quite effectively. Each grid cell is marked by the area it belongs to (using a few extra bits) so we can trivially in constant time ask if the area is relevant. One of the main strengths of the dead-end heuristic is that it can be computed very efficiently.

This approach is fundamentally different from hierarchical pathfinding because we have not committed to any high-level path beforehand. For example, in hierarchical pathfinding, if such a high-level path is blocked by a dynamic obstacle this typically does not get noticed until in the path-following phase, and the search may have to be executed again. In our case however, other possible paths are kept open and the A^* search will find another path if one exists.

The effectiveness of this method in terms of reducing exploration of the state-space depends greatly on the structure of the map. On the one hand, for maps consisting mainly of areas connected via relatively few possible pathways, this simple heuristic has the potential of giving significant improvements. However, the more alternative pathways there are the less effective the heuristic becomes. For example, if we were to open up a new pathway through the top rooms in our example map, then the dead-end heuristic would be able to eliminate only a few small areas from the search.

Also, one needs to be a bit careful with the automatic decomposition of the map because if the generated areas become too small, the abstract multi-graph will be large. The overhead of the multi-graph search may then become significant. This overhead can of course be avoided in real-time by preprocessing all the relevant area calculations, although at the cost of extra memory usage.

The heuristic we introduce next suffers from neither of the above problems.

2.1.2 Gateway Heuristic

The gateway heuristic (GH) pre-calculates the distances between entrances/exits of the areas. It also proceeds in two phases.

2.1.2.1 Preprocessing Phase

The map is decomposed into areas in an identical way as for the dead-end heuristic. The original version of this heuristic expects gates to be either vertical or horizontal. In the re-

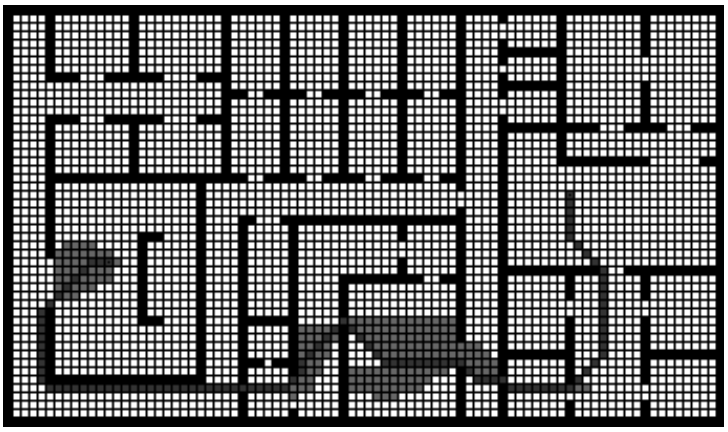

Figure 2.3: Nodes explored by the gateway heuristic. The error in the heuristic is due to the length of gates.

implementation we use in Chapter 4 they can also be diagonal at an angle of exactly 45°. Next we use multiple A^* searches or multi-goal breadth-first searches to pre-calculate the (static) shortest distance between gates. For each gateway we calculate the path distance to *all* the other gateways (cost of infinity if no path exists). Alternatively, one could calculate only the distances between gateways within each room and then use a small search to accumulate the total cost during run-time. However, our approach results in more accurate heuristic estimates and faster run-time access (admittedly though at the cost of extra memory).

In this implementation of our approach, four different costs are stored for each pair of gateways (in-in, in-out, out-in, out-out). Each gate is 2-way because we are interested in knowing separate distances for each possibility of departing from and arriving to a gate. This refinement can in some maps decrease the number of nodes expanded, but it is by no means necessary for the approach to work.

The reason for this is that while the shortest distances between gates are accurate, as they are preprocessed and stored, and in general the octile distance within a single zone is correct, the distance travelled along the length of a gate is absent in GH's calculated value. The search can thus be pulled towards a gate because the best heuristic value will be derived from the shortest path to one end of a gate and the shortest path from the other end of the gate to the goal, not taking into account the cost of moving along the length of the gate. By forcing the gates to be used only as an exit in the preprocessing searches, we eliminate this type of underestimates. This is only of consequence when the map decomposition includes many long gates. If only short gates are used, which is in general preferable, the 2-way gate refinement is unnecessary.

In this original implementation of the gateway heuristic we used 2-way gates and four separate pre-calculation pathfinding searches for each pair of gates (this is done offline so

the extra time does not affect the search time). In these pathfinding searches we are *not* allowed to pass through the departing and arriving gates.

2.1.2.2 Runtime Phase

The runtime phase is a regular A* search that uses the heuristic function below:

$$h^G(n, g) = \min_{i,j}(h^l(n, G_i) + H(G_i, G_j) + h^l(G_j, g))$$

The heuristic $h^l(n, G)$ calculates the octile distance from grid cell n to the nearest point in gate G. This can be computed trivially as a distance from a point to a horizontal/vertical line. The term $H(G_i, G_j)$ stands for the pre-calculated shortest distances between gateways G_i and G_j (in practice we would also have to pass in the gate directions but we have omitted that from the notation here for clarity). Finally $h^l(G, g)$ calculates the shortest octile distance from gate G to the goal tile, g. We need to look at all gates in the current area and compare each of them to all gates in the goal area, and take the minimum cost.

The accuracy and computing efficiency of GH is independent of the total number of gates (although that affects the memory usage). The efficiency of computing the heuristic estimates is mainly affected by the number of gates in the areas we pass through, in particular the area where the goal resides. This is because at each state we select the minimum estimated distance among all pairs of gates with the former gate in the current room and the latter in the goal room (see the heuristic function equation). The heuristic accuracy, on the other hand, is affected by two things: the shape of the rooms and the size of individual gates. Because we use the octile heuristic for estimating the distance from the current state (and the goal) to the nearest gate, we are prone to underestimate errors introduced by the octile heuristic. However, because short distances are typically being estimated, these underestimates will not have a significant effect on the overall distance estimate. Also, the area decomposition algorithm used in the original implementation tends to split maps up into convex areas where the octile heuristic gives accurate estimates. The other type of underestimation taking place has to do with the gate sizes. When calculating distances from a state to a gate we always use the closest point on the gate to ensure admissibility. This is not necessarily the same gate point that was used in our gate distance pre-calculations. The distance between these two points is a source of underestimation. The larger a gate is, the further we risk these two points being apart. In Chapter 3 we introduce a map decomposition algorithm that seeks to minimize the average length of gates while still splitting the area into relevant and generally convex regions.

In addition to causing an underestimation in the heuristic, the fact that the length of the gates is ignored causes GH to be inconsistent. This inconsistency is only present between tiles in different regions, but it is enough to cause a fair amount of re-expansions, as we will see in later evaluation in Chapter 4.

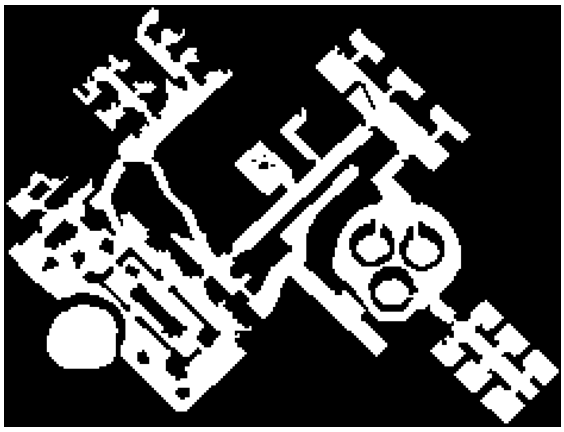

Figure 2.4: The largest game map (244 x 192).

The gateway heuristic is not dependent on whether there are dead-ends or not in the map. If the map is well divided the heuristic will pull the search quickly towards the correct gateways and hence onto the shortest path. Take for example a situation where a path needs to be found between opposite banks of a river and the only way to cross is over a bridge some way to either direction. An A* search using only octile distances as a heuristic will search almost the full distance from start to the bridge in all directions, including straight away from the goal. The dead-end heuristic will not be able to eliminate much of the search space since this is a relatively open area and therefore not do much better. If however both ends of the bridge are considered gates in the gateway heuristic the heuristic value will be the sum of an estimate from start to the bridge, the pre-calculated length of the bridge and the estimate from the bridge to the goal. This estimate is much better than the relatively straight line of the octile distance and the search will therefore hardly expand a single extra node.

2.2 Initial Evaluation

We evaluated the effectiveness of the new heuristics by running them on computer game maps, both created by us and taken from popular commercial role-playing games. All experiments were run on 3.0 GHz CPU personal computers.

Table 2.1 shows the result of our pathfinding experiments where the octile and the two new heuristics are compared. On each map 1,000 searches were performed using randomly chosen start and goal positions. The top section includes experimental data from searching our demo map (Figure 2.2) and the middle section data from nine different maps from the popular game *Baldurs Gate II* (Figure B.2). In the last section we show separately data for a particularly large game map, also from *Baldur's Gate II* (Figure 2.4). Horizontal and vertical moves have the cost of 100 whereas diagonal moves were rounded to a cost of 150.

Table 2.1: Pathfinding statistics (averages).

	Demo map	**Octile**	**Dead-end**	**Gateway**
all	path cost	7430	7430	7430
	estimate	3940	3940	7241
	nodes	955	579	220
	time (ms.)	18.6	14.7	13.2
top 10%	path cost	14373	14373	14373
	estimate	6605	6605	14179
	nodes	2397	1352	487
	time (ms.)	42.9	30.4	28.0
	Game maps	**Octile**	**Dead-end**	**Gateway**
all	path cost	10339	10339	10339
	estimate	7788	7788	9884
	nodes	1231	1120	723
	time (ms.)	27.3	24.6	22.6
top 10%	path cost	20468	20468	20468
	estimate	13290	13290	19731
	nodes	3701	3370	2313
	time (ms.)	69.2	60.7	54.5
	Large map	**Octile**	**Dead-end**	**Gateway**
top 10%	path cost	30463	30463	30463
	estimate	17201	17201	30002
	nodes	5961	4536	2361
	time (ms.)	110.1	84.0	71.3

In all map types the new heuristics are on average clearly superior to the standard octile heuristic, both in terms of number of nodes expanded and total running time. Overall, the gateway heuristic is the best. We can also see that the time overhead in calculating the dead-end heuristic is close to negligible because the time saving corresponds roughly to the node savings. This was achieved because the multi-graphs paths were pre-calculated. For the gateway heuristic the node reductions are particularly impressive. The search time does however not decrease relatively as much as the number of nodes expanded. This is due to the complexity of the new heuristic functions compared to calculating the octile distance. The time savings are none the less significant, and may be further improved with a careful implementation.

We were also interested in looking closer at how the heuristics perform on longer paths. The *top 10%* sections give the result for longer than average paths (for each map we randomly generated 10,000 paths and included the 10% longest)

These are the paths that are likely to cause a problem. The performance improvement of the new heuristics is now even more profound. Also of interest is to see how close the gateway heuristic estimates are to the true path lengths.

2.3 Summary

We presented two admissible heuristic functions for guiding heuristic search in pathfinding on large game maps. The initial results with these heuristics are promising, showing that both heuristics outperform the standard octile distance heuristic. The results in this chapter are from [1] and are somewhat outdated. Also, the map decomposition algorithm used in that work was ad hoc and tailor made for room-like structures (see Appendix B). In Chapter 4 we do rerun some of the experiments using contemporary computer hardware and larger and more up-to-date game maps. Also, we use a more sophisticated map decomposition algorithm, introduced in the next chapter.

Chapter 3

Automatic Decomposition of Game Maps

Terrain analysis for RTS (and other) video games has received considerable research attention in the past [1], [4]–[9]. 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 queries pertaining to strategic planning (e.g., how far to an important resource).

In this chapter we describe 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).

3.1 Decomposition

Our decomposition algorithm is conceptually easy to understand. First, we create a depth-map 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 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

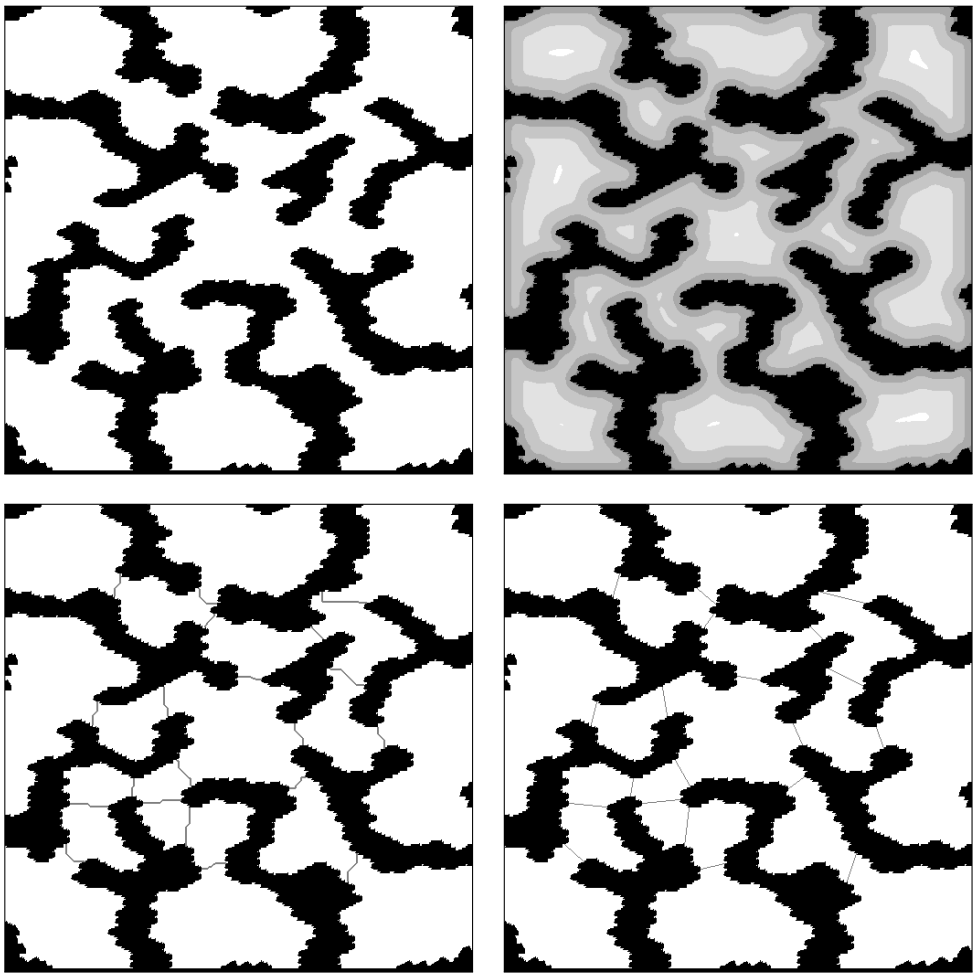

Figure 3.1: Four images of maps during intermediate stages of the algorithm's processing. The image in the top left corner is the original map with traversable tiles in white and non-traversable wall tiles in black. The other images, top-right, bottom-left and bottom-right show the maps corresponding to the output of Algorithms 1, 2 and 3, respectively.

to work with. A more detailed explanation of the algorithm follows, and the main steps are visualized in Figure 3.1.

3.1.1 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, as shown in algorithm 1.

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.

Algorithm 2 shows the water level decomposition where the water level is a variable starting at the maximum depth found during depth mapping. Each tile at that depth is labelled; 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 ← maxDepth
while currentWaterLevel ≥ 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 ← currentWaterLevel − 1
end while
```

The final step, shown in algorithm 3, 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.

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
```

3.1.2 Implementation Details

The depth map 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 particular 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

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

3.1.3 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 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

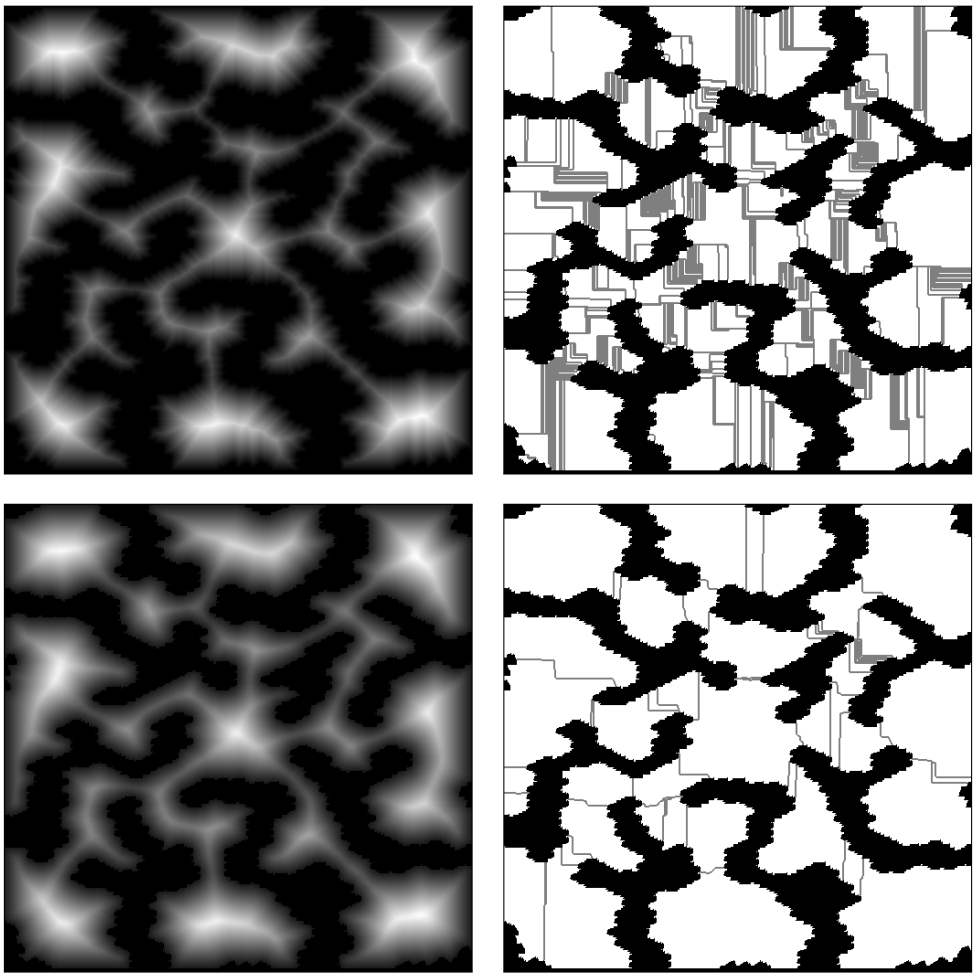

Figure 3.2: Images of two different depth mappings and the resulting water level ridges which become gates. At the top is a pure distance to closest wall tile and at the bottom the dynamic wall threshold is used alone.

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 4 shows a way of determining the depth of each tile without any refinements. Compare that to algorithm 5 which shows the same process, except with the two aforementioned refinements included.

The previous chapter 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°. 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 algorithm selects a line-segment end to move

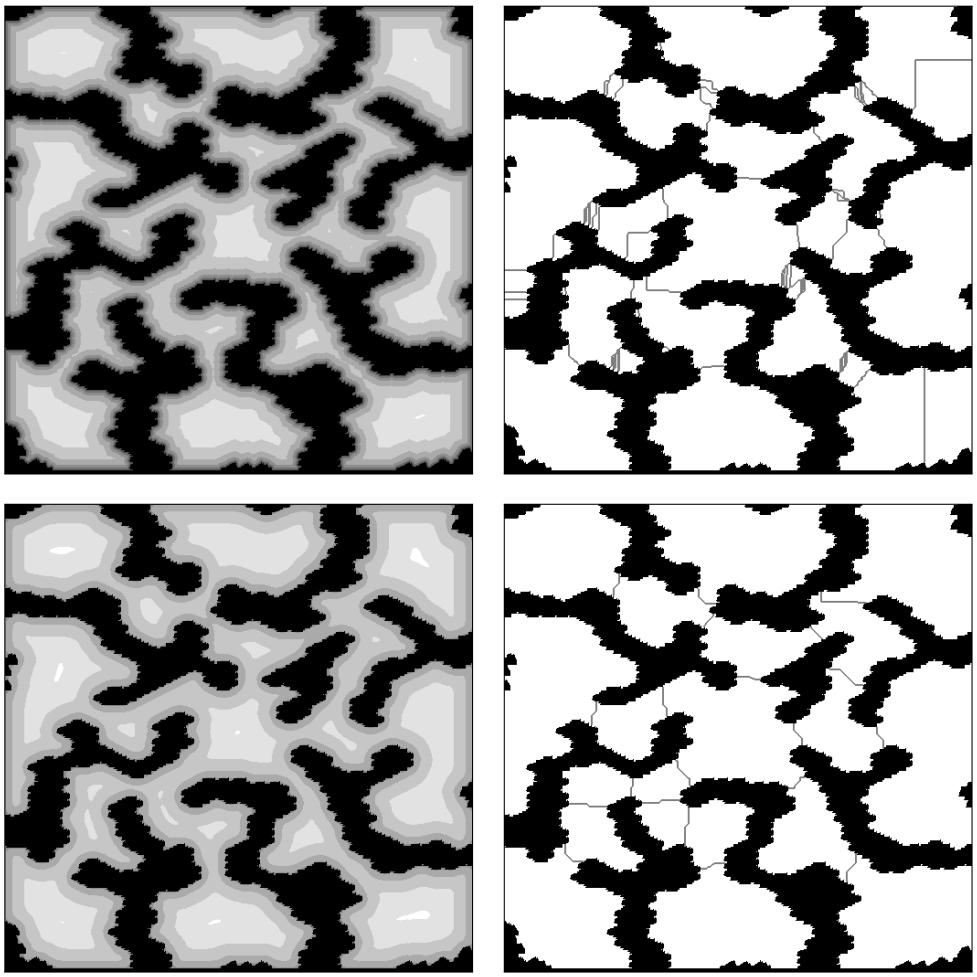

Figure 3.3: Images of two different depth mappings and the resulting water level ridges which become gates. At the top the dynamic distance denominator is used alone and in the bottom images we use the dynamic wall threshold and the dynamic distance denominator together which is the method used throughout the final executions.

such that there is as little change as possible to the length and overall position of the gate. This refinement has no significant effect on speed, but without it the evaluation function can overestimate distances resulting in non-optimal path lengths in some cases. To keep the heuristic admissible, when used in pathfinding searches, this rotation is necessary.

In order for the precalculated data to remain within a manageable size it is important not to create redundant or unnecessary gates. It is also important to keep the average width of the gates small as the maximum error of the heuristic depends on the size of the gates. Thus we seek to have our algorithm split the map where it is obviously of help to the AI systems that use it, but to make as few splits as possible beyond that.

3.1.3.1 Suboptimal decompositions

There are a few rare cases where the decomposition can have a negative effect on the search speed. This happens mainly if one of the following two properties applies. The first case is

Algorithm 5 Determine depth of tile - refined version

```
(x, y) ← tile
currentDepth ← lastFoundDepth − 1 (0 first pass)
while depth not found do
  for all offsetCoord of currentDepth do
    if (x, y) + offsetCoord is wall tile then
      if first time a wall tile is found then
        wallThreshold ← (maxDepth − currentDepth)/28 + 1
      end if
      if wall tile has been found wallThreshold times then
        depth(x, y) ← ⌊log2(currentDepth) + 1⌋
        break while
      end if
    end if
  end for
  currentDepth ← currentDepth + 1
end while
```

where a single zone is connected to many gates. An example of this could be a long hallway connecting to many rooms, where the hallway ends up as a single zone in the decomposition. When either the goal or the currently evaluated state is in this zone the amount of precalculated distances that need to be compared becomes very high and the time for each node expansion grows. If a zone like this sees a lot of traffic - if it is a big part of many search paths - the average expansion time can get high enough to negate the effect of significantly fewer expansions, even resulting in the method becoming slower than the traditional A*.

The second case is when gates become very wide, especially in large open maps where the traditional octile distance heuristic is giving very good evaluations to begin with. Then the large gates result in uninformed heuristics, thus not helping in guiding the search. In both these cases it is possible to fix the decomposition manually, in the first case by adding gates, splitting these super connected zones into smaller zones with fewer connections each, and in the second case by removing these long gates allowing the large open zones to remain undivided, as the regular octile distance is usually a good estimate within them anyway.

3.2 Initial Evaluation

In order to provide empirical evidence of the algorithm's effectiveness we select maps from a standard test-suite of game maps from commercial RTS (and role-playing) games [10], run the algorithm on each one, then visualize and contrast the resulting map partitions to both computer- and human-made ones, as well as demonstrating how the partitions improve pathfinding efficiency. First, we collect various logistics about the decomposition process,

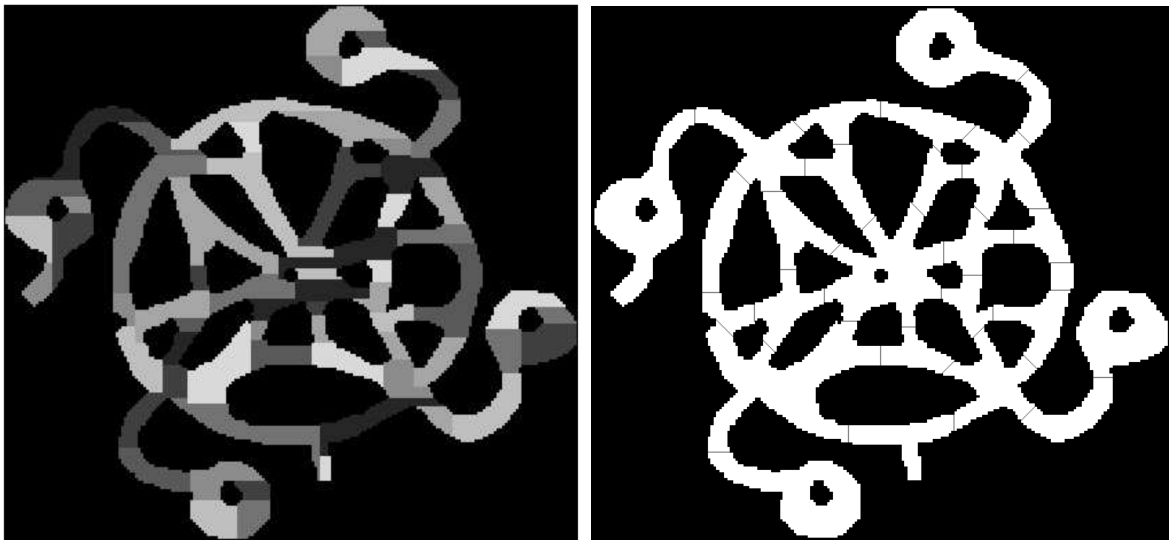

Figure 3.4: The decomposition used in the original improved heuristic study, zones shown in different colors vs. our decomposition, zones split by gates shown as grey lines. Not only does it look cleaner and more intuitive, but it also resulted in a further 20% performance gain over an already improved search method.

Table 3.1: Map decomposition statistics.

Map	Time (s)	Size	Gates	Speedup	low-res time (s)
AcrosstheCape	7.71	768x768	57	4.1	0.14
ArcticStation	10.15	768x768	82	2.8	0.40
Backwoods	3.59	768x512	79	4.2	0.07
BigGameHunters	2.26	512x512	20	3.1	0.044
BlackLotus	5.01	768x768	63	6.9	0.12
BlastFurnace	9.55	768x768	68	3.5	0.17
BrokenSteppes	12.23	768x768	86	2.8	0.27
Brushfire	1.14	512x512	29	4.9	0.045
CatwalkAlley	4.56	512x512	98	1.1	0.12
Cauldron	18.70	1024x1024	140	7.9	0.66
Crossroads	8.91	768x768	54	2.8	0.19
DarkContinent	4.92	512x768	46	2.9	0.12
Elderlands	10.54	768x768	42	2.0	0.19
Enigma	4.16	768x768	64	5.6	0.12
FireWalker	1.35	512x384	16	4.3	0.03
FloodedPlains	8.83	768x768	85	3.6	0.26
GladiatorPits	6.49	768x512	76	3.9	0.13
AR0205SR	1.31	512x512	41	-	-
AR0406SR	0.73	512x512	60	-	-
Byzantium 3.0	2.35	512x512	29	-	-

then 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.

Table 3.1 lists the maps we use for our experiments. Apart from the last three maps, we used two versions of each map, one from the Grid-Based Pathfinding Benchmark Test-Suite [10] and the other the buildable tile structure of the map, which is of a lower resolution, with each tile being 16 (4x4) tiles from the original map. We compare the running time of the decomposition algorithm to the BWTA module [11] which uses the method from [8] and runs its analysis on the full resolution map, but we wanted to compare running our algorithm directly on the low-res buildable map as well. Figure 3.7 shows the similarities in output between running on the different resolutions. Our dynamic refinement variables adapt to the resolution or tightness of a tile's surroundings in a map and so no changes are done in the algorithm or its input variables between the runs. However, since there is detail lost, we are not confident enough to base our decomposition, especially for pathfinding search, on these maps, but it does give a certain indication of possible future optimizations.

Looking at Table 3.1, the first column shows the name of the map, the second column the time in seconds it takes to decompose the map and the third column is the map size. The forth column shows the number of gates the decomposition algorithm generates, and the last column shows the relative speedup compared to [8]. The final column is the time it takes to decompose the maps using the lower resolution buildable tile structure of the map. 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). The last map (Byzantium 3.0) is the demo map used throughout the [8] paper and is included to allow for a direct comparison to that work. The entire pathfinding test-suite contains 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. We included the Baldur's gate maps (ARxx) to allow for a more direct comparison to the ad hoc decomposition algorithm in [1].

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 [11] 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).

3.2.1 Decomposition Statistics

Table 3.1 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 10-12 seconds for the larger and more computationally demanding maps, and in one case around 18 seconds, on a very large and complex map that took almost eight times as long to run in BWTA. This is a sizable speedup compared to [8], where our method runs more than four times faster on average and close to eight times faster on the largest map, using the full resolution version of the map.

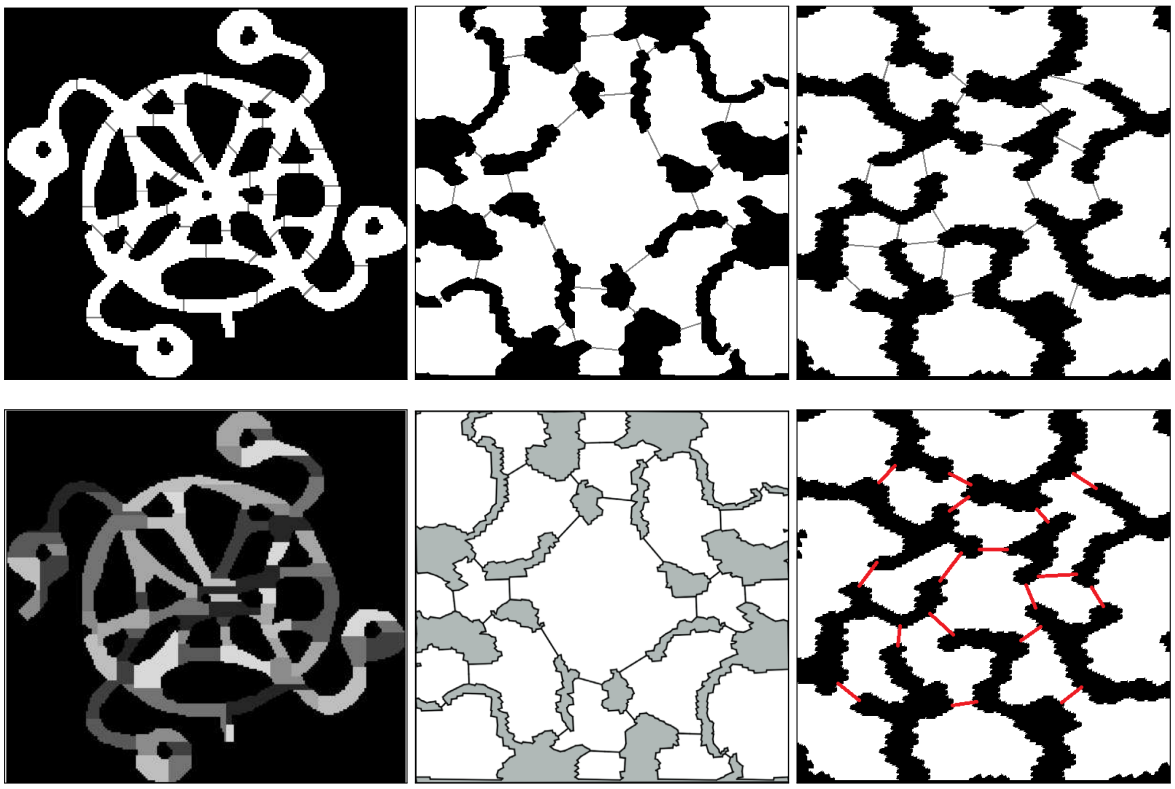

Figure 3.5: The maps in the top row are decomposed using our method, but the maps in the bottom row are done by contrasting decomposition methods: the one on the left by [1] (see Appendix B), the one in the middle by [8], and the one to the right by humans.

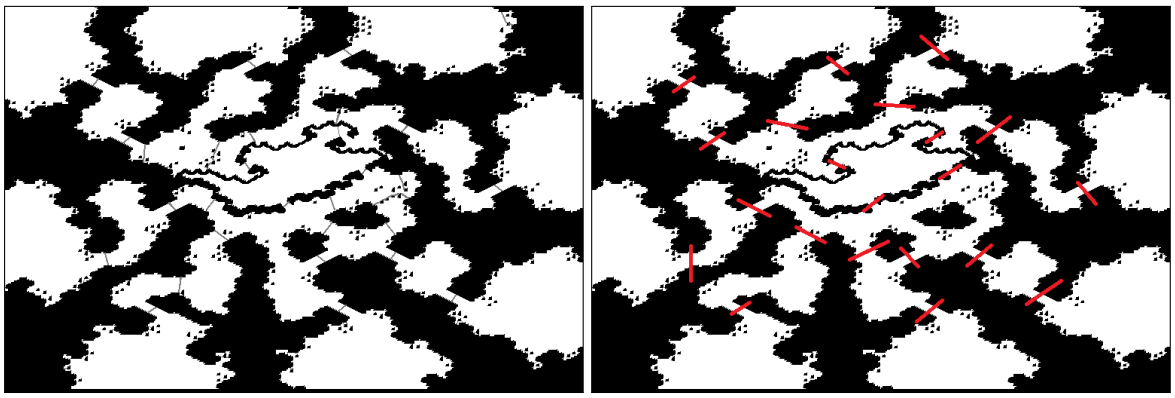

Figure 3.6: Side by side comparison of the decomposition done by our algorithm (left) and a manual decomposition (right).

We also experimented with running our algorithm on the low-res version of the maps, in order to determine if the results were comparable to running it on the full resolution maps. Figures 3.7 and 3.8 shows comparisons between the outputs on these different resolutions. While we cannot at the moment confirm that the full detail of navigability in the maps is kept in the low resolution maps, the resulting decomposition is very similar, and could probably be transferred to the full resolution map. This is achieved because of the dynamic nature of the noise-reducing refinement variables, used when the depth map is built. When run

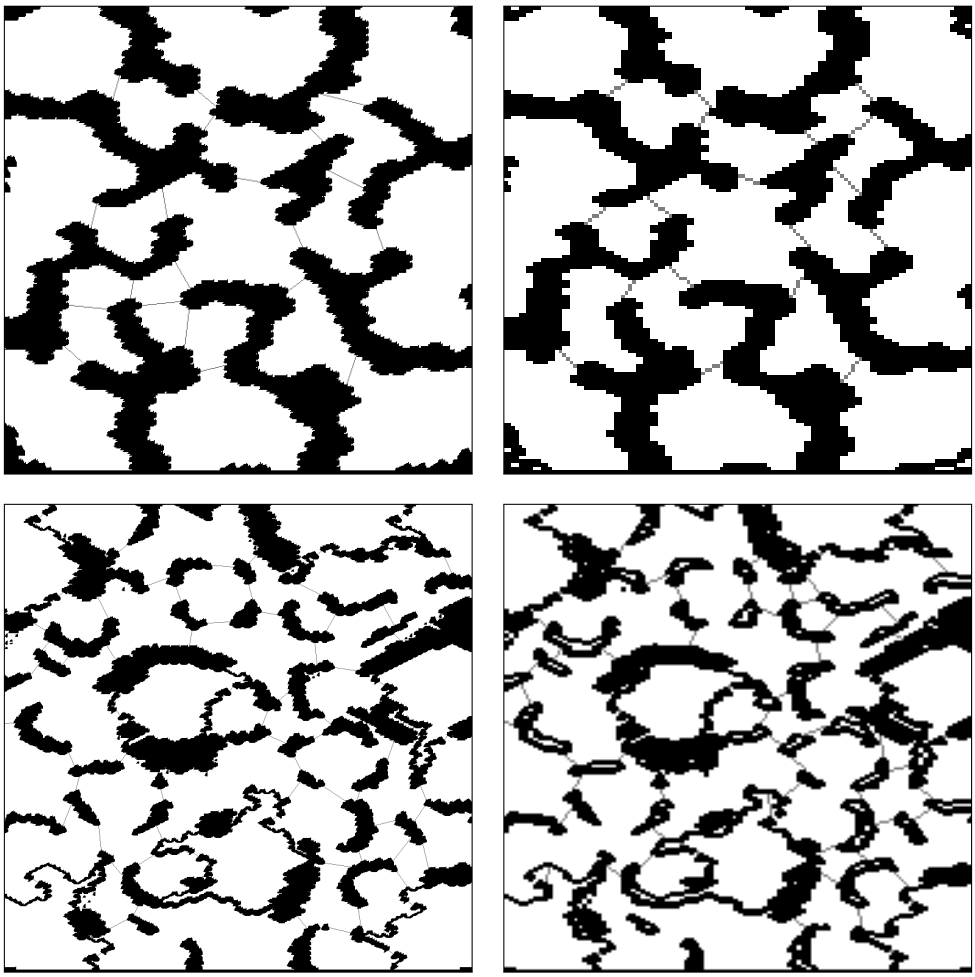

Figure 3.7: Decompositions on different resolutions. On the left are the versions from the benchmark suite [10] but on the right the low-res maps, which are of 16 (4x4) times lower resolution. In many cases the low-res runs are cleaner and less affected by noise than the high-res ones, in addition to being between one and two orders of magnitude faster (See table 3.1).

thus, on the low-res maps, the decomposition times are between one tenth and one half of a second. 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., in Warcraft when foresting connects new regions) or when regions become non-reachable (e.g., when a bridge collapses). Even when running on the high resolution maps the times are short enough to consider this. It may not be feasible to run the decomposition too frequently during gameplay, but in most games such partition-altering events only occur sporadically. Finally, 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.

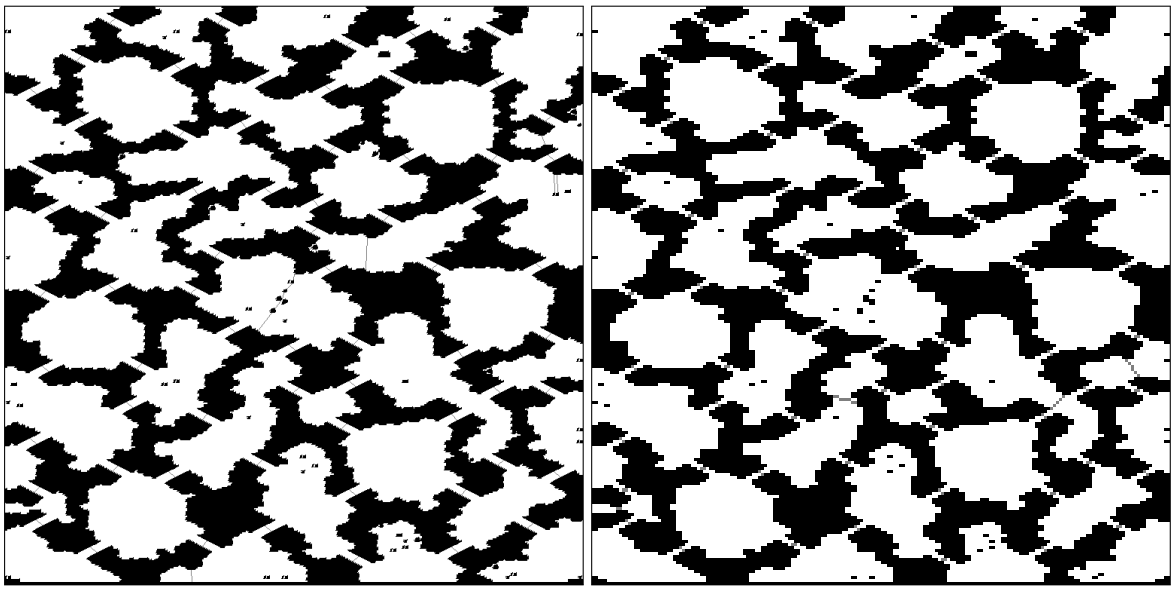

Figure 3.8: Decompositions on different resolutions cont'd.

3.2.2 Decomposition Output

We contrasted the partitioning output of our algorithm to that of the partition algorithm's introduced in [1] and [8], as well as to human-made partitioning. Figure 3.5 shows representative results from that comparison. Essentially, our partitioning looks more intuitive than that of [1] and yields very similar partitions to both [8] and the human-made ones.

It is not surprising that we do better than the decomposition method introduced in [1], 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 of more interest is that our method generates almost identical partitions to those of Perkin's 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 themselves, but the partitions were also almost identical to the ones produced by our automated decomposition method. The bottom rightmost map in Figure 3.5 shows one manual decomposition and Figure 3.6 pictures another example.

3.3 Summary

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 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.

In the next chapter we use our new decomposition method in conjunction with the pathfinding heuristics introduced in chapter 2 to further demonstrate its usefulness with a specific application in mind. We will also seek to point out specific maps and scenarios where the decomposition output is not optimal, but can be fixed with minimal intervention.

Chapter 4

Empirical Evaluation

In order to evaluate the aforementioned methods in context with each other we built a full pre-processing system that decomposes a tile based game map, calculates the shortest distances between gates and stores the resulting data in files formatted to be read and utilized by a search using the gateway heuristic.

4.1 Setup

The experiments were run on an A* search using a re-implementation of the gateway heuristic (GH). We did not use the 2-way gate refinement, as the map decompositions are now refined enough that it was expected to have minimal benefit. However, another small refinement was added that partially addresses the same problem as the 2-way gates, that is, we ensure that the heuristic at least never returns a lower estimate than a pure octile distance.

4.2 Measurement results

Most of the testing was performed on a set of 30 StarCraft maps, provided by the pathfinding benchmark test suite [10]. The maps were selected by taking the first 30 maps in the test suite, alphabetically. The main measurement was one batch where all 30 maps were automatically decomposed and pre-processed for GH, then every search from the scenario file was run on the respective maps, using A* search with octile and gateway heuristics, respectively. All numbers are averages over the entire run; time is measured in seconds and *speedup* is defined as the relative increase in problems solved per time unit, e.g. if an algorithm solves a set of problems twice as fast as a baseline algorithm, it constitutes 100% speedup.

The memory needed at runtime to store the necessary pre-calculated data is generally small, and dominated by the mapping of tiles to zones (2 bytes per tile allows for 65K gates). The data structure for storing gate information dwarves in comparison. For example,

for the largest map in our test suite, 1024x1024, the total memory needed is slightly over 2MB. If we limit the number of gates to 256 the memory requirements would be cut in half.

A closer look at the data showed that some maps were showing a much more significant speedup, while others were showing none, and in rare cases even showing slow-down when using GH. By looking at the decomposition it was apparent that there was scope for improvement, and by minor manual enhancements to the decomposition, many of the negative results could be avoided. Although we would ideally like to have a fully automated robust decomposition this shows that there is still scope for further improvement.

The results using the original and adjusted maps are shown in Tables 4.1, 4.2 and 4.3. We see that the speedup is 22% using all the original maps, 50% only looking at the best 21 maps and 40% using all maps, but now with some manually enhanced decompositions. These results, combined with the original results now showed a 40% speedup over the entire set of 30 maps, which must be considered a good result in light of the fact that only 5 out of the 9 maps showed improved results after the manual fix.

Table 4.1: Pathfinding searches in StarCraft maps using 30 first maps, alphabetically.

Heuristic	Avg len	Avg time (s)	Avg exp	Searches	Speedup
Octile	551.32	0.605	48033	77190	
Gateway		0.495	32002		22%

Table 4.2: Searches using the 21 best StarCraft maps out of the set of 30.

Heuristic	Avg len	Avg time (s)	Avg exp	Searches	Speedup
Octile	570.7	0.65	51784	55950	
Gateway		0.433	29555		50%

Table 4.3: Searches in using all 30 maps, some of them with manually fixed decompositions.

Heuristic	Avg len	Avg time (s)	Avg exp	Searches	Speedup
Octile	551.32	0.604	48033	77190	
Gateway		0.43	29327		40%

We also ran the decomposition and search on 38 randomly selected maps from Baldur's Gate II, upscaled versions taken from the test suite [10], resulting in a 34% speedup overall. The result is shown in Table 4.4. For more details, corresponding information for each individual map is provided in Appendix A. Additional data on re-expansions is there as well.

We profiled the program code during one run to see how expensive computation-wise GH is in contrast to the octile distance heuristic. We ran a full scenario (close to 3000

Table 4.4: Pathfinding searches in Baldur's Gate II maps.

Heuristic	Avg len	Avg time (s)	Avg exp	Searches	Speedup
Octile	252.79	0.09	8369	47821	
Gateway		0.067	5856		34%

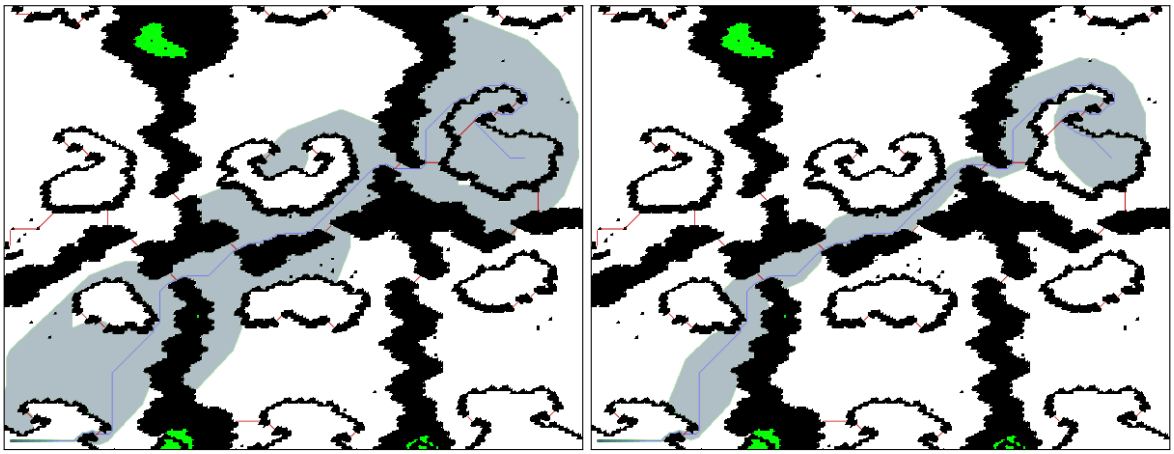

Figure 4.1: Side by side comparison of the open and closed lists (grey area) of an A* search using standard octile distance (left) and the gateway heuristic search (right). The resulting shortest path is shown in blue.

searches) on a typical map (AcrossTheCape) using both heuristics. Using the octile distance 2.5% of the total run time is spent computing the heuristic, while GH consumes 15% of the total run time.

4.3 Problematic Map Structures

While the vast majority of maps lends itself nicely to the decomposition algorithm, and the gateway heuristic will in most cases result in an improved search speed, there are certain cases that can result in a slower search.

One thing to keep in mind is that GH is inconsistent so the search must allow for the re-expansion of nodes. This stems from the fact that the length of a gate is not included in the estimate (as that could make it inadmissible) but only the estimated or true distances towards and away from it. This can present the search with fake shortcuts in the evaluations, which it will follow, only to have to re-evaluate those same nodes later in the search. When evaluating certain bad searches in the StarCraft map CrashSites we realized just how much of a problem this could be, as the worst searches would often be showing significantly smaller open/closed lists at the end of the search, but the amount of re-expansions would take the total number of expansions far above that of the regular octile heuristic search, resulting in a decrease in search speed. Knowing this though, it is possible to restructure the problematic

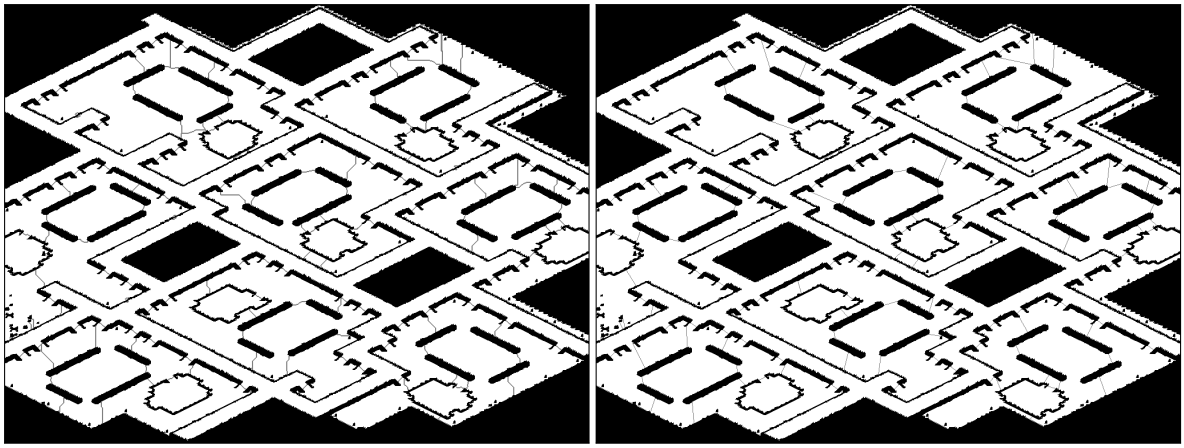

Figure 4.2: Refinement of gates from the raw water level fringes (top-left) to gates with two end points (top-right). This refinement is a problem in and of itself, and while in most cases our algorithm for it results in clean looking gates, we can see in this image that it can sometimes lead to unexpected results and can definitely be improved on, aesthetic-wise.

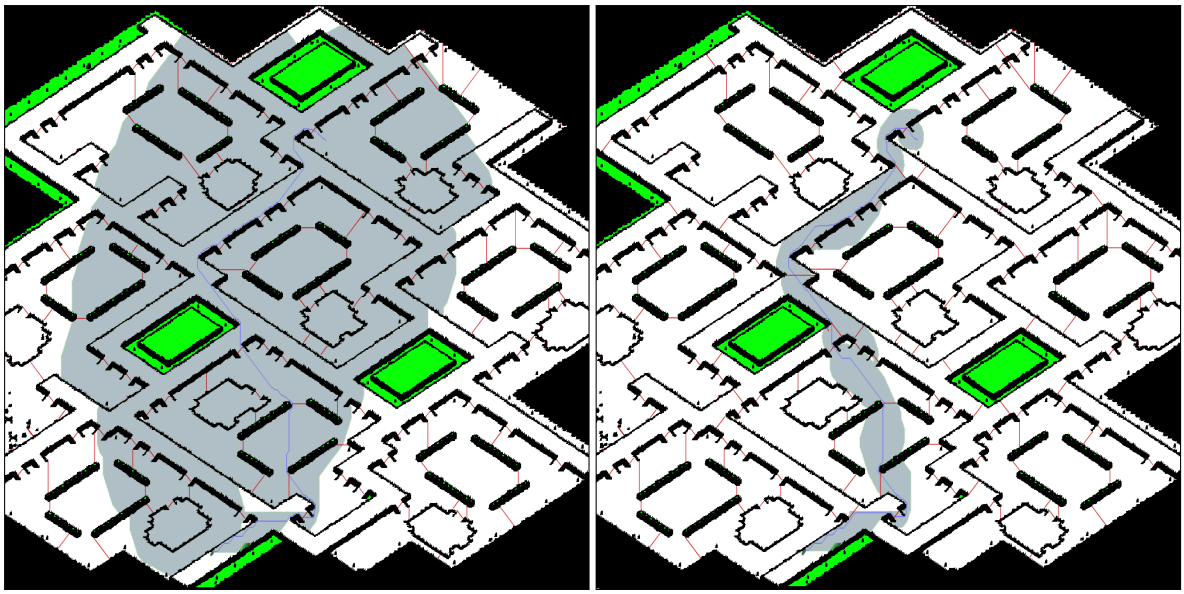

Figure 4.3: Comparison between open/closed lists after a search in the Aurora map. The search on the left uses an octile distance heuristic while the search on the right uses the gateway heuristic. The first image also shows the automatically generated and processed gates for the search, while the second one shows manually added gates as well.

gates and build a decomposition that gives a purely positive result. For example, for the Crash Sites map this manual enhancement had a very positive effect (see Table A.2).

4.3.1 The Aurora map

The Starcraft map *Aurora* (see Figure 4.2) is in many ways interesting for our methods of decomposition and pathfinding search. There is a long and winding hallway that is more or less of equal width its entire length. The decomposition ends up applying a single zone to the entire hallway. This zone has two problematic characteristics. Firstly, it is a complex and

non-convex zone, which is a bad fit for the octile heuristic (used within zones). Second, the one zone is connected to a huge amount of gates, which means that whenever a tile in that zone is associated with an evaluation, the $O(n^2)$ complexity makes the evaluation function slow. We used our analysis tool to manually add gates that split this single zone into more manageable parts and then re-calculated and re-ran our searches for this map. These added gates can be see in the rightmost image in Figure 4.3.

Table 4.5: Pathfinding searches in the Aurora map using different heuristics.

Heuristic	Avg len	Avg time	Avg exp	Searches
Octile	807.886	1.15897	88877.2	20
Gateway (automatic gates only)	807.886	1.92989	73402.5	20
Gateway (manually added gates)	807.886	0.688802	41168.2	20
Octile	444.021	0.360433	29981.2	200
Gateway (automatic gates only)	444.021	0.541439	21696.1	200
Gateway (manually added gates)	444.021	0.26021	17107.0	200
Octile	623.98	0.711368	56551.5	100
Gateway (automatic gates only)	623.98	1.16034	43569.8	100
Gateway (manually added gates)	623.98	0.431784	27382.3	100
Octile	813.996	1.25323	96780.5	50
Gateway (automatic gates only)	813.996	2.31715	99914.7	50
Gateway (manually added gates)	813.996	0.764092	46856.3	50
Octile	1005.86	1.91269	140632.0	10
Gateway (automatic gates only)	1005.86	3.35622	131430	10
Gateway (manually added gates)	1005.86	1.32063	72936.5	10
Octile	1195.73	2.89316	213212.0	20
Gateway (automatic gates only)	1195.73	5.88037	243459	20
Gateway (manually added gates)	1195.73	1.70679	101122.0	20

Table 4.5 shows detailed data on different searches in the Aurora map. When using GH with only the automatically decomposed gates, we see that there are fewer node expansions per search, but that the average time per expansion (not shown) is so high, because of the high number of gates in the main hallway zone and the $O(n^2)$ evaluation function, that the time spent searching is in fact significantly higher. Once some gates have been added manually in order to decompose the hallway itself a little bit better the time drops and in fact the adjusted decomposition results in fewer node expansions as well, in this case.

With the manually added gates the pathfinding search went from being slower using GH, to showing a 79% speedup compared to using the octile distance only.

4.4 Summary

The empirical evaluation shows that using GH on a map decomposed with the water level decomposition on average yields a sizable increase in search speed over a standard A* us-

ing an octile distance heuristic. It works well on the rather simple maps of Baldur's gate, and even better on the larger and more complex maps of StarCraft, even though a few of them were given enhanced decompositions. It is worth noting that this does not diminish their usability as general decompositions in strategic planning, even though they ended up with characteristics that badly affected their usefulness in this particular way of using pre-calculated data in a pathfinding heuristic. We also identified these characteristics and proposed ways to eliminate them.

Chapter 5

Related Work

Our work bears similarity to previous work on hierarchical pathfinding [12], [13], in particular the idea of using abstractions and map preprocessing. For example, the HPA^* algorithm [12] decomposes game maps into room-like structures, uses gates, and pre-calculates path distances. There is however a clear fundamental difference between our approach and the hierarchical ones: we use the abstract map view to improve heuristic state evaluation, but not to alter the representation of the search space. Our work bears in that respect more resemblance to work on heuristic evaluation improvement in other problem domains [14].

Some work has been done on heuristics that utilize search space abstractions and true distances stored in memory to better guide heuristic functions in pathfinding since the original publication of the Gateway Heuristic (GH) [1] in 2006. [15], [16] provide an overview of these ideas and implementations of search algorithms and heuristics using a similar approach. None of these implementations provide a method for pathfinding that gives a consistently better heuristic estimate than GH.

[17] describes the Portal Heuristic (PH), which works very similarly to GH. The difference is that instead of using each gate as a single line with two end points, every tile along the gate is stored as a portal and the distance between every pair of portals is stored in memory, while GH stores distances between pairs of gates. For any given map decomposition this would lead to the number of portals being the average length of gates times larger than the number of gates, and with the number of distances stored in memory being N*(N-1) (N being the number of gates or portals) the amount of memory needed for PH will be bigger than that needed by GH by a factor of the average length of gates squared. Given unlimited memory PH will give a more accurate estimate, as the length of gates does not introduce an error but given the huge memory requirements PH has been implemented using a memory-limited partitioning algorithm. This algorithm builds a single gate at a time, selecting its position based on the highest betweenness centrality [18], [19] of the currently largest partition, making a portal out of each tile along the gate, continuing until a maximum memory criterion is met. Given the same memory needed to store GH's pre-calculated distance data PH would have to have as few or fewer portals than GH has gates, resulting in significantly

fewer partitions and thus a much coarser abstraction of the search space, likely to yield a worse heuristic on average. Additionally the average and maximum number of portals per region, a number shown to have a significantly negative effect (see Section 4.3.1) on the effectiveness of the estimate function, is much higher in PH than in GH.

In [8] 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 traversable 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.

Chapter 6

Conclusions and Future Work

Here we have described one of the first methods of using distance data stored in memory to improve the heuristic estimate used in an optimal A* pathfinding search, a method that has since been referenced, replicated and improved on. The gateway heuristic still uses memory efficiently to significantly improve the speed of A* search, as compared to the octile distance A*, which is still the most common implementation of optimal pathfinding search. We have also addressed the problem of building consistent and intuitive map decompositions that support this heuristic. The decomposition has also been shown to work at least equally well as previous methods of decomposing for strategic AI in real time strategy games.

The methods used together are a fully automated system and have been shown to outperform a standard A* search in terms of search time, and even more significantly in the number of nodes expanded during the search which can in large searches reduce the memory needed, even though the heuristic itself requires some amount of memory to store pre-calculated distance data. The performance of the search method can be further improved in some cases by manually tweaking the decompositions to fit that purpose more exactly. This type of heuristic, coined true-distance heuristics, has since given rise to what are now the fastest optimal pathfinding searches.

The decomposition method itself has been shown to outperform the current standard method, used on a popular AI testing and competing platform, in terms of decomposition speed, and to give very similar results in terms of quality of decomposition and placement of gates. It is in fact our intention to test the water-level decomposition on that very platform with the purpose of replacing it to hopefully give a faster and more robust map analysis overall.

As for future work it has been pointed out that a possible enhancement to our decomposition as well as [8] would be to identify gates at both ends of hallways, even though they are of the same width throughout, marking the hallway not only as a path between zones, but as a zone itself. Implementing this enhancement would be beneficial if the algorithm is to be added to the BWTA library, as it would not only improve the speed of the analysis, as it already does, but the quality of the decomposition as well.

We also feel that there is room for optimization in order to further increase the speed of the decomposition algorithm and that the intermediate data collected during its execution, such as distance to nearest wall, could also be beneficial for strategic AI agents. Further work on utilizing this data in strategic AI in real time strategy games would be of interest to the field.

It would be of interest to better identify the types of maps and decompositions that the gateway heuristic does not work well with and build a system that applies different methods of decomposition when needed. For example a maze with corridors of even width throughout might not get an optimal decomposition using the water level algorithm, but our original ad hoc algorithm (see Appendix B) works especially well for that type of map.

Finally, various research can be done to explore possibilities of using a similar approach to decompose or analyze different state spaces, such as 3D game levels, real map data (rather than tile-based game maps) and partial analysis, such as continuous decomposition of maps that are gradually being explored.

We believe the methods here described, and the ideas behind them, are an important addition to modern research in game AI and that they can without doubt favorably impact implementations of pathfinding and tactical strategy AI.

Bibliography

[1] Y. Björnsson and K. Halldórsson, “Improved heuristics for optimal pathfinding on game maps”, in *AIIDE’06*, Marina del Rey, California, 2006, pp. 9–14.

[2] K. Halldórsson and Y. Björnsson, “Automated decomposition of game maps”, in *AIIDE’15*, Santa Cruz, California, 2015, pp. 122–127.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths”, *IEEE Transactions on Systems Science and Cybernetics*, pp. 100–107, 2 1968.

[4] D. C. Pottinger, “Terrain analysis in realtime strategy games”, in *Proceedings of Computer Game Developers Conference*, 2000.

[5] K. D. Forbus, J. V. Mahoney, and K. Dill, “How qualitative spatial reasoning can improve strategy game ai’s”, *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 25–30, 2002.

[6] P. Brobs, R. Saran, and M. van Lent, “Dynamic path planning and terrain analysis for games”, in *"AAAI Workshop: Challenges in Game Artificial Intelligence*, 2004, pp. 41–43.

[7] D. H. Hale, G. M. Youngblood, and P. N. Dixit, “Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds”, in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, October 22-24, 2008, Stanford, California, USA*, C. Darken and M. Mateas, Eds., The AAAI Press, 2008.

[8] L. Perkins, “Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition”, in *AAAI’10*, 2010.

[9] C. Si, Y. Pisan, and C. T. Tan, “Automated terrain analysis in real-time strategy games”, in *FDG’14*, 2014.

[10] N. Sturtevant, “Benchmarks for grid-based pathfinding”, *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144–148, 2012.

[11] A. Uriarte, *Brood wars terrain analysis 2*, `https://bitbucket.org/auriarte/bwta2`.

[12] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding", *Journal of Game Development*, pp. 7–28, 1 2004.

[13] S. Rabin, "A*: Speed optimizations", in *Game Programming Gems*, 2000, pp. 272–287.

[14] R. Holte, M. Perez, R. Zimmer, and A. MacDonald, "Hierarchical A*: Searching abstraction hierarchies efficiently", in *Proceedings AAAI-96*, 1996, pp. 530–535.

[15] N. R. Sturtevant, A. Felner, M. Barrer, J. Schaeffer, and N. Burch, "Memory-based heuristics for explicit state spaces", in *International Joint Conference on Artificial Intelligence*, 2009, pp. 609–614.

[16] A. Felner, N. Sturtevant, and J. Schaeffer, "Abstraction-based heuristics with true distance computations", in *Symposium on Abstraction, Reformulation and Approximation*, 2009.

[17] M. Goldenberg, A. Felner, N. Sturtevant, and J. Schaeffer, "Portal-based true-distance heuristics for path finding", in *Third Annual Symposium on Combinatorial Search*, 2010.

[18] M. Holzer, G. Prasinos, F. Schulz, D. Wagner, and C. Zaroliagis, "Engineering planar separator algorithms", English, in *Algorithms - ESA 2005*, ser. Lecture Notes in Computer Science, G. Brodal and S. Leonardi, Eds., vol. 3669, Springer Berlin Heidelberg, 2005, pp. 628–639.

[19] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality", in *2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, ch. 8, pp. 90–100.

Appendix A

Data from search execution

Tables A.1-A.4 show average search times for each of 30 StarCraft and 38 Baldur's Gate maps. The columns are map name, average path length, avg. search time in seconds, avg. nodes expanded, thereof how many re-expansions, and number of searches in the scenario.

Table A.1: Pathfinding searches in StarCraft maps using different heuristics.

Map (heuristic)	Avg len	Avg time (s)	Avg exp	Avg reexp	Searches
AcrosstheCape (octile)	592.002	0.699794	54869.5	0	2940
AcrosstheCape (gateway)	592.002	0.399497	26281.1	4285.77	2940
Aftershock (octile)	366.007	0.241077	19871.6	0	1810
Aftershock (gateway)	366.007	0.215062	14074.9	1406.74	1810
Archipelago (octile)	436.004	0.313657	25803.4	0	2160
Archipelago (gateway)	436.004	0.327068	23288.5	9621.68	2160
ArcticStation (octile)	823.982	1.36052	109318	0	4100
ArcticStation (gateway)	823.982	0.839048	58532.7	29480.9	4100
Aurora (octile)	601.977	0.911096	70297.2	0	2990
Aurora (gateway)	601.977	1.46451	59310.5	18799.8	2990
Aurora (octile - manual fix)	601.977	0.906402	70297.2	0	2990
Aurora (gateway - man. fix)	601.977	0.504679	31343.2	9183.33	2990
Backwoods (octile)	490.006	0.342409	28910.2	0	2430
Backwoods (gateway)	490.006	0.194723	13196	2257.62	2430
BigGameHunters (octile)	362.028	0.211085	17816.6	0	1790
BigGameHunters (gateway)	362.028	0.140646	11010.4	1833.26	1790
BlackLotus (octile)	756.024	0.940296	76370.6	0	3760
BlackLotus (gateway)	756.024	0.697397	46351.5	23181.2	3760
BlastFurnace (octile)	615.973	0.811672	64061.2	0	3060
BlastFurnace (gateway)	615.973	0.606275	39593.8	12323.4	3060
BrokenSteppes (octile)	559.994	0.632492	48071.1	0	2780
BrokenSteppes (gateway)	559.994	0.457259	30901.7	9781.41	2780
Brushfire (octile)	433.974	0.261745	22253.5	0	2150
Brushfire (gateway)	433.974	0.156646	11869.6	2685.6	2150
Caldera (octile)	342.003	0.183796	15232.4	0	1690
Caldera (gateway)	342.003	0.196134	14141.4	3483.71	1690

Table A.2: Pathfinding searches in StarCraft maps using different heuristics.

Map (heuristic)	Avg len	Avg time (s)	Avg exp	Avg reexp	Searches
CatwalkAlley (octile)	531.982	0.668454	55933.2	0	2640
CatwalkAlley (gateway)	531.982	0.408368	29066.9	6112.36	2640
Cauldron (octile)	803.988	1.40978	106068	0	4000
Cauldron (gateway)	803.988	1.02744	66193.6	19834	4000
CrashSites (octile)	566.027	0.502942	38740.1	0	2810
CrashSites (gateway)	566.027	0.961835	63996.8	39018.4	2810
CrashS. (octile - manual fix)	566.027	0.502431	38740.1	0	2810
CrashS. (gateway - man. fix)	566.027	0.366918	25281	3350.96	2810
CrescentMoon (octile)	371.949	0.245615	20151.9	0	1840
CrescentMoon (gateway)	371.949	0.23686	17359.8	4366.6	1840
Crossroads (octile)	558.019	0.552763	42992.7	0	2770
Crossroads (gateway)	558.019	0.351565	24824.1	2870.88	2770
DarkContinent (octile)	509.998	0.575449	45563.6	0	2530
DarkContinent (gateway)	509.998	0.415035	29244	11173.2	2530
Desolation (octile)	376.035	0.219499	18618.7	0	1860
Desolation (gateway)	376.035	0.170364	12633.6	1067.22	1860
EbonLakes (octile)	399.969	0.341353	27962.2	0	1980
EbonLakes (gateway)	399.969	0.231499	16850.2	5691.17	1980
Elderlands (octile)	489.992	0.345453	27669.8	0	2430
Elderlands (gateway)	489.992	0.254552	17517.3	2584.97	2430
Enigma (octile)	657.994	0.575297	48160.2	0	3270
Enigma (gateway)	657.994	0.402283	27563.4	4195.12	3270
Entanglement (octile)	337.968	0.183081	15114.6	0	1670
Entanglement (gateway)	337.968	0.205483	12634.4	2065.53	1670
Entang. (octile - manual fix)	337.968	0.182213	15114.6	0	1670
Entang. (gateway - man. fix)	337.968	0.144462	10070.3	2320.58	1670
Eruption (octile)	362.04	0.250424	20710.9	0	1790
Eruption (gateway)	362.04	0.189911	13250.9	2689.88	1790
Expedition (octile)	641.952	0.628432	47379.6	0	3190
Expedition (gateway)	641.952	0.673775	40329.1	11063.3	3190
Exp. (octile - manual fix)	641.952	0.626712	47379.6	0	3190
Exp. (gateway - man. fix)	641.952	0.615473	40322.1	12094.6	3190
FireWalker (octile)	507.999	0.387968	33734	0	2520
FireWalker (gateway)	507.999	0.23202	18072.9	615.409	2520
FloodedPlains (octile)	551.988	0.568366	43909	0	2740
FloodedPlains (gateway)	551.988	0.252242	16846.9	2041.14	2740
GhostTown (octile)	463.98	0.402711	33159.1	0	2300
GhostTown (gateway)	463.98	0.441651	29283.1	8071.04	2300
GhostT. (octile - manual fix)	463.98	0.401836	33159.1	0	2300
GhostT. (gateway - man. fix)	463.98	0.3914	24846.5	3904.62	2300
GladiatorPits (octile)	524.031	0.601765	46781.4	0	2600
GladiatorPits (gateway)	524.031	0.342251	24235.1	9119.68	2600
GreenerPastures (octile)	521.99	0.582307	46374.7	0	2590
GreenerPastures (gateway)	521.99	0.727395	52587.8	24208.2	2590

Table A.3: Pathfinding searches in Baldur's Gate maps using different heuristics.

Map (heuristic)	Avg len	Avg time (s)	Avg exp	Avg reexp	Searches
AR0011SR (octile)	256.401	0.123364	10769.3	0	1280
AR0011SR (gateway)	256.401	0.0826659	6410.2	917.714	1280
AR0012SR (octile)	256.237	0.131738	11041.3	0	1280
AR0012SR (gateway)	256.237	0.0830003	5603.82	372.948	1280
AR0016SR (octile)	244.874	0.0879769	7431.08	0	1221
AR0016SR (gateway)	244.874	0.0706569	5462.57	462.928	1221
AR0017SR (octile)	256.228	0.13387	10865.4	0	1280
AR0017SR (gateway)	256.228	0.0717539	5648.26	320.147	1280
AR0018SR (octile)	256.336	0.144329	12109.9	0	1280
AR0018SR (gateway)	256.336	0.0931953	6641.14	566.82	1280
AR0020SR (octile)	252.415	0.0579244	6572.05	0	1260
AR0020SR (gateway)	252.415	0.0595243	6161.12	16.3333	1260
AR0041SR (octile)	256.26	0.0666819	8151.08	0	1280
AR0041SR (gateway)	256.26	0.067017	8151.08	0	1280
AR0042SR (octile)	256.248	0.0736799	9115.91	0	1280
AR0042SR (gateway)	256.248	0.0739851	9115.91	0	1280
AR0043SR (octile)	256.293	0.0739815	9019.86	0	1280
AR0043SR (gateway)	256.293	0.0743828	9019.86	0	1280
AR0044SR (octile)	256.336	0.0687899	8539.81	0	1280
AR0044SR (gateway)	256.336	0.0701139	8538.38	0	1280
AR0045SR (octile)	256.25	0.0790795	8584.53	0	1280
AR0045SR (gateway)	256.25	0.0790443	8375.34	0	1280
AR0046SR (octile)	256.249	0.0701734	7785.29	0	1280
AR0046SR (gateway)	256.249	0.0622718	6999.26	0	1280
AR0070SR (octile)	256.394	0.102312	9045.7	0	1280
AR0070SR (gateway)	256.394	0.0538736	4439.68	298.205	1280
AR0071SR (octile)	256.343	0.0972062	8732.79	0	1280
AR0071SR (gateway)	256.343	0.0628387	5065.22	420.396	1280
AR0072SR (octile)	256.284	0.099796	8941.24	0	1280
AR0072SR (gateway)	256.284	0.0544059	4692.12	244.905	1280
AR0201SR (octile)	256.251	0.0439098	4410.46	0	1280
AR0201SR (gateway)	256.251	0.034195	3164.3	28.25	1280
AR0202SR (octile)	256.394	0.0744752	6884.3	0	1280
AR0202SR (gateway)	256.394	0.0540976	4364.9	699.648	1280
AR0203SR (octile)	256.206	0.086117	8356.96	0	1280
AR0203SR (gateway)	256.206	0.0830677	7868.46	0.005	1280
AR0205SR (octile)	256.256	0.0977825	8582.24	0	1280
AR0205SR (gateway)	256.256	0.0536145	4163.01	522.891	1280

Table A.4: Pathfinding searches in Baldur's Gate maps using different heuristics.

Map (heuristic)	Avg len	Avg time (s)	Avg exp	Avg reexp	Searches
AR0206SR (octile)	235.484	0.089751	7767.53	0	1175
AR0206SR (gateway)	235.484	0.0552508	4800.83	299.206	1175
AR0300SR (octile)	256.326	0.0944994	8591.98	0	1280
AR0300SR (gateway)	256.326	0.0592571	4559.64	788.565	1280
AR0302SR (octile)	247.245	0.0968242	8437.84	0	1233
AR0302SR (gateway)	247.245	0.078864	6263.26	80.0032	1233
AR0304SR (octile)	255.341	0.102257	8956.08	0	1275
AR0304SR (gateway)	255.341	0.0845309	6630.94	1110.6	1275
AR0306SR (octile)	256.239	0.129114	11153.3	0	1280
AR0306SR (gateway)	256.239	0.0740074	5899.42	215.141	1280
AR0308SR (octile)	256.272	0.142172	12231.3	0	1280
AR0308SR (gateway)	256.272	0.121287	8718.59	1368.3	1280
AR0400SR (octile)	256.507	0.0862324	7950.09	0	1280
AR0400SR (gateway)	256.507	0.0643287	5468.58	488.905	1280
AR0404SR (octile)	256.307	0.083377	7719.69	0	1280
AR0404SR (gateway)	256.307	0.0861535	6985.98	2193.13	1280
AR0405SR (octile)	256.367	0.139003	12144	0	1280
AR0405SR (gateway)	256.367	0.056619	4557.31	383.082	1280
AR0406SR (octile)	256.355	0.0800421	7411.41	0	1280
AR0406SR (gateway)	256.355	0.0453323	3617.57	345.277	1280
AR0412SR (octile)	256.23	0.0959357	8729.73	0	1280
AR0412SR (gateway)	256.23	0.0759473	6291.02	921.25	1280
AR0418SR (octile)	175.064	0.0197142	2643.4	0	874
AR0418SR (gateway)	175.064	0.0198274	2643.4	0	874
AR0500SR (octile)	256.352	0.0847354	7662.76	0	1280
AR0500SR (gateway)	256.352	0.0771017	5933.25	1248.11	1280
AR0526SR (octile)	234.007	0.0809273	7489.05	0	1168
AR0526SR (gateway)	234.007	0.0671157	5931.95	1.647	1168
AR0600SR (octile)	256.266	0.0376544	3630.59	0	1280
AR0600SR (gateway)	256.266	0.0372316	3441.07	0	1280
AR0602SR (octile)	256.484	0.0727776	6844.94	0	1280
AR0602SR (gateway)	256.484	0.0483633	3922.25	514.718	1280
AR0605SR (octile)	243.527	0.0660797	6918.65	0	1215
AR0605SR (gateway)	243.527	0.062921	6515.35	0	1215
AR0700SR (octile)	256.341	0.105874	9291.59	0	1280
AR0700SR (gateway)	256.341	0.0863231	5755.05	754.923	1280
AR0711SR (octile)	256.297	0.0583396	5422.74	0	1280
AR0711SR (gateway)	256.297	0.0405665	3650.61	28.1312	1280

Appendix B

Ad Hoc Decomposition Algorithm

In the original publishing of [1] we used an ad hoc decomposition algorithm in order to have a fully automated pre-calculation phase, although we had not yet implemented a sophisticated method. The initial results presented in Chapter 2 are from executions on maps decomposed with the algorithm described here.

The algorithm that divides the map into zones is a sort of flood-filling algorithm. Instead of having to input boundaries though, the algorithm automatically builds borders as it encounters tiles that satisfy certain conditions. The algorithm requires no input other than the tile-based map with information for each tile about whether it is passable or not. The output is information for each tile stating which zone it belongs to (or that it is non-traversable).

Pseudo-code for the decomposition method is shown as Algorithm 6. When creating a zone the algorithm starts by finding the top leftmost tile that is passable and has not yet been assigned to a zone. From that tile the algorithm starts flood-filling to the right until it hits a non-free tile. Both previously assigned and impassable tiles are regarded as non-free tiles (lines 9-15). It then proceeds to the next row, selecting a start point as far left as possible using similar stop criteria as for the right side (lines 27-36). It will then start filling to the right again, repeating the process.

The algorithm detects whether the right and the left borders grow or shrink from one line to the next (lines 17-26 and 37-42). If a border regrows after having shrunk the flood-filling for that zone is stopped (possibly having to undo the last line filled (lines 20-24)).

Figure B.1 shows examples of how the decomposition algorithm works. The top left image shows an undivided map, and in the image to its right the flood-filling has begun. The fourth row has stopped because the area opens upwards. It would be unwise to proceed in such cases as the line would cut right through another potential zone. This is the later stop condition in line 12 of the algorithm ($(x+1, y-1) \neq free$). In the bottom left image the algorithm has finished filling the zone. In the line immediately below the zone the algorithm has the chance to extend the zone to the right. However, as the zone has already shrunk from the right and regrowing is prohibited the zone filling stops. This ensures that zones have fairly regular shapes. In the last image two more zones have been similarly filled.

Algorithm 6 Automatic Map Decomposition

for all passable tiles in map **do**
  $zone(tile) \leftarrow free$
end for
$currZone \leftarrow 1$
repeat
  $(xLeft, y) \leftarrow$ top and leftmost free tile on the map
  $shrunkR \leftarrow shrunkL \leftarrow false$
  repeat
    {Mark line until hit wall or area opens upwards}
    $x \leftarrow xLeft$
    $zone(x, y) \leftarrow currZone$
    while $(x + 1, y) = free \wedge (x + 1, y - 1) \neq free$ **do**
      $x \leftarrow x + 1$
      $zone(x, y) \leftarrow currZone$
    end while
    {Stop filling area if right border regrowing}
    if $(x + 1, y - 1) = currZone$ **then**
      $shrunkR = true$
    else if $(x, y - 1) \neq currZone \wedge shrunkR$ **then**
      {Undo line markings}
      while $(x, y) = currZone$ **do**
        $zone(x, y) \leftarrow free$
        $x \leftarrow x - 1$
      end while
      $break$
    end if
    {Goto same initial x-pos in next line}
    $(x, y) \leftarrow (xLeft, y + 1)$
    {If on obstacle, go right in zone until empty}
    while $(x, y) \neq free \wedge zone(x, y - 1) = currZone$ **do**
      $x \leftarrow x + 1$
    end while
    {Move further left until wall or opens upward}
    while $(x - 1, y) = free \wedge (x - 1, y - 1) \neq free$ **do**
      $x \leftarrow x - 1$
    end while
    {Stop filling area if left border regrowing}
    if $(x - 1, y - 1) = currZone$ **then**
      $shrunkL = true$
    else if $(x, y - 1) \neq currZone \wedge shrunkL$ **then**
      $break$
    end if
  until $break$
  $currZone \leftarrow currZone + 1$
until no free tiles are found in map

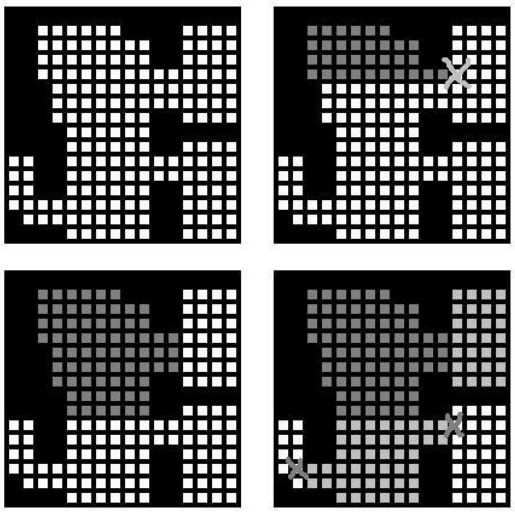

Figure B.1: Zone generation border criteria.

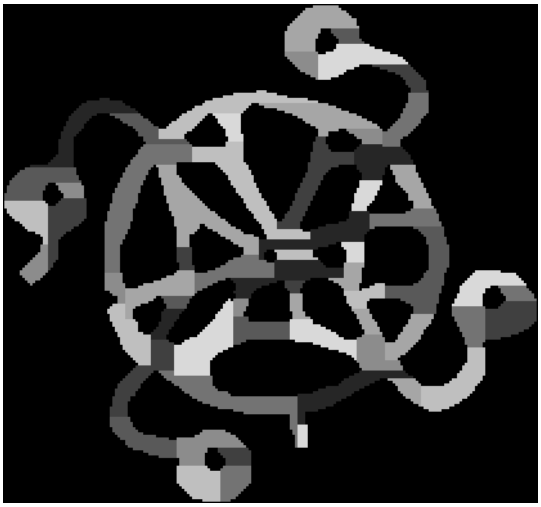

Figure B.2: Decomposed game map (212 x 214).

School of Computer Science
Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is