Improved Heuristics for Optimal Pathfinding on Game Maps

Yngvi Björnsson and **Kári Halldórsson**
Reykjavik University
Ofanleiti 2
IS-103 Reykjavik, Iceland
{yngvi,kaha}@ru.is

Abstract

As computer game worlds get more elaborate the more visible pathfinding performance bottlenecks become. The heuristic functions typically used for guiding A^*-based pathfinding are too simplistic to provide the search with the necessary guidance in such large and complex game worlds. This may result in A^*-search exploring the entire game map in order to find a path between two distant locations.

This article presents two effective heuristics for estimating distances between locations in large and complex game maps. The former, the *dead-end* heuristic, eliminates from the search map areas that are provably irrelevant for the current query, whereas the second heuristic uses so-called *gateways* to improve its estimates. Empirical evaluation on actual game maps shows that both heuristics reduce the exploration and time complexity of A^* search significantly over a standard octile distance metric.

Introduction

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. For example, in real-time-strategy (RTS) games there can be hundreds of units navigating the world simultaneously. Calculating paths for all these units in real-time is computationally demanding and may consume the better part of the available CPU resources reserved for the game logic. In RTS games pathfinding queries are also used to answer various strategic questions posed by the computer-controlled AI (e.g. how far is it to a particular resource or from where can the enemy attack). It is therefore of an utmost importance in modern games to use an efficient (and carefully implemented) algorithm for pathfinding calculations.

The *de facto* industry standard for pathfinding in games is the A^* algorithm (Hart, Nilsson, & Raphael 1968). 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. A simple and efficient heuristic is typically used for guiding the search. For example, in grid-based maps the *octile distance* (Manhattan distance extended to allow diagonal moves) is commonly used. However, as the game maps become larger and more complex such a simplistic heuristic cannot offer sufficiently targeted guidance, resulting in the search frequently exploring almost the entire map when finding a shortest path between two distant map locations.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

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.

The approach we present in this paper 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 but yet provide greatly improved search guidance. We introduce two such new heuristics, both of which are admissible and thus preserve optimality.

Our work bares similarity to recent work on hierarchical pathfinding (Botea, Müller, & Schaeffer 2004; Rabin 2000), in particular the idea of using abstractions and map preprocessing. For example, the HPA^* algorithm (Botea, Müller, & Schaeffer 2004) 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

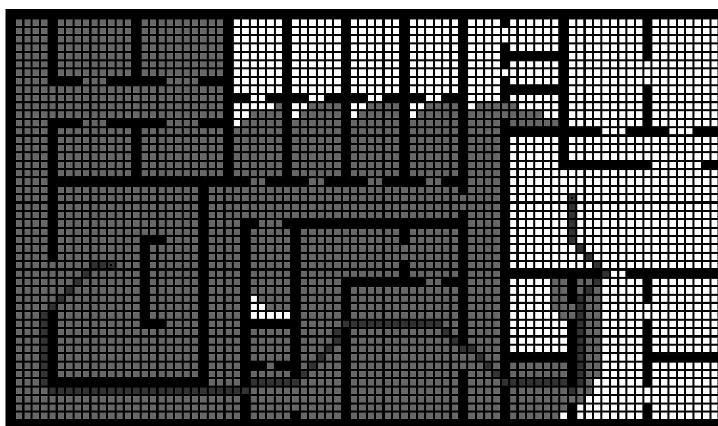

Figure 1: Example map: locations explored by A^* are shown in dark gray.

abstract map view to improve heuristic state evaluation, but not to alter the representation of the search space. Our work bares in that respect more resemblance to work on heuristic evaluation improvement in other problem domains (Holte *et al.* 1996).

The main contributions of this paper are: 1) improved admissible heuristics measures for guiding pathfinding search on (game) maps — our experiments on actual game maps show that A^* search using the new heuristics outperforms the standard octile-distance heuristic by a significant margin whether measured by nodes expanded or total search time; 2) an algorithmic method for automatic decomposition of game maps into smaller areas; the method is useful for creating abstract views not only for our new heuristics measures but also for hierarchical pathfinding techniques in general.

In the next section we describe the new heuristic functions and provide both detailed examples and pseudo-code. The subsequent section describes the automatic map decomposition, followed by a section summarizing the results of an extensive empirical evaluation of the heuristics using real game maps. We finally conclude and discuss future research directions.

Improved Heuristics

The map in Figure 1 depicts an indoor scene typical of a role-playing game. The world consists of multiple rooms that are connected via doors and corridors. State-of-the-art game maps would generally be somewhat larger and more complex, but for demonstration purposes we will use the map in the figure.

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, either because they result in dead-ends or clearly inferior paths.

The idea behind the two heuristics presented in this paper is 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. The former, the *dead-end heuristic*, avoids areas that lead to a dead-end, whereas the latter, the *gateway heuristic*, goes a step further by recognizing that moving through certain rooms can only lead to sub-optimal paths. The dead-end and gateway heuristics are described in the next two sub-sections, respectively. Computing the heuristics 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.

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.

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 decomposition algorithm on this map is shown to the left in Figure 2.

The second step in this phase is to construct a high-level graph for representing the different areas and the interconnections 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

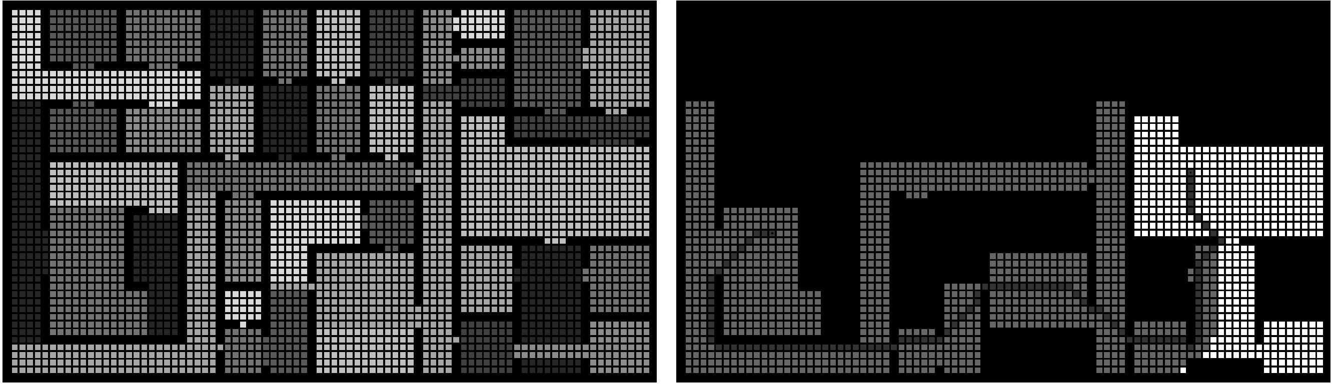

Figure 2: Dead-end heuristic: area decomposition (left) and relevant areas and nodes explored (right).

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.

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 though 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.

Gateway Heuristic

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

Preprocessing Phase The map is decomposed into areas in an identical way as for the dead-end heuristic. We define the boundaries between areas as *gateways* (or *gates*). A gateway can be of an arbitrary size, but an artifact of our decomposition algorithm is that its orientation is always either horizontal or vertical. Next we use multiple A* searches to pre-calculate the (static) 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).

An important element of our approach is that four different costs are stored for each pair of gateways. Each gate is 2-way because we are interested in knowing separate dis-

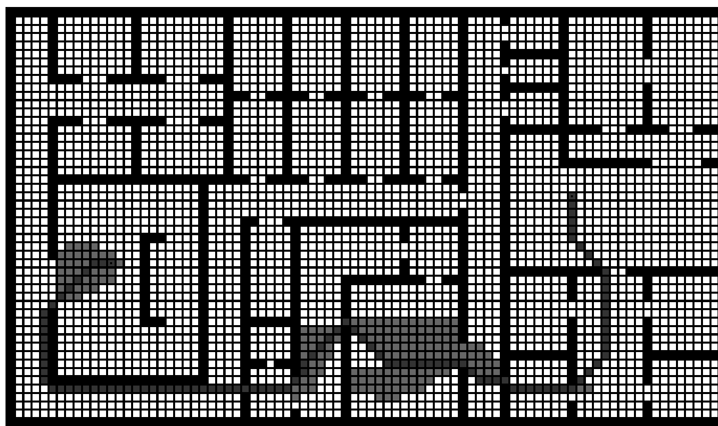

Figure 3: Nodes explored by the gateway heuristic. The error in the heuristic is due to the length of gates.

tances for each possibility of departing from and arriving to a gate. This results in significantly more accurate heuristic estimates during run-time compared to calculating only one cost value. We therefore need four separate pre-calculation pathfinding searches for each pair of gates (this is done offline so the extra time is of no importance). In these pathfinding searches we are *not* allowed to pass through the departing and arriving gate.

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

$$h^G(n,g) = min \sum_i \sum_j h^l(n,G_i)+H(G_i,G_j)+h^l(G_j,g)$$

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). 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 the gateway heuristic 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 the 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, our area decomposition algorithm 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 necessary 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.

Decomposition Algorithm

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 impassable).

Pseudo-code for the decomposition method is shown as Algorithm 1. 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 4 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

Algorithm 1 Automatic Map Decomposition

1: **for all** passable tiles in map **do**
2: $zone(tile) \leftarrow free$
3: **end for**
4: $currZone \leftarrow 1$
5: **repeat**
6: $(xLeft, y) \leftarrow$ top and leftmost free tile on the map
7: $shrunkR \leftarrow shrunkL \leftarrow false$
8: **repeat**
9: {Mark line until hit wall or area opens upwards}
10: $x \leftarrow xLeft$
11: $zone(x, y) \leftarrow currZone$
12: **while** $(x+1, y) = free \wedge (x+1, y-1) \neq free$ **do**
13: $x \leftarrow x + 1$
14: $zone(x, y) \leftarrow currZone$
15: **end while**
16: {Stop filling area if right border regrowing}
17: **if** $(x+1, y-1) = currZone$ **then**
18: $shrunkR = true$
19: **else if** $(x, y-1) \neq currZone \wedge shrunkR$ **then**
20: {Undo line markings}
21: **while** $(x, y) = currZone$ **do**
22: $zone(x, y) \leftarrow free$
23: $x \leftarrow x - 1$
24: **end while**
25: $break$
26: **end if**
27: {Goto same initial x-pos in next line}
28: $(x, y) \leftarrow (xLeft, y+1)$
29: {If on obstacle, go right in zone until empty}
30: **while** $(x, y) \neq free \wedge zone(x, y-1) = currZone$ **do**
31: $x \leftarrow x + 1$
32: **end while**
33: {Move further left until wall or opens upward}
34: **while** $(x-1, y) = free \wedge (x-1, y-1) \neq free$ **do**
35: $x \leftarrow x - 1$
36: **end while**
37: {Stop filling area if left border regrowing}
38: **if** $(x-1, y-1) = currZone$ **then**
39: $shrunkL = true$
40: **else if** $(x, y-1) \neq currZone \wedge shrunkL$ **then**
41: $break$
42: **end if**
43: **until** $break$
44: $currZone \leftarrow currZone + 1$
45: **until** no free tiles are found in map

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

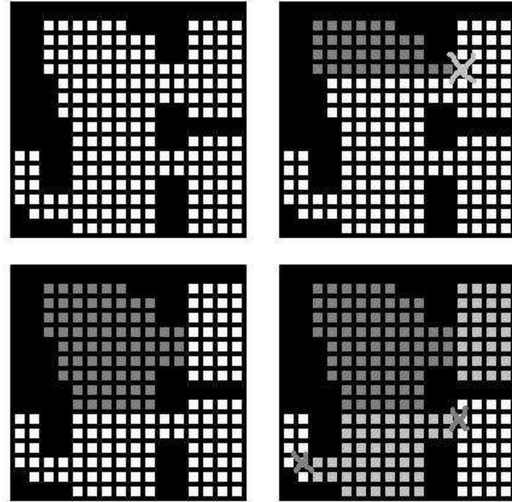

Figure 4: Zone generation border criteria.

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.

Empirical 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 1 shows the result of our pathfinding experiments where the octile and the two new heuristics are compared. On each map 1000 searches were performed using randomly chosen start and goal positions. The top section includes experimental data from searching our demo map (Figure 1) and the middle section data from nine different maps from the popular game *Baldurs Gate II* (Figure 5). In the last section we show separately data for a particularly large game map, also from *Baldur's Gate II* (Figure 6). Horizontal and vertical moves have the cost of 100 whereas diagonal moves were rounded to a cost of 150.

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.

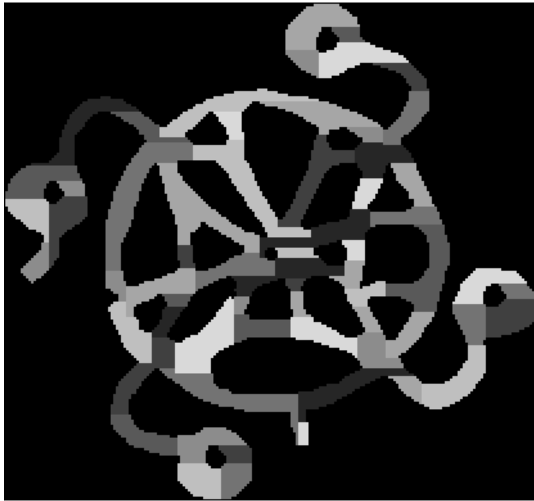

Figure 5: Decomposed game map (212 x 214).

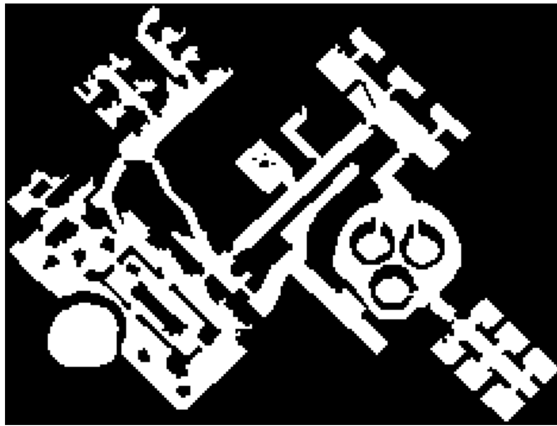

Figure 6: The largest game map (244 x 192).

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.

Conclusions

We presented two new admissible heuristic functions for guiding heuristic search in pathfinding large game maps. The initial results with these heuristics are promising. Both heuristics did outperform the standard techniques used in most modern games. However, before drawing any concrete conclusions we want to perform much more thorough experimental evaluation on many more maps. It is clear that with ever increasing game maps heuristics like the ones discussed here will become necessary.

Table 1: Pathfinding statistic (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

There is still room for improvement both in our implementation and, maybe more importantly, in improving the heuristic estimates. We have several ideas of how to continue with this research. One is to work more on the zone decomposition algorithm, to make it better adapt for various different types of terrain.

Acknowledgment

This research was supported by a grant from The Icelandic Research Fund (RANNIS). We also thank Jónas Tryggvi Jóhannsson who did an early implementation of the dead-end heuristic. BioWare Inc. kindly provided the Baldur's Gate II maps.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 7–28.

Hart, P.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 100–107.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings AAAI-96*, 530–535.

Rabin, S. 2000. A*: Speed optimizations. In *Game Programming Gems*, 272–287.