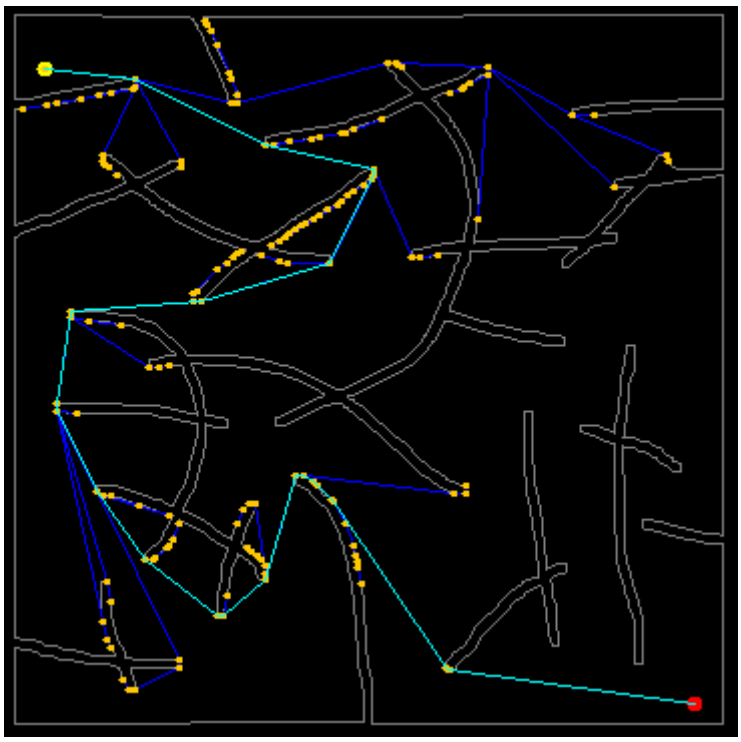
Pathfinding in Two-dimensional Worlds

A survey of modern pathfinding algorithms, and a description of a new algorithm for pathfinding in dynamic two-dimensional polygonal worlds.

Master's Thesis

Authors:
Anders Strand-Holm Vinther (20033980)
Magnus Strand-Holm Vinther (20094430)

Advisor:
Peyman Afshani



Aarhus University - Computer Science
June 2015

ABSTRACT

In this report we present and compare pathfinding algorithms used in 8-connected grid worlds and in two-dimensional polygonal worlds. These include Dijkstra's algorithm, A*, Jump Point Search, HPA* and two visibility graph based algorithms. On top of that we present our own BSP (Binary Space Partitioning) based optimal pathfinding algorithm for dynamic two-dimensional polygonal worlds.

We have seen how currently available optimal pathfinding algorithms for polygonal worlds rely on a heavy preprocessing step, in which a visibility graph is created. We introduce methods for reducing the size of the visibility graph, and for reducing the number of expanded nodes during pathfinding. With these methods and a BSP at hand, we show how the visibility graph can in fact be removed altogether while still achieving satisfying running times. With the preprocessing gone, our algorithm becomes applicable for dynamic polygonal worlds.

CONTENTS

Part I

PREFACE

1

INTRODUCTION

Pathfinding is used to solve the problem of finding a traversable path through an environment with obstacles. This is a problem seen in many different fields of study, which include robotics, video games, traffic control, and even decision making. All of these areas rely on fast and efficient pathfinding algorithms. This also means that the pathfinding problem appears in many different shapes and sizes. Applications in need of pathfinding will prioritize things differently and lay down different requirements on the algorithms. It is therefore worth exploring and comparing a wide variety of algorithms, to see which ones are better for any given situation.

In this thesis we focus our research on pathfinding algorithms for use in 2D environments. We look at two different types of world representations, namely the 8-connected grid world and the polygonal world. These world representations pose different challenges which require different strategies when it comes to pathfinding. We examine and compare Dijkstra's algorithm, A*, JPS (Jump Point Search), HPA* (Hierarchical Pathfinding), VG (Visibility Graph) and our own contributions: VGO (Visibility Graph Optimized) and BSP*.

We currently see a trend in video games, where the environments are becoming more and more interactable. Instead of having static environments, the player can now destroy or add to the environment. This is a problem when many pathfinding algorithms rely on preprocessing of a static environment. We found no currently used pathfinding algorithms that were specifically designed for dynamic polygonal environments. We therefore introduce BSP*; an optimal pathfinding algorithm for use in dynamic polygonal worlds, which can achieve satisfying running times without the need of a heavy preprocessing step. We do a comparison of all the algorithms across both world representations, to find out which algorithms are best for both static and dynamic environments.

In chapter two we specify the pathfinding problem. After that in chapter three the two world types are described. In chapter four we introduce and explain the algorithms. In chapter five and six we test and compare the algorithms, and finally in chapter seven and eight we discuss and conclude on our findings.

2

PROBLEM SPECIFICATION

The problem of pathfinding is an easy one to understand. Planning a path, from where you are, to where you want to go, is something you do on a daily basis. The hard part of pathfinding comes when we want computers to do it for us. Applications nowadays often put strict requirements on running time, memory usage and path length. Add to that a large, maybe even dynamic, environment, and you have a complex problem with many aspects to consider.

Many different fields of study will use pathfinding in some way or another. To consider all possible applications would be too broad of a study. Instead we have chosen to look at pathfinding through the eyes of a video game developer. In that case the pathfinding problem is almost always defined in a 2D environment. Even for 3D games the pathfinding problem can usually be reduced to a 2D problem, since movement is only allowed on the ground. For this reason we have chosen to consider only the 8-connected grid world and the polygonal world, since these are often seen in video games. We consider the agent in need of pathfinding to be point sized. This is by no means common in video games, but it simplifies the problem, and the solutions can often be extended to work with agents of larger size.

When comparing pathfinding algorithms one should at least consider three things: running time, memory usage and path length. A fast pathfinding algorithm ensures responsiveness, which is important for most games, and an optimal path length is needed if the game is competitive, such as is the case with a lot of real-time strategy games. Memory constraints are usually not a problem for the computers of today. Therefore we have chosen to focus our research on the running time and path length, and only briefly consider the memory usage of each algorithm. Path lengths are not explicitly measured, but all algorithms promise optimal or near-optimal path lengths.

The final thing to consider is the behavior of the environment itself. If the environment is static, one is able to perform a preprocessing step to improve the running times of all future pathfinding queries. If, on the other hand, the environment is constantly changing we do not have the luxury of a preprocessing step. In this case we are left with only the algorithms that can perform pathfinding directly

on the environment without preprocessing it. We consider both the static and the dynamic case, when comparing the algorithms.

In summary we wish to find out which algorithms are best used for static and dynamic environments, when priority is put on running time and path length.

Part II

THEORY

3

WORLD REPRESENTATIONS

Pathfinding is used in many different situations. This means that the worlds, in which pathfinding is required, have a wide variety of rules and features. It is important to consider these features to be able to create optimal pathfinding solutions. However some worlds require several steps of simplification, before pathfinding is even feasible.

In this report we examine pathfinding on two rather simple world representations, namely the 8-connected grid world and the polygonal world. These are commonly used world representations, which are often subject to pathfinding. Furthermore many other complex worlds can easily be simplified to either a grid world or a polygonal world.

3.1 THE 8-CONNECTED GRID WORLD

The first world representation we look at is the 8-connected grid world. An 8-connected grid world is a collection of cells, ordered in a grid like manner, where movement is only possible in up to eight directions (see figure 1).

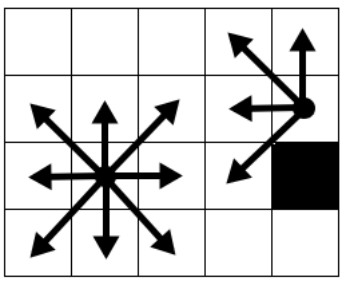


Figure 1: A small 8-connected grid map with one obstacle cell.

A grid world has two types of cells, traversable cells and obstacle cells. Movement is only possible between traversable cells. The cost of moving from one cell to another is the distance between the centers of the cells. This is typically scaled such that horizontal and vertical movement has cost 1, while diagonal movement has cost $\sqrt{2}$.

The good thing about this world representation is that it is easily made compatible with graph searching algorithms. We can think of

each cell as a node, with edges to all of its non-obstacle neighbors. The edge weights are then the cost of movement. For this reason we can apply graph searching algorithms, such as Dijkstra's algorithm and A*, directly on the grid world.

The bad thing about grid worlds is that the grid structure restricts movement to only eight directions. Some applications might require the movement to be restricted, and in these cases the grid world is perfect. But if the grid world is used for pathfinding, where the agents should be allowed to move freely you might end up with suboptimal paths. This is illustrated on figure 2. The red path is found using the grid world abstraction, and a more optimal path is marked in blue.

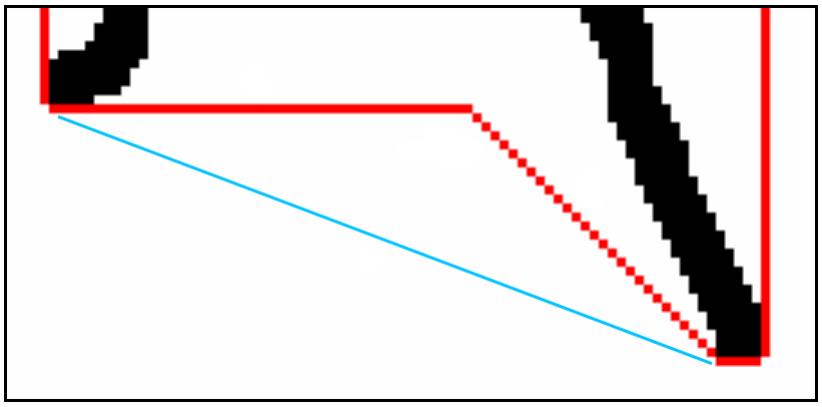


Figure 2: A path found using the grid world abstraction (red), and a more optimal route (blue).

Grid worlds are also bad at representing environments with varying passage width. The problem comes when you have to decide on a grid cell resolution. If you want to represent a detailed environment with narrow passages the grid cells has to be small. But the smaller you make the grid cells, the more of them you need. If you increase of the number of grid cells, you increase the number possible paths a pathfinding algorithm has to consider. So in order for pathfinding algorithms to work as fast as possible, you need to increase the size of the grid cells as much as possible without losing important detail.

3.2 THE POLYGONAL WORLD

The other world representation we look at is the polygonal world. Once again the world consists of traversable areas and obstacle areas. These areas are in the polygonal world defined using polygons. An example of a polygonal world is illustrated in figure 3.

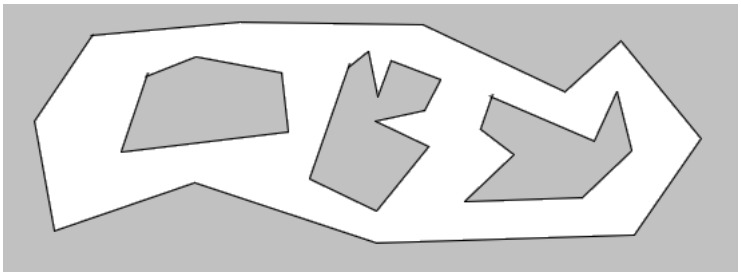


Figure 3: A polygonal world, with obstacles marked in grey, and a traversable area marked in white.

When we talk about polygonal worlds we will refer to two types of corners: protruding corners and depressed corners. Protruding corners bulge out from obstacles into the traversable area, whereas depressed corners cave into obstacles (See figure 4).

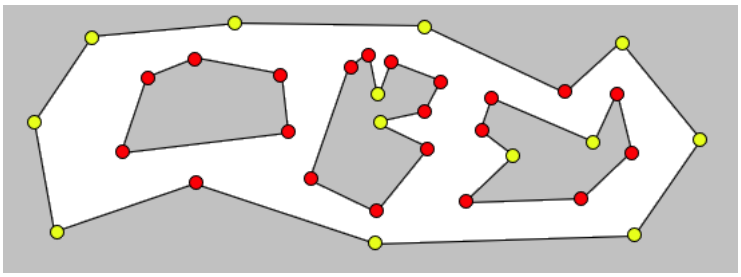


Figure 4: A polygonal world with protruding corners (red) and depressed corners (yellow).

These two types of corners are important to have in mind when talking about the shortest path in polygonal worlds. If no obstacles are present, the shortest path between two points will obviously be a straight line. If obstacles are indeed present, the shortest path will have to wrap around these obstacles. For the path to be as short as possible, one would have to move in straight lines from one corner to another, until the goal can be reached directly, as is illustrated in figure 5.

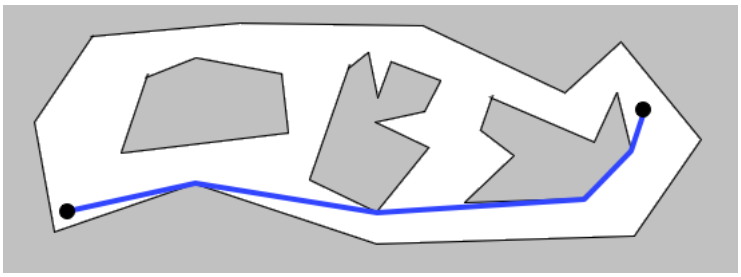


Figure 5: The shortest path from one point to another in a polygonal world.

In fact any shortest path in a polygonal world, can be described as a list of moves from one corner to another. This can be specified even further, by noticing that no shortest path will ever visit a depressed corner, and therefore any shortest path can be described as a list of protruding corners only. This means, when the algorithms search for an optimal path, they can ignore all depressed corners.

A good thing about polygonal worlds are that they allow for any-angle movement. This means that the pathfinding algorithms working on polygonal worlds are able to find the truly shortest path between two points, as opposed to those working on grid worlds. Polygonal worlds also work very well with varying level of detail. As opposed to grid worlds, parts of a polygonal environment can be detailed, without forcing other parts to be detailed too.

One final thing to notice is that graph searching algorithms such as Dijkstra's algorithm and A* cannot be applied directly to polygonal environments. A conversion of the world into a graph is needed. In the grid world each point had a maximum of eight well defined neighbors, which meant it could be interpreted directly as a graph. Earlier we saw that the shortest path in the polygonal world can be described by a list of protruding corners. If we define the neighbors of a point to be all visible protruding corners, we can interpret the polygonal world as a graph. Calculating visibility however is not easy, since every protruding corner in the world is potentially visible. This means we cannot hope to look up the neighbors in constant time; at least not without preprocessing.

4

PATHFINDING ALGORITHMS

In this section we provide an in-depth explanation of how each pathfinding algorithm works. We first look at two graph searching algorithms; Dijkstra's algorithm and A*. Following that, we look at two pathfinding algorithms specifically designed to work on 8-connected grid worlds; JPS and HPA*. Finally we switch over to the polygonal world representation where we look at three pathfinding algorithms; VG, VGO and BSP*.

4.1 PRIORITY QUEUES

Before we look at the actual pathfinding algorithms, we provide a quick explanation of the priority queue data structure. We felt like we should do this first, as it is an integral part of all the following pathfinding algorithms.

A priority queue is a data structure where each element has an associated priority. The priority is usually defined using a numerical value, where a lower value equals a higher priority. The elements in the queue is not necessarily sorted according to this value, but the data structure should provide a method for extracting the element with highest priority. We refer to this operation as the *extract-min* operation, as the element with highest priority is the one with minimum value. One should also be able to perform a *decrease-key* operation on one of the elements, which serves the purpose of decreasing its value, also known as its key, and thereby changing its priority in the queue. Finally one should of course also be able to add elements to the priority queue, which is done using the *insert* operation.

We considered three different implementations of the priority queue; an unsorted list, a binary heap and a Fibonacci heap. The time complexities of the operations mentioned above using these three implementations are shown in the table below.

Operation	Unsorted List	Binary Heap	Fibonacci Heap
extract-min	$O(n)$	$O(\log n)$	$O(\log n)$*
decrease-key	$O(1)$	$O(\log n)$	$O(1)$*
insert	$O(1)$	$O(\log n)$	$O(1)$

Table 1: Time complexities of operations performed on a priority queue of size n. Times marked with a * are amortized.

Among the three priority queues, the Fibonacci Heap seems to be the fastest, however to see if this was also the case in practice we ran some initial tests. Our tests confirmed that the Fibonacci Heap was the overall fastest, therefore we decided to use this queue only.

4.2 GRAPH ALGORITHMS

The two most well known pathfinding algorithms are Dijkstra's algorithm and A*. These were originally designed to work on graphs only, but has since proved very useful for pathfinding in almost any world representation, as many representations can be simplified to graphs.

4.2.1 *Dijkstra's Algorithm*

Dijkstra's algorithm was originally conceived by the Dutch computer scientist Edsger W. Dijkstra in 1956 [2]. Dijkstra's algorithm is an optimal pathfinding algorithm, originally designed to work on graphs. Dijkstra's algorithm is the most important optimal pathfinding algorithm, as it formulates the core strategy that all other improved algorithms use.

Dijkstra's original implementation of the algorithm ran in worst case $O(N^2)$ time, where N is the number of nodes in the graph. A faster implementation was later discovered in 1984 by Fredman and Tarjan [3], which achieve a worst case running time of $O(E + N \log N)$, where E is the number of edges in the graph.

Dijkstra's algorithm takes as input a graph, a start node and a goal node. It works in a similar fashion as breadth first search. It works its way outward in all directions from the start node, visiting the nodes that are closest to the start node first. As soon as it reaches the goal node, it can trace back the shortest path from the goal node to the start node. As it is searching it uses two sets of nodes, which we shall refer to as the explored set and the frontier set. The frontier set is implemented as a priority queue. We will see why in a moment.

An important concept of Dijkstra's algorithm is the G value. Each node in the graph has an associated G value. For nodes in the explored set the G value is equal to the length of the shortest path back to the start node. For nodes in the frontier set it is only a temporary estimate. For nodes not in these sets the value is undefined. In the

following illustrations we shall see how the G value is used to search the graph for the goal node.

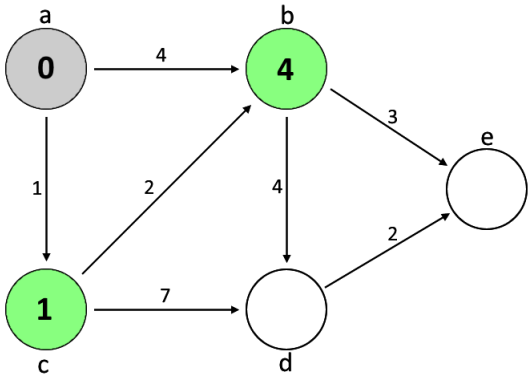


Figure 6: The nodes as they look after the first iteration of Dijkstra's algorithm.

In figure 6 we see a graph consisting of five nodes and seven edges. Each edge has an associated weight, which determines the cost of moving from one node to another along that edge. In this example we are tasked with finding an optimal path leading from node a to node e. For this and in the following illustrations we shall color the explored nodes (those that are in the explored set) grey, and the frontier nodes (those that are on the frontier set) green. The G value of each node is written inside the nodes.

After the first iteration of the algorithm only the start node is in the explored set, and only the start node's neighbors are in the frontier set. The G value of the start node is zero for obvious reasons. The G values of the neighbors are calculated by adding the G value of the start node to the weight of the edge connecting them, which is in our case 1 and 4. Notice how the G value estimation of node b is wrong (the shortest path back to the start node has length 3, not 4). In fact of all the nodes in the frontier set, only the lowest estimated G value is known to be correct. The reason for this is that this node cannot be reached faster through any other node, as it is connected to the explored set with the least weighted edge. We can therefore add node c to the explored set, and move on to the next iteration (See figure 7).

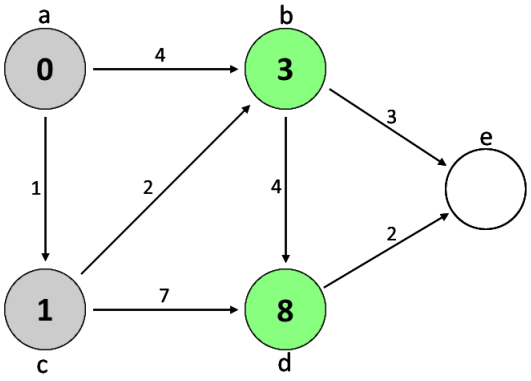


Figure 7: The second iteration of Dijkstra's algorithm.

After the second iteration we can see, how the explored set now consists of two nodes, as we picked the node from the frontier set with the lowest G value and added it to the explored set. This will not be the last time we extract the node with the minimal G value from the frontier set, which is why we refer to this as the extract-min operation.

After moving a node from the frontier set to the explored, we also need to add the unexplored neighbors of this node to the frontier set. This is done with the insert operation. In our case the neighbors of node c is b and d. If a node is already in the frontier set, as is the case of node b, we need to check if a lower G value can be estimated from a path going back through c. In our case we are able to lower b's G value from 4 to 3. This, so called relaxation of the G value, is referred to as a decrease-key operation.

We are now ready to perform another extract-min operation on the frontier to see which node can be added to the explored set.

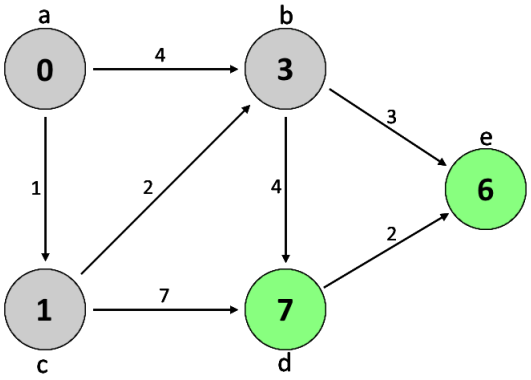


Figure 8: The third iteration of Dijkstra's algorithm.

After the third iteration (Figure 8) we have three nodes in the explored set and two nodes in the frontier set. The most recently explored node is node b. When we added b to the explored we also

added e to the frontier set and relaxed the G value of d. Once again we are ready to perform another extract-min operation.

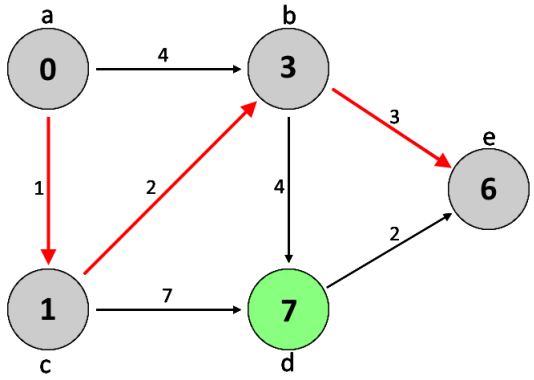


Figure 9: The fourth and final iteration of Dijkstra's algorithm.

In the fourth and final iteration (Figure 9) the goal node is added to the explored set. As soon as the goal node is explored, we need not explore anymore nodes. At this point we can trace back the path from the goal node to the start node, which is guaranteed to be the optimal path.

With this example in mind we can summarize Dijkstra's algorithm like so: We maintain a set of explored nodes, for which we have determined the G values, and a set of frontier nodes, for which we have estimated the G values. Initially only the start node is in the explored set, but in each iteration we perform an extract-min operation on the frontier set, to move one node over to the explored set. As we do this, we make sure to add all unexplored neighbors of the extracted node to the frontier set. If any of its neighbors are already in the frontier set we reestimate their G value. If a path between the start and the goal node exists we will at some point add the goal node to the explored set, and at this point we can trace the optimal path back to the start node. Our implementation follows this exact scheme, as can be seen from the pseudocode found in the appendix.

Running Time

The running time of Dijkstra's algorithm is highly dependent on the implementation of the frontier set. In the worst case we perform an extract-min operation for each node in the graph, and a decrease-key operation for each edge in the graph. This gives us a worst case running time of $O(E \cdot T_{dk} + N \cdot T_{em})$, where E is the number of edges in the graph, N is the number of nodes in the graph, T_{dk} is the time complexity of the decrease-key operation, and T_{em} is the time complexity of the extract-min operation.

If we implement the frontier set as an unsorted list, we can perform the decrease-key operation in constant time, and the extract-min oper-

ation in linear time. This gives us a running time of $O(E + N^2)$. If we implement the frontier set as a binary heap we get a worst case running time of $O(E \log N + N \log N)$. And finally if we use a Fibonacci heap we can improve the running time to $O(E + N \log N)$.

For graphs in general the number of edges is bounded to $E \leq O(N^2)$ which gives us a worst case running time of $O(N^2 + N \log N) = O(N^2)$. For 8-connected grid maps the number of edges is bounded to $E \leq O(N)$ which makes Dijkstra's algorithm run in $O(N \log N)$. We can specify the running time even further if we think of where each term came from. The extract-min operation is performed on the frontier set each time we want to explore a new node, which gives us a running time of $O(X \log F)$, where X is the number of explored nodes and F is the average size of the frontier set. This shows us that Dijkstras algorithm can be improved if we reduce the number of explored nodes.

Pros and cons

There are both good and bad things to be said about Dijkstras algorithm. The good thing is that Dijkstras algorithm formulates the core strategy for searching a graph for the shortest path between two nodes. Every other algorithm we look at is built upon Dijkstras algorithm.

The bad thing is that it makes no assumptions about what the graph looks like, other than it has positive edge weights. The graphs generated from either polygonal worlds or grid worlds does have features, which can and should be utilized to speed up the process. These features include coordinates associated with each node, and in the case of the grid world, a predictable ordering of the nodes.

4.2.2 A^*

The next algorithm we look at is called A* (pronounced: A star). A* was first described in 1968 by a group of researchers at Stanford Research Institute [4]. This algorithm is an extension of Dijkstra's algorithm, which aims to improve upon the running time using heuristics.

Dijkstra's algorithm was in some sense blindly searching the graph for the goal node. It did not use the location of the goal node to guide its search, which meant it would search with equal priority in all directions. A* uses the fact that all nodes generated from spatial domains have associated coordinates. These coordinates can be used to measure how far a node is from the goal node using a distance function. If we choose the right kind of distance function we hope to make A* work faster than Dijkstra's algorithm without sacrificing optimality. We shall return to the distance function later, but first we look at the changes A* has made to Dijkstra's algorithm.

A* does not change the overall structure of Dijkstra's algorithm. It still creates an explored set and a frontier set and uses these to search the graph. In each iteration it moves a node from the frontier set to the explored set, but where Dijkstra's algorithm moved the node with the lowest G value, A* moves the node with the lowest F value. We have not seen this F value before. The F value is the G value with an added H value. The H value of a node is calculated using a heuristic function, which estimates the path length to the goal node. The F value is only relevant for nodes in the frontier set, as it is only used to select which node to move from that set to the explored set. So to summarize we give each node in the frontier set a G, H and F value:

- G: An estimate of the shortest path length to the start node.
- H: An estimate of the shortest path length to the goal node.
- F: The sum of G and H.

We can use this F value as a ranking of the nodes in the frontier set. The node with the lowest F value is the node, which is most likely to lead us to the goal node. This change could seem like it breaks the algorithm, making it suboptimal, but if we do it cleverly, we can create a heuristic function, which ensures the algorithm to still be optimal.

For a heuristic to ensure optimality it needs to be consistent and admissible [5]. For a heuristic to be consistent, it needs to ensure that, if there is an edge between two nodes the difference in H value between them is never larger than the edge weight. For the heuristic to be admissible it must never overestimate the path length to the goal node. In the following section we explain why an admissible and consistent heuristic leads to a guaranteed optimal path.

Heuristic Requirements

The heuristic function can either underestimate, overestimate or somehow know the path length to the goal node. Let us first consider the case of a gross underestimation, namely the case where the H value is always zero. If the H value is zero the F value will become equal to the G value. If this is the case, A* will become identical to Dijkstra's algorithm, which means we have not really made it faster, but at least we know it is still optimal.

Next we consider the case where the heuristic function always guesses correctly, which means the H value is always the length of the shortest path to the goal node. Let us denote the length of shortest path between the start node and the goal node, P. In this case the start node would have a G value of 0 and a H value of P, making the F value also equal P. As we move along the shortest path towards the goal node, the F value will remain constant, since the G value increases by the same amount as the H value decreases. The F values of nodes which are not part of the shortest path would be higher than P.

Since A* always chooses the node from the frontier set with the lowest F value, it will explore only the nodes on the shortest path. So if A* had access to some sort of oracle heuristic function, the algorithm would be very fast and would still always find the optimal path.

Finally we consider the case where the heuristic function might overestimate the H value. If this was the case one can imagine a situation, where the H value of a node along the optimal path was so large that the algorithm would completely disregard this node, and instead find a suboptimal path with smaller H values. So if the heuristic function sometimes overestimates the length of the shortest path to the goal node, the algorithm will no longer be optimal. So as long as the heuristic does not overestimate the path length to the goal node and is consistent, we can be sure that the path found by A* is optimal.

A Heuristics*

Let us look at a couple of heuristics and see how they work in a grid map. On figure 10 we have illustrated the distance from the bottom left corner to the top right, measured using three different distance functions. The Manhattan distance is shown in blue, the Euclidean distance is shown in red, and finally, what we shall refer to as the move distance, is shown in green.

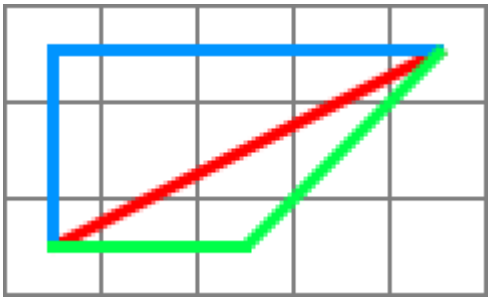


Figure 10: Three distance functions illustrated: The Manhattan distance (blue), the Euclidean distance (red) and the move distance (green).

The Manhattan distance is the total difference between the coordinates of the two nodes. The Euclidean distance is the straight-line distance between the nodes. The move distance is the length of the path between the two nodes, if you follow the rules for movement in an 8-connected grid map, and no obstacles are present. The move distance is admissible as the actual path can only be longer, such as if obstacles are present. The Euclidean distance is also admissible as it measures an even shorter distance, but the Manhattan distance is not admissible as it overestimates the path length. Now let us look at how A* performs in a grid map using these heuristics (See figure 11).

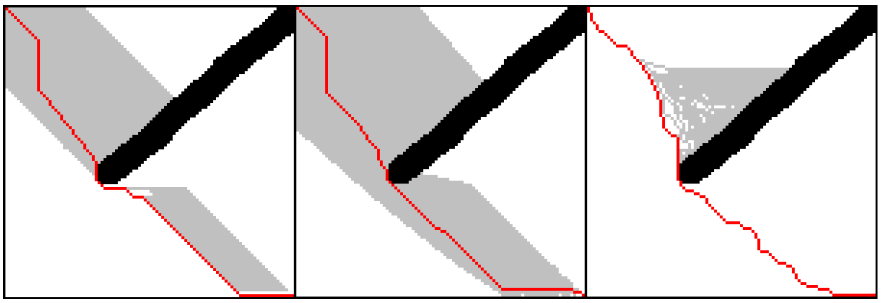


Figure 11: A*s behavior using different heuristics. Move distance (left); Euclidean distance (Middle); Manhattan distance (Right). The found path is marked in red, while explored nodes are marked in grey.

Here we see a pathfinding problem solved using three different heuristics in combination with A*. The objective here is to find a path from the top left corner to the bottom right corner. The path is marked in red, and the explored nodes are grey.

Both the Euclidean distance and the move distance provides us with optimal paths. The Manhattan distance on the other hand does not. But notice the amount of explored nodes when using the Manhattan distance. This shows that an overestimating heuristic might lead to a shorter search time, at the expense of optimality. Also note the difference between the Euclidean distance and the move distance. Fewer nodes are explored when using the move distance, as it comes closer to correctly estimating the correct path length to the goal node. Based on these considerations we have chosen to stick with the move distance when A* is used in the grid world.

We have not yet considered the heuristics used in the polygonal world, but the same rules apply. We want to make sure we never overestimate the distance to the goal node, however we also do not want to underestimate more than necessary either. In a flat polygonal world with no obstacles, the length of the shortest path to the goal node is the Euclidean distance. If obstacles are added, the path might become longer. This makes the Euclidean distance the best heuristic for use in the polygonal world.

Pros and cons

A* is a considerable improvement over Dijkstra's algorithm. It uses a heuristic to guide it towards the goal node, instead of searching blindly in all directions. This leads to fewer explored nodes, and therefore a faster running time. We have not payed much attention to the memory usage, but what we can say is with fewer explored nodes the maximum amount of memory required is also reduced. See figure 12 for an example of the improvement over Dijkstra's algorithm.

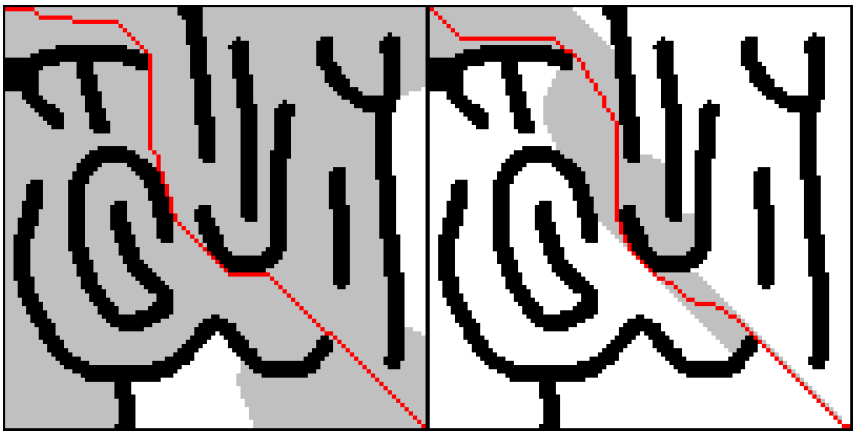


Figure 12: The same pathfinding problem in an 8-connected grid world using Dijkstras algorithm (left) and A* (right). The path is marked in red, and the explored nodes are marked in grey. The start node is in the top left corner and the goal in the bottom right corner.

A* shines when the heuristics are guiding the algorithm in the right direction, but some worlds might have dead ends. This slows down the algorithm, and the worst case running time is in fact not improved over Dijkstra's algorithm. A* will most of the time explore fewer nodes than Dijkstra's algorithm, but some maps might force it to explore every node before the goal is reached.

4.3 GRID ALGORITHMS

The algorithms described in this section are specifically designed for solving the pathfinding problem in 8-connected grid worlds. Dijkstra's algorithm and A* works on both graphs and on grid maps, but the following algorithms works on grid maps only.

4.3.1 *Jump Point Search (JPS)*

JPS is a recently developed pathfinding algorithm, for use on 8-connected grid maps. It was first described in 2011 by Daniel Harabor and Alban Grastien from The Australian National University [6].

JPS sets out to tackle one fundamental issue with the grid world representation. Grid worlds typically feature a high degree of symmetry. Symmetry in this sense is seen, when many different paths, that share the same start and end points, have the same length, but go through different nodes. Symmetry is especially present in large open areas with no obstacles, as seen in figure 13. Here, conventional algorithms are forced to explore a large number of nodes without

making any real progress, whereas JPS quickly finds special nodes of interest called jump points, which is used to quickly skip these areas.

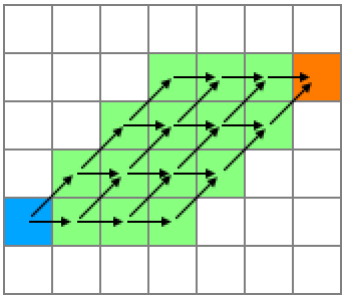


Figure 13: Several paths of equal length connects the blue and the red node.

JPS is built upon A*. It uses the same structure with an explored set, a frontier set and a ranking based on the F value. What separates JPS from A* is the successor function. In the case of both Dijkstra's algorithm and A*, the successors of a node was its neighbors (excluding those already in the explored set). JPS uses a different successor function, which exploits the ordering of the nodes inherent in the grid world.

In figure 14 we have illustrated a situation, where we are interested in finding the successors of the green node. We can start by ignoring the node we came from, referred to as our parent node, as it has already been explored. Secondly we can ignore the nodes diagonally behind us, as they can be reached directly from our parent node. The same goes for nodes above and below us. The two nodes diagonally ahead of us are ignored too, since we can choose to prefer the symmetric paths going through the nodes above or below us.

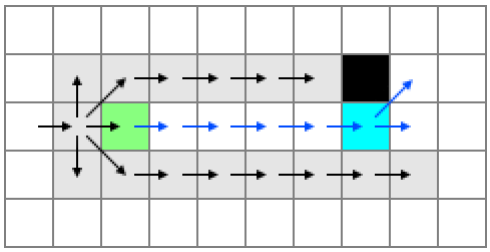


Figure 14: In this figure we want to find successors of the green node.

This leaves us with only one possible successor, but by the same argument, this node also only has one successor. So instead of taking one step at a time, we may as well jump as far as we can in the horizontal direction. We keep skipping past nodes until, we reach a node with a so called forced neighbor. In figure 14 the blue node has a forced neighbor behind the obstacle. Normally we do not have to look at the nodes diagonally ahead of us, but because of the obstacle

we are forced to look at it, hence the name. The blue node becomes the only successor of the green node.

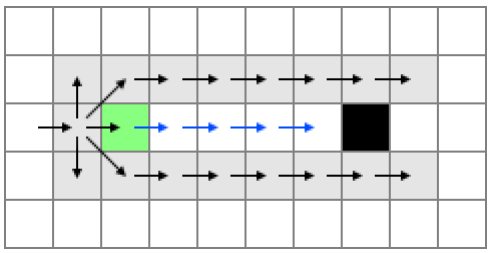


Figure 15: The green node has no successors.

Another case to consider, is if we reach an obstacle directly in front of us, as is illustrated in figure 15. Here we can safely conclude that no successors to the green node were found, as we have already assumed that the nodes above and below are preferably reached via other paths. Similar examples can be given for movement in the two vertical directions as well as in the other horizontal direction, but when we are moving diagonally it is slightly more complicated.

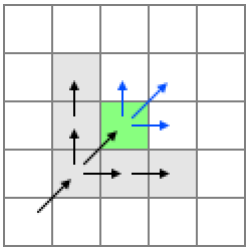


Figure 16: The green node has its parent diagonally behind it.

In figure 16 we see a node marked in green, with its parent diagonally behind it. We can see how five of its neighbors can be ignored, as they are more easily reached through other nodes. The neighbors above and to right can be reached by symmetric paths from the parent, however here we prefer the path going through the green node. If obstacles are present we might be forced to consider even more neighbors, as is illustrated in figure 17.

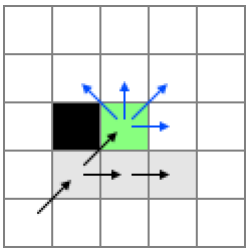


Figure 17: The green node is forced to consider four neighbors.

When moving diagonally we have three or more neighbors to look at. This makes it a little harder to jump forward like we did in the horizontal example. In figure 16 two of the neighbors require horizontal or vertical movement, so we can start by expanding these, and see if there are any nodes with a forced neighbor in these directions. Next we take another step in the diagonal direction and repeat the process of searching horizontally and vertically. This leads us to the situation illustrated on figure 18. In this case the successor of the green node is the blue node, since it is in line of sight of a node with a forced neighbor.

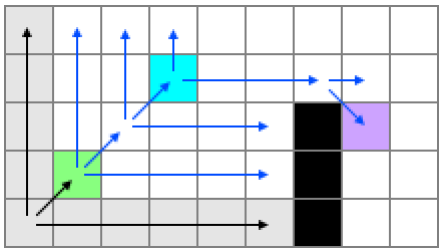


Figure 18: A node (green) is searching diagonally for successors. A successor node (blue) is found, since it is in line of sight of another node, which has a forced neighbor (purple).

With these tricks in mind we can summarize the successor function like so: We start by fetching all the neighbors like we did with A*. We then perform a pruning step, where we remove the neighbors that are more easily reached from our parent. If no parent exists, such as in the case of the start node, we do not prune any of the neighbors. We then call a recursive function on each of the remaining neighbors, in which we expand them outwards looking for forced neighbors. Some of the expanded nodes might not lead to a forced neighbor, which means they can be ignored. This leaves us with only a few successors, which might lie several steps ahead of us.

Pros and cons

JPS manages to seriously reduce the number of explored nodes. On figure 19 we can see the amount of explored nodes by JPS as compared to A*.

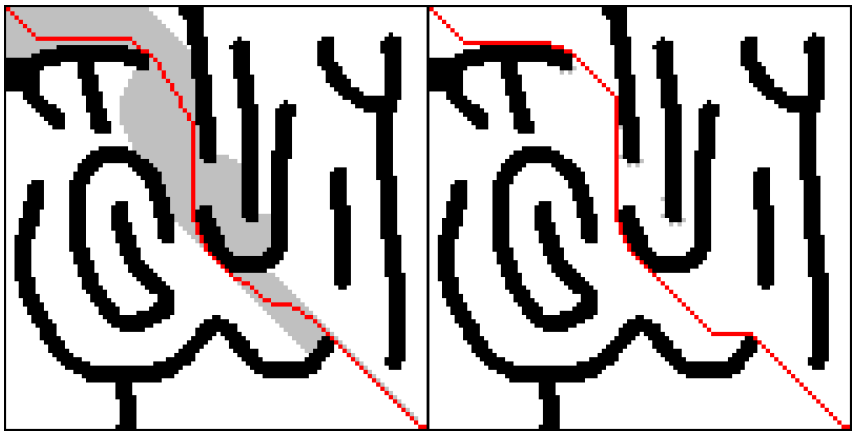


Figure 19: A* (left) compared to JPS (right), with explored nodes marked in grey.

On the surface it looks like JPS only visits a very small amount of nodes, but the truth is that many more nodes are visited. Even though only a few nodes are added to the explored set, the successor function does visit many more nodes. This means that the theoretical worst case running time is not improved over Dijkstra or A*. Whether or not JPS is faster in practice remains to be seen from the experiments.

4.3.2 *HPA**

In their 2004 article Near-Optimal Hierarchical Pathfinding Adi Botea, Martin Mller and Jonathan Schaeffer describes, how pathfinding in 8-connected grid maps can be speeded up using a hierarchy of abstract maps [7]. The 8-connected grid map is located at the bottom of the hierarchy. Above that is a less detailed abstraction, in the form of a graph. Further levels can be built on top, however in this report we consider a hierarchy with only one abstract layer. The main idea is then to use the abstract layer to quickly find an approximate path, and use this to speed up pathfinding in the original map. The drawback is that optimality is no longer guaranteed.

The abstract layer is created by first splitting the grid map into equally sized squares. We shall refer to these squares as clusters. In figure 20 we have illustrated a grid map which has been split into six 10x10 clusters (This is the cluster size used in our implementation).

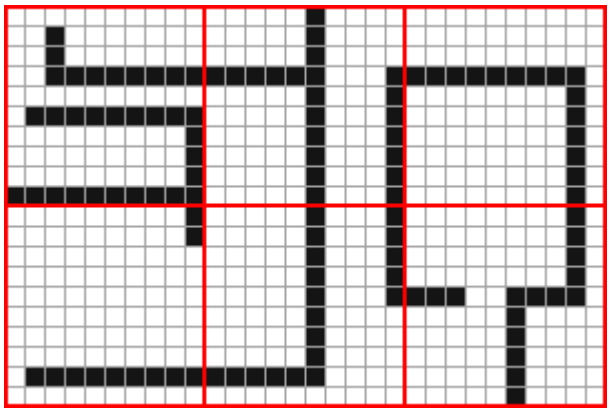


Figure 20: A grid map split into 6 clusters.

Once the grid map has been separated into clusters, we can start the creation of the abstract graph. First we need to choose which cells in the grid map should be used as nodes in the abstract graph. We define entrances as the traversable areas between clusters. For each entrance with a width of 5 or smaller, we mark the node pair in the middle of the entrance. If the width of an entrance is larger than 5, we mark the nodes furthest apart in the entrance. The result of this is shown on figure 21.

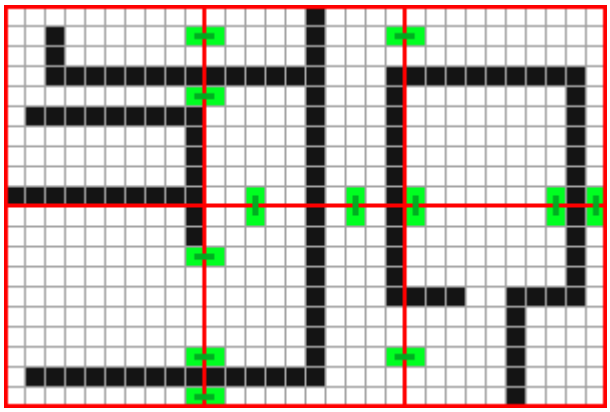


Figure 21: Node pairs at each entrance is marked.

In the abstract graph, the nodes in each marked pair are connected with an edge. We call these edges for inter-edges as they connect nodes across different clusters. The weights of the inter-edges are always 1 since the nodes are neighbors to each other in the grid map.

Next we need to create the intra-edges. Intra-edges connect nodes inside each cluster. We do this by pathfinding from each node in a cluster to every other node in that cluster. If a path exists between two such nodes, we create an intra-edge with weight equal to the path length found. In figure 22 intra-edges has been added to the abstract graph.

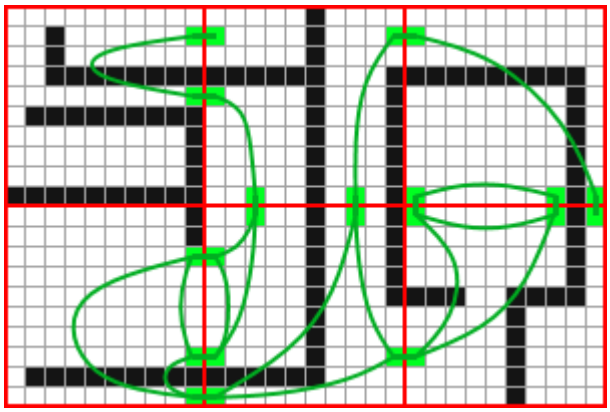


Figure 22: Both inter- and intra-edges has now been created.

At this point the overlaying abstract graph is done. Note that everything we have done so far is independent of the location of the start and goal position, which means we can do this once to speed up all future pathfinding queries. That is however only if the map does not change.

When we want to pathfind we have to start by inserting the start and goal node into the abstract graph. We want to connect the inserted node to all other nodes belonging to the same cluster. This is done by letting Dijkstra's algorithm create a shortest path tree for the cluster, with the inserted node as source node. When such a tree is done it tells us the path lengths from the inserted node to all other nodes in the same cluster. These path lengths are then used as weights for the connecting edges. With the start and goal node added to the abstract graph, we are ready to perform the next part of the pathfinding. First a path is found in the abstract graph using A*, connecting the newly added start and goal nodes. This path is then used as a list of partial goals as A* searches the grid map for the final path. By first calculating an approximate path we can avoid most dead ends, which should shorten the running time.

Pros and Cons

HPA* is able to avoid large dead ends of the map, which the other algorithms struggle to get around. It does this by first calculating an approximate path in the abstract graph. It then uses this approximation to guide the A* algorithm through the grid map to find the final path. While this may speed up the process it comes at the expense of path optimality. Since the final path is based on an approximate path we can no longer guarantee that it is optimal, however the path length is not more than 1 % longer than the optimal grid path length [7]. Another downside is the reliance on a preprocessing step. If the map is constantly changing, we do not have the luxury of a preprocessing step. However, if only a single node in the grid map changes, any

recalculations of the abstract map need only be done for the affected clusters.

4.4 POLYGONAL ALGORITHMS

Pathfinding in polygonal worlds are, in contrast to pathfinding in grid worlds, confronted with the problem of a continuous world. In this world we have to consider an infinite number of valid positions. One of the main challenges is to choose which positions to consider. The algorithms in this section describe methods for solving the pathfinding problem for polygonal worlds. The algorithms are all optimal, in the sense that they always return the shortest path.

4.4.1 *Visibility Graph (VG)*

The most basic optimal algorithm for pathfinding in polygonal worlds relies on the construction of a visibility graph. A visibility graph describes, which corners can be seen from each other. Visibility graphs are created by connecting every corner with every other corner in line of sight. In figure 23, the visibility graph of a polygonal world is illustrated.

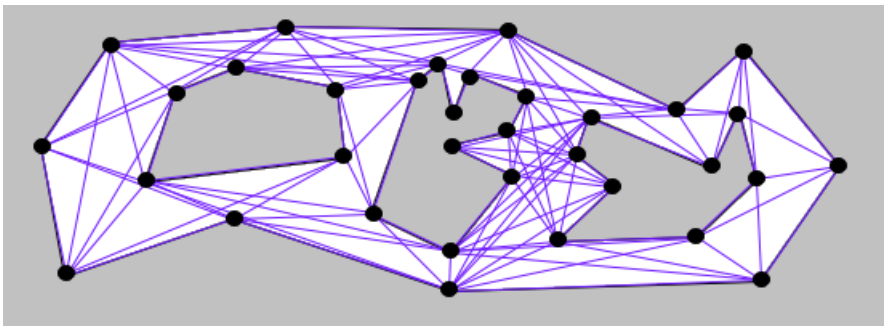


Figure 23: A polygonal world, with its visibility graph, consisting of nodes and edges.

Previously we explained how the shortest path between two positions, can be described by a list of protruding corners only. Therefore, in the context of pathfinding, the depressed corners can be excluded from the visibility graph. In figure 24 the visibility graph used for pathfinding is illustrated.

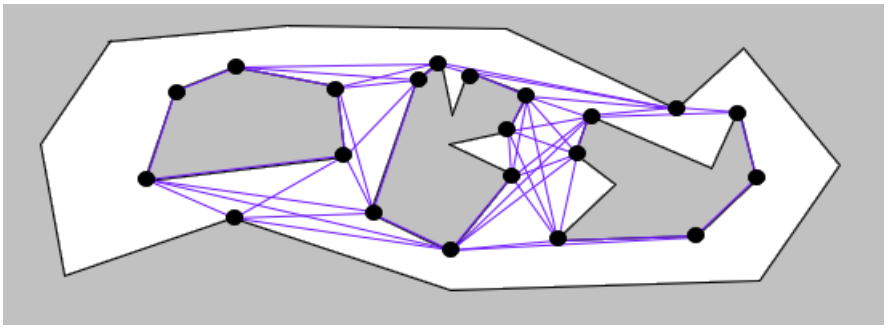


Figure 24: The visibility graph used in the context of pathfinding.

The graph is independent of the start and goal positions, which means it can be created in preprocessing. When the algorithm is then given a start and goal position it starts by inserting them into the graph, by connecting them to every protruding corner in line of sight (see figure 25).

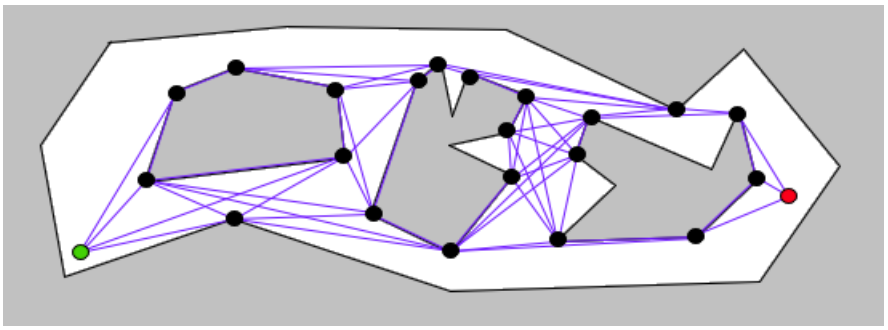


Figure 25: The start and goal nodes are added to the graph at runtime.

Once the start and goal nodes have been added, a graph searcher can be used to find an optimal path from the start node to the goal node (see figure 26).

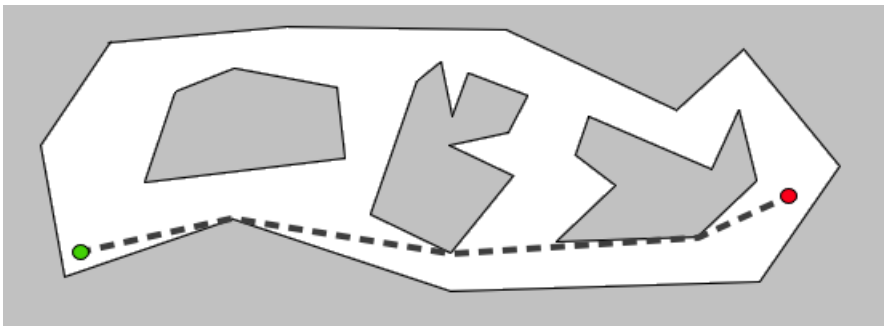


Figure 26: The shortest path between two positions in the polygonal world.

To summarize this gives us the VG (Visibility Graph) algorithm: A visibility graph is created in preprocessing, the start and goal node are inserted at runtime, and A* is used for connecting the start and goal node.

To create the visibility graph we first created a BSP (Binary Space Partitioning) of the world. Using this we were able to quickly find all visible corners as seen from any given point. We used it to calculate the visibility of each protruding corner to form a graph similar to the one in figure 24. No edge weights had to be stored in the graph as these were always the distance between each corner, which is quickly calculated at runtime. The BSP was also used when the start and goal node had to be inserted. The technicalities of how a BSP works is discussed in more detail under the section about the BSP* algorithm.

Pros and Cons

The benefit of this approach is, that during pathfinding, instead of considering the infinite number of positions existing in a polygonal world, the algorithm only consider positions of protruding corners. This, not only speeds up the pathfinding by itself, but it also permits the transformation of the problem into a graph searching problem, which we know how to solve. The drawback is that the heavy preprocessing makes the algorithm less suited for dynamic maps. If the map is dynamic a new visibility graph has to be calculated every time the map changes.

4.4.2 *Visibility Graph Optimized (VGO)*

The next algorithm we look at is the Visibility Graph Optimized (VGO). This algorithm is an extension of VG. It exploits two properties of the polygonal world that we discovered, which we hope can improve the running time. These properties lead us to both VGO, and BSP*. In this section we describe the two properties, as well as the VGO algorithm, and in the following section we describe the BSP* algorithm.

Property One

For every protruding corner in a polygon, we can split the traversable area around that corner into three separate areas. We will refer to these areas as A, B and C. A and C are the areas that lie on opposite sides of the corner, and B is the area in front of the corner. The borders of the areas are defined by the extension of the two line segments that constitutes the corner. This is all illustrated in figure 27.

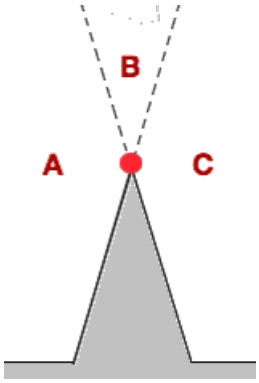


Figure 27: The traversable area around every protruding corner can be split into three separate areas.

Consider a situation during pathfinding, where we have positioned ourselves somewhere in the B area in figure 27. At this point we should either move to a visible protruding corner, or, if the path is unobstructed, move straight to the goal position. The question remains: should we ever consider moving to the protruding corner marked in red in figure 27? The answer is no, since any position, in the traversable area, is better reached, either directly or through other protruding corners. This also follows from the fact, that no straight line connecting a position in area B to a position in area A or C, will ever go through this protruding corner.

Now consider the opposite situation where we stand at the protruding corner in figure 1. Then, by the above, we can conclude, that our previous position must have been in either area A or C. Lets say, without loss of generality, that our previous position was in section A. The question is now: Should we consider moving to any protruding corner in section B? Again the answer is no, since any position in section B are better reached from our previous position in section A, either directly or via other corners than the one marked in red. This means, when we are standing at a protruding corner, then we can ignore those other protruding corners, that lie in the B area defined by the corner we stand at. This applies for any protruding corner, but the angle around different corners may vary. Because of this the B area will either be wide or narrow.

To summarize, we can say that each corner has an associated B section, which is an area extending out in the direction the corner is pointing. If a corner is pointing towards us such that we are located in its B section, we can ignore it when pathfinding. Furthermore, if we stand on top of a corner, we can in general ignore all other corners located in the B section defined by our corner (See figure 28).

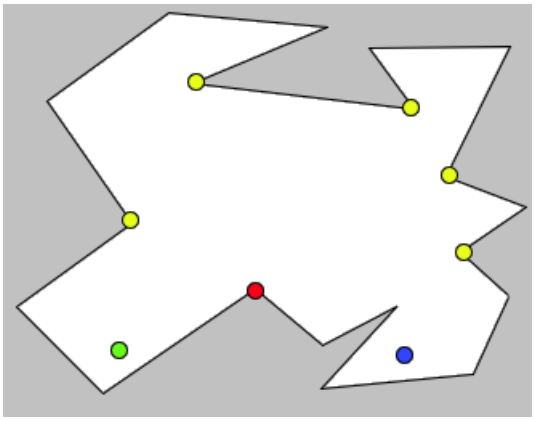


Figure 28: We want to find a path from the green node to the blue node, and we have just arrived at the red node. Because of Property One, we can at this point ignore all the protruding corners marked in yellow.

The result of this property is that we can ignore specific protruding corners, depending on where we are. The number of protruding corners to consider is therefore minimized even further. This property can be applied during preprocessing to minimize the visibility graph, as it is a static property that is independent of the start and the goal position. If no preprocessing is allowed we can apply it at runtime instead.

Property Two

We can now ignore both depressed corners as well as any protruding corner adhering to Property One. The remaining corners all have something in common: they obscure the area behind them. In this sense, all remaining corners cast a shadow as seen on figure 29.

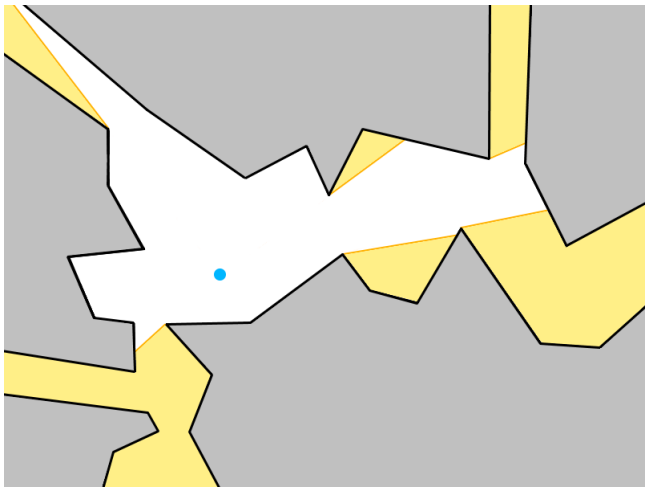


Figure 29: Seen from the blue dot, the areas marked in yellow are in shadow.

Any point which is not in shadow is best reached directly. This means that a path going through one of the remaining corners, is only optimal if it leads to somewhere in shadow. Now consider the situation, where we have moved to one of these corners. At this point, only the traversable area, which was previously in shadow, is of interest, since everything else is better reached from our preceding position. This means, that whenever we arrive at a corner we should check, where we came from, and use this to limit the area, in which we search for succeeding positions. This property cannot be applied during preprocessing, as it relies on the preceding position which is only defined during pathfinding.

VGO

The Visibility Graph Optimized algorithm (VGO), is an extension of the the basic visibility graph algorithm (VG). It still builds a visibility graph, however it reduces the number of edges in the graph using Property One. To give an idea of how many edges are pruned by Property One refer to figure 30:

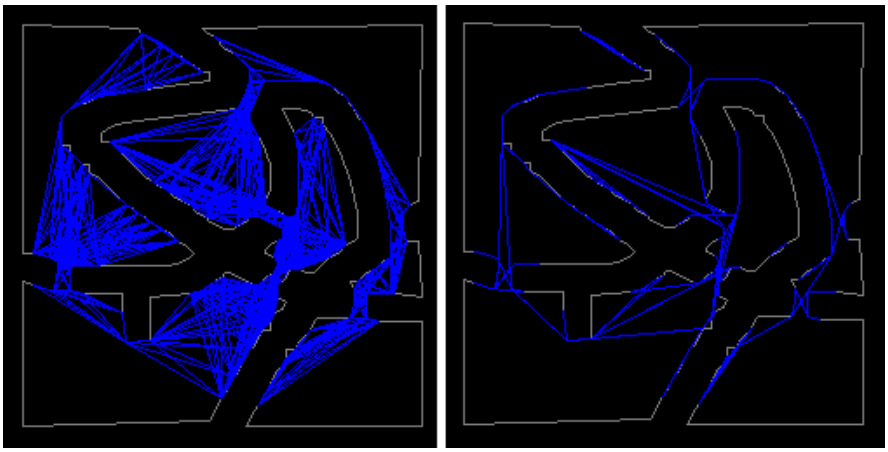


Figure 30: On the left we see the visibility graph used by the VG algorithm (VG). On the right we see what is left of this visibility graph after we have applied property one (VGO).

At this point we are left with a visibility graph consisting of only protruding corners, and edges not removed by Property One. This, in fact, is a minimal visibility graph for pathfinding, since all edges are used in at least one optimal path. This can be seen from the fact, that all edges connect corners, where none of the corners are located in the B section of the other corner. This implies that at least one of the line segments that constitutes each corner, is not visible from the other corner, and a shortest path from a point on one of those line segments to a point on the other line segment, must go through the edge.

At runtime the start and goal nodes are added to the graph, and pathfinding is done using A*. Property Two is used at runtime to limit the number of protruding corners to consider even further.

Pros and cons

A definite improvement over the VG algorithm is a reduced visibility graph. This not only speeds up pathfinding, it also reduces the amount of memory required. We also apply Property Two at runtime to ignore corners, but this requires an extra bit of calculation, which might not be worth it. It remains to be seen if Property Two effects the algorithm positively or negatively in comparison to VG.

4.4.3 *BSP**

The final algorithm we look at is called BSP*. BSP* is our contribution to the field, and is meant for use in dynamic polygonal worlds. The algorithm relies on a Binary Space Partitioning (BSP) in combination with A*. It utilizes the BSP's ability to early terminate when searching for visible corners. For the algorithm to be dynamic we have to

create a new BSP prior to each pathfinding query, however this is done relatively fast as compared to the pathfinding itself. Another possibility would be to create the BSP in a preprocessing step, but then we would not be able to consider it as being dynamic.

A BSP partitions the world into a tree like data structure. It does this by choosing a random line segment from the polygonal world, which is used to split the rest of the line segments into two groups. The line segments, that lie to the left of the chosen line segment, are placed in one group, and those to the right in the other group. If a line segment has end points on either side of the chosen line segment, it is split into two line segments which are placed in opposite groups. In this sense the chosen line segment is used to cut the space into two halves. The algorithm recurses on each group to create a binary tree structure of line segments. At the leaves of the tree are the empty areas that has no line segments in them.

When such a structure has been created, we can use it to quickly calculate visibility as seen from any given point. When traversed correctly the BSP tree will visit the line segments in order, such that the line segments that lie closest to the given point are visited first. We can therefore be sure that a visited line segment never covers an earlier visited line segment. Consider for instance the first line segment visited by the BSP. For this segment we can be certain that the corners at each end are visible, so we add them to a list of visible corners. Subsequent line segments might be fully or partially covered by the line segments already visited. This means we have to keep track of what parts of our view field have already been covered. Each line covers an angular interval of our view field as is illustrated in figure 31. Each time we visit a new line segment we have to add its angular interval to a list of intervals. If two such intervals overlap we can join them to form one interval instead. In the worst case we cannot join any intervals, which makes the list of intervals grow large. We shall see in a bit how this affects the running time.

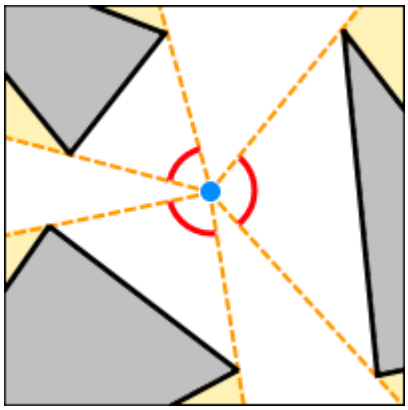


Figure 31: A point marked in blue from which we want to calculate visibility. So far the BSP has returned the three closest line segments, which means that certain angular intervals of our view field are blocked (marked in red).

We can stop this procedure as soon as the angular intervals cover our entire view field. This is what we call early termination. This is an important part of why we are able to speed up the process. Property Two told us that the area, in which we have to search for successor points, can be significantly reduced if we take our preceding position into account. In figure 32 we have marked in red the angular interval, which can be ignored because of Property Two, when we stand at the corner marked in blue. This means that Property Two combines especially well with the BSPs ability to early terminate, since we have already from the beginning of the traversal of the BSP tree excluded a large part of our view field, which means that the early termination is not many steps away.

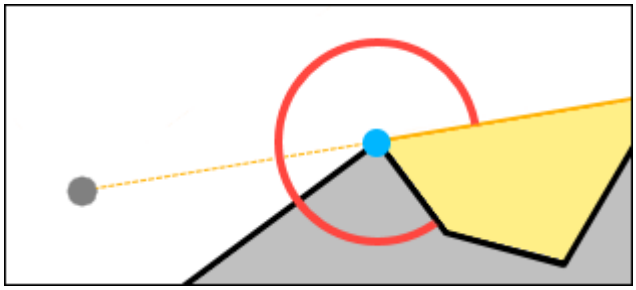


Figure 32: If we at some point have to search for successor points, we can use our preceding position (grey dot) to limit the area in which we have to search (we only have to search the yellow area). This leads to an even earlier early termination, when used in combination with a BSP.

We can summarize the entire algorithm like so: We start by creating a BSP of the polygonal world. We then use the BSP to check, which protruding corners are visible from both the start position and the

goal position. From there on we use A* to move from corner to corner all the way from start to goal. Instead of relying on a visibility graph, we rely on the BSP tree to calculate visibility every time we need it. In this process we can exclude all depressed corners and also those protruding corners adhering to Property One. Property two helps us limit the angular search space each time we have to calculate visibility. As soon as A* reach a corner, that is visible from the goal position, we have found an optimal path.

4.4.3.1 *Running Time*

BSP* uses A* to decide which corners should be explored. A* has a theoretical worst case running time of $O(E + N \log N)$, but this was assuming that the successor points at each iteration could be retrieved in constant time, as is the case with graphs and grid maps. BSP* has to do a visibility calculation at each iteration. The running time of this calculation is highly dependent on the size of the BSP tree. Therefore we should figure out the worst case size and the expected size of the BSP tree.

To figure out the worst case size of the BSP tree, consider the polygonal world illustrated on figure 33.

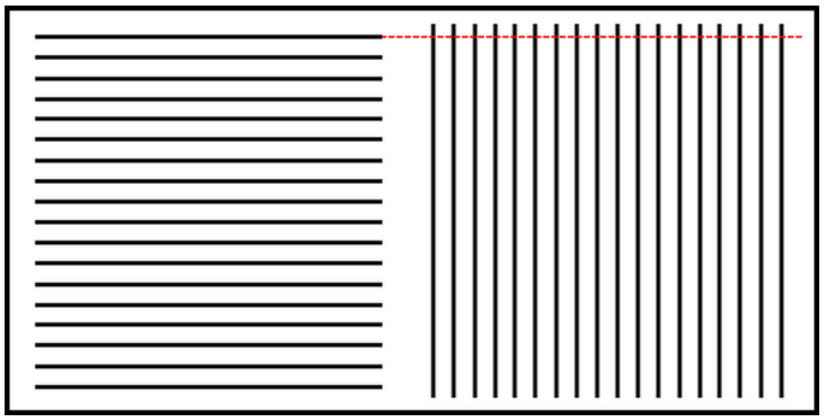


Figure 33: Line segments of a particularly bad case of a polygonal world.

Here we have N horizontal line segments and N vertical line segments. When constructing the BSP tree we choose random line segments, which splits the world in two, followed by a recursion on each half. Consider the case where we choose the top most horizontal line segment as the first splitter. It will intersect N times with the vertical line segments. The left child of the first node in the BSP tree will therefore have N small vertical line segments. The right child will have $N - 1$ horizontal line segments and N vertical line segments. In the worst case the line segments in left child will split off one at a time, creating a very unbalanced subtree of depth N. In the right

child we choose the second from the top horizontal line segment as the splitter. This creates a similar situation to the one we started with; its left child will have N small vertical line segments, and the right side will have $N-2$ horizontal line segments and N vertical line segments. If we continue doing this we create N subtrees, each with depth N, which gives us the tree illustrated on figure 34.

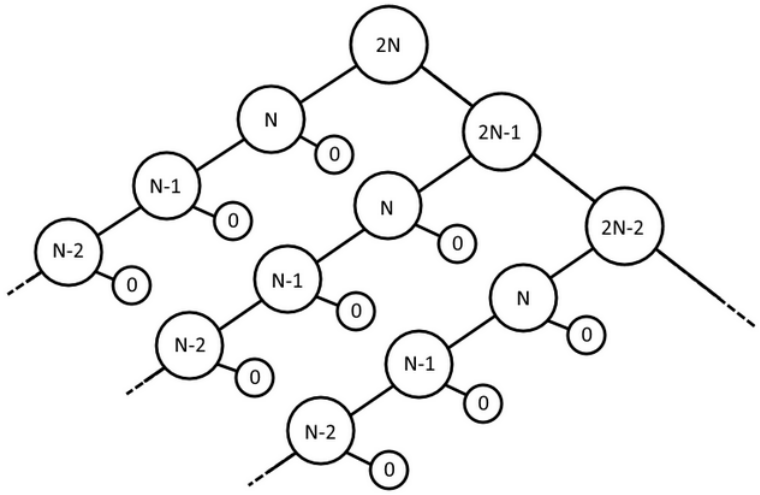


Figure 34: The worst case BSP tree created by the line segments from figure 23. The number of line segments to consider at each recursive call in the construction of the BSP tree is written inside the nodes.

From this we can see that the worst case tree size is $O(N^2)$. We can also see that at each recursive call we have to check for $O(N)$ intersections to create the child nodes, which gives us a worst case construction time of $O(N^3)$. Luckily this only happens approximately $\frac{1}{N^{N^2}}$ of the time, which means it only happens one googolth of the time for N as small as 10.

A much more interesting theoretical analysis is to figure out the expected size and expected construction time of the BSP tree. Let us define $l(s_i)$ as the line that contains segment s_i, and let us define c_i as the number of intersections made by $l(s_i)$. The expected number of splits done, at BSP construction, by s_i is $\log(c_i)$ [8]. And since c_i is bounded by N the expected number of splits per segment is bounded by $\log(N)$. This means we get an expected size of the BSP tree of $O(N \log N)$. Bounding c_i by N is a very high bound though, and initial testing showed us that the average c_i is below $0.05 \cdot N$ for all map types. This means that the N term inside the log part has a very small constant. What is also interesting is that this ratio goes down as N goes up, which means that the expected size is $O(N)$ for all practical purposes. This means that the expected construction time is $O(N^2)$.

So now we have a BSP tree with size $O(N)$. Next we have to use it to calculate visibility. For this we have to traverse the tree and retrieve the line segments in order from closest to furthest. This can be done in linear time since we only recurse on each tree node once. We also have to maintain a list of angular intervals though. If we maintain the list of angular intervals as a sorted list we can insert a new interval in logarithmic time using binary search. If intervals overlap we can join them, but in the worst case no intervals are joined and the size of the list grows to $O(N)$. Which means it takes $O(N \log N)$ time in the worst case to insert all angular intervals. The probability of having no overlaps is very small though and in practice we join almost all intervals, which gives us an expected running time of the visibility calculation of $O(N)$.

We do a visibility calculation every time we explore a new corner, and in the worst case we explore every protruding corner. We therefore expect to perform at most $O(N)$ visibility calculation. The total running time of BSP* is described in the table below:

	Worst Case	**Expected Case**
BSP Tree Size	$O(N^2)$	$O(N)$
Visibility Calculation	$O(N^2 \log N)$	$O(N)$
BSP* Running Time	$O(N^3 \log N)$	$O(N^2)$

Table 2: Worst case and expected complexities for BSP*, where N is the number of corners in the polygonal world.

This shows us that BSP* performs rather poorly in the worst case, but the expected running time looks much better. It does however still have a worse expected running time than all the other algorithms, in fact its expected running time is worse than the other algorithms worst case running times. If one had to improve on the expected running time, one would either have to come up with a sublinear visibility calculation or an improvement to A*, which guaranteed to explore a sublinear amount of corners.

Pros and cons

BSP* is to our knowledge the only optimal pathfinding algorithm meant for dynamic polygonal maps. This means that even if it is outperformed by all the other algorithms, it is still the best algorithm for finding optimal paths in dynamic polygonal worlds. We do not expect BSP* to be faster than the algorithms relying on preprocessing. We do hope for BSP* to be faster than A* and JPS on at least some maps.

As a developer you are usually forced to use the grid world representation, if you plan on implementing dynamic maps. BSP* opens up the possibility of using the polygonal representation even if the map is dynamic. It does have a worse expected running time than

all the other algorithms. This means that on large enough maps we expect it to perform worse than all the others.

Part III

EXPERIMENTS

5

SETUP

In this and the following chapters, we describe and discuss our experiments. In this chapter we explain the setup of our experiments, and in the next chapter we describe and discuss each map in detail. Following this, we discuss our findings in general. We created several maps with different structures and sizes. The algorithms we look at are:

- Grid Algorithms:
  - Dijkstra
  - A*
  - JPS
  - HPA*
- Polygonal Algorithms:
  - VG
  - VGO
  - BSP*

The map types we look at are:

- Star
- Lines
- Checker
- Maze
- MazeScale
- MazeCor5

Hardware

Our experiments were performed on a 1.4 GHz Intel Core i5 processor, with 8 GB 1600 MHz DDR3 ram, using the operative system OS X Yosemite.

Creating the Maps

Here, a general explanation is given of how the maps were created. In order to properly test our algorithms, we needed grid maps and polygonal maps of varying size. The grid maps were generated from

black and white images of varying size. The images were either created with the use of a standard paint editor or in the case of our maze maps with an online generator.

The polygonal maps were created from the grid maps. We developed an algorithm (See appendix) that were able to detect the shape of the non-traversable grid cells and then create a polygon around this shape. With this method we were able to, not only create polygonal maps in a relatively fast and systematic way, but also to have some sort of direct conversion between the worlds. This allowed for our algorithms to be compared across both world representations. In figure 35 an example of a black and white image is given, together with the resulting grid map and polygonal map.

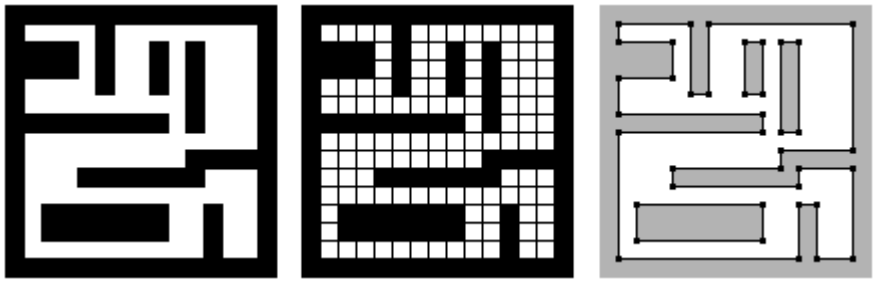


Figure 35: An example of a simple black and white image that is converted into a grid map and a polygonal map.

Smoothing

For some maps we also performed a smoothing operation on the polygonal representations. The smoothing maintained the general shape of the map, but removed jagged walls as illustrated on figure 36.

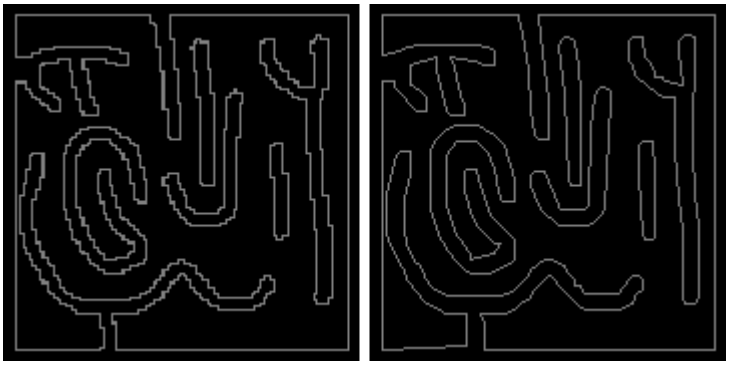


Figure 36: An example of a map before and after smoothing.

It is hard to compare algorithms across world types, but we thought that if someone ever chose to go with a polygonal representation of theirs maps, they would not make the walls jagged. If jagged walls

are what you want, you might as well go with a grid world representation instead. Smoothing reduces the number of edges in the polygonal world, but instead of seeing this as an unfair advantage given to the polygonal algorithms, we see it as something inherent to polygonal worlds.

6

RESULTS

In this section we present our results for each of our different map types.

6.1 THE STAR MAP

The first world type we look at can be described as an open world with obstacles. It is a world with no deep dead ends. Instead, it is populated with obstacles of limited size, that are evenly spread throughout the world. It exemplifies landscapes where the overall direction can be kept during pathfinding, but where small detours are needed to get around obstacles. This could for example be movement in a city by foot. To simulate this kind of world, we use the Star Map (See figure 37). The reason, we call it the Star Map, is because we have chosen the star shape for all the obstacles. We chose the star shape for obstacles as it has more detail than just squares or circles including small dead ends, letting it represent a wider range of obstacle shapes.

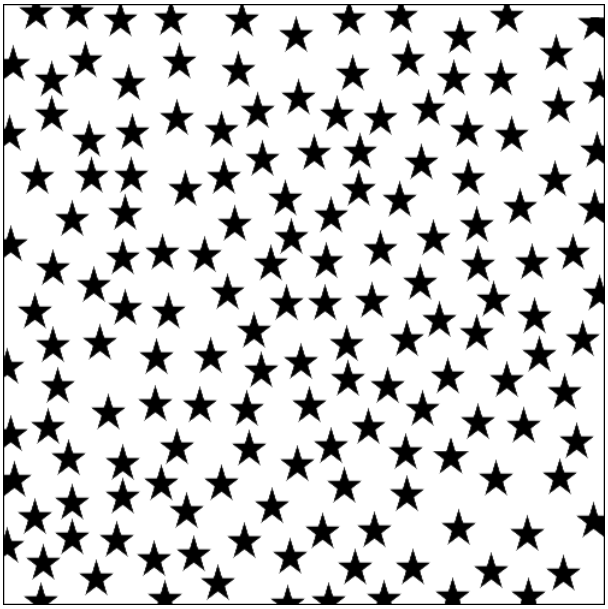


Figure 37: The Star Map with a side length of 500 pixels.

For this world type we measured the performance of the algorithms in maps with a side length of 100, 200, 300, 400 and 500 pixels. We created the different map sizes by cropping out sections of the biggest map, such that the size of the obstacles remained the same. This made it resemble smaller and smaller portions of the same kind of terrain (See figure 38).

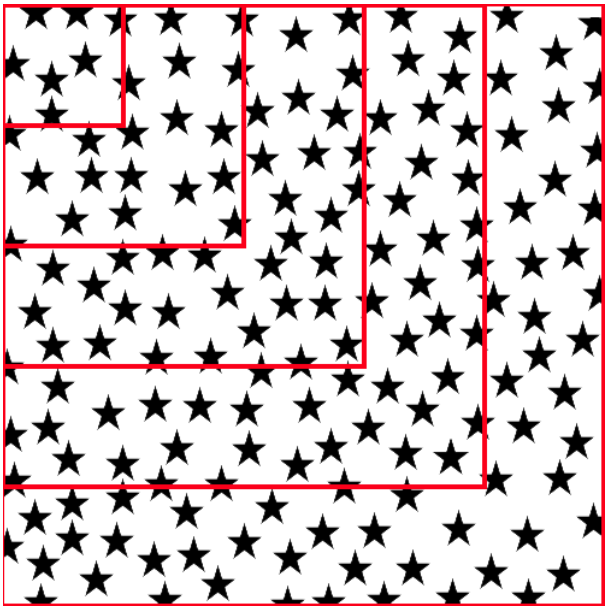


Figure 38: A depiction of how we cropped out smaller and smaller maps from the biggest map.

For each map we measured the time it took to find the shortest path from the center of the map to each of the four corners, and then we averaged the time. The reason for picking the center of the map instead of a corner, was that if we started in a corner then some directions would already be excluded for pathfinding. It would then be less clear for us to see the effect of the heuristics. The reason for not just measuring one path, instead of four, was that we wanted to measure the algorithms' overall ability to navigate in this world type. The results for pathfinding in this map can be seen in figure 39.

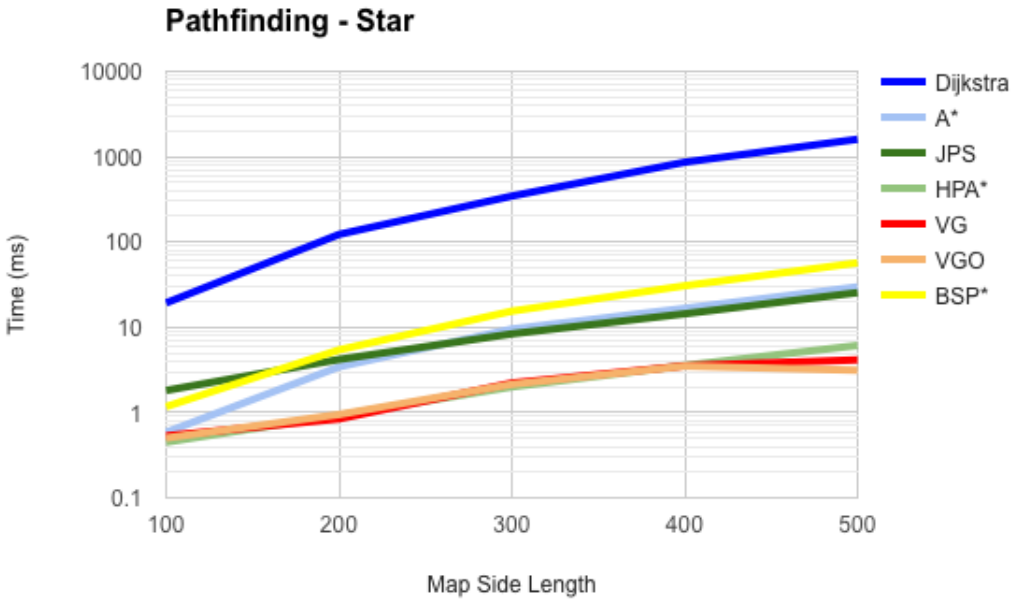


Figure 39: Results for pathfinding in the Star Map.

From the pathfinding results in this map we can see that Dijkstra is significantly slower than all the other algorithms. For the largest map Dijkstra uses between 1 and 2 seconds. We also see that VG, VGO and HPA* are the fastest. For all map sizes their running time is below 10 ms. In between lies A*, JPS and BSP*, with BSP* being the slowest of the three. What separates VG, VGO and HPA* from the other algorithms is that they preprocess the map before pathfinding.

If we now only consider the dynamic algorithms: Dijkstra, A*, JPS and BSP*, we can see that Dijkstra stands out from the others. This is probably because Dijkstra is the algorithm that does not use a heuristic to guide it towards the goal. In this map, since the overall direction can be kept during pathfinding, the heuristics provide good guidance for A*, JPS and BSP*. In figure 40 this is exemplified by showing how many grid cells Dijkstra and A* explores respectively.

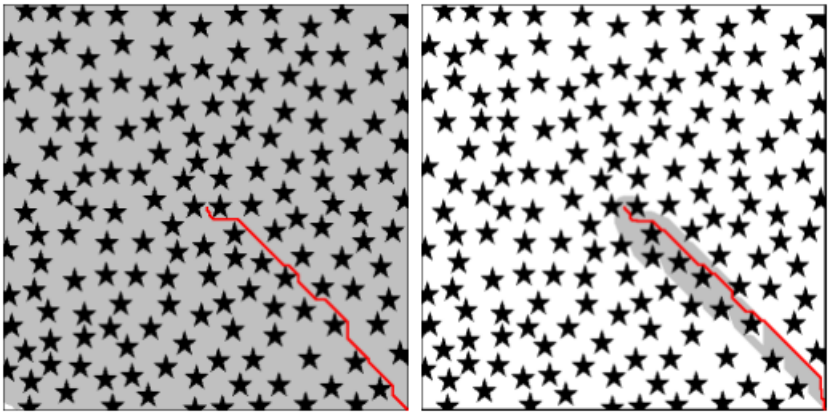


Figure 40: Explored areas are marked in grey. Dijkstra (left) explores significantly more grid cells than A* (right), before a path is found from the center to the bottom right corner.

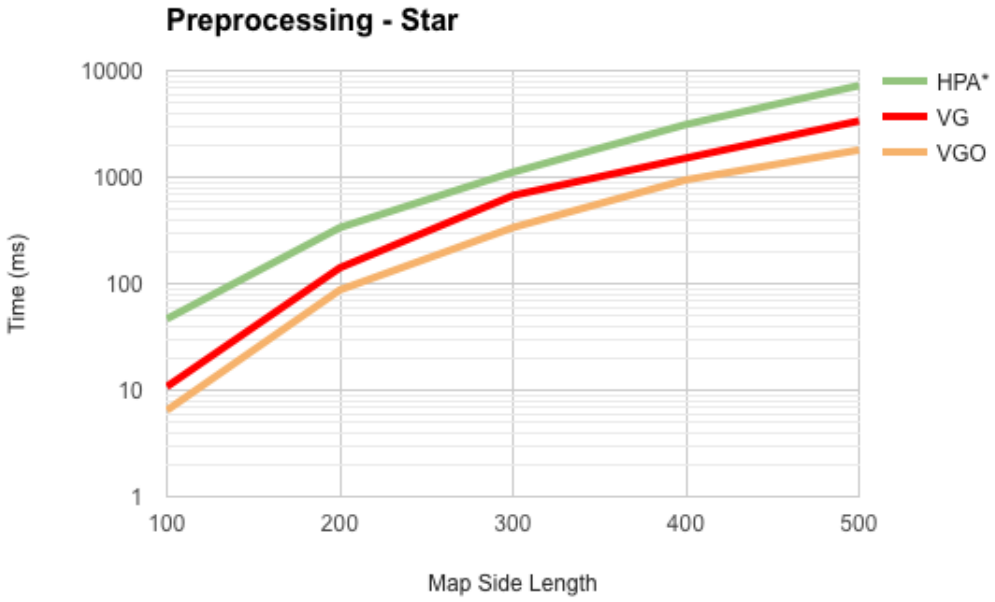


Figure 41: Results for preprocessing the Star Map.

Only three of our algorithms perform preprocessing prior to pathfinding, namely VG, VGO and HPA*. For the Star Map their time measurements for preprocessing is depicted in figure 41. The fastest preprocessing time is found with VGO. Worth noticing is it that VGO's preprocessing time is about the same as Dijkstra's pathfinding time. This shows us that the algorithms that perform preprocessing would have had problems if the map had been dynamic.

6.2 THE LINES MAP

With the Lines Map we wanted a map with big dead ends, such that large detours was needed, as opposite to what was seen in the Star map. We wanted the map to be somewhat like a maze, but instead of many narrow corridors, as is typically seen with mazes, we wanted it to have big open spaces. With this setup we expected the pathfinding time for Dijkstra and A* to be similar, as A* would not have much use of its heuristics. Furthermore we wanted to see if JPS had some benefits compared to A*, when it came to exploring big open spaces.

The Lines Maps were created by hand in a paint editor. We created Lines maps with side lengths of 100, 200, 300, 400 and 500 pixels. In a similar fashion to the Star map, we created each map size by cropping out different sizes from the biggest map. The start position in each map was the top left corner and the end position was the bottom right corner. For each map size we made sure that the shortest path between the start and the goal position followed an overall Z-like pattern, such that the algorithms could not benefit too much from using heuristics (See figure 42 and 43). The results for pathfinding in the Lines map can be seen in figure 44.

Figure 42: The Lines map with 100x100 pixels (left) and 500x500 pixels (right). The shortest path is marked in red.

Figure 43: The polygonal counter parts of the maps seen in figure 42.

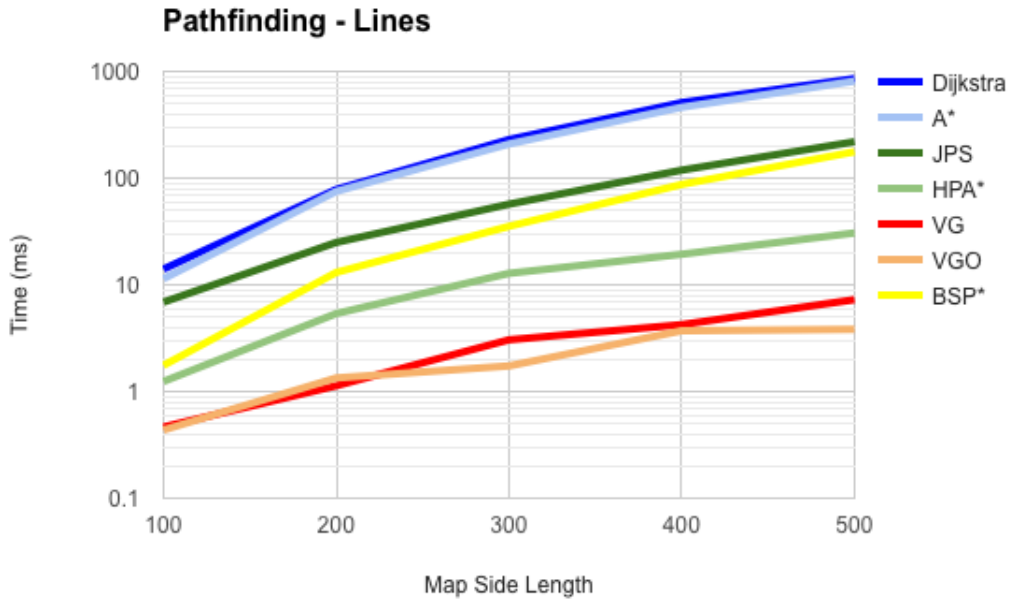


Figure 44: Results for pathfinding in the Lines Map.

The first thing to notice is that the pathfinding times for Dijkstra and A* are very similar. This shows us that using heuristics is not always of benefit. JPS is much better than both Dijkstra and A*.

BSP* performs better than the other dynamic algorithms. A simple explanation for this, is that larger spaces leads to an increased number of grid cells, but not necessarily an increased number of corners. This, in turn, leads to a slowdown of the grid algorithms only. However, for the larger maps the difference between JPS and BSP* seems to become smaller. This suggests that BSP* spends more time per corner, than JPS spends per grid cell. Since the number of corners and grid

cells both grows linearly in the map size, one would expect that BSP* would be slower than JPS, if the maps were even larger.

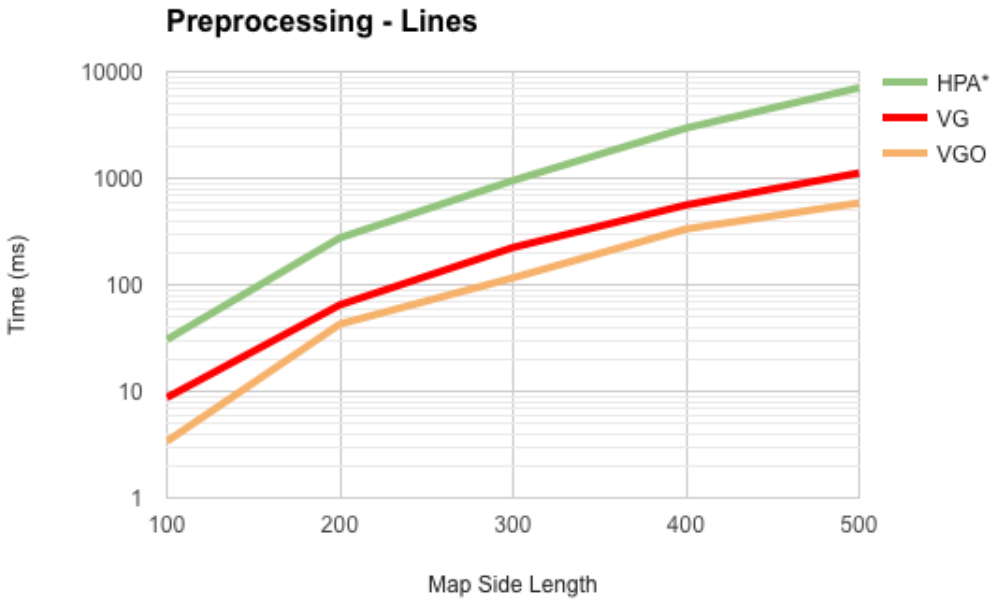


Figure 45: Results for preprocessing the Lines Map.

When it comes to preprocessing (See figure 45), we first notice that HPA* uses approximately the same time in this map as in the Star map for all map sizes. We also notice that VGO is faster than VG. This was also the case in the Star map, implying that the optimization used in VGO has an effect on the preprocessing time. The only difference between this map and the Star map, is that VG and VGO is faster in this map. This is most likely because of a smaller amount of protruding corners.

6.3 THE CHECKER MAP

With the Lines map we saw how the polygonal algorithms had an advantage when the map had big open spaces. With the Checker map, we wanted a map that was as bad as possible for the polygonal algorithms, while being easy to handle for the grid world algorithms. We needed a map that has many protruding corners and no open spaces. With such a setup the polygonal algorithms would have many more corners to consider, while at the same time making little progress with respect to getting closer to the goal position. The map we came up with consists of many small polygons arranged with narrow passages in between.

We tested map sizes of 20, 30, 40, 50 and 60 pixels. For each map size we measured the time it took to pathfind from the center of the map to each corner. We averaged the four running times to get an

estimate of the overall performance of each algorithm. The smallest maps for each world representation are depicted in figure 46.

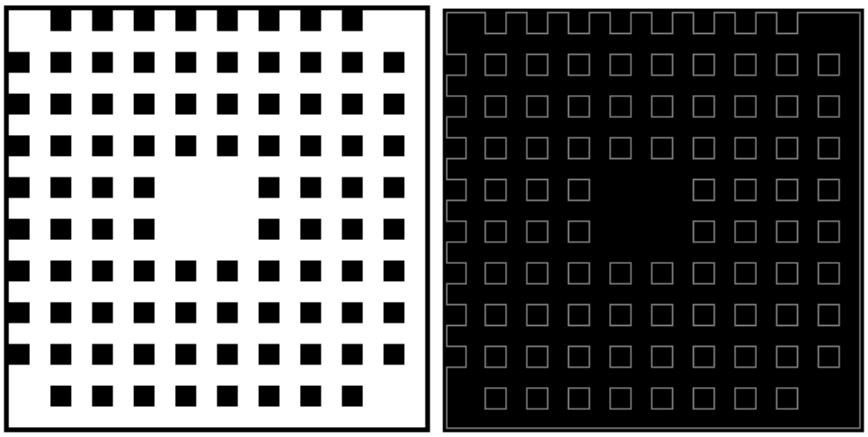


Figure 46: The 20x20 Checker maps.

While the number of neighbors for the 8-connected grid map is bounded by 8, the number of neighbors in a polygonal world is the number of visible protruding corners. For this map type this value explodes as the side length of the map increases. This is the reason why we have not tested maps beyond 60x60 pixels. In figure 47 the visibility graph for the above polygonal world is depicted after being preprocessed by VG. If the map side length is n there are $O(n^2)$ protruding corners. From each protruding corner we can see $O(n)$ other protruding corners. The total number of edges in the map is therefore $O(n^3)$. This means that in figure 47 there are approximately $20^3 = 8000$ edges. For the biggest Checker map with a side length of only 60 pixels, there are approximately $60^3 = 216000$ edges in its visibility graph.

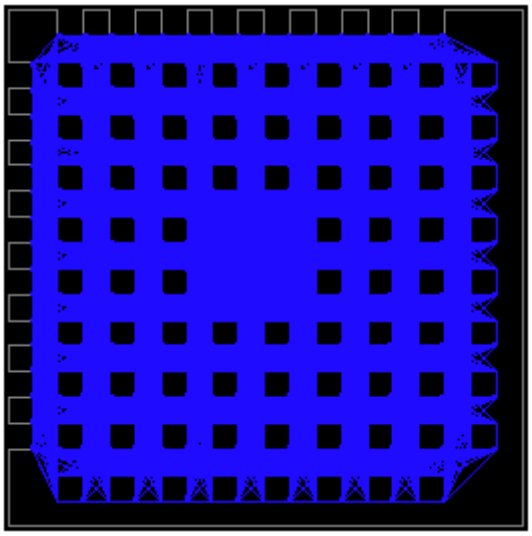


Figure 47: The visibility graph created by VG (Edges marked blue).

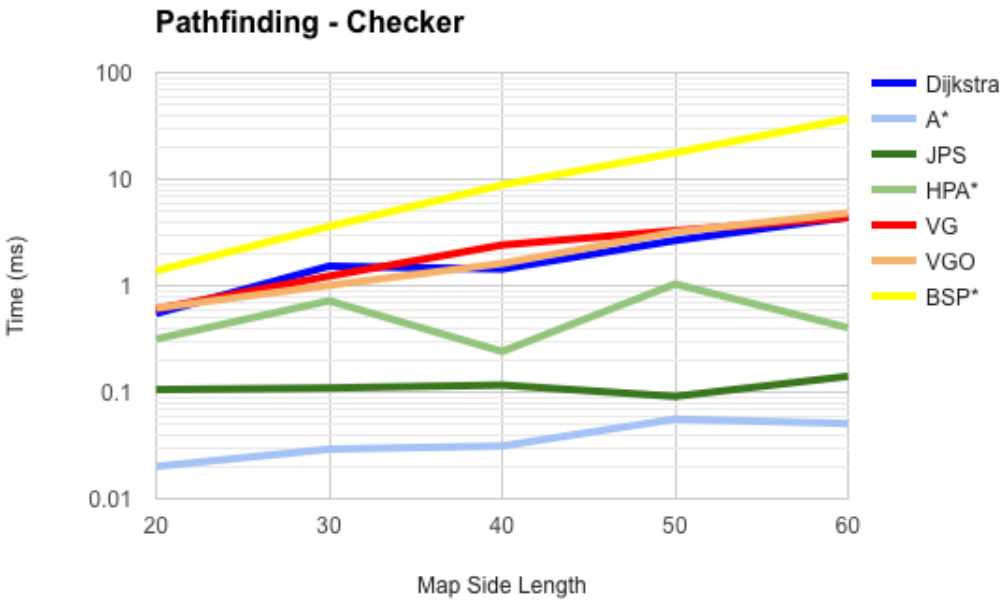


Figure 48: Results from pathfinding in the Checker map.

From figure 48 we can see that BSP* is the slowest pathfinding algorithm in this map. This is no surprise, as it is a polygonal algorithm with no preprocessing. The other two polygonal algorithms, VG and VGO, do preprocess the map prior to pathfinding, however they are still approximately as slow as the slowest grid algorithm, which is Dijkstra. Therefore in this map it is preferred to use a grid algorithm.

Among the remaining three algorithms, HPA*, JPS and A*. We see that HPA* is the slowest, and A* is the fastest. HPA* do preprocess

the map before pathfinding, but not much seems to be gained from doing so in this map. Two things work against HPA* in this map. The first thing is that HPA* has relatively many of entrances at each cluster border, which leads to a very large abstract graph. The second thing is that, since the map has no dead ends, it is really a waste of time to generate partial goals for pathfinding.

If we focus on the two fastest algorithms, A* and JPS, we see that A* is faster than JPS. This is no surprise as the map has no big spaces, which is what JPS excels at traversing. JPS is still relatively fast in this map though.

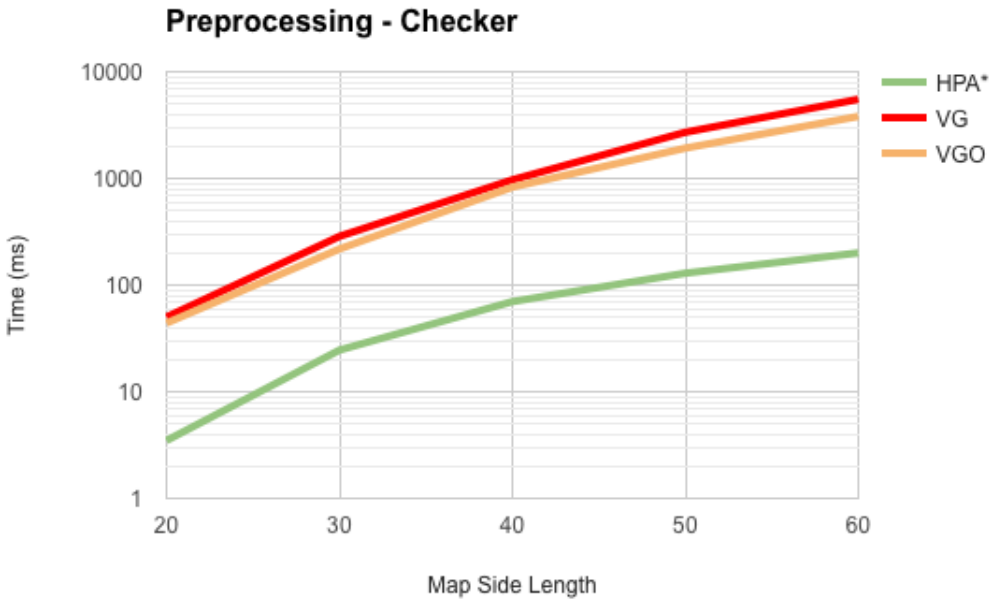


Figure 49: Results from preprocessing the Checker map.

Preprocessing this map (See figure 49) shows the same trend as was seen with pathfinding, namely that the polygonal algorithms have a hard time. In the Lines map the preprocessing time for VG and VGO was significantly faster than HPA*, however in this map HPA* is significantly faster than VG and VGO.

6.4 THE MAZE MAP

A maze is a classic example of a world in which pathfinding is applied. The maze are in some way a mix between the Lines map and the Checker map. It has narrow passages as was seen in the Checker map, and a lot of dead ends as was seen in the Lines map. Having said that, the number of protruding corners is reduced compared to the Checker map. Therefore, the polygonal algorithms should perform better than they did in the Checker map. Since there are many

dead ends we expect Dijkstra to do relatively well as the heuristics are of less importance here.

We used an online maze generator to create five mazes for our experiments, with side lengths 20, 40, 60, 80, 100, 120, 140, 160, 180 and 200 pixels (See figure 50). All ten mazes were created with a corridor width of one pixel. For each map size polygonal counterparts were created (See figure 51).

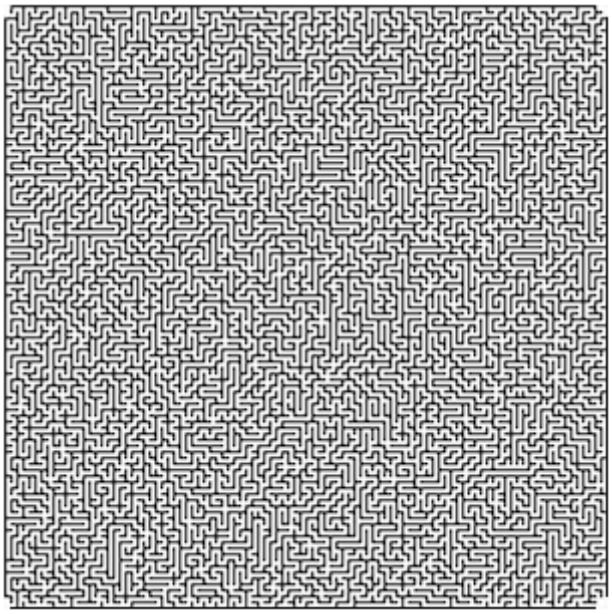


Figure 50: The biggest maze (200x200 pixels) used in our experiments.

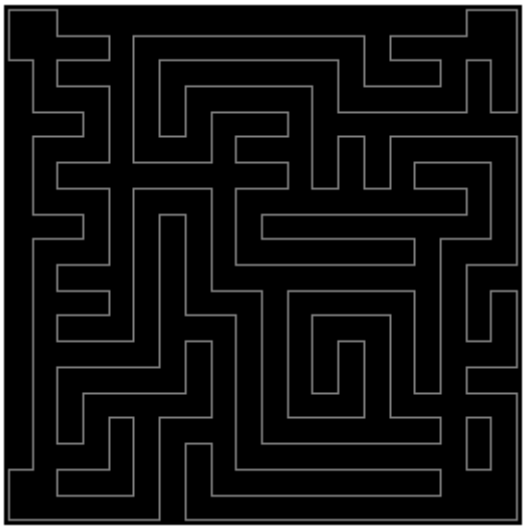


Figure 51: The smallest maze as it looks when converted into a polygon.

Each algorithm had to find a path from each corner of the maze to the opposite corner. We then used the average time it took to find these paths to compare the algorithms. The results for pathfinding can be seen in figure 52.

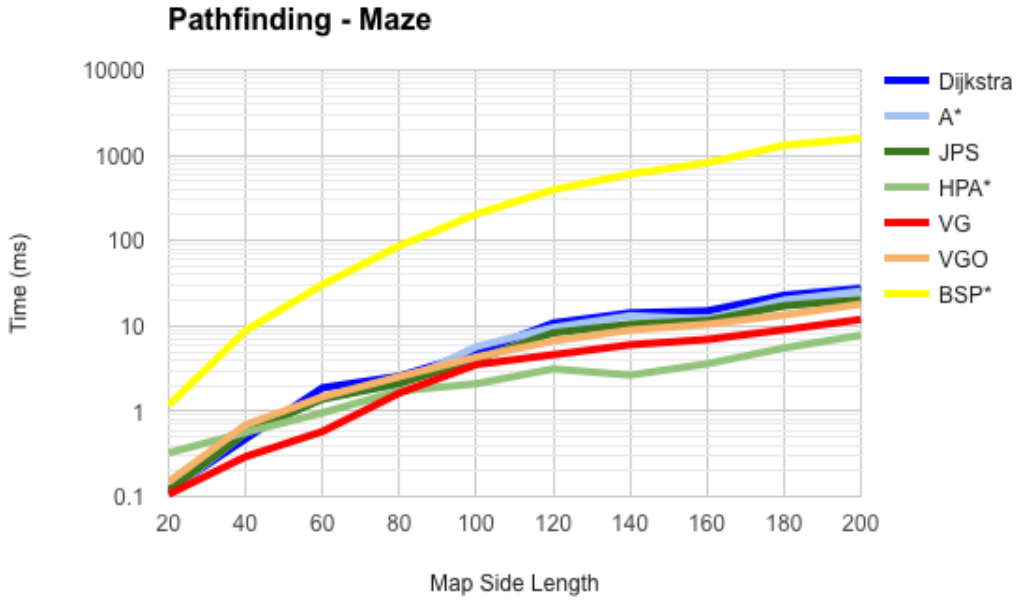


Figure 52: Results for pathfinding in the Maze map.

For this map we notice how BSP* is much slower than the other algorithms. The other algorithms lie relatively close to each other. However, as the map size grows HPA* seems to be the best. As op-

posed to the Checker map this map have dead ends. This means, HPA* can benefit from first finding a path in an abstract map, and then use this to avoid the dead ends in the second pass. The other grid algorithms are given good circumstances in this map, as the corridor width is only 1 pixel, therefore it is surprising that they are not faster than VG or VGO. In figure 53 the results for preprocessing are presented.

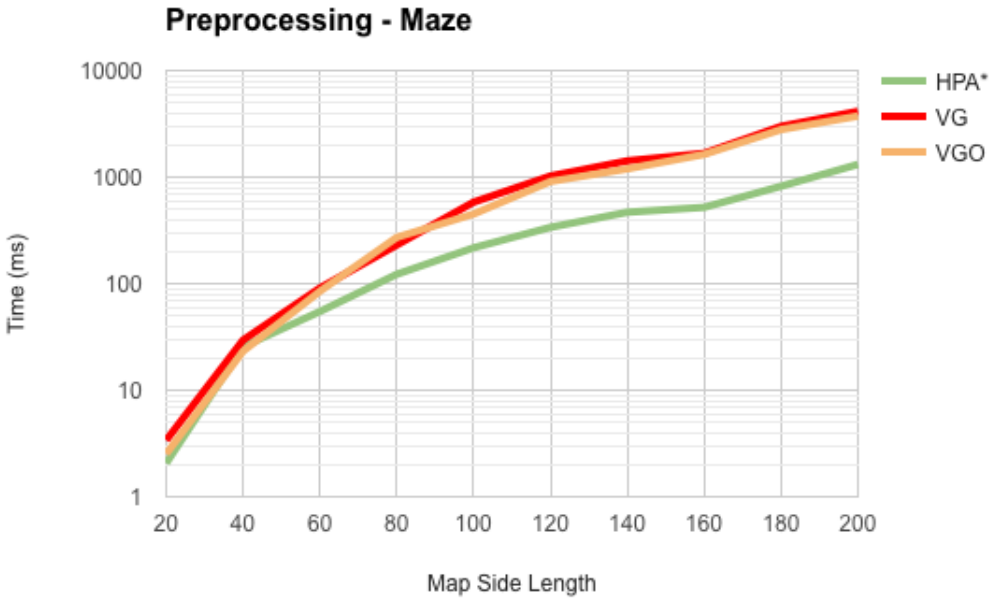


Figure 53: Results for preprocessing the Maze map.

From the results of preprocessing we can see that for a 200x200 map it takes between 1 and 2 second for HPA* to preprocess the map, while it takes about 4 seconds for VG and VGO. The preprocessing time is well spent, since these three algorithms also produced the fastest pathfinding times. This means if preprocessing is allowed it is best to use HPA* as it is the fastest for both preprocessing and pathfinding. If the map is dynamic (i.e. if preprocessing is not allowed) it is best to choose one of the grid algorithms. Here JPS seems to be a little faster than the two other.

6.5 THE SCALED MAZE MAP

With this map we wanted to test what happens when we increase the grid cell resolution in a maze, while leaving the polygonal map unchanged. From this we wanted to learn two things. First of all we wanted to see how the grid algorithms perform among each other in mazes as the corridors become wider and wider. Secondly, we wanted to see for what grid cell resolution the grid algorithms become worse than the polygonal algorithms. We tested mazes with

corridors being 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 grid cells wide (see figure 54). The results for pathfinding can be seen in figure 55.

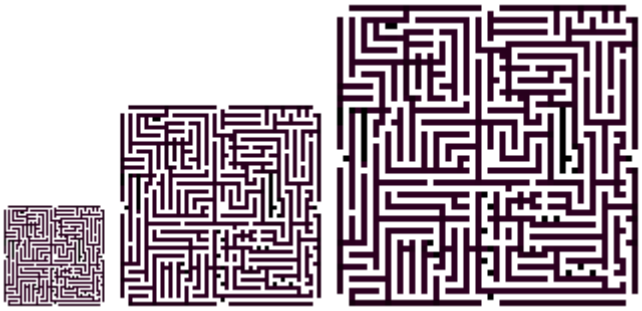


Figure 54: The same maze with three different corridor widths.

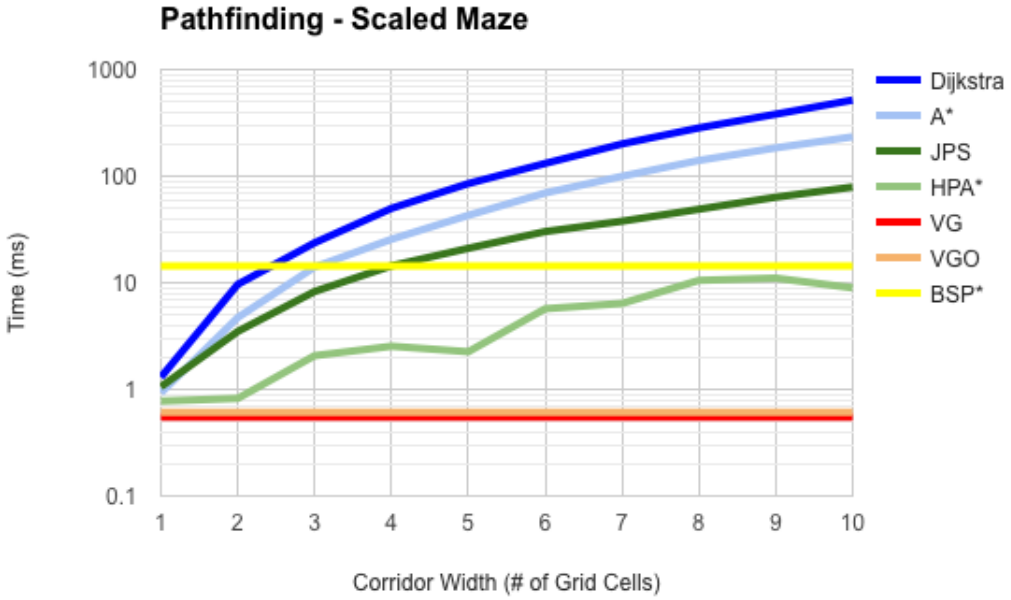


Figure 55: Results for pathfinding in the Scaled Maze map.

This map exemplifies the importance of having a low grid cell resolution. For pathfinding the grid world algorithms become slower and slower as the corridor width increase. The fastest grid algorithm is HPA*, however this algorithm also preprocess the map. If we only consider the three dynamic grid algorithms, we see that they follow the same trend with different speeds. We can see that JPS becomes increasingly faster as compared to Dijkstra and A*, as the corridor width increases.

If we look at all the algorithms, we can see that the fastest algorithm is VG and VGO. These two algorithms have the benefit of preprocess-

ing the map first, however even for the lowest grid cell resolution, they still perform better than the best grid algorithms.

For the lowest grid cell resolution we see BSP* being significantly worse than all the other algorithms. If we compare BSP* to the other dynamic algorithms, Dijkstra, A* and JPS, we must keep in mind, that with a corridor width of one, these algorithms are given optimal conditions. When the grid cell resolution is increased we see that BSP* becomes faster than the other dynamic algorithms. In fact when the corridor width is larger than 4 grid cells, the BSP* algorithm is at least as fast as the other dynamic algorithms.

When all corridors in a maze have the same width, the map can be represented in the grid world with an optimal grid cell resolution. However, some maps have a varying degree of detail, which makes it impossible to lower the grid cell resolution beyond a certain point without sacrificing important detail (See figure 56).

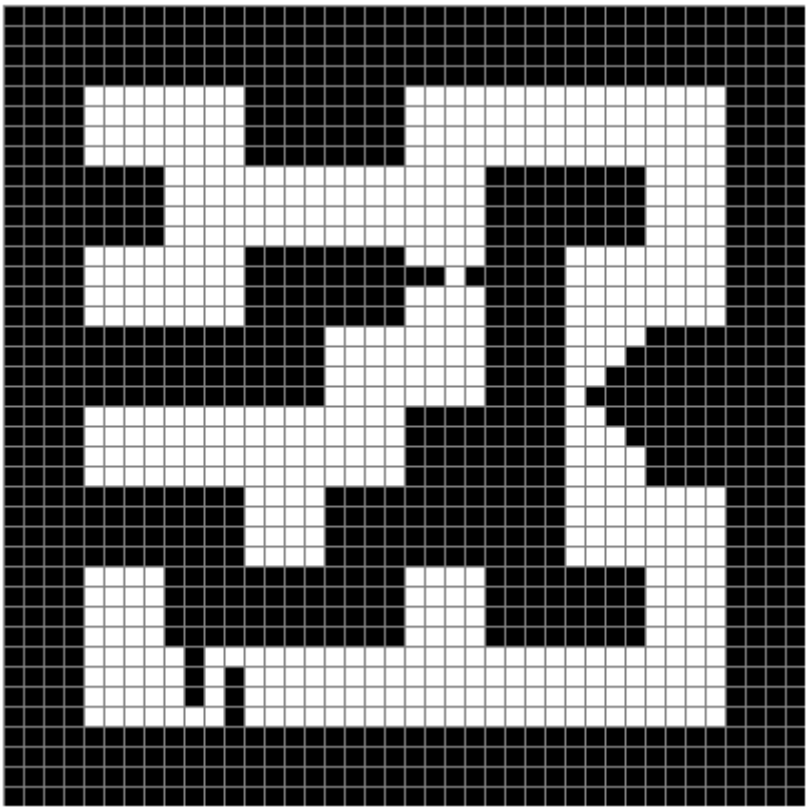


Figure 56: An example of a map with a varying level of detail. Most of the corridors are of the same width, however the grid cell resolution cannot be lowered without sacrificing detail.

This shows that the choice of algorithm, is dependent on the optimality of the grid cell resolution. The results produced by this map shows that BSP* is the preferred dynamic algorithm if the corridor width is greater than 4 grid cells. However, what remains to be in-

vestigated is whether this number stays the same, if we increase the number of corners. We investigate this in the next map, but first we present our results for preprocessing this map (See figure 57).

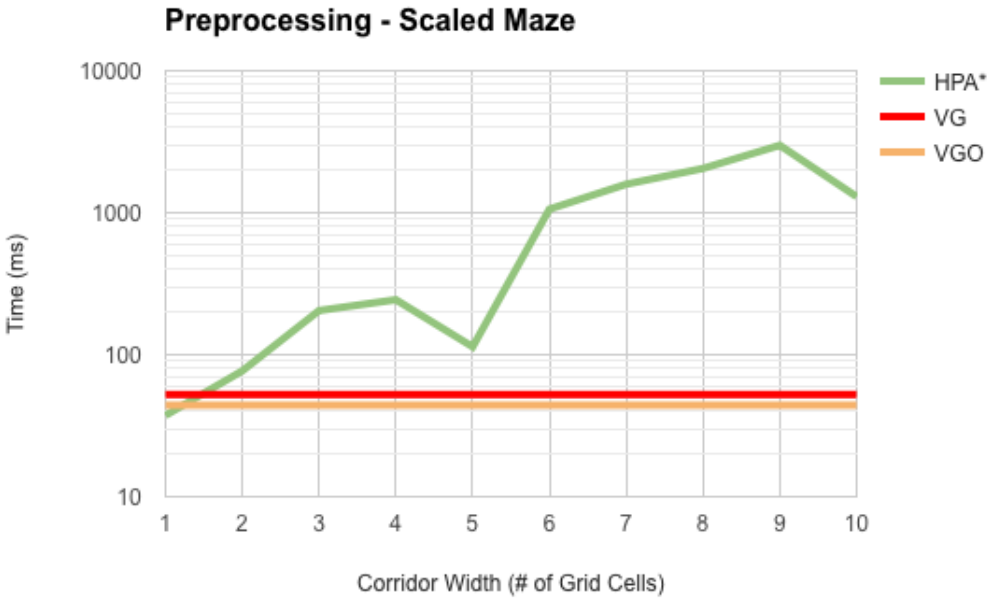


Figure 57: Results for preprocessing the Scaled Maze map.

For HPA* we notice in general a growing trend. This is no surprise as the number of grid cells in the map increase. For corridor widths 5 and 10, we see sudden drops in the time measurements. This can be explained by the fact that 5 and 10 nicely divides the cluster size which is 10. A fitting cluster size may lead to a smaller amount of inter-edges. These drops in the curve are also seen in the results for pathfinding for HPA*. This suggests that it is of importance to pick a cluster size that fits the map, and maybe even base the cluster size on the average corridor width.

6.6 THE MAZECOR5 MAP

In the Scaled Maze map we saw that BSP* was at least as fast as JPS for a corridor width of 4. We also saw that if the corridor width increased beyond 4, BSP* was consistently faster than JPS. In the MazeCor5 map the corridors are 5 grid cells wide for all map sizes. The grid cell resolution is therefore set to a level which should make BSP* the fastest for the smaller maps. We want to see if this is still the case as we increase the map size.

In the Scaled Maze map the number of corners in the polygonal map was unchanged, while at the same time the number of grid cells in the map increased. In the MazeCor5 map the number of grid cells in the grid map increases, but the number of corners in the

polygonal map also increases. Since we only compare BSP* with JPS, there are no data for preprocessing this map. In figure 58 the results for pathfinding is seen.

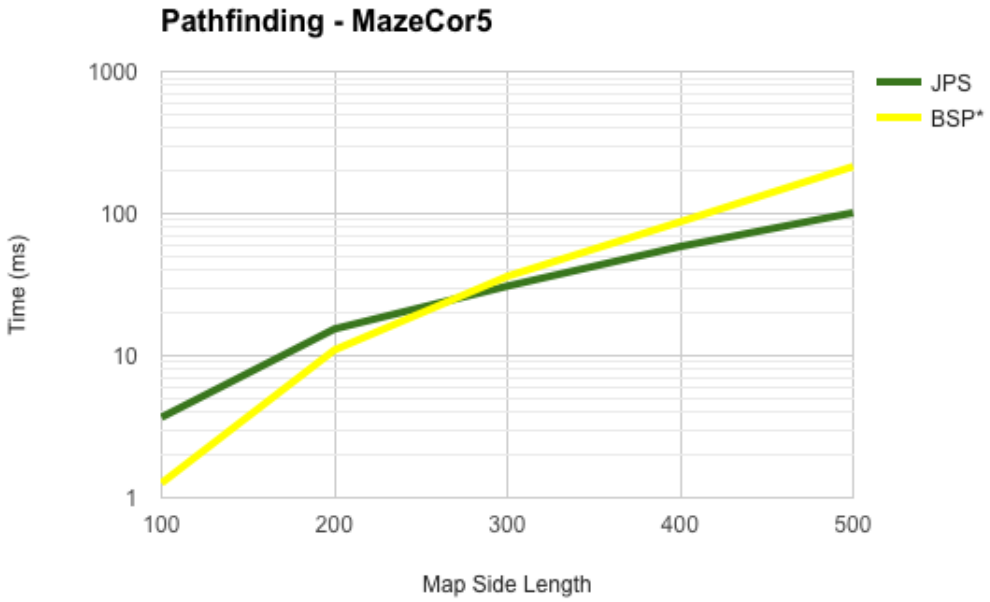


Figure 58: Results for pathfinding in MazeCor5.

From figure 58 we can see that JPS gradually becomes faster than BSP*. BSP* is the fastest for the smaller maps, but JPS is the faster when the map size increase. As we increase the map size, the number of grid cells and corners increase. Both of these increase at the same rate, namely linearly in the map size. This implies that the difference in growth rates of the running times, must come from the algorithms themselves. This is in agreement with our theoretical expected running times.

Part IV

CLOSURE

7

DISCUSSION

The purpose of this research was to get an overview of the available pathfinding algorithms. We implemented and tested seven different algorithms using a range of map types with different properties. In this section we discuss the results to give the reader a clear idea of when to use which algorithm. The results of our experiments combined with some theoretical observations will serve as basis for our discussion.

Dynamic Grid Algorithms

The first three algorithms we discuss are Dijkstra, A* and JPS. It makes sense to group these together as they all work on grid maps and none of them rely on preprocessing. They all have the same worst case running times, namely $O(N \log N)$, where N is the number of traversable grid cells. The results show us a rather clear trend between the three algorithms. JPS is always at least as good as A*, and A* is always at least as good as Dijkstra. JPS therefore seems to be preferable over A* and Dijkstra. JPS is especially good if the map contains large open areas with lots of traversable grid cells such as the Lines maps. JPS usually only explores a fraction of the nodes explored by A* or Dijkstra, which means that JPS also needs to allocate considerably less memory.

On the Maze maps we saw no real difference between the three. This map is similar to the Lines map, in the sense that there are many dead ends. In the Lines map A* and Dijkstra were also similar in running time. This means, if the map contains many dead ends the negative sides of Dijkstra becomes less significant. The Lines map did however as mentioned contain large open areas, which made JPS come out ahead.

Algorithms that Rely on Preprocessing

Next we discuss the algorithms which rely on preprocessing; HPA*, VG and VGO. These were the fastest of all the pathfinding algorithms, but did as mentioned rely on preprocessing. Reliance on preprocessing is a twofold problem. The first is that the algorithms are not applicable if the map is dynamic. The second is that the work done

by the preprocessing has to be permanently stored, which means the memory usage is increased. If none of this is of concern preprocessing allows for very fast pathfinding as compared to the dynamic algorithms. The algorithms that rely on preprocessing were for all maps the fastest. For the Star maps and the Lines maps they were about 100 times faster than the other algorithms.

When deciding on whether to use HPA*, VG or VGO you are also deciding on whether to use the grid representation or the polygonal representation. If you choose to go with HPA* you are stuck with the limitations of the grid world. These include a suboptimal path length and the problem of choosing a fitting grid resolution. A fitting cluster size is also of importance for HPA* as we saw in the Scaled Maze map. When it comes to pathfinding we saw that VG and VGO performed at least as good as HPA*, except in extreme cases such as the Checker map. Together with the limitations of the grid world, this could suggest that VG and VGO is preferred over HPA*.

When comparing pathfinding times of only VG and VGO the results were kind of surprising. We expected VGO to be considerably faster than VG, but most of the time they were equally fast and in some cases the VG algorithm was even faster. The visibility graphs created by VGO were considerably smaller, than those created by VG. A smaller visibility graph will always lead to an increase in speed. However, we also used Property Two to reduce the number of successor points at runtime. We can only imagine that this extra calculation yielded no real benefit, and instead only added to the running time. This could explain why VGO did not perform better than VG, even though its visibility graph is much smaller. We did not get to test this hypothesis explicitly.

The main reason for measuring the preprocessing times, was to see how VG, VGO and HPA* would perform in dynamic maps. As it turned out, they were all too slow to be used on dynamic maps. A more interesting thing to compare, would probably have been their memory usage.

*BSP**

Finally we have our own algorithm, BSP*. This algorithm is meant for use in dynamic polygonal maps, and is therefore not reliant on preprocessing. This means that changes can be made to the polygonal world representation without slowing down pathfinding. This flexibility however comes at the cost of pathfinding time, but if the maps are not too large the algorithm does have acceptable running times. The most problematic map for BSP* was the maze map. For the biggest maze map the running time was just below 2 seconds per pathfinding query making it approximately 100 times slower than all the other algorithms. Therefore if you want to build a dynamic maze game, it is probably better to make it grid based. In other map types

BSP* is approximately as fast as the other dynamic algorithms. In the Star map, BSP* was almost as good as JPS and A*, and in the Lines map it was even better than the two. Therefore to decide which algorithm to use one has to consider both map type and size.

In our research we have not come across any other optimal dynamic pathfinding algorithms for polygonal worlds. If you want to stay with a polygonal world representation the only optimal alternatives to our knowledge are VG or VGO. If these are chosen for dynamic maps you might have to preprocess the map once for each pathfinding query in the worst case. This makes pathfinding very slow. If the start and goal positions are close to each other, creating a full visibility graph is a waste of time. In such a situation BSP* only computes what is necessary, which is a clear advantage for dynamic maps.

8

CONCLUSION

In this report we set out to investigate pathfinding algorithms. We wanted to do this through the eyes of a video game developer. We therefore chose two commonly used 2D world representation, namely the 8-connected grid world and the polygonal world. In these worlds we tested different algorithms including our own algorithm, BSP*. We also found a simple way of reducing the visibility graph to the absolute minimum of what is required for pathfinding.

We tested a wide variety of algorithms, which all had strengths and weaknesses. From this we can conclude that when deciding on a pathfinding algorithms one must first decide on what to prioritize. If path optimality is of importance, you might want to go with a polygonal algorithm. If the map is static, you might want to go with an algorithm that takes advantage of preprocessing. If the map has a compact maze structure, you might prefer a grid based algorithm. If the map is polygonal and dynamic your best choice might be BSP*.

We set out to get an overview of current pathfinding algorithms, which we feel was accomplished. On top of that we developed a new algorithm, which proved to be useful for specific scenarios.

9

APPENDIX

9.1 PSEUDOCODE FOR DIJKSTRA'S ALGORITHM

Algorithm 1 Dijkstra's Algorithm

```
1:  procedure DIJKSTRA(graph, start, goal)
2:      explored ← empty set
3:      frontier ← empty set
4:      start.predecessor ← null
5:      start.G ← 0
6:      frontier.add(start)
7:  repeat:
8:      if frontier set is empty then return failure
9:      u ← node in frontier with lowest G value
10:     frontier.remove(u)
11:     explored.add(u)
12:     if u = goal then break
13:     for each successor s of u do
14:         if s is in explored then continue
15:         G ← u.G + weight(u, s)
16:         if s is not in frontier then
17:             s.G ← G
18:             frontier.add(s)
19:             s.predecessor ← u
20:         else if  G < s.G  then
21:             s.G ← G
22:             s.predecessor ← u
23: end repeat
24:     path ← empty list
25:     n ← goal
26:     while n is not null do
27:         path.add(n)
28:         n ← n.predecessor
29:     path.reverse()
30:     return path
```

9.2 THE GRIDTOPOLY ALGORITHM

Here we describe in general terms how our so called GridToPoly algorithm works. This algorithm was developed specifically for comparing grid algorithms with polygonal algorithms. It is able to read a grid map and convert it into a corresponding polygonal map.

The algorithm consists of two phases. In the first phase the outermost polygon encompassing the entire map is found. In the second phase we scan the inner parts of the map to find the rest of the polygons.

We start by finding a point on the outer polygon. This is done by searching the grid map for the bottom left corner. We then move along the edge of the traversable area one grid cell at a time. If the grid structure makes us turn right or left, we add a point to the polygon. When we arrive, at the grid cell from which we started, the outer polygon is done. During this first phase we mark all obstacle grid cells outside the polygon.

In the next phase we search for unmarked obstacles inside the outer polygon. This is done by scanning the grid map, and each time we meet an unmarked obstacle grid cell, we trace the surrounding polygon. This tracing is done in the same way as we did in phase one. Each time a new polygon has been created we mark the obstacles inside of it as visited. When all obstacles have been visited we are done.

10

BIBLIOGRAPHY

[1] Cormen, Leiserson, Rivest, Stein; *Introduction to Algorithms*; MIT Press and McGraw-Hill; 2nd edition; 2001.

[2] Wikipedia: Article on Dijkstras algorithm.

[3] Fredman, Tarjan; *Fibonacci heaps and their uses in improved network optimization algorithms*; 25th Annual Symposium on Foundations of Computer Science, IEEE; 1984.

[4] Hart, Nilsson, Raphael; *A Formal Basis for the Heuristic Determination of Minimum Cost Path*; IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2); 1968.

[5] Wikipedia: Article on the A* algorithm.

[6] Harabor, Grastien; *Online Graph Pruning for Pathfinding on Grid Maps*; 25th National Conference on Artificial Intelligence. AAAI; 2011.

[7] Botea, Muller, Schaeffer; *Near Optimal Hierarchical Path-Finding*; Department of Computing Science, University of Alberta; 2004.

[8] Berg, Cheong, Kreveld, Overmars; Computational Geometry, Algorithms and Applications; Springer; 3rd edition.