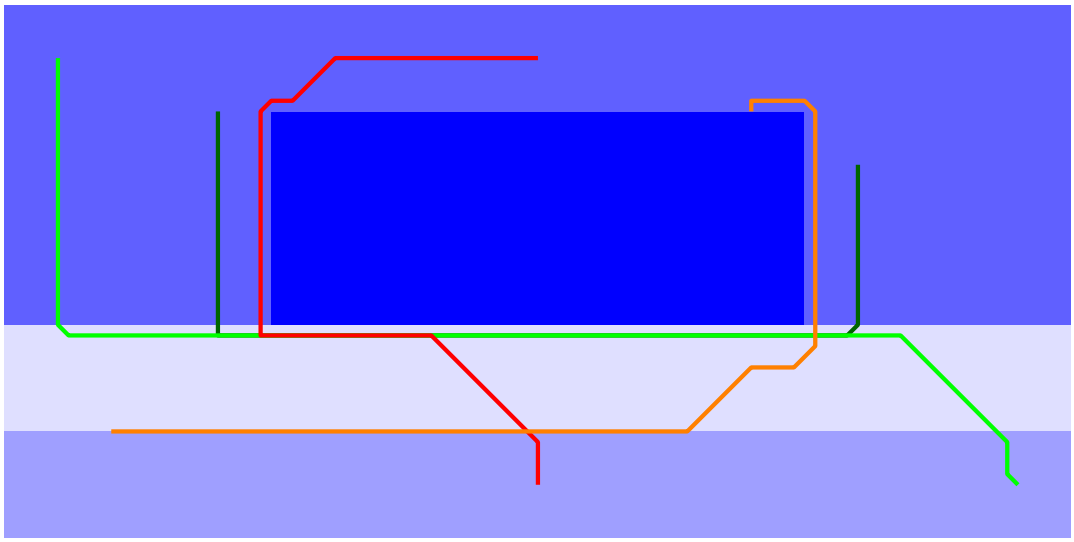
Mark Tibboel

Creating high quality indicative routes in heterogeneous virtual environments



MSc Thesis (ICA-0428701)

Game and Media Technology
Utrecht University, The Netherlands

Supervisors: dr. R.J. Geraerts and MSc N.S. Jaklin

August 21, 2013

Abstract

In this thesis we will attempt to close the gap between fast grid-based method and ϵ-approximation methods that solve the Weighted Region problem. In order to compare both methods, we will define a formula to calculate the ϵ-value of grid-based methods and define the bounds of the ϵ-value for ϵ-approximation methods. Furthermore, we will improve on heuristics for A* grid methods for weighted regions and present an adapted version of the Hierarchical A* method that improves the running times of A* grid in heterogeneous environments. Finally, we present three pruned graph methods that reduce the construction and query times of Steiner-graph methods.
Acknowledgements. This research has been supported by the COMMIT/ project (http://www.commit-nl.nl/).

Contents

1 Introduction

Path planning is an important aspect of computer games, crowd simulations and robotics. The topic has been studied for several decades, and a wide variety of subproblems have been tackled. Among those are problems such as planning smooth and visually convincing paths for humanoid characters, as described in the IRM [1] and MIRAN [2] methods, and adding weights to the environment [3, 4, 5, 6, 7]. The IRM and MIRAN methods require an indicative route as input that serves as a global route for the character. The indicative route is used as a guide to plan the final motion of the character. The final motion will deviate from the indicative route to ensure smooth curves, keep preferred clearance from obstacles, avoid unnecessary detours, and avoid other characters. An environment can be split up in regions and weights can be assigned to each region to model the costs of travel through a region or to model psychological factors in a dynamic way. Throughout this work, we call an environment heterogeneous, if each region is annotated with a weight. Virtual environments are path planning environments that are time critical. Time critical applications such as games, crowd simulators or safety training software only allow path planning methods to take up several milliseconds, because paths may need to be planned for many characters each time step.

Adding weights to a scene has many applications. For instance, giving a sidewalk a lower weight than a road will let pedestrians characters prefer to traverse the sidewalk instead of the road. However, if the sidewalk is blocked by an obstacle such as a parked car, then pedestrians may use the road in order to circumnavigate the obstacle. Rule-based approaches [8] that restrict pedestrians to walk on the sidewalk do not yield the desired behavior in such a case. This may lead to unrealistic deadlock situations.

However, there is a price to pay. The shortest path in an environment with uniform weights and obstacles always consists of a series of straight lines that use the obstacle vertices, and thus bending points of the shortest path are always obstacle vertices. The number of obstacle vertices is finite, and a graph can be constructed containing all collision-free edges between obstacle vertices. This Visibility graph [9] can then be queried to find shortest paths. In weighted regions, the bending point of an optimal path can lie anywhere on the edge of a region. Each edge consists of infinitely many points and a graph containing all possible paths will be of infinite size. As such, finding optimal paths in a weighted region is an unsolvable problem [10] due to the infinite size of the search space. Although it is impossible to find an optimal solution, approximate solutions can be found.

Researchers have been studying the problem of finding optimal paths in heterogeneous environments for more than 20 years. The problem is formally known as the Weighted Region Problem (WRP) as defined by Mitchell and Papadimitriou [3]. They state the WRP as follows: Given a straight-line planar (polygonal) subdivision on which a weighted Euclidean metric is defined, and a start and goal position, find a minimal length (in the weighted sense) path from the start to the goal. The length in the weighted sense is mostly referred to as the costs of the path. These costs are defined by the sum of all subpaths, where each subpath lies in a different weighted region. The costs of a subpath is defined by the length of the path times the weight of the region.

1.1 Project motivation

The exact unsolvability of the Weighted Region Problem created two types of solutions to the problem. The first solution takes the practical approach and focuses on fast algorithms. The resulting paths of these fast algorithms are nowhere near-optimal. This solution is most used in practice. On the other side there are solutions that focus on the optimality of the paths and create an approximation to the optimal solution. These methods create suboptimal paths within a certain bound, but they are slow. There is a huge gap between these two approaches, and currently there are no methods that provide suboptimal paths that can be used in real-time applications.

1.2 Project goals

This master thesis will focus on finding a method that is able to find near-optimal paths in real-time. First, we will identify the upper bound on the overestimation of the costs of the shortest path computed by the practical solutions that are currently used to solve the WRP in real-time. These bounds are expressed with an ϵ-value. An ϵ-optimal method will always compute paths that cost at most $1 + \epsilon$ times the costs of the optimal path. Once the ϵ-value is known for the practical solutions, we can identify the size of the gap between near-optimal methods and fast methods based on the difference in ϵ-optimality. Once the size of the gap is known, we will try to close the gap from both sides. We will attempt to improve the running times of the near-optimal methods and lower the ϵ-optimality of the fast methods. Finally, we will create a hybrid method that combines the fast and near-optimal methods to create a fast near-optimal method.

1.3 Report structure

Section 2 gives an overview of related work on path planning and the Weighted Region problem (WRP). The work of different scientific fields that work on the WRP is discussed, and a short overview of practical approaches is given.
In Section 3, we will discuss how the solutions given by grid-based approaches compare to an optimal solution in terms of path-costs. By defining the ϵ-value of grid-based approaches, we can compare grid-based approaches with ϵ-optimal methods. We will also look at the heuristics used by A* and try to improve these to better suit weighted grids.
The ϵ-approximation method as defined by Aleksandrov, Lanthier and Maheshwari [5] is described in Section 4 along with a graph search method specially designed for the WRP called BUSHWHACK [11]. We will also look at the bounds of the ϵ-value. The ϵ-value is a parameter for ϵ-approximation methods. To compare the running times of grid-based methods with the ϵ-approximation methods, we will have to make the ϵ-values of both methods compatible.
In Section 5 we present a new method that combines A* grid methods with ϵ-approximation methods. We show how A* grid results can be used to prune the search space for ϵ-approximation methods.
The implementation notes are presented in Section 6. This section gives a small overview of the implementation of the methods presented in this thesis and presents an adapted version of the A* algorithm that supports multi-layered environments. Furthermore, it describes several issues that occurred during the implementation of the BUSHWHACK method.
Section 7 describes the experiments that are conducted. First, we will find the optimal values of the two Hierarchical A* parameters: abstraction factor and number of layers. Then we will compare the different heuristics for A* to see which heuristic is well suited for weighted regions. Several newly presented hybrid methods are compared against each other and to the ϵ-approximation methods to see whether these new methods are faster, and if the ϵ is retained. Finally, we will compare the grid-based methods to the ϵ-approximation methods and hybrid methods to see the size of the gap between the grid-based and ϵ-approximation methods and to see whether the hybrid methods can narrow down this gap.
In Section 8, we will draw conclusions about our algorithms and experiments. We will show that the Hierarchical A* method lowers the running times of grid-based approaches and that the pruned-graph methods can improve on both the construction times and query times of current graph-approaches.
Finally, in Section 9 we will discuss how the algorithms and methods presented in this thesis might be further improved, and we will present several research questions that arose from the results presented in this thesis.

2 Related work

In this section, we will dive into the different methods that solve path planning problems in homogeneous regions and in weighted regions. We will look at existing techniques and discuss open research questions. First we will look at path planning in homogeneous regions and discuss methods such as graph-based methods, roadmaps, potential fields and visibility graphs. We will also look at extensions of the path planning problem in homogeneous regions by looking at planning optimal paths for disk-shaped characters with a variable radius and creating smooth and visually convincing paths. Next, we will look at the Weighted Region problem (WRP) and discuss wavefront propagation, Pathnet, Steiner points and BUSHWHACK. Research on the WRP has mainly been carried out by the computational geometry and path planning communities. In this review, we will also discuss other fields that work on the WRP, but in a different setting. Finally, we will look into practical solutions to tackle the weighted region problem.

2.1 Path planning

Path planning originated from the field of robotics and is nowadays applicable to many other fields, such as crowd simulation or video games. Path planning is originally defined as finding a collision-free path from a start to a goal position in an environment with obstacles and is a NP-hard problem [12], except for some simple instances. We will be focusing on finding shortest collision-free paths. Several methods have been devised to tackle this problem and in this section we will discuss the headlines of all these different methods along with their pro's and con's. Path planning always starts with a start position, goal position and a scene with obstacles. In all methods, the scene is first converted into a data structure that represents the scene and allows fast search methods on it.

2.1.1 Graph methods

Several methods use a grid that can serve as a graph [12, 13, 14]. Existing graph search algorithms can then be used to find a shortest path. Rectilinear grids allow characters to move between neighboring cells in 4 or 8 directions. Two cells are neighbours if the edges of both cells share at least one vertex. The grid can be seen as a graph where the center point of every cell corresponds to a node. An edge in the graph connects any two nodes that correspond to neighboring cells in the grid. A node can have two states: free or obstacle. The size of the grid can be adjusted to alter the precision of the resulting paths. A surface can be supersampled by using a larger grid (i.e. by using smaller cells). This will result in a bigger graph and slower searches, but the resulting paths will be a better approximation of the shortest path. In contrast, we can undersample the surface by taking a smaller grid. This will speed up the search, but the resulting path will generally deviate more from the shortest path.

Basis algorithm The following methods that search a graph for a path from the start to the goal node all use the same basic algorithm (see Algorithm 1). The difference between the methods lies within the selection of the next node that should be discovered. All methods start by dividing the nodes of the graph in three groups. The first group, *closed*, consists of nodes that have been discovered and do not need further exploration. The second group, *unvisited*, consists of all nodes that haven't been visited by the algorithm. The third group, *open*, consists of nodes that are still open for exploration. The algorithm starts with the start node in the open set and all other nodes in the unvisited set. With each step of the algorithm, a node N_c is selected from the open set. This selection step is different for each of the algorithms discussed below. After a new node N_c has been selected, it will be moved to the closed group and all neighbours of N_c that are in the unvisited group will be moved to the open set and have their parent pointers set to N_c. These steps are repeated until the selected node is the goal node. The resulting path is reconstructed with *parent* pointers. These pointers are an attribute of

each node. When new nodes are added to the open set, then the node from which they were discovered will be set as their parent. The path can be found by following parent pointers from the goal node to the start node. This path is reversed such that the start position is at the beginning.

Algorithm 1 Basis algorithm(G, *start*, *goal*)

```
Add start to open
Add rest to unvisited
N_c = start
while N_c ≠ goal do
    Select new N_c from open
    Move N_c from open to closed
    Move all neighbours of N_c from unvisited to open
    Set the parent pointers of these neighbours to N_c
end while
Reconstruct the path from the parent pointers
Return Path
```

Breadth first search Breadth first search [12] on a graph works the same as breadth first search on a tree. For each node that is being explored, all neighbours are added to the open set before the next node will be explored. This is implemented through the use of a FIFO queue. Once a node is selected from the open set, all neighbours will be moved from the unvisited set to the open set and the selected node is moved from the open to the closed set. This will lead to a search that will expand slowly from the start node, but it will expand in all directions. Breadth first search is a brute force approach. All nodes will be examined until a path has been found. Breadth first search runs in $O(n)$, where n is the number of nodes in the graph.

Depth first search Depth first search [12] employs the opposite exploration technique of breadth first search. Instead of exploring all neighbours gradually, depth first search fully explores a single path and will then start looking at the neighbours starting from the end of the path until the beginning. The algorithm is exactly the same as the breadth first search, but the FIFO queue is replaced by a LIFO queue. This queue will always return the last node that was inserted and leads to a search that expands rapidly in one direction. Just like breadth first search, is depth first search another brute force approach to path planning. Depth first search runs in $O(n)$, where n is the number of nodes in the graph.

Best first search Best first search [12] is the first algorithm that assigns costs to the nodes in the open set. The costs of a node are defined by the distance from the node to the goal node. Best first search uses a priority queue as open set. A priority queue supports $O(\log n)$ time insertion and $O(1)$ time retrieval of the node with the lowest costs. Best first search is often a fast algorithm because it takes a straight line to the goal position. However, when obstacles are between the start and goal position the resulting path could be far from optimal. Best first search runs in $O(n \log n)$, where n is the number of nodes in the graph.

Dijkstra Dijkstra's algorithm was founded by the Dutch computer scientist Edsger Dijkstra in 1959 [13]. Dijkstra's algorithm starts with setting a distance value for all nodes to infinity, except for the start node. The start node will have a distance value of zero and will be added to the open set. Dijkstra's algorithm uses a priority queue to find the node with the lowest distance value at each iteration. Once a node has been selected, the distance value for all nodes connected to this node are

calculated by taking the edge costs plus the distance value of the current node. If a node already has a distance value, then it will only be overwritten if the new value is lower. The strength of Dijkstra's algorithm is that it will always find the shortest path in the graph, if it exists. Dijkstra's algorithm runs in $O(E+V \log V)$ if its priority queue is implemented by a Fibonacci heap, where E is the number of edges and V is the number of vertices in the graph.

A* A* [14] is a generalized version of the Best first search and Dijkstra's algorithm. During the execution of the algorithm, each node in the open set will have four attributes. The first attribute g represents the cost to travel to this node from the start node. The second attribute h represents an estimate of the cost to travel from this node to the goal node. The third attribute f is the sum of g and h. The final attribute is the *parent* pointer. The algorithm starts at the start position, so g can always be calculated by looking at the g-value of the parent and adding the travel cost between the node and his parent. We could calculate h and have a perfect estimation of the costs. This will make the A* algorithm only explore nodes that are part of the optimal path. However, calculating the optimal path to the goal for each node is not efficient. Instead, the algorithm uses a heuristic to approximate the distance from the node to the goal. A* starts from the start position and looks at all neighbouring nodes. All neighbouring nodes are added to the open set and their attributes are set. Again a priority queue is used for the open set. In each iteration the node with the lowest f value will be removed from the open set. The A* algorithm is a commonly used search algorithm because of its flexibility. By using an admissible heuristic (see below), the time for computing a shortest path will always be at most the time that Dijkstra's algorithm requires to find the shortest path. The complexity of the A* algorithm depends on the heuristic. The worst case running time for the A* algorithm is $O(2^l)$, where l is the length of the shortest path. With an admissible heuristic the complexity of A* is equal to the complexity of Dijkstra's algorithm: $O(E+V \log V)$ if its priority queue is implemented by a Fibonacci heap, where E is the number of edges and V is the number of vertices in the graph.

Heuristic The A* algorithm has no predefined heuristic, and that's what gives the algorithm its flexibility. An important property of a heuristic is whether it is admissible or not. An admissible heuristic never overestimates the costs to the goal. Non-admissible heuristics can yield non-optimal solutions due to the overestimation of the costs to the goal. If h is set to 0 for all nodes, then the A* algorithm will behave just like Dijkstra's algorithm. The Euclidean distance from the current node to the goal position is a heuristic that is mostly used in path planning without weighted regions. The Euclidean distance always equals the actual costs if no obstacles block the straight-line connection between the current node and the goal. In a weighted region, the Euclidean distance could underestimate the costs, because the weights are not incorporated.

Iterative Deepening A* (IDA*) The Iterative Deepening A* algorithm [15] decreases the memory usage of the A* algorithm. Iterative Deepening is a method that can be applied to all graph search algorithms. At each iteration of the A* algorithm, a depth first search is executed that is cut off when a selected node exceeds a threshold. The initial threshold value is the f value of the start position. After the first step, the threshold value is equal to the lowest value of all depth first searches that exceeded the threshold. The threshold will be increased at each step until the goal is reached. IDA* has little computational overhead, because at each step it will explore the same nodes in the beginning that were already within the previous threshold. However, IDA* has a strongly decreased memory usage, because there is no open list to maintain. The depth first searches already require little memory and they are discarded when the threshold is increased.

Hierarchical A* Hierarchical A* [16] attempts to speed up the search through the use of better heuristic values. Such values can guide the search better and let the A* algorithm visit less nodes.

Several abstraction layers are created from the scene. These layers are used to calculate the heuristic value for all nodes. The abstraction layers are created by taking the nodes with the highest number of free nodes around it and grouping them with all nodes around it within a certain distance until all nodes are grouped. This process is repeated until there is a top layer with just a single node. When a node is added to the open list, then its heuristic value needs to be calculated. The node will look at the layer above him for its heuristic value. Each abstraction layer will query its parent layer until the top layer is reached. This will result in abstract path from the node to the goal. The exact length of this path is known and will be used as h value for the node. In the paper, the authors compare Hierarchical A* with Dijkstra's algorithm. A comparison with other heuristics is not made.

Hierarchical Pathfinding A* Hierarchical Pathfinding A* [17] uses abstraction layers for the search itself. The search will be faster than A*, but the path is near optimal. The methods starts with an offline step where the grid is preprocessed into several tiles. For instance a grid of 40x40 cells could be split up into 16 tiles of 10x10. Entrances are defined for each tile. An entrance is a segment of the border that has adjacent free cells in other tiles. These segments are shrunk to a single point in the middle for small entrances or two points at the edges for large entrances. The entrance points are connected within the tile with *intra-edges* and to other tiles entrance points through *inter-edges*. The Euclidean distance is used as the length of the edges. Obstacles within the tile are ignored. A graph is constructed from these intra- and inter-edges with the assigned lengths. The online search starts by connecting the start and goal positions to the entrances of their tiles. Then A* is used to find a shortest path in the abstraction layer graph. This path is the guideline for the actual path. The actual path is computed using A* within each tile separately when the current path has reached the entrance of a tile. This technique will save many wasted computations when the path is frequently changed. Furthermore, the search is pruned because it will only search in the tiles that lie within the abstract path. In the experiments conducted in the paper, Hierarchical Pathfinding A* performed 10 times faster than A* while finding paths that are within 1% of the optimal path. The memory overhead for the abstraction graph is around 9%. For larger maps, the method could be expanded by using multiple abstraction layers.

2.1.2 Roadmap methods

The previous methods are mainly used in low-dimensional spaces. The upcoming methods are less dependent on the dimensionality of the search space. The probabilistic roadmap [18] approach is a sampling-based method for solving complicated path planning problems. Sampling is used to generate an approximation of the free space of the scene. The method consists of two phases. In the construction phase a graph is constructed that captures the connectivity of the free space. By default the method uses uniform sampling, with additional sampling in difficult areas. Difficult areas are parts where the graph is disconnected, but the free space isn't. The second phase, i.e. the query phase, starts with connecting the start and goal positions to the graph and then uses a local planner to find an optimal path within the roadmap. This method is well-suited for motion planning where the degrees-of-freedom of a robot are taken into account during the planning. Furthermore, once the roadmap is built, it can answer queries efficiently. However, for single-query problems the construction phase is too expensive, the uniform sampling method doesn't always capture the connectivity of the free space and the path quality is highly dependent on the sampling. Gaussian sampling [19] or Medial axis sampling [20] methods can be used instead of uniform sampling to get better results in some cases.

2.1.3 Potential fields

Potential field methods [21, 22] guide a character from the start to the goal position with potentials. There is an attractive potential that pulls the character towards the goal position and a repulsive

potential that pushes the character away from obstacles. At each step, these potentials are combined in the potential function that results in a vectorfield that points in the direction of the next step. A major drawback of the potential field methods is that the search can end up in a local minimum. A local minimum could be avoided [23], but the extra computations required wouldn't allow for real-time performance. However, this method works well with dynamically inserted obstacles because only the potentials of the inserted obstacles need to be added to the previously calculated net potentials.

2.1.4 Visibility graphs

Visibility Graphs are graphs that consists of nodes that represent point locations and edges that represent the visibility between point locations. Each obstacle vertex in the scene is transformed to a node in the visibility graph and edges are added between each pair of obstacle vertices that are mutually visible. Sharir and Shorr [9] created a method that constructs the visibility graph in $O(n^2 \log n)$ and queries it with Dijkstra's algorithm to obtain the shortest path. The paths obtained from the visibility graph are always optimal in length. However, there isn't much room for improvement on the construction time, because the visibility graph may contain $O(n^2)$ edges.

2.1.5 Hershberger and Suri

In 1999, Hershberger and Suri [24] developed an optimal-time algorithm that computes a shortest path between two points in the presence of polygonal obstacles. The algorithm constructs a map that stores the shortest paths from a single source to all other points in $O(n \log n)$ time, where n is the total number of vertices in the obstacle polygons. This map can be queried in $O(\log n)$ time for a single goal position. The map construction uses a wavefront that extends from the source position. The expansion of the wavefront is based on Dijkstra's algorithm. Many wavefront methods have been developed before this method, but those methods had a running time of $O(n^2)$ due to the complexity of the wavefront collision calculations. Hershberger and Suri managed to improve these calculations by approximating the wavelets of the wavefront and by using a quad-tree-like subdivision of the scene.

2.1.6 Extensions to the path planning problem

In addition to the aforementioned approaches, research has been carried out on solving variants of the original path planning problem. There are many extensions to the path planning problem, but we will only discuss a few. First we will discuss the extensions to disk-based path planning and generating smooth and visually convincing paths. In the next section we will look at the weighted region problem.

Extension to disk-shaped characters with variable radius The first extension we will discuss is the extension to disk-based planning. The above methods all plan a path for a character without any dimensions, except for the probabilistic roadmap method [18]. If an actual character will traverse these paths, then he will most likely collide with several obstacles. These methods all plan optimal paths for a disk-shaped character with a variable radius.

- **Voronoi-based algorithms** Voronoi diagrams [25] can be used to create paths with maximal clearance to all polygonal obstacles. These methods create a generalized Voronoi diagram where each obstacle in the scene becomes a site in the Voronoi diagram. The Voronoi diagram consists of points and lines with maximal distance to the surrounding sites, thus path planning on a Voronoi diagram will lead to paths with maximal clearance to the obstacles. A Voronoi diagram of a 2D scene with polygonal obstacles can be constructed in $O(n \log n)$ time.

- **Minkowski Sum methods** Other approaches dilate the obstacles of the scene with a disk with radius equal to the character's size [26, 27]. Each obstacle in the scene is replaced by the

Minkowski Sum of the obstacle and the disk. The resulting scene can be used as input for a point-based path planning algorithm.

- **Explicit Corridors** The introduction of corridors [28] led to new methods for path planning for disk-shaped characters. Corridors are defined by sequences of empty disks and define the free space in which paths can be planned. The Explicit Corridor Map [29] consists of a generalized Voronoi Diagram, where the edges are annotated with event points and their closest points to obstacles. The Explicit Corridor Map can be used to find the shortest paths for disks of any size by shrinking the corridors in the map. The corridor is shrunk to account for the disk radius and a triangulation is made from this shrunk corridor. The Funnel algorithm [30] is used to find the shortest path. The corridors are also useful for online obstacle avoidance, because force-based obstacle avoidance approaches only have to consider obstacles that lie within the corridor.

Smooth and visually convincing paths Another extension to the path planning problem is the extension to finding smooth paths that are visually convincing for particular application areas such as simulating virtual pedestrians or Unmanned Aerial Vehicles (UAV). Shortest paths tend to make sharp turns and move very closely to the obstacles in the scene. Humans are more likely to keep a certain distance to the obstacles, whereas UAVs can't take sharp turns mid flight and need smooth paths. Several methods have been developed to create smooth paths.

- **Post-Processing Smoothing** Several methods have been devised that smooth an existing path. Yang [31] uses a continuous cubic Bézier curve to smooth the sharp corners in a path. This method doesn't take obstacles into account and may lead to intersections between the path and the obstacles. Pan [32] uses cubic B-splines to smooth the path. After the path has been smoothed with the spline, the path needs to be checked for collisions. They also provide a fast and reliable continuous collision detection algorithm along spline trajectories. Zhao [33] formulates the problem as a quadratic program that minimizes the weighted L_2-norm of the curvature along the path. This method results in smooth paths while avoiding obstacles. This approach is flexible because additional linear constraints on path length or local curvature can be added to the quadratic program.

- **Visibility-Voronoi diagram** The Visibility-Voronoi diagram [34] is hybrid graph between a Visibility diagram and a Voronoi diagram. The diagram is defined by a parameter c that defines the preferred clearance from obstacles. The graph changes from a Visibility graph at $c = 0$ to a Voronoi diagram for $c = \infty$. The diagram is created by dilating the obstacles and creating the Visibility graph of the dilated scene. If narrow passages disappear through this dilation, then the Voronoi edge from this passage will stay in the graph with an added attribute that holds the maximum clearance. This will allow for characters to still traverse this path if it's a shorter route. The obstacle vertices in the original scene correspond to arcs in the dilated scene. This guarantees that the resulting paths are smooth, while the visibility diagram will ensure that the shortest path will be found. The Visibility-Voronoi diagram is also very useful for disk-based path planning, if the narrow passages are disregarded in the query phase. The Visibility-Voronoi diagram can be constructed in $O(n^2 \log n)$, where n is the number of obstacles vertices. After construction, the diagram can be queried with any graph-search algorithm.

- **IRM** The Indicative Route method [1] combines the potential fields method with a guiding route to find smooth paths. Corridors are used to plan the path, because they are useful for planning paths for characters with a variable radius and for online obstacle avoidance, as mentioned in Section 2.1.6. The method requires a control path as input. This control path is usually the shortest path from the start to the goal position. An attraction point that moves along the control path is used to guide the character from its start to the goal position. Forces are used

to guide the character from start to goal. The force applied by the attraction point is called the steering force. In addition to the steering force, there is a boundary force that pushes the character away from the boundary of the corridor and there is an obstacle avoidance force that allows for avoidance of dynamic obstacles or other characters. All these forces are combined to compute the direction of the character for the next frame. The IRM produces natural looking and smooth paths in real-time. However, the IRM method might take undesired shortcuts or have trouble with intersecting indicative routes. Recently, the IRM method has been generalized to the MIRAN [2] method that fixes these problems, supports weighted regions and adds user control over the amount of path smoothing.

Future extensions The field of traditional path planning in 2D environments is covered on most aspects. There are methods that compute real-time shortest paths for multiple characters in real-time. Extensions to disk-based path planning are also widely explored with good results in terms of speed and optimal paths. Research is already being carried out on extensions to 3D, but then the path planning problem is much harder in 3D. The points where the shortest path touches the obstacles are no longer restricted to a finite set of points, but could lie anywhere on the obstacle edges [35]. Most of the 3D approaches are therefore approximations to the shortest path, that add points to the obstacle edges and create a graph out of these points. Another way to tackle 3D environments is by treating them as 2D-multilayered environments. This method is well suited for path planning in buildings, because of the clear separation between layers. Gradual differences in height, such as a mountain, are not suited for this approach. The Weighted Region problem is an extension that splits up environments in regions and adds weights to these regions. Adding weights to a scene has a similar effect on the search space as moving to 3D. Bending points can lie anywhere on an edge and thus finding optimal paths is unfeasible. The Weighted Region problem will be addresses in detail in the next section. Instead of adding weights, one could also add flows to the scene. Flows work as directional weights and the weight depends on the direction of travel through a region. Research also continues on creating smooth and visually convincing paths [2]. There are several approaches to path planning in dynamic environments, but these methods still suffer from local minima.

2.2 Weighted Region problem

The Weighted Region problem (WRP) is an extension to the path planning problem, in which weights are added to each polygonal region in the scene. Mitchell and Papadimitriou [3] state the WRP as follows: Given a straight-line planar (polygonal) subdivision on which a weighted Euclidean metric is defined, and a start and goal position, find a minimal length (in the weighted sense) path from the start to the goal. The length in the weighted sense is mostly referred to as the costs of the path. These costs are defined by the sum of all subpaths, where each subpath lies in a different weighted region. The costs of a subpath is defined by the length of the path times the weight of the region. Several assumptions about the input are made for the WRP. All polygons are simple, do not overlap each other, and cover the entire environment. Obstacles are represented as regions with a very high weight. The WRP is a generalization of the classical path planning problem. If all weights are equal, then the WRP is equal to the path planning problem. Several facts about the shortest path in a non-weighted region might not apply to the optimal path in a weighted region. For instance, the shortest path in a non-weighted region without obstacles will always be a straight line from the start to the goal position while the optimal path in a weighted region can have arbitrary shapes. Furthermore, the shortest path in a non-weighted region with obstacles is always a sequence of straight-line segments connecting vertices of the obstacle polygons, while in a weighted region the optimal path can bend at any position on an edge of one of the regions. The exact solution to the weighted region problem is proven to be unsolvable [10], because it is impossible to express the WRP as finite algebraic expressions. Even before the unsolvability was

proven in 2012, research had been carried out on finding feasible approximated solutions to the WRP. In this section, we will discuss the different methods that approximate the WRP.

2.2.1 Mitchell and Papadimitriou

Mitchell and Papadimitriou [3] have defined the WRP as mentioned in Section 1. They approximate the solution to the WRP with an approximation algorithm that is based on two lemmas that are derived from the field of optics. Mitchell and Papadimitriou prove that the shortest path in a weighted region will behave according to Fermat's Principle, which states that a ray of light will always take the path that can be traversed in the least time. Mitchell and Papadimitriou derive a useful lemma from this principle.

Lemma 2.1 *Geodesic paths are piecewise linear, except within regions of weight zero (where subpaths may be arbitrary). The intersection of a geodesic path within the interior of a face with $\alpha_j > 0$ is a (possibly empty) set of line segments.*

Here α_j is the weight of face j, and a geodesic path is a path that is locally optimal. Snell's law of refraction can be derived from Fermat's principle, and this law yields a formula to calculate the angles of refraction and incidence when light (or any other wave) passes a border between two isotropic regions. Such a region is a region that has the same properties everywhere. The WRP also contains isotropic regions, because each region has a uniform weight. A second lemma is derived from Snell's law.

Lemma 2.2 *Let α_i and α_j be the weights of two faces incident to an edge e and assume that $\alpha_c = min\{\alpha_i, \alpha_j\}$ and that $\alpha_i, \alpha_j < +\infty$. If p is a geodesic path that passes through the interior of edge e, then p obeys Snell's Law at edge e.*

These two lemmas can always be applied to the WRP and are the basis for the approximation algorithms that solve the problem. The algorithm from Mitchell and Papadimitriou solves the WRP for a finite triangulation of the plane, where all edges and faces have an assigned weight, within an error tolerance $\epsilon > 0$ with respect to the shortest path. This means that the costs of any path computed by the algorithm are at most $1 + \epsilon$ times the costs of the optimal path. The algorithm uses continuous Dijkstra [36, 37] to extend a wavefront where every point on the wavefront has the same distance to the source. Events are triggered at collisions between the wavefront and vertices or edges from the triangulation. The total running time of the algorithm is $O(n^8 \log(\frac{nNW}{w\epsilon}))$, where n is the number of vertices, N is the maximum integer coordinate of any vertex of the triangulation, and w and W are the minimum and maximum weights. The solution isn't efficient, but it is the first theoretical result.

2.2.2 Pathnet

Mata and Mitchell [4] discretize the search space to approximate the optimal path. Their method creates a graph called Pathnet that can be queried to find the optimal path between two points. The algorithm makes use of cones around all vertices, which limit the paths that extend from a vertex v. These cones discretize the orientations around the vertex from $[0, \pi]$ to k directions. The boundaries of these cones are traced, while obeying Snell's Law at edge intersections, until the boundaries intersect with different edges. The *splitting* vertex u that is adjacent to both these edges will be added to the Pathnet, together with the edge uv that lies entirely within the cone and obeys Snell's Law at each edge intersection. If the boundary of a cone intersects with an edge at an angle greater than the critical angle, then the tracing is also stopped and a *critical* vertex u is created on the intersected edge at the position where the angle is exactly critical. The critical angle is defined as the angle where the path switches from edge-crossing to edge-using, as shown in Figure 9 and 10, respectively. This *critical* vertex is added to the Pathnet, together with the edge uv that lies entirely within the cone

and obeys Snell's Law at each edge intersection. The Pathnet can be constructed in $O(kn^3)$, where k is the number of cones. Once the Pathnet has been constructed it can be queried with Dijkstra's algorithm to find the ϵ-approximate shortest path in $O(n \log n)$ time, where $\epsilon = O(\frac{W/w}{k\theta_{min}})$. W/w is the ratio of the maximum non-infinite weight to the minimum non-zero weight and θ_{min} is the minimum among the internal face angles of the subdivision.

2.2.3 Steiner points

In 1998, Aleksandrov, Lanthier and Maheshwari [5] came up with a new discretization scheme through the use of Steiner points. These points are added to the edges of the triangulation of the scene. For each vertex with an incident edge, Steiner points are added with a logarithmic distribution. Because this distribution will lead to infinitesimal distances between the Steiner points close to the vertex, the authors define a *vertex vicinity*. The vertex vicinity is a small circle around the vertex that is void of Steiner points. There are now two sets of Steiner points on each edge, one starting from each vertex. The authors define an interval to be the distance between two consecutive Steiner points in order to merge the two Steiner point sets. The sets are merged by removing the Steiner points with the largest interval for each set of overlapping intervals. A graph is created by connecting all Steiner points and vertices within each triangle with each other. The approximate shortest path is found by applying an improved version of Dijkstra's algorithm to the graph. Dijkstra's algorithm was improved through the use of Fibonacci heaps. The total running time of this algorithm is $O(mn \log(mn) + nm^2)$, where m is the total number of Steiner points. The total number of Steiner points is defined as $m = O(\log_\delta(L/r))$, where L is the length of the longest edge and r is ϵ times the minimum distance from any vertex to the boundary of the union of its incident faces and $\delta \geq 1 + \epsilon \sin\theta$, where θ is the minimum angle between any two adjacent edges of the surface. two

2.2.4 BUSHWHACK

Reif and Sun focused on improving graph-search methods for the WRP. First they introduced the interval datastructure [38]. An interval is always defined by two edges (e, e') of the triangulation and a point v on e. The interval consists of all points $v*$ on e' where the optimal costs from s to v plus the costs from v to $v*$ are lower than the optimal costs from s to v' plus the costs from v' to $v*$ for any point v' on e, where s is the start position. As shown in Figure 1, this will create a cone for each point v. Next they formalized their problem definition as the *Discrete Weighted Shortest Path Problem*: Given a 2D space of n triangular regions with weights, where each edge has m Steiner points, find a weighted shortest path that only intersects region boundaries at vertices or Steiner points for any given source and destination. Their research resulted in a new graph-search algorithm that takes advantage of the underlying geometrical properties of the graph: BUSHWHACK. BUSHWHACK [11] maintains a set of intervals to prune the search. An ϵ-approximate path can be found in $O(\frac{n}{\epsilon}(\log\frac{1}{\epsilon} + \log n)\log\frac{1}{\epsilon})$ with BUSHWHACK.

2.2.5 Steiner points on bisectors

Aleksandrov, Maheshwari and Sack [6] improved the Steiner points approach in 2005 by adding Steiner points to the bisectors of the triangles. This improvement lead to a running time of $O(C(P)\frac{n}{\sqrt{\epsilon}}\log(\frac{n}{\epsilon})$ $\log(\frac{1}{\epsilon}))$, where $C(P)$ captures geometric parameters, such as the average of the reciprocals of the sinuses of the angles of P, and the weights of the faces of the triangulation P. However, the improvements that the BUSHWHACK algorithm could provide can't be used within this method. The cones from the BUSHWHACK algorithm would start on one bisector and end on a bisector in another triangle. If the weights of the triangles differ, then the cone would be invalid, because it assumes that the weight within the cone is homogeneous. This method has a better running time when compared to Steiner

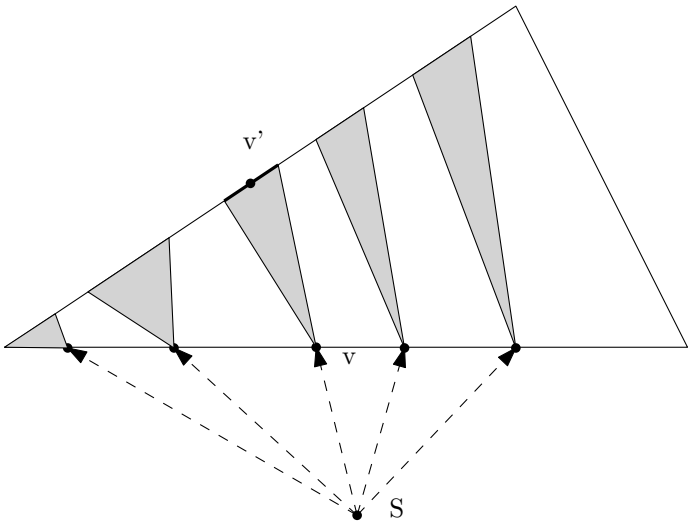


Figure 1: The BUSHWHACK algorithm uses intervals to prune the search

points on the edges of the triangulation combined with the improved version of Dijkstra's algorithm. But the authors didn't compare their method to the Steiner points on the edges of the triangulation combined with the BUSHWHACK search method.

2.2.6 All Points Query

The latest addition to the field is the All Points Query (APQ) [7] datastructure. This datastructure can find the ϵ-approximate optimal path in $O(q)$ for any pair of source and destination points, where $q = max(\frac{1}{\sqrt{\epsilon}} \log^2 \frac{1}{\epsilon}, \frac{(g+1)^{2/3} n^{1/3}}{\sqrt{\epsilon}})$. Genus g is defined as the number of holes in the polyhedral surface P. This is the first all-pairs query algorithm for weighted regions. A partitioning technique is used to create the APQ. The APQ is supported by SSSP trees [5, 6, 38]. An SSSP tree contains all optimal paths from a single source to all other points. SSSP trees are constructed by discretizing the triangulated polyhedral surface with Steiner points on the bisectors of all triangles. The APQ datastructure is constructed in $O(\frac{(g+1)n^2}{\epsilon^{3/2}q} \log \frac{n}{\epsilon} \log^4 \frac{1}{\epsilon})$ time and its size is $O(\frac{(g+1)n^2}{\epsilon^{3/2}q} \log^4 \frac{1}{\epsilon})$. This method has a large construction time, but once the datastructure is build, it can be queried very fast. The datastructure could be useful when many queries are performed on the same scene.

2.2.7 Conclusion and discussion

So far several methods have been devised to tackle the weighted region problem. Increasingly better discretization scheme's are discovered to optimize the search and a new search algorithm BUSHWHACK was presented. Through the variation of the ϵ-parameter, it is possible to balance between path optimality and computational costs. However, most articles state that the ϵ-value shouldn't be larger than 1, which leads to algorithms that currently will not be able to run in real-time. The ϵ-value is a quality measure for the approximation and is usually kept small because the focus lies on path optimality. In Section 4.3 we will explore the boundaries of the ϵ-value. All methods use a polygonal subdivision with weights as input and none of the articles takes specific assumptions about the input to optimize the search.

2.3 Weighted Region problem in other research fields

Several research fields address the WRP in their own way. These research fields work on the WRP independently, because there are no references between these fields. This section describes several of these research fields and the research that they have done on the WRP.

2.3.1 Minimum risk path planning for UAVs

Unmanned Aerial Vehicles (UAV) require robust artificial intelligence in order to navigate autonomously and in a save way. An important part is path planning. Recently, researchers started to focus on path planning while avoiding risks. A risks could be an enemy threat, but also flying low near a mountain. Because UAVs fly at a certain speed, it is impossible to stop and make sharp turns. The resulting paths must be smooth if they are to be used as guidance for a flight. Fillipis [39] compared several methods to achieve a minimum risk path. Fillipis used a geometric predefined trajectory, manual waypoints definition, automatic waypoints distribution and an A*-based approach. This approach was applied to a grid of 40x40 cells, where the cells had assigned risk values. The cost function f was slightly modified by taking a weighted sum of g and h:

$$f = \alpha g + \beta h.$$

The weights α and β are adjusted based on the mission requirements. The final path is smoothed with Dubins curves. The path finding and smoothing can be done in real-time.

2.3.2 Risk averse motion planning for mobile robots

Researches in the field of Robotics search for efficient methods to solve the WRP. They focus on autonomous navigation of robots while avoiding risks. As opposed to minimum risk path planning for UAVs, this problem is less constrained, because robots are allowed to take sharp turns. MacMillan [40] uses a riskmap based on certain beacon locations. These beacons have a limited range and scan the area for risks. The resulting riskmap is a planar subdivision with assigned weights. The map is then uniformly sampled to generate a roadmap. The risk map is used to assign weights to the edges of the roadmap. Once the roadmap is constructed, it can be queried to retrieve shortest paths. MacMillan doesn't describe any search methods that should be used to find the optimal path, but focuses on the roadmap construction. Chestnutt [41] proposes a weighted A* approach for legged robot navigation planning. The search is performed in the space of possible footstep actions the robot can perform. The cost of a path is a combination of action costs and terrain costs. The focus of the paper lies on the choice of stepping actions used for expanding nodes in the A* search, and the use of the weighted A* approach isn't elaborated on.

2.3.3 Military simulations

Military simulations also include the WRP, but these systems require additional requirements. The weights correspond to threats in the environment and the cover given by the terrain. Flexible algorithms are required to cope with the possible movements of threats. Reece [42] worked on tactical path planning for individual combatants in military simulations. He compared different methods to find a suitable method for military simulations. Skeleton methods and potential fields were disregarded because those methods don't work well with weights. The algorithm from Mitchell [3] was disregarded because it cannot cope with the varying costs of the weights. Finally, a cell decomposition method is chosen. The cell decomposition results in a weighted graph. A* is used to find optimal paths within the grid. Kamphuis [43] uses a roadmap-based method to plan paths for fireteams while maintaining formations and avoiding being seen by the enemy. The method has a preprocess phase that starts

with the creation of a roadmap. Corridors are created around the edges of the roadmap. The union of the corridors should capture the free space as much as possible. Next, weights are assigned to the spine and the edges of the corridor. The spine of a corridor is a line segment that lies on the roadmap edge incident to the corridor and is bounded by the corridor edges. The weight values are based on the distance to threats, visibility from threats and distance to the spine. The roadmap edges are given weights by taking the length of the edges times the average weight of the two corridors incident to the edge. The online phase starts with the recalculation of the weights due to enemy movement. The roadmap is then used to find a corridor sequence for the team leader of the fireteam. The paper doesn't mention which graph-search algorithm is used. The resulting sequence of nodes and edges is converted to a sequence of corridors. This corridor sequence is used to plan the path for the team leader while maintaining several fireteam dynamics and staying out of the line of sight of the enemy. The online phase can be executed in real-time.

2.3.4 Conclusion and discussion

The methods proposed by researchers from the above-mentioned fields focus on avoiding danger rather than on computing the optimal path efficiently. The UAV methods use grid-based approaches and could be improved by using exact cell decomposition approaches. The hardest issue for those UAV methods lies within the creation of smooth paths that could be executed by a UAV, while still keeping the path as efficient as possible to reduce flight times and fuel costs. The WRP is mentioned in the robotics field, but the focus lies on different aspects. The paper by MacMillan [40] has more focus on the creation of the roadmap from the beacon readings than solving the actual WRP. Chestnutt [41] focuses more on choosing the right stepping motion for legged robots. Both military simulation papers [42, 43] provide real-time methods for solving the WRP. These methods give good results, but the grid method is slow on large environments, and the preprocessing phase of Kamphuis' method needs too much time for single queries.

2.4 Weighted Region problem in practice

The Weighted Region problem has been addressed by researchers from several fields. In the gaming industry, this problem is similar to tactical pathfinding. Several games use A* with weights to model 'good' positions on the map. For instance, the bots in Counter Strike: Condition Zero [44] use this to find sniper spots and to make paths that require jumping or crouching more expensive. Killzone [45] uses tactical pathfinding to move characters from cover to cover.

2.4.1 Tactical pathfinding

William van de Sterren [45] is one of the developers of the tactical AI for Killzone. Killzone is a video game created by the dutch studio Guerrilla [46]. Guerrilla works on creating a game-independent AI framework that utilizes environment independent algorithms instead of waypoints and scripts. One the aspects they address is tactical pathfinding. Firefights between the player and bots are a common occurrence in Killzone. These bots use tactical pathfinding to move from one position to another while avoiding enemy fire. A cost function is used to determine the safety of all cells in a grid. The cost function depends on the line of sight of the enemies. A* is then used to find the optimal path while taking these cost values into account. Figure 2 shows the same scene twice. The left part shows non-tactical A*, and the right parts shows tactical pathfinding A*. The yellow cells are occupied by enemies, and the orange cells show all cells that can be seen by enemies. Tactical pathfinding clearly results in a safer path. The downside of tactical pathfinding is the number of cells evaluated. In the example shown in Figure 2 (right) almost all cells are evaluated. The Euclidean distance heuristic doesn't perform well in weighted regions, and it is hard to find an efficient heuristic that guides A*

through weighted grids. The main problem is that current heuristics estimate the costs to the goal by taking the Euclidean distance to the goal, but the weights are not taken into consideration. The lowest weight in the scene can be multiplied with the distance to keep the heuristic admissible and achieve better estimates, but the heuristic will always underestimate in regions with higher weights. Consequently, many cell will be traversed during the search, leading to increased running times. Van de Sterren suggests to use hierarchical pathfinding to lower the number of cells evaluated.

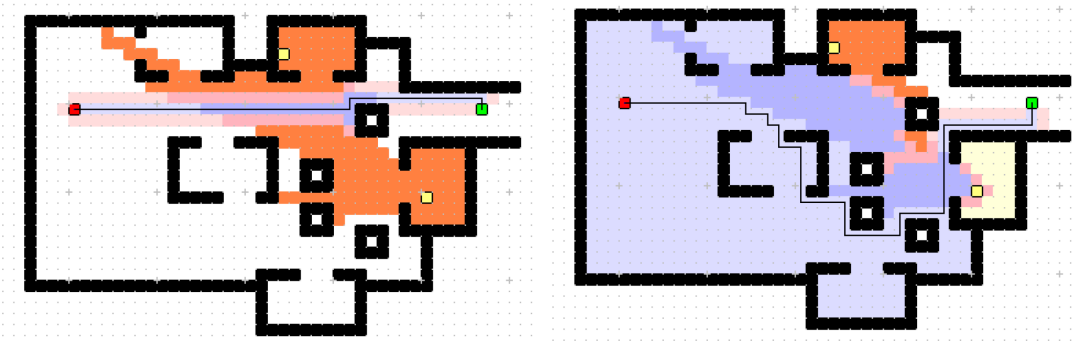


Figure 2: A scene with a red start position and green goal position. The yellow cells are occupied by enemies and the orange cells can be seen by these enemies. The left side shows a non-tactical A* path, and the right side shows the final path of the tactical pathfinding A* method. Blue and pink cells are cells that are examined by the A* algorithm, where the pink cells denote the boundary of examined cells.

2.4.2 Influence maps

Influence Maps [47] can be used to deciding what parts of the map are controlled by which team and for identifying chokepoints. These maps are commonly used in real-time strategy games. The influence map contains regions with a certain influence value. These influence values consist of two parts: static and dynamic. The static part is defined by the environment and by historical influence data. The dynamic part is defined by enemy entities, explosions or enemy fire. The dynamic influence values propagate to neighbouring nodes and they decay over time. The influence map is created by adding the influence values to the current map of the environment. This could be any type of map from grids to waypoint graphs. With the influence map in place, paths are planned with the potential fields method [48] to avoid hazards. As noted in Section 2.1.3, these potentials fields suffer from local minima and could lead to units getting stuck.

2.4.3 Conclusion and discussion

Practical applications have a hard requirement on real-time execution. Currently the ϵ-approximation methods from Section 2.2 aren't used in practical applications, because these methods focus on ϵ-optimal paths and are unable to perform in real-time. Instead, A* on weighted grids is the most common method. The computations for A* on weighted grids increase fast and will become problematic for large maps, because it's hard to find a good heuristic. The potential fields method suffers from local minima and yields even more problems on weighted maps due to the increasing number of local minima in comparison to non-weighted maps. Any of the methods from Section 2.2 might be able to overcome these problems, but the resulting paths can only be rough approximations of the optimal solution to guarantee real-time performance. Up until now there are no practical applications known that use any of these methods, and there is a large gap between ϵ-optimal and time-efficient methods.

2.5 Conclusions

Although several ϵ-optimal methods have been devised to approximate the WRP, none of these methods are able to perform in real-time. Research fields such as robotics, UAVs and military simulation, as well as practical applications such as games, use path planning algorithms with assigned weights to tackle the WRP. The ϵ-optimal methods are hardly mentioned in any of these research fields. A* on weighted grids is the most common method and is able to run in real-time. However, high-resolution grids or multiple characters are problematic because A* on a weighted grid performs much slower than A* on non-weighted regions, because the underestimation of the heuristics for weighted regions cause A* to traverse many more nodes. This leads to a large gap between optimal and efficient solutions to the WRP. The gap could be closed from both sides. Better heuristics for A* could improve the running time and allow for higher resolution grids to improve optimality. Higher ϵ-values could lead to faster execution at the cost of path optimality. A hierarchical method that combines the speed of A* on a high level with the path quality of ϵ-optimal methods at lower levels might be the solution. Once the high-level optimal path has been found by the A* algorithm, the ϵ-optimal method adds Steiner points within the cells from the high-level path. One could also think of a method that uses ϵ-optimal methods at a high level as a heuristic for A*. The high abstraction level will be a sparse graph, and ϵ-optimal methods will be able to compute results fast, while the A* algorithm will have better guidance to the goal with a better heuristic. Assumptions on the input of the WRP might also prove to be the solution to this problem, but there are currently no methods that use this strategy. If we can simplify the WRP by taking assumptions on the shape or size of regions, whether the regions are axis-aligned or the amount and values of the weights, then we might be able to solve this modified WRP faster. It is important that the gap between optimal and efficient solutions to the WRP is closed because there are many applications that can profit from a solution that is both near-optimal and efficient. Such a solution could be useful to improve crowd simulations by creating individual paths for characters with different individual preferences. These preferences could be modeled through the weights of different regions. This solution could also be used to generate the indicative routes for the MIRAN method [2]. Games could use larger maps and enable tactical pathfinding for many characters at the same time. The decrease in pathfinding time could also be used to render better graphics or spend more time on the AI. In the rest of the thesis we will attempt to close the gap. In Section 3 we improve the heuristics for A* on weighted grids. Section 4 discusses the boundaries of the ϵ-value, and in Section 5 we introduce several hybrid methods by combining A* grid results with an ϵ-optimal method.

3 A* grid approaches

In this section, we will look at A* grid approaches to tackle the WRP. These methods consist of discretizing the scene with a grid, assigning weights to the cells and using A* to find an optimal path. As mentioned in Section 2.4.3, these methods are applied in many practical applications. First, we will examine how the solutions given by grid-based approaches compare to an optimal solution in terms of path-costs. The upper bound on the overestimation of the path costs can be expressed with the ϵ-value. Any path computed by a method that is ϵ-optimal will have a path costs that is always smaller than the length of the optimal path times $1+\epsilon$. The ϵ-value of grid-based methods is currently unknown and we will introduce a formula to calculate the ϵ-value for grid-based methods that depends on the weights and the resolution of the grid. If we can determine the ϵ-value of grid-based methods, then we can compare A* grid approaches with other ϵ-approximation methods. We will also look at the heuristics used by A* and try to improve heuristics to better suit weighted grids.

3.1 ϵ-optimality of grid methods

To compare different approaches to the WRP efficiently we need to be able to compare the efficiency and path quality of these methods. Efficiency is easily compared by looking at the running times of the different algorithms. Path quality can be compared by looking at the optimality bounds of the different methods as expressed with the ϵ-value. However, there is no such bound known for grid-based methods. If we could determine such an optimality bound, then we can compare grid-based method to other ϵ-optimal methods.

3.1.1 Grids

First we have to define what kind of graph we are using. There are two possible ways of creating a graph from a grid. As shown in Figure 3, the graph nodes could be the intersection of the cell borders or the center of the cells. We will always use the cell centers as graph nodes. Each node will be connected to the nodes of all eight surrounding cells. This will allow characters to move straight and diagonally within the graph.

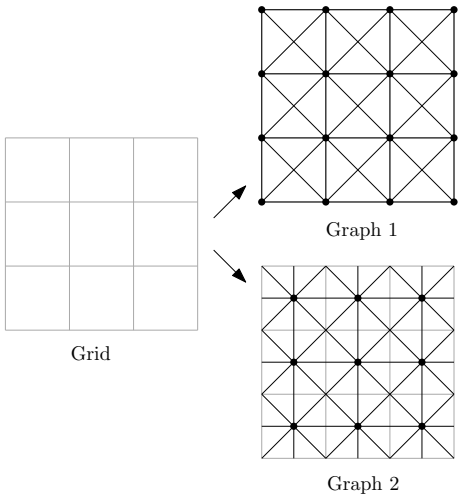


Figure 3: Two possible graphs for a rectilinear grid. In Graph 1, the nodes are defined as the cell border intersections of the grid. In Graph 2, the nodes are defined as the center points of each grid cell.

3.1.2 Assumptions

We will start by making several assumptions. These assumptions will simplify the problem definition, and solutions for the simplified version can be found in an easier way. Then we can expand the solution to more general cases by dropping the assumptions one by one. We will start with the following assumptions:

- The start and goal position lie on grid nodes
- All region boundaries lie on cell edges
- The scene consists of one region with weight 1
- All cells have unit length

3.1.3 Straight lines

First, we will look at an example scene that contains only one region with weight 1 and cells of unit length. In order to find the maximum ϵ-value, we will compare the length of the optimal straight-line path and the grid path for several goal positions around a fixed start position. As shown in Figure 4, we only have to look at one direction of 45° due to the symmetry of a regular grid. Any maximum ϵ-value found within this wedge of 45° can be reflected in the sides of the wedge to find corresponding ϵ-values in the other directions. The length of the shortest path between an arbitrary start $s = (x_s, y_s)$ and goal node $g = (x_g, y_g)$ is described by its Euclidean distance: $d = \sqrt{(x_s - x_g)^2 + (y_s - y_g)^2}$. The length of the shortest path in the grid between an arbitrary start and goal node is defined by $d_{grid} = a\sqrt{2} + (|x_s - x_g| - a) + (|y_s - y_g| - a)$, where a refers to the number of diagonal steps: $a = \min(|x_s - x_g|, |y_s - y_g|)$.

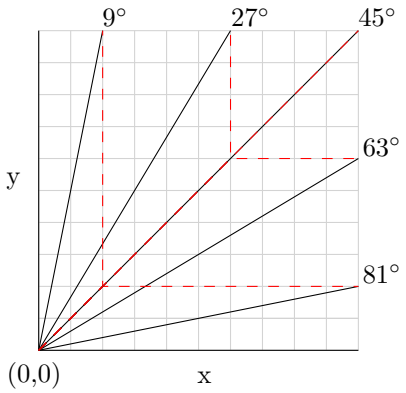


Figure 4: A grid with shortest Euclidean paths (black straight lines) and shortest grid paths (red dashed lines) from (0,0) to goals at multiple angles. The shortest paths can always be reflected in the line at 45°, thus any ϵ-value we find between 0° and 45° can also be found between 45° and 90°. This holds for all directions, and that's why we only have to look at one wedge of 45° to find the ϵ-value in all directions.

An example of Euclidean shortest path lengths and grid shortest path lengths for the example wedge are shown in Figure 5, where $s = (0, 0)$ and $g = (x, 10)$. If we subtract these two distances, then we will find the difference between the shortest path and the shortest path in the grid: $dif = d_{grid} - d$. The ϵ-value can then be found by $\epsilon = dif/d$. The ϵ-values for the example wedge are plotted in Figure 6, where $s = (0, 0)$ and $g = (x, 10)$.

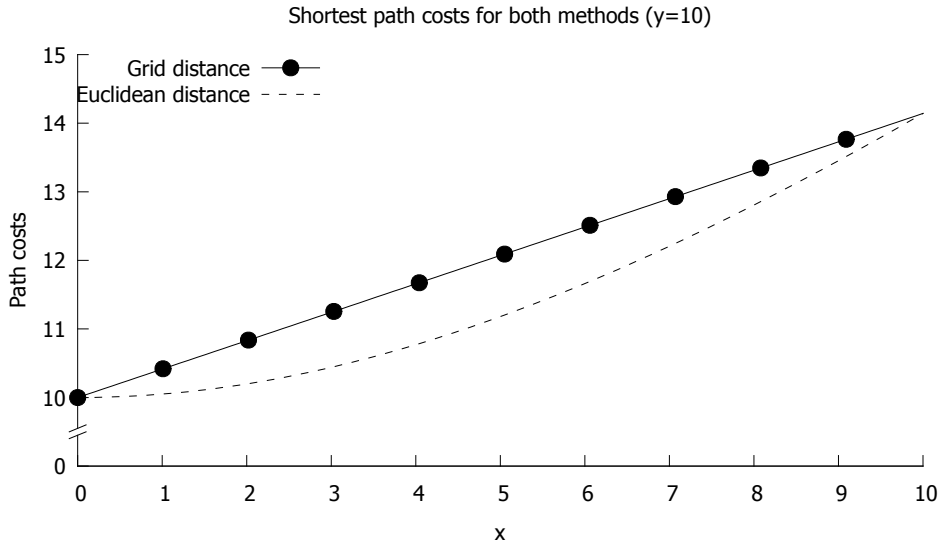


Figure 5: Graph of the Euclidean distance and grid distance for $y = 10$ and $x = [0 : 10]$.

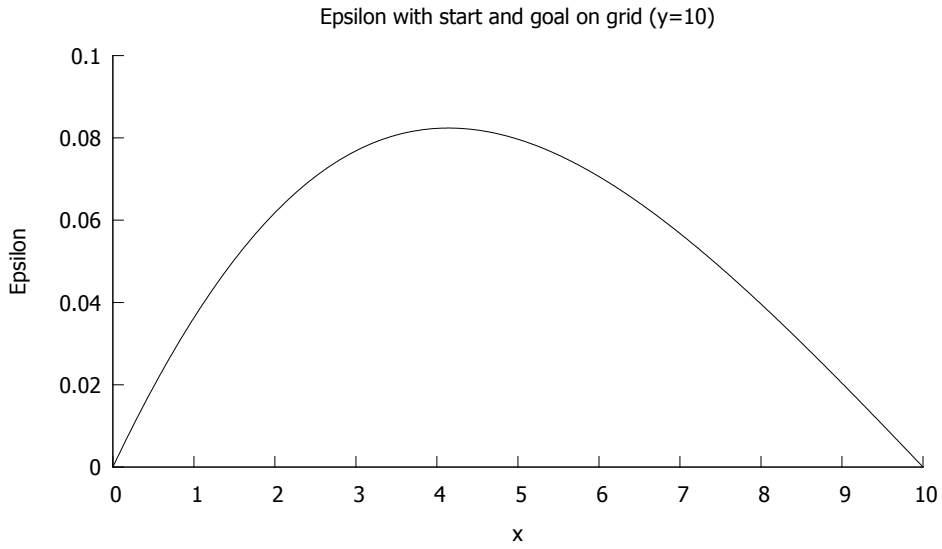


Figure 6: Graph of the epsilon values for $y = 10$ and $x = [0 : 10]$.

The maximum ϵ-value can be found by taking the derivative of ϵ, finding the x-value where $\epsilon' = 0$ and putting this x value back into the ϵ formula. To find the derivate of ϵ, we have to make three out of the four parameters constant. The start position can be set to (0,0), and we will introduce a new variable m that controls the size of the wedge (e.g. the maximum x- and y-coordinate). With static values $x_s = 0$, $y_s = 0$ and $y_g = m$ and one variables $x_g = x$ ranging from $[0 : m]$ we can transform the ϵ-formula. We first note that a is now always equal to x. Thus $\epsilon = \frac{(x\sqrt{2}+m-x-\sqrt{x^2+m^2})}{\sqrt{x^2+m^2}}$.

The derivate of ϵ is $\epsilon' = \frac{m(\sqrt{2}m-m-x)}{(m^2+x^2)^{\frac{3}{2}}}$, the derivation from ϵ to ϵ' is shown in Appendix A. Solving $\epsilon' = 0$ yields $x = \sqrt{2}m - m$ as shown in Appendix B. We can then substitute x to find ϵ. During this substitution, as shown in Appendix C, we lose the m parameter. Thus we can conclude that the ϵ-value is independent of the wedge size m. After the substitution we will have the general formula for calculating the ϵ-value: $\epsilon = \sqrt{(4-2\sqrt{2})} - 1 \approx 0.0824$.

3.1.4 Any start and goal position

Next we will drop the assumption that the start and goal positions lie on grid nodes. We can use arbitrary start and goal positions by connecting the start and goal positions to the nodes of the cells that contain the start and goal positions. This approach will not yield optimal paths, but in this paragraph we will show that the additional cost are insignificant. The costs of the path will then be the combination of the costs from the start position to the center of the start cell, the costs from the start cell to the goal cell and the costs from the goal position to the center of the goal cell. We can adjust the formula for d_{grid} to reflect this behaviour. We start by defining a mapping from the coordinates of arbitrary points to the center points of the cells to find the grid node that corresponds to a given query point. By (x'_s, y'_s), (x'_g, y'_g), we denote the mapped coordinates of the coordinates of our start and goal points, respectively. We then let $x'_s = \lfloor x_s \rfloor + \frac{1}{2}$, $y'_s = \lfloor y_s \rfloor + \frac{1}{2}$, $x'_g = \lfloor x_g \rfloor + \frac{1}{2}$ and $y'_g = \lfloor y_g \rfloor + \frac{1}{2}$. The new grid distance is then computed as $d_{grid} = a\sqrt{2} + (|x'_s - x'_g| - a) + (|y'_s - y'_g| - a)$, where $a = \min(|x'_s - x'_g|, |y'_s - y'_g|)$. The formula for ϵ now contains several floor functions and is no longer differentiable. However, we can examine the impact on the maximum ϵ-value. The maximum difference between the shortest path and grid path will occur when the start or goal position, or both, lie in the corner of a cell. The grid path will have an extra cost $\frac{1}{2}\sqrt{2}$ to reach the grid node. If both the start and goal points lie in the corner of a cell, an extra cost of $\sqrt{2}$ is needed to connect the points to the grid path. This value of $\sqrt{2}$ is only added once to the path and is completely independent of the path itself. That's why the ϵ-value will remain unchanged. One can clearly see that as soon as the path reaches a certain length, the extra costs of $\sqrt{2}$ will no longer be significant. We will remove the mapped coordinates (x'_s, y'_s), (x'_g, y'_g) from the grid distance d_{grid}, because it has no significant effect on the ϵ-value.

3.1.5 Obstacles

We will now add obstacles to the scene. A shortest path among obstacles will always be a straight line from start to goal, or it will touch the obstacle vertices as can be concluded from the Visibility Graph methods (Section 2.1.4). As shown in Figure 7, the shortest path can be split up in several segments that only have the start position, goal position or obstacle vertex as endpoints. For each of these segments we can calculate the ϵ-value with the formula from the previous paragraph, because each segment lies in a subregion without obstacles. In contrast to the previous paragraph, the additional cost of $\sqrt{2}$ seems to become significant, because the maximum number of segments is equal to the length of the path. However, each segment lies within an obstacle-free rectangle where the segment of the shortest path is always longer than the grid path that is contained within the obstacle-free rectangle. For each obstacle-free region, except for the ones containing the start and goal position, the segment of the shortest path will be equal to the diagonal of the rectangle or the longest side. There is no straight path that lies within a rectangle that is longer than the diagonal, and a straight grid path is always shorter than the longest side of the rectangle. To find the ϵ-value of the entire path, we need to link all the segments together. The end and start point of two consecutive segments are the same obstacle vertex, but they correspond to two different grid nodes. For instance in Figure 7, consecutive segments (Start,A) and (A,B) both share vertex A, but segment (Start,A) ends in cell 1, while segment (A,B) start in cell 2. Linking these nodes will cost an additional $\sqrt{2}$ per link. The maximum number

of obstacles touched by the shortest path is equal to the length of the path minus one as is shown in Figure 8. If n is the length of an optimal path through a scene with obstacle, then the grid path can be divided into a maximum of n segments. Each of these segments need to be linked to each other, making the number of links smaller than n. The relative overestimation of each segment is equal to the ϵ-value from Section 3.1.3:$\sqrt{(4-2\sqrt{2})}-1$, and the link costs are $\sqrt{2}$. For an optimal path of length n the absolute overestimation of the grid path will be bounded by $n*(\sqrt{(4-2\sqrt{2})}-1)+n*\sqrt{2}$. We can find the relative overestimation of the total path by dividing by the length of the path: n. As such, the resulting ϵ-value will be: $\epsilon=\sqrt{2}+\sqrt{(4-2\sqrt{2})}-1\approx 1.4966$.

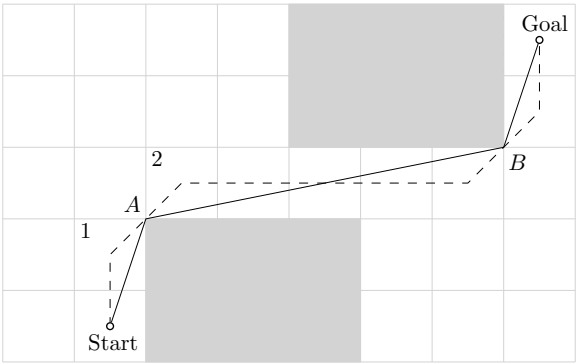


Figure 7: A shortest path (straight line) and grid path (dashed line) among obstacles (grey rectangles). The shortest path can be split up in three segments: (Start,A), (A,B), (B, Goal).

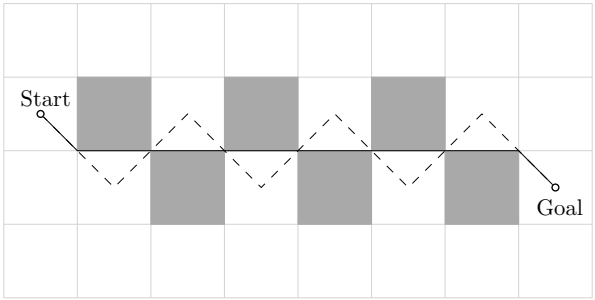


Figure 8: A shortest path (straight line) and grid path (dashed line) among obstacles (grey rectangles). This scene shows the worst case scenario where the maximum number of links is needed.

3.1.6 Weighted regions

We will now extend from obstacles to weighted regions. The main difference is that the shortest path might now intersect the weighted regions at any location, instead of always touching the obstacle vertices. In scenes with weighted regions, two types of intersections between the shortest path and an region edge may occur. The first type is shown in Figure 9, and it is an edge crossing intersection. This intersection causes the path to bend at an angle defined by the weights of both regions. The second type is shown in Figure 10, and it is an edge using intersection. Here the path uses a part of an edge of a neighboring region with a lower cost. Both intersections can be handled in one case.

The segments of the optimal path are now defined by the intersections of the optimal path with the region boundaries, and thus each segment will always be completely enclosed by one region. Because the weights within a segment are uniform, the ϵ-value for each segment remains unchanged. However, the link costs are different compared to the unweighted scenario. An example of an edge intersection is shown in Figure 11, where $w_1 < w_2$. We can find a grid node A in the region with the higher weight that has the intersection on one of its cell borders. In the other region, we can find one or two grid nodes that are candidates to lie on the shortest path in the grid. We can always find the grid node B that shares a cell border with A. The shortest path will be split up in two segments by edge e that separates the two regions. If the shortest path intersects a cell border adjacent to the intersected cell border, then the grid node that shares this border with B will also become a candidate C. The link costs will consist of the connection between A and B or the connection between d and A, where d is the intersection between AC and e. The connection between A and C is split in half because the segment costs, for the segment between $Start$ and B, already covers the costs from the start to d. The final link costs will be $\min\left(\frac{\sqrt{2}}{2}w_{max}, \frac{1}{2}w_{min} + \frac{1}{2}w_{max}\right)$, where w_{min} and w_{max} are the minimum and maximum weight of the scene, respectively.

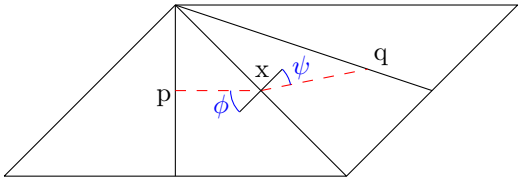


Figure 9: Edge crossing intersection.

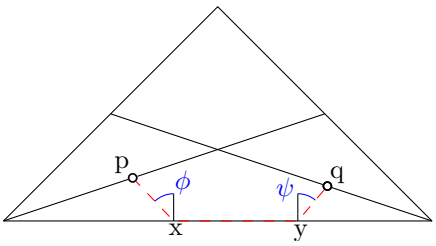


Figure 10: Edge using intersection.

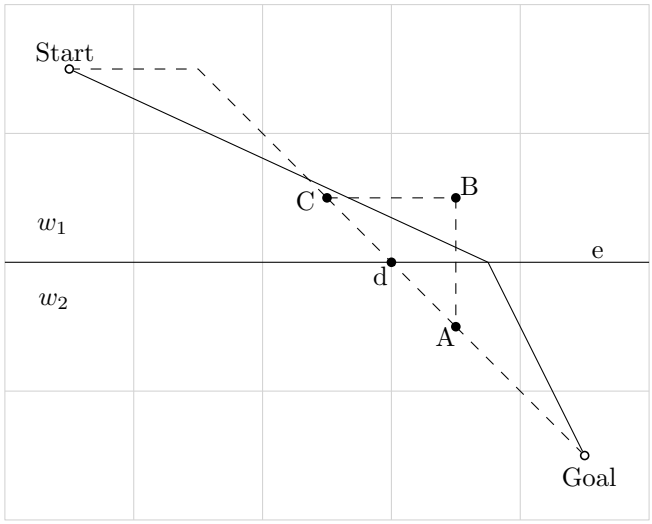


Figure 11: The shortest path (straight line) and grid path (dashed line) among two weighted regions and their boundary edge e, where $w_1 < w_2$. Candidate grid nodes are named A, B and C and the intersection between e and AC is named d.

3.1.7 Grid resolution

We will now look at the impact of different grid resolutions to the ϵ-value. We will start by introducing a new parameter r that defines the size of the cells. In the previous paragraphs we used $r = 1$, and we will now extend the discussion to arbitrary values of r to find an upper bound for the ϵ-value in the general case. First we will adjust the grid distance: $d_{grid} = ar\sqrt{2} + (|x_s - x_g| - a) + (|y_s - y_g| - a)$, where $a = \frac{\min(|x'_s - x'_g|, |y'_s - y'_g|)}{r}$. Variable r can be removed, because in a we divide by r but later we multiply a by r. The grid distance will become: $d_{grid} = a\sqrt{2} + (|x_s - x_g| - a) + (|y_s - y_g| - a)$, where $a = \min(|x'_s - x'_g|, |y'_s - y'_g|)$. The formula is exactly the same as for straight lines, and thus we can conclude that the resolution has no impact on the ϵ-value of the segments.

However, the resolution does have an impact on the link costs. The link costs will be $r\sqrt{2}$ per link for obstacles and $r * \min(\frac{\sqrt{2}}{2} w_{max}, \frac{1}{2} w_{min} + \frac{1}{2} w_{max})$ for weighted regions. An interesting resolution to look at is one that comes very close to zero. An infinitesimal resolution should lead to the best approximation of the problem in the Euclidean space. If such a resolution is used, then link costs will be reduced to an infinitesimal value, so we can conclude that the lower bound on the ϵ-value is equal to formula from Section 3.1.3: $\epsilon = \sqrt{4 - 2\sqrt{2}} - 1 \approx 0.0824$. The maximum ϵ-values are shown in Table 1 for different resolutions.

Cell size	ϵ-value		
	Segments	Links	Total
$\frac{1}{2}$	0.0824	0.7071	0.7895
1	0.0824	1.4142	1.4966
2	0.0824	2.8284	2.9108
4	0.0824	5.6569	5.7393
8	0.0824	11.3137	11.3961

Table 1: Table showing the different ϵ-values for different resolutions.

3.1.8 Region boundary on cell edges

To keep the ϵ-value below ∞, we have to keep the assumption that the region boundaries are on the cell edges. If we dropped this assumption, then the homotopy of the grid might differ from the actual scene. One could reason that the resolution could always be chosen in such a way that the homotopy remains unchanged, but increasing the resolution will just lead to more region boundaries aligning with the cell edges. A straightforward method to convert a scene to a grid is by drawing the scene and reading the pixel values from the framebuffer. For instance, OpenGL [49] samples the center of each pixel to determine its value.

As can be seen in Figure 12, if an obstacle blocks the center but not the rest of the scene, then the entire cell will be blocked while the actual scene is not. This loss of information could lead to situations where the grid does not contain a path from start to goal while the scene does. We can prevent such a situation by restricting all region boundaries to lie on the cell edges. Problems can also occur when the homotopy of the grid is equal to the scene. If the gray area in Figure 12 is a very high weight w, then the ϵ-value will become $\epsilon * w$ making it very high and thus useless.

3.1.9 Tiles

Up until now we have been using octiles as cells in the grid. Octiles are always connected to all of their eight neighbours. Tiles only allow horizontal and vertical movement and are only connected to four of their neighbours. We will now look at ϵ-values for tiles. The grid distance in this case is the Manhattan

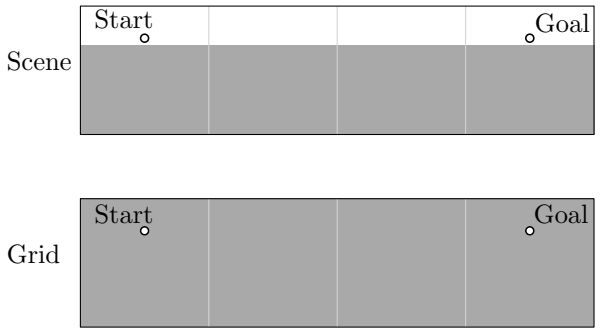


Figure 12: A scene of 1 by 4 with an obstacle (grey) and the cells of the grid that is extracted from this scene. Because the centers of each cell lie within the obstacle, all cells will become obstacle cells. In the grid there will be no solution to the shortest path from start to goal.

distance between the start and goal points, i.e. $d_{grid} = |x_s - x_g| + |y_s - y_g|$. The grid distance is shown in Figure 13 for a wedge where $s = (0, 0)$ and $g = (x, 10)$. For the tiles, one can clearly see that the maximum ϵ-values occurs with diagonal movements. This ϵ-value is equal to $\frac{2-\sqrt{2}}{\sqrt{2}} \approx 0.4142$. The link costs for obstacle regions will increase to 2, because there are no diagonal links allowed. The weighted region link costs will be $\frac{1}{2}w_{min} + \frac{1}{2}$, because the diagonal movement from candidate A to C is not allowed. One could reason that there are two possible paths that replace the diagonal from A to C: ABC and $AB'C$, where B' is B reflected in AC as is shown in Figure 14. However, A and B' lie in the region with the highest weight, thus ABC will always be less expensive than $AB'C$.

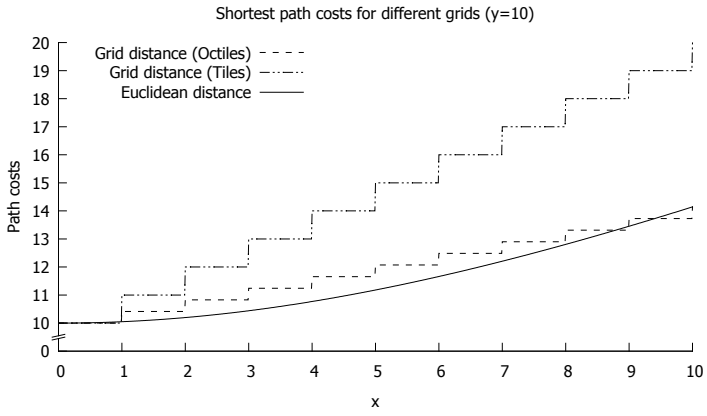


Figure 13: Graph of the Euclidean distance and grid distance for $y = 10$ and $x = [0 : 10]$ for tiles and octiles.

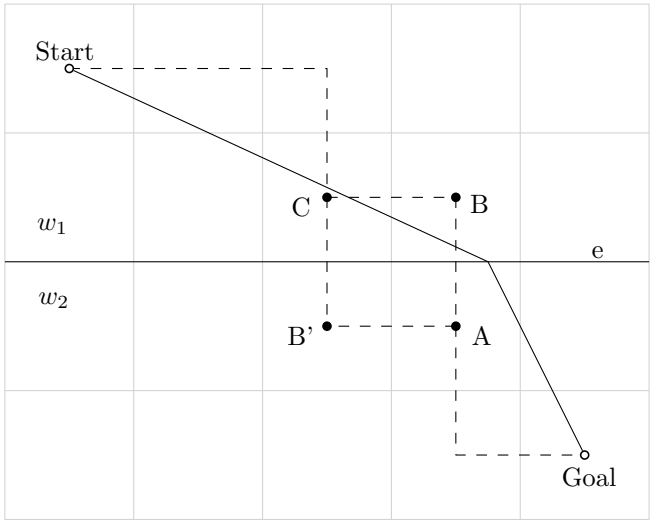


Figure 14: The shortest path (straight line) and grid path (dashed line) among two weighted regions and their boundary edge e, where $w_1 < w_2$. Diagonal lines are not allowed when using tiles, so there are two possible paths that replace AC: ABC and $AB'C$.

3.1.10 Conclusion

In this section, we have analyzed the ϵ-value of grid approaches. This value expresses an upper bound on the approximation of the shortest path in comparison to the shortest path. Knowing the ϵ-value for grid based search methods allows us to compare these methods with ϵ-approximation methods that are based on a triangulation of the scene. The final ϵ-value always consists of the combination of the segment costs and the link costs. The segments costs define the overestimation of each straight subpath, and the link costs define the additional costs required to link all segments together. The segment costs are static and will always be $\sqrt{4-2\sqrt{2}}-1 \approx 0.0824$. The link costs depend on the resolution of the grid and the weights of the scene and is defined as: $r * \min\left(\frac{\sqrt{2}}{2}w_{max}, \frac{1}{2}w_{min} + \frac{1}{2}w_{max}\right)$, where r is the cell size and w_{min} and w_{max} are the minimum and maximum weight of the scene, respectively. With the resulting ϵ-value for grid approaches, we can now focus on improving the speed of A* algorithms through better heuristics.

3.2 Heuristics for A*

The A* algorithm is a flexible algorithm, because it can use different heuristics for different scenes. The A* algorithm finds a shortest path in a graph faster than or equally fast as Dijkstra's algorithm if an admissible heuristic is used in an unweighted region, as mentioned in Section 2.1.1. An admissible heuristic never overestimates the costs to the goal. However, problems arise when weighted regions are added to the scene. The core of these problems lies with the admissible heuristic. If the Euclidean distance is used as a heuristic, then it can be multiplied with a weight in order to give a better estimation of the distance to the goal. However, the chosen weight can't be higher than the lowest weight, otherwise the heuristic might overestimate the actual cost. Picking the lowest weight as a multiplier will make the A* algorithm behave more like Dijkstra's algorithm, because the g-value could be very high due to regions with very high cost, while the h-value is rather low, because the lowest weight is multiplied with the Euclidean distance. Thus, keeping an admissible heuristic could lead to big underestimations of the actual costs to the goal. It might be tempting to just calculate the

actual costs to the goal, but this negatively influences the performance of the A* algorithm because the heuristic value is calculated for each visited node. Increasing the costs of this calculation from $O(1)$ to anything higher will increase A* running times. However, this is a trade-off. The additional costs for calculating the heuristic could be negated by visiting less nodes because the heuristic is better suited for guiding the A* algorithm. In this section, we will describe several existing heuristics and try to adapt the Hierarchical A* heuristic to the WRP. The heuristics described in this section will be compared through a series of experiments in Section 7.2.

3.2.1 Dijkstra

Using a fixed heuristic of 0 for all nodes lets the A* algorithm behave like Dijkstra's algorithm. From now on we will refer to it as Dijkstra's heuristic. Dijkstra's heuristic will always find the shortest path through a weighted region if it exists. It will be used as a baseline to compare the other heuristics with.

3.2.2 Euclidean distance times lowest weight

The Euclidean distance is the most used heuristic for A* in unweighted regions. However, in weighted regions this heuristic suffers from underestimation, thus making it perform more like Dijkstra's algorithm. A small adjustment can be made by multiplying the Euclidean distance with the lowest weight in the scene. If a scene has weights ranging from 10 to 15, then the Euclidean distance will yield a very underestimated cost to the goal. If we multiply the Euclidean distance with the lowest weight, then we can compensate a bit. However, we can't pick any weight higher then the lowest value because then the heuristic will overestimate the costs to the goal in regions with the lowest weight leading to non-optimal paths.

3.2.3 Hierarchical A*

As mentioned in Section 2.1.1, Hierarchical A* [16] uses a heuristic that creates several abstraction layers from which better heuristics are derived. This will lead to increased running times for the heuristic calculation, but the better guidance should speed up the search. The abstraction layers are only defined for unweighted regions. In an unweighted region, the abstraction layers are created by taking the nodes with the highest number of free nodes around it and grouping them with all nodes within a certain distance until all nodes are grouped. This process is repeated until there is a top layer with just a single node. The abstraction layers and the extraction of heuristic values from this hierarchy of abstraction layers will be discussed.

Abstraction layers The hierarchy of Hierarchical A* consists of one or more abstraction layers. Each abstraction layer holds an abstract representation of the scene of a certain size. The higher one goes in the hierarchy, the smaller the abstract representation of the scene will become. The difference in size between two abstraction layers is controlled by an *abstraction factor*. The size of abstraction layer k can be found by dividing the width and height of abstraction layer $k-1$. Each block of *abstraction factor***abstraction factor* nodes on abstraction layer $k-1$ will refer to a single node on abstraction layer k. Whereas the abstraction factor in the original Hierarchical A* for unweighted regions was fixed to the value of 3, the abstraction factor is now a parameter of the Hierarchical A* algorithm. In Section 7.2.1, we will analyze the impact of the abstraction factor by comparing different values. Higher abstraction factors will speed up the construction of the hierarchy, and heuristic values will be computed faster because the abstraction layers are smaller. However, the heuristic values might be inaccurate due to the high level of abstraction and thus lead to longer A* searches. The costs of travel

between two nodes in an abstraction layer is equal to the costs within the layer times the abstraction factor. In Section 7.2.1, we will compare different approaches to the creation of the abstraction layers.

Finding heuristic values Once the abstract hierarchy is constructed, heuristic values can be retrieved from it. If a heuristic value is requested for a certain node, then a search is started in the first abstraction layer to find the costs to the goal. This search again requires heuristic values, which it will retrieve from the abstraction layer above the current layer. This process is repeated on each layer, until the top layer is reached. On this layer, paths are planned using Dijkstra's heuristic. This process leads to a high number of A* executions to find the shortest path from several starting positions to the same goal position. Fortunately, we can use previously found shortest paths (heuristic values) in later searches. Several caching techniques have been devised by the authors of the Hierarchical A* method [16] that can reduce the number of A* executions. These methods will be discussed in the following paragraphs.

Several nodes on abstraction level k direct to the same node on level $k+1$. If heuristic values are requested for these nodes on level k, then Hierarchical A* will compute the shortest path from the same start and goal position on abstraction level $k+1$ several times. We can prevent repeating the same search by caching the heuristic values that are computed. We can store the heuristic value together with the weight value if we create a grid of pairs on each abstraction layer. If a heuristic value is requested, then the grid can be efficiently queried to see if the value is already known. This caching method is called V0 caching, and it will reduce the number of searches by a factor that is equal to the abstraction factor.

Further improvements can be made through the use of optimal path caching. This method, also known as V1 caching, caches the heuristic values of all nodes that lie on an optimal path to the goal. If an optimal path is found for a start and goal position, we also know the optimal costs to the goal for each node that lies on the optimal path to the goal. If we also store this information, then we can use it to our advantage in later searches because all searches lead to the same goal position. We can use the previously found heuristic values instead of requesting heuristic values one abstraction layer higher. These heuristic values can give better guidance to the current search and reduce the running time. Although this caching method has a high probability of reducing the number of nodes that are explored in subsequent searches, it has a chance to compute suboptimal paths due to the big difference in heuristic values between nodes that lie on the path and nodes that lie just next to the path.

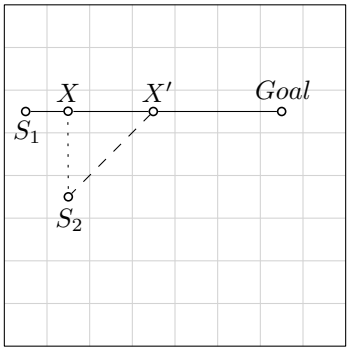


Figure 15: An optimal path is shown from S_1 to the goal position together with the starting point of a new search S_2. If Dijkstra's heuristic is used, then the search will first encounter the previously found optimal path at X. Combining paths S_2X and $XGoal$ will not return an optimal path. An optimal path can be found by connecting S_2 via X'.

A better use of optimal path caching is done with V2 caching. V2 caching also stores the heuristic

values of all nodes that lie on previously found optimal paths, but uses them to connect the new search paths to previously found optimal paths. We can't just blindly terminate the search once we reach a node that lies on a previously found optimal path because the combination of the currently explored path and the previously found path might not be optimal, as shown in Figure 15. We can, however, connect the paths and use the combined path as a candidate for the optimal path. Let X be the node that is discovered that already lies on an optimal path to the goal found in a previous search. We can add the goal position to the open list with an f-value that is equal to the g-value plus the heuristic value of X. Furthermore, we don't have to explore the nodes around X because X is already on the optimal path. This will make the previously found optimal path to the goal a border for the exploration of the current search and thus reduce the number of nodes explored, as shown in Figure 16 and 17.

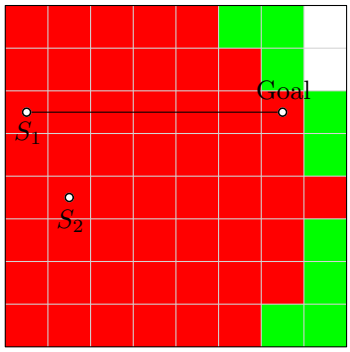


Figure 16: An optimal path is shown from S_1 to the goal position together with the starting point of a new search S_2. The exploration of Dijkstra's algorithm with start position S_2 is shown with the open nodes (green), closed nodes (red) and undiscovered nodes (white) at the termination of the search.

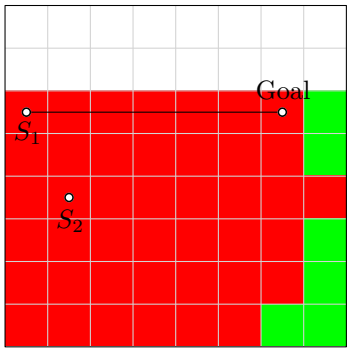


Figure 17: An optimal path is shown from S_1 to the goal position together with the starting point of a new search S_2. The exploration of Dijkstra's algorithm with start position S_2 is shown where optimal nodes are not further explored, with the open nodes (green), closed nodes (red) and undiscovered nodes (white) at the termination of the search.

3.2.4 Grid lookup

As mentioned in Section 3.2.2, the Euclidean distance could underestimate the costs because it has no knowledge of the weights. The grid lookup method tries to improve the Euclidean distance heuristic by summing up the cell costs of all nodes in a straight line between the current node and the goal node. This will return the actual costs of the straight line path from the current node to the goal node. The extra computational costs required by looking up the values could be negated by the better guidance that is given by this heuristic and thus less nodes will be explored. However, this heuristic leads to local admissibility instead of global. If the start and goal position lie within the same region, then the resulting path will be the straight line between these two points, because the heuristic never looks further. A possible shortest path via a surrounding region with a much lower weight will always be overlooked by this heuristic.

3.2.5 Conclusions

In this section, we have analyzed several heuristics for A*. We discussed the impact of weighted regions on the performance of A* due to the underestimation of common heuristics, such as the Manhattan

distance and Euclidean distance. Furthermore, we adapted the Hierarchical A* method to work with weighted regions. We discussed the creation of abstraction layers and how the heuristic values can be retrieved efficiently from the abstraction hierarchy. Finally, we devised a new method call Grid Lookup.

4 ϵ-approximation approaches

In this section, we will look at ϵ-approximation approaches to solve the WRP. These methods discretize the environment to find an approximation of the shortest path. The approximation is bounded by the parameter ϵ. The resulting shortest path of an ϵ-approximation method is always shorter than $1 + \epsilon$ times the shortest path. First we will dive into the details of the ϵ-approximation method as defined by Aleksandrov, Lanthier and Maheshwari [5] and a graph search method especially designed for the WRP called BUSHWHACK [11]. Both these methods have been implemented in our framework. We will also look at the bounds of the ϵ-value. The bounds are required to compare the ϵ-value of the ϵ-approximation methods with the ϵ-value of the A* grid approaches.

4.1 Steiner points

The Steiner points approach from Aleksandrov, Lanthier and Maheshwari [5] uses a triangulation of the scene with assigned weights. Steiner points are added to the edges of the triangulation with a logarithmic distribution. Placing a logarithmic number of Steiner points on each edge will ensure that any path p between two adjacent edges can be approximated within the ϵ-bound by the paths between the Steiner points left and right of p. Using a logarithmic distribution leads to problems near the vertices of the triangulation. It yields infinitesimal intervals between the Steiner points and thus an infinite number of Steiner points. A *vertex vicinity* is defined to solve this problem. The vertex vicinity is a polygon around the vertex that is void of Steiner points. The vertex vicinity $S(v)$ is visualized in Figure 18. For any vertex v, the radius of the vertex vicinity $r(v)$ is defined as: $r(v) = \epsilon d(v)$, where $d(v)$ is the shortest distance from the vertex v to the boundary of the union of all faces incident to v.

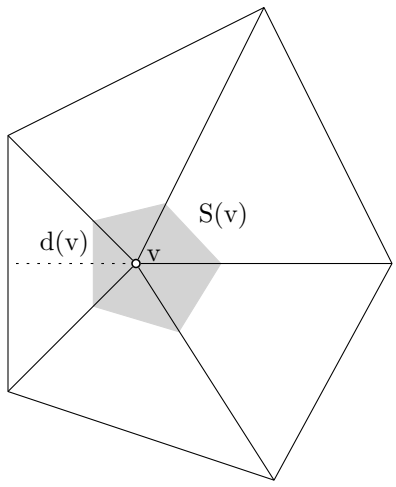


Figure 18: The vertex vicinity $S(v)$ of vertex v. The dashed line represents $d(v)$.

Given the vertex vicinity the Steiner points can be added to the edges of the triangulation. For each vertex v, Steiner points will be added on all edges adjacent to v. First we will define α_v to be the minimum angle between any pair of adjacent edges that are incident to v. The first Steiner point s_1 that will be added to edge e is always placed exactly on the border of the vertex vicinity, and thus $|vs_1|$ is equal to $r(v)$. Let $\lambda = 1 + \epsilon * \sin \alpha_v$, if $\alpha_v < \frac{\pi}{2}$ or $\lambda = 1 + \epsilon$ otherwise, and let $\mu = \log_\lambda(\frac{|e|}{r(v)})$. Following Steiner points will be added according to $|vs_i| = r_v \lambda^{i-1}$ for $s_2, s_3, ..., s_{\mu e-1}$. This strategy will create Steiner point sets for every vertex-edge pair and thus generate two sets for each edge. These two sets are merged by a single rule. If the intervals induced by two pairs of Steiner points overlap,

then the the largest interval is removed.

Once all Steiner points have been added, a graph can be created. All Steiner points and vertices within each triangle are connected to each other to form the graph. The graph will consist of $|V| = nm$ vertices and $|E| = nm^2$ edges, where n is the number of faces in the original triangulation and m is the total number of Steiner points. The total number of Steiner points is defined by $m = O(\log_\delta(\frac{|L|}{r}))$, where $|L|$ is the length of the longest edge, r is ϵ times the minimum $d(v)$ for all vertices v. Furthermore, $\delta \geq 1 = \sin(\theta)$, where θ is the minimum angle between any two adjacent edges of the triangulation. Once the graph is constructed, any graph search method can be used to find the ϵ-approximation of the shortest path.

4.2 BUSHWHACK

Several approaches to the WRP are based on a discretization of the scene using Steiner points. A graph can be created from these points that can be used to compute optimal paths. However, information is lost during the conversion to a graph. The BUSHWHACK [11] method preserves the geometrical properties of the scene by dynamically creating a new datastructure. This method uses a 'lazy' and best-first propagation scheme to find an optimal path while exploring less edges than other graph search methods. The new datastructure is a subgraph of the total graph and is created in a 'lazy' way: Edges are only added to the subgraph when they are needed. The algorithm finds a shortest path in the subgraph in $O(nm \log nm)$, where n is the number of faces in the triangulation of the scene and m is the total number of Steiner points added to the triangulation.

The geometrical properties that are preserved by this method yield a reduction in the number of edges that need to be explored in the graph. The most important property is that any two optimal paths in the graph with the same source point cannot intersect in the interior of any region. This property is derived from the unweighted path planning problem, where any two shortest paths with the same source can never intersect. Each triangle has a uniform weight and can be treated as a unweighted path planning subproblem, in which intersecting optimal paths cannot exist. With this property, we can significantly reduce the number of graph edges that are created when connecting the Steiner points. A new datastructure called *Interval* is introduced to keep track of feasible graph edges. An interval is defined by two edges (e, e') of the triangulation and a point v on e. The interval consists of all points $v*$ on e' where the costs from s to v plus the costs from v to $v*$ is optimal, where s is the start position. With this interval definition, we can partition the Steiner points on each edge into several intervals of consecutive points, as shown in Figure 19.

The interval creation is a 'lazy' process, and intervals are only generated when new points are discovered. The first point that is discovered on an edge of a triangulation will create an interval to the other two edges, where each interval will contain all points on that edge. As soon as a second point is discovered, the intervals are adjusted in such a way that all intervals will obey the above mentioned rule. This process is visualized in Figure 20. In order to quickly retrieve candidates for each interval, the intervals need to be sorted based on path costs. Each interval is split up in two *monotonic* intervals. The splitting point is the perpendicular point of v_1 on e' as illustrated in Figure 21. Having monotonic intervals will allow us to find the optimal path within an interval in $O(1)$ time.

Once a new set of intervals is created for a newly discovered point, there are also a couple of non-interval paths that need to be considered. Intervals only consider face-crossing paths, but the solution to a WRP can consist of face-crossing and edge-crawling paths. These edge-crawling paths are added manually next to the intervals. There are three types of edge-crawling paths, and they are visualized in Figure 22. The first type is a path that connects two neighbouring Steiner points on the same edge within an interval. The second type is a path that connects a Steiner point within an interval to a Steiner point outside of the interval. The final edge-crawling path is a path that connects a vertex to the closest Steiner point on an edge. The candidates of the intervals and the non-interval

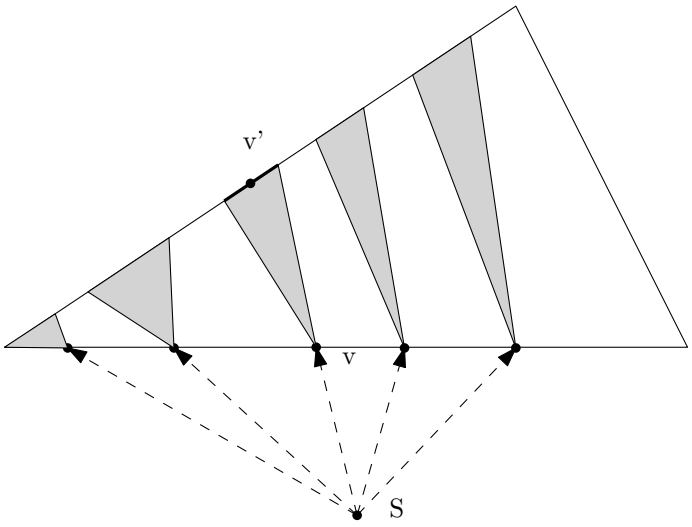


Figure 19: The BUSHWHACK algorithm uses intervals to prune the search.

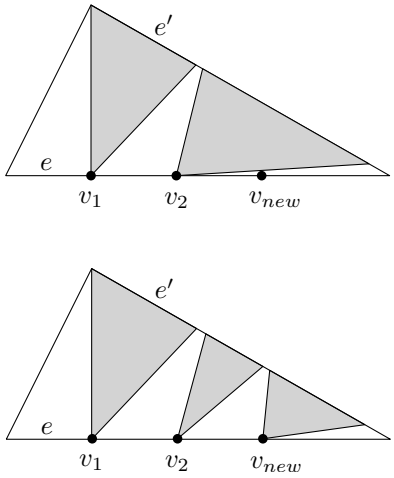


Figure 20: Adding a new interval to an edge.

paths are added to a priority queue that sorts them based on their costs to the start position. At every iteration, the shortest path will be picked from the priority queue and will be further explored until the goal is found.

The BUSHWHACK search method is a graph-search method that only plans from one node to another. In practice, paths are planned from arbitrary start and goal positions. In order to let the BUSHWHACK method cope with arbitrary start and goal positions, new nodes have to be created for those points. We will create these new nodes by creating vertices for the start and goal position and connecting these vertices to the three vertices of the triangle that contains the new vertex. The triangulation will be altered before the Steiner points are added. The downfall of this approach is that the altered triangulation cannot be used for the next search because then the number of edges and nodes will increase with every search. After a search is finished, the triangulation will be restored to its original state.

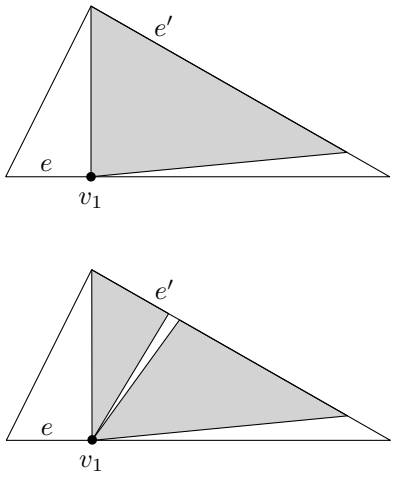


Figure 21: Splitting an interval into monotonic intervals. The splitting point is the perpendicular point of v_1 on e'.

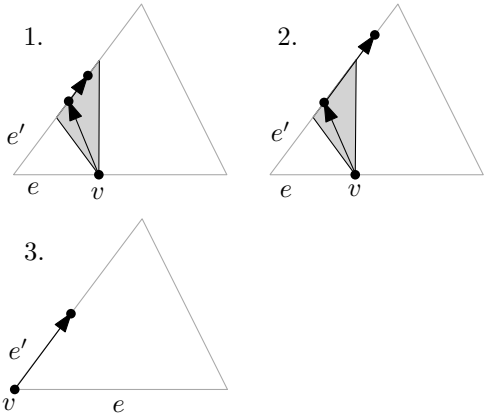


Figure 22: The three types of edge-crawling paths. 1: An edge-crawling path where the start and end point lie within an interval. 2: An edge-crawling path where the start point lies within an interval and the end point lies outside of the interval on edge e'. 3: An edge-crawling path starting from a vertex.

4.3 Upper bound on ϵ

To effectively compare A*-grid approaches with ϵ-approximation approaches, we have to match up their ϵ-values. We have to know the upper bound on the ϵ-values in order to determine our playing field. The lower bound is clearly given in all papers: $\epsilon > 0$. This follows from the definition of ϵ: the length of the approximated path is always shorter than $1 + \epsilon$ times the length of the optimal path. For $\epsilon = 0$ the approximated path is equal to the optimal path. An upper bound is not explicitly mentioned in most papers and will be researched in this chapter.

4.3.1 Steiner points

The vertex vicinity holds the key to finding the upper bound on the ϵ-value. The radius of the vertex vicinity is defined by ϵ, and if the vertex vicinity is too large then it will overlap with vertex vicinities

of other vertices. Overlapping vertex vicinities are not allowed because otherwise there may be triangle edges without any Steiner points on it. The vertex vicinity $S(v)$ is visualized in Figure 18. For any vertex v, the radius of the vertex vicinity $r(v)$ is defined as: $r(v) = \epsilon d(v)$, where $d(v)$ is the shortest distance from the vertex v to the boundary of the union of all faces incident to v. The maximum ϵ-value can be found in the case where the vertex vicinities of two vertices that share an edge exactly touch each other and the edge will contain exactly one Steiner point on the intersection. The size of the vertex vicinity is controlled $d(v)$. Let $d(v)$ be equal to $d(v_2)$ for vertex v and a neighbouring vertex v_2 that are connected through edge e. As such $d(v)$ and $d(v_2)$ are equal to the length of e. The vertex vicinities of v and v_2 will exactly touch if the radius of both vertex vicinities are equal to $\frac{|e|}{2}$, where $|e|$ is the length of e. If we substitute the variables in the formula of the radius of the vertex vicinity $r(v)$, then we will end up with: $\frac{|e|}{2} = \epsilon|e|$. This leads to an ϵ-value of $\frac{1}{2}$. Higher values for ϵ could lead to overlapping vertex vicinities. Thus, we can conclude that for this method it holds that $0 < \epsilon \leq \frac{1}{2}$.

4.3.2 Steiner points on bisectors

The Steiner points on bisectors approach from Aleksandrov, Maheshwari and Sack [6] places Steiner points on the bisectors of the triangulation of the scene, as shown in Figure 23. These Steiner points are connected to find a graph that can be used to find ϵ-optimal paths. Similar to the Steiner points on the triangle edges, they define a new vertex vicinity. Their vertex vicinity is defined as: $r(v) = \epsilon \frac{w_{min}}{7w_{max}} d(v)$, where w_{min} and w_{max} are the minimum and maximum weight of the regions incident to vertex v, respectively. We know from the previous section that $r(v)$ can not be greater than $\frac{1}{2}d(v)$. If w_{min} and w_{max} are both very high, then we will end up with $r(v) = \frac{\epsilon}{7}d(v)$. Resulting in a maximum value of $\epsilon = 3\frac{1}{2}$. Higher values of ϵ will lead to overlapping vertex vicinities. We can therefore conclude that ϵ should be between 0 and $3\frac{1}{2}$ for this method.

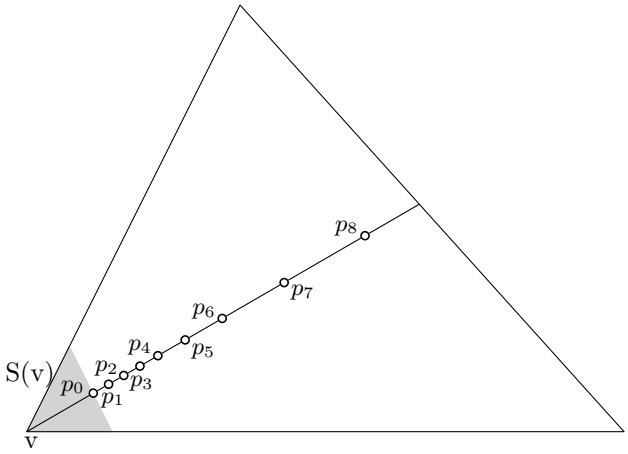


Figure 23: The placement of Steiner points a bisector of the triangulation.

4.4 Conclusions

In this section, we analyzed ϵ-approximation approaches. We discussed a method that shows how Steiner points can be added to a triangulation. A graph can be constructed from these Steiner points, that can be queried with any graph-search algorithm to find ϵ-optimal paths. We discussed the BUSHWHACK graph-search method, and how it utilizes the interval datastructure to improve upon other graph-search algorithms. Next we analyzed the bounds of the ϵ parameter that is used by these

ϵ-approximation approaches. These bounds are required if we want to compare A*-grid approaches with ϵ-approximation approaches. For the method that places Steiner points on the edges of the triangulation it holds that $0 < \epsilon \leq \frac{1}{2}$. For the method that places Steiner points on the bisectors of the triangulation it holds that $0 < \epsilon \leq 3\frac{1}{2}$. In section 7, will compare the ϵ-value of grid-based approaches with ϵ-approximation approaches.

5 Hybrid method

In this section, we will present new methods to solve the WRP. These methods use A* grid results to limit the search space for ϵ-approximation methods. This will allow us to compute ϵ-approximations of optimal paths faster than the current methods. Two of these hybrid methods have been developed. The first method uses A* grid results to prioritize the search space. However, this method does not always find the optimal path in the graph. In the upcoming section, we will discuss the idea and why it fails to find the optimal path in the graph. Afterwards, we will discuss the second approach that uses A* grid results to prune the search space. If we can decrease the size of the search space then we can lower the worst case running times of the graph-search. We will present three sub-methods that all prune the search space in a different way. Because both methods don't interfere with each other they can also be combined into one method using both mechanisms. Although this is currently of no use because the first method does not return ϵ-optimal paths, if such a method would exist it can be efficient to combine the two methods. For instance, instead of prioritizing the entire search space one could first prune the search space and then prioritize the pruned search space.

5.1 A* guided search

When searching in a huge space for ϵ-optimal paths, the search can be improved by better guidance. If the direction to follow is known at each node in the graph, then the number of nodes explored does not have to be large in order to find a ϵ-optimal path. Because the A* grid method works fast and has a adjustable grid size, we can create a coarse path to the goal and then use this path to guide the graph-search. This idea is used in the weighted Hierarchical A* method that was devised in Section 3.2.3. If we search the graph with A* and use the Hierarchical A* heuristic, then we will plan a coarse grid path that can be used to guide A* through the graph with heuristic values. However, the ϵ-value of the grid-based search performed by Hierarchical A* tends to be higher than the ϵ-value of the ϵ-optimal paths that can be retrieved from the Steiner graph. The path found by Hierarchical A* can be more expensive than the actual costs and thus overestimate the costs to the goal. This overestimation makes Hierarchical A* an inadmissible heuristic. We can compensate the overestimation by looking at the difference in ϵ-values between the grid-based and Steiner graph-based methods. If the ϵ-value for a grid is 1.5 and the Steiner set is created with $\epsilon = 0.5$, then we know that the maximum overestimation of the grid-based results in comparison to the graph-based results will be equal to: $2.5/1.5 = 1\frac{2}{3}$. We can divide all heuristic values by $1\frac{2}{3}$ to prevent overestimation of the heuristic. Separate values can be calculated for each environment, grid settings and graph settings.

Lowering the heuristic value has several side-effects. If the difference between the ϵ-values becomes bigger, the compensation factor grows and can lead to very small heuristic values. These values will no longer be able to guide the A* algorithm. If the heuristic values are very small in comparison to the g-values, then the A* node selection will mostly be based upon the g-values. The A* exploration with low heuristic values will perform more like Dijkstra's algorithm and thus no longer provide guidance to decrease the number of explored nodes.

The second problem that arises is that the guidance remains unchanged if all heuristic values are decreased by a certain compensation factor. The values might not overestimate the costs, but nodes that lie near the grid path will always be given a lower heuristic value when compared to the surrounding nodes. These node will get a higher priority and will be selected earlier than the surrounding nodes. At the time that the surrounding nodes are explored that might lead to a shorter path, the nodes close to the grid path are already closed and newly found paths can't be connected to closed nodes.

Although the grid paths gives us a good approximate path to the solution, using it as a guidance to find a optimal path in a Steiner graph will not lead to optimal paths. In situations in which the grid path and the optimal graph path do not overlap, the heuristic values generated by the grid path will let the A* algorithm first explore the graph-nodes closest to the grid path. When the search reaches

the nodes that lie on the optimal graph path, the nodes previously selected by A* are already closed and their parents can not be altered. This process is visualized in Figure 24.

The authors of the Hierarchical A* paper [16] state that the heuristic given by the Hierarchical A*

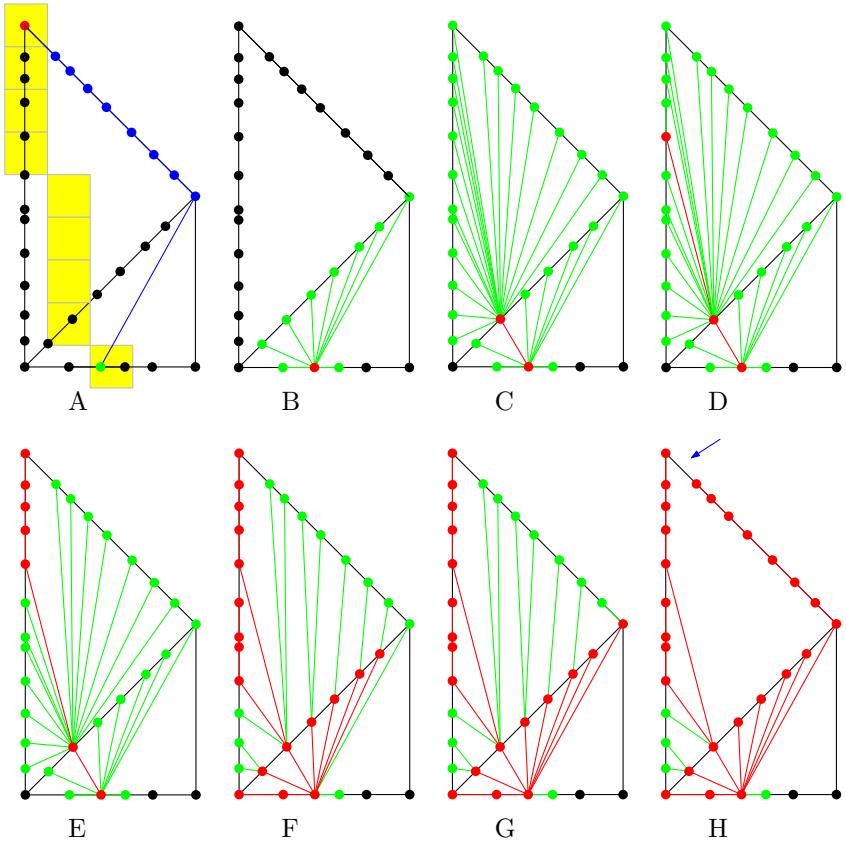


Figure 24: Several steps of the A* exploration. A) shows the properties of the exploration with a start position (green), a common point of the optimal grid path and optimal graph path (red), the optimal grid path (yellow cells) and the optimal graph path (blue). B,C,D,E) exploration of the graph with the Hierarchical A* heuristic, where the open set is green and the closed set is red. F) The state of the exploration several steps later. G,H) The last exploration steps that follow the optimal graph path. In the end, A* can not connect this shorter path to the top node, because it is already closed as shown by the blue arrow.

method is not monotone. A heuristic is monotone, if the difference in heuristic values between any two neighbouring nodes is never larger than the costs to move between these two neighbours. Furthermore the heuristic values are always increasing when nodes are further away from the goal position. An admissible heuristic is always monotone, but a monotone heuristic does not have to be admissible. The Hierarchical A* method can handle this issue on non-weighted regions because the homotopy of the paths on the hierarchical abstraction layers is equal to the homotopy of the optimal path.

Because the A* guided search is very sensitive to small differences in the homotopy of the grid path and optimal graph path, the method is very likely to return non-optimal paths. As a result, the A* guided search method is unable to compete against the existing methods.

5.2 A* pruned search

The second method attempts to decrease the running times of the search by pruning the search space. The graphs that are created from the Steiner points can contain thousands of nodes even for small environments, while the optimal path only uses a small number of them. If we only add Steiner points within the proximity of the optimal path, then we can significantly reduce the number of nodes in the graph. We can efficiently find a suboptimal path to the goal by using A* on a grid. The following example will be shown on one of the scenes that is used in the experiments. The scene is called Puddle and contains several polygons with weights ranging from 1 (gray region) to 20 (dark blue region). An example grid path on the Puddle scene is shown in Figure 25 together with the grid path overlapping a triangulation of the Puddle scene. If we only add Steiner points in the proximity of this path, then we will severely prune the graph, and this will speed up the search. The proximity can be defined by the difference in ϵ-value between the grid and the graph to insure that the ϵ-optimal path in the graph does not get pruned. However, in practice it is inefficient to calculate the distance of each Steiner point to the suboptimal path because there could be a huge number of Steiner points. We have devised several methods that all define the proximity to the suboptimal path in such a way that the graph can be pruned efficiently.

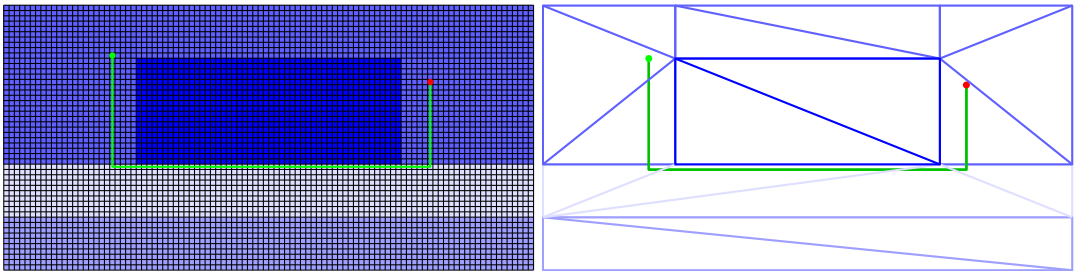


Figure 25: On the left, we display the resulting path (green) of an A* search on a grid of the Puddle scene with a green start position (20,40) and red goal position (80,35). The right side shows the grid path on the triangulation of the Puddle scene.

5.2.1 Triangle intersection based

The triangle intersection based pruning method (TIB) defines the proximity to the suboptimal path as a sequence of triangles. The pseudocode is shown in Algorithm 2. All triangles that are intersected by the suboptimal path will be added to the pruned graph triangle set. Steiner points will only be added onto the edges of the triangles in the set. The sequence of triangles will follow the homotopy of the suboptimal path and will find the ϵ-optimal path in the graph that follows the same homotopy. An example of the intersected triangle sequence is shown in Figure 26. Although grids can have a different homotopy than the original scene, due to the loss of information at low resolutions, it is always possible to find a resolution that maintains the homotopy of the original scene.

Algorithm 2 Triangle intersection based pruning(*gridPath*, *triangles*)

```
1: for all edges g in gridPath do
2:    for all triangles t in triangles do
3:       for all edges e in t do
4:          if g intersects e then
5:             Add t to intersectedTriangles
6:          end if
7:       end for
8:    end for
9: end for
10: for all triangles t in intersectedTriangles do
11:    for all edges e in t do
12:       Add e to result
13:    end for
14: end for
15: Return result
```

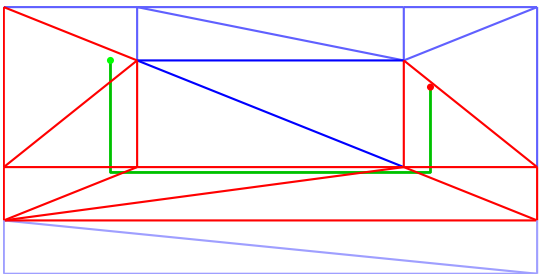


Figure 26: The sequence of intersected triangles (red) resulting from an A* grid search with triangle intersection based pruning.

5.2.2 Vertex based

The vertex based pruning method (VB) aims at identifying the areas that contain the bending points of the optimal path. These areas are identified by looking at the bending points of the suboptimal path. Steiner points are always placed with a logarithmic distribution, starting from a vertex on each edge, because a lot of possible bending points lie near the vertices. Therefore, the idea is to select the nearest vertices for each bending point of the suboptimal path. The pseudocode for vertex base pruning is shown in Algorithm 3. We will add Steiner points to all edges incident to these nearest vertices.

This will result in several disjoint sets of Steiner points near the bending points of the suboptimal path. If long straight segments occur in the suboptimal path, then these sets might be disconnected. To handle long straight segments, we add Steiner points to each intersected triangle edge. This will guarantee that the pruned graph is connected. An example of the selected edges is shown in Figure 27. The selected set of edges captures the homotopy of the suboptimal path with additional edges in the proximity of the bending points of the suboptimal path, and will find the ϵ-optimal path in the graph that follows the same homotopy.

Algorithm 3 Vertex based pruning($gridPath$, $triangleEdges$, $triangleVertices$)

```
for all bending points b of the gridPath do
    Find the set closestV of closest vertices from b in triangleVertices
    for all vertices v in closestV do
        for all edges i incident to v do
            Add i to result
        end for
    end for
end for
for all edges g in gridPath do
    for all edges e in triangleEdges do
        if g intersects e then
            Add e to result
        end if
    end for
end for
Return result
```

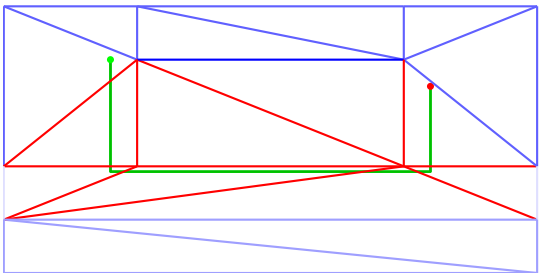


Figure 27: The selected set of edges (red) resulting from an A* grid search with vertex based pruning.

5.2.3 Closest edge based

Closest edge based pruning (CEB) is the last method we present in this section. Calculating the distance from the suboptimal path to all Steiner points might be inefficient, but we can calculate the distance from the suboptimal path to all the edges in the triangulation. If we can create a set with all edges in the triangulation that are at some point closest to the suboptimal path, then we will ensure that the graph will always contain the Steiner points that lie closest to the suboptimal path. An example of the selected edges is shown in Figure 29 on the left side.

In our first approach, we use the cellsize of the grid as stepsize through the suboptimal path and find the nearest edge in the triangulation at each step. It is possible that there are several nearest

edges with the same distance. This is the case, for instance, if the suboptimal path goes through a vertex of the triangulation. An arbitrary number of edges could spawn from this vertex, and in this case all edges incident to the vertex will be added to the set of nearest edges. This set will be a subset of the set of triangles edges obtained in Section 5.2.1, and the search for this set of edges can be improved by starting of with just the set of triangles edges instead of all edges in the scene. Once the set of nearest edges is obtained, we will connect the Steiner points and vertices of all edges that share a triangle. If there are edges that do not have a common triangle, then we only connect the Steiner points on that edge with each other to allow edge-crawling paths. Although this approach yields connected graphs, optimal paths might be lost if we only select the nearest edge(s). As shown in Figure 28, if several edges spawn from one vertex and the suboptimal path just misses the vertex, then several edges will not be selected. The ϵ-optimal path has a high chance to use any of these edges because of the difference in ϵ-optimality between the grid path and Steiner-graph path.

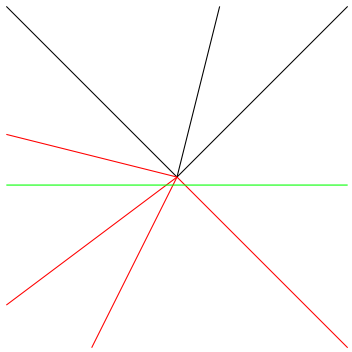


Figure 28: Although all edges lie very close to the suboptimal path (green), only the closest edges (red) will be selected.

The second approach deals with this case by not looking at the closest edge at every step, but by instead including all edges that lie within a certain radius r, as shown in Algorithm 4. The number of checks required to find all edges within the radius is equal to finding the edge(s) that lie closest to the suboptimal path. The homotopy of the suboptimal path is converted into a set of edges that refer to the closest Steiner points, and will find the ϵ-optimal path in the graph that follows the same homotopy. To keep this method from selecting too many edges, the size of the radius should be kept small. A value of 5 is suitable for the scenes in our experimental framework because the lengths of most of the triangle edges range from 10 to 100. If we encounter smaller triangle edges, then the area is most likely crowded with small triangles (for instance around the fallen trees in the Forest scene, as shown in Figure 32) and picking a lot of triangle edges there makes sense. The first approach uses the set of triangle edges obtained in Section 5.2.1 to speed up the search, but the second approach could select edges outside of this set. As a result, radius-based edge selection must consider all edges in the triangulation. The adjusted set of edges is shown in Figure 29 on the right side.

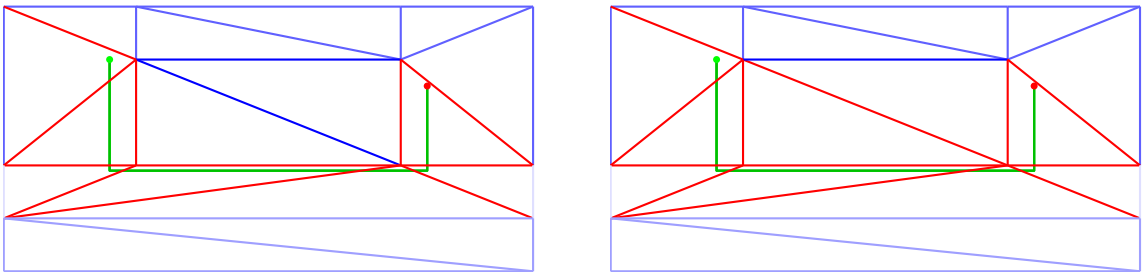


Figure 29: Left: The selected set of edges (red) resulting from an A* grid search with closest edge based pruning that only selects the closest edges. Right: The selected set of edges for closest edge based pruning that selects all edges within a radius of 5.

Algorithm 4 Closest edge based pruning(*gridPath*, *triangleEdges*, r, *stepsize*)

```
1: for all points p of the gridPath with distance stepsize between them do
2:   for all edges e in triangleEdges do
3:     if minimum distance between e and p < r then
4:       Add e to result
5:     end if
6:   end for
7: end for
8: Return result
```

5.3 Conclusions

In this section, we presented several new methods that utilize suboptimal grid paths to prioritize or prune the search space for ϵ-approximation methods. We presented an A* guided search method that generates heuristic values from the A* grid path to guide the search through the Steiner graph. However, this method is prone to small differences between the homotopy of the A* grid and ϵ-optimal path. Small differences in the homotopy can lead to overestimation of the heuristic values, and thus lead to non-optimal paths. As such, this method is unable to compete against the current ϵ-approximation methods. Next, we presented the idea to use the A* grid path to prune the search space. By selecting a subset of all the Steiner points in the scene we can reduce the size of the Steiner graph. Three approaches have been devised that all capture a subset of the Steiner graph according to different definitions of the proximity to the suboptimal path. Triangle intersection based pruning selects all triangles that are intersected by the A* grid path. Vertex based pruning selects all edges incident to the closest vertices of the bending points of the A* grid path along with all triangle edges that are intersected by the A* grid path. Closest edge based pruning selects all edges that lie within a certain distance r of the A* grid path. A comparison between the different A* pruned search methods is made in Section 7.3.

6 Implementation notes

In this section, we will discuss several implementation decisions and problems encountered during the implementation of the algorithms. All algorithms are implemented within the ECM Framework. The ECM framework is created by Geraerts and van Toll and is able to run crowd simulations using the Explicit Corridor Map [29]. It supports interchangeable path planning algorithms for each character in the simulation. The following path planning methods have been implemented: A* on a grid, A* on a graph, BUSHWHACK, and the three pruned graph methods: triangle intersection based pruning, vertex based pruning and closest edge based pruning. Both A* methods can use several heuristics: Dijkstra, Manhattan distance, Euclidean distance, Grid Lookup, and Hierarchical A*. The graph of Steiner points is created with a logarithmic distribution as described in Section 4.1.

6.1 Multi-layered A* Grid

The ECM framework is able to handle multi-layered environments as defined in [50]. Multi-layered environments consist of a set of layers and transfers. Each layer is a collection of two-dimensional polygons that all lie in a single plane and a transfer is defined as a polygon that connects two layers. The newly implemented algorithms should also be able to handle multi-layered environments, and the grid was the most challenging aspect to convert to a multi-layered version. Linking multiple grids induced by the different layers of the environments through transfers was a difficult task. Typical A*-exploration on a grid is done by altering the coordinates of the current node to all surrounding nodes. To this end, a simple subtraction or addition of 1 to the x, y or both coordinates is performed. The first idea was to add extra information to each cell in the grid. In addition to the weight-values, transfer id's could be stored. The downside of this approach is that the total memory consumption could double because each cell needs twice as much bytes. Furthermore, most of these transfer id's are never set because there are typically more cells without a transfer than cells with a transfer. That is why we decided to use a different approach, namely creating a set of *transferrules*. Each transferrule contains a start layer, an end layer, and a set of coordinates for both layers. An A* grid multi-layered method was implemented that checks if the current point is present in any of the transferrules that apply to the current layer. If a corresponding transferrule is found, then the set of coordinates of the end layer are added as candidates. This method performs slower than the regular A* implementation, but it can still be used in real-time.

6.2 BUSHWHACK

Although the principles behind the BUSHWHACK method are intuitive and easy to understand, the implementation was difficult. A doubly-connected edge list (DCEL) was chosen to store the triangulation. Each half-edge stores a sorted list of pointers to the Steiner points that belong to that edge. The first problem was dealing with the vertices of the triangulation. An interval that holds Steiner points on an edge e could possibly also contain the vertices of edge e. These vertices can however also be part of each interval that contains Steiner points on any edge incident to these vertices. Furthermore, the vertex could also be a candidate of a non-interval path from the closest Steiner point on each incident edge. When checking whether a vertex should also be contained in a newly created interval, a large number of such checks to find and compare the candidate is required. The second problem was dealing with the $O(\log m)$ time interval creation, with m being the number of Steiner points on an edge. The authors state that keeping monotonically increasing intervals allows for fast interval creation. This is true if several intervals span the points on an edge because checking the borders of the intervals will return the overlapping intervals, and splitting the intervals can be done in $O(\log m)$ time. However, it is possible that many of the Steiner points are explored by non-interval

paths. In this case, the intervals do not span the entire edge and manual checking is required for all points not included in intervals. A schematic overview of this case is given in Figure 30.

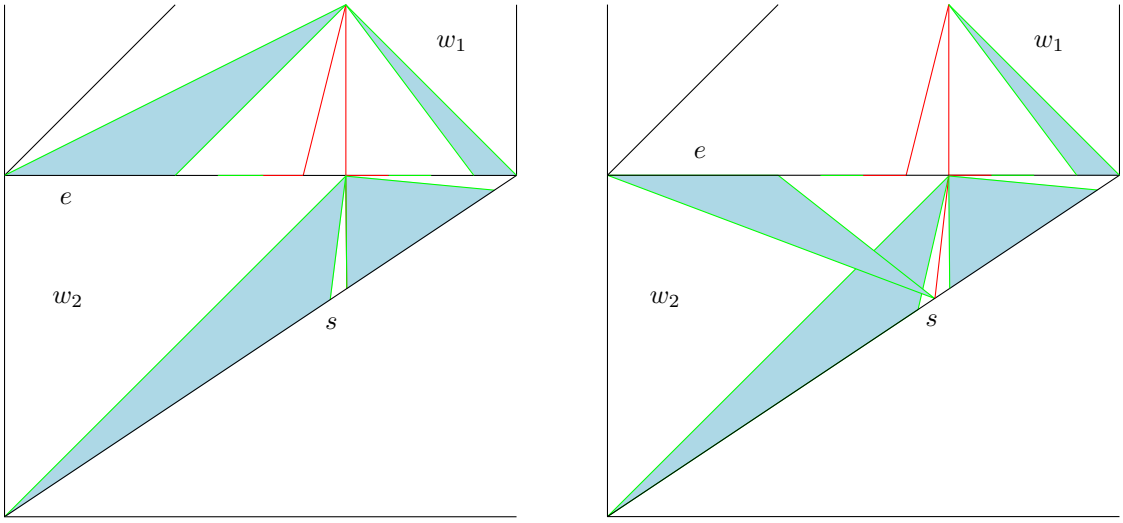


Figure 30: Schematic overview of the creation of a new interval on edge e as spawned from point s. Several edge-crawling paths are already present on edge e due to weight w_1 being higher than weight w_2.

7 Experiments

In this section, we will describe all the experiments that have been conducted for this thesis. We will start with the experiments conducted to compare the different Hierarchical A* abstraction methods. Next we will compare the different heuristics for grid-based methods to see which methods perform well in a weighted grid. After the heuristic comparison, we will compare the different hybrid methods against A* on the Steiner graph. Finally, we will compare the grid and graph methods when they configured to return paths of the same ϵ-optimality to see how big the gap between grid methods and graph methods is. We will also be able to see if the heuristics or hybrid methods can narrow the gap. All our experiments are run on an AMD Phenom II x4 3.4Ghz processor with 4 Gb RAM. All running times are averaged over 50 runs in order to decrease the error in the C++ high precision timer to 0.12 ms.

7.1 Scenes

All experiments have been conducted on five scenes. The first scene is called Puddle, and its size is 100 by 50 meters. The scene is shown in Figure 31 and contains a large puddle (weight 20) in the middle with a forest (weight 5) around it. There is a road (weight 1) from left to right, and beneath the road there is a lane of grass (weight 3). This scene is small and simple, which makes it easy to visually check the computed paths. Furthermore, the high-cost puddle in the middle along with the road next to it shows the standard behaviour of path planning methods in weighted regions: the paths will avoid the puddle and will mostly try to plan paths on the road.

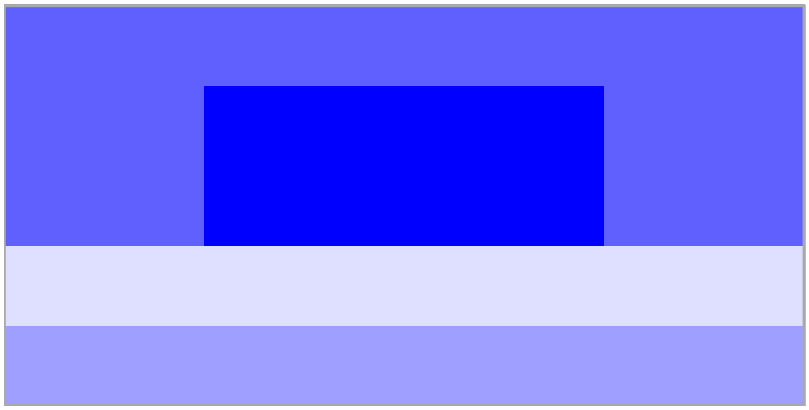


Figure 31: The Puddle scene contains a large puddle (weight 20) in the middle with a forest (weight 5) around it. There is a road (weight 1) from left to right, and beneath the road there is a lane of grass (weight 3). The higher the weight for a particular region, the darker the shade of blue.

The second scene is a large scene with dimensions of 290 by 410 meters. The scene is called Forest and is shown in Figure 32. Forest consists of deep forest (weight 99) at the sides and in the middle of the scene. Through this forest runs a grassy road (weight 3) with several puddles on it (weight 20). On the top, there is an area resembling a panoramic view or a similar attractive spot with a low weight (weight 1), and on the right are some fallen trees (weight 2). This scene is a representation of a real-life scenario, and it is a large scene, which makes it well-suited for showing problems that seem insignificant in smaller scenes. For instance, enlarging small differences in running times and path costs.

The third scene is called Bars and spans an area of 100 by 50 meters. The scene is shown in Figure

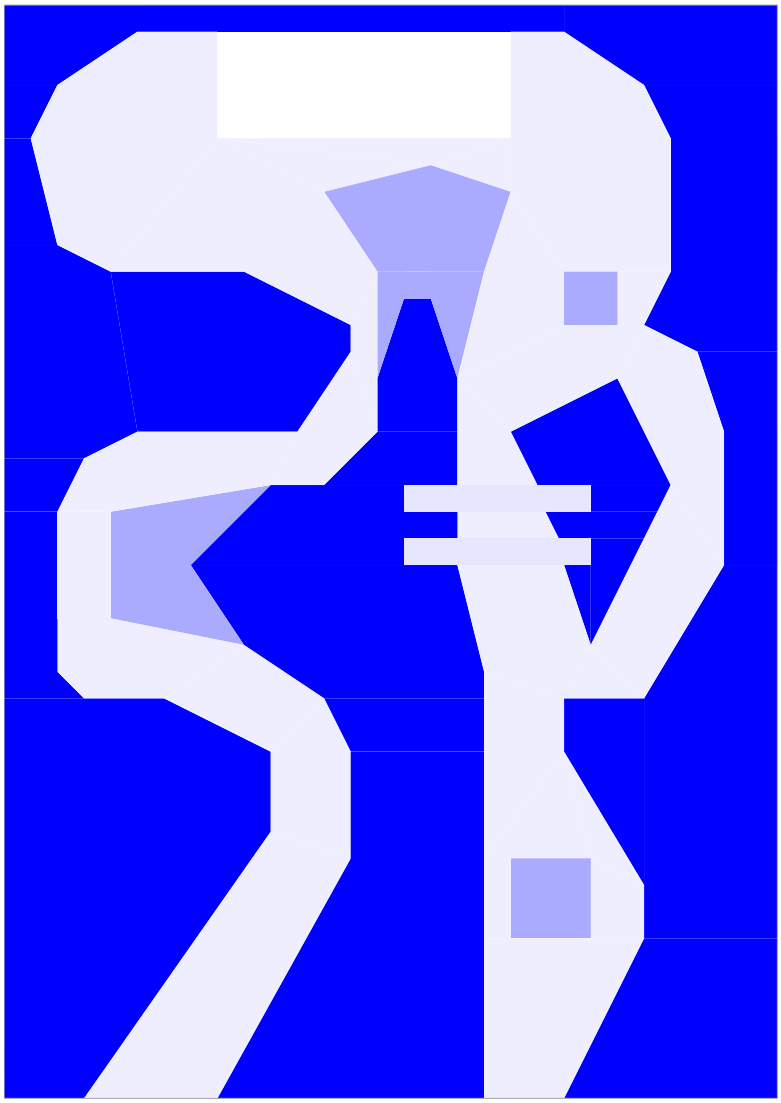


Figure 32: The Forest scene consists of deep forest (weight 99) at the sides and in the middle of the scene. Through this forest runs a grassy road (weight 3) with several puddles on it (weight 20). On the top, there is an area resembling a panoramic view or a similar attractive spot with a low weight (weight 1), and on the right are some fallen trees (weight 2). The higher the weight for a particular region, the darker the shade of blue.

33 and consist of several vertical bars with different weights. The terrain types of the bars alternate between road (weight 1) and forest (weight 5). If paths are planned from left to right on this scene, then the paths exhibit path bending according to Snell's law.

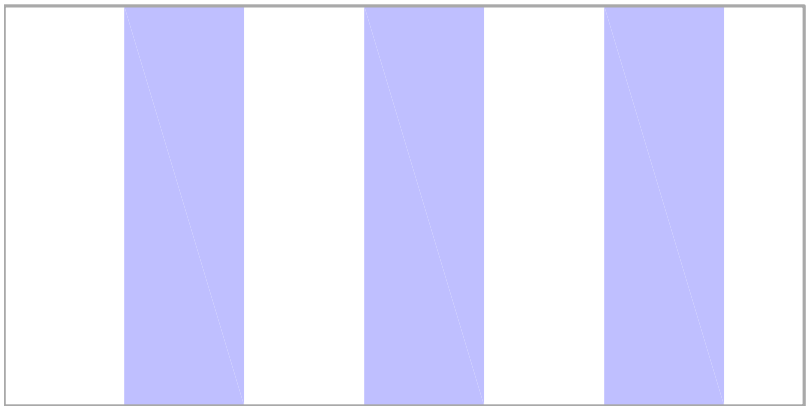


Figure 33: The Bars scene consist of several vertical bars with terrain types alternating between road (weight 1) and forest (weight 5). The higher the weight for a particular region, the darker the shade of blue.

High-low is the name of the fourth scene, which is 100 by 50 meters. The scene is shown in Figure 34 and consists of three regions. On the bottom, there is a big high-weighted region (weight 20), and above that there is a medium-weighted region (weight 3) and a low-weighted region (weight 1). Several exemplary paths of the Weighted Region problem can be shown with this scene. The optimal path can cross the same triangle multiple times. This can happen if the start and goal position lie within the high-weighted region and the optimal path uses parts of the upper region. Furthermore, we can create a setting where the start and goal positions lie in the same triangle, while the optimal path leaves this triangle to partially follow a region with a lower weight.

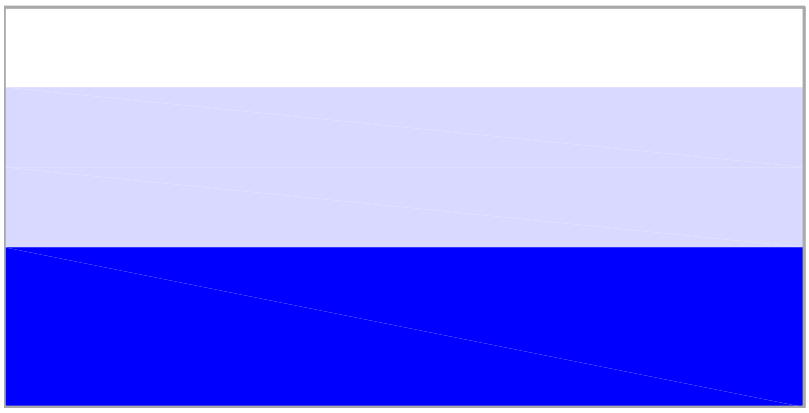


Figure 34: The High-Low scene. On the bottom, there is a big high-weighted region (weight 20), and above that there is a medium-weighted region (weight 3) and a low-weighted region (weight 1). The higher the weight for a particular region, the darker the shade of blue.

The fifth and last scene is called Zigzag and spans an area of 100 by 50 meters. The scene is shown in Figure 35, and it contains a sandy road (weight 6), a grass field above it (weight 3) and there are

alternating parts of road (weight 1) and water (weight 20). Paths planned from left to right should follow the sandy road, but switch from the bottom to the top border of this region several times. This will result in many edge-crawling paths.

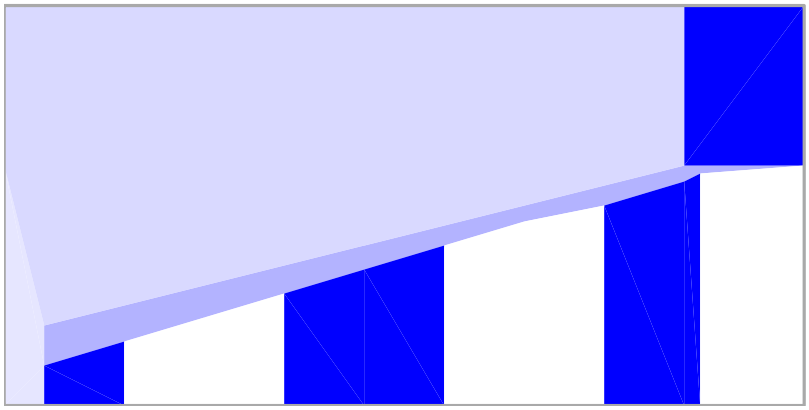


Figure 35: The Zigzag scene contains a sandy road (weight 6), a grass field above it (weight 3) and there are alternating parts of road (weight 1) and water (weight 20). The higher the weight for a particular region, the darker the shade of blue.

For easy reference, we define the following notation to refer to a scene with certain start and goal positions. The notation consists of the first letter of the scene followed by the start and goal positions between parenthesis. For instance, P(20,40)(80,35) will refer to the scene Puddle with start location (20,40) and goal location (80,35).

7.2 Heuristic experiments

Several experiments have been conducted to compare the different heuristics based on path costs, running times and number of explored nodes. The Hierarchical A* heuristic will use V2 caching, as was discussed in Section 3.2.3.

7.2.1 Hierarchical A*

First we will take a look at the two different methods that can be used to generate the abstraction layers. Our first approach to the creation of the hierarchy consists of the combination of grids sampled at different resolutions. For each abstraction layer, we will render a scaled version of the scene to the framebuffer. From the framebuffer we can extract the weights by looking at the color of every pixel. Although the framebuffer approach is fast, experiments have shown that the corresponding heuristic values overestimate the costs and result in suboptimal paths. The use of abstraction layers will always lead to inaccuracies in the heuristic values, but this is feasible as long as the heuristic values never overestimate the costs to the goal. Then the heuristics remains admissible and the resulting paths will always be optimal. When the framebuffer is used to create the abstraction layers of the scene, then OpenGL downsamples the scene to each required resolution. The OpenGL rendering pipeline draws the scene at a lower resolution and uses sampling to determine the pixel color values. Because OpenGL samples only the center of the pixel to determine the color value, there might be several regions within that pixel that all get assigned the weight of the region that lies in the middle of the pixel. If for instance a block of 4 pixels contains 3 pixels with weight 1 and 1 pixel with weight 20, then the weight value of the corresponding pixel in the downscaled scene might be 20. If Hierarchical A* searches on this abstraction level, then it might overestimate the heuristic value because it uses a weight of 20 where it could also use a weight of 1. If the pixel has a value of 1 instead of 20, then the values are still not accurate, but it will make the heuristic underestimate and thus remain admissible. To overcome the problems with OpenGL sampling, a second method has been devised to obtain an abstraction hierarchy. This method converts a block of pixels by taking the lowest value of all pixels within the block. This will create abstraction layers where the weight of each pixel is always the lowest possible weight for that pixel and will prevent overestimation of the heuristic in abstraction layers.

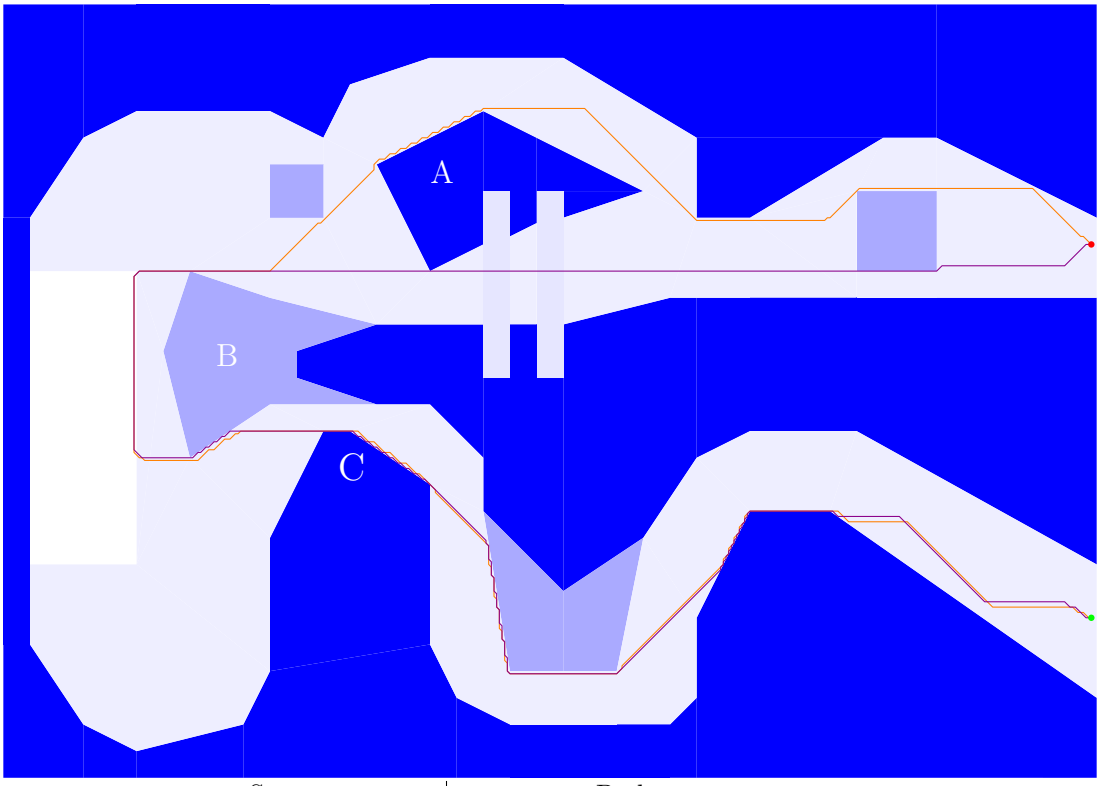
The experiments measure the path costs, because we want to see if the sampling of OpenGL has a big impact on the heuristic values. The resulting path costs for the Puddle and Forest scene are shown in Figure 36. The table shows that a simple axis-aligned scene such as the Puddle scene yields almost equal results for both methods. However, the large Forest shows a significant increase in path costs for the resolution-based abstraction method. If we look at both paths in Figure 36, then we see that there are two areas that show the limitations of resolution-based abstraction. The path takes a big detour around high-weighted region A, whereas the path under A intersects the two fallen trees. The resolution-based abstraction enlarges high-weighted region A and the high-weighted region in the center because the region borders are not aligned with the raster of the OpenGL sampling. The enlarged regions on the highest abstraction layer cause the path below A to intersect with the high-weighted region. The optimal path on this abstraction layer runs above high-weighted region A instead of running below it. As soon as a path running above region A is computed on the highest abstraction layer, the paths computed on lower levels also run above region A. The final path will end up on the top side of A instead of the much cheaper bottom side. The same problem can also be seen near puddle B on the left side. The path from the resolution-based abstraction keeps more clearance from the Puddle in comparison to the mapping-based abstraction path. This is again caused by the enlargement of obstacles because the OpenGl sampling raster is not aligned with the region boundaries. Similar behaviour also displays at several other locations, for instance near C. However, near C there is no alternative route, and all paths have to cross the high-weighted region. Due to these limitations of resolution-based abstraction, all further experiments use mapping-based abstraction.

Next we will look at the impact of the number of abstraction layers used for Hierarchical A*. We will plan several paths with the number of abstraction layers varying from one to six layers. The resulting paths for the Puddle scene are shown in Figure 37 together with the results table. The first conclusion that we can draw from the data is that the costs of the paths and number of nodes explored are independent of the number of abstraction layers. The path costs are therefore not shown in the table. It can be seen that the runnings times decrease when more abstraction layers are added. However, the decrease stabilizes soon when the abstraction layers become smaller. For instance, the level 4 abstraction layer for the puddle scene measures only 3x6 and does not speed up the search any more. The level 3 abstraction layer is already so small that Dijkstra's algorithm runs very fast on it. With the much larger forest scene we see the same when we reach the 6th abstraction layer. This abstraction layer measures only 4x6. We can conclude that using more abstraction layers speeds up the search only until the point where the highest abstraction level becomes smaller than 50 nodes. The number of 50 is chosen because then the Puddle scene will use 3 abstraction layers, and the Forest scene will use 5 abstraction layers.

Finally, we will look at different abstraction factors. The default factor is 2 because every cell maps to a 2x2 block of cells in a higher level. The number of abstraction layers is highly dependent on the abstraction factor. A higher abstraction factor will decrease the resolution between abstraction layers at a higher rate. In order to keep the highest abstraction level larger than 50 nodes, we will have to adjust the number of abstraction layers according to the abstraction factor. The results for several abstraction factors are shown in Figure 38 and Table 2.

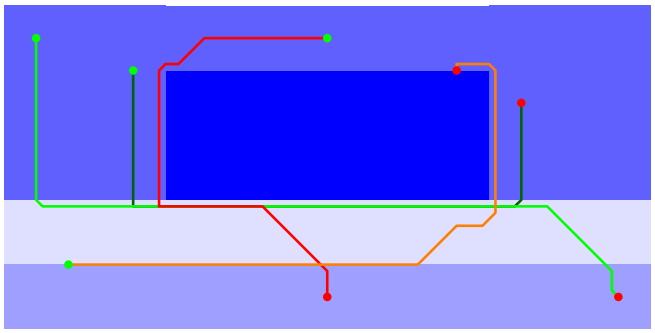
If we look at the running times, we notice that increasing the abstraction factor while keeping the same the number of abstraction layers will lead to a decrease in running time. Larger abstraction factors will create smaller abstraction layers, so there will be less nodes to explore at each level. However, when an abstraction layer is removed due to the 50-node restriction, we see an increase in running times. The top layer will increase in size, and because Dijkstra's algorithm is used on the top layer, the running times will increase. This increase is clearly visible in the Forest scene. Between the abstraction factors 3 and 4, there is a huge increase in running times. Abstraction factor 3 holds three abstraction layers with a top layer of 150 nodes, while abstraction factor 4 only has two abstraction layers with a top layer consisting of 450 nodes. Due to the large abstraction factor, the next layer would consist of only 24 and thus breaks the 50-nodes rule. In retrospect, the 50-node rule does not suffice in all cases. Therefore, we will redefine the rule in such a way that the abstraction hierarchy may only contain one layer with less than 200 nodes. The 200-node rule ensures that the Puddle and Forest scene still uses 3 and 5 abstraction layers, respectively, if the abstraction factor is 2. Furthermore, the 200-node rule will ensure that Dijkstra's algorithm will not slow down the Hierarchical A* method and will provide heuristic values for any layer larger than 200 nodes to speed up the search.

The resulting paths are negatively influenced by higher abstraction factors. The path costs and number of nodes explored both increase when higher abstraction factors are used. This is caused by two factors. First of all the overall accuracy of the heuristic is decreased when the abstraction factor is increased. Due to the inaccuracy of the heuristic, A* needs to explore a large number of nodes to find the optimal path. Furthermore, inaccurate heuristic values can lead to non-optimal paths. Next to the inaccuracy of the heuristic, using high abstraction factors leads to large blocks of nodes with the same heuristic value. The effects of such blocks are clearly visible at the abstraction factor of 7 near the panoramic view at the top of the scene. The line is bent, where one would expect a straight (diagonal) line. If a block with equal heuristic values is explored by A*, then the heuristic has no influence on the A* node selection. This will make A* behave like Dijkstra's algorithm within each block. Dijkstra's algorithm will always first encounter the edges of the block on the horizontal and diagonal lines from the entry point, because it has a circular expansion. As soon as a new block with a lower heuristic value is discovered, A* will prioritize the expansion of the new block. This will lead to some nodes of the previous block to never get explored. These unexplored nodes will always lie in a corner that is



Scene	Path costs		
	Resolution	Mapping	Optimal
P(20,40)(80,35)	241.2	241.2	241
P(5,45)(95,5)	236.3	236.9	235.6
P(50,45)(50,5)	287.6	287.7	286.9
P(10,10)(70,40)	217.5	217.8	217.3
F(60,2)(200,2)	2739.0	2556.1	2555.4

Figure 36: The resulting paths of Hierarchical A* with resolution-based abstraction layers (orange) and mapping-based abstraction layers (purple) on the Forest scene together with the paths costs of both abstraction methods for several start and goal positions on the Puddle and Forest scene. The figure is tilted 90° counterclockwise. A, B and C denote areas of special interest as explained in the text.



Scene	Abstraction layers	Running time (ms)
P(20,40)(80,35)	1	13.5
	2	5.6
	3	5.2
	4	5.2
P(5,45)(95,5)	1	16.6
	2	7.5
	3	7.1
	4	7.1
P(50,45)(50,5)	1	46.9
	2	6.6
	3	5.8
	4	5.8
P(10,10)(70,40)	1	140.5
	2	16.2
	3	6.9
	4	6.9
F(60,2)(200,2)	1	17674
	2	1034
	3	137
	4	96.0
	5	90.4
	6	90.4

Figure 37: The resulting paths of Hierarchical A* with different numbers of abstraction layers along with the running times.

opposite of the first entry point of the block and that's why these bent lines only appear on diagonal lines. The unexplored nodes have a heuristic value that overestimates the costs to the goal, because all nodes that are explored within the block have the same heuristic value and lower g-values. The abstraction factor of 8 suffers from the same problems, as can be seen near the goal position. These effects are only visible on diagonal lines that lie not too close to the start position.

There is one special case of P(10,10)(70,40) that requires examination. The path costs for abstraction factor 6 are lower than the path costs of abstraction factor 2. This is caused by the specific positioning of the start and goal positions in combination with the abstraction factor. Hierarchical A* overestimates the optimal path costs by a small amount due to the enlargement of the goal position at each abstraction layer. The goal position will also be combined with several other nodes to refer to one node in every abstraction layer. This abstraction will lead to small overestimations in the heuristics and thus suboptimal paths by a small margin. The goal position of (70,40) lies exactly in the top right corner of a node on the abstraction layer: $(70 + 1)/6 = 11\frac{5}{6}$ and $(40 + 1)/6 = 6\frac{5}{6}$. The addition of 1 is needed because the scene coordinates start at 0. Because this search starts from the bottom left corner (10,10), the paths on the abstraction layer to the abstract goal position will always be shorter than the actual path to the top right corner of the abstract goal position. All heuristic values will underestimate the costs, and thus a better path is returned.

We can conclude that using an abstraction factor of 2 leads to the best results in all cases. Higher values have a small chance to lead to lower cost paths and have lower running times, but the bound on the inaccuracy of the final path grows along with the abstraction factor. Using 2 as abstraction factor will ensure that we maintain the lowest possible bound and acceptable running times.

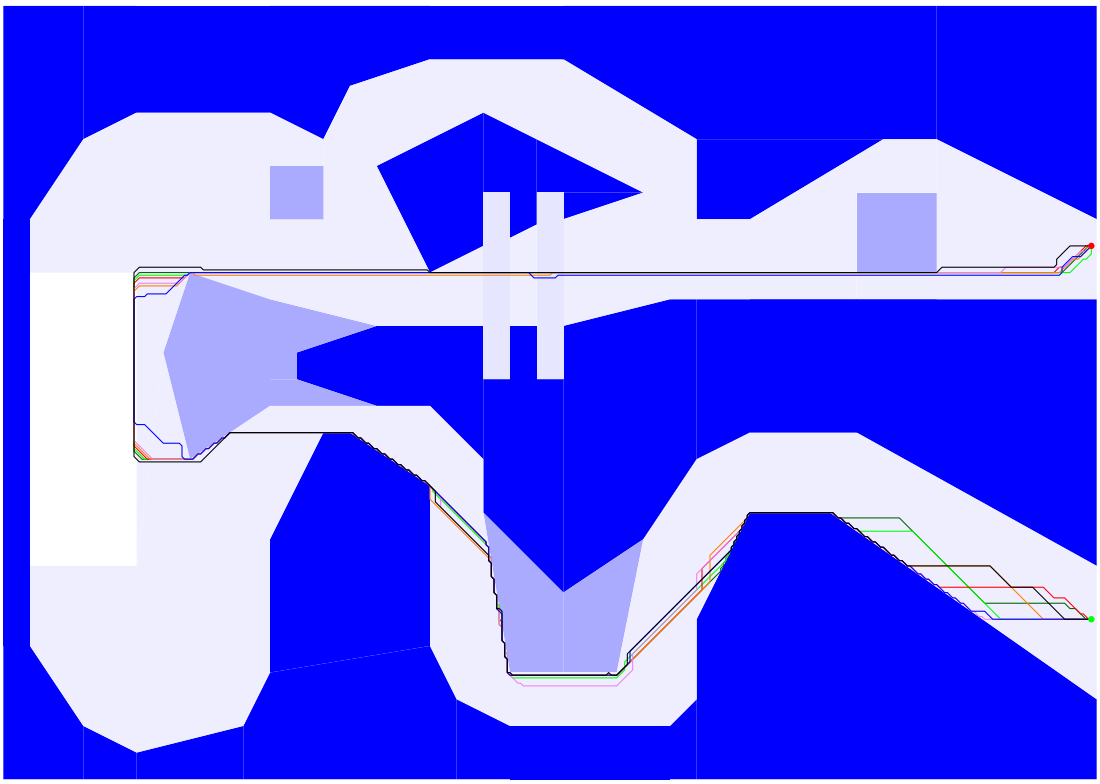


Figure 38: The resulting paths of Hierarchical A* for several abstraction factors: darkgreen = 2, green = 3, red = 4, orange = 5, pink = 6, blue = 7, black = 8 on the Forest scene with start position (60,2) and goal position (200,2).

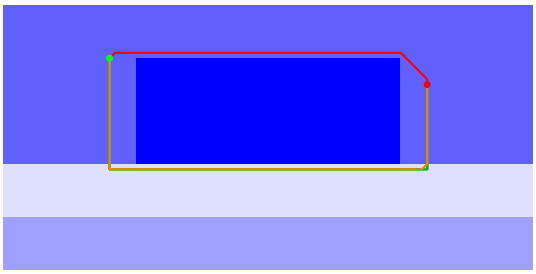
Scene	Abstr. factor	Abstr. layers	Running time (ms)	Path costs	Nodes explored
P(20,40) (80,35)	1	1	195.4	241.0	357
	2	3	5.2	241.2	490
	3	2	6.7	243.4	699
	4	1	3.6	242.3	727
	5	1	2.9	250.3	840
	6	1	3.8	244.0	1405
P(5,45) (95,5)	1	1	198.2	235.6	466
	2	3	7.1	236.0	627
	3	2	3.0	238.7	692
	4	1	4.9	235.8	1351
	5	1	3.3	237.5	1411
	6	1	3.6	236.9	1642
P(50,45) (50,5)	1	1	226.5	286.9	303
	2	3	5.8	288.9	438
	3	2	3.4	289.8	535
	4	1	4.3	305.7	824
	5	1	3.5	307.1	910
	6	1	3.2	299.3	1140
P(10,10) (70,40)	1	1	972.4	217.3	332
	2	3	6.9	217.9	776
	3	2	9.2	224.0	1198
	4	1	15.7	237.0	1117
	5	1	6.8	219.9	1121
	6	1	6.5	217.3	1987
F(60,2) (200,2)	1	1	163138.0	2555.4	3084
	2	5	96.3	2556.1	4960
	3	3	89.2	2679.1	12006
	4	2	376.7	2632.3	10726
	5	2	184.9	2623.0	12420
	6	2	138.1	2650.4	14516
	7	1	436.7	2664.8	18660
	8	1	238.2	2741.7	18915

Table 2: The number of abstraction layers, running times, path costs and explored nodes for several paths planned on the Puddle and Forest scenes.

7.2.2 Comparing heuristics

Now that we have found the optimal parameters for Hierarchical A*, we will continue with the comparison of the different heuristics. We will compare the path costs and running times of five scenario's on the Puddle scene and one on the Forest scene. The first scenario for the Puddle scene starts at (20,40) and ends at (80,35) and is shown in Figure 39. Most paths use the road with its low costs (weight 1) to find a shortest path to the goal. The Grid lookup (red) favours a straight route to the goal and ends up with very high costs. The Grid lookup method will fail to find a shortest path in all upcoming experiments and will not be considered further during the examination of the experimental results. The Hierarchical A* heuristic has slightly higher costs compared to the other heuristics because it cuts the corner on the right side. The corner is cut because the node that lies exactly on the corner has a high heuristic value in comparison to the node above it. Figure 40 shows the f, g and h-values of the Euclidean distance and Hierarchical A* around the corner node. The Hierarchical A* heuristics makes A* pick the diagonal path to the top right node immediately, while the Euclidean distance heuristic first explores to the right. This results in a difference of 0.2 in the total path costs. In terms of running times, the Hierarchical A* heuristic outperforms the other methods that return optimal paths. The actual search explores 490 nodes while the other methods explore almost ten times as many nodes. The running time isn't ten times lower than the other methods, because most of the running time of the Hierarchical A* methods is spent on creating the abstraction layers and running A* on them.

Puddle from (20,40) to (80,35)



Method	Running time(ms)	Path costs	Nodes explored
Manhattan distance	4.74	241.0	4268
Euclidean distance	4.74	241.0	4307
Dijkstra	4.78	241.0	4497
Grid Lookup	0.17	317.4	251
Hierarchical A*	3.94	241.2	490

Figure 39: The resulting paths of all heuristics: Manhattan distance(green), Euclidean distance(lightgreen), Dijkstra(darkgreen), Grid Lookup(red), Hierarchical A*(orange) on Puddle with start position (20,40) and goal position (80,35) along with the running times, path costs and explored nodes.

The second scenario is shown in Figure 41 with start position (5,45) and goal position (95,5). The grid lookup heuristic overestimates the costs to the goal and results in a path with more than twice the costs of the shortest path. The other heuristics all use the road to find their shortest paths. The paths are not exactly the same, but the total costs are equal. It does not matter whether a path runs diagonal first and then straight, or vice versa, as long as these variations happen within the same region. If the entry and exit points as well as the number of diagonal and straight steps are equal, then the order of the steps has no influence on the path costs. The Hierarchical A* heuristic suffers from an overestimation again. This time the Hierarchical A* path runs parallel to the optimal path at the end. These parallel lanes have equal heuristic costs because the heuristic values are generated for blocks of 2x2 nodes. The A* algorithm will pick the closest one to the start to traverse because of the lower g-costs, and will end up doing a diagonal step towards the goal. The Hierarchical A* method has quite a high running time in this scenario. The additional time required for the creation of the abstraction layers and searching them is higher than the time required to search without additional guidance, because only around 3000 nodes needed to be explored to find the shortest path. Dijkstra's

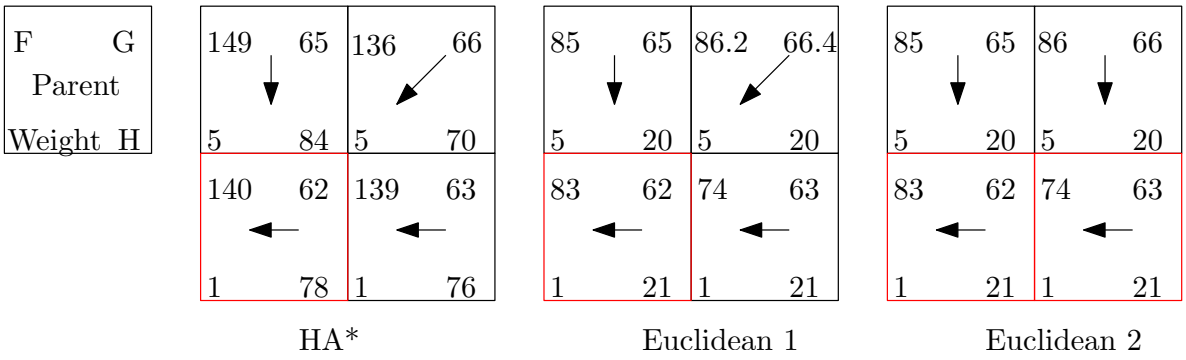
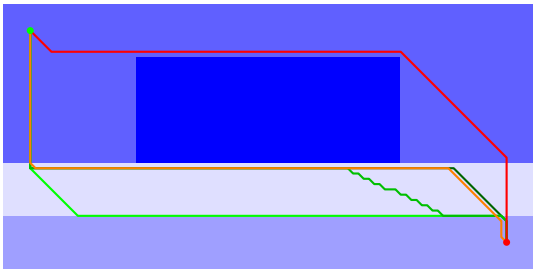


Figure 40: A* exploration around the right corner of the paths returned with the Hierarchical A* and Euclidean distance heuristics. The Hierarchical A* heuristic will make A* pick the top right node immediately, while the Euclidean distance heuristic will first explore the right and then move to the top, leading to 0.2 less in path costs.

heuristic also has higher running times because the distance between the start and goal position is almost at its maximum.

Puddle from (5,45) to (95,5)



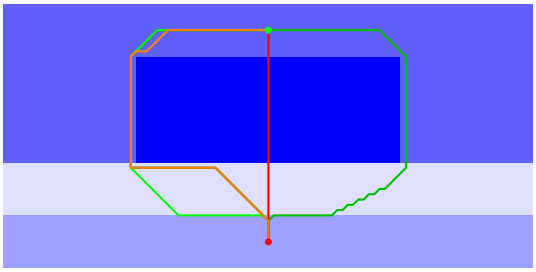
Method	Running time(ms)	Path costs	Nodes explored
Manhattan distance	3.20	235.6	2900
Euclidean distance	3.13	235.6	2942
Dijkstra	3.99	235.6	3791
Grid Lookup	0.34	530.7	424
Hierarchical A*	5.91	236.0	772

Figure 41: The resulting paths of all heuristics: Manhattan distance(green), Euclidean distance(lightgreen), Dijkstra(darkgreen), Grid Lookup(red), Hierarchical A*(orange) on Puddle with start position (5,45) and goal position (95,5) along with the running times, path costs and explored nodes.

The third and fourth scenarios show similar behaviour as the first two scenarios. The results are still provided to show the consistency of the methods. The third scenario is shown in Figure 42 and has a start position of (50,45) and goal position of (50,5) in the Puddle scene. The fourth scenario in the Puddle scene with start position (10,10) and goal position (70,40) is shown in Figure 43.

The last scenario is shown in Figure 44. In this scenario, we examine the Forest scene with start position (60,2) and goal position (200,2). This scene is a lot larger, and the Hierarchical A* method excels with only 5000 explored nodes and a reduction of 50 ms in running time. The other heuristics explore up to 16 times as many nodes to find the optimal path. As with the previous scenarios, the Hierarchical A* heuristic has slightly higher path costs due to the corners being cut at the top when entering and exiting the panoramic view (weight 1).

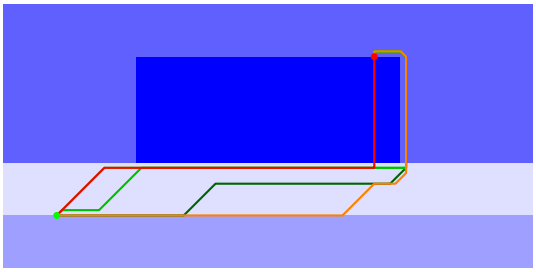
Puddle from (50,45) to (50,5)



Method	Running time(ms)	Path costs	Nodes explored
Manhattan distance	3.70	286.9	3343
Euclidean distance	3.80	286.9	3467
Dijkstra	5.26	286.9	4980
Grid Lookup	0.11	466	149
Hierarchical A*	4.63	286.9	438

Figure 42: The resulting paths of all heuristics: Manhattan distance(green), Euclidean distance(lightgreen), Dijkstra(darkgreen), Grid Lookup(red), Hierarchical A*(orange) on Puddle with start position (50,45) and goal position (50,5) along with the running times, path costs and explored nodes.

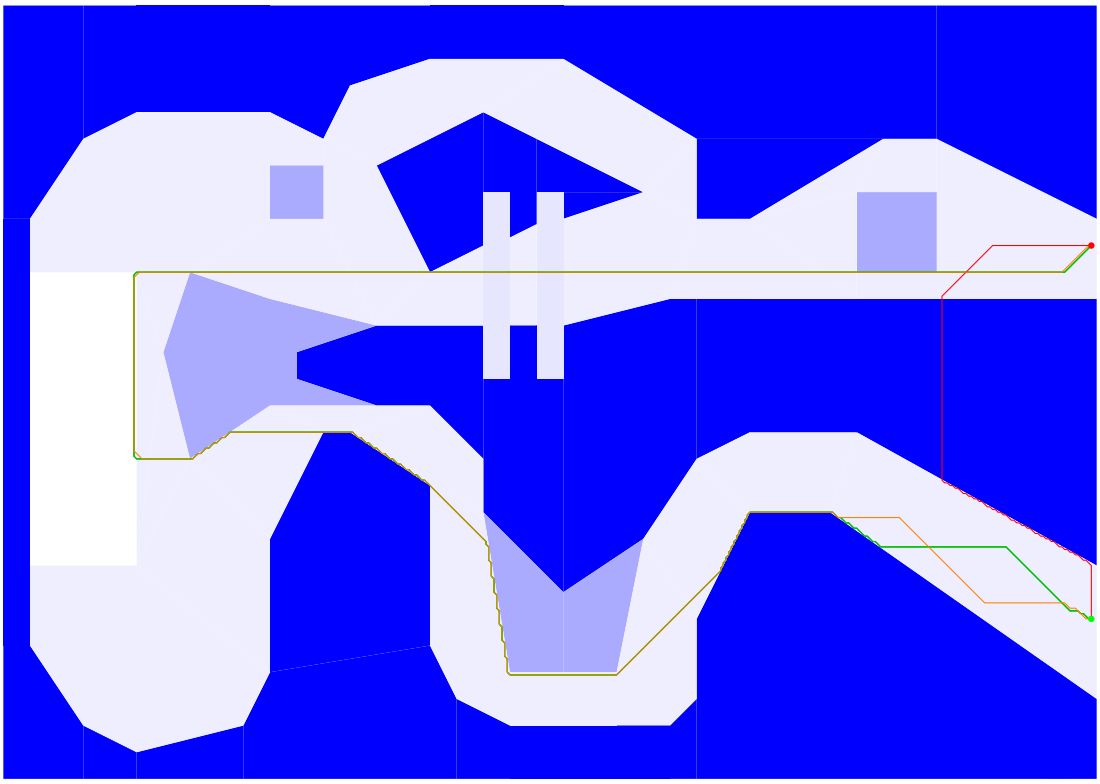
Puddle from (10,10) to (70,40)



Method	Running time(ms)	Path costs	Nodes explored
Manhattan distance	5.17	217.3	4339
Euclidean distance	5.27	217.3	4390
Dijkstra	5.17	217.3	4638
Grid Lookup	1.19	474	976
Hierarchical A*	5.80	217.9	776

Figure 43: The resulting paths of all heuristics: Manhattan distance(green), Euclidean distance(lightgreen), Dijkstra(darkgreen), Grid Lookup(red), Hierarchical A*(orange) on Puddle with start position (10,10) and goal position (70,40) along with the running times, path costs and explored nodes.

Forest from (60,2) to (200,2)



Method	Running time(ms)	Path costs	Nodes explored
Manhattan distance	141	2555.4	78604
Euclidean distance	140	2555.4	78920
Dijkstra	151	2555.4	83250
Grid Lookup	0.84	7194.4	869
Hierarchical A*	88.9	2556.1	4960

Figure 44: The resulting paths of all heuristics: Manhattan distance(green), Euclidean distance(lightgreen), Dijkstra(darkgreen), Grid Lookup(red), Hierarchical A*(orange) on Forest with start position (60,2) and goal position (200,2) along with the running times, path costs and explored nodes.

7.2.3 Conclusions

In this section, we have discussed the experiments to find the best abstraction technique and parameters for Hierarchical A* and to compare different heuristics for A* on weighted regions. We compared the resolution-based abstraction against the mapping-based abstraction. Resolution-based abstraction generates higher heuristic values due to enlarged obstacles or removed low-cost areas on the abstraction levels. The overestimation problems from resolution-based abstraction are solved by using a mapping that only keeps the lowest values. Mapping-based abstraction tends to underestimate, but that keeps the heuristic admissible.

There are two Hierarchical A* parameters. The first parameter controls the number of abstraction layers that are used in the hierarchy. The experiments show that the path costs are independent of this parameter, and only the running time was influenced by this parameter. Running times decrease with every added abstraction layer until the layers reach a certain size. We showed that a bottom layer of less than 200 nodes is small enough for Dijkstra's algorithm to outperform Hierarchical A* with another layer. The second parameter is the abstraction factor that controls the difference in cell size between each abstraction layer. Increasing the abstraction factor improves the running times at the costs of path optimality. We are looking for optimal paths, and as such we will stick to the lowest abstraction factor, which is 2.

Once the optimal parameters for Hierarchical A* had been found, we compared all the different heuristics with each other. Each heuristic has been compared with respect to path costs, running time and number of nodes explored. The grid lookup returned non-optimal paths on all experiments and is not a feasible heuristic, as expected. On the Puddle scene, the resulting running times of the other heuristics didn't show any significant differences, and each heuristic was the fastest on one of the settings. The paths of the Hierarchical A* method were all no more than 1 off in terms of path costs when compared to the optimal paths. Although A* explores up to 8 times less nodes when Hierarchical A* is used, the extra time needed for construction and planning on the abstraction layers negates the time saved on exploring nodes. However, in the Forest scene, the better guidance of Hierarchical A* is clearly visible. A significant decrease ($p < 0.001$) in running time and a near-optimal solution makes the Hierarchical A* method outperform all other methods.

7.3 Pruned graph experiments

In this section, we will compare the different hybrid methods. We will look at the performance of a plain Steiner graph and graphs that have been pruned by any of the three pruning methods from Section 5. All scenes will be taken into account to compare the performance for different kinds of environments. We will create the different graphs for ϵ-values ranging from 0.1 to 0.5. The performance of each method will be measured by the construction time of the graph, query time with Dijkstra's algorithm, path costs and the number of nodes explored by Dijkstra's algorithm. Once the running times are measured, we will try to find cases where the pruned graph methods fail to return the shortest path. We will use a uniform sampling method to generate start and goal positions across all scenes. The costs of the resulting paths of the pruned graph methods will be compared to the optimal path costs of the Steiner graph for each start and goal position.

7.3.1 Running times

The first experiments are conducted on the Forest scene with start position (60,2) and goal position (200,2). The resulting paths for the different ϵ-values are shown in Figure 45 together with the data in Table 46. All methods return the same path costs, and the figure shows the paths for the different ϵ-values. The pruned graph methods show a significant improvement in the construction times of the graphs for low ϵ-values. T-tests for all pruned graph methods construction times in comparison

with the Steiner graph returned $p < 0.001$ for $\epsilon = 0.1$ and $\epsilon = 0.5$. As the ϵ-value increases, the difference in construction times becomes smaller. The pruned graph methods save a lot of time on adding Steiner points and connecting Steiner points, but they require additional time to find the set of edges that makes up the pruned graph. The costs of finding this set of edges is independent of the ϵ-value. As the ϵ-value becomes higher the impact of this preprocessing step has bigger impact on the total construction time. However, the savings made by pruning the graph allow the pruned graph methods to outperform the Steiner graph even when using the highest ϵ-value. The query times of the pruned graph methods also show a significant improvement for all ϵ-values in comparison to the Steiner graph. T-tests for all pruned graph methods query times in comparison with the Steiner graph returned $p < 0.001$ for $\epsilon = 0.1$ and $\epsilon = 0.5$. The pruned graphs contain less nodes and are able to reduce the number of nodes explored, which leads to lower query times. The closest-edge-based pruning method is able to achieve the lowest construction and query times on the Forest scene.

Forest from (60,2) to (200,2)

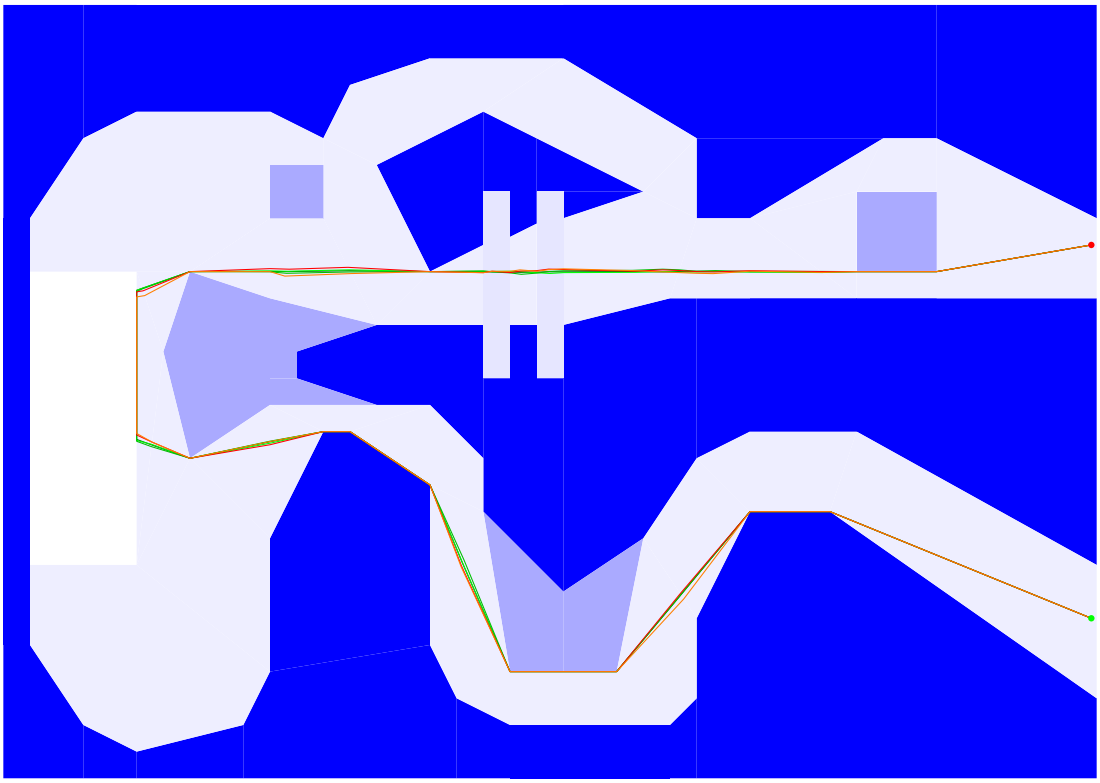


Figure 45: The resulting paths for different ϵ-values: $\epsilon = 0.1$ (green), $\epsilon = 0.2$ (lightgreen), $\epsilon = 0.3$ (darkgreen), $\epsilon = 0.4$ (red), $\epsilon = 0.5$ (orange) on Forest with start position (60,2) and goal position (200,2).

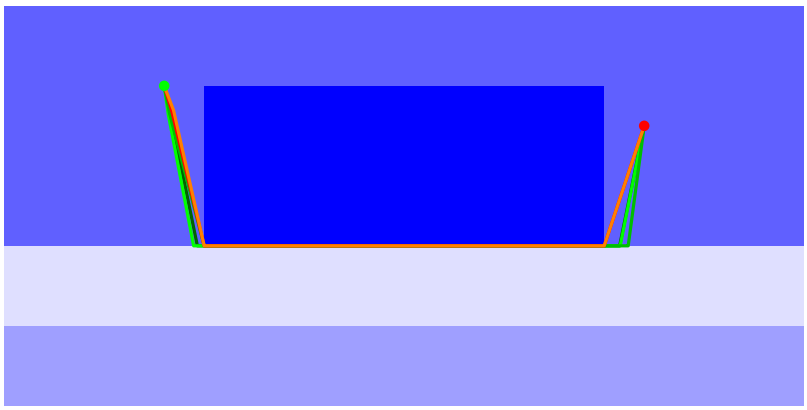
In addition to the Forest scene, the Puddle scene has been tested. Start and goal positions of (20,40) and (80,35) are used. The resulting paths are shown in Figure 47 together with their construction times, query times, explored nodes and path costs. Similar to the Forest scene, the pruned graphs methods show an improvement over the Steiner graph in this scenario. Both construction times and query times are lowered by all of the pruned graph methods. Overall, the closest-edge-based pruning is the fastest method.

Forest from (60,2) to (200,2)

ϵ	Graph type	Construction time (ms)	Query time (ms)	Nodes explored	Path costs
0.1	Steiner graph	163325.0	1141.1	47969	2461.2
	Triangle intersection based pruning	20580.6	266.4	19204	2461.2
	Vertex based pruning	16857.2	224.6	15413	2461.2
	Closest edge based pruning	14360.3	183.9	13362	2461.2
0.2	Steiner graph	9638.0	127.8	17710	2461.4
	Triangle intersection based pruning	1207.0	32.0	6990	2461.4
	Vertex based pruning	1049.6	26.8	5700	2461.4
	Closest edge based pruning	947.4	22.5	4924	2461.4
0.3	Steiner graph	1794.7	34.3	9370	2461.9
	Triangle intersection based pruning	376.8	9.2	3654	2461.9
	Vertex based pruning	351.9	7.8	3031	2461.9
	Closest edge based pruning	359.6	6.5	2610	2461.9
0.4	Steiner graph	600.6	13.4	5744	2462.5
	Triangle intersection based pruning	255.3	3.6	2206	2462.5
	Vertex based pruning	245.0	3.1	1863	2462.5
	Closest edge based pruning	273.4	2.7	1599	2462.5
0.5	Steiner graph	300.4	6.4	3826	2464.2
	Triangle intersection based pruning	226.6	1.7	1448	2464.2
	Vertex based pruning	225.4	1.4	1243	2464.2
	Closest edge based pruning	252.2	1.3	1068	2464.2

Figure 46: The construction times, query times, number of nodes explored and path costs of the Steiner graph and pruned Steiner graphs for different ϵ-values on the Forest scene with start position (60,2) and goal position (200,2).

Puddle from (20,40) to (80,35)



ϵ	Graph type	Construction time (ms)	Query time (ms)	Nodes explored	Path costs
0.1	Steiner graph	49911.9	222.3	7441	231.5
	Triangle intersection based pruning	19768.9	84.1	3834	231.5
	Vertex based pruning	18396.8	77.6	3561	231.5
	Closest edge based pruning	17572.2	76.2	3361	231.5
0.2	Steiner graph	2943.4	29.0	2856	231.5
	Triangle intersection based pruning	1170.9	11.7	1470	231.5
	Vertex based pruning	1084.5	10.7	1367	231.5
	Closest edge based pruning	1043.2	10.4	1300	231.5
0.3	Steiner graph	617.7	8.7	1560	231.8
	Triangle intersection based pruning	302.9	3.4	804	231.8
	Vertex based pruning	283.1	3.1	751	231.8
	Closest edge based pruning	273.9	3.1	717	231.8
0.4	Steiner graph	255.3	3.6	986	232.2
	Triangle intersection based pruning	168.8	1.4	511	232.2
	Vertex based pruning	164.2	1.4	479	232.2
	Closest edge based pruning	156.5	1.3	460	232.2
0.5	Steiner graph	167.0	1.8	675	232.3
	Triangle intersection based pruning	132.7	0.8	351	232.3
	Vertex based pruning	125.0	0.7	331	232.3
	Closest edge based pruning	125.0	0.7	320	232.3

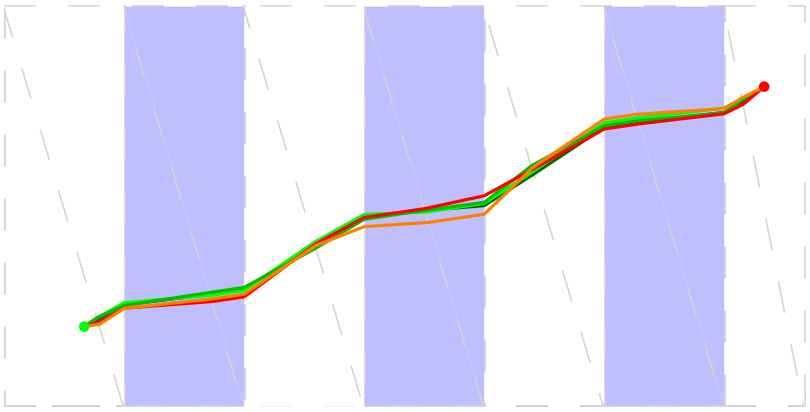
Figure 47: The resulting paths for different ϵ-values: ϵ = 0:1 (green), ϵ = 0:2 (lightgreen), ϵ = 0:3(darkgreen), ϵ = 0:4 (red), ϵ = 0:5 (orange) together with the construction times, query times, number of nodes explored and path costs of the Steiner graph and pruned Steiner graphs on the Puddle scene with start position (20,40) and goal position (80,35).

The Bars scene provides an interesting test case, because it only consists of several long vertical triangles. By planning a path from left to right, we will create a scenario were the optimal path intersects almost every triangle in the scene. It is important to test how the pruned graph methods perform when the pruned graphs are close to the original Steiner graph. The resulting paths and experimental data are shown in Figure 48. The triangle-intersection-based pruning method selects all triangles in the scene because the number of explored nodes is equal to the Steiner graph for each ϵ-value. The additional costs required for triangle-intersection-based pruning and vertex-based pruning are visible, and the Steiner graph requires lower construction times in comparison to both pruning methods. The closest-edge-based pruning method achieves lowest construction times for low ϵ-values because the edge-selection costs are low, and the number of pruned nodes and edges is highest. However, with higher ϵ-values the Steiner graph also outperforms the closest-edge-based pruning. As expected, the pruned graph methods do not outperform the Steiner graph method. The additional costs required to search for a pruned graph have a negative influence on the construction costs because in this test scenario, Steiner points are still added on all edges.

The Zigzag scene is a very complex scene that contains many long triangles with small internal angles. These small angles lead to small vertex vicinities and a large number of Steiner points. Paths are planned on the Zigzag scene from (3,7) to (90,35), and the results are shown in Figure 49. The closest-edge-based pruning method is not able to significantly reduce the number of Steiner points, and the Steiner graph achieves lower construction times for $\epsilon = 0.5$, while the construction times for lower ϵ-values are similar to each other. The vertex-based pruning method is well-suited for this scenario and is able to significantly reduce the number of Steiner points. This results in the lowest construction and query times for all ϵ-values.

The last scenario is the High-Low scene with start position (10,10) and goal position (95,10). The final paths and running times are shown in Figure 50 and show that the pruned graphs do not yield a big improvement over the Steiner graph in this scenario. Although the number of pruned nodes and edges is small, the pruned-graph methods show lower running times for low ϵ-values. As the ϵ-value increases, the difference in construction times between the methods becomes smaller and only the triangle-intersection-based pruning method achieves a significantly lower construction time for $\epsilon = 0.5$.

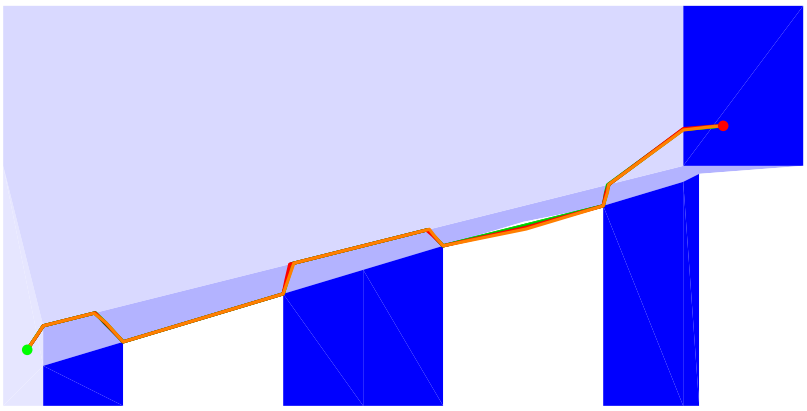
Bars from (10,10) to (95,40)



ϵ	Graph type	Construction time (ms)	Query time (ms)	Nodes explored	Path costs
0.1	Steiner graph	6993.7	62.8	4908	273.6
	Triangle intersection based pruning	7057.0	61.1	4908	273.6
	Vertex based pruning	5281.6	48.5	3957	273.6
	Closest edge based pruning	4057.1	37.4	3094	273.6
0.2	Steiner graph	450.3	8.4	1774	273.8
	Triangle intersection based pruning	470.9	8.2	1774	273.8
	Vertex based pruning	394.6	6.7	1468	273.8
	Closest edge based pruning	344.6	5.4	1203	273.8
0.3	Steiner graph	166.1	2.4	909	273.8
	Triangle intersection based pruning	181.5	2.4	909	273.8
	Vertex based pruning	171.9	1.9	776	273.8
	Closest edge based pruning	166.4	1.7	672	273.8
0.4	Steiner graph	117.6	0.9	542	274.0
	Triangle intersection based pruning	140.6	0.9	542	274.0
	Vertex based pruning	136.5	0.8	475	274.0
	Closest edge based pruning	133.3	0.7	431	274.0
0.5	Steiner graph	120.7	0.5	370	274.2
	Triangle intersection based pruning	131.0	0.5	370	274.2
	Vertex based pruning	131.5	0.4	329	274.2
	Closest edge based pruning	127.2	0.4	306	274.2

Figure 48: The resulting paths for different ϵ-values: $\epsilon = 0.1$ (green), $\epsilon = 0.2$ (lightgreen), $\epsilon = 0.3$ (darkgreen), $\epsilon = 0.4$ (red), $\epsilon = 0.5$ (orange) on the Bars scene with start position (10,10) and goal position (95,40) together with the edges of the triangulation (dashed gray). The table shows the construction times, query times, number of nodes explored and path costs of the Steiner graph and pruned Steiner graphs.

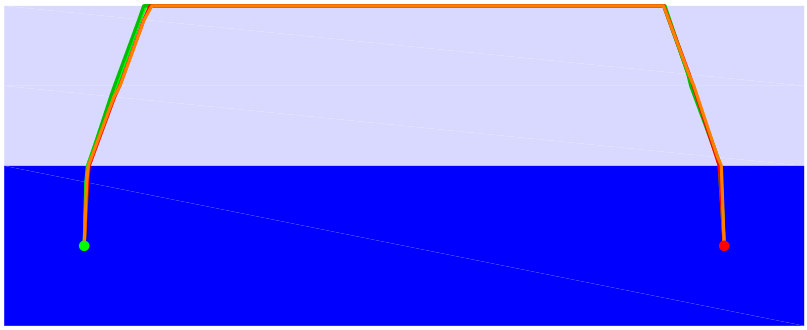
Zigzag from (3,7) to (90,35)



ϵ	Graph type	Construction time (ms)	Query time (ms)	Nodes explored	Path costs
0.1	Steiner graph	438767.4	2238.3	26245	343.6
	Triangle intersection based pruning	358115.0	2082.3	24869	343.6
	Vertex based pruning	303882.8	1789.0	21004	343.6
	Closest edge based pruning	410700.6	2116.3	23222	343.6
0.2	Steiner graph	28167.3	256.5	10334	343.6
	Triangle intersection based pruning	22321.1	238.4	9853	343.6
	Vertex based pruning	19252.0	203.3	8333	343.6
	Closest edge based pruning	26186.6	242.0	9263	343.6
0.3	Steiner graph	5089.8	72.8	5796	343.7
	Triangle intersection based pruning	4049.9	67.7	5560	343.7
	Vertex based pruning	3547.5	57.8	4717	343.7
	Closest edge based pruning	4819.5	70.1	5268	343.7
0.4	Steiner graph	1591.8	30.0	3765	343.8
	Triangle intersection based pruning	1332.0	27.6	3634	343.8
	Vertex based pruning	1192.1	24.2	3096	343.8
	Closest edge based pruning	1570.0	29.1	3471	343.8
0.5	Steiner graph	683.3	15.3	2645	343.8
	Triangle intersection based pruning	618.5	13.9	2567	343.8
	Vertex based pruning	565.3	12.4	2199	343.8
	Closest edge based pruning	725.3	14.6	2478	343.8

Figure 49: The resulting paths for different ϵ-values: $\epsilon = 0.1$ (green), $\epsilon = 0.2$ (lightgreen), $\epsilon = 0.3$ (darkgreen), $\epsilon = 0.4$ (red), $\epsilon = 0.5$ (orange) together with the construction times, query times, number of nodes explored and path costs of the Steiner graph and pruned Steiner graphs for different ϵ-values on the Zigzag scene with start position (3,7) and goal position (90,35).

High-Low from (10,10) to (95,10)



ϵ	Graph type	Construction time (ms)	Query time (ms)	Nodes explored	Path costs
0.1	Steiner graph	127198.0	471.1	7887	592.7
	Triangle intersection based pruning	105146.4	412.8	7185	592.7
	Vertex based pruning	117328.3	448.8	7409	592.7
	Closest edge based pruning	86438.5	264.5	5304	592.7
0.2	Steiner graph	7895.9	60.9	3016	592.8
	Triangle intersection based pruning	6451.6	53.4	2742	592.8
	Vertex based pruning	7245.3	59.9	2721	592.8
	Closest edge based pruning	5840.8	39.6	2187	592.8
0.3	Steiner graph	1409.0	17.7	1648	592.8
	Triangle intersection based pruning	1183.2	15.7	1497	592.8
	Vertex based pruning	1335.7	16.6	1494	592.8
	Closest edge based pruning	1168.7	13.0	1276	592.8
0.4	Steiner graph	487.5	7.3	1039	592.8
	Triangle intersection based pruning	423.3	6.7	945	592.8
	Vertex based pruning	475.6	6.9	946	592.8
	Closest edge based pruning	444.9	5.9	861	592.8
0.5	Steiner graph	251.8	3.5	723	592.9
	Triangle intersection based pruning	235.0	3.2	659	592.9
	Vertex based pruning	252.2	3.4	662	592.9
	Closest edge based pruning	251.6	3.2	632	592.9

Figure 50: The resulting paths for different ϵ-values: $\epsilon = 0.1$ (green), $\epsilon = 0.2$ (lightgreen), $\epsilon = 0.3$ (darkgreen), $\epsilon = 0.4$ (red), $\epsilon = 0.5$ (orange) together with the construction times, query times, number of nodes explored and path costs of the Steiner graph and pruned Steiner graphs for different ϵ-values on the High-Low scene with start position (10,10) and goal position (90,10).

7.3.2 Path costs

Now that we have seen the performance of the pruned graph methods, we also have to check the consistency of the pruned methods. In order to check whether the pruned graph methods always return the same paths as the Steiner graph we have conducted several experiments. These experiments are based on uniform sampling of start and goal positions on a scene. With this technique, we will be able to generate paths for numerous start and goal positions while covering the entire scene. The costs of the resulting paths are compared. If the path returned by a pruned-graph method returns different costs than the path returned by the Steiner graph, then this will be marked as an error for the corresponding pruned-graph method.

The Puddle scene is tested first. A total of 6561 paths have been planned on the Puddle scene using the uniform sampling method. The 6561 paths resulted from every combination of 81 start positions and 81 goal positions. The 81 positions are generated from every combination of 9 x-coordinates and 9 y-coordinates. The x and y-coordinates are uniformly divided over the entire scene. The number of errors of each method is shown in Table 51. The shared errors column shows the number of paths were all three pruned graph methods fail. The number of errors is low for all methods. Upon closer examination of the errors, the shared errors were all caused by a difference in the homotopy of the grid path and the Steiner graph path. An example of such a case is shown in Figure 52. If the homotopy of the grid paths diverts from the Steiner graph path, then the edges selected for pruning might contain the optimal Steiner graph path.

Pruned graph errors on Puddle scene.

Graph type	Number of errors
Triangle intersection based pruning	18
Vertex based pruning	23
Closest edge based pruning	26
Shared errors	17

Figure 51: The number of errors in pruned graph methods resulting paths for 6561 uniformly sampled start and goal positions on the Puddle scene.

Next to these shared errors, there are also cases where one pruned graph method fails, while the other pruned graph methods succeed in finding the optimal path in the graph. Such a case is highlighted in Figure 53. The difference in resulting paths is caused by the different homotopy of the grid path that is used to prune the Steiner graph. The other graph pruning methods select more edges and are able to cope with the difference in homotopy of the grid path and optimal path.

Similar experiments have been conducted on all scenes. The results for the Forest scene are shown in Table 54. We used 9000 uniformly sampled start and goal positions, because the Forest scene is larger than the Puddle scene. The triangle-intersection-based pruning method performs badly with three times as many errors as the other pruned-graph methods.

The Bars scene yields a clean sheet with no errors for any pruned-graph method, as shown in Table 55. The Bars scene is tested for 6561 uniformly sampled start and goal positions. The pruned graph methods are error-free, because the number of triangles is small, and the triangles are large. Large triangles always have a bigger chance to be selected for the pruned graphs. Small triangles can easily be missed by the triangle-intersection method or the closest-edge-based pruning method.

The number of errors on the High-Low and Zigzag scenes are shown in Table 56 and Table 57, respectively. Again the 6561 uniformly sampled paths are used. The High-Low scene shows almost only shared errors. In contrast to the High-Low scene, the Zigzag scene shows a lot more individual errors, and this caused by the complexity of the scene. The Zigzag scene contains a high number of small triangles, which are easily missed by the triangle-intersection based and closest-edge-based

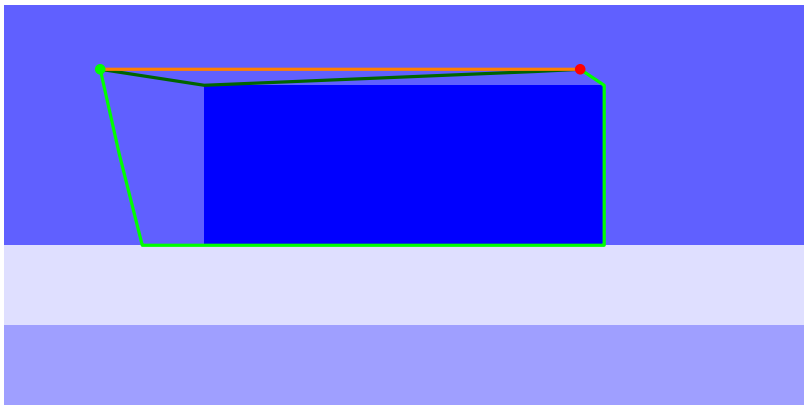


Figure 52: Different resulting paths for the pruned graph methods (darkgreen) and Steiner graph (green) on the Puddle scene with start position (12,42) and goal position (72,42). The difference in resulting paths is caused by the different homotopy of the grid path (orange) that is used to prune the Steiner graph.

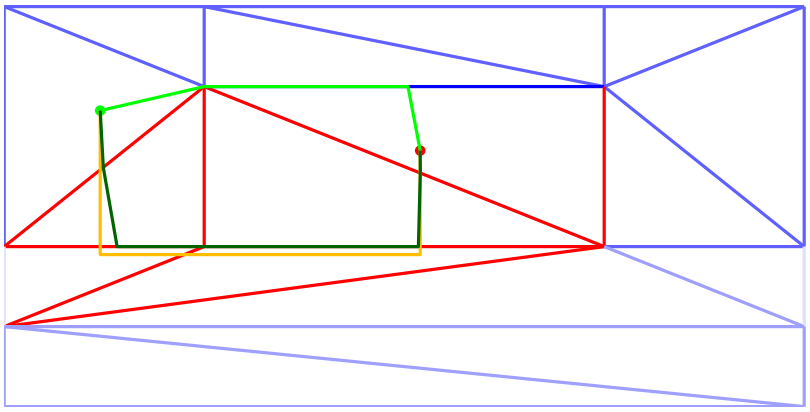


Figure 53: Different resulting paths for the Closest edge based pruning method (darkgreen) and Steiner graph (green) on the Puddle scene with start position (12,37) and goal position (52,32). The difference in resulting paths is caused by the different homotopy of the grid path (orange).

Pruned graph errors on Forest scene.

Graph type	Number of errors
Triangle intersection based pruning	1623
Vertex based pruning	587
Closest edge based pruning	590
Shared errors	583

Figure 54: The number of errors in pruned graph methods resulting paths for 9000 uniformly sampled start and goal positions on the Forest scene.

Pruned graph errors on Bars scene.

Graph type	Number of errors
Triangle intersection based pruning	0
Vertex based pruning	0
Closest edge based pruning	0
Shared errors	0

Figure 55: The number of errors in pruned graph methods resulting paths for 6561 uniformly sampled start and goal positions on the Bars scene.

pruning methods. The vertex-based pruning method suffers a lot less from these problems and almost only fails to find the optimal path when all pruned graph methods fail.

Pruned graph errors on High-Low scene.

Graph type	Number of errors
Triangle intersection based pruning	128
Vertex based pruning	132
Closest edge based pruning	128
Shared errors	128

Figure 56: The number of errors in pruned graph methods resulting paths for 6561 uniformly sampled start and goal positions on the High-Low scene.

Pruned graph errors on Zigzag scene.

Graph type	Number of errors
Triangle intersection based pruning	501
Vertex based pruning	201
Closest edge based pruning	399
Shared errors	198

Figure 57: The number of errors in pruned graph methods resulting paths for 6561 uniformly sampled start and goal positions on the Zigzag scene.

The accumulated results are shown in Table 58. Combined over all scenes, 35244 paths were planned for each pruned graph method and compared with the Steiner graph method on path costs. Overall, the vertex-based pruning method is most reliable with a success rate of 97.3%.

Total pruned graph errors on all scenes.

Graph type	Number of errors	Success rate
Triangle intersection based pruning	2270	93.6%
Vertex based pruning	943	97.3%
Closest edge based pruning	1143	96.8%
Shared errors	926	97.4%

Figure 58: The number of errors in pruned graph methods resulting paths for 35244 uniformly sampled start and goal positions on all scenes.

7.3.3 Conclusions

The pruned-graph methods rely on an investment, in terms of running times, to decrease the construction and query times. This investment is the time required to find the edges that will be pruned. The construction times can be reduced because a lot less connections have to be made when the Steiner graph is pruned. Furthermore, by confining Dijkstra's algorithm to pruned graphs, the number of nodes explored and query times can be greatly reduced. The return on investment depends on the scenario and the ϵ-value. Low ϵ-values generate a large number of Steiner points per edge, and pruning a small number of edges can already lead to improved running times. As the ϵ-value rises, it becomes harder to improve running times because the costs of pruning the graph might be higher than the costs of adding Steiner points to the edges. As such, the scenario has a big impact on the performance of the pruned-graph methods. If there is nothing to prune, then the pruned-graph methods yield unnecessary overhead in terms of construction times. Since the resulting graphs will be the same, the pruned-graph methods will need more time than the construction time of the Steiner graph. On the other hand, if the scenario only uses a small part of the scene, then massive improvements can be made. This has been shown in the Forest scene, where the pruned-graph method achieved 10 times lower construction times. In most cases, the pruned-graph methods will outperform the Steiner graph method on construction time and query time.

The downside of the pruned-graph methods is the dependency on the grid path that is used to guide the pruning process. The pruned-graph methods do not always return the optimal path that can be found in the Steiner graph. If the homotopy of the grid path differs from the optimal path, then Steiner points that lie on the optimal path might be pruned from the graph. This happens in 3%-7% of the tested cases, depending on the pruned-graph method.

Overall, the vertex-based pruning method performed the best in comparison to the other two pruned-graph methods. It improved the construction and query times of the Steiner graph on all scenarios except the Bars scenario. Furthermore, the vertex-based pruning method almost only fails to find the shortest path when all three methods fail, and it has a success rate of 97.3%.

7.4 Overall comparison

Finally, we will compare all methods presented throughout this thesis. To compare all the methods, we will have to calculate the ϵ-value of all scenes. The Forest and Zigzag scene do not fulfill the property of region boundary alignment with grid cells. Therefore, any calculated ϵ-values for these scenes are not valid. We will still use the scenes and calculate the ϵ-values to get an idea of the performance of the grid-based methods on these two scenes.

$$\sqrt{4-2\sqrt{2}}-1+r*\min\left(\frac{\sqrt{2}}{2}w_{max},\frac{1}{2}w_{min}+\frac{1}{2}w_{max}\right) \tag{1}$$

We will use Equation 1, which has been presented in Section 3.1.10, to calculate the ϵ-value for each scene based on the weights w and cell size r. For a cell size of 1 and the weights of each scene we will end up with an ϵ-value of 10.5824 for all scenes, except the Forest scene. In contrast to the other scenes, the Forest scene has a maximum weight of 99 instead of 20. This will result in an ϵ-value of 50.0824.

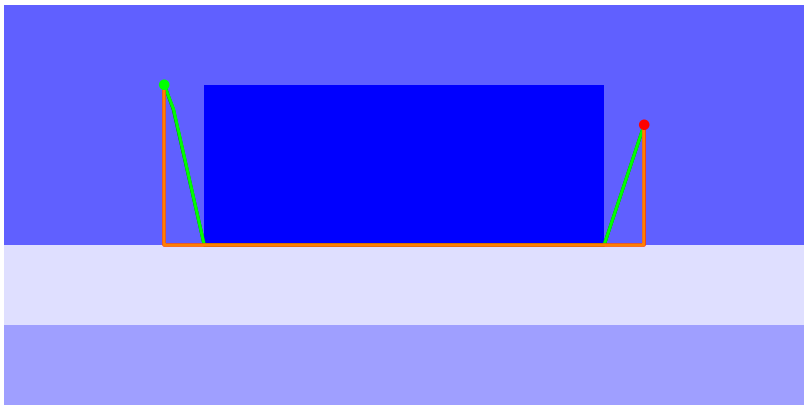
To reduce the ϵ-value of the Forest scene, we lowered the maximum weight to 20. With the ϵ-values known for each scene, we can lower the ϵ-value by decreasing the cell size. As shown in Section 4.3, the maximum ϵ-value for the ϵ-approximation method that is implemented is $\frac{1}{2}$. If we want to achieve the same ϵ-bound on the grid methods, we will have to use a cell size of 0.0397. Using such a small cell size turned out to be a problem in the Forest scene. The size of the grid of the Forest scene rises from 410x290 nodes to 10327x7304 nodes, and the 4Gb of memory on the computer that ran the experiments could not handle such a large grid. As such, the grid-paths could not be computed for

the Forest scene.

We will use vertex-based pruning (VBP) as pruned-graph method because it achieved the highest success rate of all pruned-graph methods. All graph methods will use Dijkstra's algorithm to find the shortest path. The grid methods are tested with Dijkstra and Hierarchical A* (HA*).

The first scenario is the Puddle scene, as shown in Figure 59. The construction times of the different methods are close to each other. BUSHWHACK manages to achieve quite a low construction time because it only has to create the sets of Steiner points. The query times show huge differences between the grid and graph methods. The grid methods have to use such large grids to achieve the same ϵ-values that the query times are extremely high. The path costs of the grid methods are close to the graph methods, but Dijkstra's algorithm has to explore over 2,5 million nodes to find this path. The Hierarchical A* method reduces the query time, but running the search on 7 abstraction layers prevents the Hierarchical A* method to compete with the query times of the pruned-graph methods. The BUSHWHACK method saved some time on constructing the Steiner graph, but it has to spend a lot more resources in the query phase while maintaining the intervals. Due to the higher query time, BUSHWHACK is unable to compete with the Steiner graph and pruned-graph method in terms of total running time.

Puddle from (20,40) to (80,35)



Method	Construction time(ms)	Query time(ms)	Path costs	Nodes explored
Steiner Graph	152.4	1.8	232.3	675
VBP	125.0	0.9	232.3	331
BUSHWHACK	89.0	103.7	232.3	1155
Grid - Dijkstra	135.0	7870.6	235.0	2585261
Grid - HA*	135.0	3040.9	235.0	1108

Figure 59: The resulting paths of all methods: Steiner Graph and VBP (green), BUSHWHACK(darkgreen), Grid - Dijkstra(red) and Grid - HA*(orange) on Puddle with start position (20,40) and goal position (80,35) together with the construction time, query times, path costs and explored nodes.

The Forest scenario is shown in Figure 60. The grid methods are absent because the grid became too large for the memory of the computer running the experiments. In this much larger scene, the BUSHWHACK method is able to outperform the Steiner graph and pruned-graph methods. The time saved in the construction phase compensates the higher query time with a total running time of 214.6

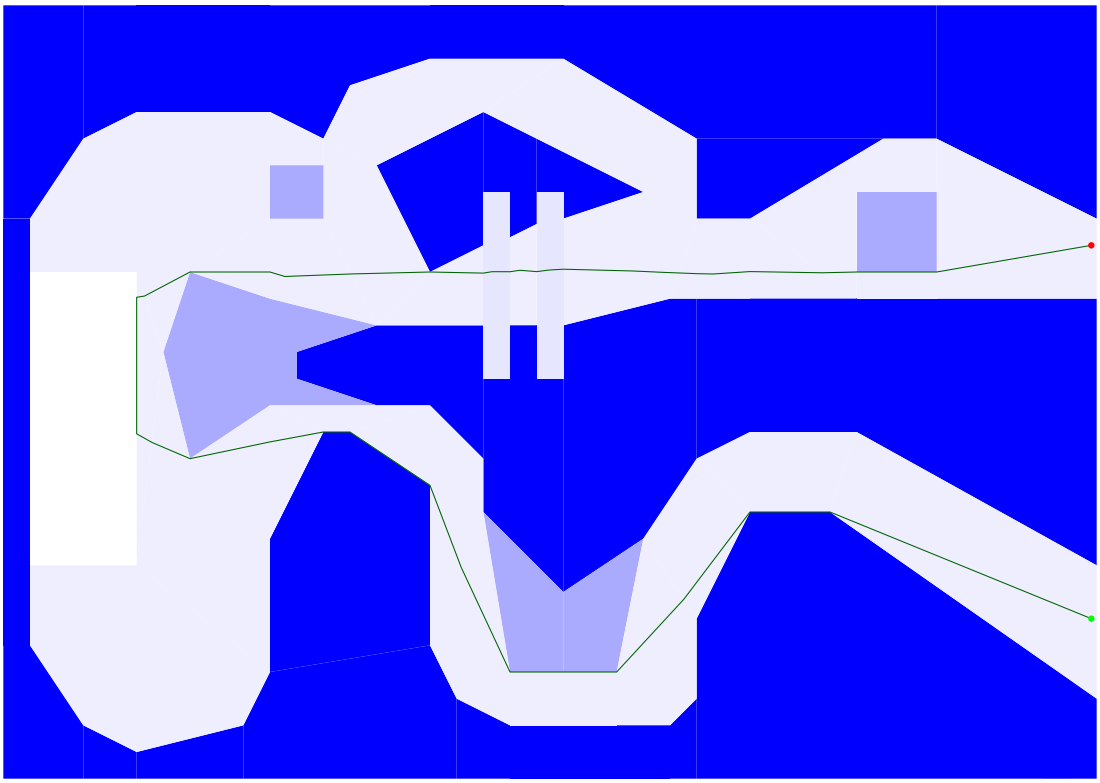
ms versus 227.3 ms of the pruned-graph method.

The Bars scene, Figure 61, shows similar behaviour as the Puddle. The grids are too large to be able to compete with the graph methods, although the path costs are close to each other. The Steiner graph method outperforms the pruned-graph method and BUSHWHACK because this is a small scene, and almost all triangles are selected by the vertex-based pruning method.

The pruned graph shows good results on the High-Low scene, as shown in Figure 62. Neither the Steiner graph nor the BUSHWHACK method come close to the total running times of the vertex based pruning method. The difference in path costs between the grid and graph methods seems a bit bigger in this scenario, but with an ϵ-value of 0.5 the optimal path costs could lie anywhere between 401.5 and 592.9.

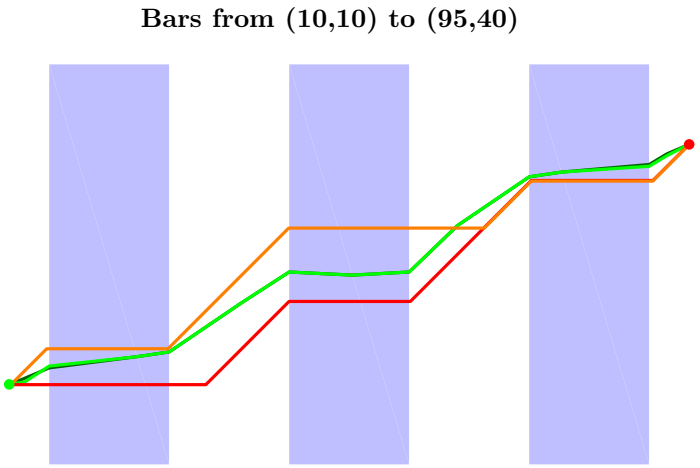
The last scenario is the Zigzag scene from (3,7) to (95,40). The results are shown in Figure 63. The BUSHWHACK method significantly outperforms all other methods on this scenario. A small difference can be seen in the path costs between the Steiner graph and BUSHWHACK. The difference is only 0.008, but the rounding of path costs made the Steiner graph path costs to be rounded down and the BUSHWHACK path costs be rounded up. The difference in path costs is caused by an accumulated rounding error. As previously mentioned in Section 4.2, the BUSHWHACK method requires a small re-triangulation of the triangles that contain the start and goal nodes in order to insert triangle vertices at the start and goal position. Steiner points will be added to these newly inserted edges. If the optimal path happens to go straight from the start or goal position to one of the three surrounding vertices of the original triangle, then the resulting BUSHWHACK path will be an edge-crawling path. Summing up the costs of moving from each Steiner point to the next on this edge can introduce small rounding errors. The Steiner graph has a straight connection between the start/goal position and all surrounding nodes and can avoid these rounding errors.

Forest from (60,2) to (200,2)



Method	Construction time(ms)	Query time(ms)	Path costs	Nodes explored
Steiner Graph	281.1	6.3	2464.2	3825
VBP	225.4	1.9	2464.2	1249
BUSHWHACK	87.1	127.5	2464.2	3416
Grid - Dijkstra	x	x	x	x
Grid - HA*	x	x	x	x

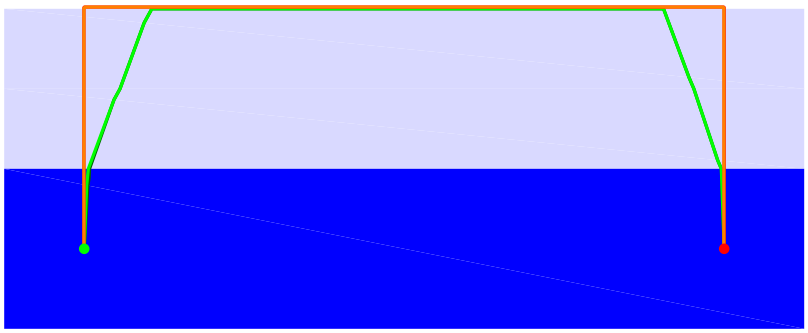
Figure 60: The resulting paths of several methods: Steiner Graph and VBP(green), BUSHWHACK(darkgreen) on Forest with start position (60,2) and goal position (200,2) together with the construction times, query times, path costs and explored nodes. The grid methods are absent because the grid was too large to be contained in the memory of the computer running the experiments.



Method	Construction time(ms)	Query time(ms)	Path costs	Nodes explored
Steiner Graph	120.7	0.9	274.5	364
VBP	131.5	0.9	274.5	323
BUSHWHACK	96.0	95.8	274.4	712
Grid - Dijkstra	130.1	9349.7	279.1	3144252
Grid - HA*	130.1	4242.4	279.1	10094

Figure 61: The resulting paths of several methods: Steiner Graph and VBP(green), BUSHWHACK(darkgreen), Grid - Dijkstra(red) and Grid - HA*(orange) on Bars with start position (10,10) and goal position (95,40) together with the construction times, query times, path costs and explored nodes.

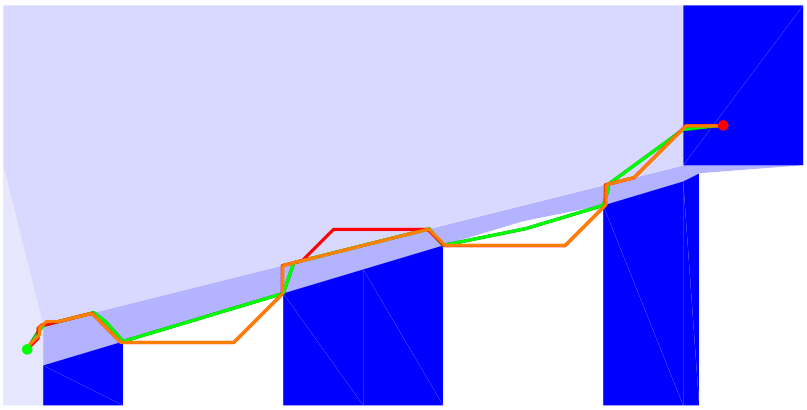
High-Low from (10,10) to (90,10)



Method	Construction time(ms)	Query time(ms)	Path costs	Nodes explored
Steiner Graph	242.1	3.6	592.9	723
VBP	96.7	4.6	592.9	660
BUSHWHACK	97.7	103.6	592.9	883
Grid - Dijkstra	141.9	8961.4	602.3	2905727
Grid - HA*	141.9	3636.2	602.6	12146

Figure 62: The resulting paths of several methods: Steiner Graph and VBP(green), BUSHWHACK(darkgreen), Grid - Dijkstra(red) and Grid - HA*(orange) on High-Low with start position (10,10) and goal position (90,10) together with the construction times, query times, path costs and explored nodes.

Zigzag from (3,7) to (95,40)

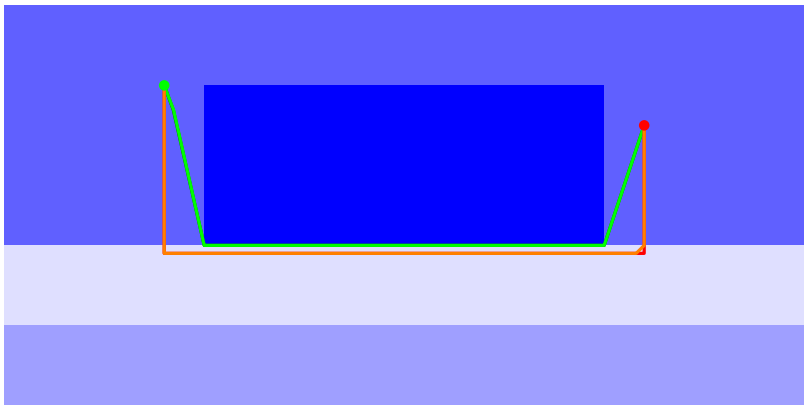


Method	Construction time(ms)	Query time(ms)	Path costs	Nodes explored
Steiner Graph	626.4	14.8	343.8	2465
VBP	565.3	13.7	343.8	2029
BUSHWHACK	125.0	155.7	343.9	2755
Grid - Dijkstra	141.2	9869.3	349.9	3085623
Grid - HA*	141.2	5125.2	363.1	10940

Figure 63: The resulting paths of several methods: Steiner Graph and VBP(green), BUSHWHACK(darkgreen), Grid - Dijkstra(red) and Grid - HA*(orange) on Bars with start position (10,10) and goal position (95,40) together with the construction times, query times, path costs and explored nodes.

Finally, we will also compare the ϵ-approximation methods against the grid methods with cells of unit length. Grid methods are mostly used with cells of unit length in practice. By comparing these grid methods against the ϵ-approximation methods, we can see how the ϵ-approximation methods perform in practice. The results for the Puddle scene are shown in Figure 64. The grid methods achieve the lowest total running times, but the Steiner graph and pruned-graph methods have lower running times. If several queries have to be performed on a single scene, then the graph methods can outperform the grid methods. The Forest scene shows similar behaviour in Figure 65. In this scene the graph methods can already outperform the grid methods if four or more queries are performed on the scene. As such, the graph methods can compete with the grid methods in practical applications that perform multiple queries on the same scene. Additionally, the computed paths of the graph methods have lower ϵ-values and are closer to the optimal path costs.

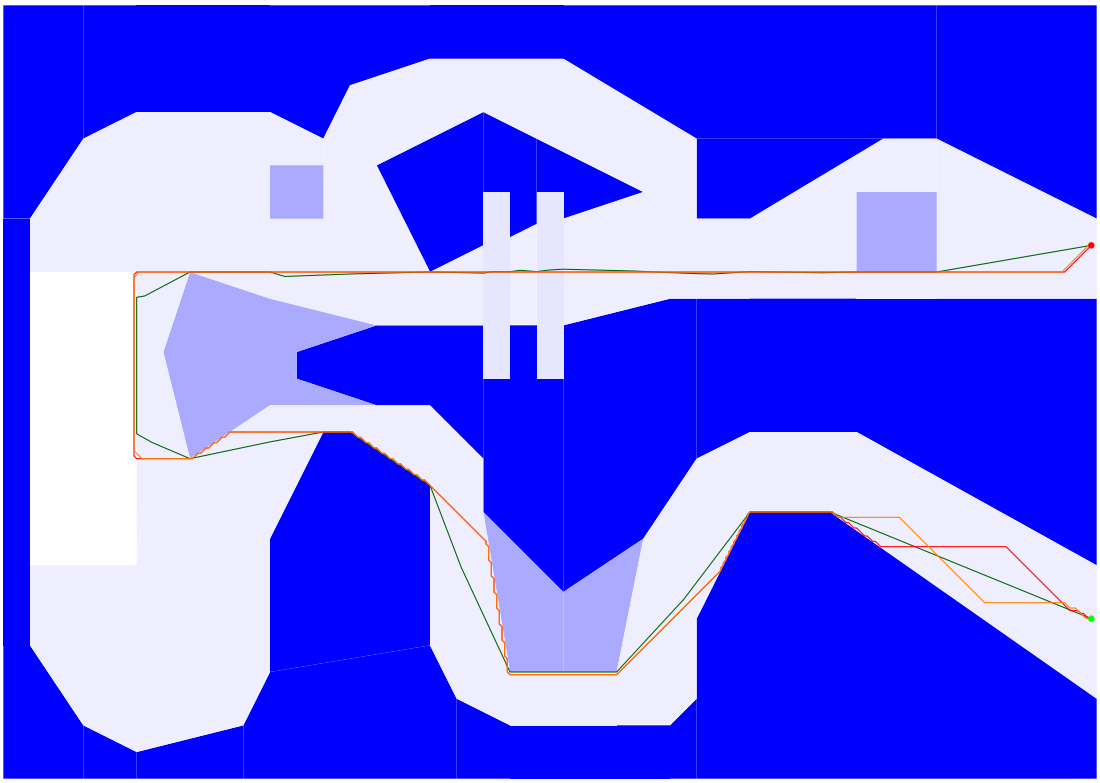
Puddle from (20,40) to (80,35)



Method	ϵ-value	Construction time(ms)	Query time(ms)	Path costs	Nodes explored
Steiner Graph	0.5	152.4	1.8	232.3	675
VBP	0.5	125.0	0.9	232.3	331
BUSHWHACK	0.5	89.0	103.7	232.3	1155
Grid - Dijkstra	10.5824	0.8	4.78	241.0	4497
Grid - HA*	10.5824	0.8	3.94	241.2	490

Figure 64: The resulting paths of all methods: Steiner Graph and VBP (green), BUSHWHACK(darkgreen), Grid - Dijkstra(red) and Grid - HA*(orange) on Puddle with start position (20,40) and goal position (80,35) together with the ϵ-value, construction time, query times, path costs and explored nodes.

Forest from (60,2) to (200,2)



Method	ϵ-value	Construction time(ms)	Query time(ms)	Path costs	Nodes explored
Steiner Graph	0.5	281.1	6.3	2464.2	3825
VBP	0.5	225.4	1.9	2464.2	1249
BUSHWHACK	0.5	87.1	127.5	2464.2	3416
Grid - Dijkstra	50.0824	4.2	151	2555.4	83250
Grid - HA*	50.0824	4.2	88.9	2556.1	4960

Figure 65: The resulting paths of several methods: Steiner Graph and VBP(green), BUSHWHACK(darkgreen), Grid - Dijkstra(red) and Grid - HA*(orange) on Forest with start position (60,2) and goal position (200,2) together with the ϵ-value, construction times, query times, path costs and explored nodes.

8 Conclusions

In this thesis, we have searched for methods to improve current solutions to the Weighted Region problem [3]. We first discussed A* grid approaches. We created a formula that can be used to calculate the ϵ-value of paths returned by grid methods, based on the minimum and maximum weights in the scene and the resolution of the grid. The A* heuristics have been examined to address the problems of A* on weighted regions.

The Hierarchical A* method has been adapted to work with weighted regions. Two abstraction methods, resolution-based and mapping-based abstraction, have been compared to see which method yields the best abstraction layers. The discretization errors in the resolution-based abstraction cause severe overestimations of the heuristic values, while the mapping-based abstraction only causes underestimations of heuristic values, which allows the heuristic to maintain admissible.

Experiments have been conducted to find optimal values for the two parameters of the Hierarchical A* method. The first parameter controls the number of abstraction layers that are used in the hierarchy. The experiments showed that this parameter is dynamic and should be adjusted to each scene. The running times of the Hierarchical A* method decrease with each added abstraction layer, until the resolution of the abstraction layer is lower than 200. Abstraction layers with such a small resolution can efficiently be explored by Dijkstra's algorithm, and adding more layers will only lead to more overhead. We conclude that one should use as many abstraction layers, as long as there is only one layer with a resolution lower than 200 in the hierarchy.

The second parameter of the Hierarchical A* method is the abstraction factor. This factor controls the number of nodes that are mapped to a single node on a higher abstraction layer. Using higher values decreases the query times, but has a negative influence on the path costs. We have determined the optimal value for the abstraction factor to be 2. In addition, we have compared different heuristics with each other. The Hierarchical A* method improves upon the performance of A* if the environments are large, while not significantly under-performing in smaller scenes. The resulting paths for Hierarchical A* are near-optimal and only overestimated the total costs by a maximum of 0.7 for path costs ranging from 217 to 2555 in the tested scenes.

The bounds of the ϵ-values for ϵ-approximation methods have been analyzed. The minimum ϵ-value must always be greater than 0. The maximum ϵ-value depends on the ϵ-approximation method that is used. For the ϵ-approximation method [5] that places Steiner points on the edges of the triangulation, the maximum ϵ-value is $\frac{1}{2}$. For the ϵ-approximation method [6] that places Steiner points on the bisectors of the triangle edges of the triangulation, the maximum ϵ-value is $3\frac{1}{2}$.

Several new methods have been devised to improve the graph-based approaches to the Weighted Region Problem. The first method aims at reducing the number of nodes explored by A* on the graph by creating a better guidance through the use of a grid path. The Hierarchical A* heuristic takes a similar approach and has been used to create heuristic values for the A* method on the Steiner graph. This method suffers from higher path costs. Several attempts have been made to correct the Hierarchical A* heuristic to yield optimal paths, but the Hierarchical A* method is not suited for graphs because the heuristic is not monotone.

The second method utilizes the A* grid path to prune the graph. This approach shows an improvement over the Steiner graph method in two ways. Firstly, the number of Steiner points that need to be added and connected can be reduced, allowing for lower construction times of the graph. Secondly, the pruned graph can be a lot smaller and thus reduce the number of nodes that can be explored by graph-search algorithms such as Dijkstra's algorithm. Three approaches have been devised to prune the graph. The first method creates the pruned graph by determining all triangles that are intersected by the grid path. The second method selects a set of vertices near the bending points of the grid path and combines all incident edges of the vertices with all edges that are intersected by the grid path. The pruned graph is constructed by adding Steiner points on all of these edges. The last method creates a

pruned graph out of a set of edges that lie within a certain radius of the grid path.

The pruned-graph methods have been compared against the Steiner graph and against each other. The pruned-graph methods showed to be able to reduce the construction and query costs, when the ϵ-values are low and the optimal path does not intersect with all triangles in the scene. Except for the Bars scenario, the vertex-based pruning method has been able to construct the pruned graph faster than the Steiner graph and achieve lower query times than the Steiner graph method. The pruned-graph methods do not always return the optimal path. If the homotopy of the grid path is different from the homotopy of the optimal graph path, then the pruned-graph methods may prune too much and the optimal path might no longer exist within the pruned graph. Experiments have been conducted to find the success rate of all pruned-graph methods. The vertex-based pruning method achieved the highest success rate with a rating of 97.3%.

Finally, we have compared the grid and graph methods while adjusting the grid methods ϵ-value by increasing the resolution of the grid. The increase of grid resolution has such a huge impact on the grid methods that the query times became extremely high. The BUSHWHACK method was able to achieve the best running times on the complex Zigzag scene and large Forest scene. The pruned-graph method performed best on the other scenarios.

Overall, we can state that the gap between the grid and graph approaches has been reduced. The newly introduced pruned-graph methods can be used to plan near-optimal paths for multiple characters in real-time applications due to its low running times. Furthermore, using the Hierarchical A* method can improve running times of grid-based methods, allowing for higher resolution grids to return paths with a lower ϵ-value. Games and virtual environments could benefit from using the Hierarchical A* method when the environments are not too big and path for many characters need to be planned. When the environments are large or if precision is more important, then the pruned graph methods will outperform the current grid and graph methods, as well as the Hierarchical A* method.

9 Future work

In this section, we will discuss how the algorithms and methods presented in this thesis might be further improved, and we will present several research questions that arose from the results presented in this thesis.

9.1 BUSHWHACK improvements

The BUSHWHACK method could be adapted to work with pruned graphs. The running times could be improved in two ways. The first idea is to only add Steiner points to the edges selected by the pruned-graph method. The second idea is to reduce the number of intervals that need to be maintained by BUSHWHACK by using pruned-graphs.

BUSHWHACK uses a wavefront expansion comparable with Dijkstra's algorithm. We might be able to improve the performance if we use a heuristic to guide the expansion. Adding a heuristic to BUSHWHACK is complicated, because BUSHWHACK maintains monotonically increasing intervals. However, BUSHWHACK is devised as a general method that could be applied to any piecewise pseudo-Euclidean optimal path problem. The only requirement is that the local extrema on an edge can be calculated in $O(\log m)$ time, where m is the number of Steiner points on that edge. Finding these local extrema might be difficult because the combination of two monotonic intervals (one for the heuristic, one for the costs to the start) could lead to an interval with two extrema. When two extrema are found, the interval should be split up in four monotonic increasing intervals. Another possibility is that the sum of the heuristic and the costs to the start add up to the same value for several neighbouring Steiner points. Additional rules need to be be defined to handle such cases.

9.2 Weight-based pruning

For specific problems, one might also use a weight-based pruning method to prune the Steiner graph. For instance, if path planning is done for virtual pedestrians in a city environment, one could only add Steiner points to the triangles that represent the sidewalks in the environment. In order to maintain dynamic pedestrian behaviour, such as avoiding a blocked sidewalk by using the read next to the sidewalk, one could additionally use Steiner points on all triangles adjacent to sidewalk triangles. This approach does not work in the general WRP because there is always a possible scenario where a character has to cross regions with all possible weights in order to find the optimal path. If a character needs to cross all regions, then all regions need to be covered with Steiner points, and the pruned graph will be equal to the normal Steiner graph.

9.3 Hierarchical Steiner graphs

The Hierarchical A* heuristic is not compatible with the graph-search methods because the difference between the ϵ-values of the grid and Steiner graph are too great. One could also construct a hierarchical structure that consists of Steiner graphs at different ϵ-values, where each layer creates heuristic values for the lower layers. Finding a path for $\epsilon = 0.0001$ can take hours with current approaches, but with a hierarchy that produces good heuristic values, the search could be reduced significantly.

9.4 Improvement of the Hierarchical A* approach

Hierarchical A* tends to overestimate the costs to the goal, as shown in Section 7.2.2. However, these near-optimal paths differ from the optimal path by not more than one node. If optimal paths are a hard requirement, one could examine all nodes that lie within a one-node radius of the path returned by

Hierarchical A* and update parent pointers if shorter paths are found. This is a simple postprocessing step that could be performed in $O(l)$ time, where l is the length of the near-optimal path.

9.5 Final words

This thesis provides some new insights and new methods to tackle the Weighted Region problem. The gap between grid and graph approaches has been reduced, but is still far from being closed. There are many open questions left and there is still room for improvement. Future gaming and simulation applications can benefit from answering those questions.

References

[1] I. Karamouzas, R. Geraerts, and M. Overmars. Indicative routes for path planning and crowd simulation. *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 113–120, 2009.

[2] N.S. Jaklin, A.F Cook IV, and R. Geraerts. Real-time path planning in heterogeneous environments. *Computer Animation and Virtual Worlds (CAVW)*, 24(3):285–295, 2013.

[3] J.S.B. Mitchell and C.H. Papadimitriou. The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM*, 38(1):18–73, 1991.

[4] C.S. Mata and J.S.B. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions (extended abstract). *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 264–273, 1997.

[5] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.R. Sack. An ϵ-approximation algorithm for weighted shortest paths on polyhedral surfaces. *Algorithm Theory, SWAT'98*, 1432:11–22, 1998.

[6] L. Aleksandrov, A. Maheshwari, and J.R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *Journal of the ACM*, 52(1):25–53, 2005.

[7] L. Aleksandrov, H.N. Djidjev, H. Guo, A. Maheshwari, D. Nussbaum, and J.R. Sack. Algorithms for approximate shortest path queries on weighted polyhedral surfaces. *Discrete & Computational Geometry*, 44(4):762–801, 2010.

[8] R. Stouffs, R. Krishnamurti, S. Lee, and I.J. Oppenheim. Construction process simulation with rule-based robot path planning. *Automation in Construction*, 3(1):79–86, 1994.

[9] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. *SIAM Journal on Computing*, 15(1):193–215, 1986.

[10] J.L. De Carufel, C. Grimm, A. Maheshwari, M. Owen, and M. Smid. Unsolvability of the weighted region shortest path problem. *European Workshop on Computational Geometry*, pages 65–68, 2012.

[11] Z. Sun and J. Reif. Bushwhack: An approximation algorithm for minimal paths through pseudo-euclidean spaces. *Algorithms and Computation*, 2223:160–171, 2001.

[12] S.M. LaValle. *Planning Algorithms*. Cambridge University Press, first edition, 2006.

[13] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions On Systems Science And Cybernetics*, 4(2):100–107, 1968.

[15] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[16] R.C. Holte, M.B. Perez, R.M. Zimmer, and A.J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. *Proceedings of the National Conference on Artificial Intelligence*, pages 530–535, 1996.

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

[18] L.E. Kavraki, P. Švestka, J. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Automation Science and Engineering*, 12(4):566–580, 1996.

[19] V. Boor, M.H. Overmars, and A.F. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1018–1023, 1999.

[20] S.A. Wilmarth, N.M. Amato, and P.F. Stiller. Maprm: A probabilistic roadmap planner with sampling on the medial axis of the free space. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1024–1031, 1999.

[21] P. Khosla and R. Volpe. Superquadratic artificial potentials for obstacle avoidance and approach. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1778–1784, 1988.

[22] E. Rimon and D. Koditschek. Exact robot navigation using artificial potential fields. *IEEE Transactions on Automation Science and Engineering*, 8:501–508, 1992.

[23] C. Connolly and R. Grupen. Harmonic control. *IEEE International Symposium on Intelligent Control*, pages 503–506, 1998.

[24] J. Hershberger and S. Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.

[25] H. Rohnert. Moving a disc between polygons. *Algorithmica*, 6:182–191, 1991.

[26] L.P. Chew. Planning the shortest path for a disc in $O(n^2 logn)$ time. *Proceedings of the first annual symposium on Computational geometry*, pages 214–220, 1985.

[27] J.A. Storer and J.H. Reif. Shortest paths in the plane with polygonal obstacles. *Journal of the ACM*, 41(5):982–1012, 1994.

[28] A. Kamphuis and M.H. Overmars. Finding paths for coherent groups using clearance. *Proceedings of the ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 19–28, 2004.

[29] R. Geraerts. *Planning short paths with clearance using explicit corridors*, pages 1997–2004, 2010.

[30] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1):209–233, 1987.

[31] K. Yang and S. Sukkarieh. Planning continuous curvature paths for UAVs amongst obstacles. *Proceedings of the Australasian Conference on Robotics and Automation*, 2008.

[32] J Pan, L Zhang, and D Manocha. Collision-free and smooth trajectory computation in cluttered environments. *The International Journal of Robotics Research*, 31(10):1155–1175, 2012.

[33] Y. Zhao and P. Tsiotras. A quadratic programming approach to path smoothing. *American Control Conference*, pages 5324–5329, 2011.

[34] R. Wein, J.P. van den Berg, and D. Halperin. The visibility–voronoi complex and its applications. *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 63–72, 2005.

[35] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[36] J.S.B. Mitchell. *Planning shortest paths*. dissertation, 1986.

[37] J.S.B. Mitchell, D.M. Mount, and C.H. Papadimitriou. The discrete geodesic problem. *SIAM Journal on Computing*, 16(4):647–668, 1987.

[38] J. Reif and Z. Sun. An efficient approximation algorithm for weighted region shortest path problem. *Algorithmic and Computational Robotics: New Directions: the Fourth Workshop on the Algorithmic Foundations of Robotics*, page 191, 2001.

[39] L. De Filippis, G. Guglieri, and F. Quagliotti. A minimum risk approach for path planning of UAVs. *Journal of Intelligent & Robotic Systems*, 61:203–219, 2011.

[40] N. MacMillan, R. Allen, D. Marinakis, and S. Whitesides. Risk averse motion planning for a mobile robot. 2011.

[41] J. Chestnutt, K. Nishiwaki, J. Kuffner, and S. Kagami. An adaptive action model for legged navigation planning. *7th IEEE-RAS International Conference on Humanoid Robots*, pages 196–202, 2007.

[42] D. Reece, M. Kraus, and P. Dumanoir. Tactical movement planning for individual combatants. *Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation*, 2000.

[43] A. Kamphuis, M. Rook, and M.H. Overmars. Tactical path finding in urban environments. *First International Workshop on Crowd Simulation*, 2005.

[44] M. Booth and Turtle Rock Studios. The making of the official counter-strike bot, 2004.

[45] W. Van der Sterren. Tactical path-finding with A*. *Game Programming Gems*, 3:294–306, 2002.

[46] Guerilla. `http://www.guerrilla-games.com/`.

[47] A.J. Champandard. The core mechanics of influence mapping. `http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/the-core-mechanics-of-influence-mapping-r2799`, 2011.

[48] J. Hagelbäck. Using potential fields in a real-time strategy game scenario. `http://aigamedev.com/open/tutorials/potential-fields/`, 2009.

[49] D. Blythe. Opengl manual: 3.5.1 basic polygon rasterization. `http://www.opengl.org/documentation/specs/version1.1/glspec1.1/node54.html#SECTION00651000000000000000`, 1997.

[50] W. Van Toll, A.F. Cook, and R.J. Geraerts. Navigation meshes for realistic multi-layered environments. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3526–3532, 2011.

A Derivative of ϵ

$$\epsilon = \frac{x\sqrt{2} + m - x - \sqrt{x^2 + m^2}}{\sqrt{x^2 + m^2}}$$

$\epsilon' = (\frac{f}{g})' = \frac{f'g - fg'}{g^2}$, where

$$f = x\sqrt{2} + m - x - \sqrt{x^2 + m^2}$$
$$f' = \sqrt{2} - 1 - \frac{x}{\sqrt{x^2 + m^2}}$$

$$g = \sqrt{x^2 + m^2}$$
$$g' = \frac{1}{2\sqrt{x^2 + m^2}} * 2x = \frac{2x}{2\sqrt{x^2 + m^2}} = \frac{x}{\sqrt{x^2 + m^2}}$$

$$\epsilon' = \frac{((\sqrt{2} - 1 - \frac{x}{\sqrt{x^2+m^2}}) * \sqrt{x^2 + m^2}) - ((x\sqrt{2} + m - x - \sqrt{x^2 + m^2}) * \frac{x}{\sqrt{x^2+m^2}})}{\sqrt{x^2 + m^2}^2} \quad \epsilon' = \frac{m(\sqrt{2}m - m - x)}{(m^2 + x^2)^{\frac{3}{2}}}$$

B Finding the x-location of the maximum value of ϵ

$$\epsilon' = 0$$
$$\frac{m(\sqrt{2} * m - m - x)}{(m^2 + x^2)^{\frac{3}{2}}} = 0$$
$$m(\sqrt{2} * m - m - x) = 0$$
$$\sqrt{2} * m - m - x = 0$$
$$x = \sqrt{2} * m - m$$

C Finding the maximum value of ϵ

$$\epsilon = \frac{(\sqrt{2} * m - m)\sqrt{2} + m - (\sqrt{2} * m - m) - \sqrt{(\sqrt{2} * m - m)^2 + m^2}}{\sqrt{(\sqrt{2} * m - m)^2 + m^2}}$$

$$\epsilon = \frac{2m - \sqrt{2} * m + m - (\sqrt{2} * m - m) - \sqrt{(3 - 2\sqrt{2})m^2 + m^2}}{\sqrt{(3 - 2\sqrt{2})m^2 + m^2}}$$

$$\epsilon = \frac{3m - \sqrt{2} * m - (\sqrt{2} * m - m) - \sqrt{(4 - 2\sqrt{2})m^2}}{\sqrt{(4 - 2\sqrt{2})m^2}}$$

$$\epsilon = \frac{(3 - \sqrt{2})m - (\sqrt{2} - 1)m - \sqrt{(4 - 2\sqrt{2})}\sqrt{m^2}}{\sqrt{(4 - 2\sqrt{2})}\sqrt{m^2}}$$

$$\epsilon = \frac{(3 - \sqrt{2} - \sqrt{2} + 1)m - \sqrt{(4 - 2\sqrt{2})} * m}{\sqrt{(4 - 2\sqrt{2})} * m}$$

$$\epsilon = \frac{(4 - 2\sqrt{2})m - \sqrt{(4 - 2\sqrt{2})} * m}{\sqrt{(4 - 2\sqrt{2})} * m}$$

$$\epsilon = \frac{(4 - 2\sqrt{2})m}{\sqrt{(4 - 2\sqrt{2})} * m} - \frac{\sqrt{(4 - 2\sqrt{2})} * m}{\sqrt{(4 - 2\sqrt{2})} * m}$$

$$\epsilon = \frac{(4 - 2\sqrt{2})m}{\sqrt{(4 - 2\sqrt{2})} * m} - 1$$

$$\epsilon = \frac{(4 - 2\sqrt{2})}{\sqrt{(4 - 2\sqrt{2})}} - 1$$

$$\epsilon = \sqrt{(4 - 2\sqrt{2})} - 1$$