2014

# Transit Search

Justin Moore
*University of Windsor*

# Transit Search

by
Justin Moore

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfilment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2014

TRANSIT SEARCH

BY

JUSTIN MOORE

APPROVED BY:

M. Hlynka
Department of Mathematics and Statistics

Z. Kobti
School of Computer Science

S. Goodwin, Advisor
School of Computer Science

May 15, 2014

## Author’s Declaration of Orignality

## Abstract

Pathfinding is an area of research and of practical importance in Computer Science. The A* algorithm is well known as a pathfinding algorithm that finds optimal paths. As demands increase on pathfinding systems, we need faster algorithms to keep up. We propose an algorithm that uses preparation to partition the search space into regions that we may be able to skip while searching. We call this algorithm Transit Search, and by potentially skipping regions of the search space we can expand less nodes than A*. It accomplishes this by using a maximum allowed heuristic value to tell the search if it is possible the goal is within a region. If this isn't the case the algorithm can skip the region entirely, saving us from expanding unneeded nodes.

# Acknowledgements

I would like to thank Dr. Scott Goodwin, my supervisor, for accepting me as his Master's student and assisting me through the completion of my thesis. His guidance was instrumental in my journey from A* to the stars. I would also like to thank my thesis committee for their time in reviewing my thesis as well as attending my proposal and defense. Their encouragement and advice helped shape my thesis to its present state. Finally, I would like to thank my family and friends for tolerating me during my graduate studies and supporting me along the way.

# Table of Contents

# List of Figures

# List of Algorithms

# List of Symbols

**G**

A graph composed of nodes and edges

**c(a,b)**

The actual cost of the shortest path between nodes a and b

$\gamma$

A goal node for a search problem

**s**

The start node of a search problem

**h(a,b)**

A heuristic estimate of the shortest path's cost between nodes a and b

**g(a,b)**

The minimum cost path from nodes a to b at the current point in the search

**h(n)**

A heuristic estimate of the shortest path cost from n to the goal state

**g(n)**

The minimum cost path from the start node to the node n at the current point in the search

**f(n)**

The f-score of the A* algorithm; the value that is to be minimized

$P_{a,b}$

A shortest path from nodes a to b

$\boldsymbol{\epsilon}$

A real valued weight of the range [0,$\infty$)

**h*(a,b)**

A perfect heuristic estimate of the cost from nodes a to b

**h*(n)**

A perfect heuristic estimate of the cost from n to the goal state

**p**

c(s,$\gamma$) or equivalently g($\gamma$)

$\mathbf{O}(f)$

The worst case computational complexity of the function $f$

$\Upsilon$

The set of boundaries in a well-formed region

# 1 Introduction

## 1.1 Problem Domain

The problem of pathfinding is finding a path between two nodes in a graph. If we let $a$ and $b$ be nodes in the graph $G$, a path is defined as a series of nodes starting with $a$ and ending with $b$ such that each successive node is connected to the previous. Often we are concerned not with any path, but specifically paths which are of the least cost. The cost of a path is usually defined as the sum of the weights along path.

Pathfinding has many uses in practical situations. Often real world data is converted into a graph and searched to find shortest routes. Pathfinding is used extensively in video games to plan the path that agents take. The game world is usually discretized into a graph with the nodes being locations agents can travel to and the edges of the graph being the paths between them. The weights on the edges are either cost of travel or distance between the nodes in the search space. The pathfinding used in video games needs to be fast to allow for other systems in the game to have enough time to run. Pathfinding also finds use in robotics and simulations for planning the routes an agent may take.

More abstractly problems that involve searching can be solved with pathfinding algorithms. Puzzles like the 15-puzzle can represent its configurations as nodes and and edges as the allowed moves between them. This allows us to search for the smallest set of moves from a start configuration towards a goal one. It can also be used to find solutions to NP-Hard problems such as the Travelling Salesmen problem.

## 1.2 Thesis

This thesis introduces a series of new algorithms that attempt to expand less nodes than A* (a well-known search algorthm) does. The research culminates in the development of the Transit Search algorithm. This algorithm uses preparation to partition the search space into Transit Regions allowing the search to skip over sections of the graph when searching. It does this by utilizing a Maximum Heuristic value, which indicates how far an agent can travel within a region along shortest paths. If we determine that a goal node is farther than this distance we know that the goal is not within the region and can skip searching it. Our theories are applicable to any non-negative weighted graph and use an admissible heuristic function much like A*.

We implement a version of Transit Search for uniform cost grid graphs, as they are common search spaces in the video game industry. We also prove that certain types of regions, that are detectable by game designers and algorithms, have the needed properties for transit search. Experiments are done comparing A* and Transit Search along with weighted variants of both algorithms. These will show that Transit Search is on average faster than A* and is comparable to the Weighted A* alternatives.

# 2 Background

## 2.1 A*

The A* algorithm [HNR68] is a pathfinding algorithm that uses heuristic information to guide its search toward a goal. It is an improvement over Djikstra's algorithm [Dij59] in that it is informed about the search space by means of an heuristic function. This function is used to measure the expected cost to reach a goal node. A* is used to search within non-negative weighted graphs composed of nodes and edges, from a start node $s$ to a goal node $\gamma$. That is, it solves the single-source shortest path problem on these graphs. A* is complete, in that if a path between $s$ and $\gamma$ exists A* will find it. The algorithm always finds a least cost path, that is, no other path has a lower cost. If there are multiple paths that have the least cost, A* will return one of them.

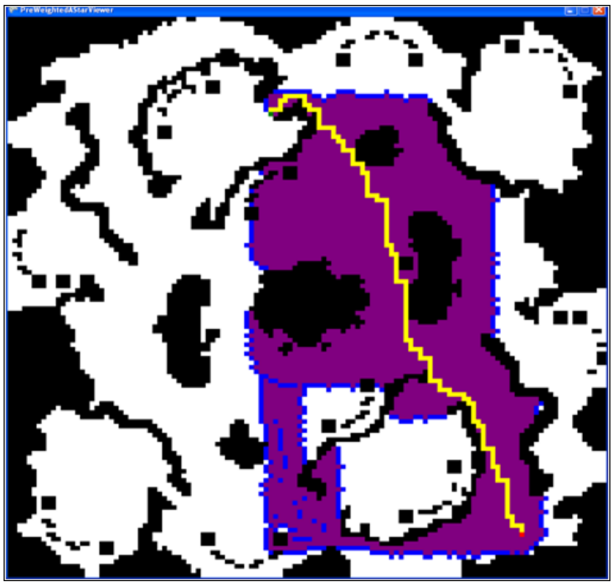

Figure 1: An example of an A* found path, showing the closed list

A* starts at the start node and evaluates the nodes connected to it. This process is known as expanding a node. Every node that is expanded is placed in a closed list, so we don't re-expand it in the future. All the connected nodes are evaluated and placed onto an open list. These are the nodes to be expanded in the future. The algorithm continues this process until either a goal is found, or we have added every node to the closed list in which case the path does not exist.

To decide the order that we expand nodes in, we calculate an f-score. The f-score is the sum of 2 functions,

1. $g(n)$ This is the cost along the current path from the start node to the node n. This is the value that we want to minimize, and when $n = \gamma$ this is the length of the shortest path from $s$ to $\gamma$

2. $h(n)$ This is the heuristic estimate of the cost from $n$ to a goal $\gamma$

Thus the f-score calculation looks like

$$f(n) = g(n) + h(n) \tag{1}$$

This is contrast to Djikstra's algorithm which essentially only considers $g(n)$ and in contrast to a greedy algorithm that would only take into account $h(n)$. This takes into account equally the cost incurred so far and the expected cost of the rest of the path. As we expand nodes we always take the node with the lowest f-score. This represents what we believe to be the next node on the shortest path to the goal.

It often happens that there are multiple nodes with the same f-score and these are referred to as the fringe. This is the set of nodes that A* could expand next. The rule we use to choose which node to expand on the fringe is called a tie breaking rule. Changing the tie-breaking rule can have a drastic effect when searching through graphs that can have a large fringe, sometimes speeding search up by an order of magnitude.

There are some restrictions on what the heuristic function can be to ensure that A* returns the shortest path. The first of which is admissibility. We say a heuristic function is admissible if it never overestimates the cost to reach the goal. More formally a heuristic is admissible if,

$$\forall n \in G, h(n, \gamma) \leq c(n, \gamma) \tag{2}$$

Admissible heuristics are optimistic as they often return a value that is less than the actual cost to the goal.

Another restriction usually put on the heuristic is that of consistency. We say a heuristic is consistent if,

$$\forall a, b \in G, h(a, \gamma) \leq c(a, b) + h(b, \gamma) \tag{3}$$

given that b is a successor of a. If a heuristic is consistent it ensures that A* does not need to recheck nodes on the closed list. It also lets A* be optimal up to a tie breaking rule. This means that A* will expand a subset of the nodes of any other equally informed search algorithm, if it is given the correct tie-breaking rule. This is important to note as other search algorithms using the same information as A*, could be rewritten as A* with a particular tie breaking rule.

The A* algorithm is still an active area of research, with many other pathfinding algorithms being based on it, or being variants of it. Work has been done to produce algorithms that limit the time A* uses such as TBA* [BBS09]. Other algorithms have dropped the shortest path guarantee in favor of speed such as Weighted A* and HPA* [HPZM96]. There has also been work in reducing the number of nodes expanded on specific search spaces, uniform grids being popular, in algorithms like RSR.

**Algorithm 1:** A*

```
add start to open list;
while open isn't empty do
    min = find min in open;
    if min is goal then
        return success;
    end
    add min to closed list;
    foreach n in neighbours of min do
        if n is in closed list then
            continue;
        end
        if n was already opened then
            change f-score if better;
        else
            n's g-score = min's g-score + edge cost from min to n;
            n's h-score = h(n,goal);
            n's f-score = n's g-score + n's h-score;
            n's parent = min;
            add n to the open list;
        end
    end
end
return failure;
```

## 2.2 Weighted A*

The Weighted A* search algorithm [Pea84] extends A* by using a weight on the heuristic in order to speed up searches. This makes a node's f-score more dependent on the heuristic estimate to the goal and less on the cost already incurred along the path. If we let $\epsilon$ be the weight, a constant positive real value, then we can define the calculation of the f-score as,

$$f(n) = g(n) + \epsilon * h(n) \tag{4}$$

Depending on the value that we choose for $\epsilon$, the resultant effect on the search can differ. Choosing an $\epsilon$ value of 1 we see that the formula for the f-score collapses to that of A*'s f-score calculation. In this case Weighted A* would perform identical to A*, and this shows that Weighted A* is a generalization of the A* search search algorithm.

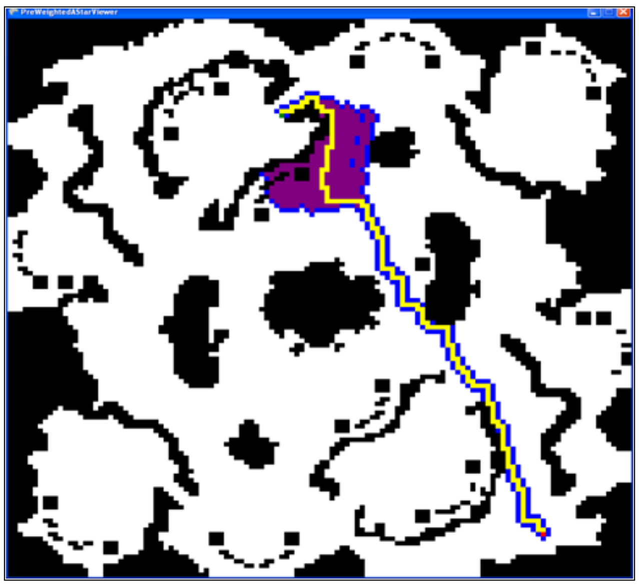

Figure 2: An example of an Weighted A* found path, showing the closed list

If we let the weight be less than one ($\epsilon < 1$) than f-score will be more sensitive to the path's cost and less sensitive to the heuristic. Viewing $\epsilon * h(n)$ as some second heuristic function $h'(n)$, we can examine the effect this has on the search. Assuming that $h(n)$ is admissible, then $h'(n)$ is also admissible since it must be less than equal $h(n)$, and in no way could overestimate the actual path cost. Since $h'(n)$ is admissible, values of $\epsilon < 1$ maintain the shortest path guarantee of A*.

Even though it maintains the shortest path, these values of epsilon change the way the algorithm expands nodes. Since we lower the effect $h(n)$ has on the f-score, we increase the relative effect that $g(n)$ has. Thus our algorithm is more likely to expand nodes that have lower costs from the start node. This gives it the effect of the fringe expanding more like a wavefront, being more conservative than a weight of 1. In fact if we let $\epsilon = 0$ then the f-score equation doesn't consider the heuristic function at all, and our algorithm will run like Dijkstra's algorithm. This extreme makes the search uniform and loses the benefit of heuristics. Given that $\epsilon < 1$ we put less trust in the heuristic and diminish its effect. Since we try to choose our heuristic to be a good estimator of remaining cost, this only decreases the effectiveness of the algorithm.

We can however choose $\epsilon$ to be greater than 1 for a different result. When we have $\epsilon > 1$ the heuristic function has more of an effect on the calculation of the f-score. This makes our algorithm more likely to expand nodes that are seen closer to a goal, based on the heuristic. This puts more trust in the heuristic function, and since the heuristic function is chosen to be a good estimator of actual cost, we would expect our algorithm to perform better.

The downside to choosing $\epsilon > 1$ is the effect it has on the path found. If $h(n)$ is admissible there is no guarantee that $h'(n)$ is also admissible, since

$$\epsilon * h(n) \geq h(n) \tag{5}$$

This voids the conditions of A* that ensures that it finds an optimal path. Thus for Weighted A* the paths returned may not be optimal (although they can be), and we have to trade-off between path quality and speed. It is known though that the cost of the paths returned by Weighted A* can be no more than $\epsilon$ times the cost of the path with the least cost. A calculated choice of $\epsilon$ can then be used to balance the need for speed against the bounded sub-optimality of the paths found.

The algorithm for Weighted A* is very similar to that of A*. The singular change is that we replace $h(n)$ with $h'(n) = \epsilon * h(n)$, as can be seen in the pseudocode below.

**Algorithm 2:** Weighted A*

```
add start to open list;
while open isn't empty do
    min = find min in open;
    if min is goal then
        return success;
    end
    add min to closed list;
    foreach n in neighbours of min do
        if n is in closed list then
            continue;
        end
        if n was already opened then
            change f-score if better;
        else
            n's g-score = min's g-score + edge cost from min to n;
            n's h-score = $\epsilon$ * h(n,goal);
            n's f-score = n's g-score + n's h-score;
            n's parent = min;
            add n to the open list;
        end
    end
end
return failure;
```

## 2.3 RSR

Rectangular Symmetry Reduction or RSR [HBK11], is an algorithm that prunes nodes within rectangles of uniform cost grid graph. It does this by exploiting the symmetry of obstacle-free rectangles so that it can create a smaller graph composed of the rectangle perimeters. It stores a link from every node in the original graph to the rectangle it is within, so it can tell when searching whether or not a node is in the rectangle of the node being expanded. This requires a preparation step to find the rectangles and create the new graph. It also requires extra memory of the complexity $O(n)$, where n is the number of nodes in the graph. In addition information on the rectangles is saved, namely the dimensions of the rectangle, and this requires space proportional to the number of rectangles.

The search phase inserts nodes from the original graph into the new graph and searches along the perimeter of the rectangles. In order to connect nodes from one side of a rectangle to nodes on the other side, edges are created to the nodes on the other side using the rectangle's information. This is done during the search and because of the simplicity of rectangles, can be done in $O(1)$ time. The algorithm also applies some extra online pruning to remove evaluating nodes that could be reached in shorter distance from another node.

# 3 Transit Region Search

## 3.1 Motivation

We want a faster search algorithm so that we can perform more searches and allow more time for other algorithms in the program to run. This is more and more important as applications keep increasing the number of agents requiring the use of pathfinding. Also as games add more complicated physics, graphics and other systems the amount of time available for pathfinding systems to run decreases.

In a variety of problems we also have perfect information before we search. We have access to the graph offline in the case of pathfinding problems with perfect information. However, algorithms like A* do not make use of this information in most cases, failing to alter the heuristic based on the underlying graph. This produces algorithms that are more general and less specific to individual graphs, and as such are unable to utilize the specific graph information.

The two ways we use to speed up searches are,

1. Trade off optimal path length for less nodes expanded

2. Use a preparation step to gather more information about the search space

Weighted algorithms make use of the first way, in that they use weights to trade off between path length and the amount of nodes expanded. Changing the weight allows for multiple different trade offs to occur. The second way increases the heuristic information of the search, making in more informed than A*. This allows it to better expand nodes based on the information gathered during the preparation step.

## 3.2 Region Weighted A*

When examining maps in video games we notice that there are areas within the maps having certain attributes. These can be the size and shape of obstacles in the area, the room layouts of buildings, the size of open areas and many others. When the search space is translated into a graph these features can be as well. We want to exploit these features to make our search algorithms more informed so they can search faster. The problem is describing areas of the maps as substructures within the produced graph. We let a Region be such a substructure and define a Region as a set of the nodes in a graph.

Since different areas of the map may have different properties, we want to reflect this when searching within the regions they correspond to. We can do this by letting A* change its behaviour in different regions. We know from Weighted A* that we can alter the f-score so that the algorithm depends more on the heuristic. If we could change the calculation of the f-score so that it is dependent on the region of the node being evaluated, we might be able to better search within regions.

We introduce the Region Weighted A* algorithm to be an algorithm whose f-score is dependent on regions. We let $a(r)$, $b(r)$ and $c(r)$ be functions of a region that produce a weight and we change f(n) to be,

$$f(n) = a(r(n)) * g(n) + b(r(n)) * h(n) + c(r(n)) \tag{6}$$

where r(n) is a function that returns the region that n belongs to. Now this requires that r(n) exists for every node n. To facilitate this our algorithm ensures that $\forall n \in G, \exists R$ s.t. $n \in R$. We accomplish this by creating $\varrho$, a set of regions that is a partition of G.

It should be noted that our new f-score can collapse back to Weighted A*. Letting $a(R) = 1$, $b(R) = \epsilon$ and $c(R) = 0$, we have

$$f(n) = 1 * g(n) + \epsilon * h(n) + 0 \tag{7}$$

Similarly, since this is now the f-score of Weighted A* this can also collapse back into A* by letting $\epsilon = 1$. These collapses only occur if we apply these changes to all R. Only changing a single region might have the effect of A* within the region, but any node outside of the region will be evaluated not as expected, due to the change in relative magnitude of the f-scores.

These changes make it necessary to have the $a(r)$,$b(r)$ and $c(r)$ functions developed so that they can be called during search. We do this by means of a preparation step to be run offline on the search space before we do any searching. This step is responsible for partitioning the graph into regions and creating the $r(n)$ function in order to tie a node back to its region during search. Furthermore we require that we find the weight functions $a(r)$,$b(r)$ and $c(r)$ such that for each region we produce a triplet of weights(a,b,c) to be applied to the search while in that region.

There are several methods that can be used to store the $r(n)$ function. Since we need to call r(n) during every node evaluation it is important that its time complexity remain low. First we can add a pointer to the node structure itself, pointing to its region. This only requires an $O(1)$ look-up and is simple to implement. The downside is that the memory is $O(n)$ and we have to modify the original node structure. Secondly, we can use a HashMap to to relate the data. Similarly, this gives us $O(1)$ look-up time and $O(n)$ space, but without the need to modify the original structure. Lastly we can implement a space partitioning tree to better manage the memory. Using the tree we can use $O(|\varrho|)$ space and a $O(log(|\varrho|))$ time complexity. This is slower per node, but the memory scales with the number of regions employed and not with the number of nodes in the graph.

Without knowing how exactly the weights affect the search it can be difficult to choose weights that improve performance. Since for each region there are three weights and changing any weight can change the performance of the search, we have many combinations of weights to consider. If we let $v$ be the number of values we are considering for weights then we can choose

$$v^{3|\varrho|} \tag{8}$$

different configurations. This number is large enough that we can't simply search every case. We can however search the cases to find a better set of weights. Since the algorithm can collapse to Weighted A*, we can find an $\epsilon$ that produces a good result in Weighted A*. By letting all our regions use $\epsilon$ and collapse to Weighted A*, we guarantee that our weights produce an algorithm that is at least as good. Using this as a starting point, we can search for better weights, changing our weights only when there is an improvement. This can be accomplished by a hill climbing algorithm in order to search toward a optimal set of weights. To ensure that we find a good result we can also implement a random restart mechanism to avoid getting stuck in local maxima.

It is not only necessary that we find good weights, we also must find good regions. Randomly choosing $\varrho$ can easily result in a region set that is unrepresentative of the underlying areas in the map. Thus, it is important that we choose regions that reflect the features of the search we wish to exploit. One way to do this is manually by visual inspection. Features such as obstacle density and likeness to a maze on a video game map are easy to inspect when viewing the map. Thus a designer could manually choose regions based on these features with relative ease. However not all features are so visually obvious, and a designer might not wish to repeat this procedure on every game map. We can then turn to an algorithm to choose regions for us, such as clustering. Clustering algorithms can take a set of data like the nodes of our graph, and produce clusters of similar nodes. We can define a similarity function to relate to a feature and use this in the clustering process. Then we can take the produced clusters and convert them into regions. This way we can have an automated algorithm that runs in the preparation step and aside from the initial development of the similarity function, does not require the work of a designer.

**Algorithm 3:** Region Weighted A*

```
add start to open list;
while open isn't empty do
    min = find min in open;
    if min is goal then
        return success;
    end
    add min to closed list;
    foreach n in neighbours of min do
        if n is in closed list then
            continue;
        end
        region = get region containing n;
        if n was already opened then
            change f-score if better;
        else
            n's g-score = a(region) * (min's g-score + edge cost from min to n);
            n's h-score = b(region) * h(n,goal);
            n's f-score = n's g-score + n's h-score + c(region);
            n's parent = min;
            add n to the open list;
        end
    end
end
return failure;
```

## 3.3 Dead End Region Weighting

### 3.3.1 Single Dead End

We define a dead end node as one with a single edge. This type of node has the property that since it leads to no other node but itself, we only want to expand this node if it is the goal node. If we let $E$ be a dead end node and $\gamma$ be a goal node, then we have

$$Expand(E) \text{ if } E = \gamma \tag{9}$$

Now this is fairly simple and we could implement an A* which checks when expanding a node, whether or not its neighbours are the goal (before putting in on the open set). But let's try a solution that utilizes region weightings.

First, let $D$ be a region that contains all dead ends. Next we let $D'$ be the complement of $D$, such that it contains all nodes which are not dead ends. Since we want our algorithm to function like weighted A*(or just A*) in $D'$, we can set its weights to that of Weighted A*. For convenience, we will represent weights in the form of a vector comprised of $< a(n), b(n), d(n) >$ and for Weighted A* this looks like $< 1, w, 0 >$ for some weight w.

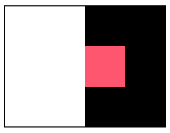
Figure 3: A dead end highlighted in red on a 4-connected grid graph

With region weighted A* our f-score function for E becomes

$$f(E) = a(D)g(E) + b(D)h(E) + d(D) \tag{10}$$

In the case where $E \neq G$ we never want to expand $E$. We can ensure this by making $f(E)$ have a greater f-score than any other node, this way it will always get chosen last for expansion. Let $w_\infty$ be a number that is greater than any f-score could possibly be on our graph, then when $E \neq G$, we let $f(E) \geq w_\infty$. We know that the heuristic estimate never overestimates the actual distance between any two nodes (assuming the heuristic is admissible), and this is true for $E$ and the node connecting to it. Thus if $h(E)$ is greater than the distance between $E$ and its connected node($C$), then $E \neq \gamma$. Given this, we have a condition we can use to help us obtain $f(E) > w_\infty$, when E is the goal, given by

$$\text{if } h(E, \gamma) > c(E, C) \text{ then } E \neq \gamma \tag{11}$$

We can rewrite the previous inequality in the form of

$$\text{if } h(E, \gamma – c(E, C) > 0 \text{ then } E \neq \gamma \tag{12}$$

Since we need $f(E)$ to be larger than $w_\infty$ in this case, we can multiply both sides by $w_\infty$. We can do this without worry of inverting the sign since, $w_\infty$ is always positive. Then we have

$$w_\infty(h(E, \gamma) – c(E, C)) > 0 \tag{13}$$

This can be used in our calculation of the f-score for $E$. Since this doesn't involve $g(E)$ at all we can let $a(E) = 1$, and update $f(E)$ to $g(E) + b(D)h(E) + d(D)$. Then we let

$$b(D)h(E) + d(D) = w_\infty(h(E, \gamma) – c(E, C)) \tag{14}$$

$$b(D)h(E) + d(D) = w_\infty * h(E, \gamma) - w_\infty * c(E, C) \tag{15}$$

From this we see that we can assign $b(D) = w_\infty$ and $d(D) = -w_\infty * c(E, C)$, giving us a final weighting of

$$D =< 1, w_\infty, -w_\infty * c(E, C) > \tag{16}$$

Now we can see how this region weighting functions, first let us assume $h(E, \gamma) > c(E, C)$. In this case, adding the last two terms produces a positive result that is a multiple of $w_\infty$ and thus makes $f(E) \geq w_\infty$, effectively making nodes belonging to D the last nodes expanded. Otherwise, $h(E, \gamma) \leq c(E, C)$, and the sum of the last two terms is either 0 or a negative multiple of $w_\infty$. Since all other f-scores are positive the negative $w_\infty$ will always be chosen first to be expanded.

In a non-monotone distance case, it is possible we expand this node falsely, if $\gamma$ is within the heuristic distance but not equal to $E$. When that sum is 0, the f-score becomes the g-score, and $E$ will be expanded as soon as all nodes with $f(n) < g(S, E)$ have been expanded. This ensures that $E$ is only ever expanded when $\gamma = E$, or all other nodes have been expanded. The latter case would only occur if no such path exists, since $f(E)$ has a greater f-score than all other nodes not in $D$ and if $\gamma$ was in $D$, it would have a negative f-score and be expanded before $E$.

### 3.3.2 Dead End Chain

We define a dead end chain as a series of nodes starting with a dead end node, followed by a sequence of nodes having 2 edges. The chain ends on the last node with 2 edges, and the non chain node connected to this, is part of the rest of the search space. Let E be a dead end chain with $E_0$ being the dead end, and $E_{R-1}$ being the last node in the chain, where $R$ is the length of the chain. We note that there is only one path between any 2 nodes in a chain, and that it is then the shortest path.

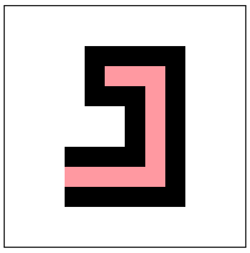

Figure 4: A dead end chain highlighted in red on a 4-connected grid graph

This means we can calculate the shortest distance between nodes on the chain, without the use of A*. We can calculate the distance from $E_0$ to $E_2$ from using the distance from $E_0$ to $E_1$ and $E_1$ to $E_2$ by $d(E_0, E_2) = d(E_0, E_1) + d(E_1, E_2)$. This is true since to get to $E_2$ from $E_0$ we must travel through $E_1$, by our definition of a chain. With this we can calculate the distance between $E_0$ and $E_3$ by $d(E_0, E_3) = d(E_0, E_2) + d(E_2, E_3)$. Generalizing this we have

$$d(E_0, E_k) = \sum_{i=0}^{k-1} c(i, i+1) \tag{17}$$

This gives us a formula to calculate the distance from the dead end to any other node in the chain. This is important because it gives us a limit on how far we can travel towards the dead end into the chain. We define a region for each node of a dead end chain (this can be simplified to use less regions). We also let the weighting for the region not containing dead ends be $< 1, w, 0 >$, to emulate weighted A* not in dead end chains.

Considering the last node in $E$ $(E_{R-1})$, we never want our algorithm to expand $E_{R-1}$ unless there is a chance the goal could be in $E$. If the goal isn't in $E$ there is no point in expanding any of it, since $E$ doesn't lead to any other node. We know from above that we can calculate the maximum distance one could travel into the dead end chain. Since the heuristic never overestimates the distance, if it is greater than the max distance in the chain, there is no chance that $\gamma$ is in $E$. Thus if $h(E_{R-1}, \gamma) > c(E0, E_{R-1}) + c(E_{R-1}, C)$ then $E$ is not in $\gamma$. Following similar steps to the dead end formulation above we can obtain,

$$b(D_{R-1})h(E_{R-1}) + d(D_{R-1}) = w_\infty * h(E_{R-1}, \gamma) - w_\infty * (c(E_0, E_{R-1}) + c(E_{R-1}, C)) \tag{18}$$

Putting this in terms of weights we have

$$D_{R-1} =< 1, w_\infty, -w_\infty * (c(E_0, E_{R-1}) + c(E_{R-1}, C)) > \tag{19}$$

We happen to know $c(E_0, E_{R-1})$ so we can write

$$D_{R-1} =< 1, w_\infty, -w_\infty * \left( \sum_{i=0}^{r-2} c(i, i+1) + c(E_{R-1}, C) \right) > \tag{20}$$

This acts much like the dead end case; if the heuristic is greater than the distance to the dead end, the goal is not in E, and the sum of the last two terms will be a positive multiple of $w_\infty$. Since $w_\infty$ is larger than any f-score, it will be expanded last, ensuring we never enter $E$ if the $\gamma$ can't possibly be in $E$. When the heuristic is less than or equal to the distance to $E_0$, then $E_{R-1}$ is either a negative multiple of $w$ or 0.

The 0 case is identical to the dead end case, but the negative case turns out a little different.

When f-score turns negative, it will be expanded before non-dead end chain nodes, but it is very possible that $\gamma$ is not in $E$ when this happens. Our weights only ensure we don't search down the chain if goal can't possibly be on it, but the more distance the chain covers, the higher the maximum heuristic distance allowed becomes. This in turn makes it become expanded more often, leading our search into the chain. Now we could just apply this weighting to all the regions associated with nodes in our chain, but it would cause our search to reach $E_0$ every time it expanded $E_{R-1}$. Instead we can change our weighting to

$$D_k =< 1, w_\infty, -w_\infty * \left( \sum_{i=0}^{k-1} c(i, i+1) + c(E_k, E_{k+1}) \right) > \tag{21}$$

This way every time we evaluate the f-score of a node in $E$, we tighten the requirement to continue down the chain. This reduces the chance we travel more than we have to in order to discover that we have taken the wrong path. In addition it works exactly like $D_{R-1}$. This lets our Region Weighted A* avoid unnecessary dead ends and paths that lead to dead ends, saving a lot of nodes in the process(depending on the search space).

It is known that A* expands the least nodes to find the shortest path for a given heuristic than any other algorithm. Upon reflection, we can rewrite the our f-score as $f(n) = g(n) + h'(n)$, where

$$h'(n) = w_\infty * h(E_{R-1}, \gamma) - w_\infty * (c(E_0, E_{R-1}) + c(E_{R-1}, C)) \tag{22}$$

That is to say $h'(n)$ is a piecewise heuristic that changes based on the local node. So really we can achieve better results than A* with a heuristic $h(n)$ because we are simply transforming the heuristic to one that better estimates the distance to the goal. We just do so in a fashion that uses the original heuristic and regions to dictate the pieces of the function.

## 3.4 Maximum Allowed Heuristic

### 3.4.1 Definition

We expand upon the idea of dead ends to provide a more general solution that affects the entirety of the search space. The core idea behind the dead ends, is that we don't search a path that is shorter than the heuristic distance to the goal. This is because the heuristic never overestimates the distance, so the absolute minimum amount we can travel to reach the goal is the heuristic distance. If a path is shorter than this, than there is no way it can be a path to the goal.

First we show that if $A$ and $B$ lie on the shortest path between $S$ and $\gamma$, then the shortest path between $A$ and $B$ is on the shortest path from $S$ to $\gamma$. If there was a shorter path between A and $B$ then there exists a path composed of $S$ to $A$, $A$ to $B$ and $B$ to $\gamma$ that uses this new shorter path. Since the path from $S$ to $A$ and $B$ to $\gamma$ are invariant to the path change, then the combined path length will be shorter than the original path from $S$ to $\gamma$. This can't be because the original $S$ to $\gamma$ path is the shortest, thus no shorter path between $A$ and $B$ exists.

We use this concept by letting $B = \gamma$, and examining the path as $S$ to $A$ and $A$ to $\gamma$. Then if $S$ to $\gamma$ is the shortest path, and $A$ lies on $S$ to $\gamma$, then $A$ to $\gamma$ on $S$ to $\gamma$ must be the shortest path. Thus any shortest path that goes through A must contain a shortest path starting with $A$. Any path follows edges between nodes and thus this path must start with an edge from $A$. If we take the set of shortest paths starting with $A$ and ending with all other nodes in the graph, we can partition this set into collections corresponding to which edge they leave $A$ from. If there are multiple edges that lead to the shortest path between A and some other node, we randomly choose a partitioned set.

Next we can transform these partitions from sets of paths to sets of lengths, each being the path length of its path. Then each collection is a set of all possible distances one can travel to arrive at a goal, while taking the shortest path. We note that for dead ends, we use the maximum amount of distance we could travel in the dead end to help exclude them. If we were expanding $A$ and evaluating the node across an edge and we knew that we could only travel so far through this edge, we could apply a similar behaviour. Thus we find the furthest we could travel along an edge, and use that as a cutoff to compare the heuristic distance to.

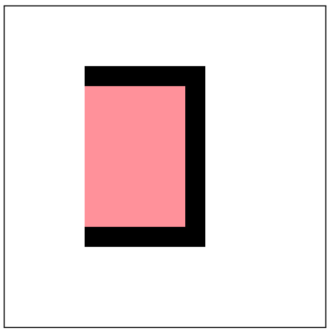
Figure 5: A concave region like the one highlighted can be avoided in some searches when using a Maximum Allowed Heuristic

Given that we have a set of the shortest distances (being the only distances we would travel on shortest paths) by choosing the maximum of these we can find the furthest we can travel while staying a shortest path. So if we were to expand $A$, and we knew the heuristic distance to the goal was greater than this value, then no path going through that edge leads to the shortest path to the goal, so we need not expand that node. By saving this maximum allowed heuristic value on the edge, we can use it when searching to potentially avoid searching down a bad path.

### 3.4.2 MAH Search

This algorithm uses a simple preparation step to give edges an extra weight that contains the maximum allowed heuristic (MAH) value. We simply do an A* search from every node to every other node in the graph. After we find each path we let the MAH be

$$MAH(edge) = Max(MAH(edge), \text{length of the new path})$$

This populates every edge in the graph with a MAH value that we can use in the search portion of the algorithm.

**Algorithm 4:** Naive Maximum Allowed Heuristic Preparation

```
foreach node A in all nodes in graph do
    foreach node B in all nodes in graph do
        if A is B then
            continue;
        end
        find shortest path between A and B;
        if path is found then
            length = length of the found path;
            followedEdge = the edge from A to the rest of the path;
            followedEdge's MAH = maximum of followedEdge's MAH and length;
        end
    end
end
```

The search portion is nearly identical to A*. The only thing we change is that before we evaluate a node while expanding some node A, we do a check. The check compares the heuristic value of A to the goal, to the MAH of the edge to the node we are evaluating.

$$\text{If}(h(A, \gamma) > MAH(\text{edge to neighbour of A})) \text{ then don't evaluate the neighbour}$$

This saves us from exploring paths through these edges and thus makes A* expand less nodes than usual. Since the MAH ensures that the goal can't be down the path, this also ensures that MAH search maintains the shortest path.

**Algorithm 5:** Maximum Allowed Heurisitic Search

```
add start to open list;
while open isn't empty do
    min = find min in open;
    if min is goal then
        return success;
    end
    add min to closed list;
    foreach n in neighbours of min do
        if min's h-score $>$ maximum allowed heuristic for edge min to n then
            continue;
        end
        if n is in closed list then
            continue;
        end
        if n was already opened then
            change f-score if better;
        else
            n's g-score = min's g-score + edge cost from min to n;
            n's h-score = h(n,goal);
            n's f-score = n's g-score + n's h-score;
            n's parent = min;
            add n to the open list;
        end
    end
end
return failure;
```

### 3.4.3 Complexity

The preparation complexity is really the problem with this approach. Since we have to check every node versus every other node the time complexity of the preparation step is $O(E * n^2)$, where $E$ is the number of edges in the graph. This is only the case in the naive approach shown in algorithm 4 though. Using an approach that utilizes the Floyd-Warshall algorithm it should be possible to reduce the time complexity to $O(n^3)$. The added space complexity is 1 value per edge so $O(E)$.

The time complexity during the search step is the same as A*. We already know the heuristic distance from the node we are expanding to the goal and the MAH is an $O(1)$ retrieval. It is expected it to be faster on average though, due to the nodes we avoided expanding.

## 3.5 Transit Regions

What we really want to accomplish is to be able to exploit the substructures found within a search space in order to speed up search. While Maximum Allowed Heuristic Search avoids dead ends, it still exhaustively searches through regions that a more simple search algorithm could search through fast. Take for instance an undirected grid graph where every grid cell is a node. Then we connect nodes that are horizontally and vertically adjacent with a weighted edge of cost 1.

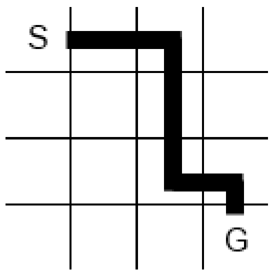

Figure 6: A Simple Search

In this graph searching is trivial with a Manhattan heuristic since we can just expand nodes with the minimum h-score. This search expands $p$ nodes, where p is the length of the path to be found. What transit region search does is make use of regions where we can search faster than A* and use A* in regions we can't.

### 3.5.1 Region Definitions

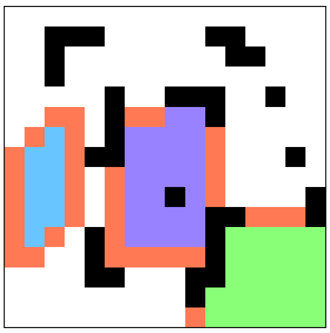

Figure 7: An example of 3 well-formed regions in a 4 connected grid graph.The regions are identified by the colours blue, purple and green with their boundaries highlighted in orange

A region is a set of nodes. It turns out a more rigid set of restrictions provides us with more properties to exploit. Thus we define a well-formed region (R) as a region having the following properties.

1. $\forall a, b \in R, \exists P_{a,b}$ s.t. $\forall x \in P_{a,b}$ , $x \in R$
2. $\forall b \in R, \exists y \in adjacent(b) \wedge y \notin R \implies b$ is a mandatory boundary node
3. $\forall b \in R, |adjacent(b)| < \kappa \implies b$ is a optional boundary node
4. A boundary node is either optional or mandatory.
5. A region is composed of internal nodes and boundary nodes.
6. Let $B \subseteq R \wedge \forall b \in B$ b is a boundary node $\wedge \forall \alpha, \beta \in B$, $\exists P_{\alpha,\beta}$ s.t. $\forall x \in P_{\alpha,\beta}$ , $x \in B$ then B is a boundary.

Here $\kappa$ is the maximum amount of nodes any node may have adjacent to it.

These restrictions exclude many of the previous regions; including all unconnected regions. An immediate consequence of well-formed regions is that when performing A*, if the start and goal reside in the same region, we have no need to expand any node outside of region. Since the shortest path is within the well-formed region, no nodes on the path are outside of the region and need not be expanded.

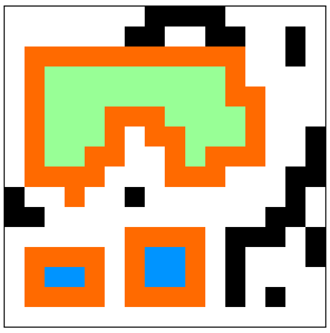

Figure 8: Examples of regions that are not well-formed. There is not a shortest path between some nodes in the green region, and the blue region is disconnected disallowing some paths altogether

Our approach depends highly upon 2 things. First how the heuristic function performs within the region. This is a measure of strictly local performance, and we will try to isolate regions with better local performance. Secondly, we depend upon the boundaries of the region. Given different types of boundaries we can make assumptions and exploit interesting properties. We identify 4 different types of heuristic performance within a well-formed region, being

1. Heuristically Perfect
   (a) $\forall a, b \in R$ let h(a,b) = h*(a,b)
   (b) min h* always picks the right node to expand

    (c) h*(a,b) = c(a,b), the heuristic distance returned is the actual distance

2. Heuristically Great
    (a) $\forall a, b \in R$ h(n,g) = h*(n,g)
    (b) min h* always picks the right node to expand
    (c) h*(a,b) is not equal to c(a,b), it can give the wrong distance
3. Normal – no special properties
4. Misleading - heuristic more than often leads you down the wrong path

The algorithms we are proposing make use of Heuristically Perfect regions. The combination of the heuristic always choosing the correct node to expand, tied with being able to accurately calculate g-score by knowing h-score, gives us some powerful properties to work with. Note that there are algorithms that make use of the other types of heuristic performance, and that they can help improve over A*. We only focus on Heuristically Perfect Regions, as they provide us with the most information that we can use to our advantage.

As for boundaries we identify them by 2 features. The first is the number of boundaries on the region, which we deal with as 1, 2 or N. These correspond to paths that travel through the region as no choice, 1 choice and many choices, given that a path enters through a boundary. The second is the boundary path type, which we divide into point boundaries or multi-point boundaries. A point boundary is comprised of a single node, and multi-point boundary has more than one node.

### 3.5.2 Heuristically Perfect Regions

We will now we go through the combinations of heuristically perfect regions and provide algorithms for searching them in a faster manner than A*. The first of these has 1 point boundary. This region is essentially a dead end, and if we knew the furthest we could travel into it, we could avoid the region entirely. Thus we can use the maximum heuristic values of the edges connecting the boundary to the rest of region. If the heuristic on these edges is greater than our maximum heuristic values, then we can't travel as far as we need to in the region, so we don't need to expand those nodes. This saves us from expanding the entire region (except the boundary) if the all the edges' values are less than the heuristic.

If this is not the case however, we then need to search the region. First, since we have $h^*(n)$, we will always expand the correct nodes based on $h(n)$. So we don't need to do A*, we can just expand the node with the lowest h-score, with no need to backtrack. If the goal is in the region we will find it on this path. Also since $h^*(n) = g(n)$, at the boundary edge we enter the region

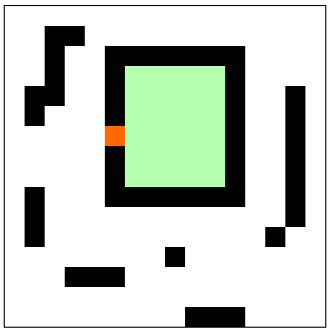

Figure 9: A region with a single boundary point

we know the distance to the goal. If we travel that distance down the path we should find the goal. If we don't then we know the goal isn't in the region, and we can skip the rest of the nodes.

This means that we at most expand the number of nodes required to travel the MAH distance. We also need only to store space for those MAH values for the boundary node, giving an extra space complexity of $O(\text{max edges of a node})$.

Next we discuss the Heuristically Perfect region with 2 point boundaries. Unlike the last region this one can have a path that goes through the region, having a start and goal node outside of the region. We can still make use of the maximum allowed heuristic to exclude nodes within the region, but the max value would be a lot higher and very few nodes would be excluded. Thus we use a local MAH; that is, we find the maximum distance one could travel within the region through an edge. Similar to the global MAH previously employed, if our heuristic value is greater than our MAH then the goal is further than any path in the region and much lie outside of it. If this is the case, we can exclude the region, but the path could still be through the other boundary, so we must expand it.

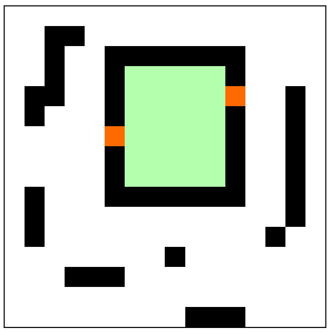

Figure 10: A region with two boundary points

The problem with expanding nodes not directly connected to the node we are expanding is that we need to calculate its f-score. The heuristic distance to the goal isn't a problem, but we don't know the actual distance from our expanding node to the other. However, remember our definition of $h*(n)$ for Heuristically Perfect regions includes that $h*(n) = g(n)$ over the region. Thus if we let $J_0$ be the boundary we are expanding and $J_1$ be the other boundary then,

$$\begin{aligned} f(J_1) &= g(J_1, S) + h(J_1, \gamma) \\ &= g(J_0, S) + g(J_0, J_1) + h(J_1, \gamma) \\ &= g(J_0, S) + h(J_0, J_1) + h(J_1, \gamma) \end{aligned} \tag{23}$$

This allows us to expand the other boundary from our boundary and know the exact f-score of the node. This ensures an optimal path and allows us to skip over the other nodes in the region, since we don't need to travel them to discover the g-score for $J_1$. In the case where the local MAH is greater than or equal to our heuristic, we have to search the region. Like the previous case we can do this by doing a pure heuristic search, and only expand as far as the heuristic distance.

Our node expansion is the same as the previous region, plus the expansion of the other boundary. In terms of extra space needed its $2 * O(\text{max edges of a node})$, as we need to store a local MAH for every boundary edge that leads into the region. We also need a edge from $J_0$ to $J_1$ to be added to the graph so we know to expand it (this is $O(1)$). We refer to this edge as a bridge edge, as it is a bridge over the region. This is still a very small amount of space.

The third region we examine is the Heuristically Perfect region with N point boundaries. This is similar to the previous case, except that we have more than one possible path exit when a path travels through the region. If we find that we can skip the region, we have to open all other exit nodes. This requires that we store a MAH value for every inner edge on a boundary point. It also means we need to add a bridge edge from every boundary point to every other boundary point. This increases the memory footprint to

$$O(|\Upsilon| * \text{max edges of a node}) + O(|\Upsilon| * (|\Upsilon| - 1)) \tag{24}$$

where $\Upsilon$ is the set of boundaries on the region. Note we leave the possibility that the graph is directed, so it is needed that the worst case account for both adding a bridge between some $J_0$ and $J_1$ and between $J_1$ and $J_0$.

Now we consider regions where the boundary is not a single point but multiple points. Let us examine regions where the boundary is a singular path. We can view this as a region with N point boundaries combined with a 1 point boundary region. Although there are multiple boundary points, there is no path that enters on a boundary, travels through the interior of the region and exits another boundary point. This is the case because the boundary is known to be a shortest path between any points in that path. So if a path went into the region and left

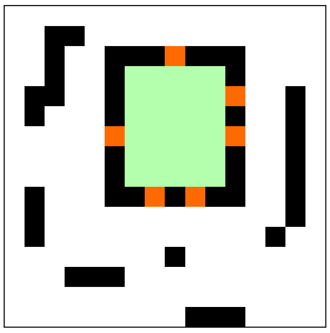
Figure 11: A region with N boundary points

through a boundary point on the singular boundary, then there exists a path between the entry and the exit that lies on the boundary which is shorter. Thus the path cannot be the shortest path. This allows us to treat the boundary path like the single point boundary and avoid the entire region based on the MAH of the boundary points, since no shortest path travels through the region.

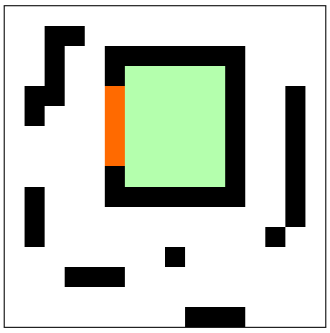
Figure 12: A region with a single boundary

There is more than one boundary point though and the extra memory is going reflect this. We need to store a MAH value for every point on the boundary, so the extra memory amounts to $O(|\Upsilon| * \text{max edges of a node})$. We do not need to create any bridge edges since we don't travel through the region; if the path we are finding lies on the boundary, we just search with normal A*. If we wanted to create bridges to speed up searches on the boundary it would be approximately $O(|\Upsilon| * (|\Upsilon| - 3))$ extra memory. This is a trade-off, but the extra memory cost seems unwarranted for a small number of nodes not expanded.

We move onto a region that has a boundary composed of two paths. In this case we do have to worry about paths that can travel through the region. This is like the N-Point boundaries but we know a path that travels through the region must connect one boundary to another. Thus if we can skip over a region based on the MAH, we need to open all nodes on the other boundary. Then the number of edges we need to add amounts to $|B_1| * |B_2|$, where $B_1$ is the entry boundary and $B_2$ is the exit boundary. Given that we also need to store the MAH values our extra memory becomes,

$$O((|B_1| + |B_2|) * \text{max edges of a node} + |B_1| * |B_2|) \tag{25}$$

We note that the number of nodes evaluated during expansion sharply increases on a boundary node, but we hope to choose regions where we save even more node expansions by entirely skipping the interior of the region.

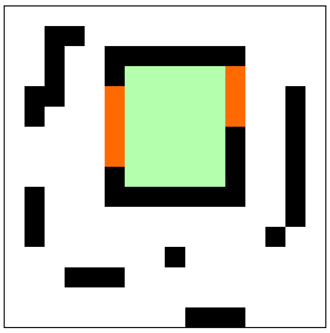
Figure 13: A region with two boundaries

The final case we look at is where the boundary is composed of N paths. This is very similar to the previous case. If a path travels through the region, it could exit through any boundary that isn't the one it entered. Thus we need to add bridge nodes from every node to every other node not on the first nodes boundary. Thus we need to add $\sum_i |B_i| * \sum_j |B_j|$ bridge edges. This increases our extra memory to

$$O(\sum_i |B_i| * \text{max edges of a node} + \sum_i |B_i| * \sum_j |B_j|) \tag{26}$$

We will be most concerned with this last case, as it is the most general and as such can be applied to our search space more readily.

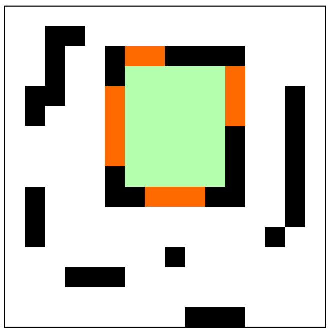
Figure 14: A region with N boundaries

We can prove that we can partition any graph into a set of N-Path Heuristically Perfect regions. First we note that an N-path heuristically perfect region can be a single node. It is a single path boundary, consisting of a single point, with 0 interior nodes in the region. It is the shortest path from itself to itself, so it meets the criteria of boundaries needing to be a shortest path. Since a consistent heuristic never overestimates the path length to the goal, the heuristic must return 0. If we make every node in the graph be its own region, those regions are N-path heuristically perfect and it makes a partition of the graph.

This example is pretty much just to prove that we can partition any graph, it would be useless for our purposes as the regions contain no interior nodes to skip. Ideally we want to ensure that we have enough nodes to skip in order to justify the extra memory we are using. Next we look into more requirements we could place onto these regions in order to reduce our memory footprint.

### 3.5.3 Optimality

It is often stated that A* is an optimal algorithm, in that for some heuristic $h(n)$ no other shortest path algorithm will expand less nodes than A*. However the fact is this isn't always the case. First this only applies to algorithms that are as equally informed as A*. This means that both algorithms are given the same heuristic information.

With our algorithms we use a preparation step to define regions where $h*(n) = h(n)$, and we use this information to change how we expand nodes. A* on the other hand does not have access to this information. Therefore it is possible for our algorithm to expand less nodes than A* while maintaining the shortest path because it utilizes extra information that A* does not.

Another interesting fact is that A* is not always optimal compared to another algorithm even if they are equally informed. Decther and Pearl [DP85] showed that there are different types of optimality and that A*'s type of optimality differs based on the class of the other algorithm as well as the domain of the problem instance. In the class of algorithms that are admissible if $h(n) \leq h*(n)$ and the common problem domain of consistent heuristics, A* is only 1-optimal.

This means there exists a tie breaking rule that will expand a subset of nodes of any other admissible algorithm. However because it is not 0-optimal, not all tie breaking rules will expand a subset. This reflects the creation of tie breaking rules that perform better on certain problems, as there is known to be at least one that will let A* outperform all other equally informed algorithms.

### 3.5.4 Transit Regions

We define a Transit Region as an N-path Heuristically Perfect region that for every boundary node $J$ there exists a boundary node $K$ on every other boundary such that

$$g(J, K_i) \geq g(J, K) + g(K, K_i) \tag{27}$$

We call $K$ a transit node of $J$. It should be noted that this is the reverse triangle inequality.

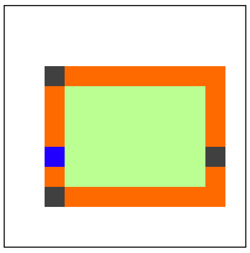

Figure 15: A Transit region showing a boundary node in blue, and its transit nodes on the other boundaries in gray

This makes it possible to greatly reduce the number of bridge edges we need for each boundary node. Since the shortest path distance from J to some $K_i$ is always greater than or equal to the distance from $J$ to $K$ and then $K$ to $K_i$, we can always take the latter path and be assured it's the shortest. Then we only need a bridge from $J$ to $K$, and we can simply search the shortest path from $K$ to $K_i$, which is along K's boundary path.

The memory requirement for an algorithm that uses Transit regions is also reduced. Since we only need a single bridge from a boundary node to each other boundary, the number of bridges becomes $(|\Upsilon| - 1) * \sum_i |B_i|$, where $|\Upsilon|$ is the number of boundaries. The extra memory we use is

$$O(|\Upsilon| * \text{max edges of a node} + (\sum_i |B_i| - 1) * |\Upsilon|) \tag{28}$$

which is far less than with the Heuristically Perfect regions.

While this uses less memory we also want to ensure that it evaluates a low number of nodes. The number of nodes evaluated this way is $(\sum_i |B_i| - 1)$ + the number of nodes evaluated from $K$ to $K_i$. This is opposed to evaluating the $|\Upsilon|$–$|B_i|$ nodes, which is usually larger. It is guaranteed larger if we search straight from $K$ to $K_i$, this adds $O(|\Upsilon|)$ extra memory though. It should usually evaluate fewer nodes this way, and save on memory, making this an acceptable improvement from the one used in Heuristically Perfect Regions.

### 3.5.5 Manhattan Distance

While all the previous algorithms will work on any non-negative weighted graph, we depart to work on graphs where the heuristic used is a Manhattan one. This is fairly common in a lot of search problems, and is often used in games. We seek to find a condition that makes a Heuristically Perfect Region be a Transit Region, with the Manhattan Heuristic. We find these for a general n dimensional Manhattan heuristic,

$$h(x,y) = \sum_i (|x_i – y_i|) \tag{29}$$

We know that on a transit region, our shortest path distance must satisfy

$$g(J, K_i) >= g(J, K) + g(K, K_i) \tag{30}$$

And on an heuristically perfect region $h* = h = g$ so we have,

$$\sum_j (|J_j – K_{ij}|) >= \sum_j (|J_j – K_j|) + \sum_j (|K_j – K_{ij}|) \tag{31}$$

We search for a coordinate wise solution, one that holds this condition for each j

$$|J_j – K_{ij}| >= |J_j – K_j| + |K_j – K_{ij}| \tag{32}$$

There is however a triangle inequality for real numbers and absolute values that states

$$|a – c| <= |a - b| + |b – c| \tag{33}$$

Thus we are left with a singular equality that must hold

$$|J_j – K_{ij}| = |J_j – K_j| + |K_j – K_{ij}| \tag{34}$$

We note that $K_j$ is chosen for $J_j$ and thus can be any node on the boundary $K$. Let us divide the problem into 2 cases.

1. $J_j$ falls outside of the interval from $K_j$ to $K_{ij}$
   (a) If $J_j$ is closer to $K_j$, then $J_j$ and $K_{ij}$ form the end points of a line, and $J_j – K_j + K_j – K_{ij} = J_j – K_{ij}$.
   (b) If $J_j$ is closer to $K_{ij}$, then this does not hold, but if we make the closest $K_{ij}$ be $K_j$, then $K_{ij} = K_j$ or this situation never occurs. Then this case holds.

2. $J_j$ falls inside the interval from $K_j$ to $K_{ij}$ a. This doesn't hold unless $J_j = K_j$, otherwise the line formed has $J$ as point on the interior of the line, and the distance some of the end points + the end point and $J$ is always greater than the sum of $J$ and the other end point. When they are equal though, $J$ becomes an endpoint and the $J_j$–$K_j$ term becomes 0.

Thus we have conditions for our equation to hold, and when the equation holds, the heuristically perfect region is a transit region.

If we use a Manhattan heuristic in a heuristically perfect region, then the region is a transit region if for every boundary node $J$, the projection of $J$ onto the ith axis either

1. Lies outside of the interval created by projecting every other boundary $K$ onto the ith axis.

2. Lies exactly on the projection of some node k on every other boundary $K$ onto the ith axis.

This gives us a test to determine whether we can use our Transit Region algorithm on a region or not.

### 3.5.6 Axis Aligned Regions

Getting even more specific we can make use of the previous proof in a case that occurs frequently. Let us prove that on a grid space, using a Manhattan heuristic any heuristically perfect axis aligned polygon is a Perfect Transit Region.

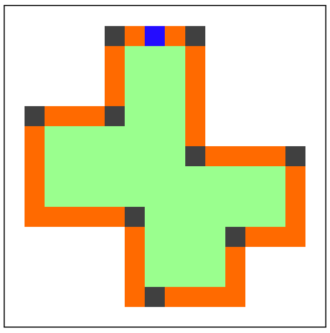

Figure 16: An Axis Aligned Transit Region on a 4-connected grid graph with a boundary node and its transit nodes highlighted

Let $J$ be a node on some boundary of the polygon and let $B$ be another boundary on on the polygon. If we project $J$ onto $B$ by the x coordinate we must satisfy one of two cases. If the projected x is outside of the interval of B, then the x-coordinate is free for the transit point on B. Otherwise, if the projected x is on the interval than there needs to be a node at that location. This is always true as a line boundary on a grid space occupies every grid cell between its two endpoints. Therefore the grid cell that is projected exists on the boundary.

The y coordinate then must project to the same node or not exist in the interval. Since the boundaries are axis aligned only one of the coordinates will be within the interval, or the boundaries overlap at node $J$. This ensures that the conditions are met for a transit region. Therefore choosing our regions to be heuristically perfect axis aligned regions lets us also use them as transit regions.

### 3.5.7 Implementation

Our implementation uses transit regions and it consists of a preparation step to partition the search space into convex polygons(we opt to use rectangles for ease of use) and populate the local MAH values of boundary nodes. Then online we search making use of the properties of Perfect Transit Regions to avoid searching the interior of regions.

The preparation step uses a quad tree [?] based system to partition the graph into rectangular regions. We set the root of the quad tree to represent the entire search space. Then if we find an obstacle in the quad tree we split it into 4 children. We repeat this until two sets of tree remain, those with only obstacles, and those without. We take the former to be our set of regions. Then we do a combining phase where we join adjacent regions, to clean up regions that were split poorly due to the where the quad tree chose to split.

**Algorithm 6:** Quad Tree Region Partitioning

```
currentTrees = a empty list of QuadTrees;
add a tree covering the entire search space to currentTrees;
while currentTrees isn't empty do
    nextTrees = a empty list of QuadTrees;
    foreach QuadTree tree in currentTrees do
        if tree contains no obstacles then
            add tree to returnList;
        end
        else
            split tree into 4 QuadTrees;
            add these to nextTrees;
        end
    end
    currentTrees = nextTrees;
end
return returnList;
```

Then we go through the boundary nodes of the regions and add the bridge edges. We

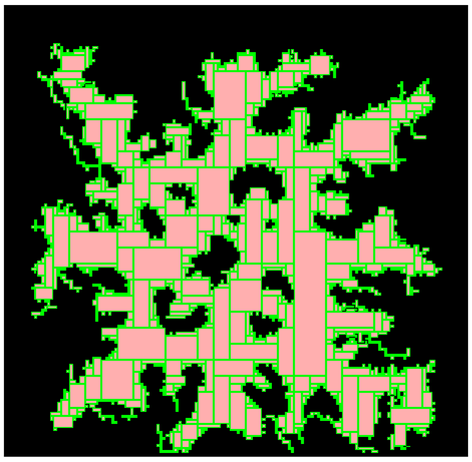

Figure 17: The Warcraft 3 ”Battleground” map partitioned using the quad tree partitioning algorithm

only need to add a single bridge edge per node per other boundary, since the regions are Transit Regions. We also don’t add bridges between adjacent edges since in a rectangle the shortest path between adjacent sides can be found by following the boundaries using a Manhattan heuristic. This means we are only adding 1 bridge per boundary node. After this we calculate the heuristic distance of every path coming from an edge of a boundary node to determine the MAH value. Normally one would have to do an A* search to obtain the path length, but since h(n) = h*(n) = g(n), we can calculate it without having to do any searching.

**Algorithm 7:** Transit Region Search Preparation

```
regions = QuadTreeRegionPartitioning();
foreach region in regions do
    bounds = a list of the region’s bounds;
    foreach bound in bounds do
        foreach boundNode in bound do
            boundNode’s MAH = maximum distance you can travel in the region’s interior
            from boundNode;
        end
        foreach other bound in bounds do
            add a transit edge from boundNode to its transit node on other bound;
        end
    end
end
```

The online searching step acts somewhat like A*. If the node we are expanding isn’t a boundary node we just evaluate all neighbour nodes just like A*. We change our algorithm though when evaluating boundary nodes. If a node is a boundary node, we first evaluate its bridge nodes. Next we check if the heuristic value is less than the MAH value. If it is, then we need to evaluate every neighbour, if it isn't, we don't need to evaluate any neighbour that is in the interior of the region. We still need to evaluate other neighbours that are boundary nodes though. We add nodes that are interior nodes that we skipped to an ignored list, so that we can keep track of nodes we have expanded and then closed versus nodes that we never even considered.

**Algorithm 8:** Transit Region Search

```
add start to open list;
while open isn't empty do
    min = find min in open;
    if min is goal then
        return success;
    end
    add min to closed list;
    if min is a boundary node then
        foreach boundaryNode connected to min by a transit edge do
            EvaluateNode(boundaryNode);
        end
        if min's h-score $\leq$ min's MAH value then
            call Evaluate Node on every neighbour of min that isn't on the skippedList;
        end
        else
            call Evaluate Node on neighbour's of min that are boundary nodes add the
            nodes that aren't to the ignoredList;
        end
    end
    else
        call Evaluate Node on every neighbour of min, including those on the ignoredList;
    end
end
return failure;
```

**Algorithm 9:** Evaluate Node

```
if n is in closed list then
    return;
end
if n was already opened then
    change f-score if better;
else
    n's g-score = min's g-score + edge cost from min to n;
    n's h-score = h(n,goal);
    n's f-score = n's g-score + n's h-score;
    n's parent = min;
    add n to the open list;
end
```

There are some differences in the program output when compared to A*. When Transit Search returns the path it can be reconstructed like A* from following each node's parent. Unlike A* however, the parents can be waypoints (boundary nodes on regions) and to get the actual path you still need to reconstruct the path in between them. This is fast since $h^*(n) = g(n)$ in a Transit Region and the resulting search can be done in $O(p)$ time, where p is the length of the path between waypoints. It should also be noted that we need not rebuild the path at all if

1. All we need is the path length, since the goal's g-score gives this.

2. The underlying system's agents can travel between waypoints.

The second case is common in video games as the pathfinding is employed to find a path that a computer controlled agent can follow. The more points in the path to follow just increases the work of agent moving. Since the agent could travel through an open area, like a transit region, using just the waypoints can be valid and improve the efficiency of the agent's movement.

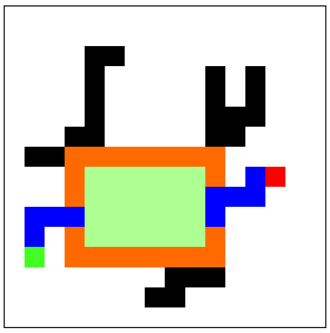

Figure 18: Transit Search returns a path that can have waypoints and thus not include all the nodes on the path found. The path between waypoints can be reconstructed after in O(p) time if needed.

Another possible difference from A* is the quality of paths returned. Transit Search will tend to return paths that axis aligned. Since we use axis aligned rectangles for regions, the boundaries are vertical and horizontal lines. Many of the waypoints found on the path will be on the boundary nodes and transit search can expand along the boundaries, producing axis aligned lines. We view path quality as not one of the algorithm's goals but it should be noted that the paths returned have a tendency to be rectilinear. If the path quality wanted is more diagonal, solutions like string pulling can always be run as a post process to improve the quality, and this can be done in $O(p)$ time.

## 3.6 Weighted Transit Search

A* can be changed to Weighted A* by adding a weight to improve performance but lose some path optimality. Similarly we can apply this weighted approach to Transit search in order trade optimality for speed. The change is near identical to that applied to A* in that we simply transform $h(n)$ to $h'(n)$ where

$$h'(n) = \epsilon * h(n) \tag{35}$$

One thing to note is that we still need $h(n)$ when checking against the maximum allowed heuristic value. If we were to use $h'(n)$ we could end up skipping over the region with the goal in it because the algorithm would think the minimum path length to the goal is larger than it really is.

# 4 Experimental Setup

## 4.1 Search Spaces

We choose to do our experiments on unit cost grid graphs. These are graphs in which the nodes are cells on a grid connected to adjacent cells. We connect nodes if they are adjacent, and for our purposes adjacency is defined as the cells above, below, right and left of a node. Each of the edges has monotone weight of 1. This type of graph is used often in video games due to its simplicity and ability to be a discretization of continuous search spaces.

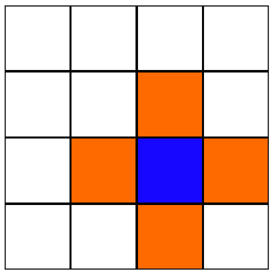

Figure 19: A 4-connected unit cost grid graph

We picked search spaces from video games in industry namely, Warcraft 3 and Baldur's Gate. These were obtained from Nathan Sturtevant's Pathfinding benchmarks freely available online [Stu12]. The maps used are on a grid graph and connect their nodes with at most 4 edges as previously described. These maps are used in practice and as such provide a good testing environment for our algorithms. Our implementation of Transit Search exploits the sub-structure of rectangles, and the maps used have a variety of larger areas and enclosed regions that can be partitioned into rectangles.

Figure 20: A map from Baldur's Gate

## 4.2 Assumptions

1. The search algorithms have perfect information. That is, the graph is known before the search and more information is not discovered during the search. This is in fact needed for Transit Search given its dependence on a preparation step.

2. The graph is static. Some problem instances are dynamic where the configuration of the graph changes. We do not test on any of these instances. The preparation step creates regions based on the input graph, any changes to the graph could invalidate the transit regions.

3. All searches performed are between a single start node and single goal node. While we could have a goal containing multiple nodes and a heuristic that accounts for this; we use the simple case. This is common in pathfinding research.

4. The graphs are connected. This means there is a path from any node to any other, or more concise
$$\forall a, b \in G \; \exists P_{a,b} \tag{36}$$

5. The search spaces tested are representative of those used in a practical setting. We choose maps from industry for this very purpose. We make the assumption that these maps are a good representation of search spaces in other games as well.

6. All searches use the same tie breaking rule. In our implementations of the algorithms we simply use the minimum element of a PriorityQueue in Java SE 8. We make no attempt to sort nodes on the fridge based on any explicit tie breaking rule.

## 4.3 Algorithms

In our experiments we test and compare 4 different algorithms.

1. A*

2. Weighted A*

3. Transit Search

4. Weighted Transit Search

We use A* as a baseline for comparison as it is the most used pathfinding algorithm. It finds least cost paths so it gives us a way to measure how sub-optimal the path lengths returned by other algorithms are. It is also on average the slowest of the tested algorithms, expanding the most nodes. This lets us also compare how much faster the other algorithms are compared to A*.

With regards to the two Weighted algorithms, we choose two different weights to reflect alternate uses of weights. We let $\epsilon = 1.1$ to represent the cautious case as the path length returned is bounded by 1.1 times the cost of the shortest path(1 in the case of our graphs). In the other case we want to more reckless, not caring for how much longer the paths returned are, we just want a faster search time. The weight we choose here is $\epsilon = 100$.

Lastly we search using Transit Search. This maintains the shortest path like A* but skips areas within the contour that A* must search. This should in most cases decrease the number of nodes the algorithm needs to expand and we expect to see this in the results. In total we search each path 6 times, twice for the weighted algorithms and once each for the others.

In all cases we use the Manhattan Heuristic defined as

$$h(a, b) = |a_x - b_x| + |a_y - b_y| \tag{37}$$

### 4.4 Paths Tested

The paths tested are chose at random from the graphs. We randomly choose a start node and a goal node, then find the path between them. We know the path exists from our assumptions, so we don't need to worry about the path not being found. We reject a path outright if the chosen start node and goal node are the same. This is done as the path length is 0 and all are algorithms will exit on their first iteration with this result. No useful information can be gained from that path, so we skip any of these paths.

Each map we test we choose 100 paths. Each of the six algorithms is run over every path so that we can compare results on each data point. With the Warcraft 3 maps this gives us 1200 such data points and with the Baldur's Gate maps we have 7400 more. This gives us a combined total of 8600 data points we can use for analysis. We use all of the data points in our analysis and leave none out, this includes results we may view as outliers.

# 5 Analysis of Results

## 5.1 Nodes Expanded Analysis

We compared the average nodes expanded by each of the algorithms tested. On maps from both games the results followed a similar pattern. A* expanded the most nodes on average, expanding 7439 nodes and 7967 nodes respectively. This was an expected result as we had planned for A* to be the baseline algorithm. All the other algorithms have some mechanism for outperforming

A*, so we would expect them to do so.

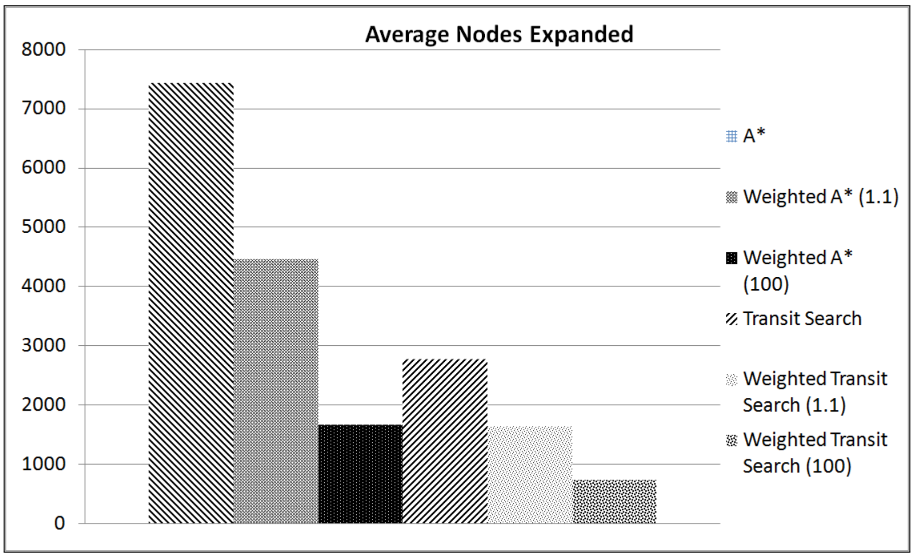

Figure 21: Average Nodes Expanded on Warcraft 3 maps

Weighted A* using a weight of $\epsilon = 1.1$ was the next most node expanding algorithm. It expanded 4460 nodes and 5589 nodes respectively. This amounts to a speed increases of 1.67 and 1.43 times that of A*. Weighted A* is known to expand less nodes than A* and the results confirm this, even with a weight only 10% higher than that of A*'s implicit weight of 1. When we use a weight of $\epsilon = 100$, the number of nodes expanded shrinks to 1658 and and 2213 nodes. This is even more of an advantage, increasing speeds by a factor of 4.49 and 3.6 respectively. Again, this is an expected result as the higher the weight we use, we care less about the increased path length returned, and more about the speed of the algorithm.

Next we examine the results of the Transit Search algorithms that we have developed. Transit Search manages to expand 2764 nodes for Warcraft 3 maps and 3464 nodes for Baldur's Gate maps. This is an improvement in speed by 2.69 times and 2.3 times over that of A*. This shows what we had hoped to prove, that Transit Search expands less nodes than A* and by a considerable margin. Compared to Weighted A*, Transit Search also performs well. It outperforms the algorithm when the weight is $\epsilon = 1.1$, this is exceptional considering that Transit Search maintains the shortest path, while Weighted A* loses its optimality to increase its speed. We note that when the weight is higher though, at $\epsilon = 100$, Transit Search does not outperform the weighted search. It should be considered though that with the increase in weight, comes and increase in the path lengths found.

We also tested Weighted Transit Search, which we expect to be faster than Transit Search, but lose path optimality. This happens to be the case, with a weight of $\epsilon = 1.1$, Weighted Transit Search expands 1625 nodes and 2341 nodes. This outperforms all the previously tested algorithms with the exception of Weighted A* with the larger weight, to which it is comparable.

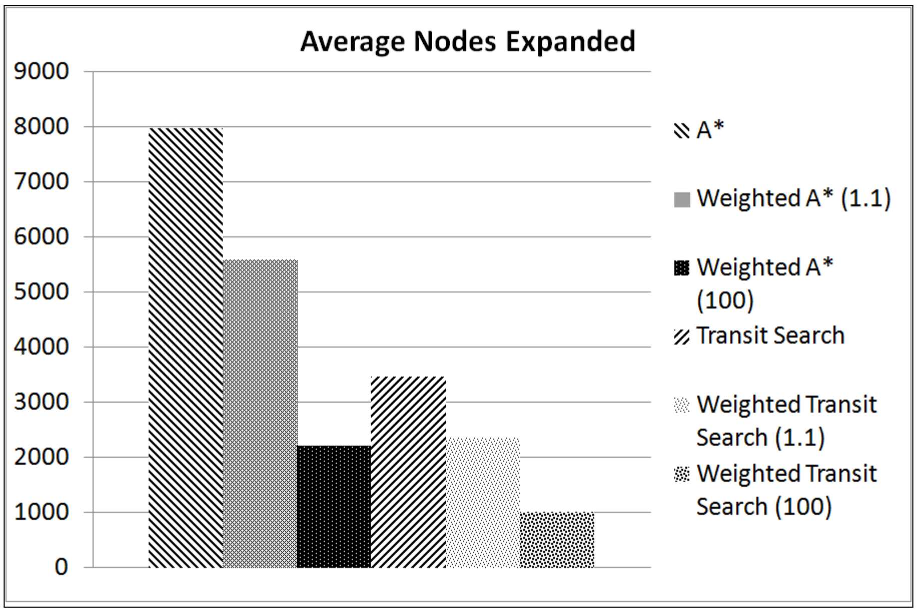

Figure 22: Average Nodes Expanded on Baldur's Gate maps

If we increase the weight to $\epsilon = 100$, the algorithm expands even less nodes at 719 and 1012 noes respectively. This is an improvement over all the other algorithms, running 10.3 and 7.87 times faster.

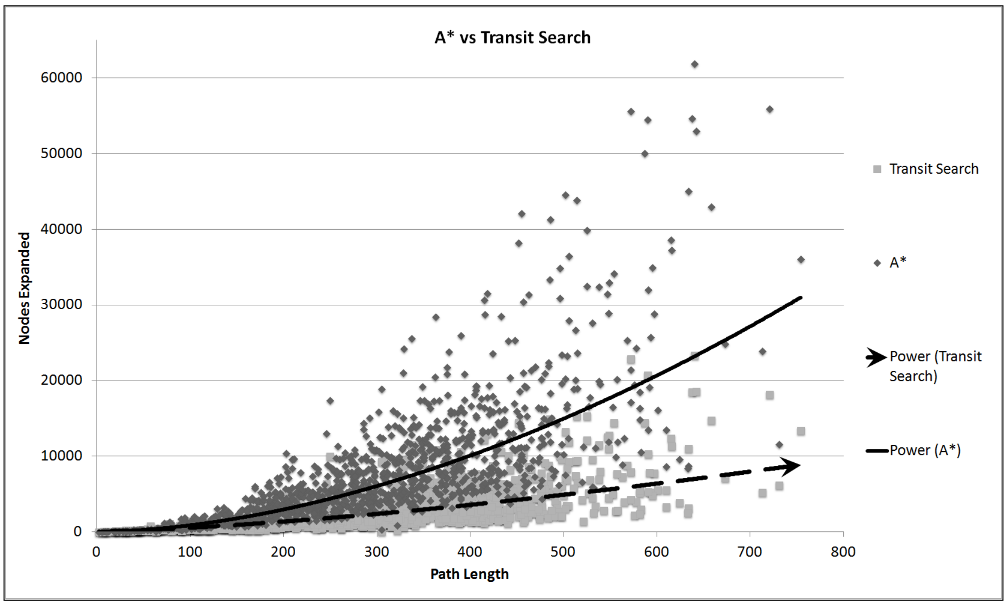

Figure 23: Comparison of A* against Transit Search on Warcraft 3 maps

Another helpful analysis we can do is to measure how the nodes expanded are affected by the shortest path length. The general trend exhibited by all algorithms is that as the path length increases the number of nodes expanded increases. This makes logical sense as A* must expand at least as many nodes as there are in the shortest path. So as the paths get longer, the lower bound on nodes expanded also increases. Also the longer the path the further the algorithms have to search. This increases their chances of searching down a wrong path, adding to the nodes expanded during search.

As expected, A* node expansion increases at the fastest rate as the path length increases. The increases in speed offered by the other algorithms not only translate on average, but also as paths get larger. The algorithms follow the same pattern exhibited by the average nodes expanded in that their ordering is the same when it comes to rate of growth. These are good results as we can expect that increasing the size of the maps and thus the average path length, our algorithms will get even more of an advantage over A*. We believe the reason for this is that all of the other algorithms in the experiments seek to improve speed by searching quickly through obstacle free areas. Weighted A* does this by putting more value in the heuristic function and the heuristic function is more accurate in open areas. Transit Search accomplishes the same by skipping over large region interiors where the heuristic is perfect. When increasing the size of the map, we often increase the size of the open areas, and allow more to occur. This increases the amount of space that the algorithms can exploit, and thus increase their speed gain over A*.

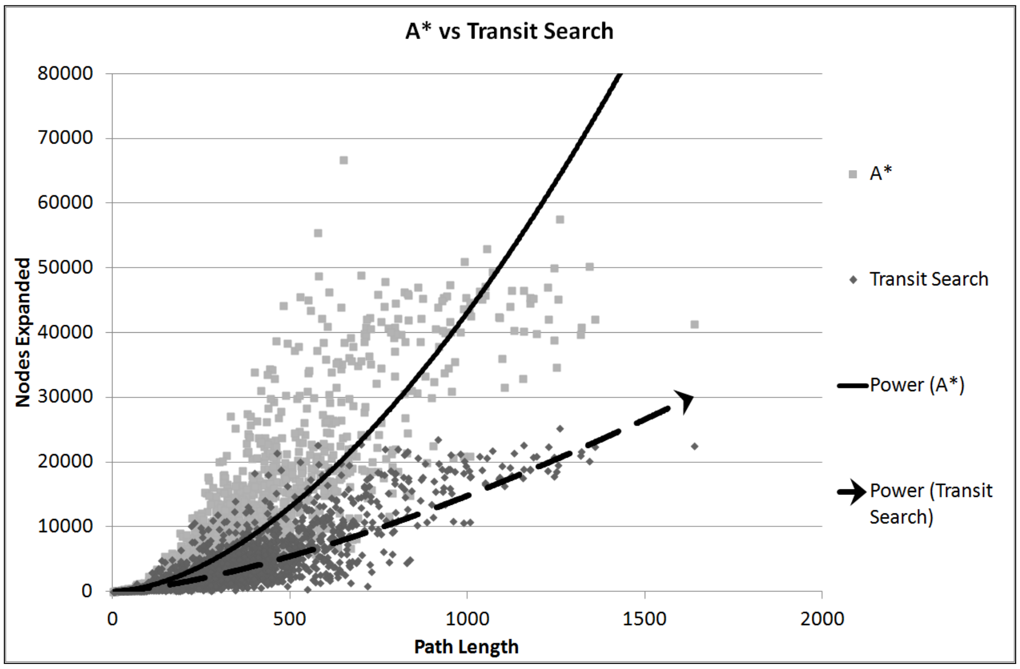

Figure 24: Comparison of A* against Transit Search on Baldur's Gate maps

## 5.2 Outliers

Although on average Transit Search handily outperforms A*, there are instances in which the opposite is true. We define these as the outlier cases in that they differ from the norm. These outliers can occur due to tie breaking employed in the algorithms. Both the algorithms use the same tie breaking, that is just the minimum f-score on a PriorityQueue, but the order in which nodes are placed into the open list can be different.

In A* when we expand a node, we evaluate its neighbours and in the case of our experiments this means the nodes above,below, left and right of the node. However in Transit Search we can evaluate those nodes as well as the transit nodes on other boundaries of the region. This can increase the number of items we place on the open list and potentially change the order in which minimum items are removed. Since the order has changed, this means A* can expand a different and possibly smaller subset of nodes than Transit Search does.

Normally, by skipping the interiors of regions, Transit Search more than makes up for extra nodes that may be searched by the change in tie breaking. If a region was smaller though, it is possible that A* could expand less nodes, since the gains by skipping are minimal in a small region. The amount of outliers recorded on the Warcraft 3 maps was 2% and on the Baldur's Gate maps was 8%.

## 5.3 Weighted Sub-Optimality Analysis

The weighted algorithms are not guaranteed to return the shortest path. Thus there is going to be some measure of sub-optimality on paths found by the weighted algorithms. While we want to increase the speed of the search, we usually also don't awful paths. It helps us to manage our trade-off of time and accuracy if we know how much sub-optimality each algorithms' results contain. A* is our baseline, as it does find the shortest path, and we will measure sub-optimality as a multiple of A*'s path length.

The average path length returned by A* was 285.74 in the Warcraft 3 maps and 304.22 in the Baldur's Gate maps. Comparing with the results of Weighted A* with $\epsilon = 1.1$ of 288.07 and 305.26, we can see that the extra path length is minimal. In fact we have that the paths are only 1.008 and 1.003 times that of A*'s or a less than 1% increase for both map sets. We did expect the results to be a low multiple as the low $\epsilon$ value ensures that the path length cannot vary too much, while still providing an decrease in nodes expanded.

In the case where $\epsilon = 100$ we expect the average path length to be larger. The experiments show that on average the path lengths found were 362.34 and 346.45 respectively. These values are in fact larger, and by a good amount in the case of the Warcraft 3 maps. The results yield a 1.268 times and 1.138 times average increase in path length over A*.

Now let us consider the Weighted Transit Searches, starting with $\epsilon = 1.1$. This algorithm returned average path lengths of 287.53 and 305.05, which are actually less than that of Weighted A*. This is surprising given that the nodes expanded for Weighted Transit Search were less than Weighted A* as well. In our data set it would always be advantageous to run Weighted Transit Search over Weighted A* at $\epsilon = 1.1$. Given the closeness of the values obtained however, we

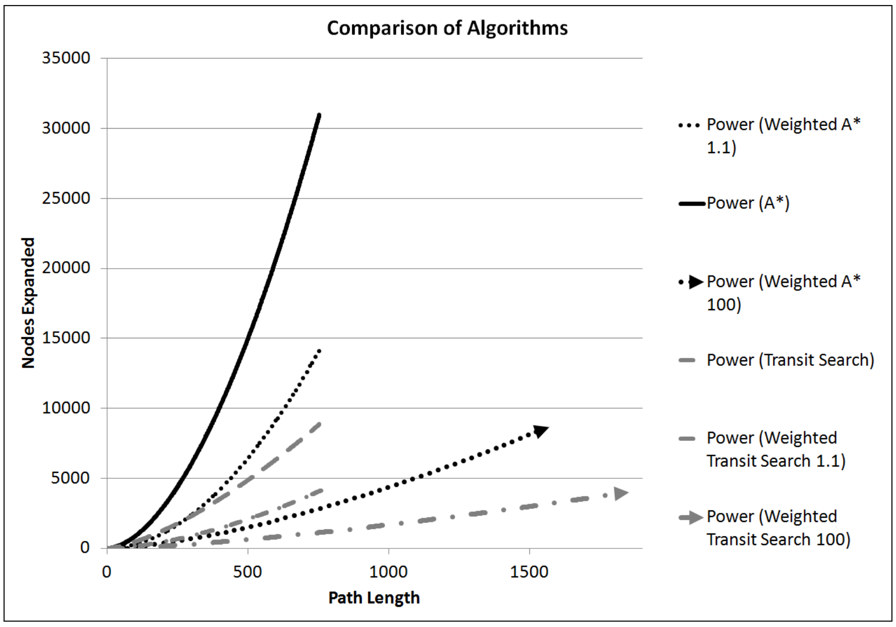

Figure 25: Interpolative Curves for each algorithm with path length vs nodes expanded on maps from Warcraft 3

cannot say this is the case always. In general we expect the Transit version to expand less nodes, and have a competitive amount of sub-optimality with plain Weighted A*.

Increasing the weight to $\epsilon = 100$ should give us insight into how the algorithm behaves when we care a good deal less about path length. With this weight the average path lengths found were, 361.35 and 354.12. This amounts to a 1.264 and 1.164 multiple of the shortest average path length. Like Weighted A* at the same weight we see a much larger jump in path length in the Warcraft 3 maps. The sub-optimality is again competitive with Weighted A* with the same weights. However recalling the amount of nodes expanded for both algorithms, Weighted Transit Search clearly expands less nodes.

While a lot of paths returned by Weighted algorithms are not the least cost, some of them are. It is important to note the incident rate of this occurring as higher rates suggest that even if you lost the guarantee of shortest paths, they might still have a good chance of happening. On the Baldur's Gate maps, with an $\epsilon = 1.1$, both algorithms have a high rate of maintaining the shortest path. Weighted A* keeps 78% of the paths optimal and Weighted Transit Search keeps 80% of the paths the same. With these low weights not only are the paths on average not much longer, but they are the shortest path the majority of the time.

A different story is told when examining the occurrence of optimality when $\epsilon = 100$. Here Weighted A* had a rate of 43% of path being the least cost while Weighted Transit Search had a rate of 21%. This is a pretty large gap, especially for Weighted Transit Search. While the average path lengths were competitive between algorithms, the Transit version has half as much incidence rate of maintaining shortest path. This continues to be the case when looking at the

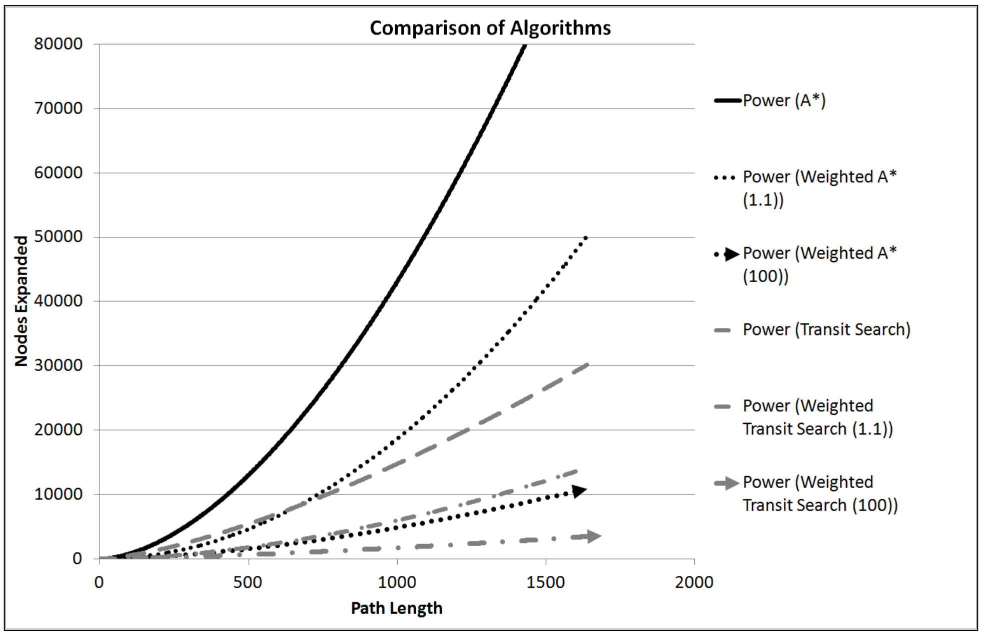

Figure 26: Interpolative Curves for each algorithm with path length vs nodes expanded on maps from Baldur's Gate

results on the Warcraft 3 maps.

## 5.4 Regions Formed

We analyze the regions formed by our Transit Search, as the quality of regions determines how well the algorithm runs. If for instance, we partitioned the graph into Transit Regions of a single node each, there would be no region interiors for our algorithm to skip. This is a very low quality partition, and result in Transit Search performing identical to A*.

First we examine the number of average number of regions formed. In the Warcraft 3 maps, there are an average number of 1728 regions per map. There are 1079.72 average region created in the Baldur's Gate maps. Also we count the average number of nodes per region. These come out to be 66.7 nodes and 147.2 nodes respectively. This tells us that in the Warcraft 3 maps we have a larger number of regions that are smaller in size and in the Baldur's Gate maps we have the opposite.

We think this is mainly due to maps in the Baldur's Gate set such as AR0044SR and AR0203SR. These maps are mainly large open areas and would be covered by a lower number of regions. This is because we can form larger rectangles within them and still maintain that h(n) = h*(n). This also increases the average number of nodes per region. In the Warcraft 3 maps, there are many areas that are large and open, but they tend to be smaller and and have a small amount of obstacles littered through otherwise open areas. This means our partition algorithm would have to split the regions more, resulting in smaller more numerous regions.

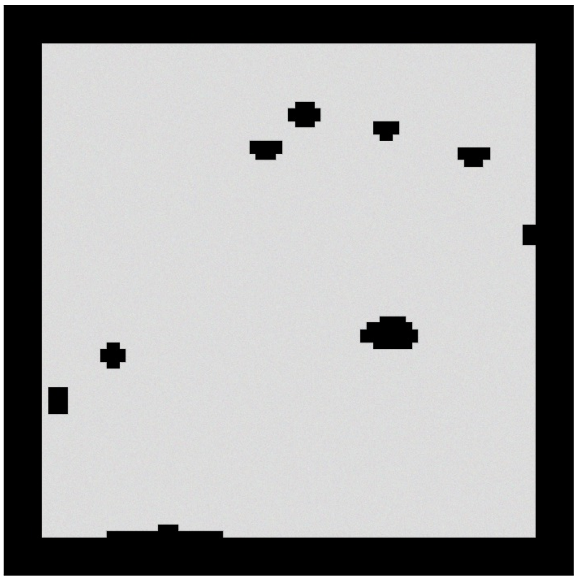

Figure 27: A very open map from Baldur's Gate

## 5.5 Comparison to RSR

The RSR algorithm turns out to be somewhat close to the one we have implemented. There are however many differences in the underlying algorithms and in the theory behind them. First, RSR only works on undirected uniform cost grids. The majority of algorithms we developed work on any graph, and the ones based on the axis aligned proof work in n-dimensions. Second, RSR is dependent upon symmetry in rectangles, whereas our implementation chooses rectangles, it could have used many other shapes (and non symmetrical). Also our approach is based using MAH not symmetry as its underlying mechanism.

Furthermore because of the MAH values, our algorithm uses less extra memory than RSR. Their extra memory is $O(n)$ as they need to store a reference on every node to its region. We do not need to do this, only the boundary nodes need extra information, and that is the MAH value and the extra edges. We don't need to know the region of the start and goal as we just A* to them when MAH value is right. Thus our extra memory is $O(R * (\text{max edges of a node} + 1))$, since our implementation has a single edge per boundary node and the max number of edges is a constant(4). This is usually much less than $O(n)$, as $n >> R$ in almost all graphs. Their algorithm also works on Octile grids though, while our implementation is made for 4 connected grids. We could the implement a Heuristically Perfect approach mentioned previously, or one with transit regions.

## 5.6 Summary

We performed experiments on 2 sets of map data from different games in industry. On these maps we tested 8600 paths on 6 algorithms resulting in 51600 searches.We found in our experiments that Transit Search outperforms A* by an average of 2.5 times less nodes expanded. It does this and unlike Weighted A* in maintains the least cost paths in the results it returns. We found that Transit Search is not always better than A*, and there exist outliers where A* expands less nodes than Transit Search. However these occasions are relatively rare occurring in only 5% of the pathfinding problems in the experiments. Overall the average nodes saved from expansion is much larger in Transit Search.

Considering Weighted Transit search and Weighted A*, the Transit version outperforms its counterpart, making it on average a better choice. In terms of increased path length, Weighted Transit Search is also competitive with Weighted A* at the same weight. We also found that there are cases where the weighted variants did return the shortest path. This occurred the most on the Baldur's Gate maps with a Weighted Transit Search where $\epsilon = 1.1$ and a remarkable 80% of the time the paths returned were the shortest.

## 6 Conclusion

This thesis presents several algorithms leading up to the development of Transit Search. We first examine Region Weighted A* which assigns to every region a set of weights to guide the search using a modified f-score. We then focus on dead ends and dead-end chains to come up with a weighting scheme that allows our search to avoid traveling down paths which are impossible given current heuristic information. We move away from weighting regions and focus more on avoiding paths that are heuristically unsound. This leads us to develop the notion of a Maximum Allowed Heuristic, which lets us avoid paths that are shorter than we need to traverse. The MAH search lets our search skip evaluating edges that lead to these paths, helping us to avoid concavities in the graph.

We declare well formed regions to gain advantage when combined with perfect heuristics. Since g(a,b) = h(a,b) within these regions we can calculate the f-score of nodes on the opposite side. This combined with using MAH values at a regional level, lets us skip entire regions, preventing unneeded node expansion. Next we introduced the Transit condition which when a region abides by, allows us to evaluate a single node on other boundaries, instead of every node on that boundary. These further reduce the extra memory requirement and also the branching factor during search. We go on to show that the Manhattan heuristic lends itself nicely to Transit Search in that it has the triangle equality. Combining this with a grid space and axis aligned regions we prove that all axis aligned regions are Transit Regions and we can leverage this to produce graphs composed entirely of Transit Regions.

Experiments were performed using Transit Search and a weighted variant on a variety of maps used in industry. We showed that Transit Search routinely ourperforms A* while maintaining the shortest path. It also outperformed Weighted A* with a small weight. The Weighted Transit Search expanded even less nodes than A* and even less than Weighted A* with the same weight. These results show that Transit Search is a more than viable search algorithm and reinforce the theory developed in this thesis.

Not only do our algorithms work on grid graphs, they are applicable to pathfinding problems on any graph, using an admissible heuristic. Search algorithms like RSR only work on uniform cost grid graphs, and only exploit the properties of rectangles. Transit Search can search many different shapes on the same domain, as well as other geometries in other search spaces.

Transit Search is a fast algorithm and works well on grid spaces making it ideal for use in video games. The fact that it returns waypoints rather than only nodes on the resultant path, make it even more appealing. This allows agents to travel between waypoints rather than every single node on the path, if it has the ability to do so. This also omits the reconstruction time of paths between waypoints that A* must do in order for its paths to be followed correctly. In addition, the preparation step is well suited to game development, as resources can be committed offline to save work on the users machines in the future. This reduces the lag that a user may encounter leading to a smoother play experience.

## 7 Future Work

In the future, work could be done in improving the method that we partition the graph with. Using our QuadTree partitioning algorithm, if a single obstacle is within a region, it would be split into smaller QuadTrees until the obstacle is in its own QuadTree. This makes Transit Regions, but it often makes many smaller regions due to the binary splitting of the algorithm. Many of these could be combined into larger axis aligned regions reducing both the number of regions and the number of boundary nodes. This has the effect of reducing the extra memory Transit Search needs and causes it to expand less nodes while searching. Better algorithms could take steps to do this, or designers could improve the partitioning manually afterwards.

Future work could also explore replacing the partition of regions with a cover of regions. Since we use the Maximum Allowed Heuristic to avoid the interior of well-formed regions, we can actually overlap the boundary nodes of multiple regions. This ensures that we don't skip nodes in other regions and lets us increase the size of region interiors. This can however increase the number of nodes evaluated on a single boundary node as it could belong to multiple regions. Overall though it would evaluate no more nodes on average, as the extra nodes evaluated might have just been evaluated on another boundary node in the partition instead.

While our implementation of Transit Search only considers rectangular regions with exactly 4 boundaries, other regions could be utilized. We have proved that axis aligned regions with a Manhattan heuristic are transit regions, not only rectangles. We could alter our regions to use axis aligned regions as well. Even with rectangular regions the boundaries we use are not always needed. Boundaries composed of only implicit boundary nodes need not be considered as there is not path through them to outside the region. With our rectangle partition approach, we generate such boundaries, but we could check and remove them. This reduces the number of boundaries and boundary nodes, while maintaining the optimality of our search.

By our definitions, Transit Search works on any graph using any admissible heuristic. However finding Transit Regions can be difficult depending on the domain. We show how to find Transit Regions easily in a Manhattan grid space, but there are many pathfinding problems that occur with different parameters. In the future, work could be done to aid in finding Transit Regions in the graphs of other problems that use pathfinding, such as the 15 Puzzle. It should be noted that in a general problem there is a brute force algorithm that could be employed to find a partition of Transit Regions. In this algorithm, we simply maintain a set of regions and add try adding each node to a region. If the region would still be a Transit Region after the addition of the node, we add it to the region. This is repeated until every node is in a Transit Region, even if it is a trivial single node region.

Finally, future algorithms could potentially search even faster by expanding boundaries themselves instead of the nodes within them. If we could group the boundary nodes together into a single node of a different graph, it might be possible to resolve the correct one after or during the search. This would reduce the number of nodes expanded on any one boundary to 1. Such an algorithm might require further restraints on the regions or the heuristics allowed but would reduce the number of nodes expanded to be a factor of the number of boundaries instead of the number of boundary nodes.

## References

[BBS09] Yngvi Bj, Vadim Bulitko, and Nathan Sturtevant. TBA*: Time-Bounded A*. *IJCAI*, pages 431–436, 2009.

[BEH$^+$03] Yngvi Bj, Markus Enzenberger, Robert Holte, Jonathan Schaeffer, Peter Yap, Edmonton Ab, and Canada Tg. Comparison of Different Grid Abstractions for Pathfinding on Maps. In *IJCAI*, pages 1511–1512, 2003.

[BMS04] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical pathfinding. *JOURNAL OF GAME DEVELOPMENT*, 1:7–28, 2004.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32(3):505–536, July 1985.

[FBFB74] R. A. Finkel, J. L. Bentley, R. A. Finkel, and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, March 1974.

[GR11] Michael R. Genesereth and Peter Z. Revesz, editors. *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation, SARA 2011, Parador de Cardona, Cardona, Catalonia, Spain, July 17-18, 2011*. AAAI, 2011.

[HBK11] Daniel Damir Harabor, Adi Botea, and Philip Kilby. Path symmetries in undirected uniform-cost grids. In Genesereth and Revesz [GR11].

[HNR68] 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), 1968.

[HPZM96] RC Holte, MB Perez, RM Zimmer, and AJ MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. *AAAI/IAAI, Vol. 1*, 1:530–535, 1996.

[Pea84] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving.* Jan 1984.

[Poh70] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3–4):193 – 204, 1970.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

[Stu12] N. R. Sturtevant. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, June 2012.

# Appendix A Additional Figures and Tables

| Algorithm | Average | Minimum | Maximum | Variance |
|---|---|---|---|---|
| A* | 7968 | 1 | 92567 | 101025544 |
| Weighted A* (1.1) | 5589 | 1 | 91536 | 87776802 |
| Weighted A* (100) | 2214 | 1 | 56168 | 23800119 |
| Transit Search | 3464 | 1 | 38024 | 18174248 |
| Weighted Transit Search (1.1) | 2341 | 1 | 33077 | 16205782 |
| Weighted Transit Search (100) | 1012 | 1 | 22308 | 4802556 |

Table 1: Nodes Expanded on Baldur's Gate maps

| Algorithm | Average | Minimum | Maximum | Variance |
|---|---|---|---|---|
| A* | 7438 | 2 | 61942 | 64285451 |
| Weighted A* (1.1) | 4460 | 2 | 53525 | 41822213 |
| Weighted A* (100) | 1657 | 2 | 29082 | 9784369 |
| Transit Search | 2763 | 2 | 23359 | 7908669 |
| Weighted Transit Search (1.1) | 1625 | 2 | 21326 | 5357713 |
| Weighted Transit Search (100) | 719 | 2 | 11706 | 1604887 |

Table 2: Nodes Expanded on Warcraft 3 maps

| Algorithm | Warcraft 3 | Baldur's Gate |
|---|---|---|
| A* | 1 | 1 |
| Weighted A* (1.1) | 1.67 | 1.43 |
| Weighted A* (100) | 4.49 | 3.6 |
| Transit Search | 2.69 | 2.3 |
| Weighted Transit Search (1.1) | 4.78 | 3.4 |
| Weighted Transit Search (100) | 10.3 | 7.87 |

Table 3: Less Nodes Expanded

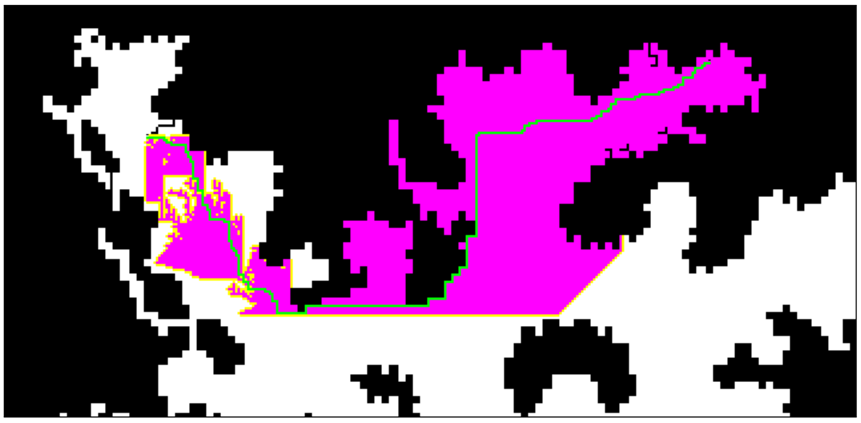

Figure 28: Search using A*

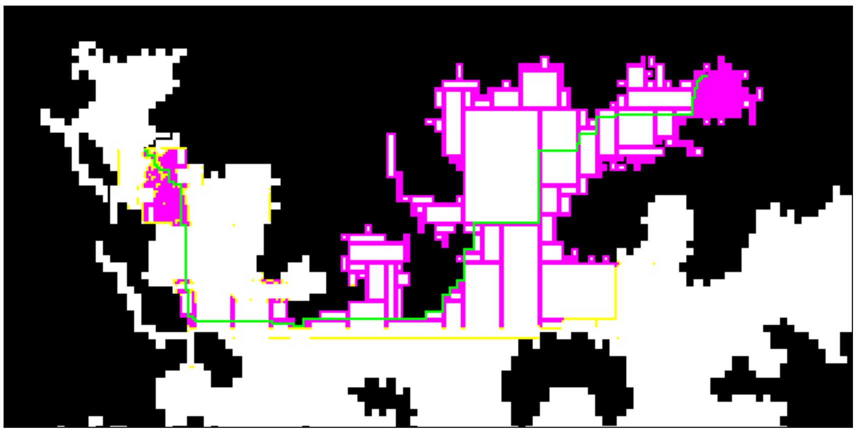

Figure 29: Search using Transit Search

| Algorithm | Average | Minimum | Maximum | Variance |
|---|---|---|---|---|
| A* | 304 | 1 | 1702 | 43410 |
| Weighted A* (1.1) | 305 | 1 | 1702 | 43682 |
| Weighted A* (100) | 346 | 1 | 1986 | 62740 |
| Transit Search | 304 | 1 | 1702 | 43409 |
| Weighted Transit Search (1.1) | 305 | 1 | 1702 | 43600 |
| Weighted Transit Search (100) | 354 | 1 | 1950 | 61982 |

Table 4: Path Length on Baldur's Gate maps. Note a single path's data was corrupted.

| Algorithm | Average | Minimum | Maximum | Variance |
|---|---|---|---|---|
| A* | 286 | 2 | 754 | 19186 |
| Weighted A* (1.1) | 288 | 2 | 758 | 19712 |
| Weighted A* (100) | 362 | 2 | 1568 | 48175 |
| Transit Search | 286 | 2 | 754 | 19186 |
| Weighted Transit Search (1.1) | 288 | 2 | 760 | 19575 |
| Weighted Transit Search (100) | 719 | 2 | 1853 | 48206 |

Table 5: Path Length on Warcraft 3 maps

| Algorithm | Warcraft 3 | Baldur's Gate |
|---|---|---|
| Weighted A* (1.1) | 58% | 78% |
| Weighted A* (100) | 20% | 43% |
| Weighted Transit Search (1.1) | 63% | 80% |
| Weighted Transit Search (100) | 11% | 21% |

Table 6: Percentage of Paths Optimal

# Appendix B Glossary

**A***

A well known search algorithm that uses heuristic information to find an optimal path.

**Boundary**

A collection of nodes in a well-formed region that connects to the rest of the graph.

**Closed List**

The list of nodes already expanded by a search algorithm.

**Heuristic**

A function that estimates the cost of a path between two nodes.

**Ignored List**

The list of nodes ignored by Transit Search, and would otherwise be added to the open list.

**Manhattan Distance**

A distance between two nodes defined as $h(x, y) = \sum_i (|x_i - y_i|)$. It is often used as a heuristic function in pathfinding.

**Maximum Allowed Heuristic**

A value representing the maximum cost that any path going through an edge can obtain. This is used to allow searches to avoid edges when we know the minimum cost we must incur is larger.

**Open List**

A list of nodes that have been evaluated and are awaiting expansion by a search algorithm.

**Pathfinding**

The problem of finding the least cost path between nodes in a graph.

**Perfect Heuristic**

A heuristic function that exactly estimates the actual cost of the shortest path between nodes.

**Quad Tree**

A tree data structure where each node has 4 children. Often used in spatial partitioning to divide 2D space into 4 quadrants.

**Region**

A collection of nodes in a graph.

**Shortest Path**

The least cost path between nodes in a graph along the edges that connect them.

**Sub-Optimality**

The measure of the loss of optimality of paths, when compared to optimal paths, such as those found by A*.

**Transit Search**

A search algorithm that uses regions to improve search and is similar to A*.

**Weighted A***

A search algorithm based on A* that uses a weight to speed up searching at the cost of optimality

**Well-formed Region**

A region where the shortest paths between nodes in the region is composed of only nodes also within the region.

# Vita Auctoris

| | |
|---|---|
| **NAME:** | Justin Moore |
| **PLACE OF BIRTH:** | Brampton, Ontario |
| **YEAR OF BIRTH:** | 1988 |
| **EDUCATION:** | 2006-2011, B. Sc.[H]<br>School of Computer Science<br>University of Windsor<br>Windsor, Ontario, Canada |
| | 2011-2014, M. Sc.<br>School of Computer Science<br>University of Windsor<br>Windsor, Ontario, Canada |