Variable-Resolution A*

BY
Kyle Walsh
B.S., Computer Science, Southern Connecticut State University ‘05

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute of Technology
Redmond, Washington
United States of America

Fall
2007

Thesis Advisor: Dr. Dmitri Volper
DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS OF [name]

HAS BEEN SUCCESSFULLY COMPLETED ON [date]

TITLE OF THESIS: [thesis title]

MAJOR FILED OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

[signature]

[name], Chair

[signature]

[name]

[signature]

[name]

[signature]

[name]

APPROVED :

[signature] [date]

[name] date
Graduate Program Director

[signature] [date]

[name] date
Associate Dean

[signature] [date]

[name] date
DEPARTMENT OF COMPUTER SCIENCE

[signature] [date]

[name] date
DEAN

INSTITUTE OF DIGIPEN INSTITUTE OF TECHNOLOGY

PROGRAM OF MASTER'S DEGREE

THESIS APPROVAL

DATE: _____[date]_____

Based on the CANDIDATE'S successful oral defense, it is recommended that the thesis prepared by

[your name]

ENTITLED

[thesis title]

Be accepted in partial fulfillment of the requirements for the degree of master of computer science from the program of Master's degree at DigiPen Institute Of Technology.

[Signature]

[Name]
Thesis Advisory Committee Chair

[Signature]

[Name]
Director of Graduate Study Program

[Signature]

[Name], Associate Dean

[Signature]

[Name], Dean of Faculty

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

Table of Contents

Abstract

This paper discusses the merits of a variable-resolution pathfinder that should speed up A* searches on a 2D-grid by cutting down on search space size. The basic idea is to start with a low resolution graph and increase resolution automatically as and where necessary. The search space starts off only as two cells, containing goal and start points, and with arbitrary obstacles. Each time A* fails to find a path on the current graph, the resolution is increased by splitting cells, and A* repeated. This approach leads to fewer total node-expansions than applying A* just once to the full-resolution graph. The advantage over related approaches like quad trees and hierarchical A* is the automatic, on-demand creation of the required variable resolution that can trivially handle a dynamic map.

Introduction

The goal of this section is to highlight the background information researched in relation to various searching algorithms and optimizations in artificial intelligence that inspired the research behind VRA*. The researcher reviewed existing sources based off of, or related to, the A* algorithm with the goal of creating an optimization or separate algorithm to use in real-time applications, such as computer games.

The research sequence progressed from introductory searching information to modifications and optimizations of A* found in published artificial intelligence journals. Uninformed search algorithms (breadth-first and depth-first search) were covered first. Learning these algorithms allows one to understand the importance of space and time

complexity in searching. Heuristics, informed search, and adversarial search algorithms were explored next, with the main focus on A*. A*'s implementation in games was then reviewed. Following A*, optimizations and extensions of A* were researched; iterative-deepening A*, real-time A*, learning real-time A*, and moving target search were all covered. Finally, the possibility of learning in search was observed, by exploring the merits of Hierarchical A* and the Parti-game algorithm.

Background Information

Before discussing searching techniques, a small amount of background information must be provided. Many of the methods researched will be described in terms of a search tree, with the starting node being the root of the tree, and subsequent nodes being expanded outward until the leaves of the tree are reached (leaves being nodes with no possibility for expansion). Nodes that can be expanded from the current node and are exactly one depth level below the current node are children of the current node. The node that the current node was expanded from is the parent node. The fringe of a search tree is any nodes that have been generated but not yet expanded.

Searching techniques will be analyzed based off of completeness, optimality, time complexity, and space complexity. If a solution exists in a search space, then a technique is complete if it finds that solution. A technique is optimal if it finds an optimal solution based off of certain criteria. Time complexity is based off how long an algorithm takes to find a solution, and space complexity is based off how much memory is needed to perform the search.

In addition to the terms mentioned above, several quantities are used in analyzing search techniques. Branching factor (b) is the maximum number of successors of any

node. The depth of the shallowest goal node is represented by *d.* Finally, the maximum length of any path in the state space is represented by *m* [1].

Uninformed Search Techniques

Uninformed search algorithms have no additional information about states in the search space beyond what's provided in the problem definition [1]. These algorithms generate successors and determine whether or not a goal state has been reached. The three uninformed search strategies covered in this research are Breadth-first search, Depth-first search, and iterative-deepening search.

Breadth-first search

A Breadth-first search (BFS) starts at the root, expands that node, and expands all successors of that node. Following that, all the successors of those nodes are expanded. This pattern continues until either the goal is reached, or the entire space is searched. All nodes at a given depth are expanded before any nodes on the next depth level are expanded [1].

BFS is complete as long as the goal node is at a finite depth and the branching factor is finite. BFS will eventually find the goal node after expanding all shallower nodes. If there is more than one goal node in the search space, it will find the shallowest goal node. Keep in mind, the shallowest goal node may not be the most optimal solution (depending on the problem).

The problem with BFS is that it can have both high time and space complexity. In terms of space, in the worst-case scenario, BFS could expand every possible node but the last node at any given depth. This might not be a problem if branching factor is trivial,

but in most cases it is not. So, in terms of big O notation, the total number of nodes generated for BFS is $O(b^{d+1})$, where b is the branching factor and d is the depth. Since every node that is generated must remain in memory (because it is either part of the fringe, or fringe ancestor), the time complexity is just as large as the space complexity plus one (the root). Time and space complexity for BFS become exponentially worse as b and d increase, and specifically, space complexity can become extremely problematic.

Depth-first search

Depth-first search, or DFS, proceeds immediately to the deepest level of the search tree, the deepest node in the current fringe, where nodes have no successors [1]. As DFS expands these nodes, they are removed from the fringe, and if the goal node is not found, the search back-tracks to the next shallowest member of the fringe.

One of the benefits of DFS is that its space complexity is generally good. DFS only needs to store the current path from root to leaf, as well as the remaining unexpanded siblings for each node on the stored path. Nodes can be removed from memory once their descendant nodes have been fully explored. Specifically, the space required for a DFS is $bm + 1$, where b is the branching factor and m is the maximum depth. In terms of big O, DFS would be $O(bm)$.

DFS is prone to getting stuck down long, incorrect paths if a wrong choice is made. DFS is neither optimal, nor complete. Finally, in the worst-possible case, DFS could generate all the nodes in the tree, making it no better than BFS.

Iterative deepening search

Iterative deepening search (IDS) can be thought of as an extension of DFS where

the depth of the search is limited to prevent expanding too deep along a suboptimal path. IDS will find the best depth limit by gradually increasing the depth limit from 0 to 1, 1 to 2, and on, until a goal is found. The increase continues until it reaches *d*, the limit of the shallowest goal node. This technique combines the benefits of DFS and BFS by virtue of having modest memory requirements (like in DFS) and completeness under finite branching factors (like BFS) [1]. Out of all the uninformed search methods, IDS is the most preferred in cases where the search space is large and the depth of the solution is unknown.

Informed Search

Informed searching, using problem-specific knowledge, can find solutions more efficiently than uninformed searching. Informed searches use an evaluation function to select which nodes to expand to (usually the low score), and store the fringe in ascending score values in a data structure. Heuristic functions are often used as the evaluation functions in informed searches. They yield the estimated cost of the cheapest path from a given node to a goal node. Almost all of the algorithms researched in this study used an admissible heuristic evaluation function. An admissible heuristic never overestimates the cost to reach a goal [1]. It assumes the cost of solving a problem is less than it actually is. Using an admissible heuristic in an informed search algorithm will lead to paths that appear to be the best (cheapest) at time of expansion.

A search background*

What sets A* apart from its competition is its ability to quickly converge on the best possible path, while saving many CPU cycles [2]. A* is able to choose the best-

available path by aid of an admissible heuristic, as well as a node score specific to each node. These two numbers are used to determine which node to expand, and is done using the simple equation:

score: f(n) = cost from start: g(n) + heuristic: h(n)

Simply put, the cost from start takes into consideration moves made up to the current point, while the heuristic attempts to estimate the future cost considering the proximity of the current location to the destination.

The heuristic can be calculated in several different ways, but the simplest method to describe to the layperson, in terms of calculating a heuristic in game programming, is the Manhattan distance. To conceptualize the Manhattan distance between any two tiles, think of the way streets are laid out in the New York borough of Manhattan: as a simple grid. When calculating the directions from one building in Manhattan to the next, one only has to know how many blocks over and how many blocks up or down is needed to get to the destination. The sum of these movements is the Manhattan distance. Note that the Manhattan distance does not take into account obstacles. This may seem like a flaw until one considers that the heuristic is only an estimate of the future, and the traveler would not know if obstacles were present without having already visiting those tiles already. This feeds directly into A*'s strength, because the Manhattan heuristic will give A* an under-estimate of the cost to get to the goal node.

A in a game-related situation*

Since A* is the basis for much of this survey's research, as well as the building block for VRA*, it will be explained in great detail here. The specific environment is a game-like situation where a non-playable character (NPC) controlled by an AI engine is

trying to find a player in hiding. Figure 1 below shows the search area split up into tiles, where shaded tiles represent obstacles and white tiles represent traversable nodes.

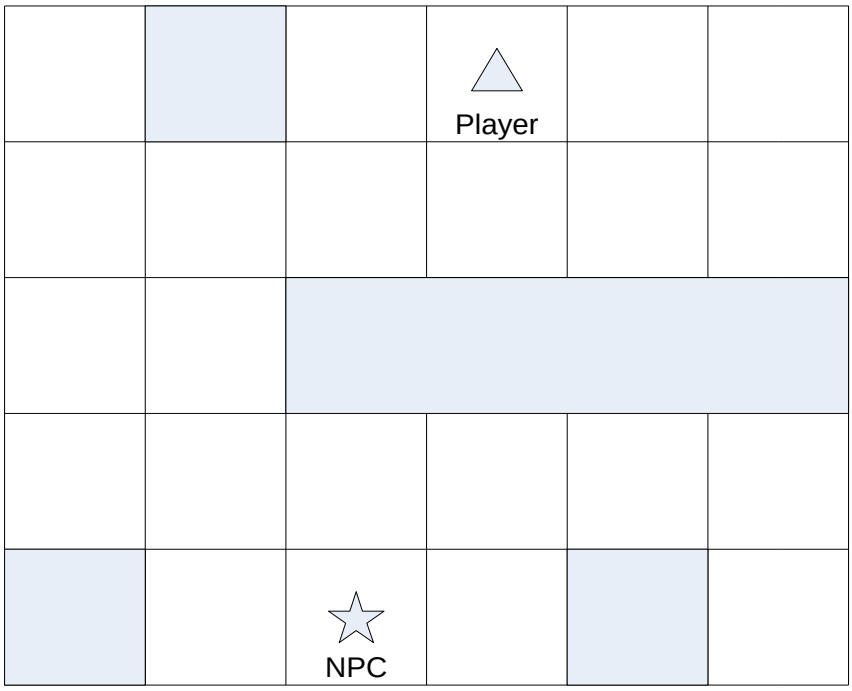

Figure 1 - A Search Area*

The triangle represents the goal, and the star represents the NPC (which is also the starting point). A* begins by analyzing the search area and calculating the shortest possible path between the NPC and its destination. To be able to do this, the traversable tiles, and the cost associated with moving to each, must be stored in a list. Doing this ensures that A*, after careful analysis of the search area, will pick the quickest and most efficient path. This list is referred to as the *open list* [3]. On the first pass through, only the root node tile is on the open list. Tiles are added to the list as A* branches out continuously on its search. Once the open list is populated, A* traverses the list and searches for the tiles adjacent to each tile in the list. Adjacent tiles are analyzed to

determine if they are traversable or not. Once a tile has been expanded, it is stored in the *closed list*. Figure 2 below expands upon Figure 1 by detailing possible tiles the NPC could traverse.

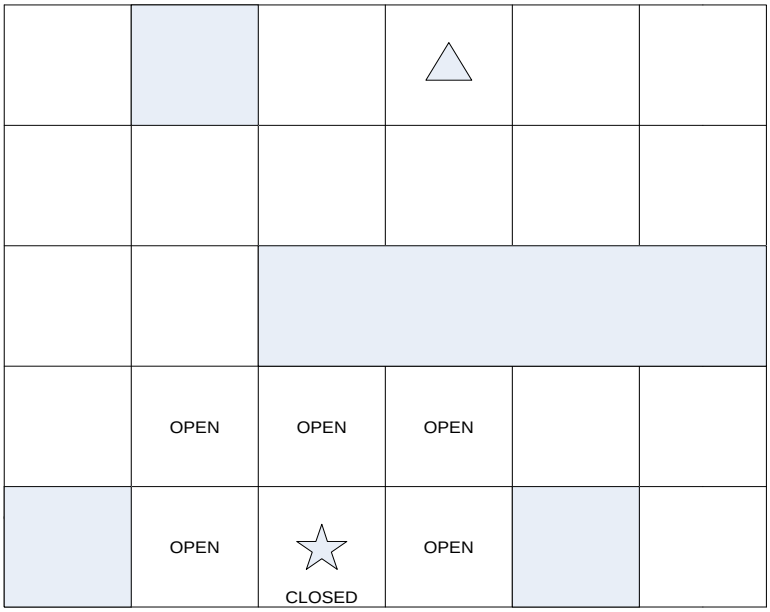

Figure 2 - Tiles Available for Traversal

As this diagram depicts, five tiles are open for traversal, while the current tile now is the lone member of the closed list. The open list keeps track of neighboring tiles, and a tile's parent is the single tile the NPC had to travel from to get to its current location. Figure 2 shows only one pass has been performed, therefore all five tiles point back to the starting tile as their parent tile. After fully analyzing the search area and reaching the endpoint, A* can use the parent-child links to trace back to the starting location and determine which path is best. The cost of moving from adjacent tiles in this scenario is one. The example, like many implementations of A*, uses the Manhattan heuristic calculation described previously. Figure 3 below details the total cost *f(n)*, tile score *g(n)*,

and heuristic *h(n)* of all open tiles from the starting point.

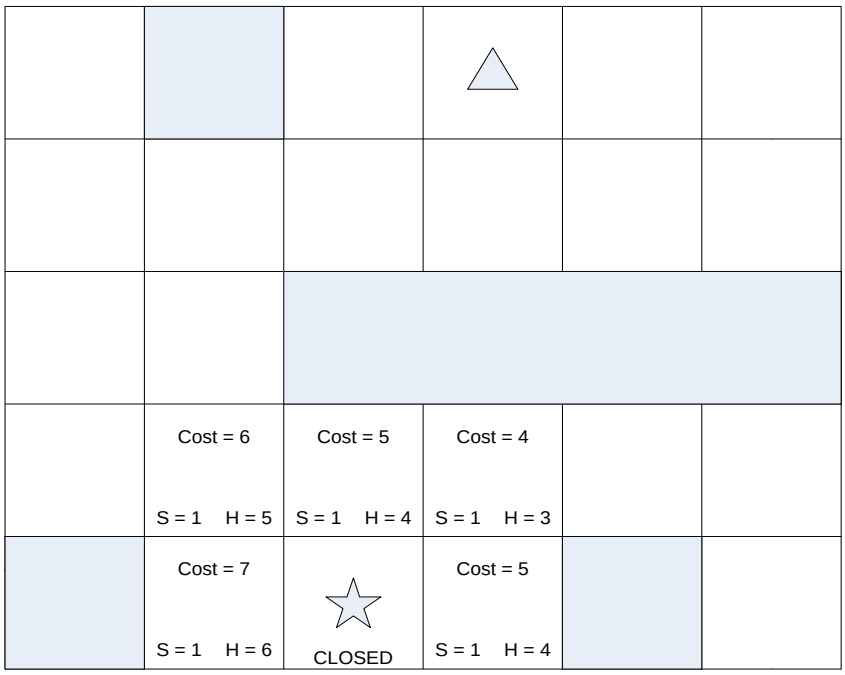

Figure 3 - Initial State for A Demonstration*

In this example, the tile with the lowest cost is the tile to the upper right of the starting location. Moving here first is not a major issue, however, because once this path reaches a dead-end or becomes too costly, A* will alter its analysis toward a different path whose tile starts with a lower cost. Moving along, the current tile is added to the closed list and the tile to its right is added to the open list. This tile to the right will also have a heuristic value of four, but because it is two moves from the starting location, its cost will total six. Knowing that there are tiles on the open list with values less than six, A* will examine those. There are two tiles on the open list with cost values of five: the tile directly above the starting location and the tile directly to the right of the starting location. Let's say A* chooses the tile to the right. Notice that there are no new paths to venture from once the tile to the right is made the current tile. The tile to its right is an

obstacle, and the tiles to its left and above are already on the closed list. Also, the tile to the upper left is already on the open list, so that would not be considered. So, A* goes back to the open list, and this path is ruled out. With only one other tile having a cost of five, the next move is clear. Figure 4 details the search area in its current state. Arrows represent the relative location of each tile's parent tile.

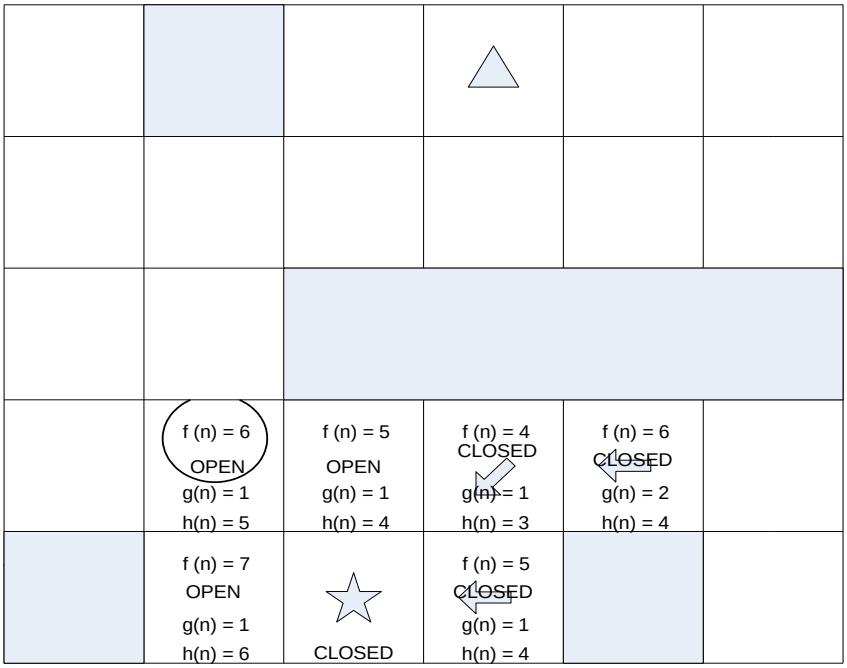

Figure 4 - Narrowing Down the Search Area

So, the current tile becomes the tile above the starting location. This tile, like the last one we evaluated, has no new tiles to link to: the tiles above it are obstacle tiles, the tile to its left is already on the open list, and the tile to its right is on the closed list. A* will ultimately go with the tile that lies up and left from the start point (marked with a circle for clarification purposes). From there, A* will calculate the cost and record the parent tiles on its way to reaching the destination. The path A* chose, along with the cost associated with moving to the endpoint on that path, is detailed in Figure 5 on the

following page.

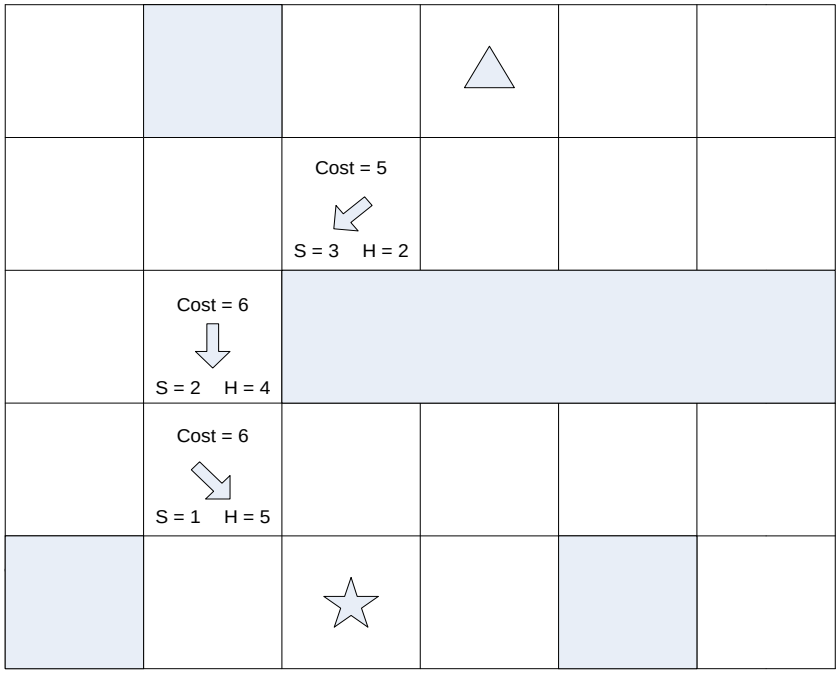

Figure 5 - Search Area after Completion of A Analysis*

As you can see, A* calculates the optimal path. This example was very simplistic, but not trivial. An endpoint and starting point were given to the algorithm and a path was found in an efficient manner.

Adversarial Search

A search space with multiple competing agents, where each agent considers the actions of other agents present in the search space, is the perfect example of an environment where adversarial searching takes place. These competitive environments are also known as games [1]. In this kind of search, both agents attempt to make optimal moves, and there are several algorithms in existence to attempt to find these moves. In terms of the background research performed for this thesis, the minimax algorithm and

alpha-beta pruning were of primary focus. Adversarial search algorithms factor into the research for VRA* because VRA*'s cost analysis functions make the best choice available assuming worst-case conditions, guaranteeing their validity in a competitive setting. Also, more advanced algorithms discussed later reference these core concepts, so it is in good practice that these concepts are discussed early on.

Defining the search space

The game examples discussed will consist of two players, *max* and *min,* which compete via alternating moves until the game is over, and points are awarded. The game is just a complex search problem, with each possible game state and legal action associated with that state having a node in the game tree. Using the familiar tree structure, much of the logic of previous searches can be extended to this example, but not without modifications. To determine how both players choose their actions, the minimax algorithm is used. This algorithm is the focus of the next subsection.

Minimax algorithm and optimality

If this were a normal search problem, then the optimal solution would be a series of movies by one player resulting in a winning terminal state. In games, though, the opponent has equal opportunity to try and win the game as well, and optimal solutions are sometimes unavailable to one player as a result of the actions of the other. Put another way, when playing games, an optimal strategy is one that leads to outcomes at least as good as any other strategy when one is playing an infallible opponent [1].

One way to find an optimal strategy in our game tree is to score each node with a minimax value. *Max* attempts to move to states of maximum value, while *min* prefers to

move to states of minimum value. In navigating the tree, one assumes that both players move optimally.

The minimax algorithm proceeds through the search tree, computing the minimax decision for each current state based off of its successors. The algorithm recursively calls itself for each successor, until the leaves of the tree are reached. It is at this point that the minimax value is backed up through the tree as the recursive calls unwind [1].

A minimax search results in a complete depth-first exploration of the search space. If *m* is the maximum depth of the search tree, and *b* is the number of legal moves at each node, then the time complexity of minimax is $O(b^m)$. If the algorithm generates all successors at one time, then the space complexity is *O(bm)*, while if the algorithm generates successors one at a time the complexity is then *O(m)*. Keep in mind time estimates are hypothetical because in real games with human players, the time varies randomly.

Alpha-beta pruning

Minimax search is problematic in terms of space complexity because it is exponential in terms of number of moves performed. Through pruning, the process of eliminating certain parts of the search tree, we can reduce this exponent. Alpha-beta pruning, when applied to a minimax search tree, can return the same move as an unaltered minimax tree, but without the need to process branches that have no possible influence on the result of the search.

In general, the algorithm works like this: start at any given node; say *n*, such that a player has a choice of moving to that node. If a better choice exists, say a new node *m*, at either the parent node of *n* or any point further back up the search tree, then *n* will never

be reached in actual game play because *m* would be the player's choice [1]. Once enough information about *n* and its descendants is learned, it can be pruned.

The name of the algorithm comes from the two parameters it stores throughout the pruning process: alpha and beta. Alpha is the value of the best choice found so far along the path for *max.* Beta is the value of the best choice found so far along the path for *min.* These values are updated as the search is performed, and remaining branches at a node are pruned when a value is worse than the current alpha or beta value depending on which player is moving. A drawback to Alpha-beta pruning is that it's highly dependent on the order of the tree to be effective. If, for whatever reason, the optimal alpha and beta values are off to the far side of the tree, then the algorithm will still have to process most of the search space before it can prune anything.

Extensions and Optimizations for A*

The first half of the research in this survey was gleaned from traditional references in artificial intelligence, covering ground-level topics used for the more complex ideas to follow. After establishing a solid background in searching techniques, the research material changed to journal articles containing optimizations and extensions to known commodities. With a now thorough understanding of A* and other search algorithms, the many optimizations and extensions researched can be discussed.

A and IDA* with optimizations*

Many implementations of A* make use of the Manhattan heuristic as its heuristic evaluation function. The open and closed lists used by A* are often times implemented

using a queue of some sort, often a priority queue [3]. The optimizations researched either used different heuristics, different data structures, modified the original algorithm, or did a combination of the three. For example, a simple modification of A* is Iterative-deepening A* (IDA*), which as the name implies is just performing normal A* but at a set depth that increases to some limit *d*. This modification of A* is nothing too groundbreaking, but is an example of how a small tweak to the algorithm can lead to performance gain in some situations. As with any trade-off, though, several of the optimizations researched either sacrifice time for optimality, or optimality for time. Whether this trade-off has a negative effect depends on the specific situation.

When expanding nodes, A* chooses the node with the smallest *f(n)* cost. The cost of finding this node, then, is very important. The methods for finding the cost of a node, and the data structures used to store the open list, are two components to A* that have been optimized in various ways in an attempt to improve the algorithm.

While the name open list implies that open nodes are stored in a list, this does not always have to be the case. In fact, for very large open lists (10,000 or above), traversing the entire open list to find the best f(n) score can be very time consuming. So, the standard implementation of the open list is usually as a priority queue. Using a queue means that insertion can be done in logarithmic time, while extraction can be done in linear time.

Certain researches maintain that because f(n) values are bounded by relatively small numbers, it is possible to implement a data structure that inserts nodes in constant time, while extracting the best node in a short time [4]. This is done by using an array of stacks, where the index of a stack in the array is the common f(n) value of all of the

nodes. There is no need to clarify why inserting into a stack takes constant time, by definition of a stack. What's curious, though, is the claim that extraction can be done in even less time. Upon further thought, though, this claim makes sense. By combining all nodes of a common f value in one stack, and storing those stacks in an array, the only traversal of open nodes is greatly reduced.

Final Thoughts on A optimizations*

After researching various optimizations to A*, it's interesting to note why there aren't that many attempts at an optimization that makes decisions on the fly about what and where to search. Is this because A* works well enough as it is, and does not need to be fixed? Is it because, in terms of game AI, the reward involved does not seem to outweigh the risk and/or development time? These questions are what inspired the creation of the centerpiece of this thesis' research: Variable Resolution A*.

Hierarchical Pathfinding

All previous methods discussed worked on a uniform grid of fixed size, which is a fairly restrictive environment (especially in the context of game programming). We're looking for a method that is more dynamic. Various research exists in the area of dynamic search spaces, but some of this research is beyond what the goal of this thesis was: a pathfinder that could operate on a set search space, but only operate on nodes in the graph as and when necessary. The closest researches have come to fulfilling this desire is in the specific realm of hierarchical pathfinders that abstracted states in the search space. Abstraction of a search space is the act of replacing one state space with another one, the new space being an amalgamation of two similar states. The end result

of this being an easier to search space, leading to less work performed overall. Abstract searching based off this practice, then, guides the movement in the original search space. Though the goal of this thesis is to go from the coarsest resolution to the finest, the lessons learned from research performed in the opposite direction cannot be ignored. The work of these researchers served as a springboard and an inspiration to the work conducted in this thesis.

*Hierarchical A**

Hierarchical A* (HA*) was one of the first algorithms to combine state abstraction with A* as an effective way to reduce the number of nodes expanded in a search on a given space. The ultimate goal of the research in [6] was to automatically create admissible heuristics for A* search via abstraction. HA* combines state space abstraction with caching techniques, making it so repeated searches on already processed data via abstraction are not necessary. [6] state that there are two types of abstraction, embedded abstractions and homomorphism abstractions; they used homomorphism abstractions in HA*. An embedded abstraction adds more edges to the state space graph, whereas a homomorphism abstraction combines states, which effectively removes edges from the graph.

HA* Abstraction Technique

HA* abstracts states using the "max-degree" STAR technique described in [7], which is a pretty straightforward strategy. First, the state with the largest degree is grouped together with any neighbors it has that are within a pre-defined radius. Then these states are combined into a single abstract state. Next, this step is repeated until all

states are assigned to a new abstract state. Finally, this whole process is completed over and over until only a trivial state space remains. This technique goes from high granularity to low, ending in the trivial case. As you shall see in the VRA* section, our algorithm does one quick preprocessing of the original search space, then proceeds from low granularity to high, only increasing granularity as and when needed. Not only the difference in abstraction "direction" (high to low vs. low to high) should be noted, but also the rigid requirement of an abstraction radius exists in HA*. This value must be tweaked to fit varying search spaces because a radius that is too small does little to reduce overall size of the search space, and a radius that is too large essentially corrupts the integrity of the search space by amalgamating too much topographical information. VRA* is not rigid in this sense because it only goes to the required level of abstraction per application, requiring no intervention on the user's part. In the specific realm of game development, this is a very good result because it's one less function a programmer has to worry about providing for the game design team.

HA* Heuristic Technique

Having a space to search on isn't the only requirement for A*; one also needs cost evaluation and estimation functions. Remember that without a heuristic (technically a heuristic of zero), an A* search is actually just a Dijkstra's search. Heuristics are supposed to speed up searching, and not hinder them. In complex search spaces, such as abstracted search spaces, if the cost of calculating the heuristic for the search space is high, then you are losing the benefit of the heuristic in the first place. The counterpoint to this is that the better the heuristic, the more focused an A* search will be. When we get to

the implementation details for VRA*, it is demonstrated how cheap the custom heuristic developed for that algorithm is since VRA* calls a simple line testing function for cost evaluations.

HA* uses heuristics from the highest level and passes them down to the next levels of abstraction, hence the need for caching of previous information. It should be noted that this adds to the space and time complexity of this algorithm, since this data must be modified on the fly while the recursive calls to the state abstraction function are taking place. Here, again, is another difference between HA* and VRA*: data is cached on the fly between levels of abstraction in HA*, where in VRA* there is only one lookup table that is accessed at constant time which contains data on only the highest resolution. No other data caching is performed or necessary.

Final thoughts on HA*

The trailblazing effect of HA* cannot be ignored, because with the initial research performed in [6], many of the ideas and improvements in this paper and others might not have been theorized. The concept of homomorphism abstractions to lessen the search space in an A* search was groundbreaking at the time, and as such was kept, in theory, in the work performed in this research. Areas of differentiation include the calculation of heuristic and cost values, the method in which the homomorphism abstractions are performed, and the caching of data. It should be noted that in [6], the authors state that the development of an abstraction technique that’s more-properly suited for a hierarchical A* search is an important topic of future research. We believe that the abstraction technique developed here addresses this need.

*Hierarchical Path-Finding A**

As previously mentioned, HA* serves as an inspiration to those researching the merits of A* search on abstracted search spaces. Inspired by both HA*, and the relatively small use of abstract pathfinders in computer and video games, the authors of [8] created their contribution to the field: Hierarchical Path-Finding A* (HPA*). Due to pathfinding on large maps tendency to cost more computationally, any way to avoid the inherent bottlenecks related to this problem would be a great benefit to all in the game development community. Like HA*, HPA* is more-or-less A* with modified cost evaluation and data caching functions. HPA*, according to its authors, does a better job at navigating search spaces since its priority is just that, whereas HA*'s priority lied in creating domain-independent heuristics. This claim, as well as the similarities and differences between HPA* and VRA*, justify its inclusion in our study.

HPA* Abstraction Details

The authors of [8] provide a logical and simple theory for state abstraction, likening their implementation to the thought process a human would follow when planning a long-distance trip between two cities. First, one would investigate how to get to a major highway from the current location in the first city. Next, the traveler would take the major highway from city A to city B, then get off the highway at city B. Finally, the traveler performs more precise maneuvers at the local level in city B. The traveler would need detailed road maps in both cities, but only a cursory knowledge of the geography in-between since he or she would be staying on the major highway system for the entirety of the middle step. This is the metaphor they use to describe their abstraction

process they refer to as clustering. The start and goal points in a search space can be thought of as two cities, as well as the abstracted nodes in the search space, are clusters. The connections between them are the major highway systems. Navigation between clusters, then, is extremely simple. In VRA*, as you'll see, we go a step further and remove even the local roadmap, so to speak, without sacrificing too much path accuracy.

It should also be noted that HPA*, though it can support more, only has two levels of abstraction. Regardless of how many levels of abstraction it has, the fact remains that this number is established and set ahead of time as a restriction. VRA*, in contrast, will go as far as necessary with abstraction. In most cases this is a benefit; in some cases, as described briefly in [8] and in this paper in the follow pages, it can be a detriment. Pathfinding can often be a delicate balancing act between finding the perfect path, finding the path that looks "right", and finding the pretty path.

A final note to make on the abstraction details of HPA* is that they only allow a certain amount of "connections" between clusters, meaning points of entry and exit are limited. This is how their algorithm is able to perform at very fast speeds because it applies restrictions on movement, which cuts down on the number of neighbors a node in the search graph can navigate toward. Using this restriction, as well as path-smoothing as a post process step, the authors of [8] are able to get paths within 1% of optimal. This is very good, but raises an issue related to gameplay immersion: how perfect do you want opponents to be? One of the drawbacks and reasons for optimization commonly cited by game developers is that sometimes A* is just too good, rendering results that are not human-like in the eyes of the player when viewing his or her computer-controlled competition. This is in contrast to VRA*, which allows any movement between cells that

are not obstructed via a line test that resembles what a computer-controlled entity can "see". Again, there is a balancing act to be performed in terms of optimality vs. aesthetics. With an abstracted search space construct for HPA* explained, the details of how the search through this abstract space can now be discussed.

HPA*'s Hierarchical Search Technique

After the search space is generated, distances between clusters is pre-computed and stored for use later in the cost evaluation functions of A*. Before the search is executed, the start and goal positions are connected to clusters in the search space. A* is then run on this abstracted search space using their cost evaluation functions, and then the steps for the agent to follow based off the abstract path are refined to correspond to maneuvers on the original search space. This technique shares similarities to the implementation of VRA* in terms of cost-lookups, but different in that moves are performed at the original level. The authors of [8] point out that path smoothing can be applied here to make a more aesthetically-pleasing path via line tests, a suggestion also made in this paper at the conclusion of VRA*'s test results section.

It is very interesting to note that HPA* only provides the first few steps an agent should take instead of providing the entire path like A* normally does. The reason given by the authors, and one that is a very valid point to make, is that since paths change very often in game pathfinding situations (whether it be to environmental change, player decision making, or anything else), it would be a waste to compute entire paths if they are only to be discarded and a new path needed as soon as just one second in the future. This could be implemented in VRA* very easily since at its heart, VRA* is just a series of A*

calls with customized cost evaluation functions on an abstract space.

HPA* Final Analysis

HPA* produces results within 1% of optimum, but at a few concessions. It has a set number of abstraction levels, it has pre-defined cell entry and exit points, and to achieve the most-optimal result, it requires post-processing to smooth paths. HPA* caches data other than a lookup table, like HA* does. Also similar to HA*, HPA* builds its abstraction from low granularity to high. VRA* does the opposite, and only increases resolution as and where necessary. HPA* made some very important improvements to abstracted state space searching, but is a bit rigid. VRA*, on the other hand, is not as rigid. It is important to note that we are not proposing VRA* as being superior to HPA*, but as a less-restrictive alternative to HPA* for abstracted searching in game environments.

Introduction to VRA* and Statement of the Problem

When pathfinding on a 2D-grid, there are many cases in which a large amount of the nodes in the search space are the same: traversable tiles with unit cost (this is not limited to just 2D-grids, but for the purposes of this paper, 2D-grids are the search space of focus). Though these tiles can all be thought of, more or less, as the same tile with only the heuristic differences influencing their node scores, they all are still processed individually in an A* search. A* is a fast and popular method for traversing these types of spaces, but there is still room for improvement.

It would be great to abstract as much of this uniform data as possible while still maintaining a realistic traversal of the search space. By reducing the size of the search

space, we can run A* faster by virtue of having less nodes to process. Achieving this desired result is the aim of this research. Search space state abstraction is not new: researchers have been working at it for years now. There are several different theories and methods, each with its own benefits and drawbacks. The theory proposed in this research, called Variable Resolution A*, is not perfect, but it takes away some of the better features of its predecessors and, for its intended search space, does a very good job of producing paths from start to finish on a variable-resolution grid. Testing shows that VRA*, in average, expands less nodes than a regular A* search would in one search at fixed resolution. It's very important to remember that all previously-discussed methods worked on a fixed resolution, even the hierarchical pathfinders most-recently presented. This is one of the great things about VRA*: its search space varies in resolution. There could be one cell that takes up half the graph and next to that cell, three other cells one-third of the larger cell's size…and the algorithm still runs with ease! Just how VRA* works will be detailed in the following pages.

VRA* Implementation Details

VRA*'s implementation is very straightforward; an algorithm overview will be presented first, followed by detailed explanations of components unique to VRA* afterwards. The search space that A* would normally search on is known as the highest resolution search space, and can be any size. Before running VRA*, a cost table is generated based off the connectivity of the graph at the highest resolution. By using connectivity data from the highest resolution, it is ensured that detecting obstacles at lower resolutions will be consistent between A* and VRA*. After generation of the cost

table, VRA* splits the search space into two nodes: one containing the origin point, and the other, the goal point. The area of these cells need not be identical; the important part is only that there are two cells.

Following these preprocessing steps, an A* search is performed on this two node search space. Connectivity between nodes is determined by a line of sight test by rasterizing a line at the highest resolution between current resolution points of interest. If starting from the origin, the line test starts from that point; otherwise, the line test starts from the current cell centroid. If aiming for the goal, the line test ends at that point; otherwise, the line test aims for the centroid of the target cell. The line rasterization test is based off Bresenham's line algorithm, the basis for rasterizing a line on a computer screen. A* is run until a path is found. If a path is not found, then nodes that failed the line test and were marked for splitting are then split. A* is then re-run on this brand new search space. Simply put, VRA* is a summation of A* searches on various low level search spaces. Splitting of select nodes in the search space would continue until either a path is found, or the highest resolution was reached. If no path can be found at the highest resolution, then no path exists.

Cost Table Generation / Analysis of Obstacles

Before any code is run related to VRA* or A*, obstacles must be placed on the grid, as well as the start and goal points. After obstacle placement is complete, the search space is processed into a two-dimensional array that serves as a cost table. Each index in the array corresponds to the (x,y) coordinates of a tile at the highest resolution. A tile is either traversable, or an obstacle tile. Traversing between open tiles carries unit cost, so all tiles that are open share this same cost. Tiles that have obstacles on them are given

infinity cost. This cost method is easily extensible to varying cost tiles since obstacles carry a much higher cost than any tile ever could, and this is by design. Since all the obstacle checks are done in this preprocessing step, computation time is saved during runtime because the algorithm need only index into the cost table to check for collisions instead of running the obstacle test several times.

Generation of Start and Goal Nodes

Start and goal nodes are generated by computing the midpoint between the start and goal points, and then comparing the x and y distances between the start and goal points to determine which axis to split on. If the x distance is greater, then the split line will be generated at the x coordinate of the midpoint, and likewise for the y coordinate. This creates our initial nodes in the search space. A* will be run on this using the line test to determine connectivity, as previously mentioned. In all likelihood, a path will not exist between the start and goal node unless the trivial case of no obstacles is being tested, or if by chance all obstacles are out of sight of the start and goal node. Connectivity is determined by the line test, whose description follows.

Line Rasterization Test

The line rasterization test follows the simple logic of the Bresenham line algorithm used very often in computer graphics to render a line on the screen pixel by pixel [4]. Instead of drawing a pixel at each point as the test increments from the starting point of the line test to the finish, it looks up the cost from the cost table at the current tile coordinate at the highest resolution. Highest resolution coordinates are used to make costs uniform amongst all search spaces, allowing us to compare cells of varying resolution. Think of the highest resolution cells as the pixels in this situation.

If the cost lookup at any point is infinite, this means we have encountered an obstacle, and the nodes being tested are not connected. The test returns failure, and the nodes are marked for splitting. If the cost is not infinite,then the test succeeds, and a path between the two nodes exists. The cost at each coordinate along the line is summed and used for the overall cost of traversal from the two points.

Splitting of Cells for Variable Resolution

The cell splitting theory was inspired by the work of Moore and Atkeson in their parti-game algorithm [5]. Their algorithm would start with the lowest-possible resolution of the search space, and increase resolution of cells when and where necessary, by splitting cells. These splits would occur around obstacles, or as they described, on the borders of winning and losing cells. Winning cells were cells that were traversable, and losing cells were cells of infinite cost. Only splits that were needed were performed, and the algorithm continued on it's way until the goal was reached.

VRA* puts this same logic to use, just a little bit differently: it only splits one cell. The choice to split just one cell was made as an optimization because two cell splits wasn't always necessary in parti-game, and is less necessary in VRA* because we are performing A* searches on each search space instead of just continuously navigating the same search space like parti-game does.

Cells marked for splitting by the line tests that occurred in the previous A* search are put into a list. If a path is not found, then before the next A* search is called, the first cell on the list is split. This cell is either a cell along an obstacle, or a cell containing an obstacle. The cell is split into two, along its longest axis, and the two new cells are added to the list of cells for the next A* search. The centroids of each new cell are computed

and stored so the line test has a target point to start from and aim for in its execution. Figure 6 depicts an example of a cell split before and after a VRA* search.

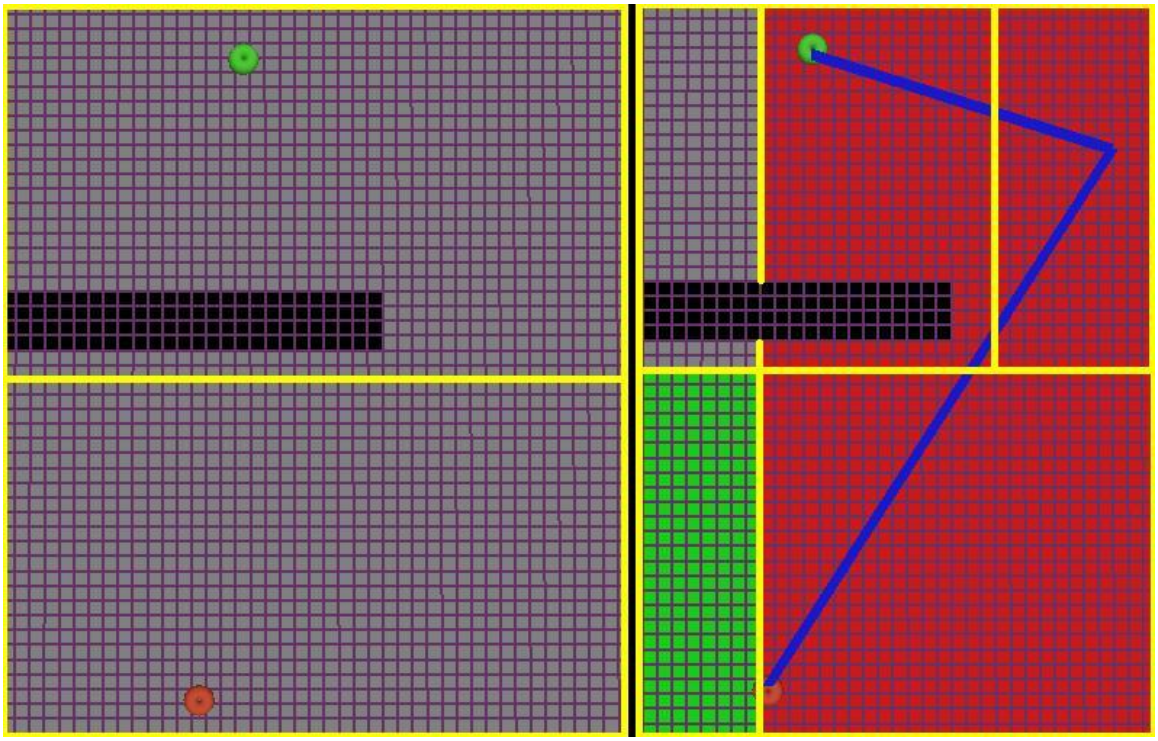

Figure 6: Cell splitting before (on the left) and after (on the right) VRA execution.*

On the left is the initial VRA* search space: a start point (in red), a goal point (in green), and an obstacle (in black). The start and goal have their own cells (outlined in yellow). Two cells are present to start. On the right, you see the end result of the VRA* search (picture is cropped). The cell borders are outlined in a thick yellow line for ease of interpretation. As explained above, the non-start and non-goal cells' centroids are used as the points of interest for the line test.

Finding Neighbors

In A* on a uniform grid, finding neighbors is simple. In 4D movement cases, just check the nodes above, below, to the left, and to the right of the current node. If they are reachable, then they are linked up to the parent node as neighbors. The same holds for the 8D case when incorporating diagonal neighbors. In VRA*, because nodes can be of varying sizes, one node could have just one neighbor to its left, or many. So the edges of nodes must be checked for neighbor status, and then the line test is called afterwards as the final check on neighbor connectivity. Edges are checked before neighbors as a way to avoid calling the line test more times than necessary. Both checks, though, perform very fast in the VRA* implementation designed for this paper.

Making it All Work

The final step in getting VRA* up and running is to hook all the proprietary VRA* functions into the normal functionality of an A* search. Normally in A* you evaluate the fitness (f) of a node by adding its cost (g) to its heuristic (h). The nodes with the better f-value are evaluated before the nodes with worse f-value, and an optimal path is found.

For VRA* execution, instead of having a predetermined g-values for each node, the g-values are calculated by the accumulated sum of highest resolution cells traversed during the line test from cells along the path. This ensures that regardless of the resolution of cells in the search space, a consistent and uniform cost analysis can be performed. Euclidean distance between centroids and the goal point are used as the h-value for the second part of the calculation of f-values.

If A* finds a path, then we are done. If no path is found, then VRA* calls the cell splitting function described above on the appropriate cell. A new search space is generated. A* is then run on this search space. The process continues until we do find a path, or in the worst-case scenario if no path can be found at the highest resolution (normal A* would also fail to find a path in this case).

VRA* Test Results

The following figures detail trial runs of VRA* versus A* on the same search space as a means of fair comparison. Memory used, A* open list expansions performed, and total time taken are tracked for analysis. The hypothesis is that VRA* will expand fewer nodes, useless memory, and in certain cases perform faster than A*. Objects in the following figures represent the same ones as in the previous figure.

Clear Search Space

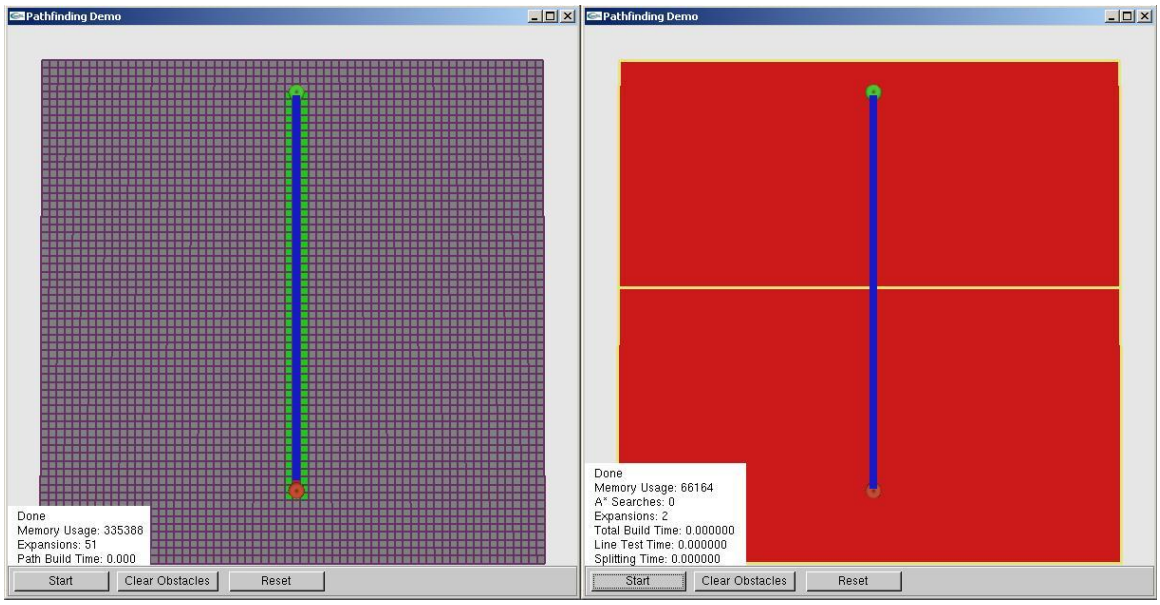

Figure 7: A (left) vs VRA* (right) in the trivial case.*

The first situation to examine is the initial search space, where just a start and goal exist. Here the savings in number of expansions will be obvious, since VRA* only runs on a two node search space.

Table 1. Statistics gathered from test in Figure 7.

Statistic	**A***	**VRA***
Memory Usage (bytes)	335,388	66,164
Expansions (nodes)	51	2
Path Build Time (seconds)	0.000000	0.000000

As you can glean from the figure and table, VRA* performs as hypothesized: it expanded a lot less nodes and used significantly less memory than A*. This is because,

since VRA* has a smaller search space, the A* open and closed lists are smaller and process faster, as intended. Time comparison in this situation is trivial since there are no obstacles to take into account. These results are encouraging, but the real test comes when obstacles are introduced.

A Simple Obstructed Search Space

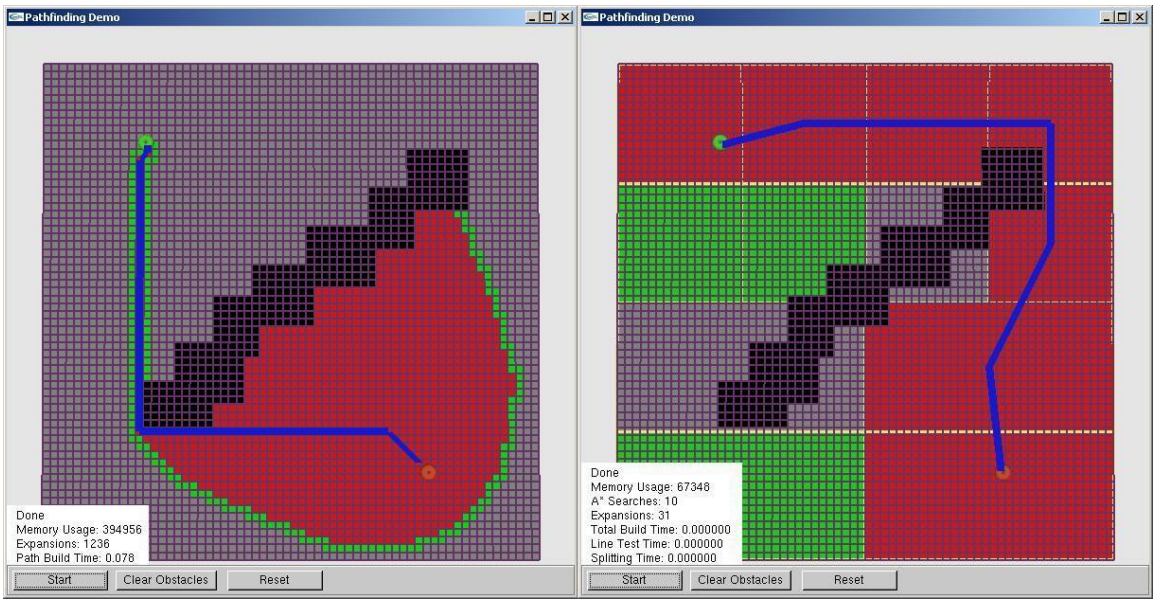

Figure 8: A simple obstacle impedes movement initially, but is no problem for VRA.*

Table 2. Statistics gathered from test in Figure 8.

Statistic	**A***	**VRA***
Memory Usage (bytes)	394,956	67,348
Expansions (nodes)	1236	31
Path Build Time (seconds)	0.078000	0.000160

In this case, a simple diagonal obstacle blocks line of sight from start to goal. This is not that difficult a case, but is something that is encountered regularly in grid worlds, and therefore has merit for our study. It also gives an easy-to-follow cell split visual for VRA*. The search space created for the final path in VRA* is only a dozen or so nodes. Like before, VRA* uses much less memory than A*. It also performs far less expansions. A* has to expand much more in this case than in the trivial case to get around obstacles. VRA* only has to process the obstacle a few times, and shortly after, a path is found. Once again VRA* ran very fast. A* took about eight-hundredths of a second to find a path.

Again, this is what was desired during the algorithm conception. Let's move on to a couple more cases where the obstacles aren't as simple.

An H-Shaped Obstacle Search Space

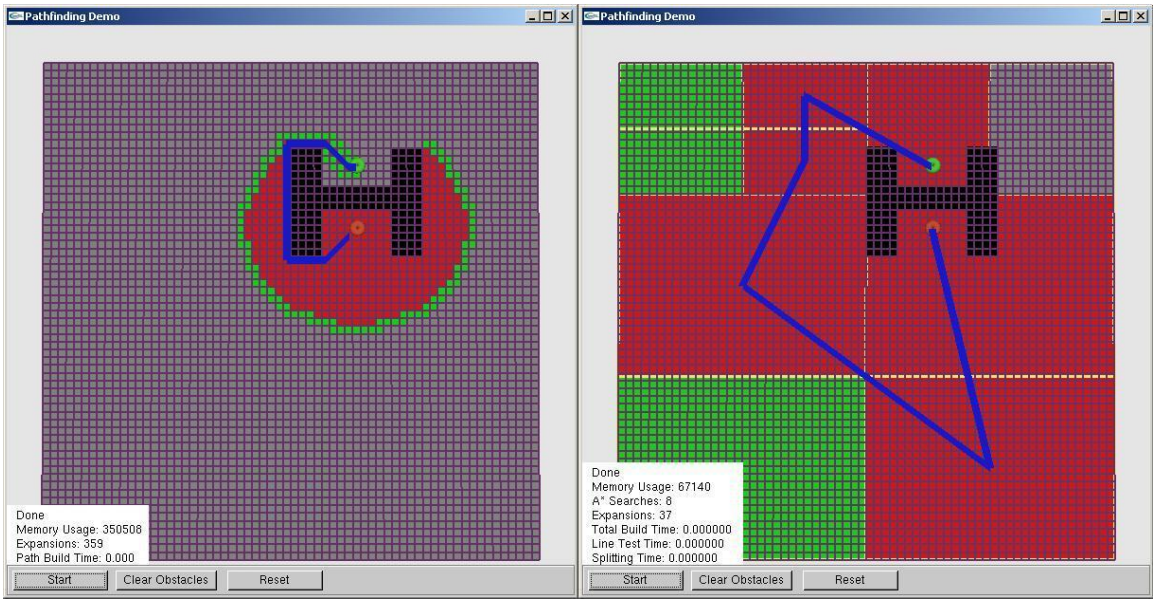

Figure 9: Here is ah H-shaped obstacle, which causes several cell splits in VRA.*

Table 3. Statistics gathered from test in Figure 9.

Statistic	**A***	**VRA***
Memory Usage (bytes)	350,508	67,140
Expansions (nodes)	359	37
Path Build Time (seconds)	0.000000	0.000000

This case is interesting because it provokes several cell splits from VRA* by virtue of the positioning of the walls and how they obstruct several line tests. It also gives A* a bit of a challenge initially, so it's a nice grounds for comparing the two algorithms. Again, VRA* uses far less memory and expands far fewer nodes. Like the previous test VRA* is blazingly-fast, but so is A*. Let's look at another letter-based case, something we refer to as the “C”-obstacle.

A C-Shaped Obstacle Search Space

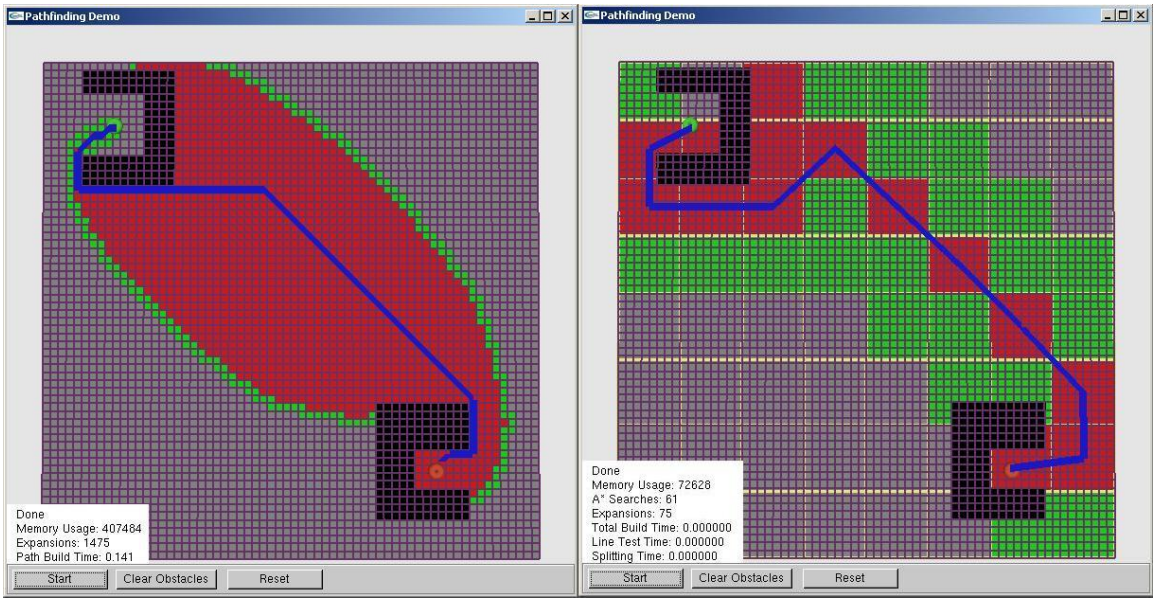

Figure 10: A search space that resembles a typical maze configuration.

Table 4. Statistics gathered from test in Figure 10.

Statistic	**A***	**VRA***
Memory Usage (bytes)	407,484	72,768
Expansions (nodes)	1,475	75
Path Build Time (seconds)	0.141000	0.000100

The "C"-shaped obstacle is one we found interesting in testing because it is designed, based off the knowledge of how VRA* selects nodes to split, to make VRA* perform poorly. One thing left unconsidered, though, is that it is no friend of A*, either. A* once again expands far more nodes than VRA*, fourteen-hundred more to be exact. A

dominant winning in time complexity is achieved by VRA*, as well as the memory consideration which is now becoming almost expected to favor VRA*.

Maze-style Search Space

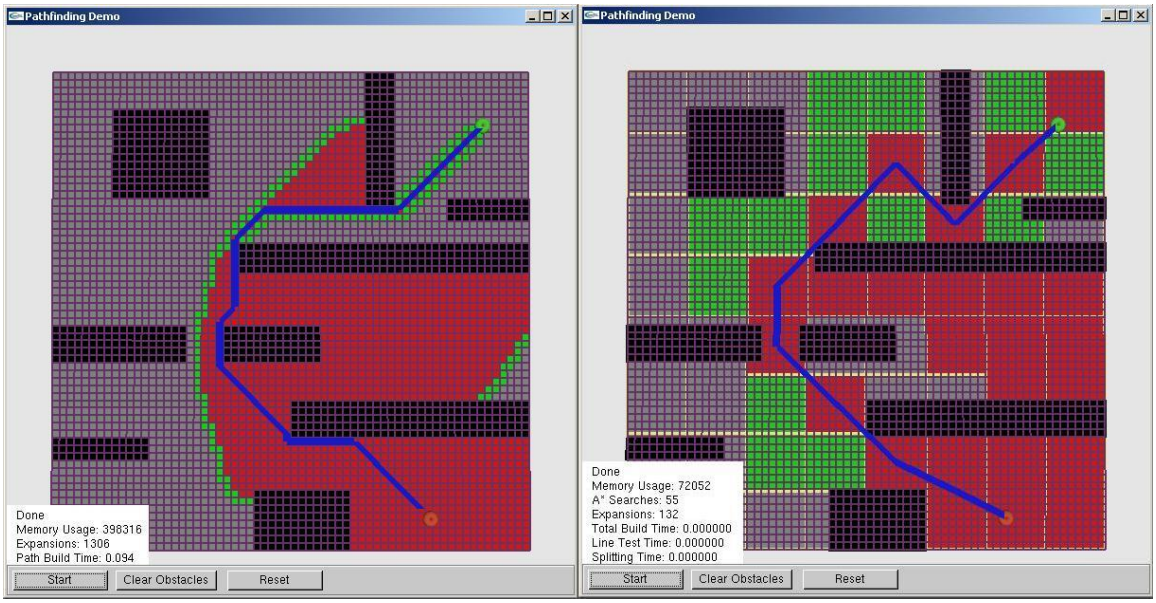

Figure 11: A search space that resembles a typical maze configuration.

Table 5. Statistics gathered from test in Figure 11.

Statistic	**A***	**VRA***
Memory Usage (bytes)	398,316	75,052
Expansions (nodes)	1306	132
Path Build Time (seconds)	0.094000	0.001000

This grid-world looks more like something you'd see in a real computer game, perhaps one of the first-person shooter variety, or a dungeon in a role-playing game. There are various passages separated by long-obstacles, easily viewed as a series of hallways and rooms, perhaps. Here we see several split-cells in the VRA* map, but in the

end, like the previous examples, VRA* is outperforming A* in medium-density maps by a very comfortable margin.

Half-Filled Search Space

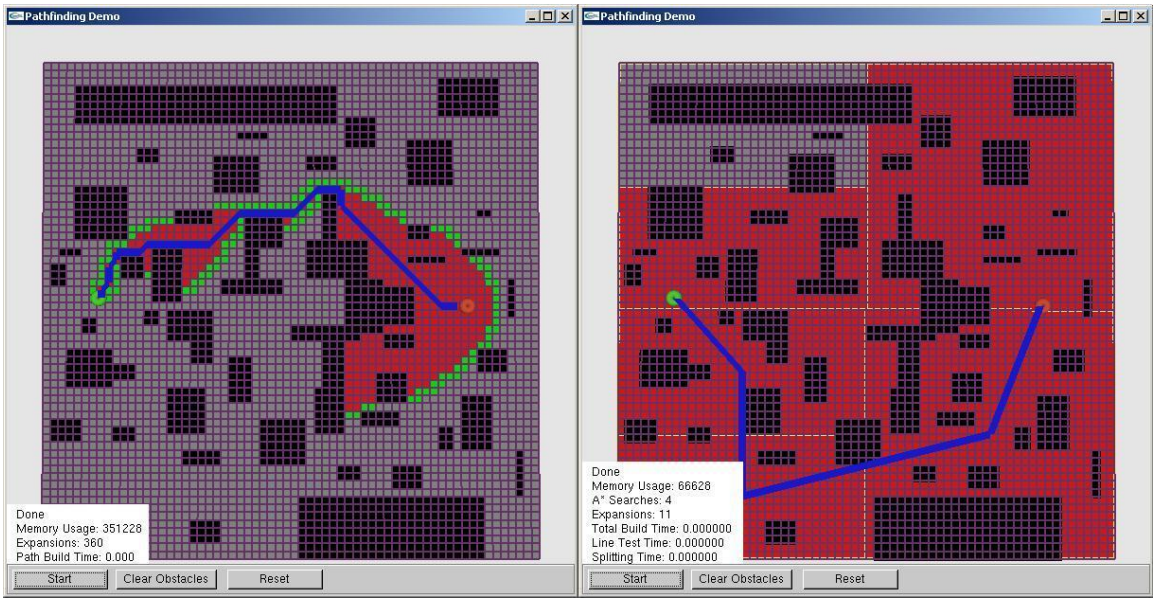

Figure 12: A search space that resembles a typical maze configuration.

Table 6. Statistics gathered from test in Figure 12.

Statistic	**A***	**VRA***
Memory Usage (bytes)	351,228	66,628
Expansions (nodes)	360	1
Path Build Time (seconds)	0.000000	0.000000

This case is a very-interesting one because it is the densest, and most sporadic search space so far. Remember, it was stated from the introduction of VRA* that it has its specific applications: 2D grid worlds that are at medium or lower density. With very little

order to the chaos in this map, it was unkown before testing how VRA* would handle this. VRA* could fail several line tests in a row consecutively, or it could find a way through the mess better than anyone could predict with the naked eye.

The results are as encouraging, as they've been throughout the test cases so far. VRA* uses far less memory and has staggeringly-less expansions than A*. Also, like before, VRA* and A* ran so quickly that a time to six decimal places could not be calculated. Since this was a surprising result, this test was run again by adding and removing a few obstacles from the search space in random areas. As predicted, A* was able to snake its way through significantly-better than VRA* with only a few modifications to the search space. In fact, the authors of [8] mention this as a reason why they shied away from line-of-sight testing. This is because, if you know the mechanics of VRA*, it is rather easy to set up a situation in which it is destined to fail. It is because of this that VRA* isn't recommended as an end-all, be-all pathfinding algorithm, but something to have in your development toolbox if you know that certain grid worlds in your game could benefit from it. Future work or optimizations exist to make VRA* applicable to all situations.

Conclusion

Results from the tests are consistent with the hypothesis that VRA* would use less memory, expand less nodes, and run quicker than A* in the 2D grids we tested both algorithms on. It should be noted that there are situations in which A* will outperform VRA*, since VRA* is, after all, a summation of A* searches with a few key modifications that enable it to run on variable resolutions. If the search space cells must

be split so many times that the variable search space ends up looking like the highest resolution grid, than results will be poor.

This, though, is not the intended application of VRA*. VRA* was designed with 2D grid worlds commonly found in computer and video games in mind, where obstacles are not completely random and trivial, but placed with strategic and artistic intent. For example, a theater of war with impassible mountains and rivers, or a shooting game taking place in a series of hallways and rooms that resemble a maze. These settings, when looked at from a bird's eye view, can be abstracted to the maze-like search space run in the final test. So, with these types of applications in mind, VRA* does its job well.

Future improvements in the VRA* algorithm are rubber-banding or smoothing of certain paths, like in the H-obstacle example, that look too jagged or backtrack farther than necessary. Also, optimization potential exists in the selection of which node to mark for splitting instead of marking several. As performed in HPA*, stopping execution short and providing just the initial movements an agent should make would also be a beneficial feature to add to VRA*. Yet another improvement could be support for dynamically-changing maps, which should not be hard to do at all seeing as insertion and deletion from the two-dimensional array that makes up VRA*'s cost table can be done in constant time. Finally, any speed upgrades on A* also will improve the speed of VRA*.

References

[1] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach.* 2nd Ed. Prentice Hall, 2003.

[2] Dalmau, Daniel Sanchez-Crespo. *Core Technologies and Algorithms in Game Programming.* 2004, New Riders Press.

[3] Rabin, Steve. *AI Game Programming Wisdom.* 2002, Charles River Media.

[4] Eberly, David H., *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufman Publishers Inc., 2001.

[5] Moore, A and Atkeson, C., "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces," *Machine Learning*, 21(3): pp. 199-233, 1995.

[6] Holte, R.C., M.B. Perez, R.M. Zimmer, and A.J. MacDonald (1996), " Hierarchical A*: Searching Abstraction Hierarchies Efficiently", *AAAI/IAAI*, Vol. 1: pp. 530-535.

[7] Holte, R.C., T.Mkadmi, R.M. Zimmer, and A.J. MacDonald (1996), "Speeding Up Problem-Solving by Abstraction: A Graph-Oriented Approach", *Artificial Intelligence on Empirical AI*, edited by Paul Cohen and Bruce Porter.

[8] Botea A., Müller M., and Schaeffer J. 2004. Near Optimal Hierarchical Path-Finding. In Journal of Game Development, volume 1, issue 1, 7-28.