1-1-2016

Enhancements to Hierarchical Pathfinding Algorithms

Xin Li
University of Denver

Enchancements to Hierarchical Pathfinding Algorithms

A Thesis

Presented to the Faculty of

The Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

in Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Xin Li

August 2016

Advisor: Dr. Nathan Sturtevant

Author: Xin Li

Title: Enchancements to Hierarchical Pathfinding Algorithms

Advisor: Dr. Nathan Sturtevant

Degree Date: August 2016

Abstract

In this thesis we study the problem of pathfinding in static grid-based maps. We apply the approach of abstraction and refinement. We abstract the grid map into a graph representation, and use the classic A* algorithm to search for a path in the abstract space, and then refine it into low-level path.

We started with a 2013 entry program to the Grid-based Path Planning Competition, and implemented several enhancements to experiment with the tradeoff between memory usage and search speed. Our program returns the refined low-level path incrementally, therefore reduces the first-move lag in large maps. We cache the low-level edge paths during runtime to avoid repeatedly refining the same abstract edge. In the precomputation step we calculate the low-level paths for all of the edges in the abstraction and directly access the data during online search. We also applied the weighted A* algorithm for online abstract pathfinding and show that the search speed can be further increased by sacrificing path optimality.

We ran our program with 132 maps and 1,739,340 queries. Results show that caching edge paths increases the search speed by a factor of 4.20 in comparison to returning the path incrementally but without caching. With precomputation, the search speed increases by a factor of 1.00 in comparison to caching edge paths. We show that online pathfinding speed can be increased by using more memory and/or offline storage.

Acknowledgements

I would like to give special thanks to my academic advisor Dr. Nathan Sturtevant, who is the most inspiring and coolest professor ever. I would like to thank my committee members, Dr. Rafael Fajardo and Dr. Scott Leutenegger, for their helpful feedback. I would also like to thank the following people: Team NP Completely Awesome—my TA friends: Amanda Kirk, Zach Azar, and Andy Brunner. Also Andrew Hannum and Will Mitchell. My friends Jingwei Chen, Hao Chen and Yipu Wang. The faculty and staff of the CS department. Susan Bolton is so awesome. My mom, for her unconditional love and support.

Contents

List of Figures

Chapter 1

Problem Statement

1.1 Introduction

Pathfinding is an exciting research field in Artificial Intelligence (AI) with a wide range of application domains, such as video games [Algfoor et al., 2015], GIS [De Smith, 2003], and self-driving cars [Dolgov et al., 2008]. In this thesis we study particularly the application of pathfinding in video games. The pathfinding problem is to calculate a valid path between two given locations as efficiently as possible in terms of path length and planning time.

In robotics, specifically robot motion planning, pathfinding algorithms are used, for instance, for a robot to find a valid path to move its arm from initial to goal configuration, and to avoid collisions with obstacles during its movement [Freund and Hoyer, 1988, Millán and Torras, 1992]. For ground-based robots, pathfinding is used to find optimal paths between start and goal locations so the robot can follow the path to reach its destination. The quality of the path can be determined by parameters such as shortest length, minimizing the number of turning or the obstacles encountered [Tiwari, 2012]. For biomorphic robots such as quadruped robots, pathfinding is used for the robot to find a valid path through the environment, and

to properly move its four legs in order to move forward along the path while respecting the local constraints between its footsteps [Delcomyn, 2007, Deits, 2014]. Pathfinding may need to be incremental if the robot does not have full knowledge of the entire environment, and must move to sense the environment.

In self-driving cars, pathfinding is a critical element for the car to navigate the city streets. The map of the city can be preloaded in the car, but other moving objects on the streets create a dynamic environment that requires the car to adjust its path online. When the car operates in an unknown environment, such as a parking lot, where obstacles are detected online, it builds the obstacle map incrementally while replanning the path [Dolgov et al., 2008].

In Geographical Information System (GIS), pathfinding tasks are performed on a variety of terrains. In park trail planning, a pathfinding program searches on the area's geographical and ecological data to determine the location, layout and length of the trails [Xiang, 1996, Tomczyk, 2011]. For search in freespace with nonuniform cost, pathfinding can be applied in domains such as infrastructures construction for pipelines [Moghaddam and Delavar, 2007] and power transmission lines [Ahmadi et al., 2008].

When humans perform a path planning task without the aid of a GPS device, it is natural to use some kind of abstraction and plan at a higher level first, and then refine the path into a detailed one. This approach is within human's computational power. Consider the problem of planning a road trip from Denver to New York City. As human drivers, we would first plan at a higher level, decide what states to drive through or pick the major cities we would like to visit on the trip. We could take the north route of I-80 passing Chicago on the way, or take the south route of I-70 and pass St. Louis. After the high-level decision is made, when reaching a city and driving through it, we then decide whether to take the highway for a shorter traveling time or to take the local, but possibly longer route, to see more of the

city. Detailed path planning decisions such as which particular lane to drive in will be made on the spot. A human driver does not need to consider such details until moments before having to take necessary actions, and he or she does not have the computational power or memory storage to do a turn-by-turn path planning ahead of time. This thesis studies the abstraction and refinement approach, which is akin to how humans do pathfinding tasks. This approach is used when pathfinding task is performed under tight memory usage and computational effort constraints.

Pathfinding is a key task in video games [Anguelov, 2011]. It is used for moving playable characters (PC) and nonplayable characters (NPC) around the world. The most basic requirement for a game AI agent is to successfully navigate the environment. For example, a PC gets a command from the human player to go from one room in the world to another room. The PC first plans a high-level path, decides which rooms it needs to pass through to get to the goal. When the PC enters each room, it will then plan a detailed path to cross the room, avoiding obstacles and minimizing the path length.

For a road trip, it is entirely unnecessary to plan out every particular road to drive in before setting out. In the context of video games, the concept is similar. It is also unnecessary to compute the complete low-level path before executing. We only need enough information for the agent to start moving in the right direction. Once the high-level path is found, it is guaranteed that a valid path leading to the goal exists. The abstract path will be continuously refined as the agent is following the low-level path. This makes the PCs respond more promptly to player's commands, and the NPCs act faster to the change of the game state. Sometimes the goal location changes during the execution of the current path because of various reasons such as the player wants to go to a different place or the map is changed, therefore it would be a waste of computational effort to calculate the entire low-level path. In the case of path planning in a dynamic environment, interleaving path planning and

path execution is a more suitable method for taking into consideration the moving objects in the task environment.

For NPCs in the games, since their movements can be scripted and they may not always be seen on the screen, the paths they follow that are off-screen do not need to be as high quality as PCs' paths. Depending on the design of the NPC, it can change its goal during execution of the current path it is on and/or change a section of the path in order to adjust to a dynamic environment. Both the PC and non-scripted NPC agents in the game follow the pattern of "format a pathfinding problem, search for a solution, execute the path" described in the classic AI textbook *Artificial Intelligence: A Modern Approach* [Russell et al., 1995].

Pathfinding in video games is a challenging problem because game environments are becoming increasingly larger and more complex, and the memory budget for commercial games is limited. Since a large portion of the allocated memory for pathfinding is used to store the maps of the game, building a memory-efficient pathfinding abstraction and reducing the space complexity of the algorithm are very important for a great gaming experience, which makes improving the pathfinding runtime performance an interesting research problem.

1.2 GPPC

The Grid-based Path Planning Competition (GPPC) [Sturtevant, 2014] started in 2012 and has run annually. The implementation of the DAO pathfinding program in this thesis was entered in the GPPC 2014. The GPPC aims to improve the understanding of grid-based pathfinding research by providing a platform for individuals and researchers to compare different approaches and the quality of their work [Sturtevant et al., 2015]. The GPPC provides a set of standard benchmark problems on a broad range of maps sizes, as well as standard implementations and metrics [Sturtevant, 2012]. The GPPC map set contains 132 maps taken from dif-

ferent resources including Dragon Age: Origins®, Warcraft III®, Starcraft®, and artificially generated maps.

1.3 Thesis Overview

In this thesis, we implemented several enhancements to the pathfinding program to improve its runtime performance. Many researchers have worked on these ideas, but their work have not been compared as part of the GPPC. The contributions of this thesis are:

1. Modifications are made on the pathfinding module in the 2013 DAO code [Rabin, 2014] and tested in the GPPC framework.

2. Added real-time performance to the 2013 DAO program. This is publicly available on GPPC for comparison with other researchers' work.

3. Implementation of caching edge paths, which stores the low-level paths of each edge of the abstract path in memory to speed up the runtime pathfinding performance.

4. Implementation of precomputating edge paths. The low-level edge paths are precomputed offline and load to memory for direct access during online search.

5. Implementation of the weighted A* feature [Pohl, 1970], using a biased heuristic function to reduce the search space but sacrificing the optimality of the path.

6. The enhanced pathfinding program was tested on a standard set of benchmark problems which are used in the GPPC. It successfully solves the entire collection of $1,739,340$ problems in the GPPC problem set.

The remainder of this thesis is organized as follows: Chapter 2 describes the problem definition of a pathfinding problem, and presents a selection of related work. Chapter 3 describes the abstraction mechanism of building a memory-efficient graph representation of a grid-based map. Chapter 4 presents our approaches to improve the runtime performance of the pathfinding program and describes each enhancement in detail. Chapter 5 presents and discusses the experimental results. Lastly, Chapter 6 presents our conclusions and work for future research.

Chapter 2

Problem Definition and Alternative Approaches

2.1 General Pathfinding Problem Definition

A search problem P can be generally defined as the tuple (S, A, c, s, g), where S denotes the set of states, A denotes the set of available actions, $c(v, a, v')$ denotes the cost of taking action $a \in A$ at state $v \in S$ to reach state $v' \in S$, s denotes the start state, and g denotes the goal state. The search space (S, A, c) can be represented as a graph $\mathscr{G} = (V, E)$, where the set of vertices V represents the set of the states S, and the set of edges $E = (v_i, v_j, c_k)$ represents the set of available actions and the associate cost c_k that maps from state v_i to v_j. The successor function of state v_i is defined as $succ(v_i) = \forall_{v \in V}(v_i, v) \in E$, which returns the set of states reachable from v_i within one action. A solution to a search problem P is an action sequence that leads from the start state to the goal state.

The above definition is generally used in literature, however, it does not model the search domain correctly. During runtime, the search domain $\mathscr{G} = (S, A, c)$ is not being given as input repeatedly for each problem instance. For static environments

the search domain does not change during runtime. Therefore, it is more efficient to input the search domain $\mathscr{G}$ only once, and acquiring the start and goal locations for each problem instance and solve it using the same $\mathscr{G}$. Since the general pathfinding problem definition does not correctly model the problem being studied in thesis, in the next section we give a more specific definition.

2.2 GPPC Pathfinding Problem Definition

The pathfinding problem in this thesis as well as in the GPPC, is a single-agent search in a static environment. We use an enhanced definition to allow further techniques that re-use pre-processed data and generate the path incrementally. We define the pathfinding problem P as the tuple $(\mathscr{G}, Q)$, where $\mathscr{G} = (V, E)$ is a graph and denotes the search domain, and Q is a set of start and goal locations and denotes the set of problem instances. $\mathscr{G}$ is given as offline input, therefore we can perform pre-processing on $\mathscr{G}$, store the pre-processed data on disk, and load it to memory during program execution. Q is given as an online input during runtime. For each problem instance, a path Π is found. Each path Π contains a set of sub-paths because Π can be returned incrementally.

The pathfinding task in the GPPC is: given $\mathscr{G}$, perform precomputation on $\mathscr{G}$, then compute solution to Q. The full details are in the next three subsections.

2.2.1 Search domain $\mathscr{G}$

In this thesis we choose an eight-connected grid-based representation for the search domain $\mathscr{G} = (V, E)$. Each grid cell is either traversable or blocked. The agent uses octile movement in the grid domain. Considering the agent is located in the center grid cell of Figure 2.1, the agent can move in eight compass directions. Cardinal moves are of directions {N, S, W, E} and cost 1, whereas diagonal moves are of directions {NW, NE, SW, SE} and cost $\sqrt{2}$. The set of vertices V represents

the unblocked locations v_i in the grid, whereas the set of edges E represents the valid octile movements.

We assume the agent is the same size as the grid, and the agent can occupy one traversable grid at a time. If the agent attempts a diagonal move from grid (x_1, y_1) to (x_2, y_2), then all four grids $(x_1, y_1), (x_2, y_2), (x_1, y_2), (x_2, y_1)$ must be traversable. In Figure 2.2, since grid (x_2, y_1) is blocked, the diagonal move from (x_1, y_1) to (x_2, y_2) is not available; the agent has to take two cardinal moves to go from (x_1, y_1) to (x_1, y_2), and then to (x_2, y_2).

$\mathscr{G}$ is given *a priori*, as offline input, thus allowing precomputation on the domain.

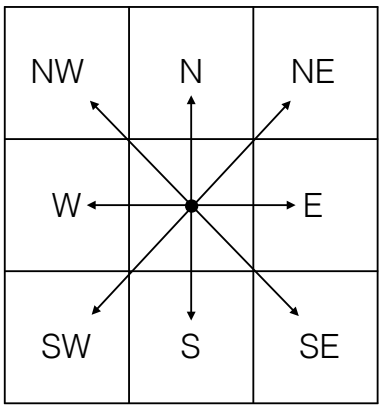

Figure 2.1: Octile movement.

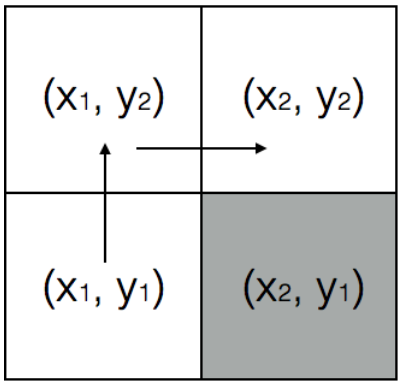

Figure 2.2: Diagonal move

2.2.2 Problem instances Q

The set of problem instances Q can be represented as $(q_0 = \{s_0, g_0\}, q_1 = \{s_1, g_1\} \dots$
$q_n = \{s_n, g_n\})$, where $\{s_i, g_i\}$ represents the start and goal locations for each problem instance. Q is given during runtime as an online input. Once the pathfinding task for q_i is completed, q_{i+1} is then given. The pathfinding computation for each problem instance $q \in Q$ cannot be shared.

2.2.3 Pathfinding task

The pathfinding task is: given the set of problem instances Q, for each $q_i \in Q$, return a valid path Π_i. The solution to a pathfinding problem $P = (\mathscr{G}, Q)$ is a set of paths $\Pi = \{\Pi_1, \Pi_2, \dots \Pi_n\}$. Each path $\Pi_i \in \Pi$ is the corresponding solution to problem instance $q_i \in Q$.

Π_i is composed of a sequence of sub-paths $\{\pi_{i0}, \pi_{i1}, \dots, \pi_{in}\}$, $n \geq 0$. Each sub-path is a sequence of nodes, where $\pi_{i0} = \{v_0 \dots v_i\}, \pi_{i1} = \{v_i \dots v_j\}, \dots \pi_{in} = \{v_k \dots v_n\}$, and $\{v_{i-1}, v_i\} \in E$. v_{i-1} and v_i are adjacent locations. v_0 and v_n correspond to the start and goal locations in $q_i = \{s_i, g_i\}$, hence, $v_0 = s_i, v_n = g_i$.

The DAO pathfinding program in this thesis computes the solution to a problem instance incrementally. Therefore, the complete path Π_i can be represented by a sequence of sub-paths $\{\pi_{i0}, \pi_{i1}, \dots, \pi_{in}\}$, where $\pi_{i0} = \{v_0 \dots v_i\}, \pi_{i1} = \{v_i \dots v_j\}, \dots,$ $\pi_{in} = \{v_k \dots v_n\}$.

2.3 Selected Related Work

This section gives a brief description of some selected grid-based pathfinding approaches that have been entered the GPPC in the past. A comparison of these approaches is given in Table 2.1.

Compressed Path Databases (CPD)

A straight-forward method to speed up online search is to precompute all pairs shortest paths (APSP) and store the data for use during runtime. We computed the size of the APSP for the DAO maps in the GPPC, and it takes 5069.6 GB. The GPPC server has 500 GB of storage. The large amount of storage it requires renders this method infeasible in practice.

Botea proposed the CPD method that precomputes APSP and compresses the first moves of each state [Botea, 2012]. The CPD method for the DAO maps requires 52 GB of storage.

CPD is based on the fact that states in a contiguous regions have the same first move to the goal. There are only eight options for the first move of each state because the authors used octile distance hence allowing movements in eight different directions. For example, in Figure 2.3, the first moves from the states N_1 to N_6 to the goal G are all to the east, but the distances to G are different. Therefore, the first moves are a good structure for compression but the distances to the goal are not. After precomputation, for each state we store compressed data of regions, where each region has the same first moves.

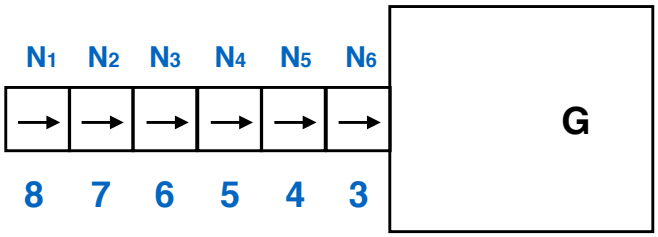

Figure 2.3: CPD first move.

Given a map, the CPD method first conducts pre-computation. It builds a first-move table for each node $n \in V$, which contains the direction of the first move to n from states $s \in V$ for all nodes $s \neq n$.

The next step is to decompose the first-move table into a series of rectangles, where each rectangle contains the same move labels for n. Building the compressed database can be done in parallel because the iterations for each node are independent. The memory complexity for the pre-computation's worst case performance given for n states is $O(n^2)$. The worst case computation of the single source shortest path data requires n node expansions. Hence, for n states in the state space, the worst case complexity of calculating the APSP data is $O(n^2)$ node expansions.

At runtime, the program looks up optimal moves from the database until a complete path is found. This approach is optimal and has very low first move lag, but it requires a significant amount of preprocessing time (81,408 minutes) and memory (52 GB). In the GPPC it solved $1,689,740$ out of $1,739,340$ problems.

Contraction Hierarchy (CH)

Geisberger *et al.* proposed the CH method for road networks [Geisberger et al., 2008]. It has also been adapted to run on grid-based game maps [Sturtevant and Geisberger, 2010, Storandt, 2013].

The basic idea of a graph contraction is to remove nodes and add auxiliary edges (when needed), called shortcut, to preserve the shortest paths in the remaining graph. For example, in Figure 2.4, the cost of path $v \rightarrow w$ is 3. If node u is contracted, then the cost from v to w becomes ∞. We add a shortcut edge (v, w) of cost 3, thus the shortest path from v to w is preserved. If we can find another path $v \rightarrow w$ with cost ≤ 3, then a shortcut edge is not needed. Removing a node from the graph must not change the shortest paths of the remaining nodes. The CH chooses a node order, where the nodes are sorted according to an importance

value, and then get contracted one by one in increasing order of importance. The graph is contracted fully to a single node, or partially when the contraction stops at a particular hierarchy level.

To solve a pathfinding query, we perform a bidirectional search using Dijkstra's algorithm in the CH. We search the CH in increasing order of the node importance value, that is, only expanding a node when it is more important than its predecessor. The search eventually stops at the most important node in the shortest path.

To get the path in the original graph, we recursively unpack the shortcut edges. If an edge is a shortcut, we replace it with the two edges that form the shortcut. If the unpacked edges contain additional shortcut(s), we keep on unpacking the shortcut edge(s), until all the edges are from the original graph. In Figure 2.4, if a path contains the shortcut edge (v, w), then (v, w) will be replaced by (v, u) and (u, w), and we get the path from v to w in the original graph $v \to u \to w$.

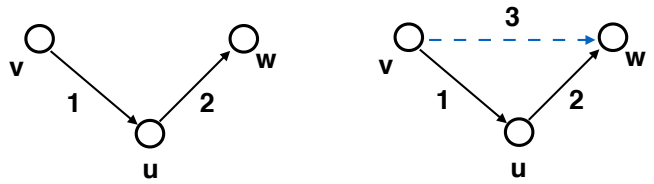

Figure 2.4: Add a shortcut from v to w to remove node u.

In the GPPC entry, the CH method requires 2.4 GB of storage and 968.8 minutes of pre-computation time. It guarantees optimal paths, and it solves all of the problem sets in the GPPC.

Jump Point Search (JPS & JPS+)

Harabor *et al.* introduced the JPS method for pathfinding on undirected uniform-cost grid maps [Harabor et al., 2011]. JPS follows a canonical ordering—it takes

diagonal moves first, then cardinal moves. Canonical orderings are effective in reducing path symmetries, but obstacles may prevent the natural canonical path. Thus, jump points can be placed around obstacles to partially reset the canonical ordering to allow the search to move around obstacles. JPS identifies jump points from the grid and only adds jump points and the goal to the open list, which reduces the number of nodes being expanded. JPS is fast, requires no preprocessing and no additional memory overhead. Harabor later modified JPS by adding an offline preprocessing technique that identifies jump points *a priori* [Harabor et al., 2014] . We refer to this approach as JPS+.

Figure 2.5 shows an example of JPS. A total number of 11 jump points, represented by black dots, are identified on the map. From the start location S, the search follows canonical ordering and keeps moving until it reaches a jump point or an obstacle. The jump point will be added to the open list. But if we reach an obstacle, then search in this direction stops and returns no results.

JPS and JPS+ return optimal paths. The JPS+ entry in the GPPC solves all problems sets using 13,449.1 seconds. It requires 947 MB storage and 1 minute of precomputation time.

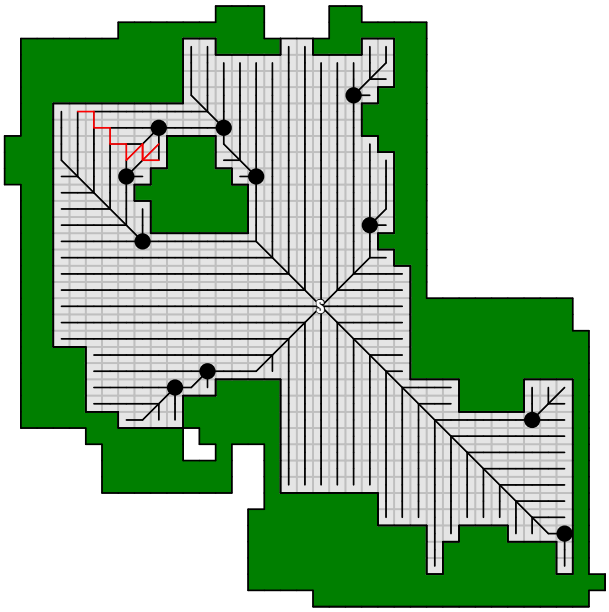

Figure 2.5: A JPS example showing the start location and the jump points. (Figure courtesy of Nathan Sturtevant.)

Table 2.1: Summary of approaches

Approach	Total Time (s)	avg_time (ms) per path	Optimal	Storage	Pre-cmpt (total min)
CPD	1348.5	0.798	Yes	52 GB	81,408
CH	630.4	0.362	Yes	2.4 GB	968.8
JPS+	13449.1	7.732	Yes	947 MB	1.0

Chapter 3

Background: Abstraction and Refinement

In this chapter we describe the approach of abstraction and refinement [Holte et al., 1996, Sturtevant, 2007] in pathfinding problems. The objective of this approach is to speed up the pathfinding process by giving up optimality. In video games, either optimal paths or shortest paths are often not a requirement as long as the paths look reasonable. Reducing the pathfinding time improves the gaming experience more than finding the optimal paths. Sometimes the shortest paths look unreasonable because they are too close to obstacles, as shown in Figure 3.1. A suboptimal path may look more natural and it is easier for the character to avoid colliding with obstacles.

In section 3.1, we first describe how to abstract a original map into a graph representation by dividing it into sectors and regions. Then we describe how to store the abstraction compactly in memory. In section 3.2, we show how to find a path in the abstraction and then refine it to a usable path.

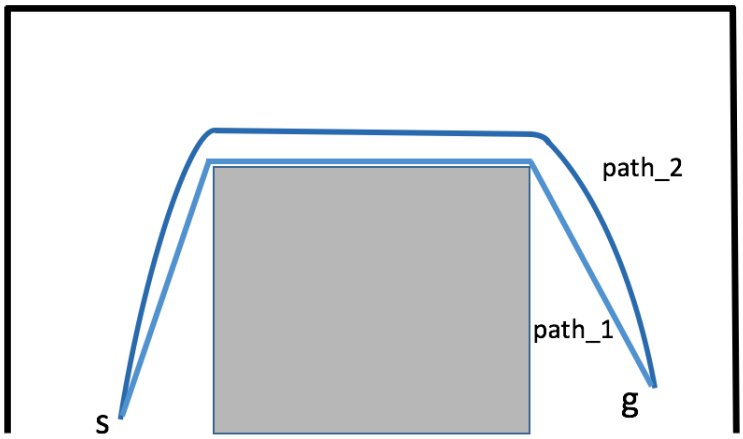

Figure 3.1: The shortest path compared to a more reasonable path. path_2 looks more natural and it is easier for the character to avoid colliding with obstacles.

3.1 Building and Storing the Map Abstraction

3.1.1 Pre-computation

Compute sectors and regions

The map sets used in this thesis are gridworld domains from Dragon Age: Origins™ by BioWare Corp®. Since the original map is usually high-resolution and takes up majority of the limited memory budget for pathfinding, the first task is to abstract the original map into a memory-efficient graph representation, using the map abstraction method described by Sturtevant [Sturtevant, 2007, Sturtevant and Jansen, 2007], which we reproduce here. The abstraction hierarchy can be extended to any number of levels, in this thesis the DAO pathfinding approach only abstracts a single level. The world is static, therefore we can perform pre-computation on the maps. The task of building and storing the map abstraction is done during the pre-computation stage.

The original map of the world is referred to as the low-level map. Locations on the low-level map are represented as x/y coordinates. We place a lower-resolution grid over the original map. This divides the original map into fixed-size sectors. The sector index can be calculated from the map width and sector size. Given any x/y coordinate, the sector this grid is in can be calculated by:

$$\text{sector index} = \left\lfloor \frac{x}{s} \right\rfloor + \left\lceil \frac{w}{s} \right\rceil \times \left\lfloor \frac{y}{s} \right\rfloor \tag{3.1.1}$$

where s denotes the sector size, and w denotes the map width.

The low-level map is divided into sectors without regard of its topology. Each sector is further divided into region(s), so that every location inside each region is strongly connected. An agent can travel to any location within a region without leaving the sector.

A region is represented as a node in the abstract graph, referred to as the region center. A region center corresponds to a x/y location on the low-level map, and can be chosen arbitrarily within the region. We initially choose the node closest to the weighted average of the region to be the region center. Every x/y location within one region on the low-level map is represented in the abstraction by the region center. The relationship of the x/y locations on the low-level map and a region center in the abstract graph is surjective, that is, a many-to-one relationship. For every node v in the abstract graph, there is one or more x/y coordinates in the low-level map. Hence, multiple x/y coordinates can correspond to the same v. To determine regions, we perform a breadth first search (BFS) from the region center to label all the grids in that region, if they have not already been labelled.

The map shown in Figure 3.2 is of size 32×32 and divided into four 16×16 sectors. The black grids represent blocked locations. The sector indices $0 \ldots 3$ are shown at the four corners of the map. Regions within each sector are referred to

as *sector* : *region*. For example, sector 0 has two regions: region 0:a and 0:b, and sector 1 has three regions, region 1:a, 1:b, and 1:c.

After the low-level map is partitioned into sectors and regions, the next step is to determine the edges between adjacent sectors. We iterate through the shared borders of the sectors and compare the regions on both sides of the border. For each pair of regions that share a sector border, an edge connecting these two adjacent regions is added to the abstract graph G. The cost of an edge is determined based on its length, which is the octile distance between the two region centers of this edge.

In Figure 3.3, four edges are added for region 3:a and its adjacent regions in neighboring sectors: 0:b, 1:b, 1:c and 2:a. The original 32×32 map is abstracted into a graph with 7 nodes and 10 edges, as shown in Figure 3.4.

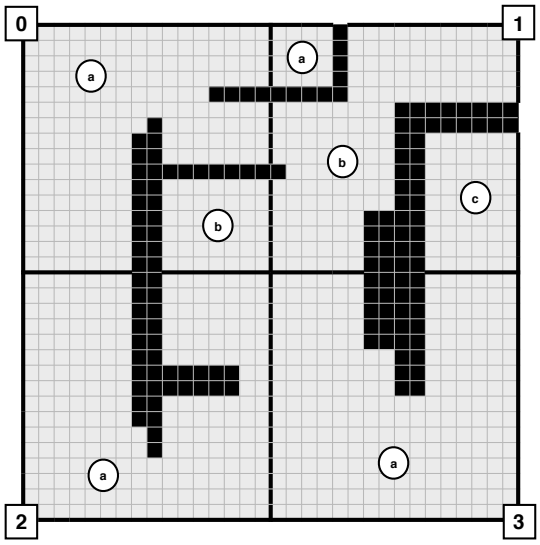

Figure 3.2: Computing sectors.

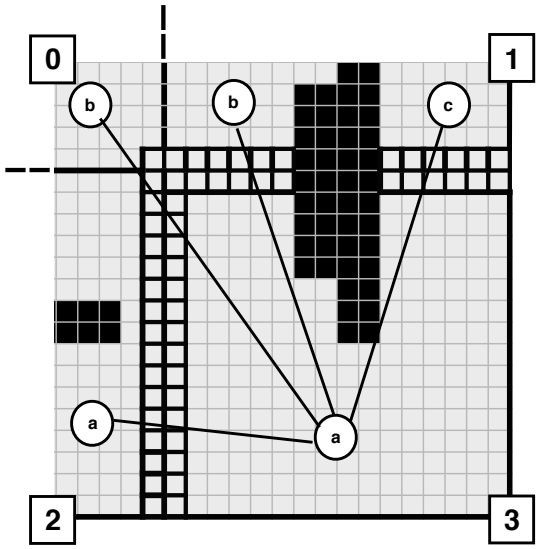

Figure 3.3: Computing edges.

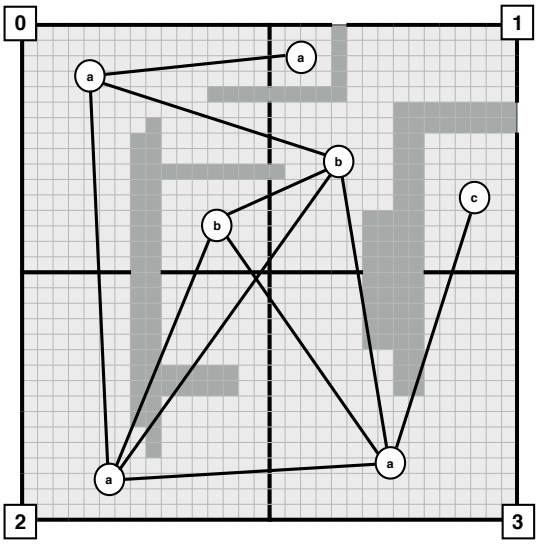

Figure 3.4: Full abstract graph.

3.1.2 Storing the map abstraction in memory

The memory needed to store the map abstraction is divided into two parts: a fixed-sized portion (32 bits) to store the sector data, and a variable-sized portion to store the region and edge data. The size of the sector data is determined by the size of the original map being abstracted, whereas the size of the region and edge data is based on the complexity of the original map.

In the 32-bit sector data, we store the number of regions in this sector (8 bits) and the memory address of where each region is stored in the second portion (16 bits). The rest of the 8 bits are not used. In the DAO program, we choose the sector size to be 16×16 because any location in the sector can be represented by one byte.

Figure 3.5 shows the sector data for sector 0 in Figure 3.2. There are two regions in sector 0, and memory address of the first region, region 0:a, starts at 0 because this is the first region we are storing.

Figure 3.6 shows the region data of region 0:a and 0:b. For each region we store the location of the region center (8 bits) and an index into its edge data in memory (8 bits). The region center is stored as the offset from the top left corner of the sector. In Figure 3.6, region center 0:a is offset by 5 rows and 5 columns from the top left corner of the sector, which is 85 cells. The edge index is the index of the first outgoing edge of the next region, and it also represents the cumulative number of edges seen in the abstraction so far. In Figure 3.6, region 0:a has 3 edges, which are stored in edge data offset 0, 1 and 2. Thus, the first outgoing edge of the next region (region 0:b) starts at index 3. Region 0:b has 3 outgoing edges, therefore edge index into the next region is 6 ($3 + 3 = 6$).

The edge data is variable-sized. We use 8 bits to store each edge: 3 bits for the edge direction, and 5 bits for the region that is connected by the edge. The start sector and region of the edge is implicitly known by the sector and region data. The target sector can be computed from the direction.

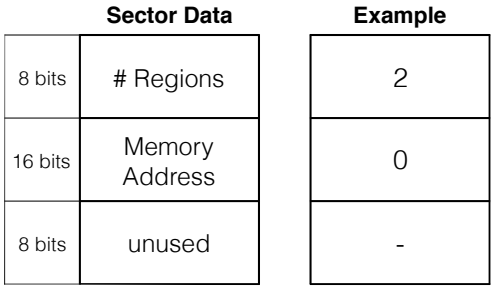

Figure 3.5: Sector data.

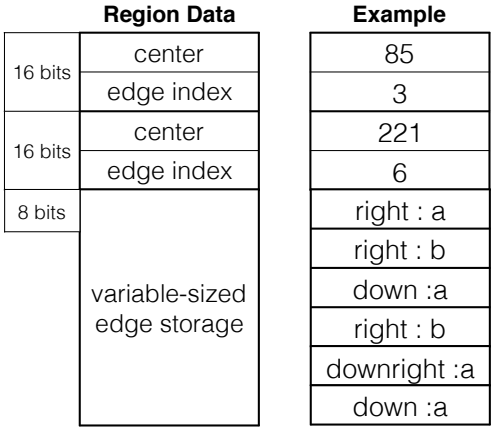

Figure 3.6: Region data.

There are a total number of 132 maps in the DAO map set, and they take 6.19 MB of memory. The average size of a DAO map is 46.87 KB. We compute the size of the DAO maps based on using one bit per grid cell, which is passable or blocked. In practice, a game may use 16 or 32 bits per grid cell. There may be other meta-data associated with each grid cell that needs to be stored with the map, thus a grid usually needs more than one bit, depending on the game design. The meta-data contains information such as area sound effects or character's personal space, etc.

After building the abstraction, the total memory required to store the DAO map set abstraction is 3.5 MB. The average size of a DAO map abstraction is 26.52 KB.

3.2 Pathfinding Using the Abstraction

There are three steps to perform a pathfinding task using the abstraction. Given start and goal locations, which are represented as x/y coordinates in the low-level map, we first find the corresponding sector and region in the abstract graph, as described in subsection 3.2.1. The second step (3.2.2) is to find a path in the abstract space using the A* algorithm [Hart et al., 1968]. The third step (3.2.3) is to refine the abstract path into a complete low-level path. The solution is a series of adjacent locations on the low-level map representing a path from the start to goal location.

3.2.1 Finding the start and goal locations in the abstract space

The first step is to convert x/y coordinates into corresponding nodes in the abstract graph, denoted by $sector : region$.

We use equation 3.1.1 to calculate the sector index. To determine which region a given location x/y is in, there are two cases. If there is only one region in the sector, then we must be in that region (region a). If there is more than one region in the sector, then we perform a BFS on the low-level map from the given location x/y until we find a region center to identify which region we are in. The breadth first search does not leave the current sector, therefore it is limited and fast.

3.2.2 Finding a path in the abstract space

From the previous step we have the start and goal in the abstract graph, the pathfinding task in this step is to find an abstract path to connect the start and goal by performing an A* search.

The map abstraction is a graph representation $G = (V, E)$, where the nodes V are region centers which are represented as $sector : region$. From each abstract node there are one or more outgoing edges that connect to another $sector : region$. The edge cost is based on the octile distance between region centers.

We use A* to find a path in the abstract graph, and the heuristic is octile distance. The abstract path is represented by a a series of nodes $(sector : region)$.

3.2.3 Path refinement

During the refinement step, we take as input the abstract path which is a series of abstract nodes $(sector : region)$, and compute the low-level paths edge by edge between the region centers. The low-level path is a series of x/y coordinates.

The most straight-forward way to refine an abstract path is to first find the start location in the low-level map and compute a path from the start to the first region center. Then we keep computing paths to successive region centers in the abstract path. Lastly, we compute from the last region center to the goal location.

In practice this method may cause a situation shown in Figure 3.7, where the start and goal locations are in adjacent sectors but very close to each other. If we plan from the start location to the first region center, and from the last region center to the goal, we will have a much longer path. The path in Figure 3.7 is $s \rightarrow (0 : a) \rightarrow (1 : a) \rightarrow g$, whereas the optimal path is just a straight line $s \rightarrow g$.

To fix this problem, we can plan the low-level path from the start location directly to the second region center, and then compute one edge path at a time to the second to last region center, from where we plan to the goal location.

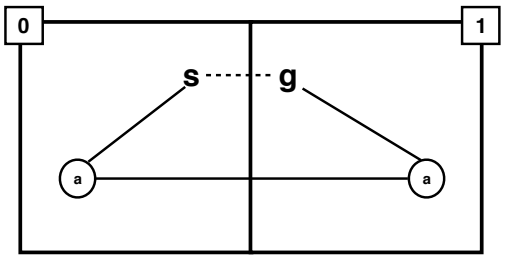

Figure 3.7: Skipping the first and last region centers.

An alternative method that is not implemented in this thesis but worth mentioning is to dynamically adjust the region centers during runtime. When solving a new problem instance, we temporarily move the first region center of/on the abstract path to the start location, and plan from there. Likewise, we move the last region center to the goal location.

Chapter 4

Abstraction and Refinement Enhancements

In this chapter we first describe the Application Programming Interface (API) of the GPPC. As an entry in the 2015 GPPC, the DAO program with our modifications follows the same API specifications.

We implemented additional enhancements to the DAO program to improve the pathfinding performance. The implemented features include: 1) returning the low-level path incrementally, 2) caching paths during runtime for repeated use, 3) precomputing the low-level path for the abstract edges, and 4) using a weighted A* algorithm. We describe each feature in sections 4.2 to 4.5.

4.1 The API

4.1.1 The API of the GPPC

The GPPC API has the following function calls.

```
void PreprocessMap(std::vector &bits, int width, int height,
const char *filename);
```

```
void *PrepareForSearch(std::vector &bits, int width, int height,
const char *filename);
bool GetPath(void *data, xyLoc s, xyLoc g, std::vector &path);
const char *GetName();
```

In `PreprocessMap`, the function can perform preprocessing on the map and write any necessary data to the file with name provided. The GPPC does not allow reading or writing from any other file.

In `PrepareForSearch`, the function is provided with the map, and can load the data from the file generated in the preprocessing phase.

`GetPath` does the pathfinding computation and returns true when the entire path is found.

`GetName` returns the name of program. The DAO program is referred to as "DAO-1-level" because it abstracts the map one level.

The constraints of the GPPC are:

1. The map size is at most 2048×2048.
2. The maps are static.
3. The agent uses octile movement.

4.1.2 The API of the DAO program

The DAO program uses the GPPC API. The map is static, therefore we can perform precomputation.

The `PreprocessMap` function is passed a pointer to the low-level map. It builds the sector and region abstraction, writes the sector and region data to a file, and does additional precomputation work according to the program configuration. We will describe the precomputation implementation in section 4.4.

The `PrepareForSearch` function loads the file and reads the sector and region data into memory. If the precomputed edge path feature is enabled, then it also reads the precomputed edge path data into memory.

In `GetPath`, the function is given the pathfinding query which is the start and goal locations in x/y coordinates. It maps the x/y coordinates to the abstract graph, calculates an abstract path, and then refine it into a low-level path.

4.2 Incremental Pathfinding Program

We started with a sample code from *AI Game Programming Wisdom 4* [Rabin, 2014], which was an entry to the 2013 GPPC. The 2013 DAO program returned the low-level path to the GetPath function call after the entire path is computed. We modified it to return the low-level path incrementally as each edge segment of the abstract path is being computed.

The benefit of returning the low-level path incrementally is that the agent does not have to wait until the entire path refinement is finished to start moving. This reduces first-move lag and makes the agent more responsive to player's commands, thus allowing the agent to handle larger maps. Sometimes, before the agent reaches the goal, the player may change the desired destination causing the agent to change course half-way, giving up the previous pathfinding query. Therefore, it is not necessary and may also be wasteful to compute the entire low-level path before starting execution, given the possibility that the pathfinding query may be altered before the search finishes.

Another benefit of returning the low-level path incrementally is the ability to adjust for dynamic environments. The abstract edge is being refined when the agent is close to the area, so that changes in that region of the low-level map can be taken into account.

The new DAO program is an entry in the 2015 GPPC and is referred to as "DAO-1-level". The pseudocode of the 2013 DAO program GetPath-Complete and "DAO-1-level" GetPath-Incremental are listed below.

GetPath-Complete(P)

```
1  // Compute abstract path for query P and return the result
2  AbstractPath = GetAbstractPath(P)
3  // Refine the abstract path to real path
4  while (AbstractPath.nextSegment != emtpy)
5      // Append the next segment of the refined path to RealPath
6      RealPath += GetRealPath(AbstractPath.segment)
7      AbstractPath.segment = AbstractPath.nextSegment
8      return RealPath
```

The abstract path is the input for the path refinement step. The output is a low-level path which is a series of x/y coordinates. In GetPath-Complete, `GetRealPath` (line 6) repeatedly takes the next edge in the abstract path, refines it into a low-level path, append it to `RealPath`. The iteration of the `while` loop terminates when the complete low-level path is computed.

```
GetPath-Incremental(P)
1  if P ≠ currentProblem
2      // Update current problem
3      currentProblem = P
4      // Compute abstract path for current problem and store the result
5      AbstractPath = GetAbstractPath(currentProblem)
6  // Refine next segment of the abstract path
7  RealPath = GetRealPath(AbstractPath.segment)
8  AbstractPath.segment = AbstractPath.nextSegment
9  return RealPath
```

In GetPath-Incremental, `GetRealPath` (line 7) takes an abstract edge as the argument, computes the low-level path for this edge and returns it. The agent can start moving along this path immediately. GetPath then refines the next edge and returns the output, until the complete low-level path is computed.

The game engine calls GetPath-Incremental repeatedly with pathfinding queries. For each query, GetPath-Incremental returns true when a complete path is found. When GetPath-Incremental is called with a pathfinding query P, it checks if the query is a new one. If a new query occurs, GetPath-Incremental will find an abstract path, and then refine the first segment of the abstract path into a low-level path and return it; if the query is the same as in the previous iteration, then GetPath-Incremental refines the next segment of the abstract path and returns it.

The DAO program with incremental path refinement solves the problem set in the GPPC in a total time of 27,451.9 seconds. It has an average suboptimality of 1.1285. It is faster than JPS by a factor of 3.96, but slower than JPS+ by a factor of 2.04, CPD by a factor of 19.36 and CH by a factor of 43.55.

4.3 Caching Edge Paths

4.3.1 Motivation

In the abstract graph, the edges connect regions in adjacent sectors. During the online search, some edges may appear very often in the abstract paths because the regions they connect in the map are traveled frequently. During the path refinement step, these frequently traveled edges are refined into low-level paths in x/y coordinates each time they appear in an abstract path. In the GetPath-Incremental function, `GetRealPath(AbstractPath.segment)` is called with the same abstract path segment multiple times.

Since the world is static, once we have computed the low-level path for an edge, we have the potential to reuse it to save some online search effort. For example, in Figure 4.1, the abstract path $\pi_1 : s_1 \rightarrow g_1$ consists of edges e_1 to e_5. e_1 connects s to the next region center $(1 : a)$, and e_5 connects $(4 : a)$ to g. These two edges cannot be reused because they are specific to this pathfinding query. e_2, e_3 and e_4 can be reused because they are edges connecting region centers. In Figure 4.1, e_2 and e_3 appear in $\pi_2 : s_2 \rightarrow g_2$. If we do not cache the previously calculated low-level paths of e_2 and e_3, they will be refined again in π_2, which is a waste of computational power. When we can cache the low-level path of the edges during runtime in the DAO program, we save online search time by not repeatedly computing the low-level paths at the cost of using more memory.

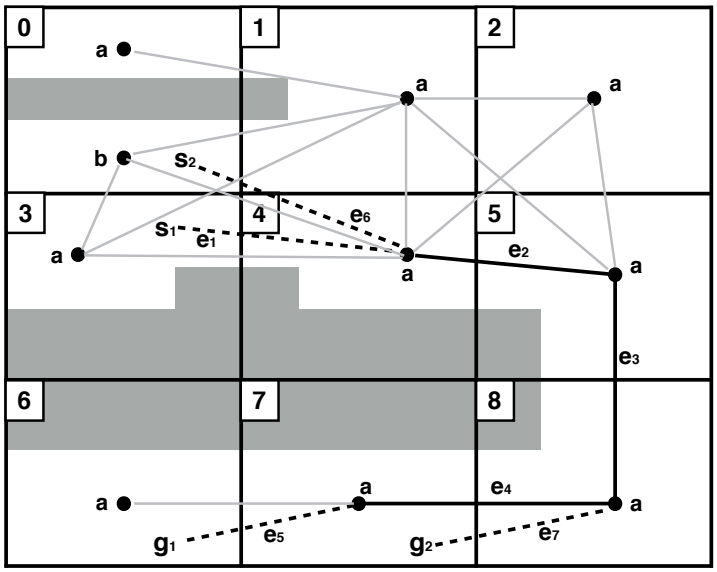

Figure 4.1: Caching edges.

4.3.2 Implementation details

Recall from Chapter 2 that the low-level path of an abstract edge $e_i = (v_{i-1}, v_i)$ is denoted as $\pi_i = \{(x_{i1}, y_{i1}), \cdots, (x_{in}, y_{in})\}$, where (x_{i1}, y_{i1}) and (x_{in}, y_{in}) are the two ends of e_i in the low-level map and correspond to the abstract node v_{i-1} and v_i in the abstract graph, respectively. The complete path for a pathfinding query is $\Pi = \{\pi_0, \pi_1, \cdots, \pi_n\}$. For each subpath π_i, the first point in π_i is the last point of π_{i-1}, and the last point of π_i is the first point of π_{i+1}.

To cache the low-level paths we store $\{\pi_0, \pi_1, \cdots, \pi_n\}$ in a hash map. The key used to index the item in the hash map is the pair of region centers that are the two ends of the edge, and the value is the low-level edge path π_i. During the refinement step, for each edge in the abstract path, we check the hash map in an attempt to find this edge. If found, we return π_i; if not, we compute π_i and store it in the hash

map. The access time of a hash map is $O(1)$. Therefore, using a hash map to store and retrieve the low-level edge paths is fast and effective.

Since the agent can travel the edge path from both directions, we can store the edge paths with regard to symmetry. We use two 32-bit unsigned integers to store the two region centers that the edge connects to. These two integers are used as the key to the hash map in their canonical order. We compare the two integers and store the edge path π_i from the region center that is the smaller integer to the region center that is the larger integer. When retrieving the edge path, we first check the direction of the abstract path, if the start and goal are in the same order as the key in the hash map, then return π_i directly; if not, then we return π_i reversely. When taking into consideration of the edge path symmetry, we use slightly less memory (0.16 MB) during runtime.

The DAO program with caching edge paths feature solves the GPPC problem set in 6640.3 seconds without considering symmetry, and the total time decreases to 6616.1 seconds when considering symmetry. The program performance with caching edge paths implemented is listed in column 4 and 5 in Table 5.1. Compared to the DAO-program without this feature, it improves the search speed by a factor of 4.20.

4.4 Precompute Segments

We precompute the low-level paths of the abstract edges with the goal to reduce the overall search effort. We use more preprocessing time and more storage.

In `Preprocess`, after the map abstraction is built and stored in the file, we get the edge information and do a A* search to find the low-level paths for all of the edges in the abstraction. We store this data in the file and load it in a hash map in function `PrepareForSearch`. During runtime, we retrieve the low-level paths of the edges from the hash map.

In caching edge paths, the low-level paths of the abstract edges need to be refined once before they can be accessed from the hash map. In precompute segments, the abstract edges that need to be refined during runtime are the ones from start to the next region center, and from the last region center to the goal. Therefore, by precomputing the low-level paths of the edges, we reduce the number of edges that need to be refined during runtime. Precomputation without consideration of edge symmetry requires 91 MB of disk space. It increases the online search speed by a factor of 1.60. The full results of DAO with precomputation is listed in Table 5.1.

The pseudocode of PrecomputeSegments is listed below.

PrecomputeSegments

```
1  // for all sectors
2  for i = 1 to Sectors.size
3  // for all the regions in sector i
4       for j = 1 to Regions.size
5            // Get all edges from region j
6            edges = GetEdges(Sector i, Region j)
7            // Get start and goal in (Sector:Region) for each edge
8            for x = 1 to edges.size
9                 goal = edges.GetAdjacentSector()
10                // The start location is the current sector and region
11                // Get x/y coordinates for start and goal location
12                s = start, g = goal
13                // use A* to find the low-level path from s to g
14                path_x = GetPath(s, g)
15                // write path to file
16                fwrite(path, filename)
```

4.5 Weighted A*

The weight A* algorithm [Pohl, 1970] is a variant of the classic A* algorithm [Hart et al., 1968].

A* uses the evaluation function $f(n) = g(n)+h(n)$, where $g(n)$ is the actual cost from start to the current node n, and $h(n)$ is the estimated cost from the current node to goal. Weighted A* has an evaluation function of $f(n) = g(n) + w \cdot h(n)$, $w > 1$. A* is complete on finite graphs and optimal when $h(n)$ is consistent and admissible [Hart et al., 1968]. Weighted A* is also complete on finite graphs and gives w-optimal results.

Weighted A* prioritizes nodes that are closer to the goal. Once the search is close to the goal, the objective is to quickly reach the goal instead of finding the optimal path. If A* gives an optimal solution with cost C, then weighted A* gives a suboptimal solution of at most $w \cdot C$. Figure 4.2 demonstrates an example of weighted A*. At node a, $g(a) = 1$, $h(a) = 30$, and $w = 10$, hence $f(a) = g(a) + w \cdot h(a) = 1 + 10 \times 30 = 301$. Weighted A* will not expand node a and will take the path that cost 300 and reach g. In comparison, A* will expand node a and find the optimal path $s \to a \to g$ with an f-cost of 31.

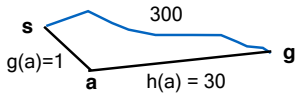

Figure 4.2: Weighted A* example

In this thesis we explore the weighted A* algorithm with different weights in the abstract graph pathfinding only, giving up path optimality for speed. The DAO program weight A* performance is given in Table 5.2.

Chapter 5

Experimental Results

In this chapter we present the DAO program's performance with the implemented enhancements described in Chapter 4. The complete result is summarized in Table 5.1 and Table 5.2.

5.1 Test Environments

The DAO program was run on the server with the following specifications:

CPU: 2×Intel xeon E5620 Quad-Core 2.40GHz, 12MB Cache.

RAM: 12GB.

Hard Drive: 500GB Seagate Constellation ES (3Gb/s, 7.2K RPM, 32MB Cache).

The metrics tested include:

- The total time to find a complete path $\Pi = \{\pi_0, \pi_1 \cdots \pi_n\}$ for the entire problem set of GPPC.
- The time to find the first 20 steps of the path, which is the first few π_i along the complete path Π. This describes how fast the pathfinding starts. Since

the abstract path needs to be computed before taking the first move, this metric also indicates the performance of pathfinding in the abstract space.

- The maximum time for returning any portion of a path, which describes the pathfinding performance in steady state.
- The total path length (suboptimality). In Table 5.1 we present the average suboptimality. This takes into consideration that some entries the GPPC performs well on long paths rather than short paths. An overall suboptimality does not differentiate an entry's performance on different path lengths.
- The total memory (RAM) usage, which includes RAM (before) and RAM (after). RAM (before) describes the size of the data that is loaded into memory before runtime. For the DAO program, RAM (before) represents the size of the program itself, the map abstraction and any precomputation data. RAM (after) represents the space the program uses during runtime, for example, the memory needed for operating the open list.
- The total disk usage, which describes the storage needed for the map abstraction and the precomputation data.
- The preprocessing time.

The test that we performed on the DAO program are:

1. Return the low-level paths incrementally.
2. Cache the low-level edge paths during runtime without considering symmetry.
3. Cache the low-level edge paths during runtime considering symmetry.
4. with `PrecomputeSegments` but does not consider symmetry.

5. with `PrecomputeSegments` and considers symmetry.

6. Weight A* on abstract pathfinding with weights: 1.5, 2, 4, 8, and 15.

5.2 Experimental Results

The GPPC problem set has 347,868 distinct problems. We ran the problems 5 times for each map for statistical significance. Therefore, there are a total number of 1,739,340 problems in the GPPC.

5.2.1 Comprehensive results

Table 5.1 gives full results of the DAO program with each enhancement. The entries in each column use the following enhancements: returning the path incrementally, using weighted A* ($weight = 15$) and no caching, caching edge paths during runtime without consideration of edge symmetry, caching with consideration of edge symmetry, precompute edge paths without consideration of symmetry, and precompute edge paths with consideration of symmetry.

Comparing columns 2 and 3 from Table 5.1, returning the path incrementally and using weighted A* with a weight of 15, the average time per path of the latter decreases by a factor of 1.59 compared to the former. Weighted A* is effective at improving the pathfinding speed in the abstract space, as time for the first 20 moves decreases by a factor of 2.19 with weighted A*. The metric " time for the first 20 moves" is dominated by the abstract computation. The maximum time per segment decreases in accordance with the time for first 20 moves because the abstract pathfinding work is being done before taking the first move.

Comparing columns 2 and 5 from Table 5.1, returning the path incrementally and caching with symmetry, the average time per path of the latter decreases by a factor of 4.20 compared to the former. Both of these two methods do not use

weighted A*, so there is no speed up in the abstract space, as the time of the first 20 moves are within 99% confidence interval which we will describe later in subsection 5.2.3.

Comparing columns 5 and 7 from Table 5.1, caching and precomputation with symmetry, the metrics "average time per path" and "time for the first 20 moves" are both within 99% confidence interval. These two methods have similar pathfinding speed in the abstract graph, since the maximum time per segment is 2.495 ms and 2.494 ms respectively.

Table 5.1: DAO-1-level performance

Entry	Incremental noCaching	w=15 noCaching	Caching noSym	Caching Sym	Precmpt noSym	Precmpt Sym
Total Time (s)	27814.5	17439.0	6640.3	6616.1	6553.6	6609.4
avg_time (ms) per path	15.991	10.026	3.818	3.804	3.768	3.800
first 20 (ms)	2.664	1.216	2.530	2.536	2.537	2.534
MaxTime (ms) (Per seg)	2.620	1.178	2.492	2.495	2.494	2.494
Avg_seg Time (ms)	0.090	0.058	0.029	0.028	0.025	0.028
Avg Path Length	2463	2532	2463	2463	2463	2463
Average Subopt	1.1285	1.1848	1.1285	1.1285	1.1285	1.1285
RAM (MB) (before)	14.82	15.28	15.28	15.28	16.39	15.28
RAM (MB) (after)	55.15	56.37	56.08	55.90	56.85	55.89
Storage (MB)	3.5	3.5	3.5	3.5	91	48
Pre-cmpt time (min)	0	0	0	0	1	0.5

5.2.2 Weighted A* results

Table 5.2 presents the results with the weighted A* algorithm using weights $w = \{1.5, 2, 4, 8, 15\}$. The DAO program with caching (no symmetry check) is listed as $w = 1$ for comparison. Comparing the performance of $w = 15$ with not using weighted A*, the suboptimality of the path is greater by a factor of 1.05 but the pathfinding time is shorter by a factor of 1.57.

Improvements of the pathfinding performance in the abstract space are seen with larger weights. The first 20 moves time decreases from 2.049 to 1.169 as the weight increases from 1.5 to 15.

Table 5.2: DAO-1-level weighted A* performance

Entry	Caching noSym	w=1.5	w=2	w=4	w=8	w=15
Total Time (s)	6640.3	5769.2	5515.0	4871.3	4382.5	4228.7
avg_time (ms) per path	3.818	3.317	3.171	2.801	2.520	2.431
first 20 (ms)	2.530	2.049	1.911	1.542	1.260	1.169
MaxTime (ms) (Per seg)	2.492	2.015	1.880	1.519	1.242	1.154
Avg_seg Time (ms)	0.029	0.022	0.021	0.019	0.018	0.018
Avg Path Length	2463	2481	2492	2515	2529	2532
Average Subopt	1.1285	1.1452	1.1548	1.1714	1.1818	1.1848
RAM (MB) (before)	15.28	15.28	15.28	15.28	15.28	15.28
RAM (MB) (after)	56.08	56.20	56.25	56.61	56.76	56.80
Storage (MB)	3.5	3.5	3.5	3.5	3.5	3.5
Pre-cmpt time (min)	0	0	0	0	0	0

5.2.3 Confidence interval analysis

Table 5.3 shows the 99% confidence interval for results of Caching with Symmetry (Caching Sym), Caching without Symmetry (Caching noSym), and Precomputation without Symmetry (Precmpt noSym). For the full results, we ran the entire GPPC problem set 5 times for statistical significance, and the confidence interval is calculated based on these 5 different results.

For Caching Sym and noSym, the confidence intervals of the average time per path, the first 20 moves time, and the maximum time per segment overlap with each other. Therefore, we conclude that the performance of caching with and without consideration of edge symmetry are similar at a 99% confidence level. Average time per segment is stable since the confidence bounds are tight. The same conclusion on the aforementioned metrics can be drawn for Precmpt Sym and NoSym.

Comparing Caching and Precmpt, their confidence intervals of the average time per path, the first 20 moves time, and the maximum time per segment overlap with each other. This suggests that precomputing the low-level edge paths does not improve the performance by a significant margin. Precomputing uses 91 MB and 48 MB of disk storage with and without consideration of symmetry. In comparison, doing caching only uses 3.5 MB. Therefore, using caching and refining the low-level path at runtime is an appropriate method for the DAO program.

5.2.4 Cache hit analysis

To evaluate the effectiveness of caching the edges, we select four maps from the GPPC map set and show the number of times an edge is accessed in the hash map.

The hash map is built incrementally during runtime. An edge is put into the hash map the first time it appears in an abstract path. We count this as one cache hit. If an edge does not appear in any abstract path, then it will not be stored in the hash map. Therefore, the minimum cache hit is one.

Table 5.3: Confidence interval (99%)

Entry	avg_time(ms) per path	first 20 moves (ms)	MaxTime per seg(ms)	Avg_seg Time
Caching noSym	3.8176 ± 0.0104	2.5298 ± 0.0095	2.4914 ± 0.0091	0.029 ± 0.0
Caching Sym	3.8036 ± 0.0085	2.5360 ± 0.0072	2.4952 ± 0.0070	0.028 ± 0.0
Precmpt noSym	3.7680 ± 0.0078	2.5350 ± 0.0081	2.4920 ± 0.0079	0.025 ± 0.0
Precmpt Sym	3.8010 ± 0.0022	2.5350 ± 0.0030	2.4940 ± 0.0030	0.028 ± 0.0

In the "wounded coast" map of Dragon Age 2 (Figure 5.1), the abstraction has 404 regions and 1450 edges. With consideration of edge symmetry, the hash map will store at most 725 low-level edge paths. For 2140 pathfinding queries, 248 out of 725 edges are accessed in the hash map. The minimum and maximum cache hits are 1 and 1224, respectively. The cache hit distribution is shown in Figure 5.2. 15.6% of edges (113 out of 725 edges) contribute to 95.7% of the total cache hits. This indicates that if we precompute the edge paths and store all of them, we can reduce the hash map entries by 84.4% if we only save the most accessed 113 edges for this map, although this implementation is not explored in this thesis.

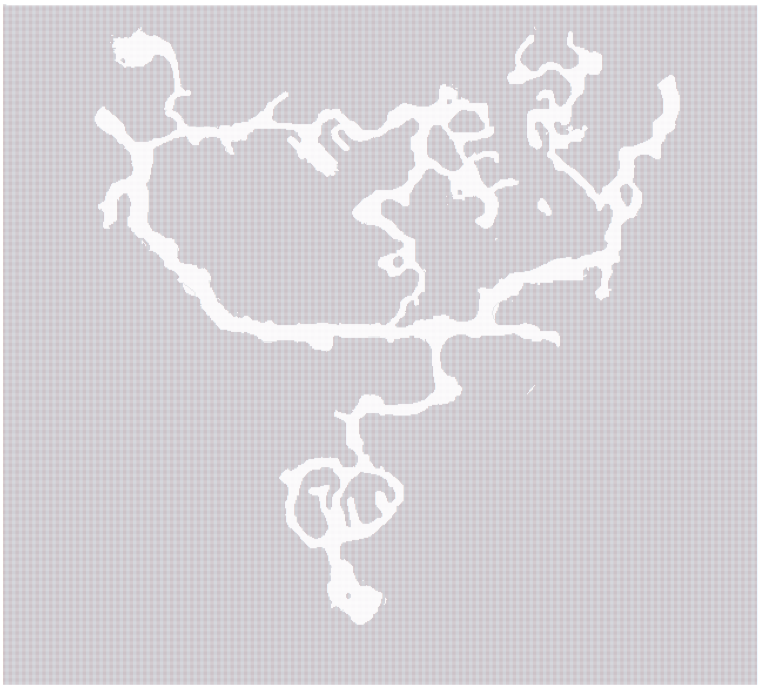

Figure 5.1: The “wounded coast” map (height 578, width 642).

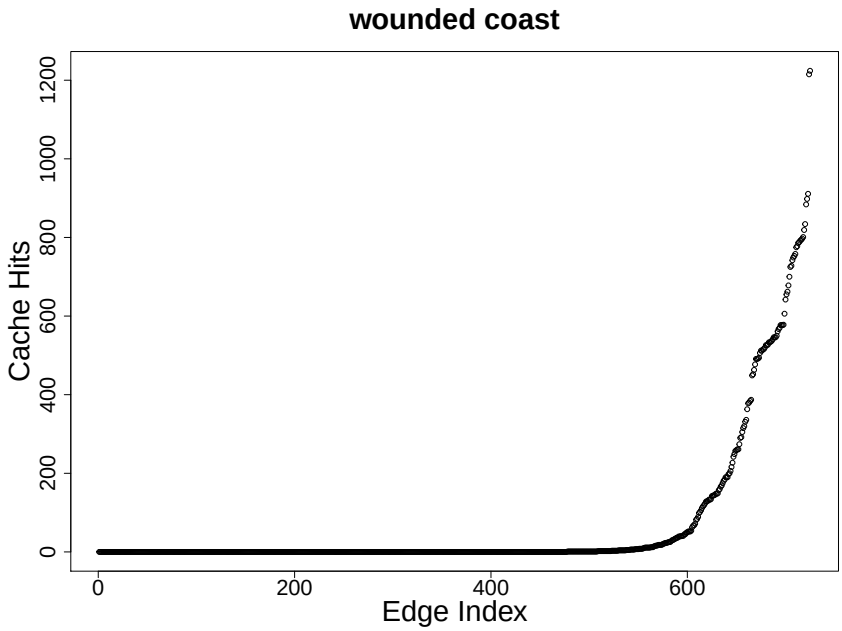

Figure 5.2: Cache hits for map “wounded coast”. Number of queries: 2140.

In the orz901d map from Dragon Age: Origins® (Figure 5.3), the abstraction has 375 regions and 1750 edges. With consideration of edge symmetry, the hash maps will store at most 875 low-level edge paths. For 4206 queries, there are a total number of 255 cache hits. The minimum and maximum are 1 and 3503, respectively. Figure 5.4 shows the cache hits distribution. 9.6% of edges (84 out of 875 edges) are accessed in the hash map over 1000 times. They account for 92.0% of the total cache hits.

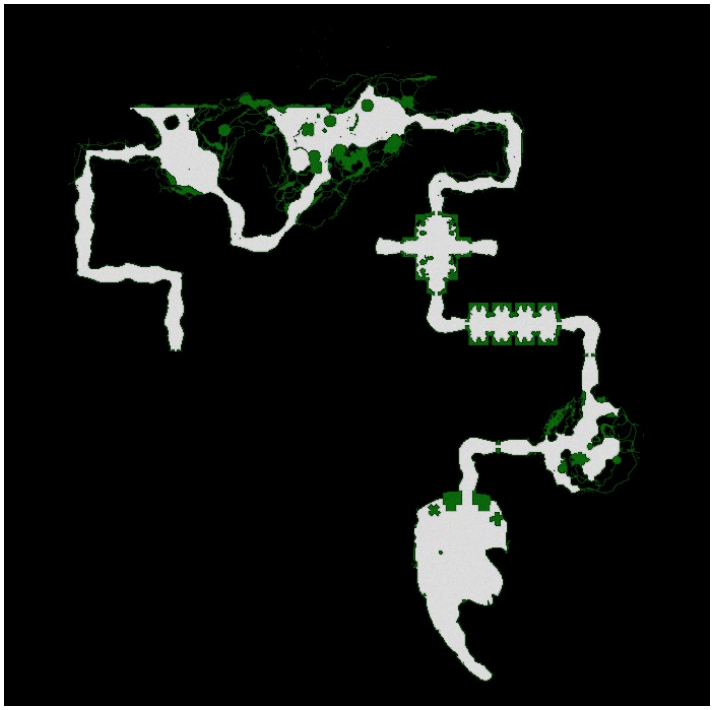

Figure 5.3: orz901d map (Dragon Age: Origins), height 678, width 601.

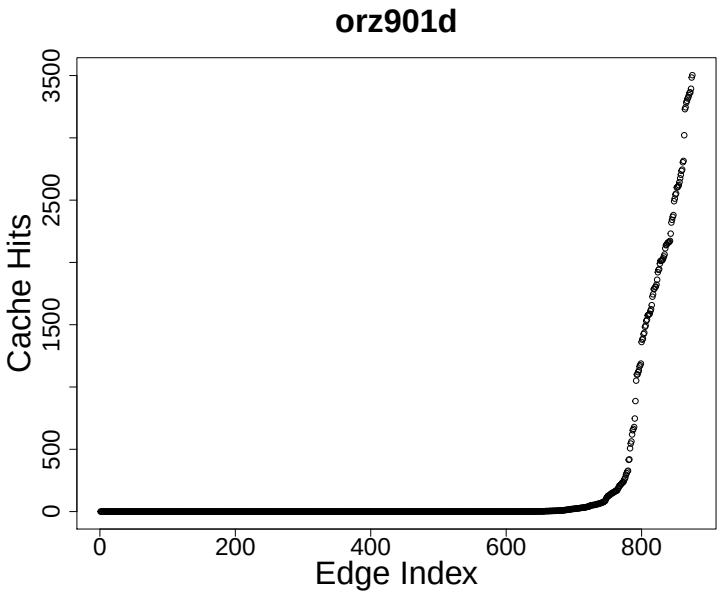

Figure 5.4: Cache hits for map orz901d

In the Aurora map of Starcraft® (Figure 5.5), the abstraction has 3538 regions and 17890 edges. With consideration of edge symmetry, the hash maps will store at most 8945 low-level edge paths. For 2990 pathfinding queries, there are a total number of 3179 cache hits. The minimum and maximum cache hits are 1 and 512, respectively. 16.8% of edges (1503 out of 8945) edges contribute to 92.9% of the total cache hits.

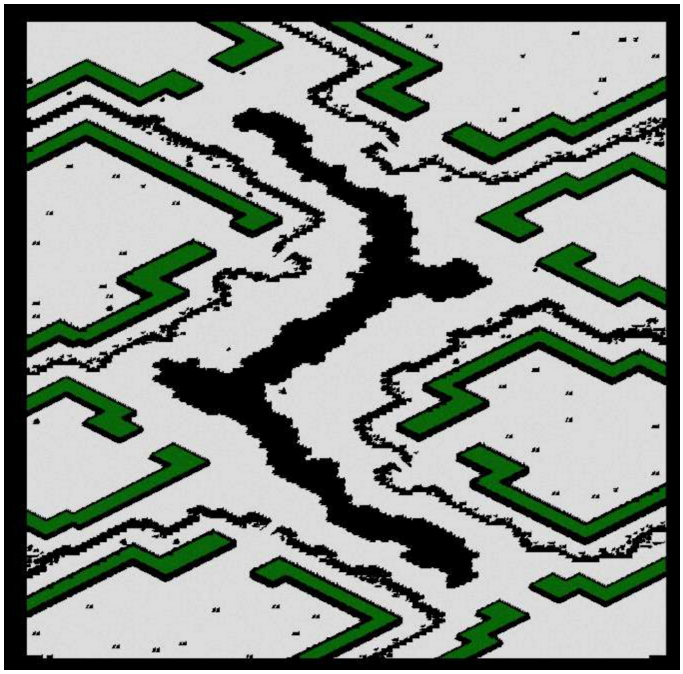

Figure 5.5: Aurora map (Starcraft), height 768, width 1024.

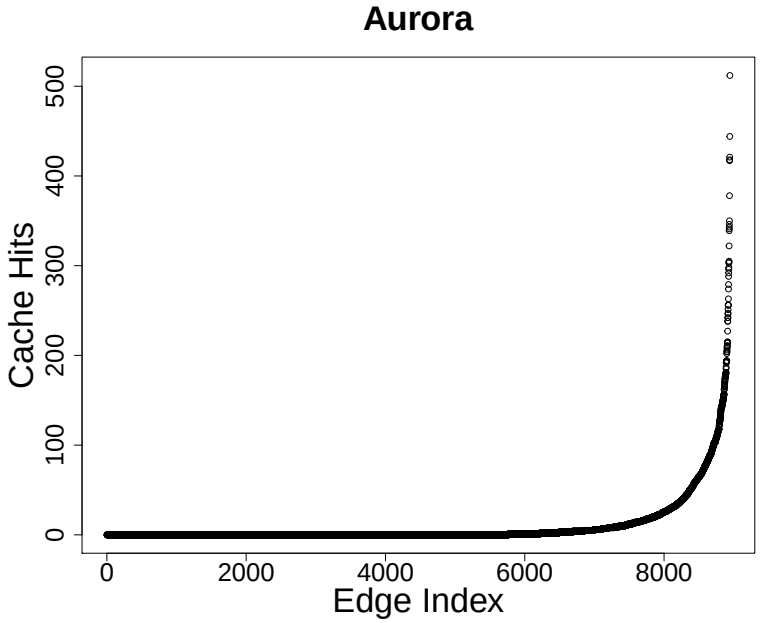

Figure 5.6: Caching hits for map Aurora. Number of queries: 2990.

In a random map of size 400×400 (Figure 5.7), the abstraction has 2436 regions and 6894 edges. With consideration of edge symmetry, the hash maps will store at most 3447 low-level edge paths. For 1660 pathfinding queries, there are a total number of 1298 cache hits. The minimum and maximum cache hits are 1 and 331, respectively. Figure 5.8 shows the cache hits distribution. Figure 5.8 shows that 26.5% (913 out of 3447) edges contribute to 97.3% of the total cache hits.

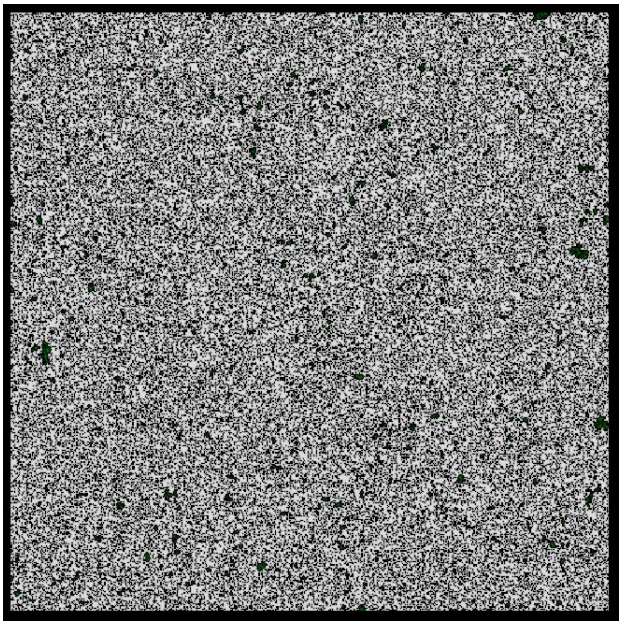

Figure 5.7: example random map, exactly 35% filled

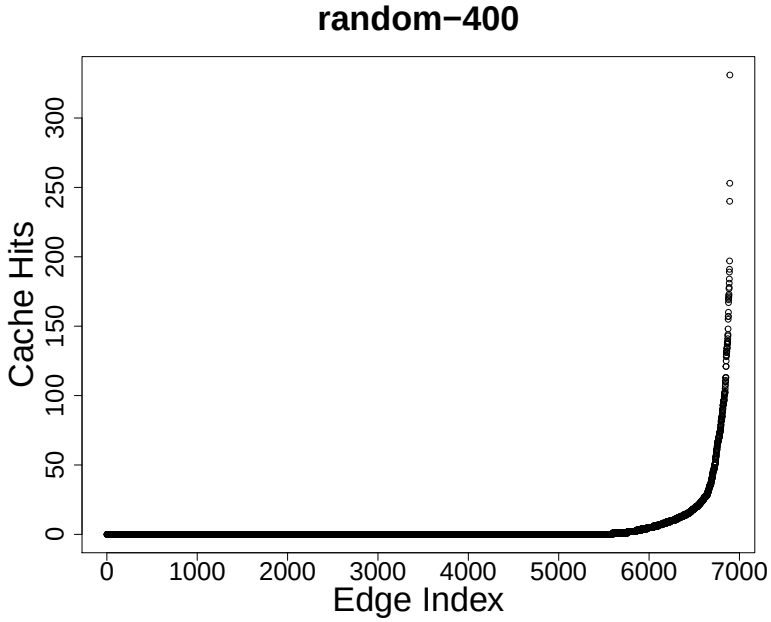

Figure 5.8: Caching hits for map random-400-33. Number of queries: 1660.

5.2.5 Summary

To summarize what we have learned from the experimental results:

1. Weighted A* improves the time required for computing the abstract path. This is done first, therefore improvements are shown in the time for the first 20 moves, and maximum time per segment.

2. The caching edge of paths during runtime improves the average time per path, but not the maximum time per segment or time for the first 20 moves, since these are dominated by the abstract path computation.

3. The cache hit analysis in Section 5.2.4 shows that only a small percentage of the edge paths need to be cached.

The pathfinding time is still dominated by search in the abstract graph. Therefore, the best configuration is to use caching, weighted A*, and other methods that

improve the abstract search such as adding another level of abstraction [Sturtevant and Geisberger, 2010].

Chapter 6

Conclusion

In this thesis we described the approach of building a memory-efficient map abstraction for pathfinding in static grid-based maps. We implemented enhancements to the 2013 DAO program to improve its performance.

In the DAO-1-level program, we return the low-level path incrementally, therefore, our program can handle larger maps. We also tested methods that use more memory to improve the online search speed. Firstly, we cached the low-level paths of the abstract edges during runtime. Secondly, we precomputed the low-level paths of edges and used them online to save search effort. Last but not least, we tested the weighted A* algorithm to speed up the search by reasonably sacrificing optimality.

We presented the performance of the DAO program with each enhancement implemented in Chapter 5. We showed that online pathfinding speed can be improved by using more memory and/or more disk storage by doing as much work as possible during precomputation.

There are several areas for future research:

1. The DAO program can be modified to accommodate for dynamic environment.

2. Build an α-level abstraction of the map.

3. Apply canonical ordering JPS method on high-level pathfinding.

4. Modify the caching edge paths component to only cache edges that are frequently hit, both in caching and in precomputation, as discussed in section 5.2.4.

Bibliography

[Ahmadi et al., 2008] Ahmadi, S., Ebadi, H., and Valadan, Z. (2008). A new method for path finding of power transmission lines in geospatial information system using raster networks and minimum of mean algorithm. *World Applied Sciences Journal*, 3(2):269–277.

[Algfoor et al., 2015] Algfoor, Z. A., Sunar, M. S., and Kolivand, H. (2015). A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, vol. 2015:11.

[Anguelov, 2011] Anguelov, B. (2011). *Video game pathfinding and improvements to discrete search on grid-based maps*. PhD thesis, University of Pretoria Pretoria.

[Botea, 2012] Botea, A. (2012). Fast, optimal pathfinding with compressed path databases. In *SOCS*.

[De Smith, 2003] De Smith, M. (2003). Gis, distance, paths and anisotropy. *Advanced Spatial Analysis: the CASA Book of GIS*, pages 309–326.

[Deits, 2014] Deits, R. L. (2014). *Convex segmentation and mixed-integer footstep planning for a walking robot*. PhD thesis, Massachusetts Institute of Technology.

[Delcomyn, 2007] Delcomyn, F. (2007). *Biologically inspired robots*. INTECH Open Access Publisher.

[Dolgov et al., 2008] Dolgov, D., Thrun, S., Montemerlo, M., and Diebel, J. (2008). Practical search techniques in path planning for autonomous driving. *Ann Arbor*, 1001:48105.

[Freund and Hoyer, 1988] Freund, E. and Hoyer, H. (1988). Real-time pathfinding in multirobot systems including obstacle avoidance. *The International journal of robotics research*, 7(1):42–70.

[Geisberger et al., 2008] Geisberger, R., Sanders, P., Schultes, D., and Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer.

[Harabor et al., 2011] Harabor, D. D., Grastien, A., et al. (2011). Online graph pruning for pathfinding on grid maps. In *AAAI*.

[Harabor et al., 2014] Harabor, D. D., Grastien, A., et al. (2014). Improving jump point search. In *ICAPS*.

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

[Holte et al., 1996] Holte, R. C., Perez, M. B., Zimmer, R. M., and MacDonald, A. J. (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI, Vol. 1*, pages 530–535. Citeseer.

[Millán and Torras, 1992] Millán, J. D. R. and Torras, C. (1992). A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning*, 8(3-4):363–395.

[Moghaddam and Delavar, 2007] Moghaddam, H. K. and Delavar, M. R. (2007). A gis-based pipelining using fuzzy logic and statistical models. *International Journal of Computer Science and Network Security*, 7(2):117–123.

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

[Rabin, 2014] Rabin, S. (2014). *AI Game Programming Wisdom 4*, volume 4. Nelson Education.

[Russell et al., 1995] Russell, S., Norvig, P., and Intelligence, A. (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27.

[Storandt, 2013] Storandt, S. (2013). Contraction hierarchies on grid graphs. In *KI 2013: Advances in Artificial Intelligence*, pages 236–247. Springer.

[Sturtevant, 2012] Sturtevant, N. (2012). Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148.

[Sturtevant and Jansen, 2007] Sturtevant, N. and Jansen, R. (2007). An analysis of map-based abstraction and refinement. In *Abstraction, Reformulation, and Approximation*, pages 344–358. Springer.

[Sturtevant, 2007] Sturtevant, N. R. (2007). Memory-efficient abstractions for pathfinding. In *AIIDE*, pages 31–36.

[Sturtevant, 2014] Sturtevant, N. R. (2014). The grid-based path planning competition. *AI Magazine*, 35(3):66–69.

[Sturtevant and Geisberger, 2010] Sturtevant, N. R. and Geisberger, R. (2010). A comparison of high-level approaches for speeding up pathfinding. In *AIIDE*.

[Sturtevant et al., 2015] Sturtevant, N. R., Traish, J., Tulip, J., Uras, T., Koenig, S., Strasser, B., Botea, A., Harabor, D., and Rabin, S. (2015). The grid-based path planning competition: 2014 entries and results. In *Eighth Annual Symposium on Combinatorial Search*.

[Tiwari, 2012] Tiwari, R. (2012). *Intelligent Planning for Mobile Robotics: Algorithmic Approaches: Algorithmic Approaches*. IGI Global.

[Tomczyk, 2011] Tomczyk, A. M. (2011). A {GIS} assessment and modelling of environmental sensitivity of recreational trails: The case of gorce national park, poland. *Applied Geography*, 31(1):339 – 351. Hazards.

[Xiang, 1996] Xiang, W.-N. (1996). A gis based method for trail alignment planning. *Landscape and Urban Planning*, 35(1):11–23.