UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)

DOCTORAL THESIS

Toward human-like pathfinding with hierarchical approaches and the GPS of the brain theory

Author:
Vahid RAHMANI

Supervisor:
Dr. Nuria PELECHANO

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Research Center for Visualization, Virtual Reality and Graphics Interaction (ViRVIG)
Department of Computer Science

September 9, 2020

Declaration of Authorship

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)

Abstract

Department of Computer Science

Doctor of Philosophy

Toward human-like pathfinding with hierarchical approaches and the GPS of the brain theory

by Vahid RAHMANI

Pathfinding for autonomous agents and robots has been traditionally driven by finding optimal paths. Where typically optimality means finding the shortest path between two points in a given environment. However, optimality may not always be strictly necessary. For example, in the case of video games, often computing the paths for non-player characters (NPC) must be done under strict time constraints to guarantee real time simulation. In those cases, performance is more important than finding the shortest path, specially because often a sub-optimal path can be just as convincing from the point of view of the player. When simulating virtual humanoids, pathfinding has also been used with the same goal: finding the shortest path. However, humans very rarely follow precise shortest paths, and thus there are other aspects of human decision making and path planning strategies that should be incorporated in current simulation models. In this thesis we first focus on improving performance optimallity to handle as many virtual agents as possible, and then introduce neuroscience research to propose pathfinding algorithms that attempt to mimic humans in a more realistic manner.

In the case of simulating NPCs for video games, one of the main challenges is to compute paths as efficiently as possible for groups of agents. As both the size of the environments and the number of autonomous agents increase, it becomes harder to obtain results in real time under the constraints of memory and computing resources. For this purpose we explored hierarchical approaches for two reasons: (1) they have shown important performance improvements for regular grids and other abstract problems, and (2) humans tend to plan trajectories also following an top-bottom abstraction, focusing first on high level location and then refining the path as they move between those high level locations. Therefore, we believe that hierarchical approaches combine the best of our two goals: improving performance for multi-agent pathfinding and achieving more human-like pathfinding.

Hierarchical approaches, such as HNA* (Hierarchical A* for Navigation Meshes) can compute paths more efficiently, although only for certain configurations of the hierarchy. For other configurations, the method suffers from a bottleneck in the step that connects the Start and Goal positions with the hierarchy. This bottleneck can drop performance drastically.

In this thesis we present different approaches to solve the HNA* bottleneck and thus obtain a performance boost for all hierarchical configurations. The first method relies on further memory storage, and the second one uses parallelism on the GPU.

Our comparative evaluation shows that both approaches offer speed-ups as high as 9x faster than A*, and show no limitations based on hierarchical configuration. Then we further exploit the potential of CUDA parallelism, to extend our implementation to HNA* for multi-agent path finding. Our method can now compute paths for over 500K agents simultaneously in real-time, with speed-ups above 15x faster than a parallel multi-agent implementation using A*.

We then focus on studying neurosience research to learn about the way that humans build mental maps, in order to propose novel algorithms that take those finding into account when simulating virtual humans. We propose a novel algorithm for path finding that is inspired by neuroscience research on how the brain learns and builds cognitive maps. Our method represents the space as a hexagonal grid, based on the GPS of the brain theory, and fires memory cells as counters. Our path finder then combines a method for exploring unknown environments while building such a cognitive map, with an A* search using a modified heuristic that takes into account the GPS of the brain cognitive map.

Acknowledgements

This PhD. Thesis would not have been possible without the great supervision of my advisor, *Dr. Nuria Pelechano*, to whom I ought being where I am now. She has been the perfect guide to a work she envisioned from the first moment, and she has been able to push me in those moments when I think my work is stuck. She has almost as much credit as I have in all the work I have done.

I gratefully acknowledge research center of *Visualization, Virtual Reality and Graphics Interaction (VIRViG)*, for providing me financial support and all professors and staffs to allow me to carryout my research in this esteemed laboratory.

I would also to express my appreciation to my friends for the mode that me enthusiasm and encouragement. Special shout-outs to Nogol panahi, Hani Baloochi, Mohammad Milani, Leila Mashhori, Mahdieh Farrokhi, Farzaneh Aghbali, Mohammad Akram Narouei, Sahel Naraghi, Porochista Dorost, Mojtaba Taherkhani, Behnoosh Molaie, Chris Bennetts-Cash, Adam Rumbold, Ahmed Sabir, Leonel Toledo, Alfred Etxabe, Deniz charlo, Jordi Ramirez and Alexandra Nappelo.

I am grateful to my love, *Mahdiye*, who encouraged, supported and otherwise put up with me throughout this long endeavour. Her love has been like a light, shining brightly always and helping me find my way home.

Thank you to my brother, *Mohsen*, who constantly reminds me there is a more amusing side to life. Some of my favourite moments from all my life have been while spending my time with you.

Finally, my special words of thanks should also go to my parents, who uprooted their lives for the sake of my brother and I. You taught me the importance of education, you taught me drive and ambition, you taught me respect, compassion, forgiveness and love. You raised me as a boy and then you shaped me into a man. You showed me the path and I follow it still. Thank you.

Vahid Rahmani

Contents

List of Figures

List of Tables

Dedicated to all the heroic doctors, nurses, paramedics and medical staff that keep us safe and healthy from COVID-19…

Chapter 1

Introduction

1.1 Pathfinding

Pathfinding consists of solving the problem of finding a traversable path from a starting position to a goal position within an environment. Such path typically attempts to minimize certain cost, such as fuel, time, distance, equipment, money, etc. Path planning for multi-agents in large virtual environments is a central problem in a variety of fields such as robotics, video games, and crowd simulation for both static and dynamic environments. There are several aspects that need to be considered when computing paths. The first one, which attracts most of the interest from the research community, is how to compute paths efficiently so that we can handle large environments with many autonomous agents in real time. The second one, which has been mostly ignored by the literature, is how to compute paths that are as human-like as possible, and not simply the result of some optimization (such as time, length, effort, etc).

In the case of video games, the need for highly efficient pathfinding techniques is crucial as modern games place high demands on CPU and memory usage. So, in video games and any real-time application that needs to be populated with autonomous agents, the effort is put in finding visually convincing paths with low computational cost. Typically, it is not necessary to obtain the optimal path for all agents, but those paths should at least look convincing to the viewer. Typically the plausibility of a path can be evaluated through user studies, where results are shown to a group of viewers and they have to answer questions regarding the quality of those trajectories. In order to compute paths for a video game, it is necessary that the total time required for all computations (e.g: rendering, physics simulation, AI, etc.) is kept under 40 ms to guarantee 25 frames per second. So often, it is possible to lessen the optimality requirement as long as the path is believable to the observer.

The problem of pathfinding can be separated from local movement, so that pathfinding provides the sequence of cells to cross in the navigation mesh, and other methods can be used to set waypoints to steer the agents and to handle collision avoidance. In previous work by my advisor [Pelechano and Fuentes, 2016a], a hierarchical approach for general navigation meshes was presented, known as HNA*. The method provided a hierarchical solution adapted to the peculiarities of navigation meshes where cells are convex polygons of different shapes and sizes. HNA* offered very good speeds ups for pathfinding, however only for certain configurations of the hierarchy. For other configurations, performance could drop drastically when inserting the start and goal position into the hierarchy. In chapters 3 and 4 of this thesis, we focus on abstraction hierarchies applied to pathfinding to improve both computing and memory performance to eliminate the bottleneck that appeared in HNA*.

There have been many efforts to simulate virtual humans in a way that resembles real humans. For instance to perform natural collision avoidance in a virtual environment. However, to the best of our knowledge, there is currently no relevant work in the literature when it comes to achieving long term paths that closely resemble the way that humans decide their whereabouts and that can be computed in real time. There are some algorithms that mimic animal behavior, such as ants leaving pheromones, of flocks of birds following basic rules. But there is no work in the literature trying to mimic the human brain theory on mental maps and way finding. There are still many unknowns about the way our brain works, and it is beyond this thesis to provide a complete solution for human-like pathfinding. In this thesis we have tried to get one step closer to real humans' pathfinding, by proposing algorithms that follow neuroscience theories that have been tested on other mammals but are believed to also apply to humans. Therefore, in chapter 5, we focus on studying neuroscience research to learn about the way that mammals build mental maps, in order to propose novel algorithms that take those finding into account when simulating virtual humans.

In this first chapter, we introduce the motivations and the main problems of pathfinding which are the challenges that we aimed to solve with this thesis. We also present our goals, and list our contributions to the pathfinding research. Finally we present the list of publications resulting from this thesis.

1.2 Thesis Claim And Document Organization

Pathfinding (also referred to as path planning) is one of the most important and also interesting research topic in the artificial Intelligence community. The challenge of pathfinding in video games is to compute optimal or near optimal paths as efficiently as possible. As both the size of the environments and the number of autonomous agents increase, this computation has to be done under hard constraints of memory and CPU resources. The problem of pathfinding can be separated from local movement, so that pathfinding provides the sequence of cells to cross in the navigation mesh, and other methods can be used to set attractors (or waypoints) and to handle collision avoidance. After having laid out the goals and motivation of this thesis in chapter 1, in chapter 2 we present a literature review and essential concepts in pathfinding to establish the research grounds for our work.

In this thesis, we focus on abstraction hierarchies applied to pathfinding to improve performance. More specifically we focus on the HNA^* algorithm (Hierarchical PathFinding for Navigation Meshes) [Pelechano and Fuentes, 2016b], which is a bottom up method to create a hierarchical representation based on a multilevel k-way partitioning algorithm (MLKP) of a navigation mesh.

Hierarchical approaches, such as HNA^* can compute paths very efficiently, since they drastically reduce the size of the search space. However, in the case of the original HNA^*, experimental results showed that this is only true for certain configurations, while for others performance could drop drastically. This presented a huge limitation when using HNA^* because it required that the programmer had to determine the best values for the configuration parameters through a long trial and error process, which involved exhaustively testing with the specific environment. Previous work, observed that a bottleneck could appear when inserting the start and goal position into the hierarchy. This bottleneck seemed to appear when there were large high level nodes with many *inter-edges*. Therefore, when starting with this thesis, the first motivation was to detect and correct the source of such limitation, in order to guarantee that we could achieve high speed-ups using HNA^* for any navigation mesh regardless of the hierarchy configuration.

In chapter 3, we first provide a formulation of the base problem, to then provide mathematical proofs for the upper-bounds on the number of *inter-edges* which is the source of the bottleneck. Then we present improvements to HNA^* and successfully eliminate the bottleneck. We propose different methods that rely on either further memory storage or parallelism on both CPU and GPU, and carry out a comparative evaluation.

In chapter 4, we propose methods to achieve a parallel version of HNA^* that can compute pathfinding for large groups of agents. We focus on abstraction hierarchies applied to multi-agent pathfinding to improve performance. In this chapter we studied how to parallelyze our improved HNA* algorithm from chapter 3, and manage threads and memory correctly to exploit the performance boost of HNA^* over large crowds of agents.

Finally in chapter 5, we propose a new algorithm for pathfinding that is inspired by neuroscience research on how the human's brain learns and builds cognitive maps. In this method we have used the human's brain path planning theory and strategy to implement a human-like pathfinding algorithm. We also propose a novel heuristic based exploration algorithm as an attempt to mimic human behavior in unknown or partly known environments.

1.3 Motivations

From the begging of this thesis, the main motivation was not only to improve performance for pathfinding algorithms, but to research algorithms that could better resemble the way real humans behave. Pathfinding has been used for a long time in robotics and for simulation of autonomous agents. In robotics the goal is to find a path between two points without colliding, but it does not matter whether this path is human-like or not. In autonomous agents simulation, often the goal is similar, just to get agents moving through an environment without colliding with others or getting stuck in local minima. But neither of those research areas typically worry about the quality of the paths from a human perception point of view. When attempting to simulate virtual humans to populate large virtual environments, the previous goals still hold (finding optimal paths, with low computation), but we also aim to find paths that exhibit some resemblance to human pathfinding. Achieving human-like behavior is extremely difficult, due partly to the heterogeneity of human decision making, but also due to the difficulties in evaluating the resulting paths from a perceptual perspective. Therefore, in this thesis we pursued two main goals: (1) research hierarchical pathfinding approaches, since they can improve performance, while mimicking better how humans plan their paths, and (2) simulating the GPS of the brain theory as a new insight into more human-like virtual agents. More specifically the two main motivations for this thesis are:

1. **Hierarchical path planning:**

   Path planning deals with finding a sequence of state transition actions that transform a start position to a goal position, where each passing action has an associated cost, and the sum of costs of all passing actions describes some measurements for the path. In most of the applications pathfinding algorithms should provide the traversable shortest path in real-time in a large environment. So, it is necessary to improve the pathfinding algorithms to find paths as efficiently as possible in terms of path length and planning time. Hierarchical approaches can reduce the size of the problem, by creating higher levels nodes that contain a subset of the navigation mesh cells, and which are connected by *inter-edges* that guarantee that if there is a path in the navigation mesh, then there will also exist a solution in a higher level of the hierarchy. Therefore hierarchical approaches can still guarantee a solution if a path exists, although they cannot guarantee finding the shortest path. But the most important element is that solving a problem with such a hierarchical approach, better resembles how humans behave. For instance, when humans think about how to get form a house in one city to another house in another city, they first plan the main roads and cities they need to go through, then for each of the cities they need to traverse, they think of the exact sequence of streets to get from one end of the city to the other, and they will only need to do this in a sequential order as they move through the road. There is no need to plan ahead the exact way of driving through a city, as it may change later on because we may decide to go through a different city, due to traffic conditions. Another important motivation which appear half way through this research, was to extend the path planing algorithms to cloud computing in order to evaluate the performance of path planing methods on multi-player games. This goal was brought to us by the video game company Improbable [Improbable, 2020] which was highly interested in our results and with which we have been collaborating during the last year of my thesis.

2. **Mental maps inspired by neuroscience research:**

   While existing techniques for pathfinding give possible solutions for practical applications, none of them take into account human factors to closely simulate how humans behave in the real world. There are many aspects of human behavior that affect route choice during navigation, such as: memory, mental maps, or visibility. In this thesis we were motivated to study pathfinding methods inspired by research from neuroscience. For this purpose, we wanted to develop novel models that could mimic how humans are believed to build mental maps according to research of the human brain navigation research, also known as the GPS of the brain [Hafting et al., 2005].

1.4 Problem Statement

As mentioned before in this thesis, the problem of pathfinding refers to planning a path from a start location to a goal location that meets some criteria such as: the shortest distance, the lowest cost or the fastest in a spacial network. But for computers, it can be hard to compute path for an agent in a dynamic or static environment. It can be even harder when the computer/agent does not have any knowledge about the world. Moreover, very often this computation needs to be done under hard constraints of limited memory and CPU resources. Solving this problem can become an important bottleneck specially when there are many agents navigating a large virtual environment.

The problem of pathfinding usually can be classified into two classes based on the number of agents: when we have just one agent, which is known as Single Agent Pathfinding (SAPF) [Silver, 2005], or when we have a group of agents where each agent hast its own start and goal points, which is known as Multi-Agent Pathfinding (MAPF) [Sharon et al., 2015a].

In the case of autonomous agents wandering around virtual worlds, we first need to generate a representation of the walkable space. This can be done with 2D grids where squared cells are marked as free to walk of obstacles, or else with some kind of polygon mesh where each polygon represents a walkable cell (with each cell being a convex polygons with 3 or more vertices). The problem of pathfinding can be separated from local movement, so that pathfinding provides the sequence of cells to cross in the navigation mesh, and other methods can be used to set waypoints and to handle collision avoidance against other moving agents in the cell. This case makes pathfinding easier, as each agent can focus exclusively on finding the sequence of cells that can take him from one point to another in the navigation mesh, without needing to worry about the whereabouts of other agents. This problem assumes thus that cells are large enough for several agents to walk through them simultaneously and that collision avoidance is solved with local steering algorithms.

In this thesis we refer to both Single agent and Multi-agent pathfinding problems, focusing exclusively on finding a path as a sequence of cells in a navigation mesh, and thus leaving collision avoidance with other moving agents to the local movement algorithm (for more information on local movement techniques and how they can be combined with high level pathfinding, we refer the reader to the following books: [Kapadia et al., 2015, Thalmann and Musse, 2013, Pelechano et al., 2016]).

1.5 Objectives

In order to achieve our main objective of exploring human-like approaches for pathfinding in real time, we have aimed at the following specific research goals:

- ***Improve Single Agent pathfinding (SAPF):*** After studying the state of the art, we decided that hierarchical approaches offered the possibility to simulate the way humans plan their whereabouts (starting from a high level sequence of key locations and then solving lower resolution paths as they get move through their high level plan). We thus chose to take the HNA^* (Hierarchical Navigation A*) previously developed by my supervisor, and we aimed at improving the bottleneck of the connecting start and goal positions with the hierarchy. We focused on studying in depth memory and CPU usage to decide on the best approaches to further boost performance for all hierarchical configurations. Our proposed solutions dealt with paralyzing parts of the HNA^* algorithm on CPU and GPU, to decrease the computational time of HNA^* for one single agent pathfinding.

- ***Improve Multi-Agent pathfinding (MAPF):*** The next goal of our research was to extend the HNA* algorithm to achieve fast and efficient pathfinding for large group of agents. To achieve this end, we focused on further paralyzing the improved version of HNA^* for multi-agent pathfinding in real-time.

- ***Mental maps based on the GPS of the brain*** Finally we propose a new algorithm for pathfinding that is inspired by neuroscience research on how the brain learns and builds cognitive maps. The main goal of this part of our research was to implement a method for pathfinding based on human pathfinding strategy to set the basis for future work on simulating virtual humanoids that closely mimic real humans.

1.6 Contributions

There have been three main contributions from this PhD thesis, each of them resulting in a publication (the last one is under review at the time of writing this document). Each contribution corresponds to one of the research goals stated in the previous section.

- ***Solving the bottleneck of HNA*:***

  Vahid Rahmani and Nuria Pelechano. "Improvements to hierarchical pathfinding for navigation meshes". In Proceedings of the Tenth ACM SIGGRAPH International Conference on Motion in Games (MIG '17), Barcelona, 2017.

- ***Proposing a parallel extension of HNA* to handle Multi-agent Pathfinding:***

  Vahid Rahmani and Nuria Pelechano. "Multi-agent parallel hierarchical pathfinding in navigation meshes (MA-HNA)." Computers & Graphics 86 (2020): 1-14.*

- ***Neoroscience-inspired pathfinding:***

  Vahid Rahmani and Nuria Pelechano. "Towards Human-like Agent Path Planning". Submitted for publication to ACM Transactions on Games.

Chapter 2

Literature review

2.1 Pathfinding

Pathfinding refers to the problem of an agent navigating from a start position to a goal position on a map. This research area has been well studied in Computer Science and demonstrates an active area of investigation in several sub-fields such as Artificial Intelligence, Computational Geometry, Computer Graphics, Video Game Development and Robotics. Many pathfinding methods exist, often targeting solutions for a specific context. In this chapter, we cover a broad number of topics from across the academic literature and review a range of both classical and more recent results. We focus especially on two popular variations of the artificial agent path planning problem: finding the shortest path in a discrete search graph and finding the shortest path in a continuous map. A wide range of methods has been considered for building discrete search graphs including grid maps, road maps, and other popular and successful techniques. We have then compared a variety of techniques like heuristic methods, abstraction techniques, and search space pruning strategies, that have been introduced through the years, for finding the shortest paths in discrete graphs.

2.2 Single-Agent Pathfinding Problem

Single-agent pathfinding or path planning is the problem of navigating a single entity like a robot routing (Cohen, Chitta, and Likhachev, 2014), (Bnaya et al., 2013), network routing(Broch et al., 1998), GPS navigation (Sturtevant and Buro, 2006) or a virtual agent, from a source position to a destination position in a given operating environment. In the usual traditional setting, environments can be two-dimensional Euclidean (i.e. flat) or three-dimensional Geodesic (i.e. curved) spaces [Harabor, 2014]. Generally, the environments can have the form of a spatial arrangement (i.e. a set of connected points) or they can be represented as a combination of walkable and non-walkable polytopes (the latter often being called obstacles). There are many varieties of the single-agent pathfinding problem. These occur by modifying certain parameters of the problem such as:

- ***The objective function.*** In the standard case, the purpose is to minimize travel distance from two given start and end points, although it is possible to have other functions based on things such as energy expenditure.

- ***The Agent's type.*** In the standard case, agents are represented as oriented points but they could have arbitrary shapes, sizes, and capabilities that limit or enhance the agent's movement.

- ***The performing environment***. Agents can operate in a (i) completely static environment, which are those that rely on data-knowledge sources of the environment not changing across time or dynamic environment with information being update frequently. (ii) a fully observable, which has access to all needed data to complete the target task, or else there are parts of the environemnt that are not known by the agent. (iii) discrete environment, like Chess, that a finite set of possibilities can drive the final outcome of the task or continuous environment like self-driving car. The type of environment has a strong influence on the most adequate pathfinding algorithm needed.

- ***The quality of the Solution.*** In the standard case, agents are challenged to find an optimal path between two given points. In some real-time or resource-constrained settings, a near-optimal or bounded suboptimal path may be favored.

- ***Path constraints.*** Typically agents simply need to move from the start point to the goal point without crossing any obstacles. In different settings additional limitations may complicate the agent's task; for instance, the agent may need to visit certain pre-specified places before reaching the goal position [Harabor, 2014].

2.3 Multi-Agent Pathfinding Problem

Multi-Agent Pathfinding (MAPF) is the problem of finding paths for a set of agents each with its own start and goal positions. The MAPF problem is a generalization of the single-agent pathfinding problem for $k > 1$ agents. The main task is to find the path for every agent while avoiding collisions. MAPF has practical applications in video games, traffic control [Silver, 2005; Dresner and Stone, 2008], robotics (Bennewitz, Burgard, and Thrun, 2002) and aviation [Pallottino et al., 2007]. Techniques for solving the MAPF problem can be classified into two categories: optimal and sub-optimal solutions. Obtaining an optimal solver for the MAPF problem is known to be an NP-hard problem [Yu and LaValle, 2013], since the search space increases exponentially with the increasing number of agents. Sub-optimal solutions are normally used when the number of agents is very large. In such cases, the purpose is to instantly find a path for several agents, and it is often indomitable to guarantee that a given solution is optimal [Sharon et al., 2015b].

Research in multi-agent path planning has observed a lot of progress in recent years, in part due to the first Competition of Distributed and Multi-Agent Path Planners, CoDMAP-15 [Komenda, Stolba, and Kovacs, 2016]. Many recent multi-agent planners are based on the MA-STRIPS formalism [Brafman and Domshlak, 2008], and can be loosely classify into one of two categories: centralized, in which agents have full information and share the goal; and distributed (decentralized), in which agents have partial information and individual goals [Furelos Blanco and Jonsson, 2018].

In CoDMAP-15, the most successful centralized planners were Agent Decomposition Planner (ADP) [Crosby, Rovatsos, and Petrick, 2013], MAP-LAPKT [Muise, Lipovetzky, and Ramirez, 2015] and CMAP [Borrajo, 2013], while prominently distributed planners included PSM [Tožička, Jakbuv, and Komenda, 2014], MAPlan [Štolba, Fišer, and Komenda, 2016] and MHFMAP [Torreño, Onaindia, and Sapena, 2014].

In this thesis we consider the problem of concurrent Decentralized and Non-communicating multi-agent planning in which agents can act in parallel at each time step with partial information and individual goals. This problem is challenging for several reasons:

In the case of video games, often computing the paths for non-player characters (NPC) needs to be done under strict time constraints to guarantee real time simulation. In those cases, performance is more important than finding the shortest path, specially because often a sub-optimal path can be just as convincing from the point of view of the player.

Another very important challenge is to compute paths as efficiently as possible for groups of agents. As both the size of the environments and the number of autonomous agents increase, it becomes harder to obtain results in real time under the constraints of memory and computing resources.

2.4 Search Graphs

Regardless of the problem variation at hand, practitioners typically all begin by constructing a model of the operating environment $G = (V, E)$ known as a search graph where$V$ is a set of admissible positions that an artificial agent can occupy. These are usually introduced as the nodes or vertices and E is the set of edges that connect adjacent vertices of the search space graph. Edges can be considered as paths or corridors that an agent can walk on or actions that can be executed in order to move the agent from one position to another position. The cost associated with each such move is called the edge weight. Weights often represent distance travelled but they could stand for other types of metrics as well; e.g. travel time or fuel consumption. When the cost of moving between two vertices a and b can differ to the cost of moving from b to a the graph is said to be directed. Otherwise the graph is said to be undirected [Harabor, 2014].

2.5 Paths and Instances

In pathfinding theory, A path $P =< v_0, v_1, \ldots, v_{k-1}, v_k >$ can be defined as a walk in a search graph $G = (V, E)$. Each v_i is a vertex in V and each couple of neighboring vertices (v_i, v_{i+1}) are connected by an associated edge in E. When searching for a path we define the start position of the agent s and its goal position g. When a pathfinding algorithm has found a path, its quality is typically evaluated in terms of path length or path cost. The path length refers to the number of edges that contain the path and the cost of a path refers to the total weight of all edges that contain the path. The lowest cost path from node s to node g in graph G will be an *optimal* path.

2.6 Graph Representations

The pathfinding problem is fundamentally a graph search problem, and thus it is performed through the use of a graph search technique. Generally, A graph search algorithm is an algorithm that, given a start and end nodes in a graph, attempts to obtain a minimum cost path between them. The obtained path is referred to as the optimal path, and when it comes to pathfinding, it typically corresponds to the shortest possible path. The pathfinding terms of a graph search method are that the algorithm will always obtain a path between two points if one exists, and that such path will be optimal (or near-optimal depending on the algorithm). Moreover, the processing calculation and memory usage of such graph algorithms should be minimized to perform successfully within tight performance and memory limitations. In this section we present the most common graph search techniques and methods that are employed in the literature.

Complex 3D environments can be represented in an abstract way using navigation graphs. A navigation graph can be treated as a search graph to perform pathfinding. There are many techniques to build such a search graph and, in this section, we review a wide range of popular methods like grid maps, navigation meshes, visibility graphs, shortest path maps, and road maps. All these graph search methods are instances of explicit search graphs. Explicit means that all nodes and all edges of the search graph are specified before any pathfinding query can begin. Such graphs appear in many applications including video games [Davis, 2000; Champandard, 2009], routing [Sanders and Schultes, 2005; Goldberg, Kaplan, and

Werneck, 2006] and robot motion planning [Latombe, 2012; Choset et al., 2005]. But some of the search graphs are implicit, which means that the nodes and edges of the graph are identified on-the-fly during the search. Implicit graphs arise in higher dimensional pathfinding applications [LaValle, 1998; Bohlin and Kavraki, 2000] and related fields such as AI Planning [Russell and Norvig, 2016].

When comparing different types of search graphs, there are two important properties that depend on the operating environment: solution existence and solution optimality [Harabor, 2014]. Solution existence in a search graph will guarantee that all non-obstacle areas in the search environment can be mapped to a vertex and that if two non-obstacle positions can be connected by a path in the search environment then those points can also be connected by a path in the search graph. A search graph which has solution optimality performs a similar but stronger guarantee: if a path between two locations exists in the search space graph then there also exists in the graph a path which is cost-optimal concerning the operating environment. But not all search graph preserves existence or optimality and each representation has its unique advantages and disadvantages. Choosing the most suitable one depends on the distinct requirements of the pathfinding problem at hand.

2.6.1 Grid maps

A grid map is one of the most known graph search types which uses a uniform subdivision of the operating environment into small regular squares which are usually called tiles or grid cells [Anguelov, 2012]. Each of these grid cells, can have up to eight adjacent neighbors and a traversability flag which shows whether each tile or grid cell is traversable or non-traversable. Traversable cell refers to the walkable areas and non-traversable refers to the obstacle cells. The overlaid grid is transformed into a graph by constructing an abstract vertex for each tile and then using the tile/cell connection geometry to define the graph edges. The tile connection geometry is defined by the type of tiles used to form the grid: a standard grid cell features a 4-neighborhood, hex tiles grant a 6-neighborhood, while the most common cell variety, the octile, features an 8-neighborhood [Yap, 2002]. Figure 2.1 illustrates these three cell connection geometries.

Constructing a grid cell search graph for a large environment like a video game will be simple and efficient [Millington and Funge, 2009] because the connection geometry is constant for each tile in a grid cell and it is not necessary to perform a

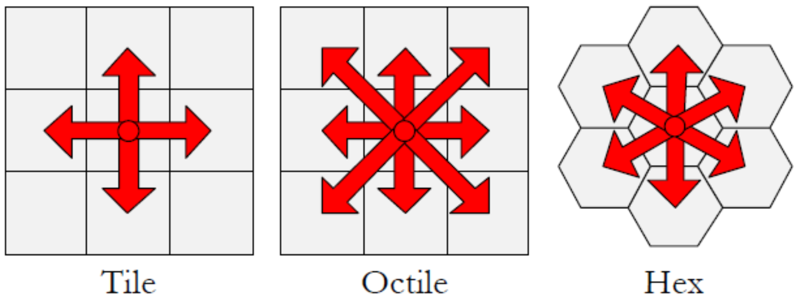


FIGURE 2.1: Examples of the most common grid cell geometries.

complex examination of the environment. The superimposition of a grid cell, the estimation of obstructed cells, and the construction of the navigation graph are demonstrated in Figure 2.2.

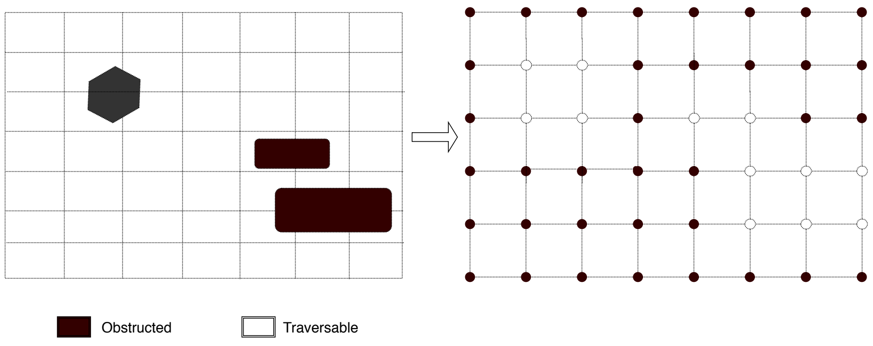


FIGURE 2.2: Grid Based representation of a game world environment

Generally, grid maps are highly popular for several reasons: (i) they are easy to understand and easy to implement (ii) they can be described as a matrix of bits and stored efficiently (iii) every single node can be updated in constant time. One of the important disadvantages of grid maps is their fixed resolution. In many cases, grids are too coarse to correctly represent the underlying environment. Another problem with this kind of representation is that it is not easy to increase the number of tiles without increasing the memory footprint. Therefore, in order to achieve a finer resolution it is necessary to increase the number of cells, which will result in higher memory requirements and a larger graph size. Consequently, pathfinding becomes more challenging as it requires to explore a larger graph. Finally, having such a regular grid structure has another disadvantage which is that it provides paths that are constrained to the points of the grid cells. Such paths may not only be unsuitable path but they can also be longer than essential and may need post-processing to “smooth” them.

2.6.2 Waypoint Based Navigation Graphs

Waypoint-based graph navigation is one of the traditional methods of abstraction for building a navigational graph from a path planning environment like a video game. These Waypoints can be distinct for each path or be a portion of the environment map. Waypoints can be placed manually throughout a level during the design scenario by level designers or calculated automatically and then linked together by hand or automatically to build the final navigation graph. Figure 2.3 shows the placing and connecting of waypoints in an example environment. As these waypoints do not cover the whole area of possible positions, start and end vertices of a search are determined by finding the closest waypoint that has a clear line of sight to a required position (the start or goal positions).

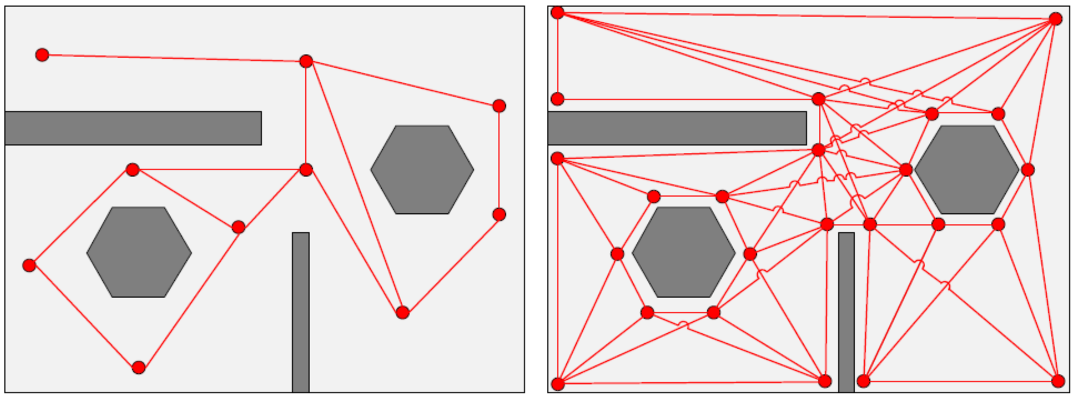

a) A waypoint based navigational graph created by hand placing waypoints throughout the game environment.
b) A waypoint based navigational graph automatically constructed using a point of visibility approach.

FIGURE 2.3: Waypoint based representation of a navigation graph

Waypoint positions can be assigned manually by designers. However, there are many techniques to automate the creation of waypoints by exploring the 3D level. Those automatic methods tend to have a high computational cost which limits their computation to pre-processing (offline) roles [Rabin, 2000b]. The roadmap is one of the most popular techniques used to solve high dimensional pathfinding problems in the area of robotics. This technique includes a set of connected points that are drawn from a given map. There are many varieties of roadmap techniques. The Probabilistic Roadmap (PRM)[Kavraki and Latombe, 1994] is one of the most well-known roadmap techniques. RPM is generated by randomly sampling a configuration area to create a practical connected graph for traveling through a region. Reachability Roadmap (RRM) [Geraerts and Overmars, 2005] is another popular roadmap method. RRM uses first a grid tessellation and then obtains waypoints from the generated grid. Voronoi Diagrams [Abraham et al., 2010] are a variety of roadmap techniques. In this approach, a mesh of edges is created which are all equidistant from the two closest obstacles and the vertices of the generated network are located at the intersections of those edges.

One of the disadvantages of waypoint-based graphs is that they cannot guarantee to provide a full coverage of the entire environment (due to human errors during the waypoints placement) and it may also include a large number of redundant Waypoints, which are unnecessary and increases the overall search space.

2.6.3 Mesh Based Navigation Graphs

The majority of modern video games create navigation graphs using polygonal navigation meshes that can be compute automatically from a given geometry [Mononen, 2009; Johnson, 2006; Demyen and Buro, 2006; Hamm, 2008; Rabin, 2014]. Navigation mesh (navmesh) methods create a graph that minimizes the number of navigation vertices required to represent a world environment while ensuring near-perfect coverage of the traversable environment. Generally, A navigation mesh can be considered as a low-fidelity representation of an operating environment consisting of convex contiguous polygons. Navigation meshes are usually applied in video games to represent traversable and non-traversable surfaces in two and three dimensions [Snook, 2000; Tozour, 2002]. Many techniques have been introduced to build navigation meshes. Some methods perform a triangulation of the environment[Demyen and Buro, 2006; Kallmann, 2010b]. While others perform a convex subdivision using polygons [Oliva and Pelechano, 2013,Mononen, 2009]. Navigation meshes based on convex polygons, can significantly decrease the number of vertices and thus the branching factor, leading to smaller search space graphs [Millington and Funge, 2009; DeLoura, 2001]. Van Toll et. al. presented a comparison of different types of navigation meshes to highlight the benefits and limitations of each type (e.g. grids, triangles, convex polygons, or overlapping circles)[Van Toll et al., 2016]. Navmesh based on convex polygons often provide a more accurate representation of the walkable areas because they can assure that polygon sides match the edges of the geometry representing the environment [Demyen and Buro, 2006]. Since carrying out the initial subdivision of the surface of a typical video game environment has a high computational cost, navmeshes are usually generated in offline mode [Rabin, 2014].

There are some popular and novel navigation mesh toolsets. Recast [Mononen, 2009] is one of the most well known and state of the art navigation mesh generator used in complex applications such as virtual simulation and video games development, and it is also employed in the Unity3D game engine [Van Toll et al., 2016]. Recast is an open-source, fast and also completely automatic toolset, which means that it is possible to launch a geometry at any level and get a robust generated navigation mesh. The Recast mesh navigation process begins by creating a voxel mold from a given level geometry and then calculating a navigation mesh over it. This process consists of three major steps, firstly, it constructs a voxel mold, which guarantees that the method can be robust against declines in the input model as well as simplifies the furniture; secondly, it partitions the mold into simple regions; and thirdly, it peels off the regions as simple polygons.

NEOGEN [Oliva and Pelechano, 2013] is also another novel automatic approach for generating near-optimal navigation meshes from 3D multi-layered virtual environments. Similarly to Recast, this method consists of three steps: (i) it first performs a GPU coarse voxelization, which is used to classify and extract the different walkable layers. (ii) Then it carries out a layer refinement phase, performing a high resolution render using the fragment shader to achieve a 2D floor plan of each layer. (iii) The final part is the Navmesh Generation, where a convex decomposition of each layer is calculated and layers are linked to generate a navigation mesh of the input geometry.

Polygonal navmeshes are interesting because they provide a complete representation of the environment, and they are typically more efficient in memory usage than other graph representations like grid maps [Van Toll et al., 2016]. The other advantage of polygonal navmeshes is their flexibility which can provide a hand-editing facility for game designers and offers them more control over the agent navigation.

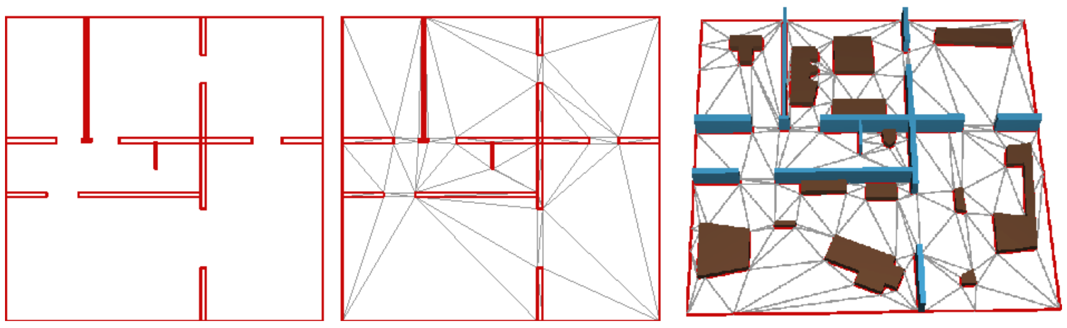


FIGURE 2.4: An example of a navigation mesh representation based on a triangulation of the walkable space [Kallmann, 2010b].

However, navigation meshes also have some disadvantages. One of the major disadvantages is the high computational cost of navmesh generation in dynamic environments. So that any changes and updates on the environment could affect a large number of mesh cells and thus the navmeshes could need to be completely rebuilt in the affected region. So, making dynamic updates on navmeshes is not a straight forward process and it may require to be fully recomputed. In general, most navmesh approaches like [Rabin, 2014], [Farnstrom, 2006] and [Hamm, 2008] are not ready to be used in dynamic environments due to their high processing costs and outcome delays in navmesh updates. There are some other subdivision methods such as [McAnlis and Stewart, 2008] and [Demyen and Buro, 2006] which claim suitability with regards to usage within dynamic game environments. The other disadvantage of the navmesh method is that the computed paths often need to be smoothed or post-processed because they use the polygon edges to compute paths. However, when edges are simply used to set attractors for the local movement algorithms (e.g. steering), it may not be needed as the local moment method can deal with smoothing the agents' trajectories when turning.

2.7 Search Algorithms

2.7.1 Dijkstra's Algorithm

Dijkstra's algorithm is a graph search algorithm introduced in 1959 [Dijkstra, 1959]. Considering a graph $G = (V, E)$ consisting of a set of vertices V and a set of edges $E \subset V \times V$, and for each edge $(u, v) \in E$ an associated positive cost $c(u, v)$, Dijkstra's algorithm can find the shortest path from a single source node $U_(start) \in V$ to all nodes in V. This algorithm has many different variants. The main algorithm was introduced as an algorithm to find the shortest paths from a start point in a map to all other points in a weighted graph. In such a graph, each edge weight shows the traversal cost incurred traveling across that edge. The basic steps of Dijkstra's algorithm are as follows: The algorithm searches every node or vertices in the given graph, and while doing so stores shortest paths information at each vertex. Once the algorithm has finished, a path is then created by starting at the goal node and working backward towards the start node using the paths information stored at each node. Dijkstra's algorithm performs iteratively and each iteration will search (another common term encountered is expand) a single node. The node being searched is referred to as the parent node for that iteration and its neighboring nodes are referred to as the node successors (or child nodes).

The Algorithm

Dijkstra's algorithm uses a single metric, the cost-so-far (CSF) value, in discovering the shortest paths in the graph [Anguelov, 2012]. The CSF value is simply the traversal cost acquired in traveling to a graph vertex from the start vertex. Dijkstra's algorithm calculates a CSF value for each of the successor nodes when a node in the graph is explored. The CSF value for each successor is the sum of the parent's CSF value and the traversal cost of traveling from the parent to the successor. Figure 2.5-a shows the CSF calculation for a node in the given graph. Multiple paths can exist to a single graph node, meaning that when Dijkstra's algorithm explores the graph it may face successors that already have CSF values (i.e. a previous path to the node has been found). In this case, Dijkstra's algorithm checks whether the current path (from the current parent node) to a successor node is shorter than the previous path found. A new CSF value is then calculated from the current parent to the specific successor node. If the new CSF value is smaller than the successor node's CSF value, it means that the current parent represents a shorter path to that successor than the previous path found. The successor node's CSF is then set to the new smaller CSF value (i.e. the new shorter route). This guarantees that only the shortest paths found to each node are stored. In addition to the CSF value stored at each node, a link to the parent node, from which the CSF value originated from, is stored as well. This parent node link is required so that paths can be followed back from each node to the start node. This parent node link is updated whenever the CSF value of that node is updated. Simply put, whenever a new shorter path to a node is found, the shorter path value is stored as well as the node from which that path originated from. During the exploration of the search space, Dijkstra's algorithm will face nodes that classify into the three unseen, unexplored and explored nodes categories. In case that an unseen node is first faced during the exploration of a parent node to that node, a CSF value originating from the parent node is calculated [Harabor, 2014].

The unseen node will now require to be explored; it is stored on a list which contains all nodes that the algorithm has faced but not explored yet. This list is known

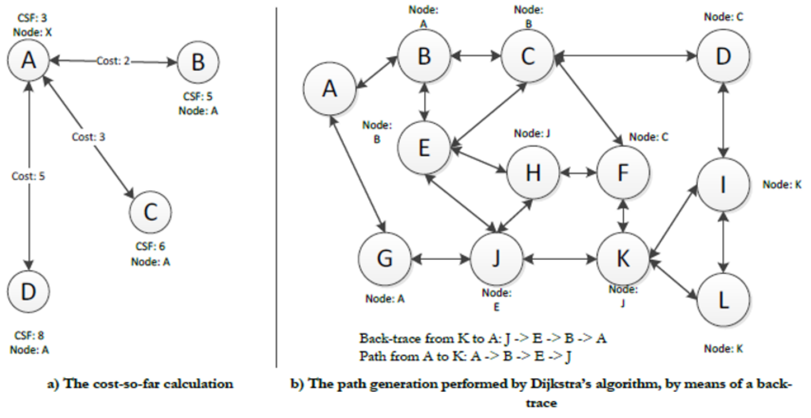

CSF: 3
Node: X
A
Cost: 2
B
CSF: 5
Node: A
Cost: 3
Cost: 5
C
CSF: 6
Node: A
D
CSF: 8
Node: A
a) The cost-so-far calculation
Node: A
Node: B
Node: C
A
B
C
D
Node: B
E
Node: J
Node: C
H
F
I
Node: K
G
J
K
L
Node: A
Node: E
Node: J
Back-trace from K to A: J -> E -> B -> A
Path from A to K: A -> B -> E -> J
Node: K
b) The path generation performed by Dijkstra's algorithm, by means of a back-trace

FIGURE 2.5: The Dijkstra's algorithm's cost-so-far calculation and path generation [Anguelov, 2012].

as the open list; nodes on the open list are all awaiting exploration. When the algorithm explores a node, it is inserted into a list which includes all other explored nodes. This list is known as the closed list. The open and closed specification is based on the fact that when a node is faced, it is opened for exploration and it is only closed once the node has been explored. Dijkstra's algorithm is initialized with just the start node present on the open list and an empty closed list. At each iteration of the algorithm, the node which has the smallest CFS value is eliminated from the open list. This routine guarantees that the nearest node to the start node is always explored. Once node N is explored, each of N's successor nodes S is considered. Each S can refer to one of three unseen, open or closed categories. If S is unseen then a CSF value will calculate directly from N, S's parent link is set to N and finally, S is located on the open list. In the case when S has already been checked(i.e. either an open or closed node), the new path to S from the start node (the CSF value originating from N) is compared to S's existing CSF value, if the new path is shorter than the existing path then S is updated with the new path's CSF value and its parent node link is set to N. Dijkstra's algorithm terminates when the open list is empty, i.e. all the nodes in the search space graph have been explored. When the algorithm has terminated, a path can be created from the start node to any node (the goal node) in the graph by starting creating a path at the goal node and following the stored parent node links back to the start node (see Figure 2.5-b).

2.7.2 The A* Algorithm

Despite the speed of Dijkstra's algorithm, there are some cases in which it is suitable to optimize the performance of finding the shortest paths (for instance when the search space is very large). Dijkstra's algorithm finds the shortest paths to all nodes in the graph, but often one is only interested in the shortest path to a specific goal node $U_(goal)$. The A^* algorithm was introduced by Peter E. Hart, Nils Nilsson, and Bertram Raphael in 1968. A^* algorithm [Hart, Nilsson, and Raphael, 1968] adjusts the search in the graph to move quicker towards the goal node, whereas in Dijkstra's algorithm the shortest paths distances are propagated breadth-wise. However, the A^* algorithm guarantees to find the optimal path from the start node to the goal node in the search space graph. Figure 2.6 and Figure 2.7 illustrated the search space exploration pattern of both Dijkstra and A^* algorithms. In this case, the A^* algorithm, is clearly favorable because it needs to explore a smaller number of nodes to find a solution. In autonomous agent simulation, the A^* is a pathfinding algorithm that is used to navigate an autonomous agent to find a path between two locations in a navigation mesh. Due to the high performance and accuracy of this algorithm, it is widely used in the computer simulation field. As mentioned before, this algorithm is a generalization of the Dijkstra's algorithm, but with better search performance by using meta-heuristic methods.

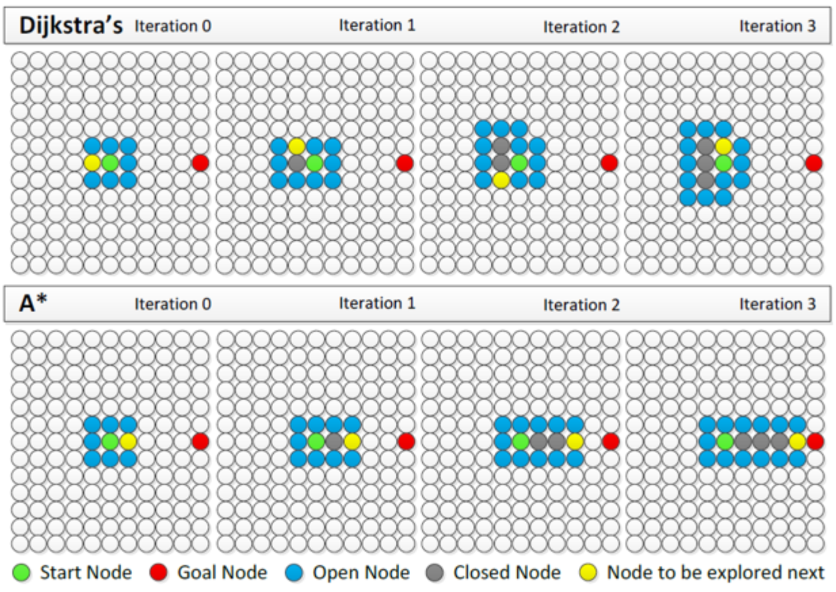


FIGURE 2.6: Node selection for Dijkstra's algorithm compared to A^* for an example search [Anguelov, 2012]

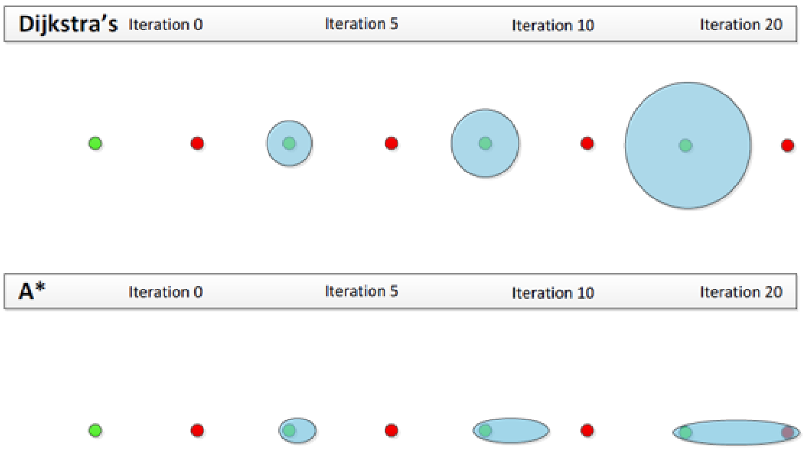


FIGURE 2.7: Overall search space exploration of Dijkstra's algorithm compared to A^* [Anguelov, 2012]

The Algorithm

The A^* algorithm uses the best-first search routine and finds the shortest path between two given start and end nodes. This method evaluates the cost of reaching a node n from a start node by combining $g(n)$ (the cost of getting to the node n) and $h(n)$ the Heuristic or estimated cost of reaching the end node from n as:

$$f(n) = g(n) + h(n) \tag{2.1}$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the shortest path from n to the goal. The heuristic cost is an estimation of how close a given node is to the goal node, or alternatively an estimation of the likelihood of a node leading to the goal. The heuristic value is calculated by a heuristic function that, given two nodes, returns a numeric measure of how close the nodes are together. A simplistic way of describing the heuristic value is to term it the "estimated remaining cost".

A^* algorithm terminates once the path it selects to extend is a route from the start node to the goal node or if there are no paths favorable to be extended. The heuristic function of the A^* algorithm is problem-specific. If the heuristic function is admissible, meaning that the A^* algorithm never overestimates the exact cost to get to the destination node, the A^* is guaranteed to obtain the lowest-cost path from the start node to the goal node in the search space graph. Generally, implementations of A^* algorithm use a priority queue which is known as the open list to perform the iterated choice of minimum (approximated) cost nodes to expand. At each level of the algorithm, the node with the cheapest $f(x)$ value is eliminated from the open list queue, the f and g values of its neighbors are updated, and these neighbors are added to the queue. The A^* algorithm continues to search when a goal node on the search space graph has a lower f value than any other node in the queue (or until the queue is empty). Finally, the f value of the goal node will be the measured cost of the shortest path, since h value at the goal node will be zero in an admissible heuristic function. The A^* algorithm explained so far simply provides the length of the shortest path. To find the exact sequence of steps, the algorithm can be easily updated so that each node on the path keeps track of its predecessor. After this algorithm is run, the end node will point to its predecessor, and so on, until some node's predecessor is the start node. Figure 2.8 shows the pseudocode of A^*.

2.7.3 ARA^*

ARA^* (Anytime Repairing A^*) algorithm was first proposed by Maxim Likhachev, Geof Gordon and Sebastien Thrun [Likhachev, Gordon, and Thrun, 2003] to offer an alternative to the rather computational expensive generic A^* algorithm. As describe by Likhachev et al. [Likhachev, Gordon, and Thrun, 2004] this algorithm performs " [...] an efficient **anytime heuristic**[1] search that [......] runs A^* with **inflated heuristic** [2] in succession [and] **reuses search efforts** [3] from previous executions in such a way that the sub-optimal bounds are still satisfied". From the aforementioned definition, some salient points that characterizes the ARA^* can be drawn accordingly.

To start, ARA^* is an **anytime algorithm**[1], in the sense that it is a computational algorithm that can return valid solutions to problems regardless of interruptions between start and end. ARA^* functions by computing sub-optimal solutions to problems which improves with more run-time, therefore it offers a trade-off between computational time and quality of algorithmic solutions.

As indicated in the previous section, A^* returns optimal solutions albeit a consistent heuristic $h(n)$. ARA^* therefore seeks to **inflate this heuristic**[2] by an inflation factor ε, $\forall \varepsilon \leq 1$, saving search times by expanding fewer nodes. Using inflated heuristics provides sub-optimal solutions but proofs to be fast and most importantly the sub-optimality is bounded by ε, which enables a tuning of the trade-off between

```
Main()
    $open := closed := \emptyset$;
    $g(s_{start}) := 0$;
    $parent(s_{start}) := s_{start}$;
    $open.Insert(s_{start}, g(s_{start}) + h(s_{start}))$;
    while open $\neq \emptyset$ do
        $s := open.Pop()$;
        if $s = s_{goal}$ then
            return "path found";
        $closed := closed \cup \{s\}$;
        foreach $s' \in nghbr_{vis}(s)$ do
            if $s' \notin$ closed then
                if $s' \notin$ open then
                    $g(s') := \infty$;
                    $parent(s') := NULL$;
                UpdateVertex$(s, s')$;
    return "no path found";
end
UpdateVertex(s, s')
    $g_{old} := g(s')$;
    ComputeCost$(s, s')$;
    if g$(s') < g_{old}$ then
        if $s' \in$ open then
            $open.Remove(s')$;
        $open.Insert(s', g(s') + h(s'))$;
end
ComputeCost(s, s')
    /* Path 1 */
    if g$(s) + c(s, s') <$ g$(s')$ then
        $parent(s') := s$;
        $g(s') := g(s) + c(s, s')$;
end
```

FIGURE 2.8: Pseudocode of the A^* algorithm.

computational time and quality of the solution by manipulating the sub-optimal bounds.ARA^* basically works by executing A* multiple times, tuning ε from a set value $\varepsilon \leq 1$ till $\varepsilon = 1$. The algorithm can therefore be described as [Likhachev, Gordon, and Thrun, 2004]:

$$f(n) = g(n) + \varepsilon * h(n) \tag{2.2}$$

As highlighted in the definition of ARA^*, there is the **reuse of search effort**[3] in execution of the algorithm which results in a faster algorithm. In the execution of multiple A^* with appropriate tuning of ε, the algorithm avoids repetition of execution cycles by reusing results from previous searches to improve search time. To illustrate this procedure, the concept of **locally inconsistency** of nodes is introduced, this describes the instance after a node's g-value is decreased and until the next time they are expanded. By modifying the A^* algorithm, ARA^* introduces a third list, the incons; besides the open-list and the closed list of the A^* algorithm, the open list which is a **locally inconsistent** node stores a lowered g-value of a node until it is subsequently expanded and put into a closed list. Therefore the open list only contains nodes that are not expanded yet, the ARA^* with the use of the augmented list; incons, stores locally inconsistent nodes that have been expanded in previous executions, serving as a cache of expanded nodes for use in subsequent search iterations.

The minimum between ε and the ratio between $f(n_{start})$ and the lowest non-weighted f-value of all locally inconsistent nodes .i.e the sub-optimality bound (ε^1) is thus given as:

$$\varepsilon^1 = min(\varepsilon, \frac{f(goal)}{min_{n \in OUI}(g(n) + h(n))}) \tag{2.3}$$

2.7.4 D^*

In most of the path planning work in real world scenarios, the robot or agent would initially have an incomplete and inaccurate graph of the environment model to plan on. The graph takes continuously and frequently changes as time passes and as the agent moves. In this case, the calculated path may become wrong or sub-optimal. It can be then costly to plan from scratch using A* to maintain validity and optimality every time a change occurs on the search graph, particularly in large and complex environments with a big number of nodes. Moreover, the updates on the graph might not even affect the current path, or simply affect slightly its optimality, and therefore the path could be easily fixed without a complete re-computation. In these situations, fixing the path or re-planning is much more logical than starting planning from scratch [Khattab, 2018]. The D* algorithm (Dynamic version of A*) [Stentz, 1993] can plan optimal traverses in real-time by incrementally repairing paths to the agent's state as new knowledge is discovered.

The D^* algorithm, first proposed by Anthony Stentz, was initially conceived as an optimal pathfinding algorithm which could enable robots to navigate through environments in which they have little to no information. The D^* closely resembles the A^* algorithm, but differs by being dynamic, involving a problem solved where edge path cost parameters change during processing [Stentz, 1993].
D^* primarily differs from the A^* by propagating information backwards. Hence information is propagated towards the goal node and ends at the robot's position (start node) or with the open list empty. With a non heuristic function ***h*** and ***n*** nodes to the goal, the path cost function is given as ***h(n)***. D^* functions by using both the closed and open lists as done in the A^*. Nodes are subsequently distinguished with tags $t(n)$, such that a node that has never been in the open list, is labelled, NEW ($t(n) = NEW$), nodes currently in the open list as $t(n) = OPEN$ and $t(n) = CLOSED$, if the node is out of the open list. When $t(n) = OPEN$, nodes are sorted by a key function $k(n)$, which defines a minimum of $h(n)$, the path cost function before modification. Per the state of the node, *OPEN, CLOSED, NEW*, the key function value $k(n)$ is described as:

$$k(n) = \begin{cases} \boldsymbol{h(new)}, & \text{if } t(n) = NEW \\ \boldsymbol{min(k(n),h(new))}, & \text{if } t(n) = OPEN \\ \boldsymbol{min(h(n),h(new))}, & \text{if } t(n) = CLOSED \end{cases} \tag{2.4}$$

With the use of the Key function $k(n)$, each node in the open list is put in classes of **Raised**, or **Lower** states, where the former propagates information about path cost increases, for example about an increase in edge cost and the latter involves the propagation of information concerning path cost decreases, therefore the conditions of classification is such that; **LOWER**,$\forall k(n) = h(n)$ & **Raised**,$\forall k(n) < h(n)$. From the collection of nodes in the open list, the lowest key function value is used as a benchmark for optimality, if path costs are lower or equal to the lowest key function

k_{min}, the path cost is deemed optimal, whilst path costs greater than k_{min} are not guaranteed to be optimal. Through repeated removal of nodes from the open list, information is propagated. Once they are removed from the open list, cost information is expanded to neighbouring nodes, which are then placed in the open list. the iterative process continues[Stentz, 1993], with the prior k_{min} relegated to a k_{old}.

The D^* algorithm has two main functions; the PROCESS-STATE and MODIFY-COST functions. The PROCESS-STATE function which is called repetitively, determines the optimal path to the goal and the function MODIFY-COST, thereafter effects updates on edge cost functions due to a change in edge costs $c(n, n^{'})$ and then eventually it puts the updated nodes into the open list.

2.7.5 *Theta**

Introduced by Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner [Nash et al., 2007], *Theta** is an any-angle path-planning algorithm, to put it simple, it allows path directions in any angle. There are basically two types, the Basic *Theta** and Angle-propagation *Theta** which proofs to have desirable properties as a better search algorithm to the traditional A^* (see Figure 2.9).

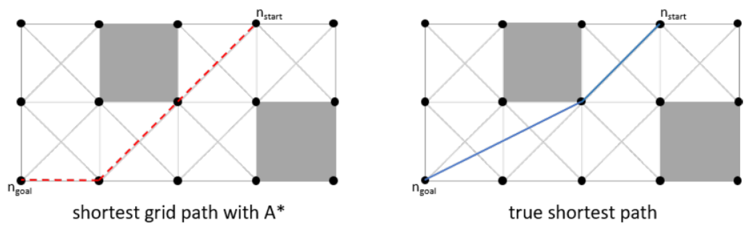


FIGURE 2.9: A^* path vs any-angle path [Daniel et al., 2010]

Basic *Theta** is simple to implement and understand, due to its similarities to the fundamental A^*. It propagates information along edges of grids, avoiding putting constraints on paths to be formed by graph edges [Daniel et al., 2010]. Whilst the A^* only considers paths along the grid edges from n_{start} to n, and n to $n^{'}$ as shown in figure 2.10 (**Path1**-A^* & **Path2**-Basic *Theta**), the Basic *Theta** updates the g-value and parents of unexpanded neighbouring nodes $n^{'}$ of n when expanding n such that:

$$g(n^1) = g(n) + c(n, n^1) \tag{2.5}$$

Paths from n_{start} to the parent of n, parent(n) and from parent of n to n^1 in a straight line are also taken into consideration, see **path2** of figure 8 .

$$g(n^1) = g(parent(n) + c(parent(n), n^1) \tag{2.6}$$

From figure 2.10 , path 2 provides a minimum path compared to path 1 and thus, is chosen by *Theta** when there exists a line-of-sight between n^1 and parent(n)(if no blocking is present). In the event of blocking path 2 is chosen.

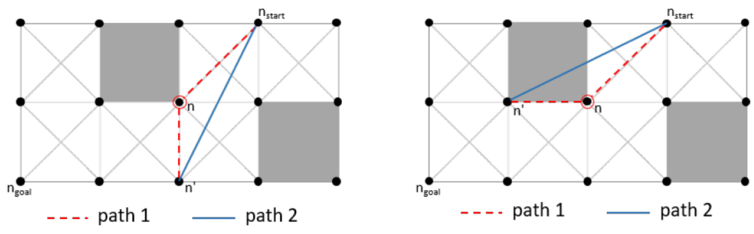


FIGURE 2.10: Paths considered by Basic *Theta** [Daniel et al., 2010]

Figure 2.11 shows an example trace of Basic *Theta**. non-relevant extraction of nodes are not shown for purposes simplicity. Red circles shows nods which are currently being extracted.

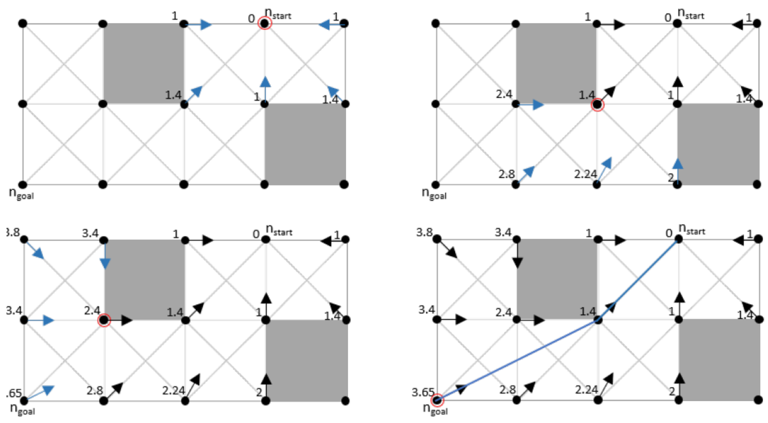


FIGURE 2.11: Example of Basic *Theta** [Daniel et al., 2010]

Basic *theta** can be described as not optimal, stemming from the fact that the parent of a vertex has to be either a visible neighbor of the vertex or the parent of a visible neighbor, but is correct**1** and complete**2**, in the sense that the algorithm finds unblocked paths from the start node to the goal node (correct[1]) and finds paths that are unblocked (complete[2]), but shortest path is not guaranteed [Nash, 2010].

Angle-Propagation *Theta** reduces run time of the Basic *Theta** node expansion from linear to constant. The contrast from the Basic *Theta** and the Angle-propagation *Theta** is that the AP *Theta** propagates angle ranges, determining whether two nodes have a line-of-sight. AP *theta** determines angle range of a node when it expands the node and subsequently propagates it along grid edges which results in a constant run time per vertex expansion. Due to the propagation of the angle ranges in constant run time, the line-of-sight checks are also in constant time [Daniel et al., 2010].
AP *Theta** exhibits the same properties of being complete and correct with no guarantee of finding shortest path. There are also occasional unnecessary heading changes.

2.7.6 D^* Lite

D^*Lite , a simpler and shorter version of the D^* is a heuristic search algorithm, first introduced by Koenig and Likhachev to aid robot navigate in unknown environments. This approach reuses information from previous searches to find solutions for successive similar searches [Koenig and Likhachev, 2002a], avoiding beginning from start at every search, thereby decreasing run time. D^*Lite is built on the life long planing A^* [Koenig and Likhachev, 2002b] which is an incremental version of the A^*, that involves a finite search problem on known edges in which costs increases or decreases over time. D^*Lite are desirable for implementation of optimization problems with inadmissible heuristics [Koenig and Likhachev, 2002c].

In D^*lite, performs the computation of two estimates for each node, the g-value (objective function value) and the rhs estimates, which is a one step lookahead of the path cost based on g-values of its successors. Based on the value of the g-values and the rhs, the node can be described as either consistent or inconsistent, where inconsistent nodes are placed in an open list, prioritized by their key value. Therefore given a successor (n^1) and a predecessor node (n), a directed edge can be established between n and n^1 with an associated edge cost c(n,n^1) [Choset, 2007]. With the associated edge cost, the rhs-value of node n, *rhs(n)* can then be computed with its g-value, *g(n)*:

$$rhs(n) = \min_{n^1 \in Succ(n)} (g(n^1) + c(n, n^1) \tag{2.7}$$

such that consistency is thus determined by

$$\begin{cases} \textbf{\textit{Consistent,}} & \text{if } g(n) = rhs(n) \\ \textbf{\textit{Inconsistent,}} & \text{if } g(n) > rhs(n) \\ \textbf{\textit{Under-consistent,}} & \text{if } g(n) < rhs(n) \end{cases} \tag{2.8}$$

A key value $k(n)$ of a node, n is determined as the minimum of its $g(n)$, $rhs(n)$, a heuristic term h, and a factor k_m which helps in avoidance of reordering anytime there is a change in the start node as shown below. The equation comprises of a primary and secondary part, with the secondary used in case of a tie breaker:

$$k(n) = \Big(min(g(n), rhs(n)) + h(n_{start}) + k_m)^{\text{PRIMARY}}; \\ (min(g(n), rhs(n))^{\text{SECONDARY}} \Big)$$

where $h(n^1, n)$ is non-negative and backward consistent [Choset, 2007]. In the event of a change from one node to another, say (n to n^1), there is an evaluation of new key values $k(n)$ such that movements from node n to n^1 may cause primary key elements from the equation to decrease by $h(n, n^1)$ or primary keys with $h(n, n^1)$ may be too low compared to key-values previously placed in the open list, therefore the factor k_m is added to augment such changes that may occur.

2.7.7 AD^*

Anytime Dynamic AD^* can be thought of as a hybrid between the ARA^* and the D^*Lite algorithms, both of which have been explained earlier. It basically involves a heuristic-based anytime algorithm presented in by Maxim Likhachev, Dave Ferguson, Geoff Gordon and Sebastian Thrun which bridges algorithms in a dynamic environment (e.g D^*Lite) and handling of complex planning problems(ARA^*) [Likhachev et al., 2005].

AD^* capitalizes on benefits of sub-optimality in short run time associated with anytime algorithms from a backward version of of the anytime ARA^* algorithm and re-planning characteristics of the D^*Lite which offers the ability of a dynamic environment.

As stated before, ARA^* seeks to compute sub optimal solutions with an inflation factor $\varepsilon > 1$ to the problem of pathfinding that improves with time, reusing information from past searches to decrease run time. The AD^* algorithm uses a backward version of the ARA^*, with the goal node (n_{goal} placed in the open list instead of the start node (n_{start}). Instead of computing distance from n to n_{goal} edges, an estimate of the distance between a particular node n and the start node n_{start} is evaluated.

With Partial or little information about dynamic environments, the dynamic D^*Lite comes in handy in this union of algorithms. It ensures re-planning of found paths by reusing information from previous searches when variations in edge costs are detected [Likhachev et al., 2005]. A combination of these properties provides a platform for applications such as games, which predominantly require fast computation (ARA^*) in partially known environments (D^*Lite) .

The AD^* utilises the concept of consistency of nodes in the D^*Lite by also calculating the g(n) and rhs(n) of each node n, but instead of using only the open lists, the algorithm incorporates the use of the additional lists of the open and icons list from the ARA^*. The algorithm computes the key value $k(n)$ for each node n which are put into classes of *over-consistent* and *under-consistent* [Likhachev et al., 2005].

$$k(n) = \begin{cases} \textbf{\textit{(rhs(n)+}}\varepsilon\textbf{\textit{*h(}}n_{start}\textbf{\textit{,n); rhs(n)),}} & \text{if n} = \textit{Over-consistent} \\ \textbf{\textit{(g(n)+h(}}n_{start}\textbf{\textit{,n);g(n)),}} & \text{if n} = \textit{Under-consistent} \end{cases} \tag{2.9}$$

Nodes that are designated as inconsistent are placed in open lists and are thereafter moved to the closed list once they are expanded. Inconsistent nodes that are already in the closed list are moved to the incons list. Note that for calculated under-consistent nodes to propagate changes in cost to affected neighbours, keys are calculated using a non-inflated heuristic [Likhachev et al., 2005].

2.7.8 Field D^*

Optimal grid planning, usually for navigation purposes provides sub optimal solutions to pathfinding problems given that most grid based planning utilises discrete state transitions that restricts an agent's movement to a small set of possible headings [Ferguson and Stentz, 2005]. Field D^* introduced by Dave Ferguson and Anthony Stentz, is an interpolation based process that determines comparatively better estimates of path costs which are globally smooth paths.

Attempts have been made over the years for finding algorithms well suited for grid-based path planing, for example the Dijkstra's algorithm, the A^* as well as well known variants of the A^* such as the incremental A^* and the D^*Lite have been used, but these algorithms are constrained by a limited number of discrete set of possible transitions permitted between grid sets [Ferguson and Stentz, 2005].
As an extension of the D^* and the D^*Lite, the Field D^* approximates path costs through interpolation which is important since classical grid based methods offers transitions which are only possible in a straight line from one node to the other, hence the Field D^* providing a relaxation to this limitations. Field D^* utilises a different grid layout of nodes such that, grid nodes are assigned to corners of cells instead of centers as shown in the figure 2.12.

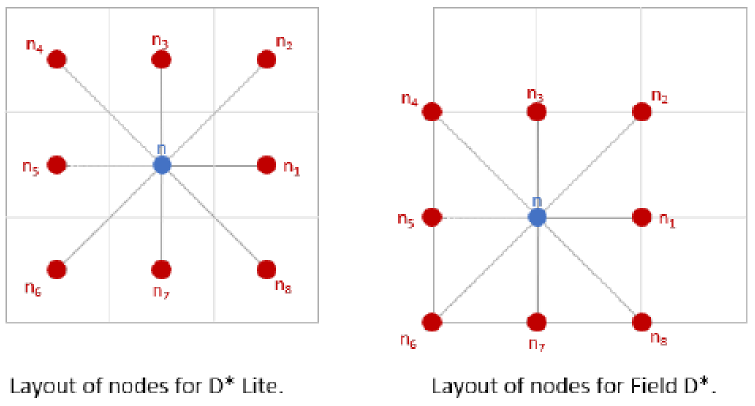


FIGURE 2.12: Layout of Nodes [Ferguson and Stentz, 2005]

The path costs are then subsequently calculated through interpolations. Generically with a node n and a set of its neighbours n_{NE_n}, such that each neighbour $n^1 \in$

n_{NE_n} and with the cost of traversing from a node n to a neighbour $n^1 \in n_{NE_n}$ given as $c(n, n^1)$, the path cost is evaluated as:

$$g(n) = \min_{n^1 \in n_{NE_n}} (g(n^1) + c(n, n^1)) \tag{2.10}$$

Equation (10) leads to a sub optimal solution which results in unnatural results, as movements are restricted to grid edges form one node to a neighbouring node. Field D^* relaxes the condition of being restricted to neighbours by considering optimal path from node n to all points on the boundaries of a grid cell, $n_1 \in n_{Nb_n}$. Hence with a knowledge of the value of every point on the boundary, the $g - value$ of a node n to a boundary point n_1 can calculated by:

$$g(n) = \min_{n_1 \in n_{Nb_n}} (g(n_1) + c(n, n_1)) \tag{2.11}$$

But n_{Nb_n} is an infinite set, it is impossible to calculate an infinite set of costs, hence an interpolation based method is employed such that there is an interpolation for values of g(n_1), .i.e points on the boundary, of two edge points n_x and n_y, given as:

$$g(n_y) = y(g(n_2) + (1 - y)(g(n_1)) \tag{2.12}$$

y is the distance between n_1 and n_y as shown in the figure below. The interpolated value is then used in the minimization problem to find the optimal cost between the nodes

$$g(n) = \min_{x,y}(bx + c\sqrt{(1 - x)^2 + y^2} + yg(n_2) + (1 - y)g(n_1) \tag{2.13}$$

Explained with figure 2.13, $x \in [0, 1]$ is the travelling distance on the lower grid edge from n before traversing through the grid cell to reach n_y, travelling a distance $y \in [0, 1]$ from n_1. When the optimization solution is $\{x, y\} = \{0, 0\}$, path is along the bottom edge, but the cost is computed from the path direction through the center.

(x^*, y^*) is thus the solution of the minimization problem, with either of the solutions taking a binary value due to the interpolations. From the illustrations of travels in grid cells as shown in figure 2.13, if the cost of travelling around boundaries from n to n_2 along edges is more expensive than transverse even partially through the center of the cell, then it makes complete sense to assume that a complete central crossing will have the least cost, the proof of this is extensively done in [Ferguson and Stentz, 2005]. Hence a complete movement through the centre as shown in the figure is possible with $y^* = 1$, and $x^* = 0$, whilst conversely with a complete movement along the edges from n to n_2, $\{x^*, y^*\} = \{1, 0\}$.

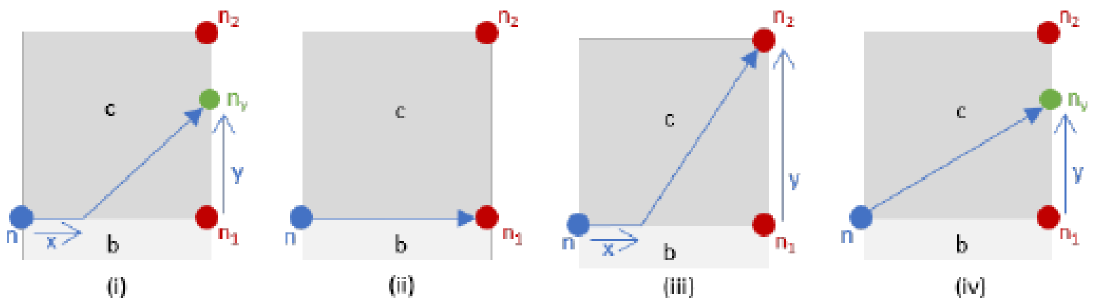


FIGURE 2.13: Shortest path of n through edge n_1 n_2 [Ferguson and Stentz, 2005]

Taking that the cost from i node to a j node to be $f(n_i, n_j) = g(n_i) - g(n_j)$, and considering that $f(n_i, n_j) < 0 \; \forall g(n_i) < g(n_j)$, then this path results in the cheapest path. From here the illustrations from figure 2.13 will be used to describe this algorithm. Assuming movement from n to n_2 with minimum path cost and assigning b and c as the cost of going through the bottom edge of the cell and complete movement through the center respectively, if $f < 0$ from $f = g(n_1) - g(n_2)$ then it is cheaper to go through the bottom edge first to n_1 before going from n_1 to n_2, so that $g(n) = min(c, b) + g(n_1)$, in this case b is the cheapest option from the figure. But in the case where $f < b$, it is cheaper to traverse through the center of the grid cell for the smallest cost as shown in figure 2.13-(IV).

To formulate the path cost of n, we assume, $f = b$. In this case the cost of a path through a part of the bottom edge, figure 2.13-(iii), will be the same as the cost of using the bottom edge, figure 2.13-(IV). By taking the case that none of the bottom edge, figure 2.13-(iii), is used, we solve for an optimal y^* (since y corresponds to the right edge) that minimizes the cost path such that, with $k = f = b$ the cost is given as:

$$c\sqrt{1 + y^2} + k(1 - y) + g(n_2) \tag{2.14}$$

The equation is solved for the minimum value y^* by differentiating the function and making it equal to zero, thus the optimal y^*:

$$y^* = \sqrt{\frac{k^2}{c^2 - k^2}} \tag{2.15}$$

Therefore from equation (2.14) and taking into account that b is the cost of the bottom edge as stated before; If $f < b$ the right edge is used with path cost calculated with the equation such that $k = f$. Also if $f > b$ the bottom edges are used with $k = b$ and $y^* = 1 - x*$ substituted into the equation.

2.8 Hierarchical Search Algorithms

Regardless of the search algorithm used from the ones described in the previous section, it is possible to further optimize processing time and memory cost by using a hierarchical approach combined with the search algorithm.

There are many hierarchical approaches in the literature, for example the hierarchical pathfinding for grid maps (HPA*) [Botea, Müller, and Schaeffer, 2004], the dynamic HPA*, (DHPA*) [Kring, Champandard, and Samarin, 2010], the partial refinement A* [Sturtevant and Buro, 2005] algorithm and the minimal memory (MM) [Sturtevant, 2007]. They all provide solutions with a reduced processing as a trade against optimality and increased algorithmic memory cost [Jurney and Hubick, 2007]. They undertake search procedures by organising problems as connected sub-problems which are then solved individually with an end result similar to the former parent problem. The process of splitting the primary problem is known as *Problem Subdivision*.

2.8.1 Problem Sub-division

In problem subdivision, the primary problem is decomposed into smaller sub-problems that have some connection which results in a segmented but related sub-problems which are solved resulting in a solution similar to the primary algorithm [Botea, Müller, and Schaeffer, 2004]. This provides the advantage of minimizing the processing cost of discreet searches. The connection of the segmented problems is such that the start and goal nodes of each sub-problem; the pair referred to as sub-goals, are connected in series i.e. the goal of one sub-problem is the start node of the successive sub problem. The resultant less demanding sub-problems offers individually, easier problems which requires less processing and memory costs compared to the initial problem even when considering accumulated processing demands from all the sub-problems combined [Botea, Müller, and Schaeffer, 2004].

From figures 2.14(a-c), the process of splitting an algorithmic problem is presented, showing the path cost ramifications of the segmentation procedure. Figure 2.14-a shows the search space for the overall algorithm using a discrete A*, which results in 50% of the total search space explored. The original search space from (figure 2.14-a) is segmented into 4 smaller sub problems as shown in figure 2.14-c. The reduced individual space of the sub problems offers a less computational demanding problem (reduced processing and memory cost) forming an overall decreased search space as shown in 2.14-b. Figure 2.14-b shows the total exploration space formed from the splitting process into sub problems. Figure 2.14-c shows the reduced search area as compared to 2.14-a.

Problem sub division also enhances savings in memory costs; each sub-problem is "self-contained". With each problem solved independently with the discrete search algorithm, memory is freed after each sub-problem is solved. Therefore the peak memory usage with the problem in sub-division procedure is equivalent to the maximum memory usage when solving a segmented problem.

A sub-optimal solution is found with sub-division as a result of selection of sub-goals. A set of solutions from individual sub-goals results in a sub-optimal solution, which will thus be optimal if the resultant selection lie on the optimal path as produced by the main problem, but finding an optimal path is impossible to say the least as expensive optimal graph search must be performed. This would invariably defeats the purpose of splitting the problem in the first place.

Hierarchical approaches thus uses the sub-division process by considering a base navgraph, from which a hierarchy of abstract graphs are produced, such that these

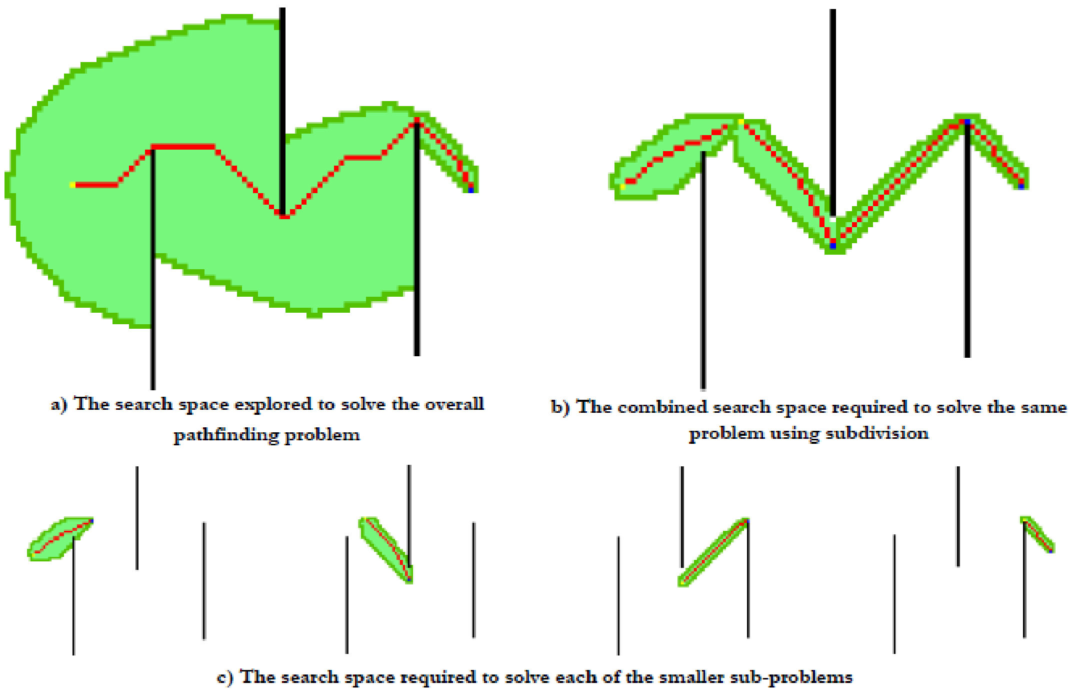

a) The search space explored to solve the overall pathfinding problem
b) The combined search space required to solve the same problem using subdivision
c) The search space required to solve each of the smaller sub-problems

FIGURE 2.14: The subdivision of a large pathfinding problem into smaller sub-problems [Ferguson and Stentz, 2005]

abstract graphs helps in the segmentation of a problem in a cheaper and faster manner.

As explained in the problem sub-division process, the resultant optimal search path are determined by the selection of sub-goals, hence the hierarchical approaches use abstract nodes (sub-goals), which are formed from a knowledge of an environment. These abstract nodes are then interconnected to form abstract graphs from which a search is performed to obtain an optimal path.
The hierarchical approaches involves three primary stages, **The build stage**, **The abstract planning stage** and **the path refinement stage**. The approach involves creating sub-goals from a known environment at the *build stage*, then the *abstract planning stage* creates an abstract path from the abstract graph, resulting in the selection of specific sub-goals to be solved. Finally at the *path refinement stage*, the selected sub-problems from the sub-goals are solved, with a final solution combined into a single low-level path.

2.8.2 The Abstraction Build Stage

Primarily, hierarchical approaches are categorised by the abstraction technique used in building an abstract graph. A number of techniques exists in literature[Botea, Müller, and Schaeffer, 2004][Sturtevant and Buro, 2005][Sturtevant, 2007], which uses different methodologies such as those based on environmental features which finally results in a single abstract node for an entire region as done in [Botea, Müller, and Schaeffer, 2004] and [Sturtevant, 2007] or the use of actual node topology in abstraction build as in the case of the Connection Hierarchies (CH) [Sturtevant and Geisberger, 2010a] methodology to construct abstract graphs from navgraphs offering overall reduced search paths containing a significantly lower number of nodes. But different techniques offer different processing costs which sometimes requires an offline build. For example in the case of procedures such as HPA* and CH.

The choice of the type of abstraction technique usually depends on the operation environment, for example when working in a dynamic environment, there is a parallel dynamic change in the abstract graph which occurs during run time. Therefore for any necessary alteration to the abstract graph there is a need for a short simplified build stage, therefore higher abstraction cost arises if complex abstraction techniques are selected.

For searches in very large environments, some techniques suggest a multi-level abstraction [Sturtevant and Geisberger, 2010a], that is to say, creating several layers of abstractions to reduce processing costs. But this may lead to the requirement of additional memory costs to store augmented abstraction layers which may sometimes have costs which may counter the advantages provided by the cost improvements associated with abstract graphs for splitting [Kring, Champandard, and Samarin, 2010][Botea, Müller, and Schaeffer, 2004].

But in [Sturtevant and Geisberger, 2010a], addition of abstraction layers is proven to be essential in some cases. In [Sturtevant and Geisberger, 2010a], a video game required a second abstraction layer to maintain pathfinding actions completion bounded in some specific time constraint. But in a dynamic environment, the inclusion of levels of abstractions leads to a further increase in cost as the environment changes, this results from all added abstraction level requiring checking and correcting as the environment changes.

Granularity in abstraction techniques describes the extent at which search spaces are reduced during abstractions. A coarse granularity e.g. *minimal memory abstraction* [Sturtevant, 2007] results in a drastic reduction of search paths. For example, considering a coarse granularity technique on an entire 16×16 navgraph, results in a single abstract node, but coarse granularity loses a lot of low-level details due to the severity of its search area reduction and may results in an inaccurate underlying graph.
A fine granularity conversely results in a comparatively smaller reduction of search space, this results in a more detailed abstract graph but building cost and search spaces are comparatively higher. Examples of fine granularity are the "clique" and "orphan" techniques.

2.8.3 The Abstract Search Stage

As discussed earlier, a set of sub-problems are produced from the abstract graph of the main navgraph for determining an optimal search path. With the abstract graph, a selection process of sub-goals from sub problems is undertaken for an abstract path from which a sub-optimal overall solution can be found.
For illustration of this process, we assume a single layer of abstraction. Considering the pair $start_{navgarph}$ and $goal_{navgarph}$ as the start and goal nodes of the primary problem (i.e. navgraph), we find the closest and reachable pair of start and goal nodes of the abstract $start_{abstract}$ and $goal_{abstract}$ through a series of discrete searches from the primary start and goal nodes on the navgraph.
An abstract path between the $start_{abstract}$ and the $goal_{abstract}$ found is then used to build an abstract path. Processing and memory cost are reduced with the abstract path process due to the comparatively smaller size of the abstract path to the original problem [Botea, Müller, and Schaeffer, 2004]. The resulting abstract path comprises of all sub-goals in the primary navgraphs from its start to goal, with $start_{navgarph}$ and $goal_{navgarph}$ inclusive.

2.8.4 Path Refinement

Finally, with the abstract path formulated from selection of sub-goals, the abstract path is refined into a low-level path. The edges between two abstract node are thereby refined into low level. Nodes making up these edges, are selected making up the start and goal node of each sub-problem. For this refinement, a discrete search is performed on the primary navgraph resulting in a partial path, which will be a part of the final low level path.

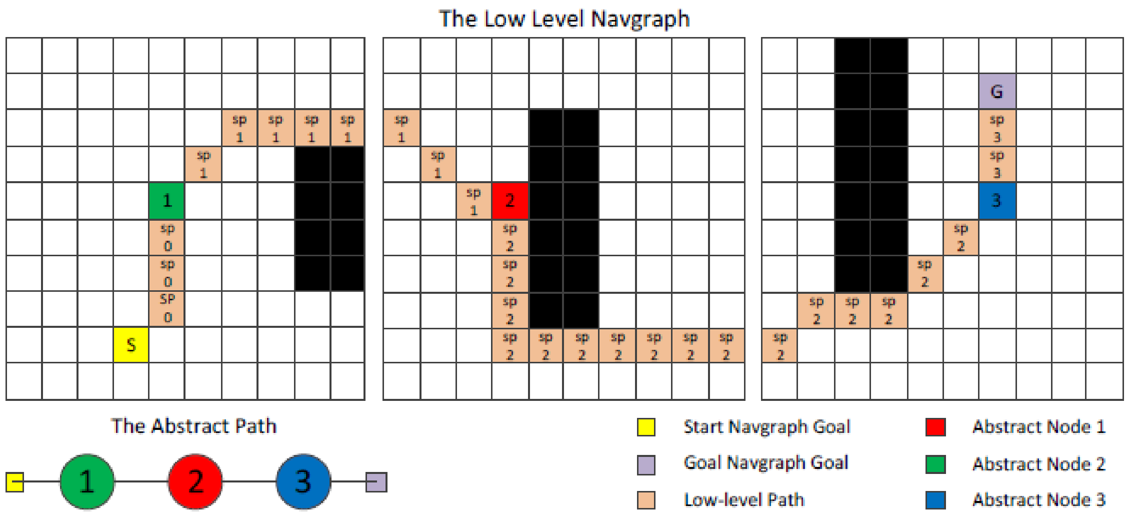


FIGURE 2.15: A basic overview of a hierarchical search [Ferguson and Stentz, 2005]

Figure 2.15 shows a simple example of the search problem using hierarchical approaches. Simple abstraction techniques are used, even though the outcome solution is sub-optimal but this illustration makes for easy understanding. The navgraph is segmented into a 10×10 region with single sub-goals in each region. With three neighbouring regions selected in figure 2.15, the start and the goal nodes which is included in the abstract path (start and end) are given as the yellow and purple nodes respectively. The closet reachable nodes are subsequently selected; the green and blue nodes of the abstract nodes. Abstract searches are performed resulting in an abstract path, these abstract paths are refined into a final solution as shown by the beige nodes in figure 2.15. The edge (sub problem), is given as the labels between the beige nodes.

2.8.5 Merits Of Using Hierarchical Pathfinding

As inferred from the preceding sections, hierarchical approaches lead to a decreased memory and processing cost. Additionally, response time is reduced since agents do not require a complete path before inception of movement, agents can move as soon as the first partial path of the set of segmented paths (sub problem) is found and the next partial path after this needs not be solved until the last partial path is solved.

Hierarchical pathfinding also has an advantage over continuous search processes by exhibiting the property of not reusing information (planned actions are self-contained and atomic) in contrast to its continuous search counterparts. This advantageous property enables for pseudo-continuous processes over an extended range of time without memory implications.

Low cost associated with hierarchical approaches allows for more planning actions to be undertaken in the same time frame as a non-hierarchical approach will perform a single planning action [Botea, Müller, and Schaeffer, 2004]. Multiple agents thus have the ability to complete actions at the same time with good agent responsiveness than would have been required by a single agent requiring no increase in memory usage. Due to this advantage it is currently very popular in games such as Relic Entertainment's *Dawn of War* and *Company of Heroes* and the *Dragon Age* from Bioware [Sturtevant and Geisberger, 2010a][Lamiraux and Lammond, 2001].

2.8.6 Hierarchical Pathfinding A* (HPA^*)

The nascent of video gaming pathfinding with hierarchical approaches started with the HPA* [Botea, Müller, and Schaeffer, 2004], a discrete hierarchical search algorithm. In this technique, the navgraph is segmented into fixed sized clusters as shown in figure 2.16-a and 2.16-b. A 30×30 game environment is divided into a nine, 10×10 clusters of the same size. Abstract nodes are then made through connections of clusters of the game environment. The maximal obstacle free segment along a common border of two adjacent clusters termed entrance are created on a cluster's border, with corresponding symmetrical counterparts also traversable for each segment of traversable nodes. From figure 2.17-b, and considering a certain predefined constant (.i.e 6 in this case for figure 2.17), conditions for transitions between entrances are as follows:

1. Condition (I)- If entrance segment is less than the threshold, a single transition is made in the center of the entrance segment as shown in figure 2.17-b, where

there is a single transition between clusters 3 and 6.

2. Condition (II)-For entrance segment greater than the predefined constant, two transitions are made for the entrance segment, with each at the end of the cluster as illustrated in figure 2.17-b, showing the pair of transitions in clusters 6 and 9.

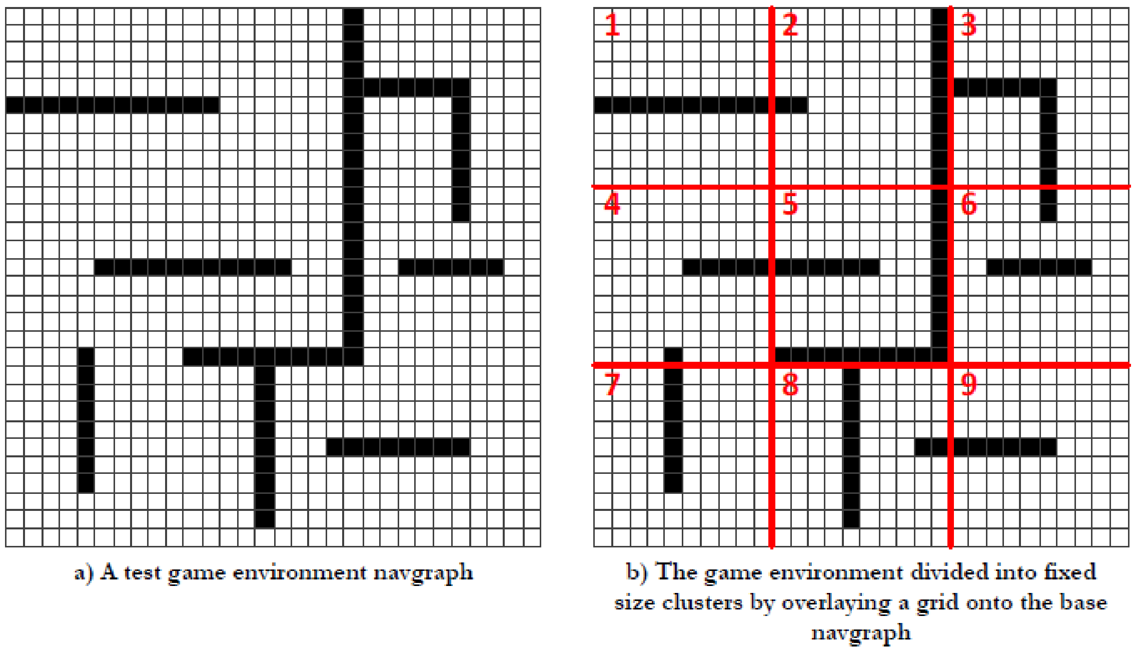

1
2
3
4
5
6
7
8
9
a) A test game environment navgraph
b) The game environment divided into fixed size clusters by overlaying a grid onto the base navgraph

FIGURE 2.16: The HPA* graph abstraction cluster creation [Anguelov, 2012]

Within clusters, inter edges are used for transitions across clusters and intra edges within a single cluster respectively to make up abstract graphs as shown in figure 2.17-c.

The process of constructing (inter edges & cluster entrance) and searching (intra edges) for these interconnections to form abstract graphs is expensive requiring a high cost of processing. To solve this problem, abstract graphs are constructed offline, usually when loading up a game. Abstract graphs are also constructed offline for dynamic environments as static environment. When an environmental change occurs, the both *intra-edges* and *inter-edges* of the affected local clusters need to be re-computed [Botea, Müller, and Schaeffer, 2004]. In the HPA*, modifications are made to recognise clusters that change recalculating all cluster entrances as well as searches for *intra-edges* resulting in an updated set of cluster data.

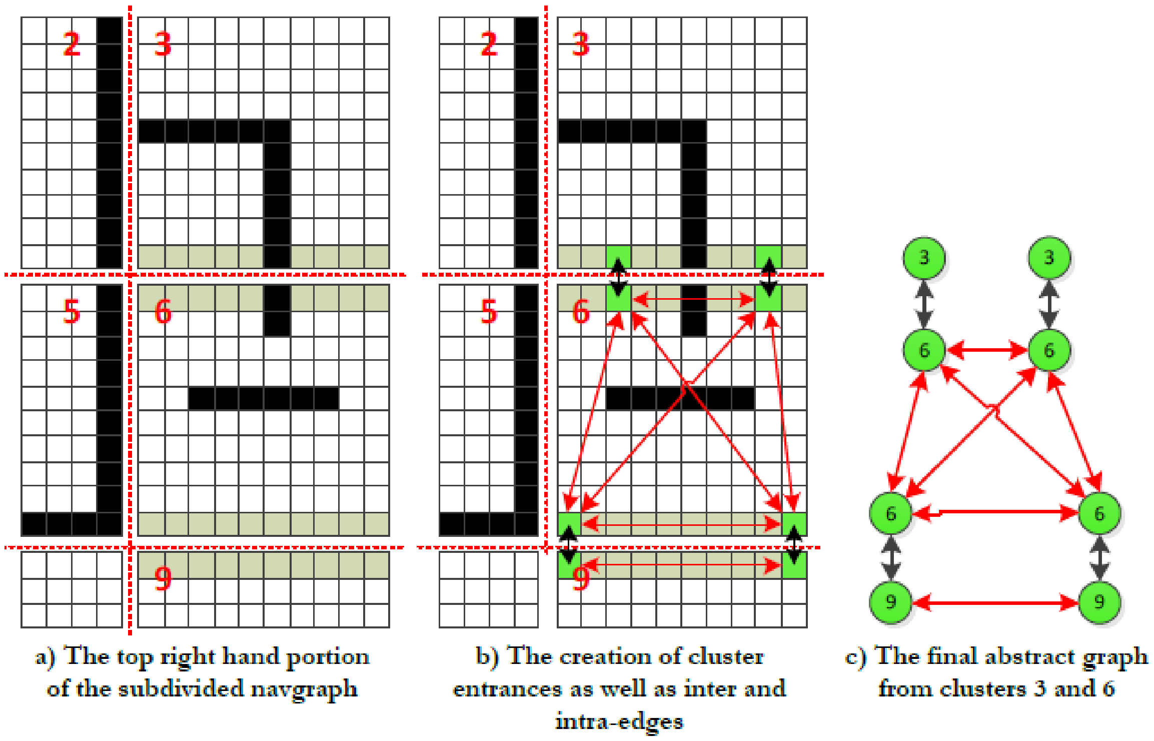

2
3
5
6
9
3
6
9
a) The top right hand portion of the subdivided navgraph
b) The creation of cluster entrances as well as inter and intra-edges
c) The final abstract graph from clusters 3 and 6

FIGURE 2.17: The HPA* graph inter-cluster and intra-cluster links [Anguelov, 2012]

Intra edge searches proves to be costly in hierarchical algorithms, to solve this problem, search for intra edges are postponed with the anticipation that intra edges may not be required in the short run, or the set of cluster data may change before the need of intra edges, avoiding expensive operation which may lead to a costly search in the long run [Jansen and Buro, 2007].

Using HPA* [Botea, Müller, and Schaeffer, 2004], presents a methodology of initially creating from the start nodes, an abstract node link. Through a series of A* searches, the nearest abstract node sharing the same cluster as the start node is selected. With the abstract node found, an edge is subsequently created to link the abstract and start nodes. The goal node also goes through the same process to ensure that both the start and goal nodes are included in the abstract graphs. with the pair of start and goal nodes inserted, a refined abstract path is the created. But the use of several A* searches results in more computational costs [Jansen and Buro, 2007].

The computational cost of inserting both start and goal nodes into the abstract graph cab be deductible by using Dijkstra's algorithm rather than A* to compute the shortest path to all bordering abstract nodes as outlined in [Jansen and Buro, 2007]. Therefore [Jansen and Buro, 2007] proposes the use of the Dijkstra's algorithm to find shortest path to all abstract nodes in the quest of finding the nearest reachable abstract node which offers a comparatively cheaper computational burden. Even though this method offers a less cost, the search for shortest paths only works within clusters of many abstract nodes which has complex or long paths between them. A* still offers a faster option when dealing with large open environment or clusters with few entrants.
Dynamic HPA* [Kring, Champandard, and Samarin, 2010] use a node cache to eliminate the cost of including the start and goal nodes in the formulation of the abstract graph. It does this by making use of the cache within a cluster to store information of the subsequent optimal nodes in the process of reaching an abstract node within particular clusters. With a store information of nearest path to abstract nodes, DHPA* eliminates the use of A$*$ for path refinement. Applications of DHPA* are however limited, even though they present a faster option (approximately 1.9 times) as compared to the HPA*, mainly due to the fact that DHPA* has more memory cost as compared to the HPA* . Also DHPA* provides less optimal solutions as compared to HPA*. From [Botea, Müller, and Schaeffer, 2004] HPA* provides solutions that are less than 10% sub-optimal with respect to DHPA* . This better path sub-optimality is due to the placement of abstract nodes in cluster entrances.

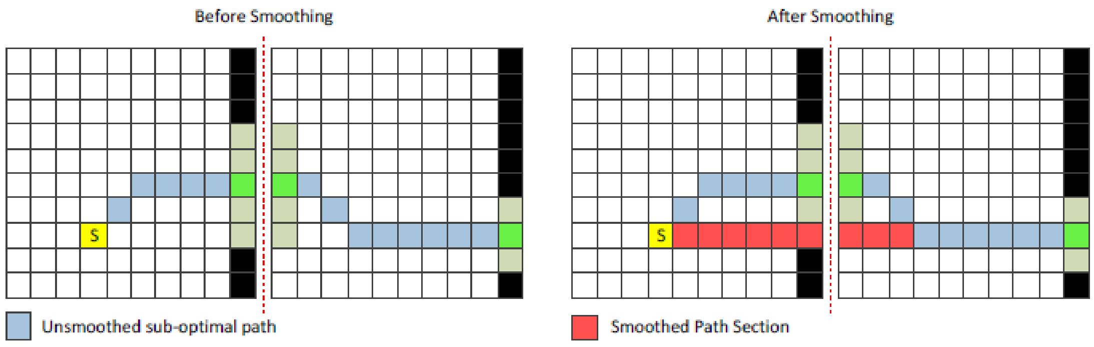


FIGURE 2.18: The effects of the post-processing smoothing phase on path optimality [Anguelov, 2012]

A post processing phase can be included in the HPA* algorithm known as the smoothing phase to improve the optimality of the HPA* solution. At the end of a path, the smoothing phase begins by replacing sub-optimal parts in a path with straight lines. At each nodes in a path, rays are propagated in all directions until an obstructed node is encountered. In the course of ray tracing, initial sub-optimal path segments between two nodes are replaced with straight lines when a node on a returned HPA* path is encountered as shown in figure 2.18-b. The smoothing step then proceeds from two nodes before the encountered path node. Path smoothing helps in improving suboptimality by almost 1% [Botea, Müller, and Schaeffer, 2004]. Path smoothing unfortunately sacrifices improved suboptimality with cost, this could be curtailed through an optimization setup, which involves constraining in a box, the extent at which rays are propagated. This reduces the costs associated with the ray tracing procedure.

2.8.7 Partial Refinement A* (PRA^*)

PRA* [Sturtevant and Buro, 2005]algorithms presents partial but more optimal solutions than HPA* which returns complete paths by implementing a sort of partial hierarchical algorithm. Partial paths are however possible in HPA* with incremental hierarchical pathfinding process, in which the HPA* algorithm is modified such that the refinement of abstract paths are distributed over the duration of an agent's movements. This however leads to a high degree of suboptimality which cannot be improved through path smoothing.

The PRA* technique defers by using the clique and orphan abstraction technique with a multi-level abstraction hierarchy to return partial but optimal paths [Sturtevant and Buro, 2005]. A clique is a set of nodes where edges exists between each node in the set whilst an orphan is a node reachable only from a single other node of the set of nodes. The clique and orphan nodes are represented in red in figure 2.19.

Cliques are represented with a node on the abstract graph which is one level above them, thus requiring a complete navgraphs abstraction until a single node remains at the highest abstraction level. Attached to the single cliques with a single edge are orphan nodes, since they can only be attached to one node. Figure 2.19 illustrates the process of navgraph abstraction with cliques and orphans. It involves

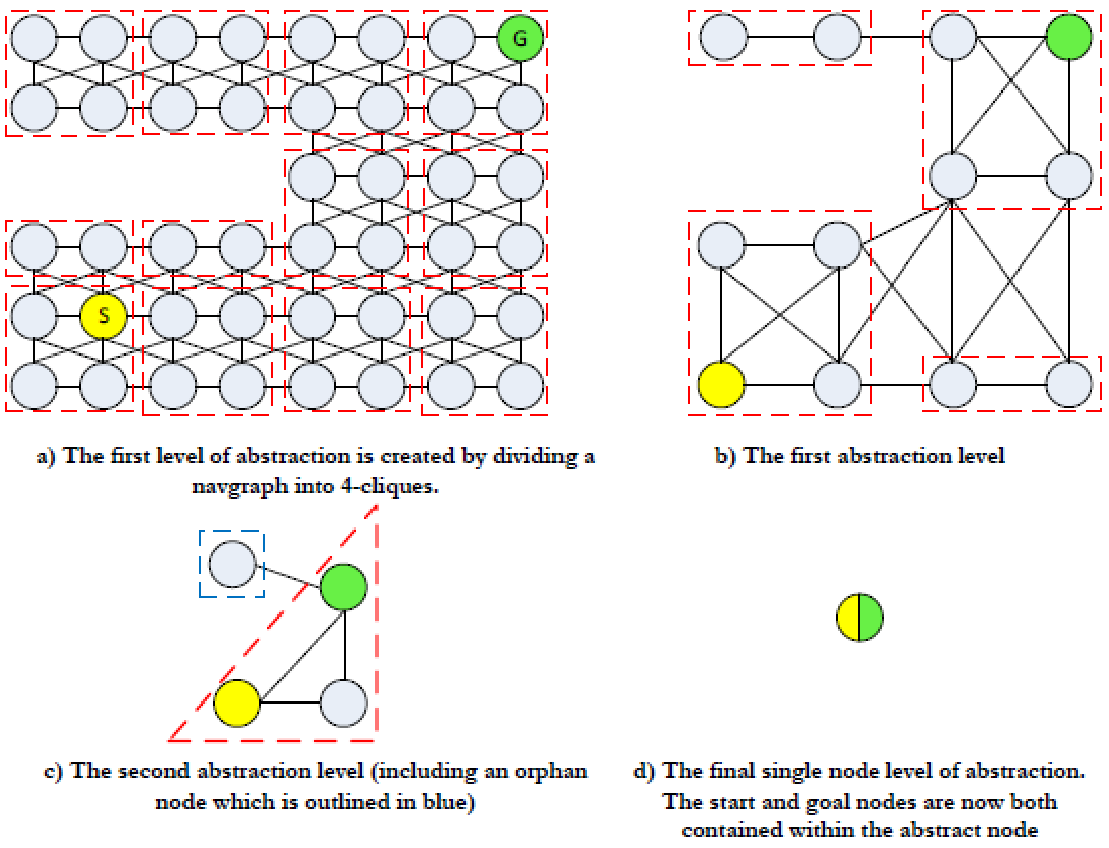

G
S
a) The first level of abstraction is created by dividing a navgraph into 4-cliques.
b) The first abstraction level
c) The second abstraction level (including an orphan node which is outlined in blue)
d) The final single node level of abstraction. The start and goal nodes are now both contained within the abstract node

FIGURE 2.19: The PRA* Abstraction [Anguelov, 2012]

reduction of the search space over the lower level through the process of the already discussed fine granularity, resulting in a 4× reduction. This granularity enables processing and memory cost savings which invariably helps in handling of the creation of the abstraction layers. This is different from the HPA*, which uses abstraction methods that vastly reduces search spaces which as discussed earlier may lead to a loss of valuable information.

In path refinement, the PRA* finds the abstraction level SL where through abstraction, the pair of start and goal nodes are represented as one abstract node (figure 2.19-d). From the level, SL/2, refinement of the path begins which offers merits of keeping a high path optimality coupled with low cost of processing. A* searches are subsequently done at the SL/2 to produces refined paths. If a complete solution is obtained with path-refinement in one step, it is known as the PRA * (∞) (infinite refinement PRA*).

Extremely fine granularity is used for navgarph abstraction in PRA*, this leads to some advantages over HPA*. Primarily, the use of this fine granularity leads to a reduction of the search space between abstract levels to a significant extent as well as providing more optimal solutions in abstract paths than HPA* algorithm.

Partial path refinement in PRA* takes place through the process of truncating abstract paths to a fixed length, K, for each abstract level, this process is referred to as the PRA* (K) algorithm. The tail node of each k-length truncated path, serves as a new goal for the lower level abstraction level. In the PRA* (K) algorithm results in a partial path in one atomic planning process, while ensuring that agents follow the returned partial path and guarantee that the resultant partial path reaches the goal node[Sturtevant and Buro, 2005]. The truncation procedure leads to further minimization of search cost as PRA*(K) enables a reduction of the scope of the abstract search problem at lower abstract levels through the truncation of paths at each level. High optimality is guaranteed through fine granularity for PRA$^*(\infty)$ in a similar time frame as compared to the HPA* (i.e. PRA$^*(\infty)$ return paths that are within 1% of optimal 98% of the time). PRA$^*(\infty)$ however, returns partial paths that are least minimum compared to paths from PRA* (K). Return paths from PRA*(K) has optimality that depends on K, such that for example a K value of 16 is proven to have complete paths within 5% of optimality 98% of the time.

Also fine granularity required in clique abstraction coupled with the demand of a full abstraction hierarchy makes PRA* use in dynamic environments undesirable since dynamic changes in an agent's environment demands a parallel change in all abstract layers, therefore a large variation may subsequently require large abstract layer alterations. PRA* may also not be suitable for applications with limited memory as fine granularity increases the demand of memory storage of abstract levels compared to HPA*. As such, the PRA* algorithm may not be recommended to employ within dynamic environments or on limited memory platforms. The purpose of returning partial paths rather than a complete solution must not be discarded considering it can potentially reduce the volume of wasted effort spent on re-planning a complete path every time when an environmental change occurs.

2.8.8 Minimal Memory Abstraction (MMA)

The algorithm of minimal memory (MM) abstraction introduced by [Sturtevant, 2007] helps in situations where there is a limited memory space. In this abstraction, the navgrid is represented in a grid format as done in the HPA* , thus dividing the surface of the navgrid into clusters of equal and constant sizes. A search space explored by a breath-first search from a traversable node within a sector is termed as a region. The clusters (sectors) are divided such that each node in one region is reachable from another, forming a set of fully traversable regions. A repetitive process is used through the breath-first search for each traversable node not in a region to ensure that all regions within a sector are found. From figure 2.20-b a three sector divided into full traversable regions is presented.

From the region center (centroid node of a region's node set), abstract nodes are created to produce regions in the abstract graph. No *intra-edges* are involved in this process, hence there do not exist connections between the regions with the sector but there exists *inter-edges* which connect regions with adjacent sectors via their region centers which creates abstract graphs.

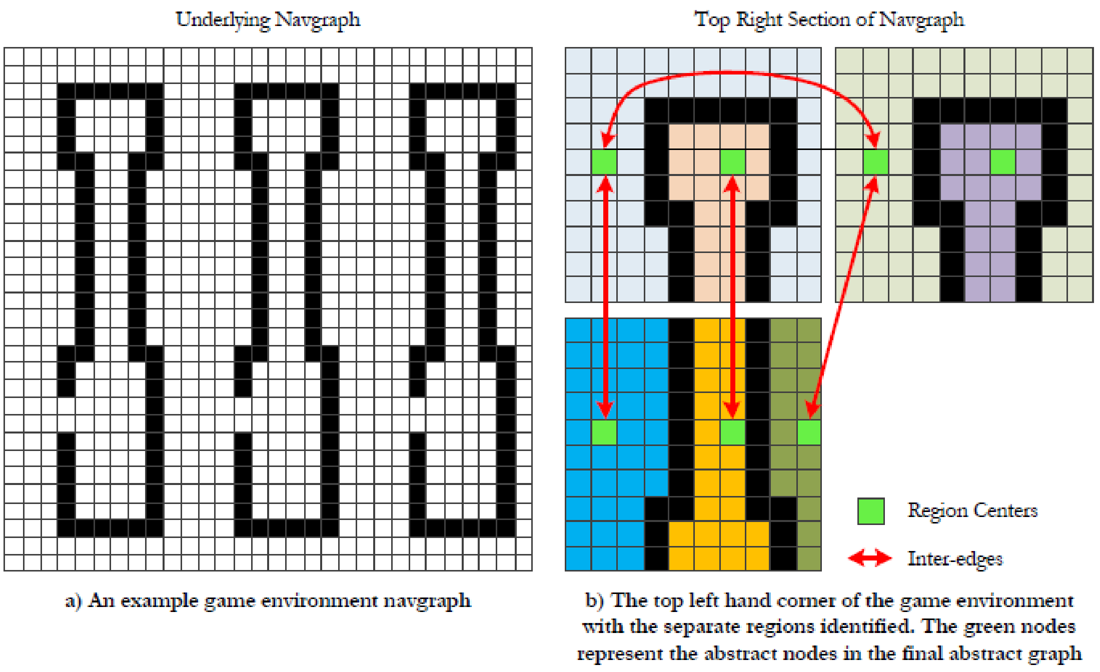

Underlying Navgraph
Top Right Section of Navgraph
Region Centers
Inter-edges
a) An example game environment navgraph
b) The top left hand corner of the game environment with the separate regions identified. The green nodes represent the abstract nodes in the final abstract graph

FIGURE 2.20: The Minimal Memory Abstraction [Anguelov, 2012]

Unlike the HPA* which produces multiple abstract nodes per region through sector transitions, the MM abstraction method seeks to represent entire regions with a single node reducing the number of abstract nodes for abstract graph creation. This abstraction technique minimizes the complexity and size of the abstract graphs, leading to a decrease in memory demand for storing of abstraction and processing costs associated with searches. Abstractions in HPA* and MM are compared in figure 2.21.

Region centres operates the same as sub-goals in HPA*. These region centres are placed in a manner such that the search space exploration of all refinement actions from these centres is minimized. In [Sturtevant and Geisberger, 2010a], the overall search space is decreased by a factor of two with this process. But the use of region centres which are positioned centrally in regions, results in high sub-optimality of low-level paths when used to plan abstract paths (refer to Figure 2.22-a). In order to improve sub-optimality, [Sturtevant, 2007] presents a method of trimming the ends of sub-problem solutions in path refinement which provides slightly more direct low level paths (refer to Figure 2.22-b).

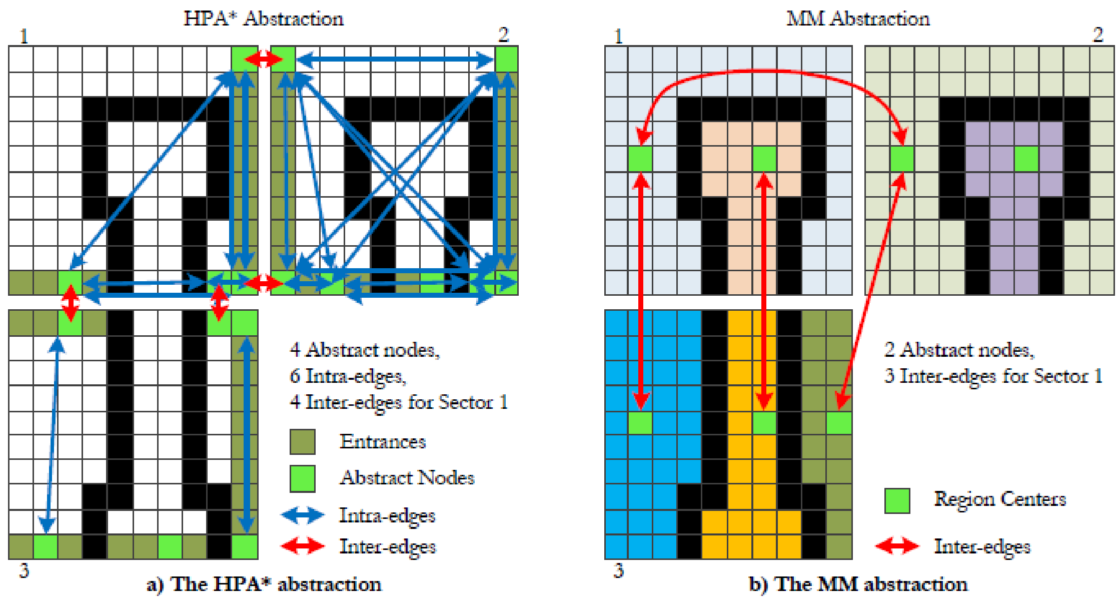


FIGURE 2.21: A comparison between the HPA* and MM* abstractions. (a). (b). [Anguelov, 2012]

However, as described in [Sturtevant, 2007], trimming may lead to an increased in processing cost and unnecessary waste in effort. This wastage arises from the fact that as complete sub-problem solutions are planned, part of the solutions are deleted. As evidenced in [Sturtevant, 2007], a trimming of 10% of sub-problems results in 5% sub-optimality, a further tuning of percentage of trimming to 15% and solving two sub-problem per step, results in improvements of sub-optimality. A post processing stage of smoothing is applied to further improve optimality as done in the HPA* algorithm.

From [Sturtevant and Geisberger, 2010a], it was realised that problems were encountered during the search process in the "Dragon Age" game when using the minimal memory abstractions. This was mainly due to the fact that the game presented a large environment which needed the creation of large and complex abstract graphs for efficient searching which could lead to pathfinding searches running out of time. Also with small environments, the abstract graphs produced did not accurately represent the environment due to coarse abstractions. Accurate representation can be improved by increasing granularity but this leads to longer abstract search times.

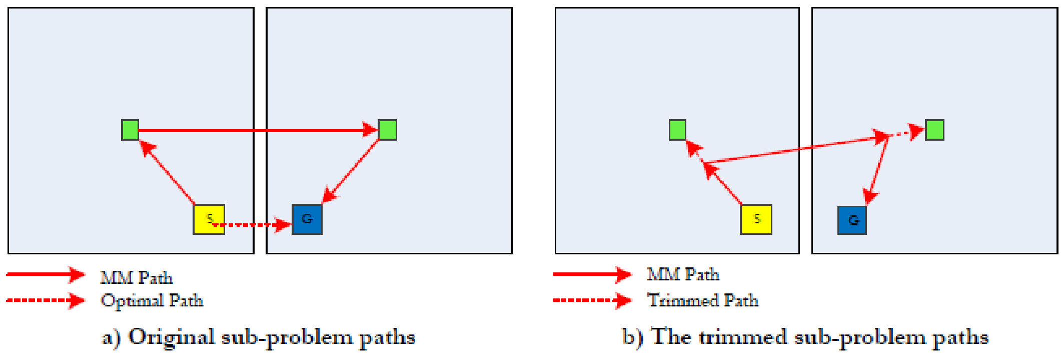

S
G
MM Path
Optimal Path
a) Original sub-problem paths
S
G
MM Path
Trimmed Path
b) The trimmed sub-problem paths

FIGURE 2.22: Improving path sub-optimality through the trimming of sub-problem solutions [Anguelov, 2012]

A second level of abstraction is therefore introduced that produces finer granularity at the 1_{st} abstraction level and also improves search times. This second level of abstraction enables a faster completion of abstract searches on both abstraction levels which results in a decreased agent response time.

The minimal memory abstraction methodology manifests simplicity in application as well as a low demand for memory, hence it makes it suitable for memory limited environments. The MM abstraction technique has proven to be applicable to dynamic environment albeit the problem of cost arising from the addition of a second abstract layer.

2.8.9 Dynamic Hierarchical path-finding A* (*DHPA**)

Kring and et al [Kring, Champandard, and Samarin, 2010], introduced the Dynamic Hierarchical path-finding A* (DHPA*) and Static Hierarchical path-finding A* (SHPA*) hierarchical path-finding algorithms, along with a metric for comparing the dynamic performance of path-finding algorithms in games. Both SHPA* and DHPA* consist of a build and a search algorithm. In DHPA* the run-time cost is reduced by spending more time and memory usage in the build algorithm and less time in the search algorithm. In SHPA* the performance is improved and the memory requirements of HPA* are reduced.

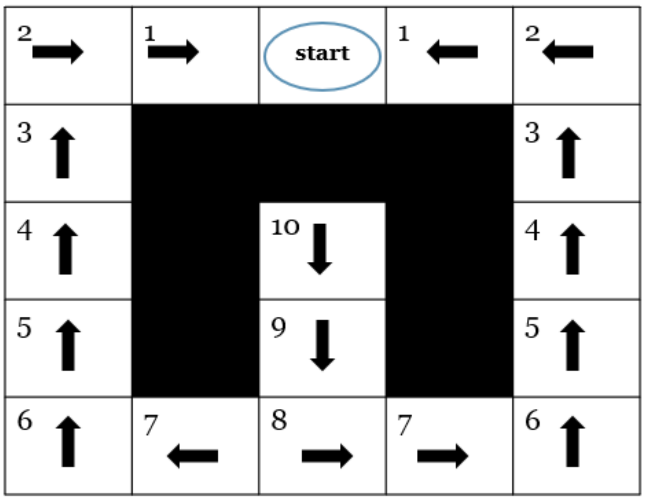


FIGURE 2.23: DHPA* cache for a single abstract node in a cluster of size 5. In this figure, we do not consider diagonal distance, for simplicity [Kring, Champandard, and Samarin, 2010]

Like HPA*, the DHPA* makes clusters in the build stage but it stores more data in order to speed up both abstraction search and also low level search. The DHPA* uses Dijkstra search algorithm to make a separate cache for each abstract node inside a cluster (see figure 2.23). The cache includes an entry for each low-level node inside the cluster, representing data about the optimal path which is founded by the Dijkstra search algorithm from the low-level node to the abstract node. One cache entry includes the optimal path length from two start and end nodes and also a path index steering to a neighbor node inside the corresponding cluster. Effectively, the cache describes the optimal path tree for each abstract node. The abstract search uses the cached path length, and the low-level search uses the cached path index. When a dynamic change occurs inside a cluster, the build algorithm will run just inside the corresponding cluster and recalculate the corresponding part of the cache, the *intra-edges* inside the cluster, and the *inter-edges* connected to the cluster. Additionally, the build algorithm optimizes the re-computation by just rebuilding a cluster's *intra-edges* and *inter-edges* when needed. It is just required to rebuild these edges if each of the border nodes dynamically changes. Otherwise, the abstract graph remains the same, and just the cache is re-computed. Since the DPHA* stores some additional information, it requires more memory space than the HPA* algorithm. The DHPA* search algorithm uses the cache that was produced in the build algorithm to produce the abstract path, and improve it within a low-level path. This method enhances the run time of the abstract search by omitting the time consuming "SG effort" that is present in the HPA* algorithm. It means that the DHPA* does not have any SG effort unlike the HPA*.

DHPA* identifies the abstract graph nodes that refer to the start cluster at the starting of the abstract search. These nodes are then pushed over the open list as the primary search space, in place of the abstract node that HPA* adds to the graph [Kring, Champandard, and Samarin, 2010]. The costs for any of those nodes are regained from the cache that was generated by the build algorithm, rather than searching for the costs as the HPA* does. Finally, the algorithm searches the abstract graph just like in HPA*.

The SHPA* search is similar to the DHPA* search, but it does not use a cache. Rather, it utilizes an Euclidean distance heuristic to calculate edge weights, and it searches for low-level paths using A* search algorithm.

2.9 Summary

In this chapter, different pathfinding algorithms are introduced, describing in detail their characteristics which best suits applications such as video games. The A*, is introduced, as it lays the foundation for other algorithms which offer augmented abilities. For example, any time algorithms, such as the ARA* algorithm have the added ability of producing sub-optimal solutions, but faster, thus improving the computational time. In dynamic environments, algorithms such as the D* and D**Lite* have the advantage of reacting to changes in an agent's environment. The AD* then combines the benefit of the two schemes. The Theta* and the Field D* are not limited to movements on grid edges, thus allowing for smooth paths.

In section 2.8 the search process of hierarchical algorithms is explored. It involves an efficient discrete graph search algorithm which decomposes a problem into multiple sub-problem which results in a reduction in search times and also a reduction in peak memory usage. The algorithm involves the segmentation of the main problem into sub-problems, with the help of an abstract graph which is formed through abstraction techniques on the navigation graph.

The HPA*, was explained in this section as a hierarchical method that builds an abstract graph from connections between fixed size clusters. Usually HPA* algorithms offers a relatively high sub-optimality when compared to other hierarchical methods (usually 6% sub-optimality), but this can be improved by the addition of a post-processing smoothing stage, where sub-optimal part of paths are replaced with straight lines to improve sub-optimality. HPA* technique is suitable for both static and dynamic environments and also memory limited cases with large environments. HPA* has many interesting properties that but the limitation of having been developed for regular 2d grids, whereas more recently game developers are using mostly navigation meshes based on convex polygons.

The PRA*, is a hierarchical technique that introduces fine granularity which results in reductions of the search environment. There are two types of PRA*; one type returns partial abstract paths of length k, thus its name PRA*(K), and provides solutions with low sub-optimality, and the PRA$^*(\infty)$ which returns complete paths. The PRA* has shown to be unsuitable for dynamic environments since high costs are encountered while updating the abstraction hierarchy every time there is a change in an agent's environment. It is also not appropriate for memory limited platforms primarily because of the need for a complete abstract hierarchy and abstract levels having a low level of search space reduction between them.

Due to the PRA* algorithm's shortfall when it comes to memory limited platforms, the MM hierarchical algorithm is introduced. The coarse abstraction is used which unfortunately leads to an increase in sub-optimality requiring a trimming method to improve the suboptimality to about 5%. This technique is suitable for dynamic and memory limited environments. There are many other methods on pathfinding which we have summarized in table 2.1.

TABLE 2.1: Recently reported pathfinding algorithms used in robotics and video games

Topologies	**Environment**	**Pathfinding addresses**	**Examplification**	**Memory complexity**	**Time complexity**	**technique**	**Reference**
Undirected uniform-cost square grid maps without diagonal movement	static	Single-agent	Game development	-	A* $O(n \log_n)^{\frac{1}{7}}$ IDA*	A* and IDA	[2006]
Undirected uniform-cost square grid maps without diagonal movement	static	Single-agent	Game development	-	$O(n \log_n)^{\frac{1}{7}}$	Improved A* algorithm	[2010]
Undirected uniform-cost square grid maps without diagonal movement	static	Single-agent	Game development	No memory overhead	JPS algorithm $O(n \log_n)^{\frac{1}{10}}$	A*, HPA*, and JPS	[2011]
Undirected uniform-cost square grid maps without diagonal movement	static	Single-agent	Game development	$O(b^k)$ b= maximum branching factor, k=iteration	-	IEA* and IDA* algorithms	[2015]
Undirected uniform-cost square grid maps without diagonal movement	static	Single-agent	Game development		SUB algorithm $O(n \log_n)^{\frac{1}{100}}$	SUB, BlockA*, CPD-full, CPD-mbm, JPS-offline, JPS-online, PDH, PPQ, and Tree	[2013]
Undirected uniform-cost square grid maps without diagonal movement	Real-time	Multi-agent	Game development	-	-	ITF, EITA, and MC-ITA schemes	[2013]
Undirected uniform-cost square grid maps without diagonal movement	Real-time	Multi-agent	Game development	-	$O(n \log_n)^{\frac{1}{10}}$	A*, FS, PBS, and PRS algorithms	[2012]

Table 2.1 continued from previous page

Topologies	**Environment**	**Pathfinding addresses**	**Examplification**	**Memory complexity**	**Time complexity**	**technique**	**Reference**
Hexagonal grid	Real-time	Multi-agent	Robotics systems	-	-	Augmented A* and Accelerated A*	[2011]
Hexagonal grid	Real-time	Single-agent	Robotics development	-	-	D* algorithm	[2013]
Triangular grid	Real-time	Multi-agent	Robotics and games development	-	-	AD* algorithm	[2013a]
Cubic grid	static	Single-agent	Robotics and games development			Theta*, Lazy Theta*, and A*	[2013]
Mesh navigation	Real-time and dynamic	Multi-agent	Robotics and games development	45 KB of memory per agent		Framework	[2010a]
Mesh navigation	static	Single-agent	Game development	-	-	HNA*	[2016b]
Mesh navigation	Dynamic	Single-agent	Robotics and games development	-	-	Anytime Dynamic A*	[2013b]
Mesh navigation	Dynamic	Multi-agent	Robotics and Game development	1-5 MB	-	Weighted A* (WA*), Near-Optimal Bidirectional Search (NBS)	[2019]
Mesh navigation	real-time	Multi-agent	Game development	-	-	Bounded Multi-Agent A* (BMAA*),	[2018]
Visible graph	Dynamic	Multi-agent	Robotics and games development	AA* less than A*	-	AA* algorithm	[2009]
Waypoint	Real-time	Multi-agent	Robotics and games development		-	Ant colony optimization algorithm	[2010]
Grid Mesh	static	Multi-agent	Robotics and games development	-	-	Evolutionary Heuristic A* search (EHA*)	[2019]

Chapter 3

Hierarchical Pathfinding for Navigation Meshes.

As we have seen in the previous chapters, the challenge of pathfinding in video games is to compute optimal or near optimal paths as efficiently as possible. As both the size of the environments and the number of autonomous agents increase, this computation has to be done under hard constraints of memory and CPU resources. Hierarchical approaches can compute paths more efficiently and have been widely applied on 2D regular grids. On previous work by my advisor [Pelechano and Fuentes, 2016b], a hierarchical approach for general navigation meshes was presented, known as HNA*. The method provided a hierarchical solution adapted to the peculiarities of navigation meshes where cells are convex polygons of different shapes and sizes. HNA* offered very good speeds ups for pathfinding, however only for certain configurations of the hierarchy. For other configurations, performance could drop drastically when inserting the start and goal position into the hierarchy. In the original HNA*, the step of inserting the Start and Goal positions in the hierarchy was done sequentially and thus it could turn into a bottleneck. In this chapter we first explain in detail the HNA*, then we discuss the problems and study alternatives to improve performance. We propose three novel methods that rely on further memory storage or parallelism on either the CPU or the GPU. Finally, we carry out a comparative evaluation, with results showing an important speed-up for all tested configurations and scenarios.

3.1 Introduction

Path planning for multiple agents in large virtual environments is a central problem in the fields of robotics, video games, and crowd simulations. In the case of video games, the need for highly efficient techniques and methods is crucial as modern games place high demands on CPU and memory usage.

Pathfinding is a subset of AI which has intersections with group coordination, animation, goal selection, etc [Vermette, 2011]. Increasing search space in pathfinding while improving its accuracy and speed, is always demanded by game developers and industry professionals.

Video games applications provide therefore a perfect testbed for researchers working on this field. Path planning should provide visually convincing paths for one or many autonomous agents in real time.

Agents should move towards their destination along a realistic path, maintaining an appropriate amount of clearance with respect to the obstacles while avoiding collisions with other agents as smoothly as possible. The most popular solutions in the literature are based on a combination of global and local movement techniques.

Most of the papers which are referenced in this thesis are focused on proposing new methods for the AI community, developing video games and others combine empirical experiments using video game as test problems.

Even though optimally is typically the goal in path fining, when it comes to games, often it is not necessary to obtain the optimal path for all agents but paths that look convincing to the player.

The problem of pathfinding can be separated from local movement, so that pathfinding provides the sequence of cells to cross in the navigation mesh, and other methods can be used to set waypoints and to handle collision avoidance. In this chapter, we focus on abstraction hierarchies applied to pathfinding to improve performance. A general notation consists of labelling the hierarchy as levels or layers in ascending order, with the lowest, $L0$ being the un-abstracted map in the game space and consecutive layers numbered $L1$, $L2$ and so on being the different levels of abstraction. The key idea consists on performing a search at a high level, which is then "filled in" with more refined sections of the path at lower levels, until a complete path is specified which can be followed by an agent [Bulitko, Björnsson, and Lawrence, 2010]. Typically a high level solution can be rapidly calculated, and the challenge lies on inserting the specific Start (S) and Goal(G) positions to link them with the high level graph. Work in the literature shows that this inserting S/G step can become a bottleneck in both 2D grids [Botea, Müller, and Schaeffer, 2004] and Navigation Meshes [Pelechano and Fuentes, 2016b]. There are many techniques in the literature that have shown impressive improvements for the case of 2D regular meshes to increase speed without a large memory footprint [Sturtevant, 2007]. However general navigation meshes consisting of convex polygons of different complexity, present more challenges given their irregular nature (i.e. not all the cells have the same size and edge length) [Van Toll et al., 2016]. In this chapter we propose several approaches to speed up the existing bottleneck in hierarchical pathfinding for general navigation meshes, and evaluate their advantages and limitations in terms of both memory usage and performance improvements.

3.2 Related Work on Hierarchical Approaches

There has been a large amount of work in the field of hierarchical abstractions to speed up pathfinding or general graph search.

Hierarchical graph representations have been used for example, for visualization purposes of large data sets [Tominski, Abello, and Schumann, 2009]. The goal in these applications is to offer an overview first, and then be able to zoom and filter to offer details on demand.

Planning via hierarchical representation has been used to improve performance in problem solving for a long time [Sacerdoti, 1974]. Holte et. al introduced hierarchical A* to search in an abstract space and use the solution to guide search in the original space [Holte et al., 1996a]. There has also been work on abstraction based on bottom-up approaches for general graphs [Holte et al., 1994] [Holte et al., 1996b]. Sturtevant and Jansen extended the theoretical work slightly and provided examples of a number of different abstraction types over graphs. In this work graphs are created from 2D grid-like structures by setting a node for each walkable cell [Sturtevant and Jansen, 2007].

The work by Rabin et al. performs pathfinding using a two-level hierarchy, by creating clusters with the rooms of a building or the square blocks of a filed [Rabin, 2000a]. An abstract action crosses a room from the center of an entrance to another. Firstly, it partitions the problem map into clusters such as square blocks. Secondly, abstract actions are calculated as block crossings. And thirdly, it abstracts a block entrance into one transition point. This leads to fast computation but gives up the solution optimality.

Bulitko et al. showed that the quality of paths can decrease exponentially with each level of abstraction [Bulitko, Björnsson, and Lawrence, 2010]. Sturtevant and Geisberger studied the combination of abstraction and contraction hierarchies to speed up pathfinding [Sturtevant and Geisberger, 2010b].

Botea et al. introduced the HPA* (Hierarchical Pathfinding A*) algorithm [Botea, Müller, and Schaeffer, 2004] which we described in detail in section 2.8.6. HPA* is a hierarchical approach to reduce problem complexity in pathfinding on grid-based maps. The HPA* technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing each cluster are pre-computed and cached. At the global (high) level of this method, an action consists of crossing a cluster in a single big step rather than moving to an adjacent atomic location and small clusters are grouped together to create larger clusters.

Another important hierarchical approach for pathfinding in commercial video games uses points of visibility [Rabin, 2000a]. In this method the domain local topology is used in order to define an abstract graph that covers the map efficiently. The graph nodes represent the corners of convex obstacles. For each node, edges are added to all the nodes that can be seen from the current node (i.e., that can be connected with a straight line).

Pelechano et al [Pelechano and Fuentes, 2016b] presented a hierarchical NavMesh technique to speed up pathfinding in navigation meshes. The method is based on a bottom-up approach to create a hierarchical representation using the multilevel k-way partitioning algorithm (MLkP), annotated with sub-paths information. Their approach is flexible in terms of both the number of levels in the hierarchy and the number of merged polygon between levels of the hierarchy. The advantage of their method is that this technique provides a balanced number of both walkable cells and *inter-edges* between partitions. Figure 3.1 shows an example of their hierarchical NavMesh Graph (HNG) for a two-levels-hierarchy and $\mu = 4$ (where μ is the number of merged polygons).

Hierarchical Annotated A* (HAA*) [Harabor and Botea, 2008] is en extension of HPA that takes into account the size of the agents and the terrain traversal capabilities. Therefore it allows for different agents' sizes to efficiently plan high quality paths in heterogeneous-terrain environments. Kring and et al [Kring, Champandard, and Samarin, 2010], introduced the Dynamic Hierarchical pathfinding A* (DHPA*) and Static Hierarchical pathfinding A* (SHPA*) hierarchical pathfinding algorithms, along with a metric for comparing the dynamic performance of pathfinding algorithms in games. In DHPA* the run-time cost is reduced by spending more time and memory usage in the build algorithm and less time in the search algorithm. In SHPA* the performance is improved and the memory requirements of HPA* are reduced. In the DHPA* algorithm, the clusters are created in the build algorithm similarly to HPA*, but additional information to improve the speed of both the abstract search and the low level search are cached. In this method they used Dijkstra algorithm once for each abstract node within a cluster, creating a separate cache for each abstract node. In DHPA*, improves the search performance by eliminating the time consuming "SG effort" that is present in HPA*. Our work is inspired by their method, but extended to the more general problem of navigation meshes where certain assumption such as cell size cannot be made beforehand.

In the SHPA*, the build algorithm runs just one time to creates the clusters, and it does not repair the clusters dynamically. In this method the build algorithm is the same as in [Sterren and Champandard, 2008]. In the SHPA* build algorithm, the map is decomposed into many variable-sized fully connected clusters based on a greedy heuristic instead decomposing the map into many same-sized clusters as HPA* and the SHPA* search algorithm is similar to the DHPA* search with the main difference being that HPA* search algorithm does not use a cache.

In [Samet, 1988] a method for doing hierarchical map decomposition is proposed. In this method a map is partitioned into a set of different size square blocks and each block includes only walkable cells or only blocked cells. The map is partitioned into four blocks. If one block contained both walkable and obstacle region, then it will be decomposed into four smaller blocks, and so on. One of the actions that the agents will have to perform is to move between the centers of two adjacent, which leads to a sub-optimal solution since the agents will not follow the shortest path.

Ammar et al. have presented RA* algorithm which is a new linear time relaxed version of A* [Ammar et al., 2016]. This method proposed to solve the path planning problem for large scale grid maps. The main goal of this algorithm is finding an optimal or near optimal path with small deviance from the optimal solutions by spending much smaller execution times than traditional A*. This method exploits the grid map structure to build a highly accurate approximation of the optimal path, without visiting any block more than once, unlike A* for which the cost $g(n)$ of a node n may be computed more than one time. The Another variant of A* search algorithm is the Theta* algorithm [Nash et al., 2007]. The fundamental Theta* calculation is an existing algorithm that produces near-optimal results for a running time near A* around 8-directional grids. But the main problem of this algorithm is that it often finds non-tough paths that make unnecessary turns.

Shunhao Oh et al. shown that by restricting the search scope of Theta* algorithm to tough paths, the algorithm can provide much shorter paths than the basic algorithm. Before a vertex v is relaxed with parent u, the sub-path $(parent(u), u, v)$ is first checked for tautness [Oh and Leong, 2016]. In this case if the path is not taut, an additional penalty value will be added to distance(v) after relaxation. The vertex v is additionally marked as not taut, so that the increment in the $g - value$ can be reversed later on when the vertex v is extracted from the priority queue.

The HNA* algorithm [Pelechano and Fuentes, 2016b] is a bottom-up method to create a hierarchal representation based on a multilevel k-way partitioning algorithm (MLKP) of a navigation mesh. Similarly to HPA*, HNA* also pre-computes sub-paths and stores them to be accessed by the on-line search algorithm. The HNA* consist 4 main step: The first step connects S (Start) and G (Goal) points to their partitions at the level *i* by calculating the shortest paths to each portal in their respective node (*inter-edges*). The second step calculates paths at the current level. Step 3 extracts the *Intra-edges* from level *i-1* and step 4 obtains the refined path in the level 0. In this chapter we focus on the limitations of HNA* and present several methods to solve the bottleneck that appears in step 1, and perform a thorough comparison of our results in terms of memory and performance.

3.3 Hierarchical problem formulation

A world map is typically given as a polygon soup. In order to have agents navigating a world map, it is necessary to find a representation of the walkable space. This can be done with a navigation mesh, which represents the walkable space as a collection of convex polygons called cells (could be triangles or polygons of more than three sides), where borders between adjacent cells are called portals ["Recast" 2017]. Agents can move within any two points of a cell or cross portals to move between adjacent cells, without colliding with the static obstacle borders of a cell. This representation can be expressed as a graph $G = (N, E)$, where the collection of cells or convex polygons are the nodes or vertices of the graph $N =< p_0, p_1, ..., p_n >$, and the portals are the edges E, with each edge e_{ij}, corresponding to the edge between two adjacent polygons p_i and p_j. The cost of an edge $c(e_{ij})$is calculated as the distance between the center of polygon p_i to the center of polygon p_j, and thus it is always a positive value. Pathfinding involves finding a path $P = \langle S, ..., u, ..., v, ..., G\rangle$ which is a sequence of nodes connected by edges, from the starting position S to the goal position G. The cost of a path $c(P)$ is the sum of all the costs assigned to the edges along the path P, and since all edges costs are positive values, the cost of a path will always be a positive value. The shortest path between S and G is the path of minimum cost among all possible paths. A* performs an informed graph search, by computing for each node being explored the function $f(x) = c(x) + h(x)$, where $c(x)$ is the current cost from S to node x, and $h(x)$ is the heuristic that estimates the optimal cost of the path from x to G [Hart, Nilsson, and Raphael, 1968]. When dealing with maps, $h(x)$, can be computed as the Euclidean distance between the position of the center of node x, and the position of the center of node G. With this heuristic, A* can always find the optimal path, which is the path of minimum distance.

Each level of the hierarchy $Lx, x > 0$, is represented by a new graph G_x which is created by merging μ connected nodes from G_{x-1} (the value of μ is decided by the user). The new graph $G_x = (N_x, E_x)$, consists of a set of nodes $N_x = \langle n_x^0, n_x^1, ..., n_x^m\rangle$, where each node in G_x is a subgraph of μ connected nodes from G_{x-1}, so that $n_x^i = \langle n_{x-1}^j, n_{x-1}^k, ..., n_{x-1}^l\rangle$. Edges E_x in G_x are the subset of edges from G_{x-1} that connect two nodes n_x^s and n_x^d, where $s \neq d$.

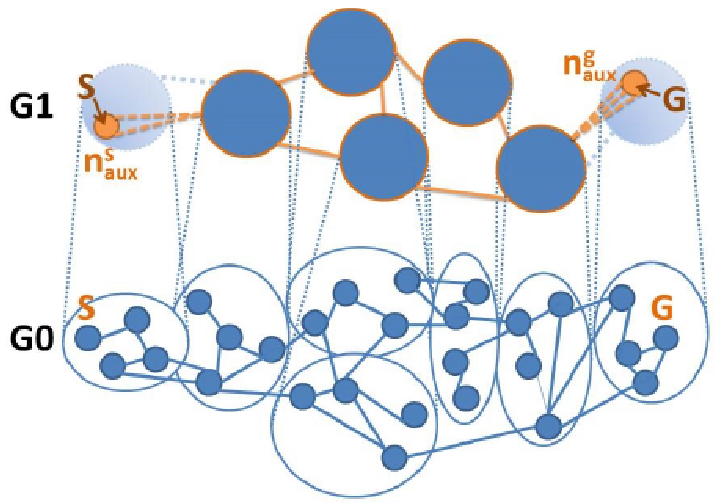


FIGURE 3.1: Example of HNG with two levels and $\mu = 4$. The orange circles and discontinuous links represent the temporal nodes and edges created after linking Start and Goal points to the HNG. This temporal graph is where the HNA* runs [Pelechano and Fuentes, 2016b].

Definition 3.3.1. An *Inter-edge*, ι_x^{sd}, in G_x is an edge e_{ij} from G_{x-1} that connects two nodes n^i_{x-1} and n^j_{x-1}, such that n^i_{x-1} is inside n^s_x, n^j_{x-1} is inside n^d_x, and $s \neq d$.

For those edges e_{ij} from G_{x-1} that connect two nodes n^i_{x-1} and n^j_{x-1}, such that both n^i_{x-1} and n^j_{x-1} are inside n^s_x, they become internal edges of node n^s_x. Therefore, there is no loss of connectivity between G_{x-1} and G_x, since all the set of edges in E_{x-1} are now either internal edges of nodes n^s_x in G_x or *inter-edges* in G_x.

These concepts are shown in Figure 3.1. In the case of L1, the merged nodes from L0 are polygons of the navigation mesh. Figure 3.2 shows an example of a simple navigation mesh from level L0 to L3. Colors are used to represent nodes at each level, so we can appreciate how each navigation mesh polygon turns into a node at L0, and then several connected polygons from L0 are merged in one larger node at L1, and similarly for L2.

The graph G_x contains a partition of G_{x-1}, with nodes at Lx being groups of adjacent nodes from $L(x-1)$, and edges E_x being a subset of the edges of E_{x-1}. Each node n_x can be traversed by finding an internal path between a pair of *inter-edges*. Such internal paths are represented by a sequence of polygons and can be pre-computed and stored.

Definition 3.3.2. An *Intra-edge*, $\pi_x^{s(dk)} = \langle p_0, p_1, ..., p_k \rangle$, is a sequence of polygons from G_0 that represent the optimal path to traverse a node n_x^s between two *inter-edges* ι_x^{sd} and ι_x^{sk}. Therefore, $\pi_x^{s(dk)} = optimalPath(\iota_x^{sd}, \iota_x^{sk})$. Its weight is computed as the sum of costs of the edges e_{ij} along the path, $c(\pi_x^{s(dk)}) = c(e_{01}) + c(e_{12}) + ... + c(e_{(k-1)k})$, where e_{ij} is the edge between nodes p_i and p_j.

A node n_x^s will have an *intra-edge* for each pair of *inter-edges*. In order to find a high level path, we need a Hierarchical Navigation Graph, $HNG_x = (V'_x, E'_x)$, which captures the connectivity of G_x given by the relationships between *inter-edges* and *intra-edges*. In HNG_x, the vertices are all the *inter-edges* in the partition represented by G_x, $V'_x = \langle \iota_x^{sd}, \iota_x^{dk}, ..., \iota_x^{lm} \rangle$, and the edges, E'_x are *intra-edges*, $\pi_x^{d(sk)}$ connecting each pair of *inter-edges*, for which a path exists.

Note that HNG_x maintains the connectivity of the navigation mesh, but in a more compact representation, where only some edges are kept as nodes in HNG_x (those *inter-edges*, which depend on the hierarchical level L and the merging factor μ), and the shortest paths at $L0$ between those nodes are precomputed as *intra-edges*. Therefore HNG_x is built in a way that guarantees that the connectivity between polygons at L0 is kept regardless of the hierarchical configuration.

If a path, $P_0 = \langle p_S, p_1, p_2, ..., p_G \rangle$, exists at G_0 , then there will be a path at level Lx. Computing pathfinding in HNG_x gives as a result the path $P_x(S,G) = \langle \pi_{temp}^S, \pi_x^{s(dk)}, \pi_x^{k(sq)}, ..., \pi_x^{r((m-1)m)} \pi_{temp}^G \rangle$. $P_x(S,G)$ is the high level path. The temporal paths, π_{temp}^S and π_{temp}^G, were created during the connect S and G steps, which computes a path at level L0 for the subgraph represented by the high level node S, and similarly for G. Therefore $\pi_{temp}^S = \langle p_s, p_0, p_1, ..., p_n \rangle$ where p_n is a polygon with one of the edges being the *inter-edge* that connects p_n with the first polygon in $\pi_x^{s(dk)}$. Extracting the sequence of polygons from each *intra-edge* $\pi_x^{i(jk)}$ we obtain the full sequence of polygons to traverse the navigation mesh between S and G (Proof in Appendix A).

3.4 The HNA* algorithm

The focus of this chapter consists of solving the bottleneck that appears in HNA* when inserting start (S) and goal (G) positions into the high level abstraction graph. Before explaining the details of our approach, we would like to remind the reader

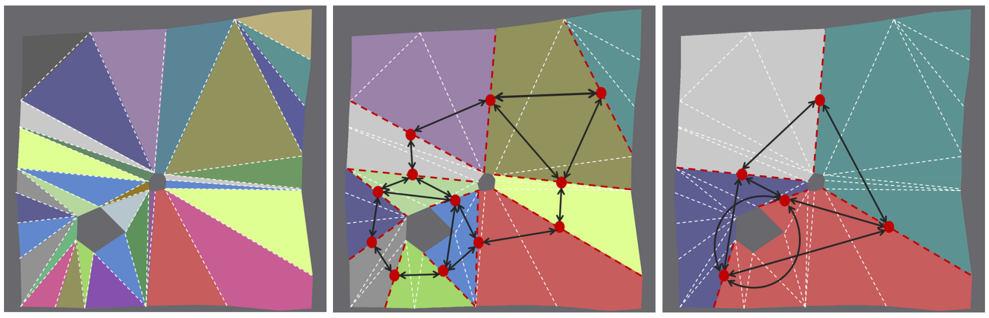


FIGURE 3.2: From left to right we can see a simple map at L0, L1 and L2. The HNG has been built with $\mu = 3$. Note that colors are used to identify nodes in each level, and the overlapping of a node in L1 with colored nodes in L0 visually identifies which nodes in L0 are merged to form a node in L1 and similarly between L1 and L2. White dotted lines indicate portals at L0, red dotted lines in L1 indicate *inter-edges* (connections between nodes at L1), and the same applies for L2 (on the right hand side). Finally, black arrows in L1 and L2 indicate *intra-edges* (pre-computed A* paths to cross a high level node from one *inter-edge* to another). HNG consists of the set of vertices represented by red dots (one per each *inter-edge*), and the set of edges represented by black arrows (one per each *intra-edge*)

the origin of this problem. A hierarchical navigation mesh consists of several layers, where a node of a higher level contains a group of merged nodes from a lower level. Finding a path in this representations consists of four steps (as illustrated in Figure 3.4):

- (1) Insert S and G.
- (2) Find path at high level.
- (3) Extract sub-paths (stored from an off-line phase).
- (4) Delete S and G from high level graph.

In the first step, two given S and G points (as start and goal positions) are inserted in the geometry at the lowest level ($L0$) of the hierarchy and then higher levels of the hierarchy recursively are created by corresponding nodes at each higher level L. Both S and G points also are inserted temporally in the higher level of the hierarchy graph G_L as n^s_{aux} and n^G_{aux}.

In order to connect n^S_{aux} and n^G_{aux} to the higher level graph G_L, a path needs to be computed from S to each inter-edge at the higher level node n^S_l which contain S. Each node at the higher level contains a subset of the nodes from the lowest level ($L0$) at the corresponding level L. All the paths between a given start point S and each inter-edge are computed to build a temporal *intra-edge* as a link with the higher level graph G_l. Similarly for a goal point G, all the paths between G and the inter-edges at level L are computed to build temporal intra-edges that link G to the higher level graph. The performance of this step depends on the computational cost of computing all the intra-edges for S and G. The original HNA* used the A* algorithm to find all the shortest path between S and G to each inter-edges containing S and G points, and it run those searches sequentially in the CPU.

In the second step, once both S and G have been temporally linked to the higher-level graph G_L, pathfinding is calculated by the A* algorithm in the hierarchical navigation graph (HNG) to find the path from n^S_{aux} to n^G_{aux}. The calculated path at the level i of the hierarchy results in the following sequence:

$$ie(n^s_{aux} - v^1_i), v^2_i, v^3_i, ..., v^m_i, ie(v^m_i - n^G_{aux}) \tag{3.1}$$

Where $ie(n^s_{aux} - v^1_i)$ contains the sequence of the nodes at the lowest level of the hierarchy ($L0$) that appear in one of the temporal *intra-edges* added when linking S with the first high level node of the path v^1_i. And similartly, $ie(v^m_i - n^G_{aux})$ contains the sequence of nodes at the lowest level of the hierarchy ($L0$) that appear in one of the temporal intra-edges added when linking G with the first high level node of the path v^m_i. Note that the sequences $ie(n^s_{aux} - v^1_i)$ and $ie(v^m_i - n^G_{aux})$ where calculated in step one of HNA*.

In the third step of HNA*, the intra-edges of each node in the sequence of nodes $\langle v^1_i, v^2_i, ..., v^m_i \rangle$ which were part of the optimal path at level i, are extracted. The final sequence of connected *intra-edges* is the optimal path from a given S and G points at level 0.

In the final step of HNA* (step 4), the temporal nodes n^S_{aux} and n^G_{aux}, and their temporal intra-edges are deleted from the graph to recover the initial HNG.

Algorithm 1 shows the pseudocode of HNA*, and figure 3.3 illustrates the 4 main steps of HNA*.

Running experimental tests with the original HNA* [Pelechano and Fuentes, 2016b] it was observed that for certain configurations of the hierarchy there was

Algorithm 1 Find Path with HNA*

```
1:  procedure FINDPATHHNA*(S, G, L)
2:      if $L = 0$ then
3:          $path \leftarrow FindPathA^*(S, G, 0)$
4:          return $path$
5:      $n_L^S \leftarrow getNode(S, L)$
6:      $n_L^G \leftarrow getNode(G, L)$
    ▷ Step 1: Connect S and G at level L:
7:      $linkNodeToGraph(L, n_L^S)$
8:      $linkNodeToGraph(L, n_L^G)$
    ▷ Step 2: Find path between S & G nodes at level L:
9:      $tempPath \leftarrow findPathA^*(S, G, L)$
    ▷ Step 3: Extract subpaths (intra-edges):
10:     for $highNode \in tempPath$ do
11:         $path \leftarrow path + getIntraEdges(highNode, L - 1)$
    ▷ Step 4: Remove temporal nodes:
12:     $deleteTempNodes(n_L^S, n_L^G)$
13:     return $path$
```

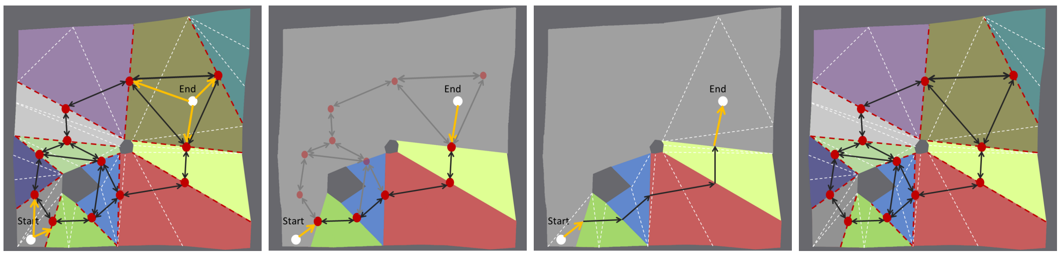


FIGURE 3.3: From left to right we can see the 4 steps of HNA* at L1. Step 1 connects S and G to the HNG by creating temporal connections between S/G and the *inter-edges* of the high level node (yellow arrows). Step 2 computes A* at the HNG (highlights the resulting path). Step 3 extracts the *intra-edges* which contains the sequence of polygons from *L*0. Step 4 removes S/G and the temporal connections to recover the original HNG at L1.

a large performance boost, However, there were certain configurations for which no benefits were observed when running the hierarchical search, and instead there was a performance drop. Experimental studies allowed the previous paper to discover that the source of the problem was the increment of the number of inter-edges per node, which had an impact on the performance of inserting S and G.

So, the bottleneck of HNA* appears in step 1, since it is necessary to compute A* from S and G to each *inter-edge* in their corresponding high level node. This cost increases rapidly with the number of *inter-edges*. And the number of *inter-edges* increases as we add more levels to the hierarchy or merge a larger number of polygons between levels of the hierarchy (for more details we refer the reader to the original paper [Pelechano and Fuentes, 2016b]). This effect has a negative impact on the overall performance of HNA* as it puts an upper limit on the performance benefits of the algorithm. Figure 3.5 and 3.6 show an example where the number of *inter-edges* for a high level node is so large that connecting S and G would be more computationally expensive than simply running A* between *S* and *G* at *L*0.

In this chapter we study in depth the problem to include a mathematical formulation of the source of the problem and include a theoretical upper bound for the performance in section 3.4.1.

The purpose of our work is to build a model that can handle general navigation meshes, without limitations on the implementation or cell shape (triangle, quads, or convex polygons). Our solution works with any NavMesh, and pathfinding over the hierarchy is currently done with A*. However, it is not limited to a specific A*

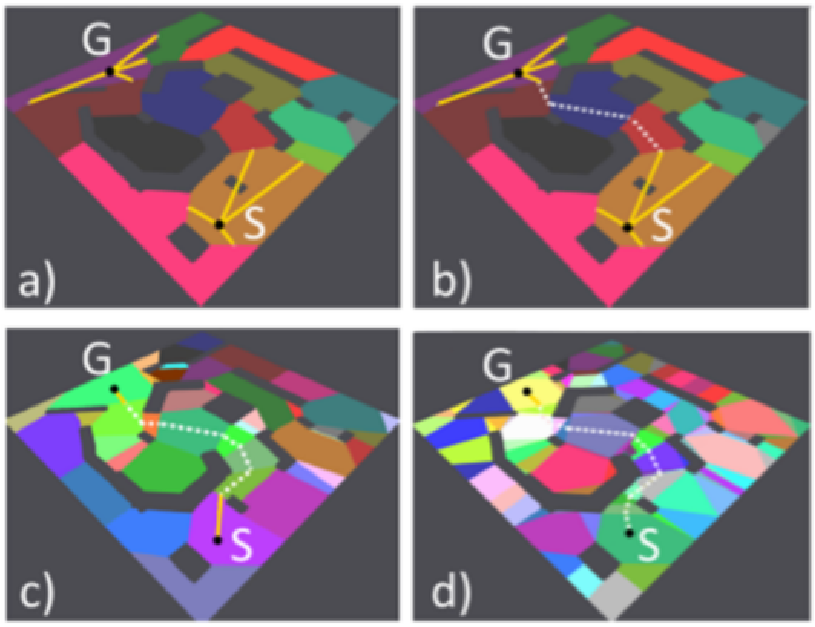


FIGURE 3.4: Pathfinding computation: S and G are inserted and linked to their partitions at level 2 by calculating shortest paths to each portal in their respective node(a). Paths are calculated at level 2 (b), and then *intra-edges* are extracted from lower level 1 (c) and the final path is obtained for level 0 (d) [Pelechano and Fuentes 2016].

implementation, and thus an alternative pathfinding algorithm could also be tested.

This chapter extends the original HNA* [Pelechano and Fuentes, 2016b] by presenting three methods to solve the bottleneck of the Start/Goal connection step. The first one is based on doing further pre-computation and storing additional data that can be rapidly queried during simulation time, the second and third ones are based on exploiting parallelism to compute several connecting paths simultaneously (first on the CPU and then on the GPU).

3.4.1 Theoretical upper bound on the number of *inter-edges*

Each node n_L^i in level L is created by merging μ nodes of level $L-1$. For the first level, $L1$, of the HNG, n_1^i is created by merging μ nodes of $L0$, which are the polygons in the navigation mesh. Each polygon has s sides, so we can work with triangular meshes when $s = 3$, but also with convex polygons of 4 or more sides. Each *side* of a polygon can either be a *portal* (edge between two walkable polygons), or an *obstacle edge* (edge with an adjacent obstacle or the limits of the map).

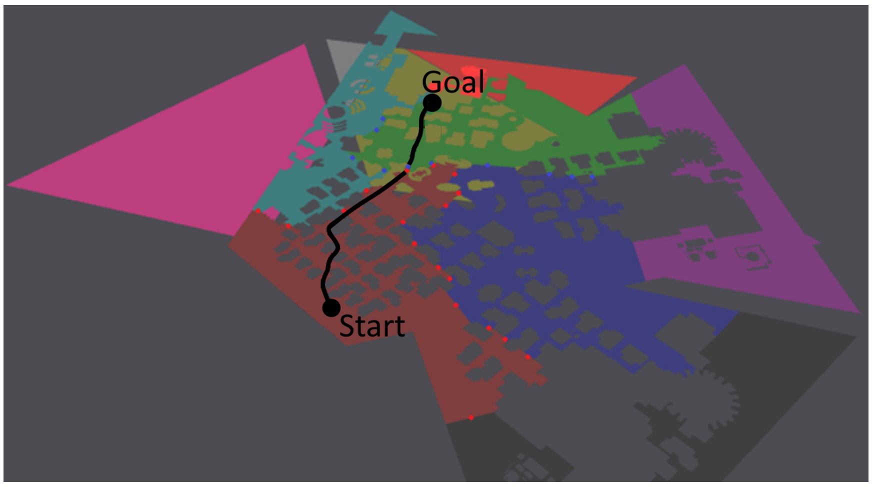


FIGURE 3.5: Example scenario, where connecting S and G becomes a bottleneck due to the large number of *inter-edges*.

When μ polygons are merged to form a node n_1^i (node i in Level 1 of the HNG), then some *portals* will be internal to n_1^i (portals between two polygons that belong to the same high level node, n_1^i), while others will connect a polygon in n_1^i with a polygon in n_1^j (with $i \neq j$). Those portals connecting different nodes in L1 will become *inter-edges* of n_1^i. For the purpose of obtaining an upper bound on the number of *inter-edges* per node, we will consider all polygon edges to be *portals* (this would only be possible if we had a navigation mesh with all polygons having s adjacent walkable cells, but in reality some of those edges will be adjacent to obstacles and thus cannot turn into *inter-edges*).

To compute the upper bound number of *inter-edges* for L1, $I_{(1,\mu)}$, we need to consider the following facts: (1) Merging μ polygons of s sides each to generate n_1^i, means that we have a total of $s \cdot \mu$ edges. (2) Only edges that are not interior to n_1^i can become *inter-edges*, therefore we need to remove those edges that were used for the merging. Merging μ polygons removes at least $2(\mu - 1)$ edges (one for each polygon being merged and assuming only one shared side). Note that for $\mu > 2$ there could be even more, but since we are computing the upper bound of the number of *inter-edges* we will be conservative and assume the minimum possible number of removable edges. Therefore we obtain that the number of *inter-edges* for a node in L1 can be computed with equation 3.2.

$$I_{(1,\mu)} = s\mu - 2(\mu - 1) = \mu(s - 2) + 2 \tag{3.2}$$

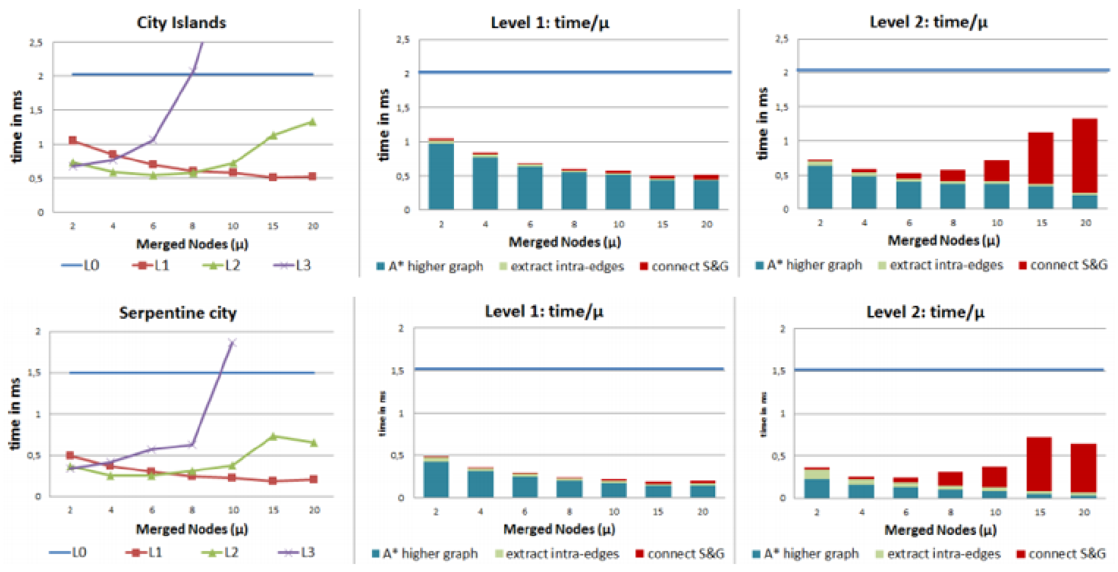


FIGURE 3.6: Performance results for city island (up) and the serpentine city scenario(down) [Pelechano and Fuentes, 2016b].

This equation shows that the number of *inter-edges* for L1 increases linearly with the value of μ, as we had observed empirically in our previous work [Pelechano and Fuentes, 2016b].

Similarly, we can compute the number of *inter-edges* for a node n_x^i in level x of the hierarchy Lx, by merging μ nodes n_{x-1}^i. Following the same logic, *inter-edges* of level $(x-1)$ will become *inter-edges* of level x if they belong to the border between two different nodes at level x.

$$I_{(x,\mu)} = \mu I_{(x-1,\mu)} - 2(\mu - 1) \tag{3.3}$$

Which can be written as (proof in Appendix B):

$$I_{(x,\mu)} = \mu^x (s-2) + 2 \tag{3.4}$$

Equation 3.4 shows the upper bound for the number of *inter-edges* in a node at level x. Therefore, as we build higher levels in the hierarchy, the number of *inter-edges* increases exponentially. This trend was already observed through experimental analysis in [Pelechano and Fuentes, 2016b].

It is important to note that the number of *portals* will be smaller than s since some of the polygon sides will be obstacles. Moreover, the MLkP method merges nodes minimizing the total number of *inter-edges*. Therefore, in practice this upper bound is never reached. However, it proves the impact on the number of $inter-edges$ per node at level x, with respect to μ and x. In the original HNA* algorithm, connecting S and G with the $n+m$ *inter-edges* (n for S and m for G) was done as $n+m$ sequential calls to the A* algorithm. Thus the cost of such step could become prohibitive for certain configurations. With the two alternative approaches presented in this chapter, we are drastically dropping that cost, by either using additional storage or performing the S/G connection step as $n+m$ parallel computations of A*.

3.5 New insert S and G approaches

In this section, we present three alternative solutions to solve this step and we carry out a quantitative evaluation of their advantages and limitations. The first solution focuses on storing further data, while the other two propose parallel implementations in both CPU and GPU.

3.5.1 Pre-calculated connecting paths (PCCP)

The simplest way to solve this problem consists of pre-storing further information to speed-up the connection step. We can calculate the A* path from a point p in each polygon at level 0 ($L0$ which corresponds to the original navigation mesh) to the *inter-edges* that appear in the higher level node of the hierarchy (in *L#* where # represents the number corresponding to the highest level in the hierarchy). Therefore during the on-line phase it is only necessary to determine which polygon of L0 contains S, and extract the set of paths that connect p with the high-level graph without the need to run A* between p and each inter-edge (from now on, since the algorithm is the same for both S and G, we will only refer to S).

Therefore the method includes an off-line and an on-line phase. In the off-line phase, the center point p_c of each polygon at L0 is calculated and the shortest paths and cost from p_c to the *inter-edges* in *L#* are calculated using the A* algorithm and stored in memory using a MultiMap hash table. Table 3.1 shows an example of such a table. This table has for each cell, one entry per *inter-edge*, with the *Path* and *Cost* information returned by A*. The *Path* stores the sequence of polygon IDs at L0 that the agents will have to walk through to go from that cell to the corresponding *inter-edge*. *Cost* indicates the length of the path from the center of the cell to the *inter-edge*.

Connects to indicates the high level node reachable with that path. In this particular example, there are 3 *inter-edges* for polygon 18, and thus for entry polyID=18 we can find 3 alternative paths with their corresponding cost. Note that one of the entries for polygon 17 does not show any path for one of the *inter-edges*, because one of its segments is already an inter-edge. These paths would be the temporal connecting edges with the high-level graph (HNG) in order to compute A* at the higher level of the hierarchy during the on-line phase of the algorithm.

TABLE 3.1: Structure of MultiMap for some example nodes in Figure 3.7.

Polygon	Path	Cost	connects to
p_{10}	13-16	12.5	n_1^1
p_{10}	12-11-14	16.3	n_1^3
p_{17}		2.7	n_1^2
p_{17}	22-23-21-19	22.8	n_1^5
p_{17}	22-23-21-19	21.3	n_1^5
p_{18}	20-15	15.3	n_1^2
p_{18}	20-29	11.4	n_1^4
p_{18}	24	4.9	n_1^1

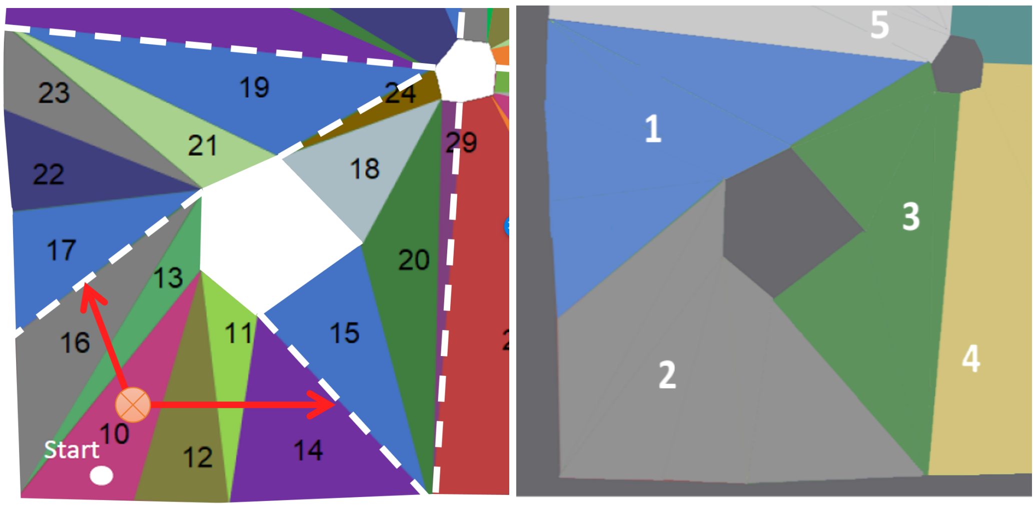


FIGURE 3.7: Section of the example map for L1 with $\mu = 6$. On the left, map at $L0$ with numbers indicating polygon IDs. On the right, L1 of the HNG with numbers indicating node IDs at L1.

Algorithm 7 shows the off-line phase of our method. For navigation meshes, it is necessary to compute the exact path from the center of each polygon, since we cannot assume that the shape and size of all cells is the same as it happens with 2D regular grids. It is important also to note that center points are computed simply to obtain estimated distances to *inter-edges*, since the real S/G points could lie anywhere in the polygon. However this does not imply that the local movement of the agent has to cross the center point. Agents are simply steered towards the portal connecting to the next cell in their paths, without necessarily going through the center. Since our navigation mesh guarantees that all cells at L0 are convex, then paths are free of collisions against the static geometry and collisions against other moving agents can be handled through steering techniques [Pelechano, Allbeck, and Badler, 2007].

During a path search, HNA* needs to find the node containing S and connect it to the high-level graph. The algorithm checks the ID of the polygon containing S, obtains its center position and extracts the temporal edges from the MultiMap table containing PCCP. We thus simplify the connect step with a query for the stored paths of nodes S and G as opposed to computing A* sequentially for each *inter-edge* of the node $n^S_\#$ (node of level # containing S) and $n^G_\#$.

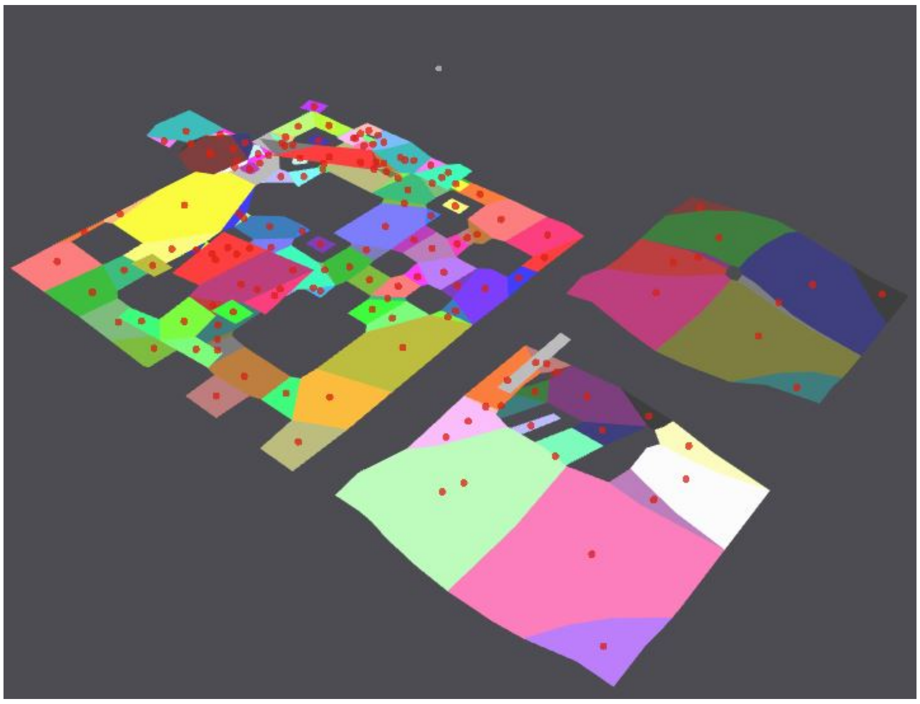


FIGURE 3.8: Center of each polygon in level 0 computed for computation and storage of shortest path to each inter-edge.

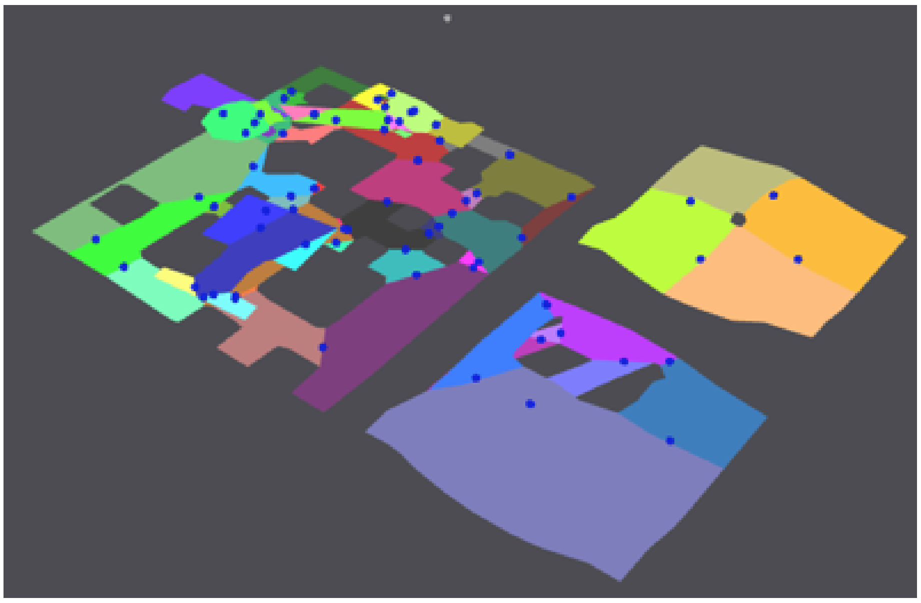


FIGURE 3.9: Inter-edges of each polygon in level 1 of hierarchy

3.5.2 Parallel Search on CPU

From a conceptual point of view, the problem of connecting S and G to the high-level graph can be run in parallel as there are no inter-dependencies between path searches. It should be thus possible to compute simultaneously all connecting paths between S/G and the corresponding *inter-edges*. Such parallel computation can be done either on the CPU or the GPU.

To exploit the parallel hardware architecture in depth, the algorithm should be adapted to run concurrently using multiple threads. The algorithm should be changed to use multi threading, shared memory access, and achieve concurrency controls. The connecting S and G step is a highly parallelizable problem, as we can simply run each A* search in a different thread. For the adoption of A* using multiple threads, some improvements are required in the basic algorithm. In order to find a path from S/G in a polygon to their corresponding *inter-edges* using multiple threads per polygon, we have used N threads concurrently to find an optimal path where $N = n + m$ with n being the number of *inter-edges* in $n_{\#}^{S}$ and m the number of *inter-edges* in $n_{\#}^{G}$. These threads work concurrently so that each thread calculates the optimal path from S or G to one of the *inter-edges* in the corresponding node $n_{\#}^{S}$ or $n_{\#}^{G}$. For example, if we have a node with 4 *inter-edges*, then we will have 4 threads where each thread works individually to find an optimal path from S or G to a specific *inter-edge*. Algorithm 3 shows a group of threads is created to carry out such computation. This

Algorithm 2 Calculate Connecting Path

```
1: procedure CALCULATE_CONNECTING_PATH()
2:     $N \leftarrow NumOfPolygons$                      ▷ in L0
3:     $C \leftarrow NumOfCluster$                       ▷ in L1
4:     for $i \in [1..N]$ do
5:         $SId \leftarrow GetPolygonID[i]$
6:         $S \leftarrow GetPolygonCenterPos[i]$
7:         for $k \in [1...C]$ do
8:             $CId \leftarrow InterEdgeID[k]$
9:             if $SId = CId$ then
10:                $G \leftarrow InterEdgePos[k]$
11:                $\{PolyId, Path, Cost\} \leftarrow Astar(S, G)$
12:                $SavePCCP(PolyId, Path, Cost)$
13:            end If
14:        end for
15:    end for
```

algorithm is run for both *S* and *G* points. Our implementation uses the Boost library [*BOOST* 2017].

Algorithm 3 Thread Definition

```
1: procedure GET_PATH
2:     $N \leftarrow NumOfPolygonInterEdgs(i)$
3:     boost::thread_group grp
4:     for j:=1 to N do
5:         grp.create_thread(boost::bind(LinkToGraph,this,N))
6:     end for
7:     grp.join_all()
8:     End
```

3.5.3 Parallel search on GPU (CUDA HNA*)

The CPU usually contains several highly optimized cores for sequential instruction execution, while the GPU typically contains thousands of simpler but more efficient cores that are good at manipulating different data at the same time. In addition, the

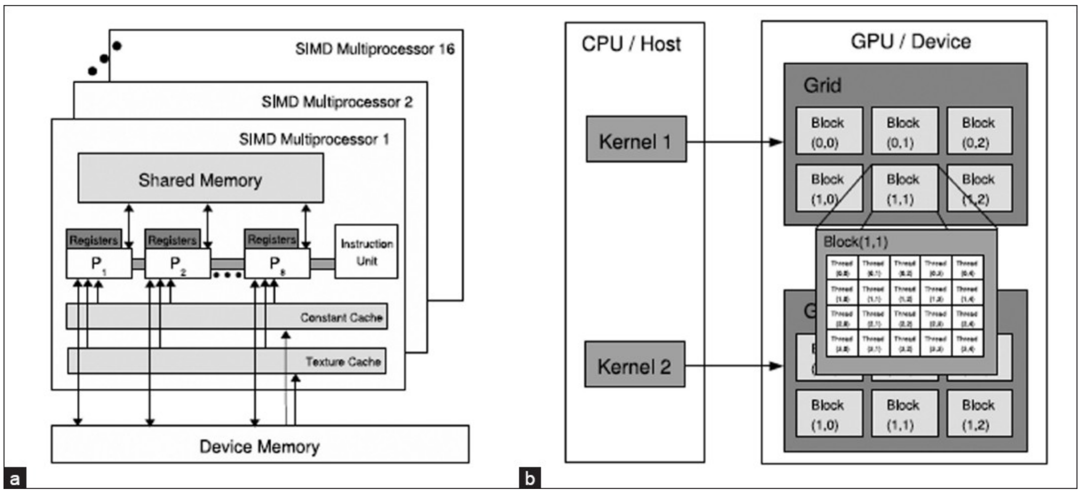


FIGURE 3.10: GPU architecture; (a) CUDA hardware interface, (b) CUDA software interface

GPU has a memory system which is independent of that of its CPU. Such a design provides a higher bandwidth for accessing the global memory. In other words, cores of a GPU can retrieve and write data from/to the global memory much faster than a CPU [Zhou and Zeng, 2015].

When several paths are being calculated in parallel in the multi thread implementation, the Binary heap used for computing A* can become a bottleneck because it stores the information in local memory. The A* search algorithm usually requires many accesses to the global memory (especially in big scenarios) for storing and retrieving nodes to/from both the open and the closed lists. The A* algorithm also needs higher global memory bandwidth which can lead to a faster expansion rate during A* search.

In order to overcome this weakness and speed up the search process, we decided to use the GPU shared memory facility. All the required data is stored into shared memory before any computation takes place. We thus benefit from using the shared memory which is much faster than local and global memory, because it is on-chip memory. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. We have used the CUDA platform in order to implement our search algorithm [*NVIDIA. CUDA* 2017]. CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. Figure 3.10 shows GPU and CUDA architecture.

A program designed to run on the GPU is called a kernel, and in CUDA the level of parallelism for a kernel is defined by the grid size and the block size [Nickolls et al., 2008]. One of the most important factors that can have an effect on parallelism performance is the degree of parallelism (DOP), which in our case corresponds to the number of *inter-edges* N (counting for both nodes of the high level graph containing S and G). We have defined a kernel with one block for the polygon containing S and another for G, plus n or m threads per block respectively. Algorithms 4 and 5 show our CUDA parallel method. The first argument in the kernel execution configuration specifies the number of blocks in the grid, and the second specifies the number of threads in a block.

Algorithm 4 Thread Definition

```
1: procedure PARALLEL_FINDPATH
2:     $N \leftarrow NumOfPolygonInterEdgs(i)$
3:     CudaMalloc(Device Data)
4:     CudaMemcpy(Device Data, cudaMemcpyHostToDevice)
5:     Findpath «<1, N »>(node_i_data , Device Data)          ▷ Kernel function
6:     CudaMemcpy(Host Data, cudaMemcpyDeviceToHost)
7:     End
```

Algorithm 5 Kernel Definition

```
1: procedure PARALLEL_FINDPATH
2:     —global—void FindPath(node_i_data)
3:     —shared—Piority_Queue, device data;
4:     int t = blockIdx.x*blockDim.x + threadIdx.x;
5:     Astar_FindPath(t, node_i_data , Device Data);
6:     —syncthreads();
7:     End
```

3.6 Experimental Results

In this section we present the results achieved in terms of performance, but also discuss the limitations of each approach. All methods described in this chapter have been implemented using C++ and CUDA, with an Inter Core i7 Cpu @3.5 Gz, 1 MB L2 cache and 8MB L3 cache, 16 GB RAM. We have used an Nvidia GTX 420 with 2.4 GB off-chip global memory and 2496 CUDA core.

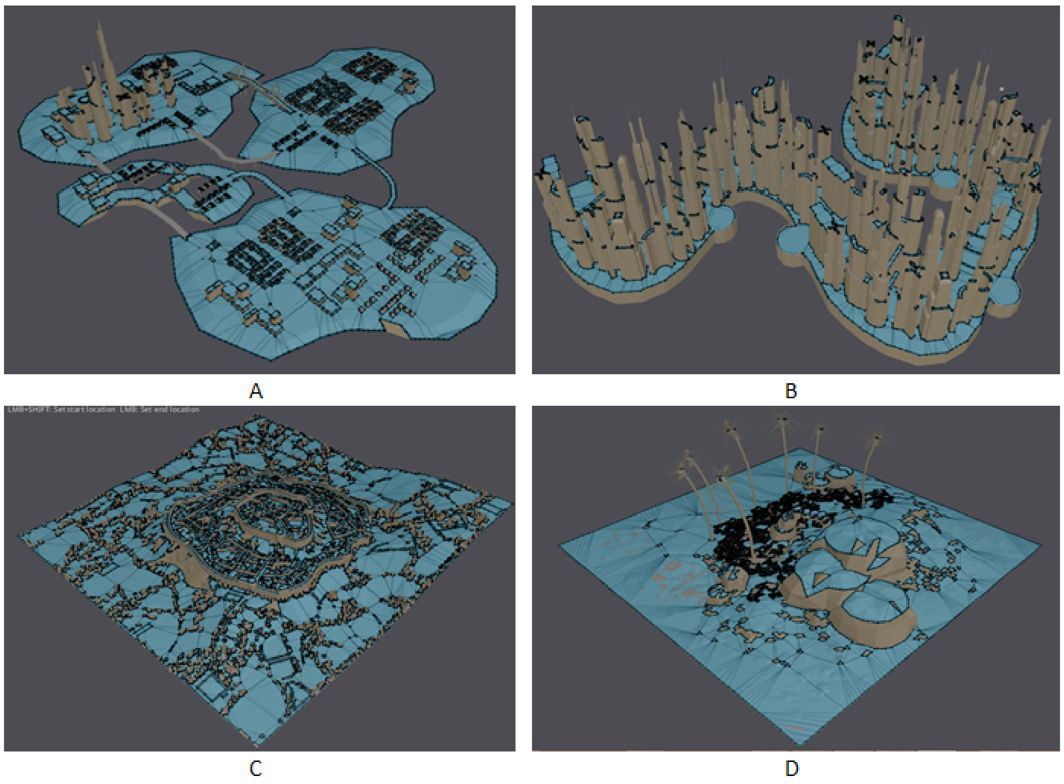


FIGURE 3.11: Different scenarios with their corresponding number of triangles in the mesh. A: City Island (110.3K), B: Serpentine City (135.1K), C: Medieval City (774.7K) and D: Big Tropical scenario (239.1K).

3.6.1 Game world geometry

Typically the world geometry in a game is stored in a structure called a map. This geometry is given as a polygon soup and pathfinding can be computed over the map which is simply an abstract representation of the walkable space [Graham, McCabe, and Sheridan, 2015]. Generally, in order to reduce the search space of the game world for pathfinding, a game map is broken down and simplified. The pathfinder then uses this simplified representation of the map to determine the best path from the starting point to the desired destination in the map [Botea, Müller, and Schaeffer, 2004]. One of the most common forms of simplified representations is the Navigation Mesh which is used for example in the Recast Navigation tool [“Recast” 2017]. In order to evaluate our methods, we have used several maps with different sizes (see Figure 3.11) for the purpose of running a fair comparison we have used the same maps from the original HNA* paper [Pelechano and Fuentes, 2016b]).

Map Name	Geometry # Triangles	NavMesh # Poly
Serpentin City	135.1K	3.9K
City Island	110.3K	5.5K
Medieval City	774.7K	16.9K
Tropical Island	239.1K	12.7K

3.6.2 Error and memory usage in PCCP

As we expected, the pre-calculated paths method (PCCP) achieves the best performance. However, it requires additional memory and also introduces a small offset between the real position of S/G and the center position of each polygon. Therefore we need to measure the impact of both memory and offset in the results obtained. Figure 3.12 shows the memory usage in 5 different scenarios of a variety of sizes (shown as number of triangles in the original mesh).

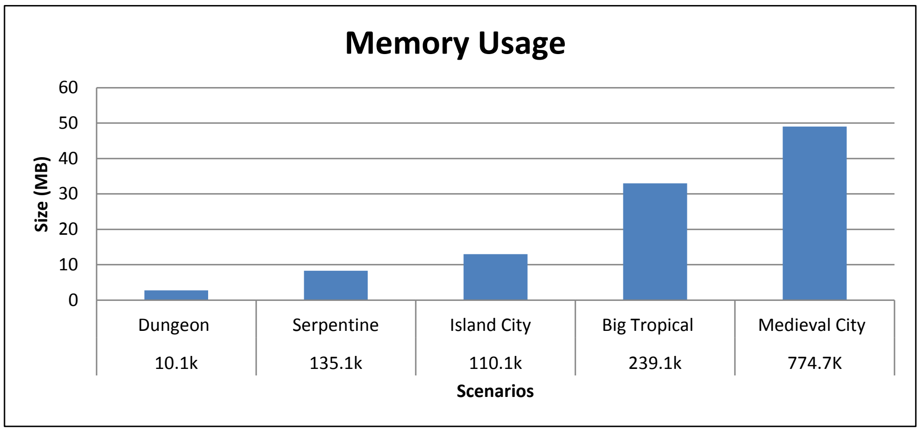


FIGURE 3.12: Memory usage in 5 different size scenarios.

Memory usage increases with the size of the scenario, as it requires to pre-compute and store local connecting paths for each cell in the navigation mesh (Figure 3.12). The allocated memory for the Dungeon scenario with 119 polygon is 2.9 MB while the allocated memory for the Medieval City scenario with 16,867 polygons is 49.6 MB. Memory could be further reduced by storing only the next cell as opposed to the whole path in the hash table. However, this would require further accesses to the hash table, thus reducing performance during online pathfinding. In any case, the memory usage in PCCP is insignificant for our tested scenarios, so these results confirm that PCCP can be a simple yet powerful way of eliminating the bottleneck of connecting S and G in the original HNA* algorithm. Also note that this information depends on the number of *inter-edges* in each cell, and it is meant to be used for as many path searches as needed. Therefore, memory size is independent of the number of path searches being computed during the online phase of the algorithm.

In PCCP we have computed paths and costs from the center of each polygon to the *inter-edges* of its cell and stored them in a hash table. When inserting the new S and G points in any location of a polygon, the algorithm queries the hash table for paths with the IDs that correspond to those polygons containing S and G. Undoubtedly this introduces an offset between the center positions of the polygons and the real S and G positions. However, this offset represents only a marginal error when compared to the total length of the path (in most cases, it simply adds a small offset at the beginning and at the end of the total path). The reason is that S and G will not necessarily be located at the center of the cell, and yet the HNA* search is done assuming that they are. However, it is also important to emphasize that this offset simply affects the global path computation, and not the local path, as agents are not forced to walk through the center points (See figure 3.13).

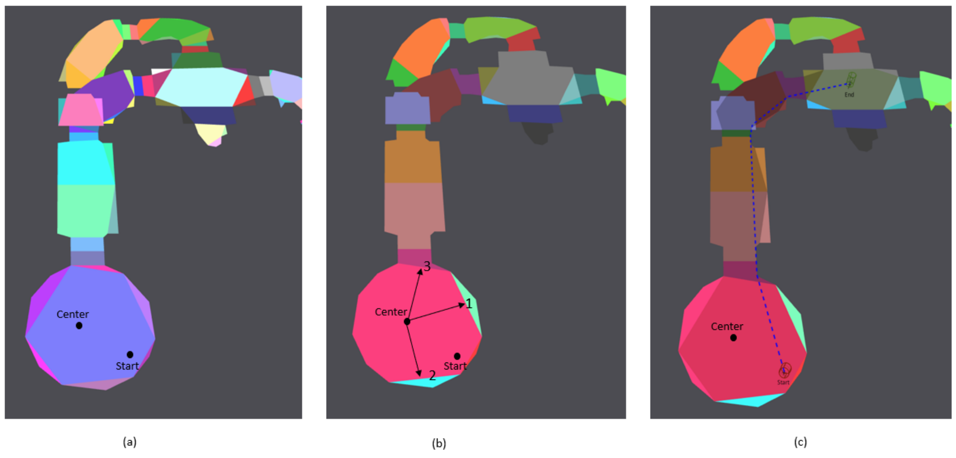


FIGURE 3.13: (a) Hierarchical representation at level L_0, (b) Hierarchical representation at level L_1 with the path pre-computation from the center of the polygon to all *inter-edges*, (c) Final paths computed between the Start and Goal points.

Figure 3.14 shows the difference in path length between the proposed pre-calculated path method (PCCP) and the original HNA* method. For this figure, 100 random S/G position were used to compute paths with PCCP and HNA* in the Medieval City scenario that appears in Figure 3.11c. The horizontal axis shows the 100 paths sorted in ascending order based on the distance between S and G. Our experimental results show a small impact on the total length of the path (3% on average for paths over 100m, and 5% on average for shorter paths).

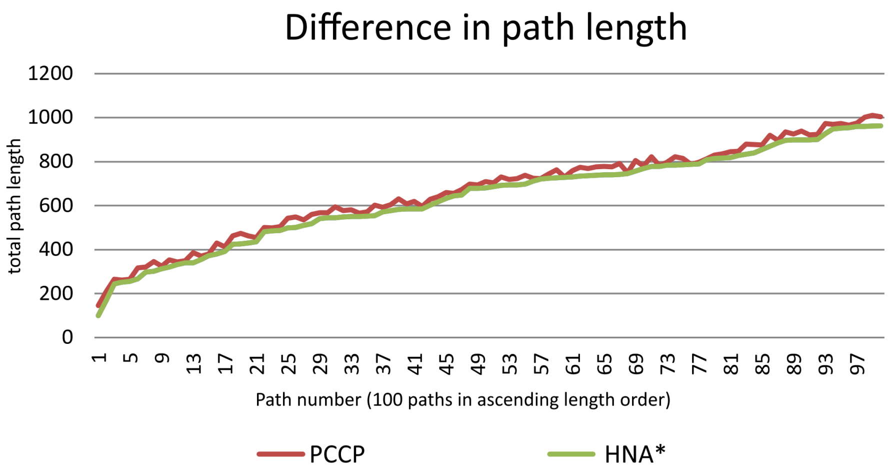
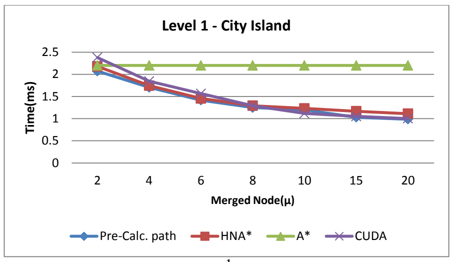


FIGURE 3.14: Difference in the total path length between PCCP HNA* and the original HNA*.

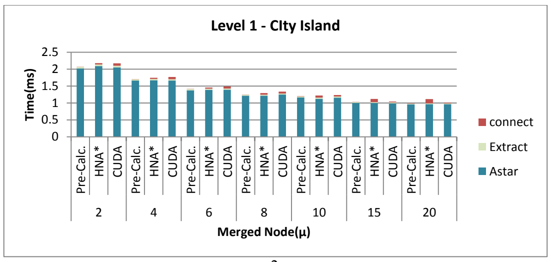
3.6.3 Performance results for PCCP

For the evaluation of this method we have used several 3D scenarios as shown in Figure 3.11, with increasing numbers of cells in the original NavMesh and different hierarchical configurations. To compare the overall computational time of our pre-calculated paths method against HNA*, we have computed the average cost of calculating 100 paths. Paths are computed for up to 3 levels in the hierarchy and increasing values of $\mu = \{2,4,6,8,10,15,20\}$, where μ indicates the number of nodes merged from one level to the next one of the hierarchy. Results show that we can achieve significant speed-ups for all configurations using PCCP, as opposed to the original HNA* which suffered from a bottleneck in the connect step (in red).

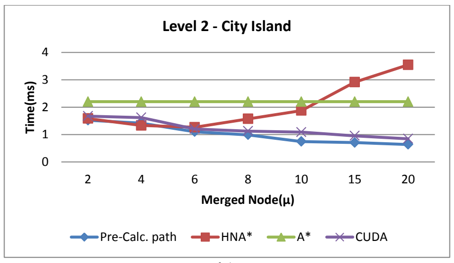
For the City Island scenario, we can see in Figure 3.15-a1 the average cost of performing A* in this scenario is 2.2ms (Note that A* is always computed on the navigation graph at L0, and thus it is not affected by the hierarchy configuration). Figure 3.15-a1 shows that the performance of the PCCP method at $L1$ is not significantly faster than the original HNA*; this is due to the fact that at $L1$ the connecting S and G step does not represent an important bottleneck as can be appreciated in Figure 3.15-a2, and thus there is not a big difference between the two methods. The strength of the pre-calculated paths (PCCP) can be observed for higher levels of the



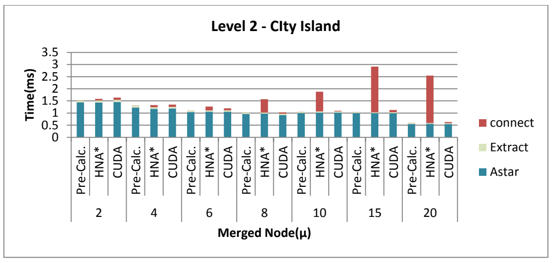
a1



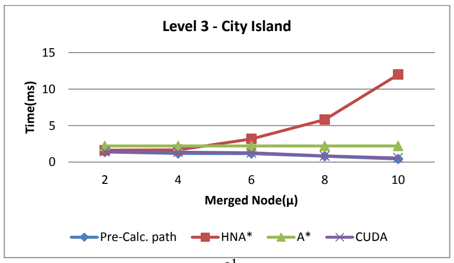
a2



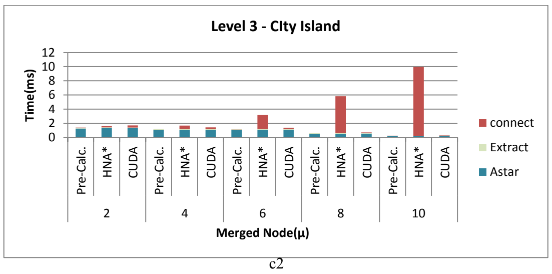
b1



b2



c1



c2

FIGURE 3.15: Performance results for the city Island scenario.

hierarchy. Figure 3.15-b1 and Figure 3.15-c1 show significant performance improvements when compared against HNA*. These improvements can be seen in Figure 3.15-b2 and Figure 3.15-c2 where we have clearly managed to drop the cost of the connecting S and G step. Compared to A*, PCCP provides its largest speedup (9.3x faster) for $L = 2$ and $\mu = 10$.

Results are similar for the Big Tropical Island (Figure 3.16). The average cost of performing A* in this scenario is 1.7ms. At L1, there is not a large performance gain, since the bottleneck of inserting S/G in HNA* is negligible. Our results show performance gain for all the values of μ tested ($\mu \in [2,20]$) at L1. The advantages of the new implementation are noticeable for L2 and L3 after a specific value of μ. HNA* had a performance of 2.06ms for L2 and μ = 20 and 9.9ms for L3 and μ = 10 while PCCP HNA* obtained paths in 0.39ms for L2 and μ=20 (4.3x faster than A*), and 0.25ms for L3 and μ = 10 (6.8x faster than A*).

Similar results were obtained for the Medieval city scenario (A* performance of

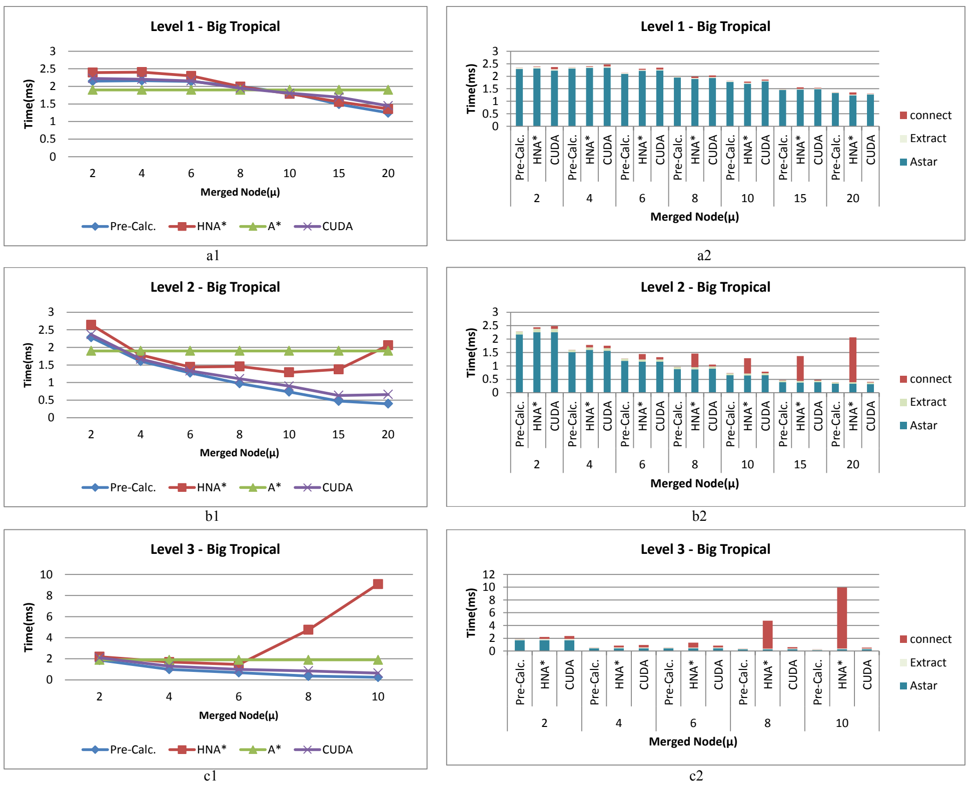


FIGURE 3.16: Performance results for the big tropical scenario.

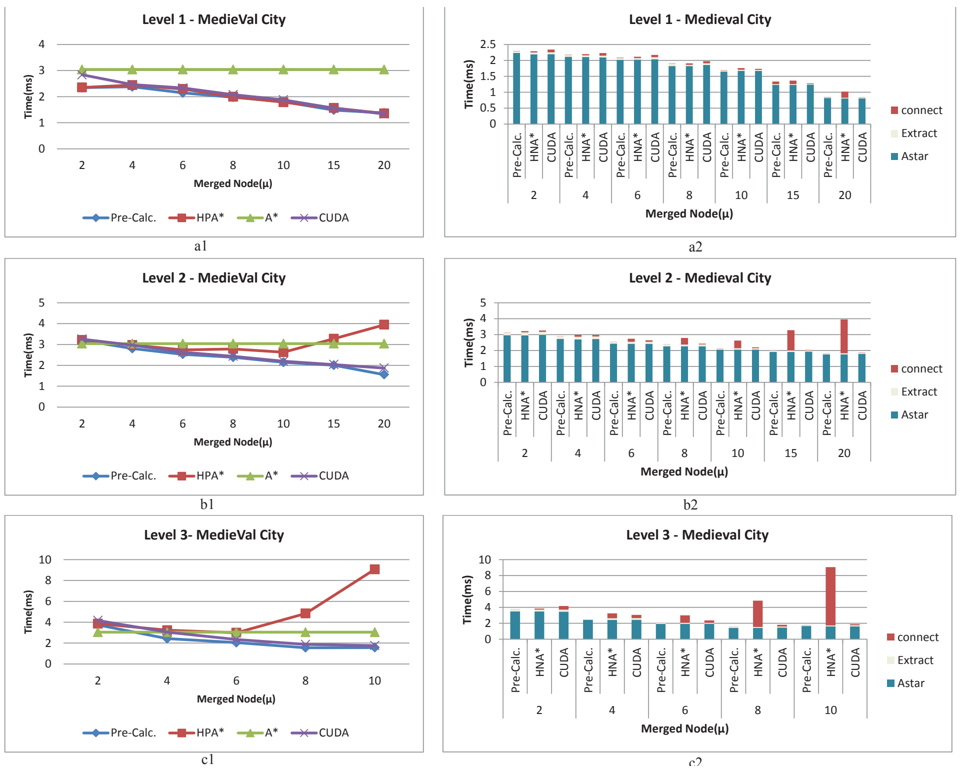


FIGURE 3.17: Performance results for the Medieval city scenario.

3ms) (Figure 3.17). The original HNA* method suffered from the insert S/G bottleneck after a specific value of μ. With 1.28*ms* in L2 and 3.29*ms* in L3 for μ = 15 while PCCP HNA* had a computational time of 1.83*ms* (1.6x faster than A*) in L2 for μ = 20 and 1.76*ms* (1.7x faster than A*) in L3 for μ = 10, thus offering a speed-up for all configurations.

3.6.4 Achieved Results of parallel search on the CPU

In order to evaluate our parallel CPU method, we have carried out experiments with the same set of scenarios (Figure 3.11).

As explained earlier in this chapter, in parallel programming the performance of the method depends on the degree of parallelism of the problem to be solved (DOP), which in our case corresponds to DOP=N, with N being the number of *inter-edges*.

The number of *inter-edges* can rapidly increase with the number of levels in the hierarchy and the number of merged nodes as shown in Figure 3.18 for the example of *L*2.

As we can see in Figure 3.19 with increasing DOP (number of *inter-edges* in our work) the total cost of our parallel CPU implementation reduces the cost of connecting S and G step, but it converges. This is due to the fact that even though the increment of μ also increases the value of the DOP, the overhead of multi-threading outweighs the gains achieved. Moreover, although the memory access is trivial in the sequential version, with the parallel implementation, the threads have to share memory which can take more time than for the sequential version.

As it is obvious from the results in Figure 3.19, the pre-calculated path and the Multi-threads implementation are much faster than the original HNA* implementation on the CPU. However the pre-calculated path method still shows the most efficient results. HNA* and parallel CPU method exhibit similar results for small values of μ (i.e.while the number of *inter-edges* does not represent a bit bottleneck in HNA*). However for larger values of μ, the cost of inserting S and G in HNA* can become increasingly expensive compared to pre-calculated path method or CPU parallel method. CPU parallel is more costly than pre-calculated because the binary heap used to implement the priority queue of A* can turn into a bottleneck in Multi thread implementations. The reason is that even though N (number of *inter-edges* per polygon) threads run in parallel, when it comes to inserting values in the binary

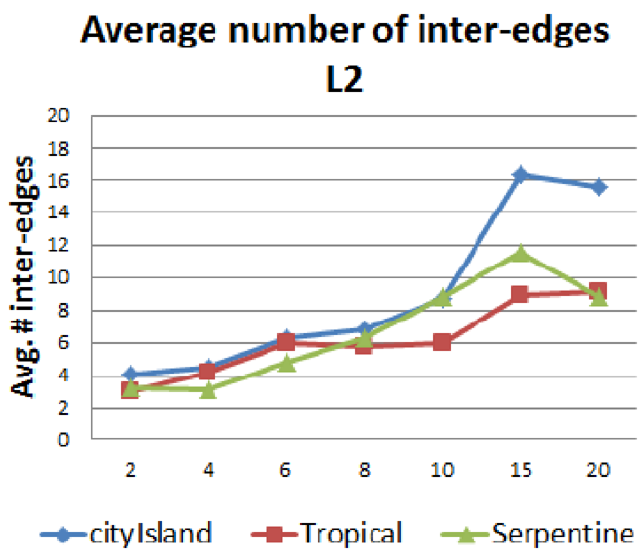


FIGURE 3.18: Average number of *inter-edges* per high level node for L2 as the value of μ increases.

heap, only one thread can remain active and all the other threads have to wait.

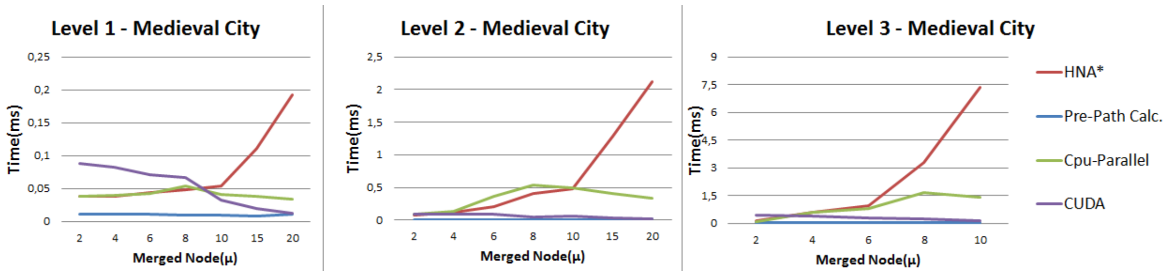


FIGURE 3.19: Performance cost for inserting S and G step with the parallel implementation on the CPU. Results shown the Medieval city scenario using a hierarchy of 3 levels.

3.6.5 Achieved Results of Parallel Search on the GPU

To calculate the overall computational time of our CUDA parallel method and compare it against the Pre-Calculated path (PCCP) and the original HNA* method, we have once again computed the cost of calculating 100 paths in the same scenarios and configuration (see Figures 3.15, 3.16 and 3.17. The CPU used in these experiments is also an Intel core i7-4770 CPU@3.5Gz with 16GB global memory. The graphic card (GPU) that we used was a single NVIDIA Geforce GTX 420 with 2.4GB off-chip global memory and 2496 CUDA cores.

For the City Island scenario consisting of a NavMesh with 5,515 polygons, we have tested the same levels and values of μ as in previous results. Figure 3.15-a1 shows that the average cost of performing A* in this scenario is 2.2 ms. Figure 3.15-a1, for $L1$ of hierarchy and $\mu = [2, 20]$ the performance of Pre-calculated path method is faster than both CUDA parallel method and HNA*, which CUDA outperforming HNA*. As in previous experiments, the performance difference is not significant for L1, but for L2 and L3 it becomes highly significant. The time of computing a path for $L2$ and $\mu = 20$ is down to 0.648ms, and for $L3$ and $\mu = 10$ is down to 0.561ms for CUDA and 0.411ms for Pre-calculated path method.

As we can see in the right column of Figure 3.15, CUDA has a slightly higher cost when inserting S and G than the pre-calculated path. However the difference is negligible while saving memory footprint and avoiding the offset between S/G and the center point of each polygon.

Figure 3.16 shows the comparative results for the same configuration in the case of the Big Tropical scenario. Similarly as for the previous scenario, the performance differences are not relevant to L1, but show drastic improvements from L2 onward. For instance, the performance of CUDA in L2 and for μ=20 drops to 0.659ms while the performance of HNA* increases up to 2.058ms. However, the performance of Pre-Calculated paths is still faster than CUDA in Level 2 with the time being 0.396ms for μ=20.

Finally, we have obtained similar results for the Medieval city scenario (Figure 3.17) which consists of a NavMesh with 16,867 polygons. As in previous results, the differences in performance results become noticeable from L2 onwards.

The right column shows that in HNA*, the time of connecting S and G points increases up to 7.43ms in L3 for μ =10 whilst it drops to 0.14ms for CUDA, and 0.011ms for Pre-calculated path.

3.7 Conclusion

In this chapter we have studied the problems of pathfinding in large Scenarios for hierarchical representations based on navigation meshes. Our results have provided improvements over the basic HNA* algorithm. The problem with the original HNA* method was very limiting, because it could not guarantee speed-ups for any configuration, and thus it required the programmer to test many configuration on a given map to obtain the optimal one. This could be very time consuming and made it difficult to incorporate any scenario into a game. The main concern of this chapter, was thus to study the source of the connecting S/G problem, and to propose a theoretical formulation for both the hierarchical pathfinding problem and for the upper bounds of the bottleneck. Once the source of the problem was found, we focused on proposing and testing alternative solutions.

The first improvement that we have presented consists in the computation of pre-calculated paths from the center of each polygon in L0 (lowest level of the navigation mesh) to the *inter-edges* of the corresponding cell in the higher level of the hierarchy. Those paths are then stored in a MultiMap hash table and can be accessed efficiently during the on-line search.

Given the highly parallel nature of our problem, the second improvement that we have implemented, consists in having a multiple threads version of the basic HNA* algorithm on the CPU. In this implementation we have used threads in order to calculate paths concurrently for each A* search between S/G and the *inter-edges* of the high level node.

Finally our third approach consists in a parallel version of HNA* on the GPU using CUDA. To evaluate our different methods we have tested several 3D scenarios with increasing numbers of cells in the their navigation mesh and different hierarchical configurations by increasing the number of merged polygons. Our results show that both the Pre-calculated Paths method and the CUDA version are faster than the original HNA* but Pre-calculated path method requires more memory usage than others. For all tested scenarios, the performance improvements are not very significant for L1, but they become very relevant from L2 onward, as they eliminate the bottleneck of HNA* which was the connect S and G step.

With the algorithms proposed in this chapter, we have eliminated the bottleneck from HNA* and thus obtained hierarchical pathfinding algorithms that are suitable for any navigation mesh and for any hierarchical configuration. We have achieved high speed-ups for a much larger number of scenarios regardless of the configuration. It is now up to the programmer to determine whether speed is the most critical issue even if it requires increasing the memory footprint (PCCP method) or else it is better to save memory by using parallel computation. In this second case, the best results can be achieved by using the GPU, however our parallel CPU implementation could still be used in cases where it is not possible to use the GPU (either because there is none or because it needs to be fully dedicated to rendering purposes as it often happens in video games).

Chapter 4

Multi-agent parallel hierarchical pathfinding in navigation meshes (MA-HNA*)

4.1 Introduction

In the previous chapter we presented three methods to solve the original HNA* bottleneck, and obtained a new version of HNA* that enhances performance for any hierarchical configuration. Our first method relies on further memory storage, and the other two use parallelism on either the CPU or the GPU. In this chapter we propose a parallel implementation using our novel HNA* methods to handle multi-agent pathfinding. Since finding path for each agent based on HNA* is completely independent to other agents, we could compute the path for each agent in parallel. In this chapter, we consider the problem of concurrent Decentralized and Non-communicating multi-agent path planning in which agents can act in parallel at each time step with partial information and individual goals so that the problem of synchronizing the movement of agents is not addressed.

In order to paralyze this computation, we have used GPU blocks and threads. In this chapter we studies in depth the GPU architecture to maximize the parallel computation abilities for combining HNA* with the multi-agents pathfinding problem. Our experimental results show that we can compute over 500K paths simultaneously in real-time taking advantage of parallel computing using CUDA and HNA*, with speed-ups above 15x faster than a parallel multi-agent implementation using A*.

4.2 Problem formulation

The Multi-Agent Pathfinding problem (MAPF) is formalized as a graph $G(V, E)$ and one set of agents $A = \langle (a_0, s_0, g_0), (a_1, s_1, g_1), ..., (a_n, s_n, g_n) \rangle$, where $s_i \in V$ is the start node position for the agent a_i and $g_i \in V$ is the goal node position for the agent a_i. A solution to the problem is a list of paths $\langle P_0 ... P_n \rangle$ each of which takes the corresponding agent from its start position to its goal position, where P_i includes a set of nodes $(s_i, n_1, n_2, ..., g_i)$ where $s_i, g_i, n_t \in V$, that agent a_i walked through. Each pair of sequential nodes on the path should be connected by an edge, $e_t \in E$. In some cases, there is a further constraint, which is that no agent may have in its path the same node or edge as another agent at the same time step; therefore $P_i(n_t) \neq P_j(n_t)$ and $P_i(e_t) \neq P_j(e_t)$, where $e_t \in E$ and e_t is the edge that connects n_t and n_{t+1}. However, this constraint does not apply for the case of navigation meshes, since a cell is a convex polygon where several agents could be walking by at any given time, and an edge is the segment shared by two polygons which can also be crossed at any time by several agents. Conflicts in those situations are sorted through the local movement algorithm being used to steer agents from one point to another. Not having to consider other agents' trajectories during pathfinding, makes it easier to parallelize the multi-agent pathfinding problem.

Every computed path is given a computational cost, $\langle C_0 ... C_n \rangle$, which is computed to be the sum of the cost of the moves (actions) taken by the agent on its path to reach the goal location. The moves representing traversing cells, and it is calculated as the distance from the center of one cell to the center of the next cell. Note that in navigation meshes where each cell corresponds to a convex polygon of different shape and size, the cost of traversing portals can vary a lot from one cell to another. The cost of a solution is the sum of the costs of all of the individual paths that comprise the solution [Kraft, 2017].

4.3 Related work on Multi-Agent pathfinding

Modern video games require efficient pathfinding to support large numbers of agents moving through expansive and increasingly environments. There has been many works focusing on Multi-Agent Path Finding (MAPF). Bounded Multi-Agent A*

(BMAA*) [Sigurdson et al., 2018] is a real time heuristic search algorithm for multi-agent path-finding. In this method, each agent operates its own real-time heuristic search and treats the other agents as moving obstacles. In this method, agents do not share their path or their heuristic values with each other. In BMAA*, each agent executes its individual copy of RTAA* algorithm.

Li et al [Li et al., 2019], proposed a LAMAPF method (MAPF for large agents pathfinding) which is an adapted version of Conflict-Based Search (CBS), to solve LA-MAPF, called Multi-Constraint CBS (MCCBS). The MCCBS adds multiple constraints instead of one constraint for an agent when it generates a high-level search node.

To run searches for thousands of agents simultaneously, Caggianese et al. exploit the fact that given a set of start and goal points, it is likely that the explored paths share sub paths with other agents [Caggianese and Erra, 2012]. The method tries to determine these shared sub paths by computing simultaneously all potential sub path types inside the planning blocks and considering that the sub path should converge toward the goal position. This method is thus limited to all agents sharing the goal position.

Parallel computing for multi-agent pathfinding has also been used for other types of navigation meshes, such as 2D regular grids [Garcia, Kapadia, and Badler, 2014], and triangulations [Farias and Kallmann, 2019], where the approaches are strongly dependent on the specific implementation of the grid or the triangulation. Therefore neither solution can be easily applied to a general navigation mesh given as an input.

Some researchers have proposed methods based on parallelizing a single path search (typically loosing path optimality just like with hierarchical approaches [Caggianese and Erra, 2012]), whereas others have kept A* and applied parallelism to compute multiple agents' paths simultaneously. Most of the effort focuses on finding the best way to distribute work and syncing tasks, to make the most of the GPU kernels and threads, while managing correctly memory accesses. Caggianese and Erra proposed a parallel version of A* using a grid map decomposition and CUDA, and obtained results that run faster than a GPU implementation of Real-Time Adaptive

A* (P-RTAA*) [Caggianese and Erra, 2012]. Merrill et al. presented a parallelisation of BFS (Breadth First Search) tailored to the GPU's requirement for large amounts of fine-grained, bulk-synchronous parallelism [Merrill, Garland, and Grimshaw, 2012]. Ortega-Arranz et al. presented a parallel implementation of Dijkstra's algorithm, which achieved between 13x and 220x speed up compared to the CPU sequential Dijkstra's algorithm [Ortega-Arranz et al., 2013]. Caggianese and et al. proposed an A* implementation for the GPUs, based on planning block (P-BA*) and suited for grid based maps [Caggianese and Erra, 2012]. First the search space is subdivided into small regular regions called tiles and then a parallel search is performed.

The purpose of this chapter is to present a parallel implementation for multi-agent pathfinding based on HNA*, in order to investigate the extent to which HNA* can offer a performance boost for large crowds. Since we mentioned before, as both the size of the environments and the number of autonomous agents increase, it becomes harder to obtain results in real time under the constraints of memory and computing resources. The goal of speeding up pathfinding is to be able to run multiple path searches simultaneously in order to handle large number of crowds simulation. To achieve this, we have proposed a parallel implementation for multi-agent pathfinding based on the improved versions of HNA* explained in chapter 3). As we discussed in chapter 3-section 3.6, our previous results showed that the parallelization on GPU was much faster than CPU. So in the next sections we decided to focus on the GPU version of the HNA* algorithm for multi-agent pathfinding. Finally we provide a thorough comparison of our results in terms of performance, and run stress tests to determine the number of paths that can be computed in parallel with our methods to handle multi-agent simulation.

4.4 Multi-Agent Parallel Pathfinding

As we mentioned before, the goal of speeding up pathfinding is to be able to run multiple path searches simultaneously in order to handle crowd simulation. In order to extend our system to handle large crowd simulation, it is necessary to run multiple path searches simultaneously. Having agents compute paths in parallel is an obvious way to speed-up pathfinding. Therefore one could simply use the basic A* algorithm, but have as many agents computing paths in parallel as the computer architecture allows. The interesting problem here is to determine whether the performance boost of our hierarchical pathfinding algorithms would also benefit a parallel multi-agent simulation.

Even with the hierarchical map representations, the problem at hand is highly parallelizable, since we simply need all agents to have access to the hierarchy information. This could be done by either storing it in shared memory or keeping local copies for each agent. The trade-offs to explore are the access to share memory by multiple entities, and the options for local memory based on size and access speed. Note also that the connecting paths for both S and G steps of HNA* are completely independent from each other, so we can still parallelize this step for all agents.

Considering the architecture of 1D CUDA grids and blocks, we have dedicated N blocks (where N is the number of agents) and four threads per each block. The purpose of those four threads is to handle the following steps of the HNA* algorithm:

- **Thread 1:** Get the connecting edges for the Start position S (step 1 of HNA*).
- **Thread 2:** Get the connecting edges for the Goal position G (step 1 of HNA*).
- **Thread 3:** Handle synchronization tasks.
- **Thread 4:** Computes the high-level path, extracts *intra-edges* and deletes S and G from the hierarchical navigation graph (HNG). Steps 2, 3 and 4 of HNA* respectively.

where each step of HNA* is (see chapter 3 for more information):

- **Step 1:** Connect S and G with HNG.
- **Step 2:** Find path at level L.
- **Step 3:** Extract subpaths (*intra-edges* from the high level path).
- **Step 4:** Remove temporal nodes.

The maximum number of agents computing paths in parallel is limited by the number of blocks that CUDA can run in parallel, which is 65,535. Inside each block, we have 4 threads running. Note that thread 4 cannot start until threads 1 and 2 have finished connecting S and G to the *inter-edges* in the corresponding high level node.

In order to perform step 3 of HNA*, we need to pre-compute and store the *intra-edges* for all high level nodes. This information together with the high level nodes, forms the hierarchical navigation graph (HNG) which needs to be available to all agents during simulation time, and thus access to it should be as efficient as possible. In order to decide the best solution to handle the storage of such information, we have tested both texture and local memory to evaluate which one allows faster access. Texture and shared memory are both on-chip, which makes access to them much faster than local and global memory. Texture memory shows a friendly cache behavior when we perform several reads that are spatially close to each other [Liu, Zou, and Luo, 2011]. Unlike traditional CPU caches that store sequential addresses, the GPU texture memory is optimized for 2D spatial locality. In our experimental results we observed that access to texture memory was 1.2x faster than local memory, therefore we decided to use texture memory for our implementation.

The details of our parallel multi-agent pathfinding method are shown in algorithm 6. The next issue to study is the performance benefits of implementing $LinkNodeToGraph(L, polyId)$ using hash tables (section 4.4.1) or computing the connecting paths in parallel (section 4.4.2).

Note that as we have mentioned in chapter 3, section 3.6, the parallel CPU implementation of HNA* did not offer great benefits when compared to GPU, since the CUDA implementation was much faster than the CPU paralellization. Therefore, we decided to only use the PCCP and CUDA versions (GPU) to develop the multi-agent pathfinding.

4.4.1 Parallel pathfinding with PCCP

As described in chapter 3, PCCP uses a hash table to store connecting paths from the center of each polygon to the set of *inter-edges* in the corresponding high level node. During the online phase, all agents are run in parallel. Each agent computes its own high-level path, querying for both *intra-edges* and connecting paths for S and G from texture memory. Therefore the $LinkGodeToGraph$ method in lines 10 and 15

Algorithm 6 Multi-Agent HNA*

```
1: procedure MultiAgentHNA*($S,G,L$)
2:     $AgentId \leftarrow blockIdx.x;$
3:     if $L = 0$ then
4:         $path \leftarrow FindPathA(S, G, 0)$
5:         return $(AgentId, path)$
6:     end if
   ▷ Step1. Connect S and G in parallel at level L:
7:     if ($threadIdx.x = 1$) then
8:         $S_{polyId} = AgentsData[AgentId][0]$
9:         $n_L^S \leftarrow getNode(S, L)$
10:        $LinkNodeToGraph(L, S_{polyId}, n_L^S)$
11:    end if
12:    if ($threadIdx.x = 2$) then
13:        $G_{polyId} = AgentsData[AgentId][1]$
14:        $n_L^G \leftarrow getNode(G, L)$
15:        $LinkNodeToGraph(L, G_{polyId}, n_L^G)$
16:    end if
17:    if ($threadIdx.x = 3$) then
18:        $syncthreadsCUDA()$
19:    end if
20:    if ($threadIdx.x = 4$) then
   ▷ Step2. Find path between S & G notes at level L:
21:        $tempPath \leftarrow findPathA(S, G, L)$
   ▷ Step3. Extract subpaths:
22:        for $highNode \in tempPath$ do
23:            $path \leftarrow path + getIntraEdges(highNode)$
24:        end for
   ▷ Step4. Remove temporal nodes:
25:        $deleteTempNode(n_L^S, n_L^G)$
26:    end if
27:    return $(AgentId, path)$
```

of algorithm 6, is performed with a query to texture memory where all connecting paths were previously stored.

4.4.2 Parallel pathfinding with CUDA HPA*

The previous algorithm still required that some portion of the allocated memory for HNA* algorithm was used to store the PCCP information. Since for very large scenarios, the size of allocated memory could become a bottleneck, and computing multiple paths simultaneously is highly parallelizable, we also propose a solution that does not require additional memory other than the hierarchical graph and *intra-edges*. This solution not only saves memory, but it also provides a more scalable solution.

For the parallel implementation of the step connecting S and G, two threads are dedicated to launch two child kernels (one to connect S and another one to connect G with the HNG)

.

In order to launch the child kernels to connect S in parallel, we consider m 1D blocks in the 1D CUDA grid where, m is the number of *inter-edges* of the polygon containing S. For each block B_i ($i \in [0, m]$) we compute A* from S to *inter-edge* ie_i (and similarly for G). All connecting paths to S and G are stored in shared memory as temporal edges of the HNG before computing the high-level path.

So the connect step for S and G will take as long as the longest of the A* searches to link S or G to an *inter-edge*. Note that this A* search is computed over a small section of the navigation graph. For example, for the case of connect S, this section corresponds to a set of connected polygons $\langle p_i, p_{i+1}, ...p\mu \rangle$, such that $\forall p_j \in n_x^s$, and $S \in n_x^s$. The number of polygons being limited by the user input value μ, and the hierarchy level Lx. The total number of connects S/G running in parallel is thus:

$$ParallelConnectsSG = \sum_{i=0}^{N} (m(i) + n(i)) \qquad (4.1)$$

The maximum value of N that can run in parallel is 65,535, and also the maximum number for each agent of $(n + m) \leq 65,535$ So, in this new method, the call *LinkNodeToGraph* (lines 10 and 15 of algorithm 6), to connect S and G to the *inter-edges* of n_L^S and n_L^G respectively, is performed by a kernel pathfinding search (A*), running all in parallel inside each block.

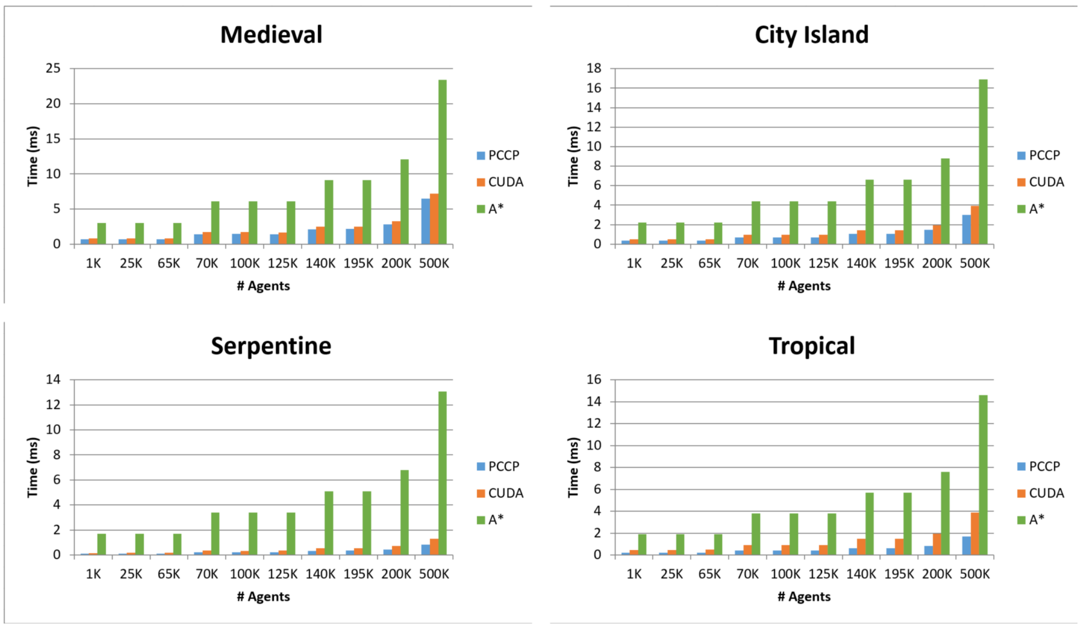


FIGURE 4.1: Time taken in *ms* to compute the corresponding number of agent's paths in parallel for A*, PCCP and CUDA-HNA. From 1K to 500K agents computing paths simultaneously under 6.5*ms* for PCCP and 7.2*ms* for CUDA.

4.5 Experimental Results

In this section we present the results achieved in terms of performance, but also discuss the limitations of each approach. All methods described in this chapter have been implemented using C++ and CUDA, with an Inter Core i7 Cpu @3.5 Gz, 1 MB L2 cache and 8MB L3 cache, 16 GB RAM. We have used an Nvidia GTX 420 with 2.4GB off-chip global memory and 2496 CUDA core. In order to implement and evaluate our methods, we have used several maps with different sizes (see Figure 3.11) for the purpose of running a fair comparison used the same maps as in [Pelechano and Fuentes, 2016b]). However, we have also tested with much larger scenarios, such as the Paris scenario which consists of *46,484* Vertices and *22,366* Polygons (Figure 4.2).

If we performed pathfinding for multi-agent systems in a sequential manner, we would have strong limitations on how many agents we could run in real time. However, the exact number of agents depends strongly on the map and the hierarchy configuration (especially for the old HNA*). For example, if we consider that we run sequential pathfinding based on A* and HNA* (for the original, CUDA and PCCP approaches), we would obtain that the maximum number of agents that could be run on average are those shown in table 4.1. Therefore, if we want to compute

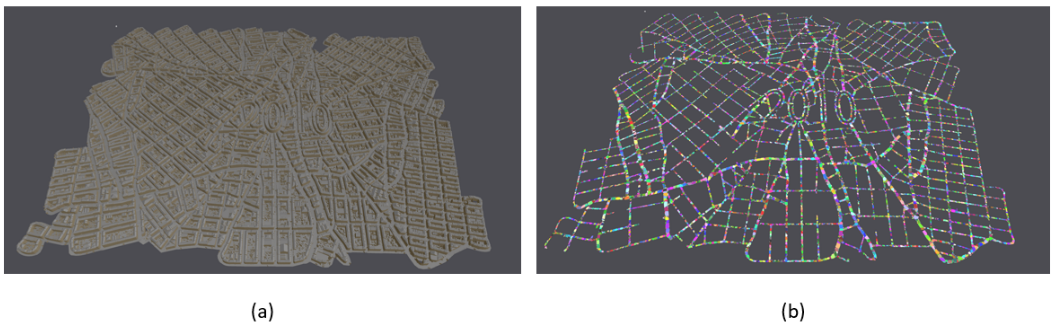


FIGURE 4.2: (a) Paris scenario and (b) Hierarchical representation at L_1

pathfinding for a large number of agents, it is necessary to apply parallelism also at the level of each agent's computation.

We have evaluated the performance of parallel multi-agent pathfinding, using the two methods described in this chapter (PCCP and CUDA-HNA*), and A*. All three methods use the same CUDA implementation to compute all agents' paths in parallel for a multi-agent system. This will allow us to study, whether the gain that we can achieve with a hierarchical path finder for a single agent, also holds when using multi-agent parallel pathfinding. For this comparison, we have used again the 4 scenarios shown in Figure 3.11 and compared three algorithms:(1) A*, (2) PCCP, and (3) CUDA-HNA*.

As shown in Figure 4.1, performance times increase in all four methods with the number of agents. Performance of the multi-agent parallel PCCP method is the fastest, followed by the CUDA-HNA* version, which also outperforms A*. As we can see, the parallel implementation in CUDA can handle real time pathfinding for over 500K agents even when using the basic A* algorithm, but with an important speed-up achieved by using HNA* with the connection step in parallel.

We have chosen the number of agents for our simulation to show that the jumps in the computational time are due to the number of blocks available to run 65,535 agents in parallel. With our CUDA version, we had 65,535 number of blocks to launch our parallel pathfinding, which is consistent with the computational jumps appearing for each multiple of 65,535 agents. The negative impact of the number of blocks is much more noticeable for A*, than for our implementations with PCCP or

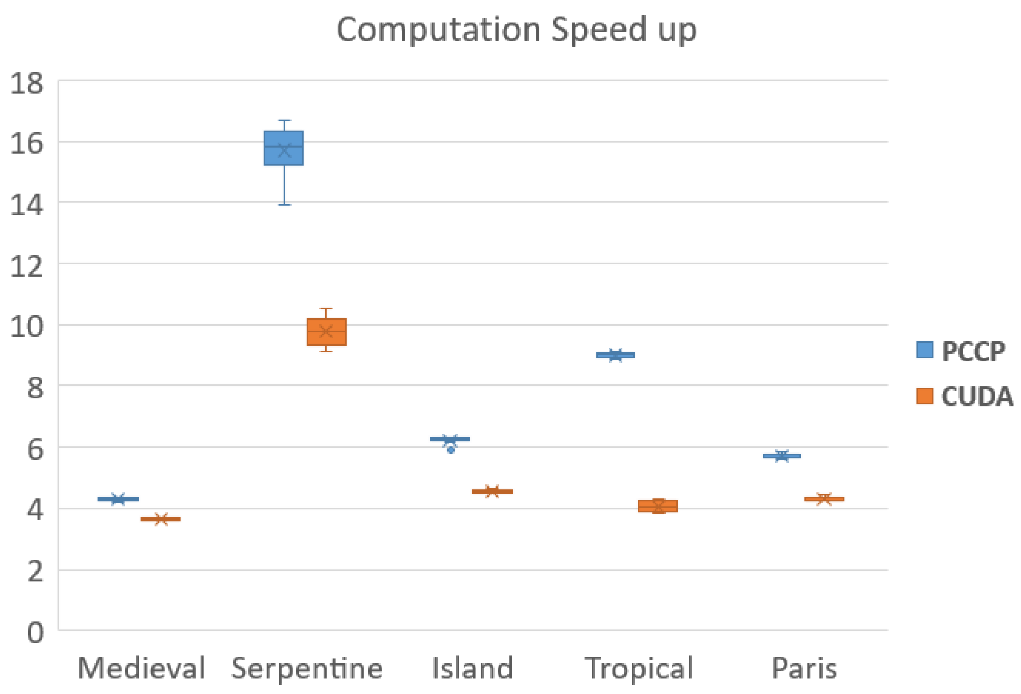


FIGURE 4.3: Speed-up achieved for each of the scenarios, with PCCP and CUDA-HNA* over A*.

TABLE 4.1: Maximum number of agents that can run in real time (25FPS) in sequential multi-agent pathfinding for each algorithm.

Map Name & hierarchy	A*	HNA*		
configuration		Original	CUDA	PCCP
City Island ($L1\mu20$)	18	11	48	63
Tropical Island ($L2\mu10$)	24	5	62	140

CUDA, making our HNA* more scalable.

Our main interest with this thorough evaluation was to determine whether HNA* offered important speedups for multi-agent parallel implementation. As we can see in Figure 4.3, for the 4 scenarios tested, we can observe speedups on average between 4.3x and 15.7x for PCCP, and between 3.6x and 9.8x for CUDA-HNA*. Therefore, the benefits of our hierarchical representations still hold even when a parallel implementation could be carried out for both HNA* and A*.

4.6 Conclusion

In this chapter we have studied the problems of pathfinding for large number of agents in large scenarios for hierarchical representations based on navigation meshes. we have carried out a thorough performance comparison of a parallel multi-agent implementation of A, PCCP and CUDA-HNA in order to determine the potential of using hierarchical pathfinding.

For the parallel implementation of the step connecting S and G in CUDA-HNA*, two threads are dedicated to launch two child kernels (one to connect S and another one to connect G). As we have shown in our results, the speed-ups achieved by both our methods outperform the parallel A* solution. As we can see in figure 4.3, for the 4 scenarios tested, we can observe speedups on average between 4.3x and 15.7x for PCCP, and between 3.6x and 9.8x for CUDA-HNA*. For this comparison all three methods can compute pathfinding for multiple agents in parallel. As we have shown in our results, the speed-ups achieved by both our methods outperform the parallel A solution. Therefore hierarchical implementations can allow us to run a potentially much larger number of agents simultaneously.

Chapter 5

Towards Human-like Agent Path Planning

5.1 Abstract

Pathfinding for autonomous agents has been traditionally driven by finding optimal paths. Where typically optimality means the shortest path length between two points in a virtual environment. The most famous algorithm is the A* search, which efficiently explores a graph by balancing the cost of the current path with a heuristic that estimates the cost to the destination. There are many variants of A*, which do not guarantee an optimal solution, because their goal is to either reduce memory requirements or to improve computational performance. However, when it comes to simulating virtual humanoids, none of these solutions considers aspects of human memory or orientation. In this chapter, we propose a new algorithm for pathfinding that is inspired by neuroscience research on how the brain learns and builds cognitive maps. Our method represents the space as a hexagonal grid, based on the GPS of the brain theory, and fires memory cells as counters. Our path finder then combines a method for exploring unknown environments while building such a cognitive map, with an A* search using a modified heuristic that takes into account the GPS of the brain cognitive map.

5.2 Introduction

Path planning for autonomous agents and robots have been widely applied for many decades. There is a large range of applications, from robotics to agent and multi-agent simulation. The problem in robotics is typically to steer a robot through an optimal path between two points, avoiding obstacles and satisfying other constraints.

In autonomous agents and multi-agent simulation, the emphasis is usually to find an optimal path between two points, or sub-optimal if time performance is critical. For a good state of the art in path planning we refer the reader to [LaValle, 2006] [Kallmann and Kapadia, 2016].

Path planning for video games plays a primary role in complex and large environments. In general, path planning deals with finding a sequence of state transition actions that transform a start position to a goal position, where each passing action has an associated cost, and the sum of costs of all passing actions describes some measurements for the path. Creating transition actions for path planning generally involves the following conditions: (1) travel time between the start position and the goal position (also referred to the time factor); (2) energy used of an agent traveling a path; (3) Agents do not conflict with other objects and agents; and (4) smoothness of a path is aimed to ease steering of agents. Currently, most path planning methods aim at planning an optimization model that considers one or more of the above-mentioned features and then conducting a minimization procedure to achieve an optimal path. For instance, the shortest path, minimum time-consuming path, minimum energy cost, and coverage path planning are, individually, studied in the literature [Mei et al., 2004] [Goldberg and Harrelson, 2005] [Galceran and Carreras, 2013] for a given navigation task. Path planning includes four developing steps: (1) graph-based methods (e.g. Dijkstra search, Voronoi diagram [Dolgov et al., 2008], A* and its variants[Hart, Nilsson, and Raphael, 1968] [Dechter and Pearl, 1985] [Ferguson and Stentz, 2007], artificial potential field methods (APF) [Khansari-Zadeh and Khatib, 2017][Koren and Borenstein, 1991] probabilistic methods (e.g. probabilistic roadmap method (PRM) [Kavraki et al., 1996] rapidly exploring random tree (RRT) [LaValle, 1998] [Jaillet, Cortés, and Siméon, 2008], and machine learning-based methods[Willms and Yang, 2006] [Otte, 2008].

While existing techniques give possible solutions for practical applications, neither of them take into account human factors to closely simulate how humans behave in the real world. There are many aspects of human behavior that affect route choice during navigation, such as: memory, mental maps, or visibility. The method that we present in this chapter is inspired by research from neuroscience, regarding building mental maps following the human brain navigation research. When humans perform pathfinding, they appear to be influenced by many factors, but there are two that are highly valuable, which are the estimated distance, and the familiarity with the path. The first one, is typically incorporated in pathfinding, as a heuristic which assumes people can guess the shortest distance between several nearby positions. The second one is mostly ignored, and thus either agents are simulated as super-humans that know the entire environment, or else the nodes of the graph are discretized in a binary way as known/unknown (thus the search is performed only over the set of known nodes which represents a subgraph of the environment).

When a human is looking for a path within a large environment, such as a city, there can be two opposite scenarios: (i) The person know very well the city, or (ii) the person has never been in the city before. Of course there can be many situations in between, such as the person knows very well a part of the city but has no information about other parts. We will first focus on providing algorithms for cases (i) and (ii), and then explaing how both algorithms can be combined to fit any situation in between those two cases. In the first situation, the person has a mental map in his memory and he will follow the path based on the previous experience. In our work, we will build this mental map based on the human GPS of the brain theory [Hafting et al., 2005]. But in the second state, when the person does not know the environment, he can either try to find the given goal position randomly or following some vague knowledge (for example to simulate how humans move around an unknown city after looking at a map or asking for directions). In this second case we are interested in the case of searching an unknown environment but with some vague idea of where the goals could be roughly located, because a map completely unknown with lack of information would require an exhaustive search such as Breath First Search which is rarely used by humans (we expect humans to ask for guidance or else have a quick glance at a map, and thus have some rough knowledge of the environment).

In this chapter we propose a novel pathfinding method for intelligent agents that better simulate humans by implementing methods based on the human brain research. We first consider how humans learn about the environment and memorize spatial information following the *GPS of the brain* [Hafting et al., 2005]. Then we propose pathfinding methods which explore unknown graphs in a way that is closer to how humans would wander an unknown environment. We finally combine known and unknown areas to propose a new pathfinding model that better resembles what we would expect humans to do.

This chapter is structured as follows: in the next two section, we explain the theory of the GPS of the brain, followed by related work in pathfinding. The next section explain our approach in detail, and finally we present results and conclusions. We believe that our approach opens future research to have more human like behavior in video games and virtual reality applications in general that will enhance the realism of populated scenarios.

5.3 Human brain navigation

In the mid-20th century, Edward Tolman observed rats moving around in labyrinths and proposed that the brain might contain a cognitive map, which allows animals to learn to navigate and find their way [Tolman and Honzik, 1930]. In 1971, John O'Keefe [O'Keefe and Dostrovsky, 1971] discovered the first key to the inner GPS in the mammal's brain which is called place cells. He recorded nerve activity in the hippocampus region of the brain in unobstructedly moving rats. He obtained single cells that just activated when the rat was in a certain location in the environment. These places cells were active for different locations, generating an inner map in the hippocampus region of the animal brain showing the animal where it is in the environment. The hippocampus can create various maps, represented by the collective activity of the place cells that are activated in the various environments. Therefore the memory of the environment can be stored as a particular combination of place cell activities in the hippocampus. Figure 5.1 shows the places cells.

Inn 2005 May Britt and Edvard Moser [Hafting et al., 2005], observed cells in the Entorhinal cortex of rat brain, which is a region close and very well connected to the hippocampus. Here, they obtained nerve-cells that were not active in just one location but fired when the rats passed multiple locations.

Each of these cells was fired in a single spatial pattern and collectively these grid cells create a coordinate system, with an hexagonal grid shape, that allows for spatial navigation. This coordinate system separates the environment into latitudes and

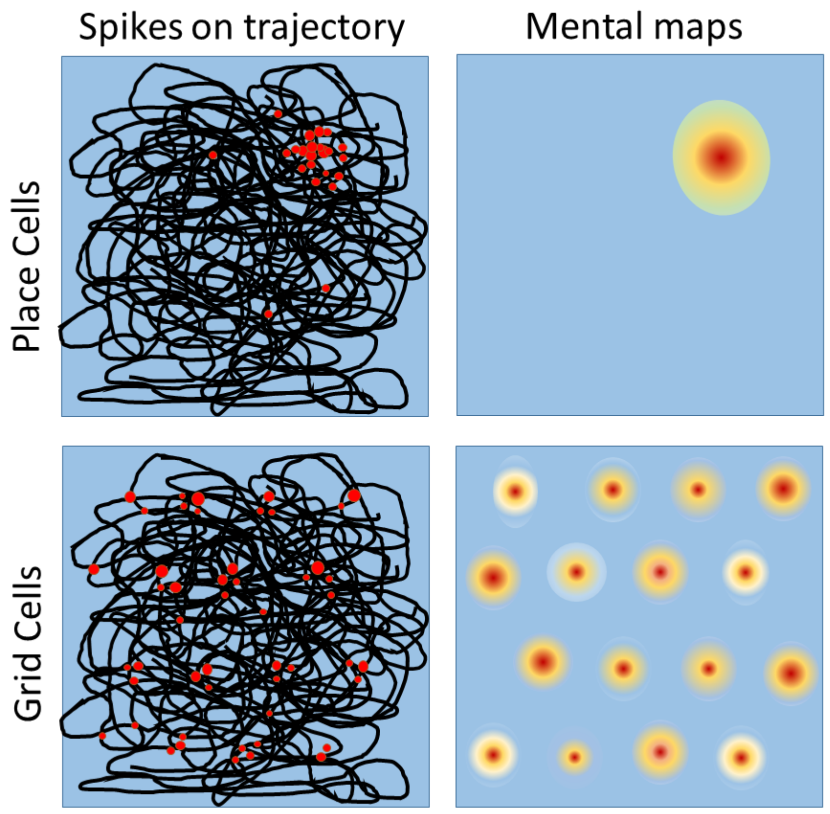


FIGURE 5.1: A schematic example of place cell and grid cell firing. The first column shows in black the path taken by a rat as it traverses the square. Electrodes implanted within the hippocampus and entorhinal cortex record from individual neurons. Place and grid cells show increased firing (red dots) at discrete locations in the environment. Individual place cells (top) fire only in one location, whereas grid cells (bottom) have multiple firing fields forming a hexagonal shape. The hexagonal symmetry of the spacing between these latter fields gives rise to the term "grid cells". The firing frequency of place and grid cells within environment (mental map) is shown in the second column, with yellow and red depicting higher rates of firing on a background of no cell activity (blue) [O'Keefe and Dostrovsky, 1971]

longitudes that keeps track of how far rat is from a turning and/or starting position. The brain GPS emerges from the combined work of place and grid cells. Both place and grid cells operate together to provide the rat's GPS. Place cells appear to be inspired by visual information, the position of boundaries such as corners and walls in the environment seems very important to their function and on the other hand, grid cells track the animal's motion. Even though this studies have been mostly performed with rats, this nerve cell system has been observed also in rodents, bats, monkey and humans and neuroscience researchers now think that it is most like present in all mammals. Therefore, from a human simulation perspective, it is key to consider such hexagonal formation, with cells being fired based on movement as a more plausible model to simulate human spatial memory and pathfinding.

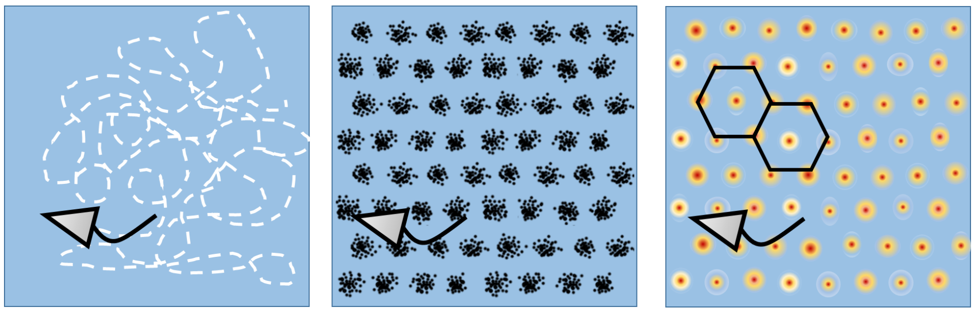


FIGURE 5.2: A schematic drawing of grid cell firing as the rat moves through a square [Hafting et al., 2005]. The hexagonal pattern gives high spatial resolution that allows the animal to recognize its locations and orientation.

5.4 Related work

In this section, we provide an overview or previous work on typical pathfinding methods focus mostly on real-time search within 2-D scenarios. Pathfinding has been widely studied in both real and virtual environments like video games and robotics.

Various methods to resolve the navigation problem exists. Applicability of various methods depends on the properties of the agent's environment (known or unknown environment, availability of global position systems, etc.) and on-board sensors. Modern navigation approaches can be categorized into two categories: Reactive and Deliberative.

The Reactive techniques operate by manipulating "at-the-moment" sensors' information and perform the movement based on the current situation of the system and the surrounding environment. This method is mainly applied for navigation in a dynamically changing environment, e.g. for obstacle-avoiding problems, path following, etc. This method is also beneficial when the time limit is short and decisions for acts should be made very rapidly.

The Deliberative approaches imply that the agent has some knowledge about his environment (e.g. it has a map or memory) and operates the navigation taking into account this information. This category is described by path planning algorithms, simultaneous localization and mapping (SLAM) algorithms etc.

Currently, there are many methods for locating and mapping (SLAM) that allow a single agent or a group of agents to gather knowledge from their environment and generate the map by employing different sensors like laser telemetry sensor or LIDAR [Jiménez et al., 2018] [López et al., 2017][Alismail, Baker, and Browning, 2014] [Li et al., 2016].

Our proposed method is classified in Deliberative approaches since the agent may or may not have knowledge about the environment in its mental map (or memory) and performs navigation taking into account this partial information while extending such mental map. In a way, it could also be considered a SLAM technique Milford, Wyeth, and Prasser, 2004.

Some researchers have focused on graphs being built from visibility information [Wolfe, Fitzgerald, and Gracer, 1981] [Toth, O'Rourke, and Goodman, 2017]. In these cases, obstacles are considered as polygons in the configuration space and a graph is created based on the start and the goal position and vertices within the environment. Finally, the path from start and goal position will be obtained by graph search approaches like Dijkstra's algorithm. A large number of robots has been built that explicitly simulate biological navigation behaviors for obstacle avoidance, such as the ones simulating the migration of seabirds [Otte, 2008] [Franz and Mallot, 2000] and ant colony behavior navigation model [Milford and Schulz, 2014]. Inspired by the social interplays in human crowds or animal swarms, Savkin and Wang [Savkin and Wang, 2014] have proposed an efficient obstacle avoidance method in dynamic environments by combining representation of the information about the environment. Typically, the development of robotics (real of virtual robots) has been directly or indirectly affected by human's experiences and behaviors [Chen and Sun, 2012] [Zhang and Wang, 2004]. The work by Rodrigues et al. proposed an agent steering model based on the biologically-motivated space colonization algorithm [Rodrigues et al., 2009].

Artificial potential field (APF) [Khansari-Zadeh and Khatib, 2017] is a very popular pathfinding algorithm specially in the field of robotics. In this method, the field area is generated by considering repulsive and attractive spaces. The obstacles will be considered as repulsive areas, and the goal position is considered as an attractive area in the artificial potential field. The repulsive areas avoid agents from moving close to the obstacles and the attractive areas move agents towards the goal location. The APF provides smooth paths, but the main disadvantage of APF is that it suffers from local minima problems.

Sampling Based Planning (SBP) methods are the most important improvement in path planning [Karaman and Frazzoli, 2011]. Main benefits of SBP methods are that they have very low computational cost and also they have applicability to high dimensional problems with better success rate for complex queries [Elbanhawi and Simic, 2014]. SBPs are probabilistically complete, and the paths created by these algorithms for the same problem are not unique. Probabilistic pathfinding algorithms are very effective approaches for path planning. The search strategy of the probabilistic pathfinding algorithm is to choose collision-free points randomly in free movement space and connected them to arrange a path. Probabilistic roadmaps (PRM) [Kavraki et al., 1996], Rapidly-exploring random trees (RRT) [LaValle, 1998] and Rapidly-exploring Random Tree Star (RRT*) [LaValle, 1998] are the most representative methods of Probabilistic pathfinding algorithm. PRM based techniques are mostly applied in a highly structured static environment such as factory floors [Noreen, Khan, and Habib, 2016]. As long as enough time is provided, RRT* can converge to an optimal solution. RRT* has achieved a large success rate in finding the solution for high dimensional complex problems with various successful applications.

The A* search algorithm [Hart, Nilsson, and Raphael, 1968] is the most popular algorithm for pathfinding, since it has many beneficial properties. First, the provided path by A* search algorithm will be an optimal path between the given start and end positions in a scenario. Secondly, the A* search has the ability to return a result in a finite time even in the case that there is no solution for the problem. Thirdly, a suitable admissible function can lead to an acceptable time-consuming even for a big scenario. Today, there are many variants of A* search algorithm to deal with different problems and tasks, such as D* Lite [Koenig and Likhachev, 2002a] any-angle

A* [Yap et al., 2011], [Phi*Nash, Koenig, and Likhachev, 2009] and Field D* [Ferguson and Stentz, 2007], and hierarchical approaches [Rahmani and Pelechano, 2017], [Pelechano and Fuentes, 2016b], [Rahmani and Pelechano, 2020].

Kapadia et al. [Kapadia et al., 2013a] proposed a path planning framework that explores the satisfaction of multiple spatial constraints like walking beside walls, staying behind a building between obstacles and agents at the global navigation layer. Their method introduced a hybrid environment representation to balance computational performance and discretization resolution, where they first applied a triangular navigation mesh and then obstacle annotations, by adding additional nodes to the previous triangular mesh to provide static spatial constraints. Anytime Dynamic A* has used as an underlying path planner that combines the incremental planning properties of D* Lite and the anytime planning properties of ARA* in order to efficiently repair solutions after world changes and agent movement. By modifying the cost of the nodes, Their method allowed to obtain paths that would not aim for the shortest path, but instead choose one that would try to pass through save areas while avoiding others.

Multi-agent Navigation Graph (MaNG) [Sud et al., 2008], was based on first and second-order Voronoi diagrams. The MaNG is used to do path planning and vicinity computations for each agent in real-time so that it computes a graph that provides maximal clearance to the obstacles and remaining agents. In this method, the set of Voronoi sites P is divided into two subsets. The set of obstacles Po and the set of agents Pa. The MaNG, defined *MG(P)*, is the union of all the vertices and edges from the first-order Voronoi graph $VG^1(P)$ and a subset of the vertices and edges from the second-order Voronoi graph $VG^2(P)$ contained inside the first order Voronoi domain of each agent. In this method, graphics hardware (GPU) is used to compute the MaNG.

Gradient-based methods [Schulman et al., 2014] are two-step algorithms for providing an optimal path. Usually, these algorithms use a direct line to connect the start and goal points, even if such line goes through obstacle zones, and then eliminate the obstacle points in the gradient directions. Since this method creates non-smooth paths, a post-smoothing process is required.

Considering current pathfinding methods, A* search algorithm offers the possibility of being deterministic and highly adaptable, mostly by altering the heuristic function. A* will find an optimal solution, however, humans are not always likely of finding an optimal solution, specially when not all the environment is fully known. Therefore, to carry out pathfinding in partly known environments, we will use A* with a new heuristic function that considers human-like cognitive maps and explores the environment based on the reliability of the acquired knowledge.

5.5 Human-like pathfinding model

Our goal is to create a pathfinding model to simulate human behavior more closely than previous work in the literature. Most previous models focus on finding optimal paths, smooth paths, and/or finding solutions within certain time constraints. Our model is focused on building the first path finder method able to closely imitate the human brain memory which is iteratively build based on previously visited places. The proposed method consists two phases: (1) Generating the cognitive map and (2) Pathfinding based on the current map information combined with typical human-like pathfinding behavior. Most navigation maps in the literature used for pathfinding, consists in either a regular grid of squared cells, or a polygonal mesh. Our work proposes a new navigation mesh consisting of regular hexagons. The reason behind this decision is that, as we have explained in section 5.3, the grid cells that are fired in our brain when we move through an environment, follow such spatial structure. This hexagon grid help us recognize location and direction of previous visited positions in the environment. So if we want to have a better model of the human brain to build mental maps, it is best to follow the actual spacial structure contained in our brain.

In our proposed method, we have created a hexagonal mesh of the walkable region of a given map (obstacles are not included since the agent will never walk through them and thus brain cells should never get activated for those locations). This hexagonal mesh will operate as grid cells of our virtual human's cognitive map.

5.5.1 Hexagonal cognitive maps generation

In order to build the agent's mental map, they need to wander the environment so that grid cells are fired based on the movement of the agent.

Lets consider persons $\mathcal{P}$ trying to reach room G from room S in a building consisting of many offices (rooms). There are two possible situations that will impact the person's decision: *(i)* either the person knows where the room G is located (has been there before) or else *(ii)* the person does not know the environment because he has never been there before. The first situation, implies that the person has moved through the building in the past, and has thus its own hexagonal mental map. In this case, he can simply perform an optimal search based on A*. In the second situation, when the person is unfamiliar with the environment and does not have a mental map yet, he needs to search with a naive approach while building such map. Human cognitive maps are created by firing the neurons of hippocampus region of the brain when the person visits a location in the environment. Similarly our agents will fire the cells corresponding to the location that the agent walks by. This cell firing is implemented with a counter that increases as the agent walks repetitively through a location. Therefore the value of the cell counter is an indicator of how much the agent knows that corresponding location in the virtual space.

Our pathfinding algorithm inspired by the GPS of the brain, will then use the value of the counters to introduce a new heuristic function for the A* search. By doing so, our A* search gives higher priority to choosing paths that move through cells with higher counter value. Figure 5.3 shows the counter values of each hexagon when the agent searches the environment.

5.5.2 Path Planner

We will first explain how our agents explore an unknown environment in order to build their hexagonal mental maps, and then how those mental maps are used to carry out an A* search with a new heuristic driven by the cell counters.

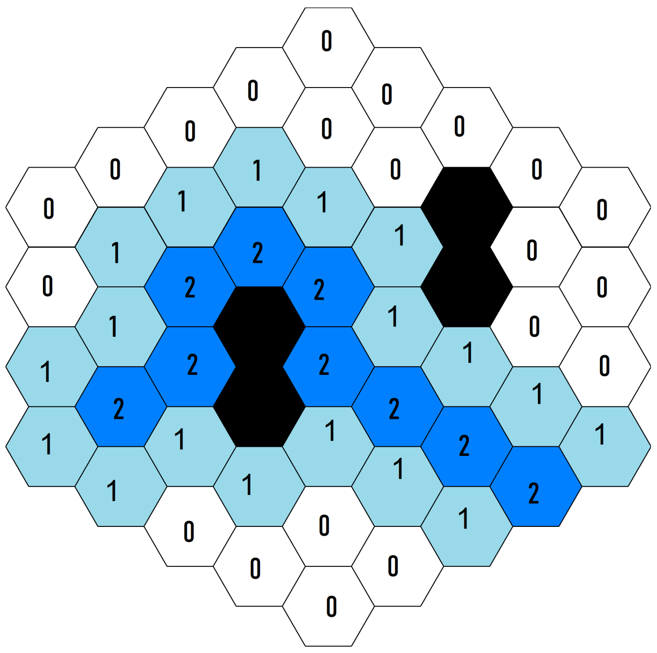


FIGURE 5.3: Hexagon grid cell with corresponding counters.

Unknown environment

When a human is located in an unknown environment for the first time, he would have no mental map of the environment in his brain. In order to search for a path between S and G, he would have to fully explore the environment. There are well known algorithms to perform a full exploration of an unknown environment such as Depth First Search (DFS) or Breadth First Search (BFS). However, humans do not typically perform such a blind exhaustive search. Whereas we are inside a large building or outdoors in a city, it would be reasonable to consider that humans would either have a glance at a map, or else ask for basic directions. So, in order to simulate human pathfinding behavior in unknown environments, we have developed an algorithm that performs a naive search with some basic knowledge of directions. Our method also assumes that humans prefer to walk along the longest sight-line towards the goal direction, which is a behavior that has been reported on real humans.

Therefore, our agents' naive navigation algorithm is based on two principles: Firstly, humans are likely to perform exploration in a sequential manner moving towards a goal *G*, and thus our search is based on DFS with a greedy heuristic based on rough knowledge of the goal direction, and secondly humans are likely to walk along the line of sight, and only reconsider direction if they feel that they are not moving towards the goal.

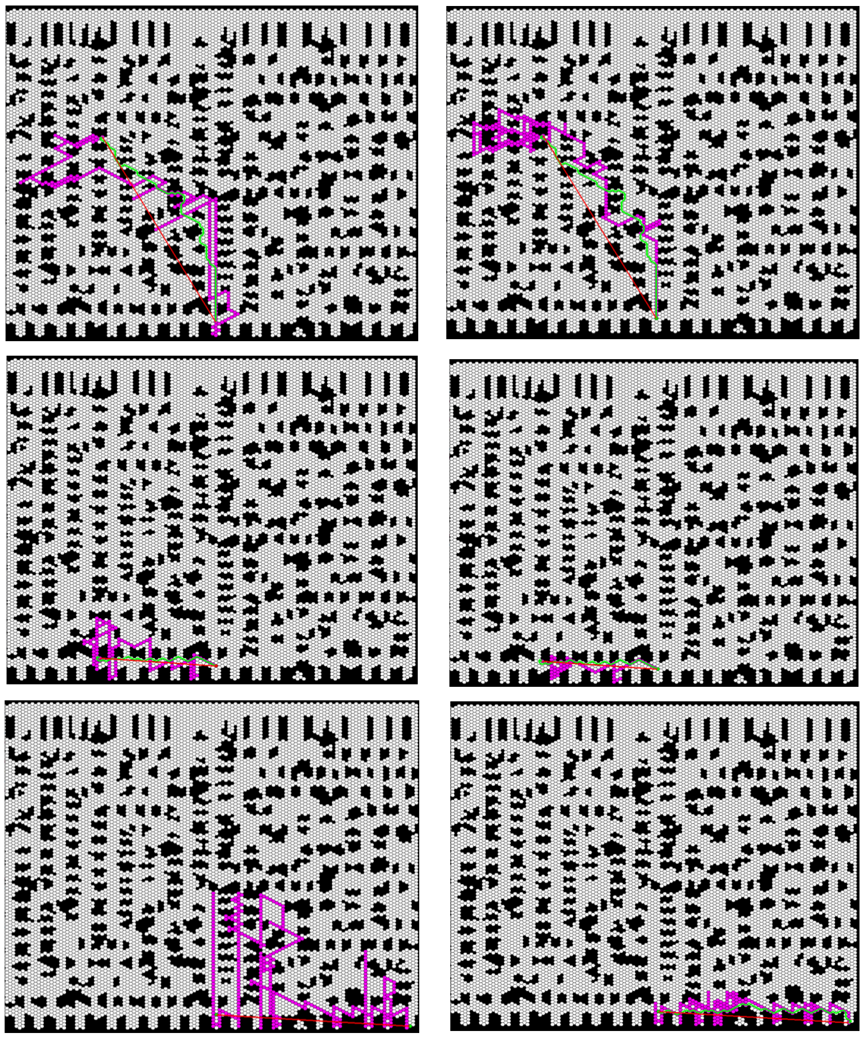


FIGURE 5.4: Affect of $\mathcal{C}$ on path length. Green line show A* path and Pink line shows obtained path base on our algorithm.

When the agent is located in an unknown environment, the counters for all cells is initially set to zero (no cognitive map). The agent will then move toward the goal position with our naive approach. The first step is thus to compute the forward direction for the agent. As shown in figure 5.3, each hexagon in the grid cell has at most 6 neighbors. In order to calculate the movement direction, we calculate the vectors $\vec{u}_i, i \in [1, 6]$ which point from the current cell towards each neighbor, and keep those that are approximately in the desired direction of movement as the set $\mathcal{D}$. Note that vectors pointing towards an obstacle cell will be discarded since they do not represent a valid agent movement. The set of possible direction of movement is:

$\mathcal{D} = \bigcup \vec{u}_i$, such that $cos(\angle(\vec{u}_i, \vec{u}_G)) > \delta$

$\vec{u}_G$ is the unit vector from the current cell pointing towards the goal G. δ is a user defined value in the range $[-1, 0.8]$. The next direction $\vec{v}$ chosen for graph exploration is randomly picked from the set of directions $\mathcal{D}$. Note that when $\delta = -1$ we would have a completely uninformed search similar to DFS. They key in our method is that δ represents the level of confidence regarding the goal direction. The larger the δ the more directly the agent will explore the straight line towards the goal direction. The maximum possible value of δ is 0.8 which corresponds to an angle of 30^o with respect to the goal direction $\vec{v}_G$ to guarantee that there will be at least one possible direction in our hexagonal grid. Having a large value of $delta$ will give fewer possible directions pointing towards the goal direction. If $\mathcal{D} = \emptyset$ then the next direction $\vec{v}$ is chosen randomly amount the possible directions of movement (towards obstacle-free cells).

The agent will follow direction $\vec{v}$ until it either hits an obstacle or its trajectory appears to be moving away from its desired direction. This second case is done by triggering a new computation of preferred direction every $\mathcal{C}$ cells. Figure 5.4 shows the affect of $\mathcal{C}$ on path length. We have empirically found that $\mathcal{C} = 5$ provides good perceptual results for our naive search exploration. However, this parameter could be defined by the user, or could be set to a range of values to provide more heterogeneous graph exploration in the case of crowd pathfinding. Figure 5.5 shows an example path of our naive path exploration algorithm (in pink) in an unknown area of the environment, against the A* solution in green. Note that our naive exploration algorithm has the agent walking along the cells as whilst exploring, and thus there can be some small loops in the trajectory, which can resemble when a human needs to walk back to a street junction after realizing he may not be in the right path towards the destination. As the agent walks by the environment, the cells counters will be increased starting to build the mental map of the agent.

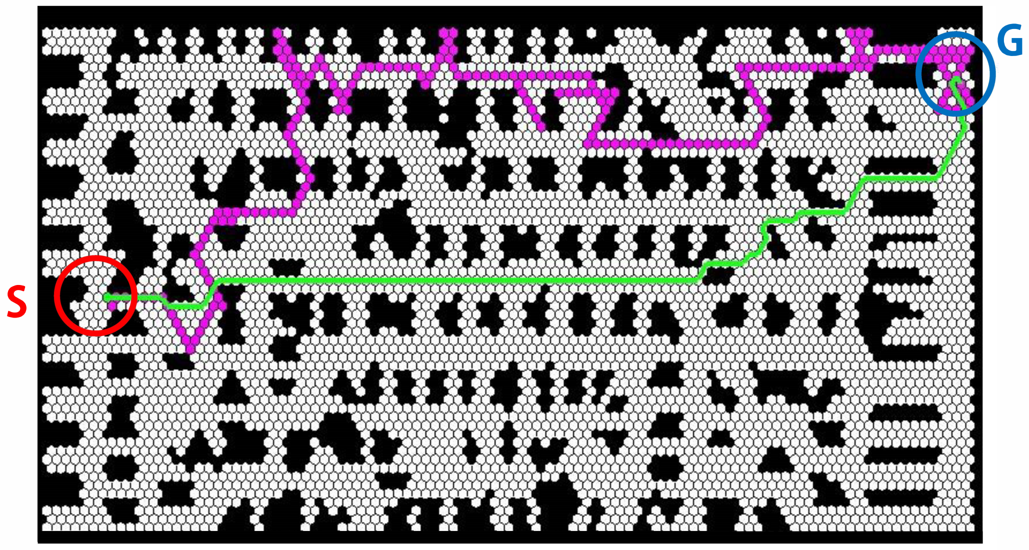


FIGURE 5.5: Comparison of the path obtained with the naive exploration algorithm (in pink), and the optimal solution obtained with A*.

Known environment

Pathfinding in a known environment can be done using the agent's cognitive map that has been built by performing many searches following our naive exploration algorithm. Every time an agent visits a hexagonal cell, the counter value of the corresponding cell is increased. By increasing the counter values of the cells, the agent's knowledge about the environment is also increased. In order to find a path in a familiar environment, we use the A* search algorithm, but with a modified heuristic function that takes into account the agent's knowledge.

Given a goal position G and the starting location of the agent S, we need to find the path that can get the agent from S to G avoiding the static obstacles in the environment. Note that obstacle cells will not appear in the mental map, as the agent will not have walked through them previously. The agent needs to find a path $\pi = \langle S, p_1, p_2, ..., G \rangle$ by running the A* search algorithm with a modified heuristic.

The main key for the A* search method is to define an admissible heuristic function $h(p_i)$, so that it avoids overestimating the actual cost to arrive at the goal location. In this thesis, we define the heuristic function $h(p_i)$ from a point p_i to a goal position G as follows:

$$h\left(p_i\right) = \|p_i - G\| + \lambda_i \tag{5.1}$$

where $\|p_i - G\|$ is the 2D Euclidean distance from the current position to the goal G, and the term λ_i is defined as:

$$\lambda_i = \begin{cases} 2 \times \mathcal{C}_{max} & if\ c_i = 0 \\ \frac{\mathcal{C}_{max}}{c_i} & if\ c_i > 0 \end{cases} \tag{5.2}$$

$\mathcal{C}_{max}$ is a user defined value, which sets an upper limit to the level of knowledge about a cell. Therefore, the larger heuristic would be assigned to those cells with counter, c_i, equal to 0 (unknown cells). For known cells, the heuristic value becomes smaller as the counter increases, and thus known cells have higher priority in the A* algorithm to be selected for exploration. When all cells have the highest counter value, our algorithm is equivalent to a basic A* search.

With our proposed heuristic function, agents will find paths towards a goal position based on their previous knowledge of visited places (i.e. their cognitive map). This heuristic makes agents more likely to move within familiar environments, and only when knowing the entire environment would they be able to find an optimal path. Unknown cells are thus avoided.

5.5.3 Combining known and unknown areas

Even though we have presented so far two different algorithms for path exploration, based on whether the environment is known or not, it is of course possible to encounter scenarios with partly known areas. In such case, both algorithms can be combined, so that, if the agents has knowledge about the area covering the space between the start and goal positions, then algorithm A* with modified heuristic is applied, but when the agent is in an unknown cell, the the naive search algorithm is executed to further explore the environment. By alternating between both algorithms, the agent will gradually increase its internal cognitive map based on the GPS of the brain. The details of our path planning method are shown in Algorithm 7.

Algorithm 7 GPS of the Brain

```
procedure GPS_PathFinder($x$,$G$)
    $c \leftarrow get\_counter(x)$
    if ($c = 0$) then  // Naive Exploration
        $Step \leftarrow 0$
        $n \leftarrow S.NumNeighbors()$
        for $i \leftarrow 0$ to $n$ do
            if ($cos(\vec{u}_i, \vec{u}_G) > \delta$) then
                $\mathcal{D}.push(\vec{u}_i)$
            endIf
        endFor
        if ($\mathcal{D}.size > 0$) then
            $\vec{v} \leftarrow PickRandDir(\mathcal{D})$
        else
            $\vec{v} \leftarrow PickAnyDir(x)$
        endIF
        $x \leftarrow NextCellInDir(\vec{v})$
        $c \leftarrow get\_counter(x)$
        while ($c = 0$ AND $Step < 5$) do
            $Add\_Cell\_to\_Path(x)$
            $x \leftarrow NextCellInDir(\vec{v})$
            if $collision(x, \vec{v})$ then
                $GPS\_PathFinder(x, G)$
            else
                $Step = Step + 1$
                $x \leftarrow NextCellInDir(\vec{v})$
                $c \leftarrow get\_counter(x)$
            endIf
        endWhile
        if ($c = 0$ AND $Step = 5$) then
            $Add\_Cell\_to\_Path(x)$
            $GPS\_PathFinder(x, G)$
        endIf
    else //A* with counter based heuristic
        $Add\_Cell\_to\_Path(x)$
        $GPS\_PathFinder(x, G)$
    endIf
    end procedure
```

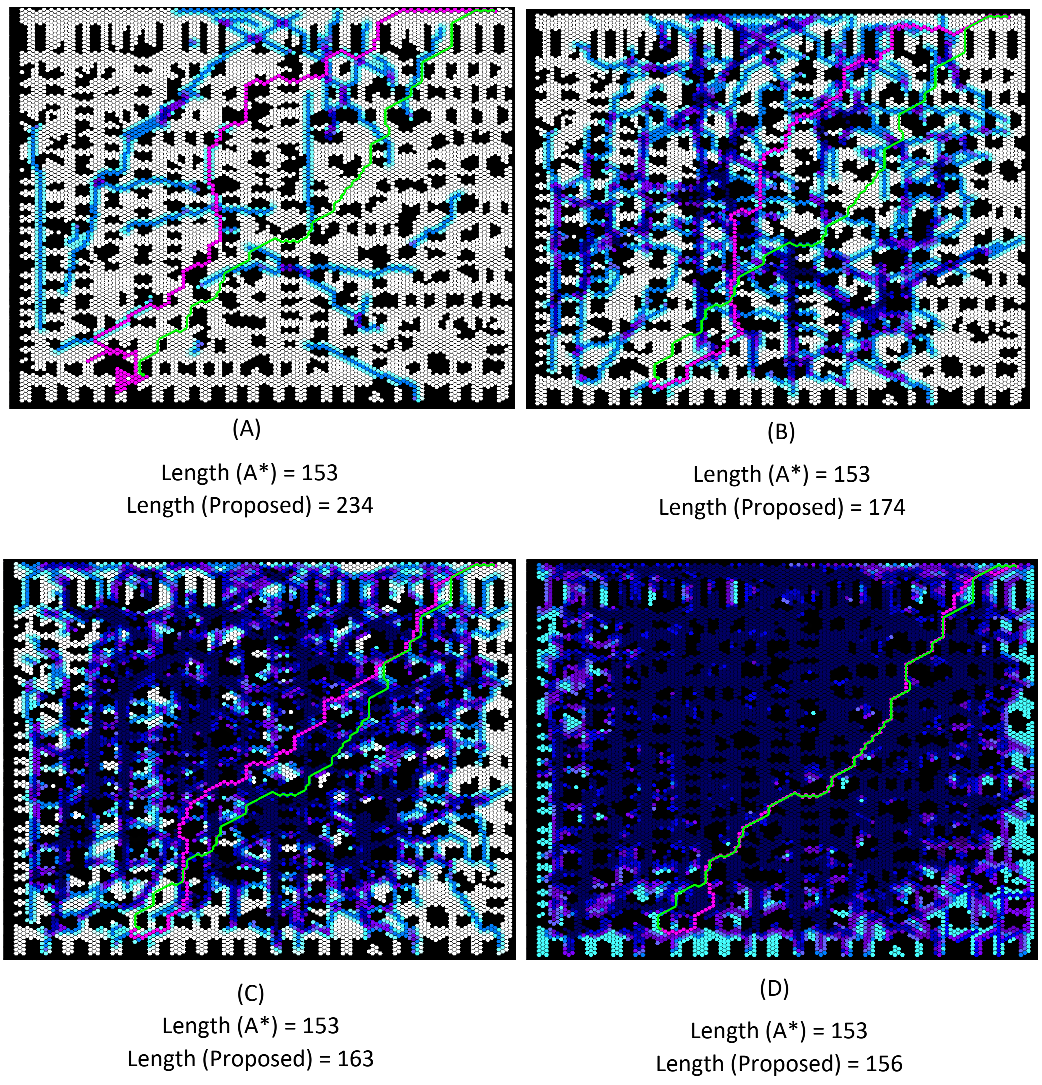


FIGURE 5.6: Percentage of agent's knowledge and obtained path length (Pink color for proposed method and Green for A* search). (A) 25% , (B) 50% , (C) 75% and (D) 100% of knowledge. Color intensity indicates level of knowledge based on how many times it has been visited before (the darker the color the more times it has been visited)

5.6 Experimental Results

In this section we present the results achieved for both informed and uninformed pathfinding, and compare against A*. We show the visual aspects of the path, as well as path length. Thew proposed method has been implemented using C++, with an Inter Core i7 Cpu @3.5 Gz, 1 MB L2 cache and 8MB L3 cache, Nvidia GTX 420 with 2.4GB off-chip global memory and 16 GB RAM. In order to evaluate the affect of percentage of agent's knowledge on calculated path, we have tested the proposed path planner on X=[0,25,50,75,100] percentage of visited places by agent. Figure 5.6

shows results for different percentages of agent's knowledge. By increasing the percentage of agent's knowledge, the length of the informed path will get shorter and the path will get closer to the result provided by the A* search.

For the case of completely unknown environments, we should expect our agents to choose a path that appears rather random and is far from optimal. The random appearance is the result of agents needing to explore and thus checking more locations. In order to see the agent's behavior as we gradually move from unknown to known areas, we have computed paths in different regions of an environment which is only partly known. Figure 5 shows the resulting paths for both our method (exploration with GPS heuristic) and the A* algorithm. As we can see in Figure 5.7, when the agent is located in an unknown area, it first explores a path moving roughly towards the goal direction. Our example avoids the agent from moving too far off the goal direction by using a large δ, while showing the lack of knowledge regarding the exact location of the goal. We can see in the results, how the path shape and length gets closer to A* only for those areas of the environment that the agent knows very well (darker blue indicates higher counter values).

By increasing the percentage of agent's knowledge, the total path length decreases. Figure 5.8 shows the comparison of total path length of our method combining informed and uninformed search, against A*. The image shows how the path length of our algorithm decreases as the familiarity with the environment increases. The paths provided by our method are almost as optimal as A* when $\mathcal{P} > 75\%$.

Figure 5.9 shows the average path ratios as the level of knowledge increases. We can observe how for no knowledge ($\mathcal{P} = 0\%$) the path length we obtain is up to $3.6x$ longer, which demonstrates that our naive exploration provides a solution that is longer than the optimal solution but not too far off from it. When the knowledge is up to 75% the ratio is $1.07x$ which indicates that is very close in length to the optimal solution (ratio 1).

Even though simulating human behavior is a huge challenge, and there is still a lot of work to be done, we have presented a method that attempts to imitate more closely how human find paths in the real world. In order to evaluate whether our

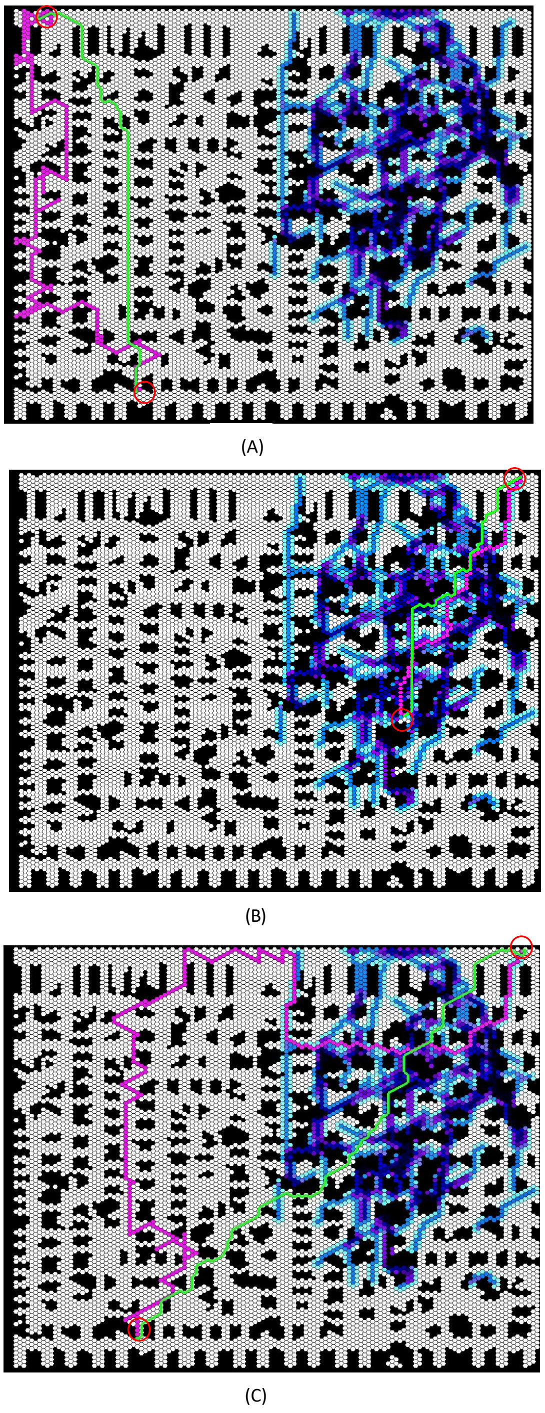


FIGURE 5.7: Illustration of proposed method and A* path planning. Green color shows A* search path, pink color shows proposed method and blue shows agent knowledge with color intensity representing level of knowledge. (A) path planing between two points in an unknown area, (B) Path between two points in a known and (C) path from an unknown area to a location in a known area.

agents' paths could appear as being more or less knowledgeable to a human observer, we have run two perceptual studies.

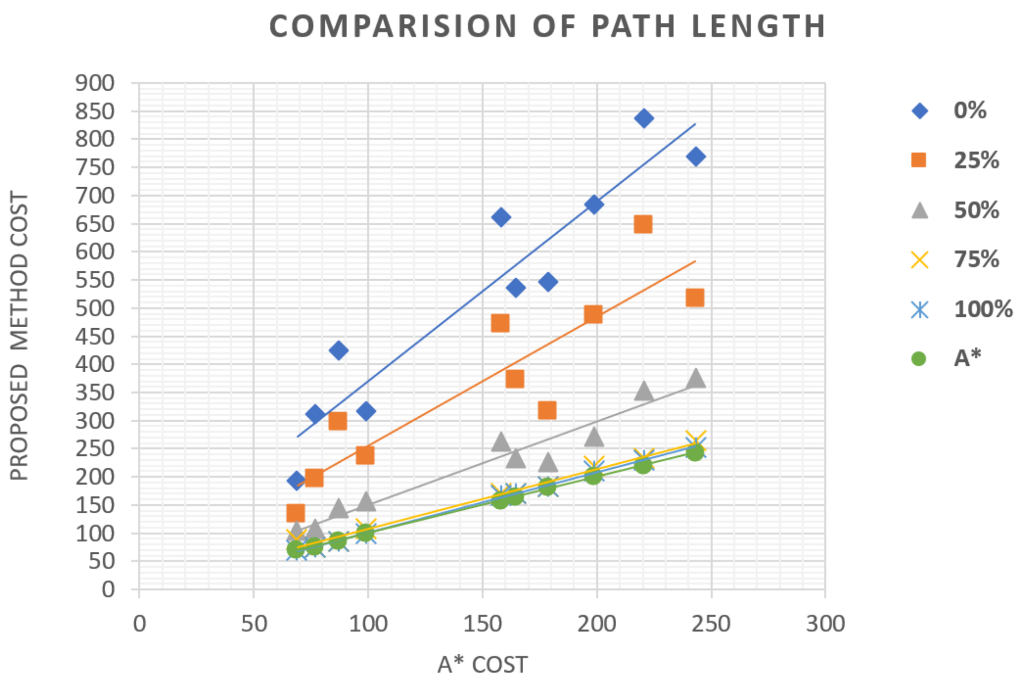


FIGURE 5.8: Comparison of path length

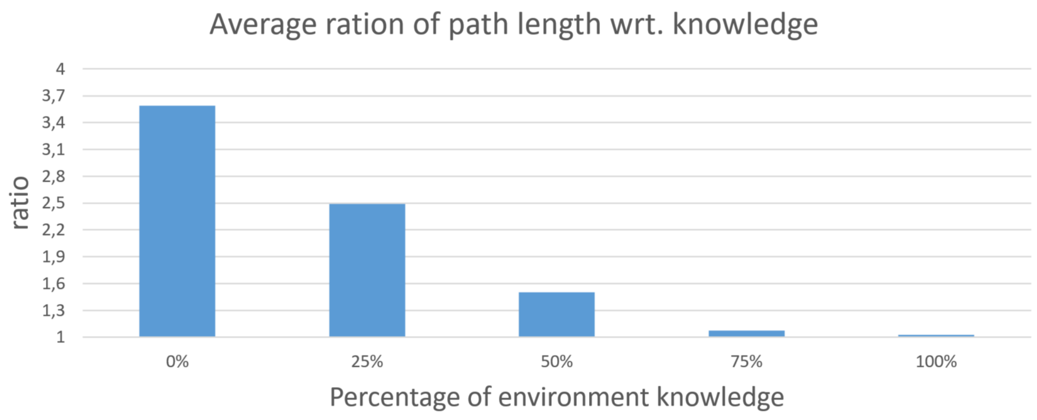


FIGURE 5.9: Ratios of average path length as the level of knowledge about the environment increases, with respect to the A* path length.

Our algorithm combines two types of searches: (1) naive search and (2) a path finder with a heuristic based on the counters in the mental map. The former can exhibit different levels of confidence on the direction of the goal (higher values of δ and smaller values of $\mathcal{C}$ can move the agent quicker towards the goal), and the latest will get closer to A* as $\mathcal{P}$ increases. For the first user study, we set a fixed δ and $\mathcal{C}$, while increasing $\mathcal{P}$. Figure 5.10 show the sample obtained path with fixed δ and $\mathcal{C}$ and increasing value of $\mathcal{P}$.

For the second study, we use varying values for the three parameters. We had a

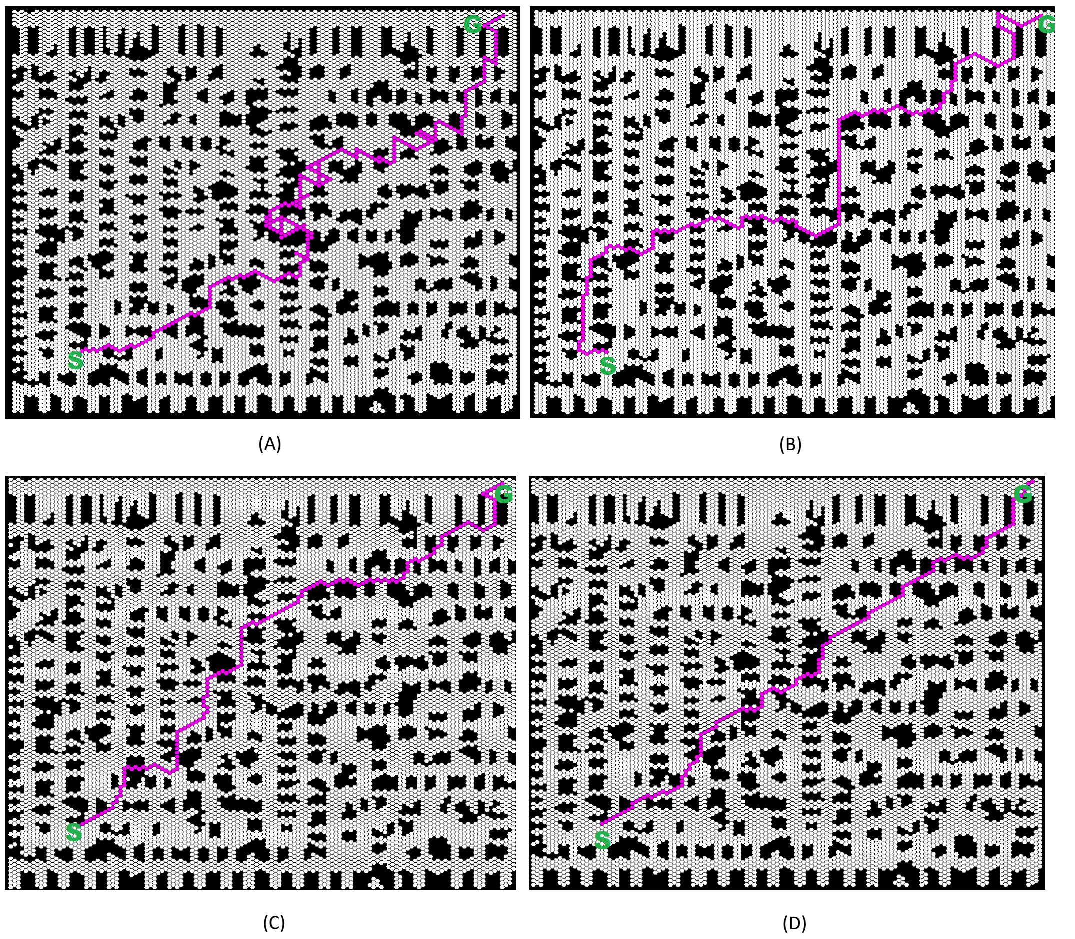


FIGURE 5.10: Percentage of agent's knowledge ($\mathcal{P}$) and obtained path length. (A) $\mathcal{P} = 25\%$, (B) $\mathcal{P} = 50\%$, (C) $\mathcal{P} = 75\%$ and (D) $\mathcal{P} = 100\%$ of knowledge.

total of 40 participants, 20 doing each test. Each study consisted of 2 environments, 4 configurations of start and goal positions per map, and 4 configurations of agents' knowledge. Figure 5.12 shows the user interface of one example page from the 32 pages of our perceptual test framework.

Experiment 1 had 4 configurations: map 0:$\{\mathcal{P} = 0\%\}$, map 1:$\{\mathcal{P} = 35\%\}$, map 2:$\{\mathcal{P} = 75\%\}$, map 3:$\{\mathcal{P} = 100\%\}$, all four maps with $\delta = -0.5$ and $\mathcal{C} = 5$. Figure 5.11 shows a sample of obtained path with different configurations. Participants saw a total of 32 paths, and were asked to look at the path and rank the agent's knowledge, $\mathcal{K}$, as: 0 meaning "very little", 1 "a bit", 2 "quite well" or 3 "extremely well". As we show in the top graph in figure 5.13, participants ranked map 0 with mostly $\mathcal{K} = \{0, 1\}$, map 1 with $\mathcal{K} = \{1, 2\}$, map 2 with $\mathcal{K} = \{2, 3\}$, and map 3 with mostly $\mathcal{K} = \{3\}$. We ran a χ^2 and obtained a p–value=0 indicating that there is a statistically significant relationship between the map configuration and the user's

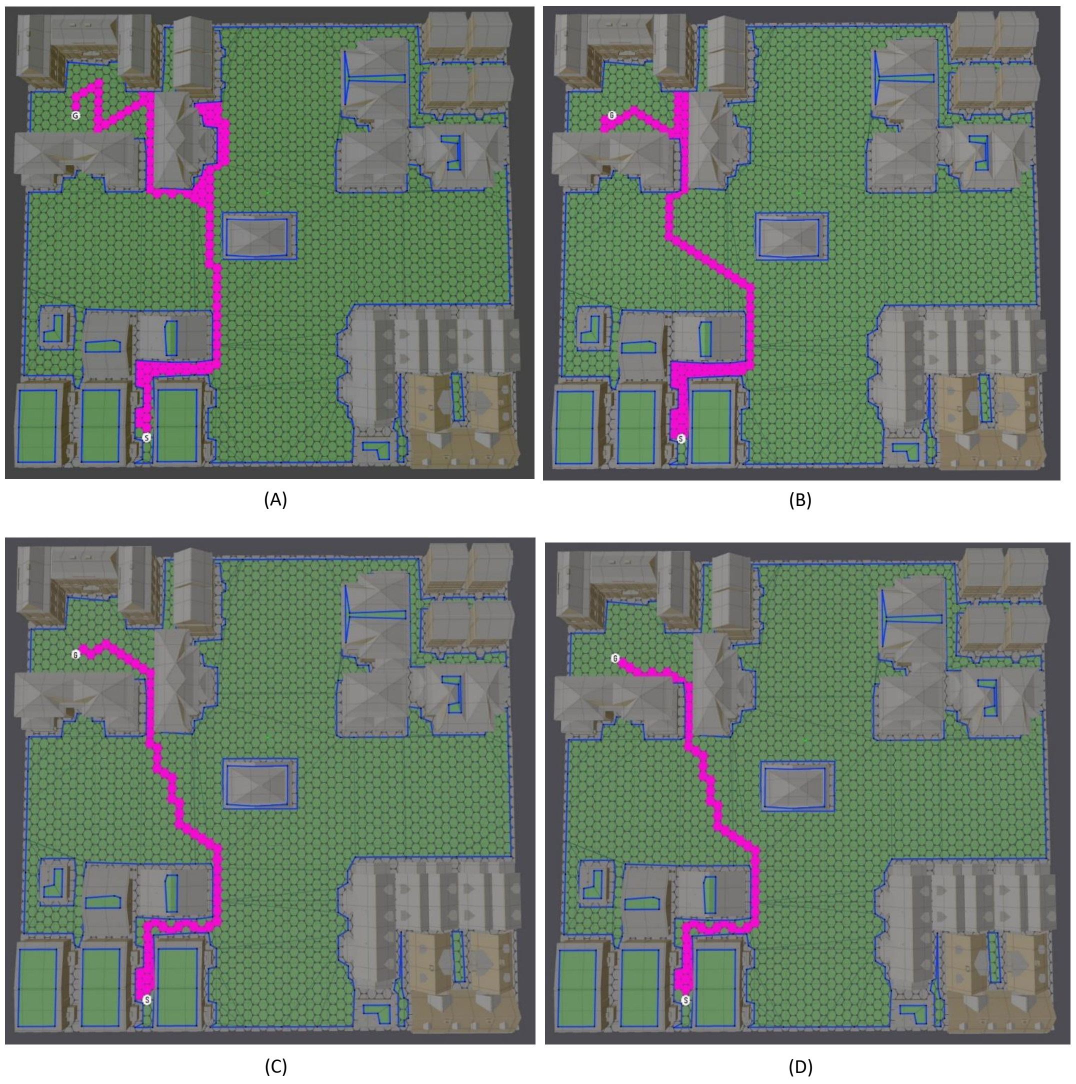
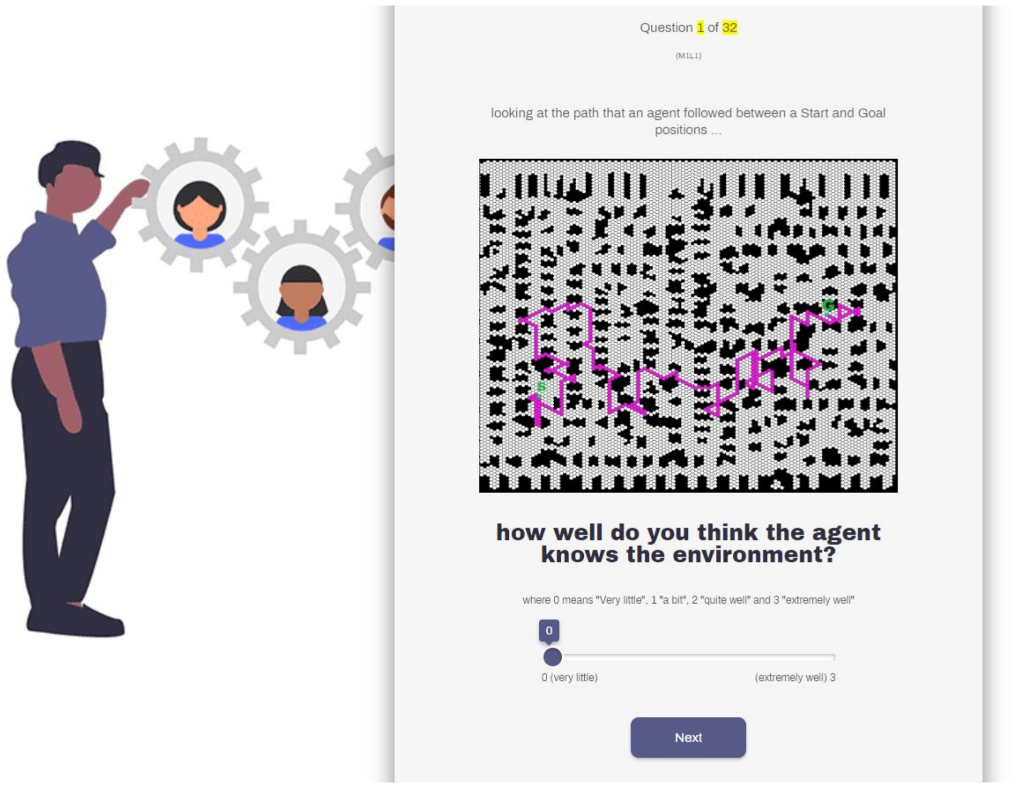


FIGURE 5.11: Different values of three $\mathcal{P}$, δ and $\mathcal{C}$ parameters and obtained path length. (A): $\{\mathcal{P} = 0\%, \delta = -1, \mathcal{C} = 10\}$, (B): $\{\mathcal{P} = 35\%, \delta = -0.8, \mathcal{C} = 8\}$, (C): $\{\mathcal{P} = 70\%, \delta = -0.6, \mathcal{C} = 6\}$, (D): $\{\mathcal{P} = 100\%, \delta = -0.4, \mathcal{C} = 4\}$

perceived level of knowledge. This means that participants either guessed correctly the level of knowledge for each map, or else they slightly overestimated it. The reason for this, is that the naive search made the agents move quite well towards the goal direction.

We then run experiment 2, trying to assign δ and $\mathcal{C}$ with values that better matched level of confidence with levels of knowledge. Therefore, we had the following map configurations:

- map 0: $\{\mathcal{P} = 0\%, \delta = -1, \mathcal{C} = 10\}$
- map 1: $\{\mathcal{P} = 35\%, \delta = -0.8, \mathcal{C} = 8\}$



Question 1 of 32

(M1L1)

looking at the path that an agent followed between a Start and Goal positions ...

how well do you think the agent knows the environment?

where 0 means "Very little", 1 "a bit", 2 "quite well" and 3 "extremely well"

0

0 (very little) (extremely well) 3

FIGURE 5.12: User interface of perceptual test

- map 2: $\{\mathcal{P} = 70\%, \delta = -0.6, \mathcal{C} = 6\}$
- map 3: $\{\mathcal{P} = 100\%, \delta = -0.4, \mathcal{C} = 4\}$

The χ^2 test gave us a p–value=0 indicating again that there is a statistically significant relationship between the map configuration and the user's perceived level of knowledge. The bottom graph of Figure 5.13, show that for the second experiment, users perceived the resulting paths as being closer to our intended configuration, therefore each map level got the highest number of answers matching the corresponding knowledge level intended for each map. The Pearson rank correlation between the map knowledge and the user's perceived agent knowledge was $r_s = 0.86$, indicating a strong relationship between them.

5.7 Conclusion

In this chapter we have proposed a pathfinding method that attempts to consider the human's brain navigation system to simulate more human-like autonomous agents. We also propose a more human-like exploration method for unknown environments

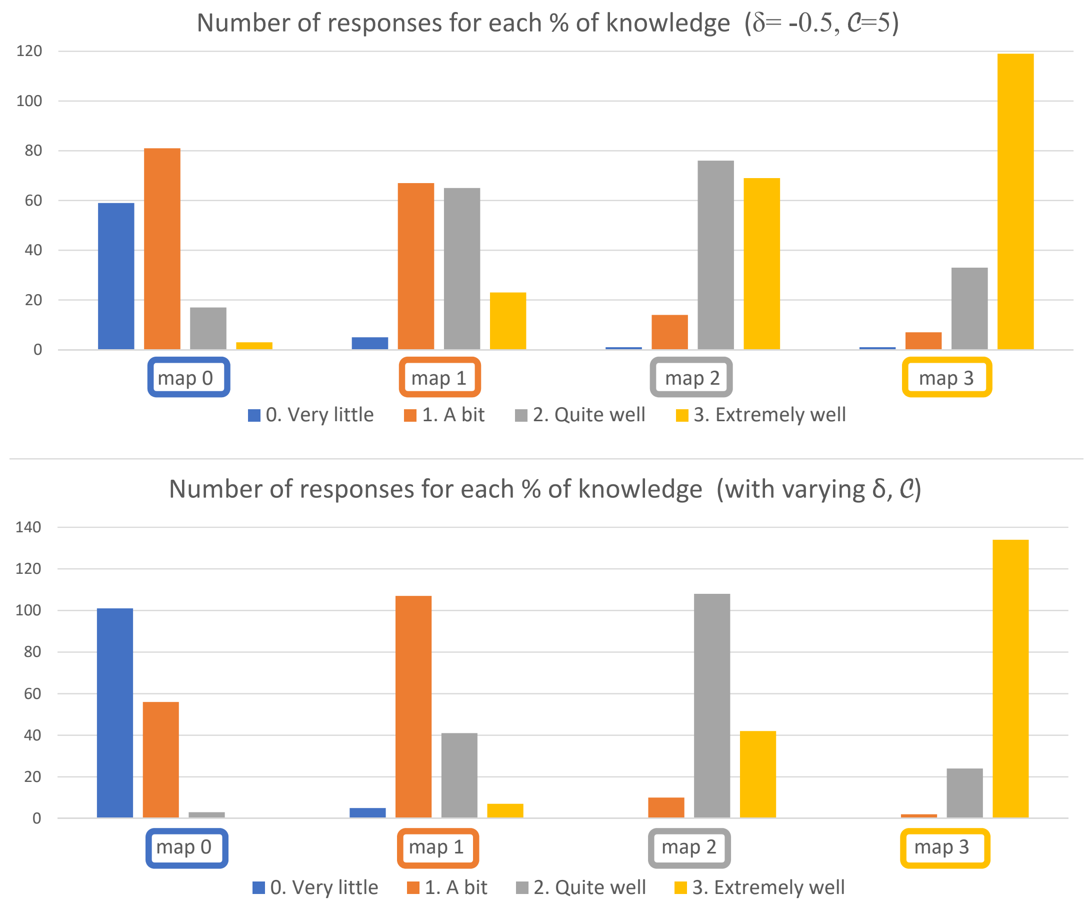


FIGURE 5.13: Perceptual Evaluation. The top graph shows the perceived level of familiarity for maps of increasing $\mathcal{P}$, with $\delta = 0.5$ and $\mathcal{C} = 5$. The bottom shows also maps of increasing $\mathcal{P}$, but varying δ and $\mathcal{C}$ to also exhibit increasing levels of confidence on the goal direction.

with vague knowledge of goal direction. We believe that this is the first attempt towards simulating more human-like pathfinding.

Our method can work with known, unknown and mixed environments. The hexagonal grid navigation mesh mimics the humans' brain grid cell. Cell counters simulate the way our brain keeps track of visited places as agent's memory. The proposed naive exploration uses a variation of the Depth First Search (DFS) algorithm to consider vague information of the environment (rough knowledge of goal direction), and builds a cognitive map for the agent as it wanders the environment.

Pathfinding in known environment, is carried out by applying a modified heuristic to A*. The new heuristic considers the cognitive map counters as the agents' memory. Our experimental results show that path length for the proposed method converges towards the traditional A* search as the agent acquires more knowledge of the environment. We have also shown how the resulting paths are perceived as being more or less knowledgeable, based on the values assigned to the parameters of our model. As future work it would be interesting to consider memory decay and also other aspects of human perception that may affect the way we remember places (for example based on their saliency or uniqueness).

Chapter 6

Conclusion and future work

6.1 Conclusion

The focus of this work was twofold, first to research hierarchical solutions for pathfinding that could be parallelized to support multi-agent pathfinding in real-time, and second to develop novel methods that could better mimic the way humans navigate. Hierarchical approaches are by nature closer to human pathfinding, because we humans typically plan our trajectories from a high level conceptual map and then refine the path as needed.

Part of this thesis has been built upon the original HNA* algorithm that was previously developed by my supervisor. During the years leading to its completion, we have worked on first identifying the source of the bottleneck that had been empirically found in the first version of the algorithm. And then focus on finding solution that could guarantee that using our hierarchical approach would always lead to high speed-ups regardless of the hierarchy configuration.

In chapter 3 we presented two solutions to the S/G connection step. The first one consists of using pre-calculated paths (PCCP) from the center of each polygon in L0 (lowest level of the navigation mesh) to its *inter-edges* in the higher level of the hierarchy. Those paths are then stored in a MultiMap hash table and can be accessed efficiently during the on-line search. The second one takes advantage of the highly parallel nature of the problem, and presents a new approach using the CPU or the GPU (with CUDA), so that all sub-paths to connect S and G can be computed in parallel. As we have discussed in chapter 3-section 3.6, achieved results showed that the parallelization on GPU can be much faster than the CPU.

To evaluate our different methods we used several 3D scenarios with increasing numbers of cells in their navigation mesh and increasing numbers of merged polygons. Our results show that both PCCP and CUDA methods can be much faster than the original HNA*.

PCCP requires more memory usage than CUDA, although this does not present a limitation. The allocated memory for the Dungeon scenario with 119 polygon is 2.9MB while the allocated memory for the Medieval City scenario with 16,867 polygons is 49.6MB. For all tested scenarios, the performance improvements are not very important for L1, but they become substantial from L2 onward, as they eliminate the bottleneck of HNA* which was the connect S and G step.

Our results showed that both PCCP and the CUDA implementation could achieve significantly better speed-ups than the original HNA*, and most importantly they could guarantee that we could benefit from the hierarchical approach for any given configuration, as opposed to the original HNA* which required a careful selection of parameters. It is though recommended a hierarchy of two or more levels to obtain the best speed-ups, because for just one level and a low value of μ, the original HNA* did not suffer much from the connect S/G bottleneck.

Therefore, with the improvements presented in this thesis, we have completely eliminated the bottleneck from HNA* and thus obtained a hierarchical pathfinding algorithm for general navigation meshes that offers great speed-ups for a larger number of scenarios, regardless of the hierarchy configuration.

In chapter 4 we studied in depth the levels of parallelism available in CUDA and presented a parallel version for multi-agent system using the HNA*. We then carried out a thorough performance comparison of a parallel multi-agent implementation of A*, PCCP and CUDA-HNA* in order to determine the potential of using hierarchical pathfinding. For this comparison all three methods can compute pathfinding for multiple agents in parallel. For the parallel implementation of the step connecting S and G in CUDA-HNA*, two threads were dedicated to launch two child kernels (one to connect S and another one to connect G). As we have shown in our results, the speed-ups achieved by both our methods outperform the parallel A* solution. For the 4 scenarios tested, we can observe speedups on average between 4.3x and

15.7x for PCCP, and between 3.6x and 9.8x for CUDA-HNA*. Therefore, the benefits of our hierarchical representations still hold even when a parallel implementation could be carried out for both HNA* and A*. As our results showed, the parallel implementation in CUDA can handle real time pathfinding for over 500K agents even when using the basic A* algorithm, but with an important speed-up achieved by using HNA* with the connection step in parallel.

Finally in chapter 5, we have presented a new pathfinding method that attempts to consider the human's brain navigation system to simulate more human-like autonomous agents. We also propose a more human-like exploration method for unknown environments with vague knowledge of goal location. We believe that this is the first attempt towards simulating more human-like pathfinding. Our algorithm performs graph exploration differently depending on whether the environment is known or unknown, and can adjust to partly known environments. The hexagonal grid navigation mesh mimics the human's brain grid cell. Per cell counters, simulate the way our brain keeps track of visited places as agent's memory.

The proposed method uses a variation of the Depth First Search (DFS) algorithm to consider vague information of the environment (rough knowledge of goal position), and creates a cognitive map for the agent as it wanders the environment. Pathfinding in known environment, is carried out by applying a modified heuristic to A*. The new heuristic considers the cognitive map counters as the agents' memory. Our experimental results showed that path length for the proposed method converges towards the traditional A* search as the agent acquires more knowledge of the environment.

We also run a perceptual study to evaluate whether our resulting paths did exhibit the intended level of knowledge. The results of this study showed that users successfully identified the knowledge of the autonomous agents. We believe that our model will open a new way of programming autonomous agents to closer simulate virtual humanoids.

6.2 Future Works

We discuss future work in terms of our three main contributions: Improvements to HNA* (see chapter 3), Multi-agent parallel HNA* (MA-HNA*) (see chapter 4) and Towards human like agent path planning (see chapter 5).

6.2.1 Improvements to HNA*

As we have mentioned in chapter 3, the PCCP consists of two online and offline phases. In the offline phase, PCCP stores paths from center of each polygon in L0 (lowest level of the navigation mesh) to its *inter-edges* in the higher level of the hierarchy. These paths are used to connect start and goal points to the higher level graph at the online phase. One of the challenge of the PCCP on the offline phase is that if any dynamic changes happen on the given scenario, all the paths from offline phase should recalculate. On the other hand, hierarchical navigation graph (HNG) doses not consider dynamic changes and dynamic environment. As future work we would also like to consider dynamic updates of the NavMesh and how they could affect the hierarchical representation and then recompute the paths which are from the changed area of scenario instead of recomputing all the paths.

6.2.2 Multi-agent parallel HNA* (MA-HNA*)

As it described at chapter 4, for the parallel implementation of the step connecting S and G, two threads are dedicated to launch two child kernels (one to connect S and another one to connect G). In order to launch the child kernels to connect S in parallel, we consider m 1D blocks in the 1D CUDA grid where, m is the number of *inter-edges* of the polygon containing S. For each block B_i ($i \in [0, m]$) we compute A* from S to *inter-edge* ie_i (and similarly for G). All connecting paths to S and G are stored in shared memory as temporal edges of the HNG before computing the high-level path. So the connect step for S and G will take as long as the longest of the A* searches to link S or G to an inter-edge. As future work we would like to paralyze the A* search algorithm inside each child kernel and using GPU shared memory to store the open list. Since the shared memory is much faster than global memory, the GPU based parallel A* could be more faster than CPU based A*. Another possible way to improve multi-agent path finding could be to handle the case when parts of the subpath computed by one agent could be used by other agents sharing part of the path.

We would like also to implement our MA-HNA* for cloud computing applications like SpatialOS frameworks. SpatialOS [Improbable, 2020] is a cloud development platform that provides networking, hosting, online services, and tools for developing and operating online multiplayer games, using any engine.

6.2.3 Towards human like agent path planning

In chapter 5 we have proposed a novel bio-inspired perspective for agent path planning based on human brain structure. As we have explained, the proposed algorithm works on static environments. As a future work it would be interesting to consider dynamic environments. The other future work which we think it would be interesting is considering memory decay and also other aspects of human perception that may affect the way we remember places (for example based on their saliency or uniqueness).

As future work we would like to combine our new GPS pathfinding algorithm with the hierarchical search on navigation meshes. When humans plan trajectories, they typically think of a high level map where main known locations appear and then they refine their search (e.g cities and main roads, to then focus on specific streets). Such way of planning is combined in our brains with the hexagonal distribution of neurons that allow us to remember locations. Therefore, our virtual agents should also combine both spacial structures, with the hexagonal one being used to gather information and drive the heuristics, but without consciously planning at that level. Both the navigation mesh and the hexagonal grid structure could be overlapping the navigation mesh, and then we could transfer knowledge information from the hexagonal cells to the convex cells that are overlapping with them.

Appendix A

Mathematical profs 1

If a path exists over the original navigation mesh, G_0, $P_0(S,G) = \langle p_S, p_1, p_2, ..., p_G \rangle$, then there will also be a path at level Lx. Computing path finding in HNG_x gives as a result the path $P_x(S,G) = \langle \pi^S_{temp}, \pi_x^{s(dp)}, \pi_x^{p(sq)}, ..., \pi_x^{r((m-1)m)}, \pi^G_{temp} \rangle$. $P_x(S,G)$ is the high level path.

Proof. Starting with the sequence of polygons in the path:

$$P_0(S,G) = \langle p_S, p_1, p_2, ..., p_G \rangle \tag{A.1}$$

the starting polygon p_S will be inside a high level node n^s_x in the HNG_x. Moving from left to right in the sequence of polygons while $p_i \in n^s_x$, we will eventually reach a polygon p_j such that $p_j \in n^d_x$, where $d \neq s$. According to the definition of *inter-edge* given in Section 3.3, this means that there is an *inter-edge* ι^{sd}_x connecting the nodes n^s_x and n^d_x in HNG_x The sequence of polygons from p_s to p_{j-1} correspond to the temporal path connecting S to the *inter-edge* ι^{sd}_x:

$$\pi^S_{temp} = \langle p_S, ..., p_i, ..., p_{j-1} \rangle \tag{A.2}$$

From p_j, we can continue sequentially while the polygons we encounter are still inside n^d_x, until there is a polygon p_k such that $p_k \in n^p_x$, and $p \neq d$. By the definition of *inter-edge*, there is an *inter-edge*, ι_x^{dp}, that connects n^d_x, with n^p_x.

Also, the sequence of nodes $\langle p_j, p_{j+1}, ..., p_{k-1} \rangle$ indicates that there is a path traversing the node n^d_x between the *inter-edges* ι^{sd}_x and ι_x^{dp}, which guarantees that there is at least one path between those two *inter-edges*, and thus there will be an *intra-edge*, $\exists \pi_x^{s(dp)}$. Note that this does not mean that the stored *intra-edge* is specifically the sequence $\langle p_j, p_{j+1}, ..., p_{k-1} \rangle$, since the subpath P' was computed with A* between optimal positions inside the polygons in the navigation mesh, whereas $\pi_x^{s(dp)}$ was computed assuming the position at the center of the polygons. If there was only one possible path, then this path will be the optimal, and thus we will have $P' = \pi_x^{s(dp)}$. So our proof guarantees that if there is a path in $P0$ crossing a node, there must be an *intra-edge* to cross the node.

$$\pi_x^{s(dp)} = \langle p_j, p_{j+1}, ..., p_{k-1} \rangle \tag{A.3}$$

Following the same logic, it is thus straight forward to proof that every sequence of polygons inside the same high level node, guarantees that there will be an *intra-edge* traversing such node, and that for every pair of polygons in the sequence, which appear inside different high level nodes, there will be an *inter-edge* in the HNG_x.

The the last sequence of nodes that are inside the high level node containing p_G, will correspond to the temporal path connecting G π_{temp}^G

$$\pi_{temp}^G = \langle p_{l+1}, p_{l+2}, ..., p_G \rangle, \forall p_i \in n_x^q \tag{A.4}$$

Finally we can see that replacing the subsequence of polygons from temporal paths and *intra-edges* guarantees that there will have a path between S and G:

$$\begin{aligned}
P_x(S,G) &= \langle \pi_{temp}^S, \pi_x^{s(dp)}, \pi_x^{p(sq)}, ..., \pi_{temp}^G \rangle \\
&= \langle < p_S, ..., p_{j-1} >, \pi_x^{s(dp)}, \pi_x^{p(sq)}, ..., \pi_{temp}^G \rangle \\
&= \langle < p_S, ..., p_{j-1} >, < p_j, p_{j+1}, ..., p_{k-1} >, \pi_x^{p(sq)}, ..., \pi_{temp}^G \rangle \\
&= \langle < p_S, ..., p_{j-1} >, < p_j, p_{j+1}, ..., p_{k-1} >, \\
&< p_k, p_{k+1}, ..., p_{m-1} >, ..., \pi_{temp}^G \rangle \\
&= \langle < p_S, ..., p_{j-1} >, < p_j, p_{j+1}, ..., p_{k-1} >, \\
&< p_k, p_{k+1}, ..., p_{m-1} >, ..., < p_{l+1}, p_{l+2}, ..., p_G > \rangle \\
&= \langle p_S, p_1, p_2, ..., p_G \rangle \\
&= P_0(S,G)
\end{aligned} \tag{A.5}$$

□

Appendix B

Mathematical profs 2

Proof. The number of *inter-edges* increases exponentially with the number of levels in the hierarchy:

As indicated in Equation B.1, the number of inter-edges for a node in level x are:

$$I_{(x,\mu)} = \mu I_{(x-1,\mu)} - 2(\mu - 1) \tag{B.1}$$

if we replace $I_{(x-1,\mu)}$ by its corresponding equation, then we have:

$$\begin{aligned} I_{(x,\mu)} &= \mu \left[\mu I_{(x-2,\mu)} - 2(\mu - 1) \right] - 2(\mu - 1) \\ &= \mu^2 I_{(x-2,\mu)} - 2\mu(\mu - 1)) - 2\mu + 2) \\ &= \mu^2 (I_{(x-2,\mu)} - 2) + 2 \end{aligned} \tag{B.2}$$

replacing the next level down in the hierarchy, then we have:

$$\begin{aligned} I_{(x,\mu)} &= \mu^2 \left[\mu I_{(x-3,\mu)} - 2(\mu - 1) - 2 \right] + 2 \\ &= \mu^2 \left[\mu I_{(x-3,\mu)} - 2\mu + 2 - 2 \right] + 2 \\ &= \mu^3 (I_{(x-3,\mu)} - 2) + 2 \end{aligned} \tag{B.3}$$

and if we continue recursively until we reach level 1:

$$I_{(x,\mu)} = \mu^{x-1} (I_{(1,\mu)} - 2) + 2 \tag{B.4}$$

finally, replacing $I_{(1,\mu)}$ by Equation B.1 we obtain:

$$\begin{aligned} I_{(x,\mu)} &= \mu^{x-1}(\mu(s-2)+2-2)+2 \\ &= \mu^{x}(s-2)+2 \end{aligned} \tag{B.5}$$

□

Bibliography

Abd Algfoor, Zeyad, Mohd Shahrizal Sunar, and Hoshang Kolivand (2015). “A comprehensive study on pathfinding techniques for robotics and video games”. In: *International Journal of Computer Games Technology* 2015.

Abraham, Ittai, Amos Fiat, Andrew V Goldberg, and Renato F Werneck (2010). “Highway dimension, shortest paths, and provably efficient algorithms”. In: *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, pp. 782–793.

Alismail, Hatem, L Douglas Baker, and Brett Browning (2014). “Continuous trajectory estimation for 3D SLAM from actuated lidar”. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 6096–6101.

Ammar, Adel, Hachemi Bennaceur, Imen Châari, Anis Koubâa, and Maram Alajlan (2016). “Relaxed Dijkstra and A* with linear complexity for robot path planning problems in large-scale grid environments”. In: *Soft Computing* 20.10, pp. 4149–4171.

Anguelov, Bobby (2012). “ideo game pathfinding and improvements to discrete search on grid-based maps”. In: *Doctoral dissertation, University of Pretoria*.

Bennewitz, Maren, Wolfram Burgard, and Sebastian Thrun (2002). “Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots”. In: *Robotics and autonomous systems* 41.2-3, pp. 89–99.

Bnaya, Zahy, Roni Stern, Ariel Felner, Roie Zivan, and Steven Okamoto (2013). “Multiagent path finding for self interested agents”. In:

Bohlin, Robert and Lydia E Kavraki (2000). “Path planning using lazy PRM”. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 1. IEEE, pp. 521–528.

BOOST (2017). "http://www.boost.org/".

Borrajo, Daniel (2013). "Plan sharing for multi-agent planning". In: *DMAP 2013-Proceedings of the Distributed and Multi-Agent Planning Workshop at ICAPS*. Citeseer, pp. 57–65.

Botea, Adi, Martin Müller, and Jonathan Schaeffer (2004). "Near optimal hierarchical path-finding". In: *Journal of game development* 1.1, pp. 7–28.

Brafman, Ronen I and Carmel Domshlak (2008). "From One to Many: Planning for Loosely Coupled Multi-Agent Systems." In: *ICAPS*. Vol. 8, pp. 28–35.

Brand, Sandy and Rafael Bidarra (2012). "Multi-core scalable and efficient pathfinding with Parallel Ripple Search". In: *computer animation and virtual worlds* 23.2, pp. 73–85.

Broch, Josh, David A Maltz, David B Johnson, Yih-Chun Hu, and Jorjeta Jetcheva (1998). "A performance comparison of multi-hop wireless ad hoc network routing protocols". In: *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pp. 85–97.

Bulitko, Vadim, Yngvi Björnsson, and Ramon Lawrence (2010). "Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding". In: *Journal of Artificial Intelligence Research (JAIR)* 39, pp. 269–300.

Caggianese, Giuseppe and Ugo Erra (2012). "Exploiting gpus for multi-agent path planning on grid maps". In: *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, pp. 482–488.

Cazenave, Tristan (2006). "Optimizations of data structures, heuristics and algorithms for path-finding on maps". In: *2006 IEEE symposium on computational intelligence and games*. IEEE, pp. 27–33.

Champandard, A (2009). "Modern pathfinding techniques". In: *AIGameDev. com*.

Chen, Haoyao and Dong Sun (2012). "Moving groups of microparticles into array with a robot–tweezers manipulation system". In: *IEEE Transactions on Robotics* 28.5, pp. 1069–1080.

Choset, Howie (2007). "Robotic motion planning: A* and D* search". In: *Robotics Institute*, pp. 16–735.

Choset, Howie M, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E Kavraki, and Sebastian Thrun (2005). *Principles of robot motion: theory, algorithms, and implementation*. MIT press.

Chrpa, Lukáš and Peter Novák (2011). "Dynamic trajectory replanning for unmanned aircrafts supporting tactical missions in urban environments". In: *International*

Conference on Industrial Applications of Holonic and Multi-Agent Systems. Springer, pp. 256–265.

Cohen, Benjamin, Sachin Chitta, and Maxim Likhachev (2014). "Single-and dual-arm motion planning with heuristic search". In: *The International Journal of Robotics Research* 33.2, pp. 305–320.

Crosby, Matt, Michael Rovatsos, and Ronald PA Petrick (2013). "Automated Agent Decomposition for Classical Planning." In: *ICAPS*, pp. 46–54.

Daniel, Kenny, Alex Nash, Sven Koenig, and Ariel Felner (2010). "Theta*: Any-angle path planning on grids". In: *Journal of Artificial Intelligence Research* 39, pp. 533–579.

Davis, Ian Lane (2000). "Warp speed: Path planning for star trek: Armada". In: *AAAI Spring Symposium (AIIDE)*, pp. 18–21.

Dechter, Rina and Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A". In: *Journal of the ACM (JACM)* 32.3, pp. 505–536.

DeLoura, Mark A (2001). *Game programming gems 2*. Cengage learning.

Demyen, Douglas and Michael Buro (2006). "Efficient triangulation-based pathfinding". In: *Aaai*. Vol. 6, pp. 942–947.

Dijkstra, Edsger W et al. (1959). "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1, pp. 269–271.

Dolgov, Dmitri, Sebastian Thrun, Michael Montemerlo, and James Diebel (2008). "Practical search techniques in path planning for autonomous driving". In: *Ann Arbor* 1001.48105, pp. 18–80.

Dresner, Kurt and Peter Stone (2008). "A multiagent approach to autonomous intersection management". In: *Journal of artificial intelligence research* 31, pp. 591–656.

Elbanhawi, Mohamed and Milan Simic (2014). "Sampling-based robot motion planning: A review". In: *Ieee access* 2, pp. 56–77.

Farias, Renato and Marcelo Kallmann (2019). "Optimal Path Maps on the GPU". In: *IEEE transactions on visualization and computer graphics*.

Farnstrom, Frederick (2006). "Improving on near-optimality: More techniques for building navigation meshes". In: *AI Game Programming Wisdom* 3, p. 2006.

Ferguson, Dave and Anthony Stentz (2005). "The Field D* algorithm for improved path planning and replanning in uniform and non-uniform cost environments". In: *Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-05-19*.

Ferguson, Dave and Anthony Stentz (2007). “Field D*: An interpolation-based path planner and replanner”. In: *Robotics research*. Springer, pp. 239–253.

Franz, Matthias O and Hanspeter A Mallot (2000). “Biomimetic robot navigation”. In: *Robotics and autonomous Systems* 30.1-2, pp. 133–153.

Furelos Blanco, Daniel and Anders Jonsson (2018). “Solving concurrent multiagent planning using classical planning”. In: *Stolba M, Komenda A, editors. DMAP 2018. Proceedings of the 6th Workshop on Distributed and Multi-Agent Planning; 2018 Jun 24-29; Delft, the Netherlands. Palo Alto (CA): AAAI; 2018. p. 8-16.* Association for the Advancement of Artificial Intelligence (AAAI)-Congrés . . .

Galceran, Enric and Marc Carreras (2013). “A survey on coverage path planning for robotics”. In: *Robotics and Autonomous systems* 61.12, pp. 1258–1276.

Garcia, Francisco M, Mubbasir Kapadia, and Norman I Badler (2014). “GPU-based dynamic search on adaptive resolution grids”. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 1631–1638.

Geraerts, RJ and Mark H Overmars (2005). “Creating small roadmaps for solving motion planning problems”. In: *Proc. 11th IEEE International Conference on Methods and Models in Automation and Robotics*, pp. 531–536.

Goldberg, Andrew V and Chris Harrelson (2005). “Computing the shortest path: A search meets graph theory”. In: *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, pp. 156–165.

Goldberg, Andrew V, Haim Kaplan, and Renato F Werneck (2006). “Reach for A*: Efficient point-to-point shortest path algorithms”. In: *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, pp. 129–143.

Graham, Ross, Hugh McCabe, and Stephen Sheridan (2015). “Pathfinding in computer games”. In: *The ITB Journal* 4.2, p. 6.

Hafting, Torkel, Marianne Fyhn, Sturla Molden, May-Britt Moser, and Edvard I Moser (2005). “Microstructure of a spatial map in the entorhinal cortex”. In: *Nature* 436.7052, p. 801.

Hamm, David (2008). “Navigation mesh generation: An empirical approach”. In: *AI Game Programming Wisdom* 4, pp. 113–124.

Harabor, Daniel and Adi Botea (2008). "Hierarchical path planning for multi-size agents in heterogeneous environments". In: *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium on*. IEEE, pp. 258–265.

— (2010). "Breaking path symmetries on 4-connected grid maps". In: *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Harabor, Daniel et al. (2014). "Fast and Optimal Pathfinding". In:

Harabor, Daniel Damir and Alban Grastien (2011). "Online graph pruning for pathfinding on grid maps". In: *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Hart, Peter E, Nils J Nilsson, and Bertram Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

Holte, R.C., R. C. Holte, M.B. Perez, M. B. Perez, R. M. Zimmer, R. M. Zimmer, A.J. MacDonald, and A. J. Macdonald (1996a). "Hierarchical A*: Searching Abstraction Hierarchies Efficiently". In: *In Proceedings of the National Conference on Artificial Intelligence*, pp. 530–535.

Holte, Robert C, Chris Drummond, Maria B Perez, Robert M Zimmer, and Alan J MacDonald (1994). "Searching with abstractions: A unifying framework and new high-performance algorithm". In: *Proceedings of the biennial conference-Canadian society for computational studies of intelligence*, pp. 263–270.

Holte, Robert C, Taieb Mkadmi, Robert M Zimmer, and Alan J MacDonald (1996b). "Speeding up problem solving by abstraction: A graph oriented approach". In: *Artificial Intelligence* 85.1, pp. 321–361.

Improbable (2020). In: URL: `https://improbable.io/spatialos`.

Jaillet, Léonard, Juan Cortés, and Thierry Siméon (2008). "Transition-based RRT for path planning in continuous cost spaces". In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, pp. 2145–2150.

Jansen, M Renee and Michael Buro (2007). "HPA* Enhancements." In: *AIIDE* 7, pp. 84–87.

Jiménez, Andrés C, Vicente García-Díaz, Rubén González-Crespo, and Sandro Bolaños (2018). "Decentralized Online Simultaneous Localization and Mapping for Multi-Agent Systems". In: *Sensors* 18.8, p. 2612.

Johnson, Geraint (2006). "Smoothing a navigation mesh path". In: *AI Game Programming Wisdom* 3, pp. 129–139.

Jurney, Chris and S Hubick (2007). "Dealing with destruction: Ai from the trenches of company of heroes". In: *Game Developers Conference*.

Kallmann, Marcelo (2010a). "Navigation queries from triangular meshes". In: *International Conference on Motion in Games*. Springer, pp. 230–241.

— (2010b). "Shortest Paths with Arbitrary Clearance from Navigation Meshes." In: *Symposium on Computer Animation*, pp. 159–168.

Kallmann, Marcelo and Mubbasir Kapadia (Jan. 1, 2016). *Geometric and Discrete Path Planning for Interactive Virtual Worlds*. Morgan and Claypool, p. 201. published.

Kapadia, Mubbasir, Kai Ninomiya, Alexander Shoulson, Francisco Garcia, and Norman Badler (2013a). "Constraint-aware navigation in dynamic environments". In: *Proceedings of Motion on Games*, pp. 111–120.

Kapadia, Mubbasir, Alejandro Beacco, Francisco Garcia, Vivek Reddy, Nuria Pelechano, and Norman I Badler (2013b). "Multi-domain real-time planning in dynamic environments". In: *Proceedings of the 12th ACM SIGGRAPH/Eurographics symposium on computer animation*. ACM, pp. 115–124.

Kapadia, Mubbasir, Nuria Pelechano, Jan Allbeck, and Norm Badler (2015). "Virtual crowds: Steps toward behavioral realism". In: *Synthesis lectures on visual computing: computer graphics, animation, computational photography, and imaging* 7.4, pp. 1–270.

Karaman, Sertac and Emilio Frazzoli (2011). "Sampling-based algorithms for optimal motion planning". In: *The international journal of robotics research* 30.7, pp. 846–894.

Kavraki, Lydia and J-C Latombe (1994). "Randomized preprocessing of configuration for fast path planning". In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*. IEEE, pp. 2138–2145.

Kavraki, Lydia E, Petr Svestka, J-C Latombe, and Mark H Overmars (1996). "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE transactions on Robotics and Automation* 12.4, pp. 566–580.

Khansari-Zadeh, Seyed Mohammad and Oussama Khatib (2017). "Learning potential functions from human demonstrations with encapsulated dynamic and compliant behaviors". In: *Autonomous Robots* 41.1, pp. 45–69.

Khattab, Asem (2018). "Static and Dynamic Path Planning Using Incremental Heuristic Search". In: *arXiv preprint arXiv:1804.07276*.

Koenig, Sven and Maxim Likhachev (2002a). "Dˆ* lite". In: *Aaai/iaai* 15.

— (2002b). “Improved fast replanning for robot navigation in unknown terrain”. In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*. Vol. 1. IEEE, pp. 968–975.

— (2002c). “Incremental a”. In: *Advances in neural information processing systems*, pp. 1539–1546.

Komenda, Antonín, Michal Stolba, and Daniel L Kovacs (2016). “The international competition of distributed and multiagent planners (CoDMAP)”. In: *AI Magazine* 37.3, pp. 109–115.

Koren, Yoram and Johann Borenstein (1991). “Potential field methods and their inherent limitations for mobile robot navigation”. In: *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. IEEE, pp. 1398–1404.

Kraft, Aaron R (2017). “Abstraction Hierarchies for Multi-Agent Pathfinding”. In:

Kring, Alexander William, Alex J Champandard, and Nick Samarin (2010). “Dhpa* and shpa*: Efficient hierarchical pathfinding in dynamic and static game worlds”. In: *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Lamiraux, Florent and J-P Lammond (2001). “Smooth motion planning for car-like vehicles”. In: *IEEE Transactions on Robotics and Automation* 17.4, pp. 498–501.

Latombe, Jean-Claude (2012). *Robot motion planning*. Vol. 124. Springer Science & Business Media.

LaValle, Steven M (1998). “Rapidly-exploring random trees: A new tool for path planning”. In:

— (2006). *Planning algorithms*. Cambridge university press.

Li, Jiaoyang, Pavel Surynek, Ariel Felner, and Hang Ma (2019). “Multi-Agent Path Finding for Large Agents”. In: AAAI.

Li, Jiayuan, Ruofei Zhong, Qingwu Hu, and Mingyao Ai (2016). “Feature-based laser scan matching and its application for indoor mapping”. In: *Sensors* 16.8, p. 1265.

Likhachev, Maxim, Geoffrey J Gordon, and Sebastian Thrun (2003). “Ara: formal analysis”. In:

— (2004). “ARA*: Anytime A* with provable bounds on sub-optimality”. In: *Advances in neural information processing systems*, pp. 767–774.

Likhachev, Maxim, David I Ferguson, Geoffrey J Gordon, Anthony Stentz, and Sebastian Thrun (2005). “Anytime Dynamic A*: An Anytime, Replanning Algorithm.” In: *ICAPS*. Vol. 5, pp. 262–271.

Liu, Yunhui, Qi Zou, and Siwei Luo (2011). “GPU Accelerated Fourier Cross Correlation Computation and Its Application in Template Matching”. In: *International Conference on High Performance Networking, Computing and Communication Systems*. Springer, pp. 484–491.

López, Elena, Sergio García, Rafael Barea, Luis M Bergasa, Eduardo J Molinos, Roberto Arroyo, Eduardo Romera, and Samuel Pardo (2017). “A multi-sensorial simultaneous localization and mapping (SLAM) system for low-cost micro aerial vehicles in GPS-denied environments”. In: *Sensors* 17.4, p. 802.

Lucas, François, Christophe Guettier, Patrick Siarry, Anne-Marie Milcent, and Arnaud De La Fortelle (2010). “Constrained navigation with mandatory waypoints in uncertain environment”. In:

McAnlis, Colt and James Stewart (2008). “Intrinsic detail in navigation mesh generation”. In: *AI Game Programming Wisdom* 4, pp. 95–112.

Mei, Yongguo, Yung-Hsiang Lu, Y Charlie Hu, and CS George Lee (2004). “Energy-efficient motion planning for mobile robots”. In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA’04. 2004*. Vol. 5. IEEE, pp. 4344–4349.

Merrill, Duane, Michael Garland, and Andrew Grimshaw (2012). “Scalable GPU graph traversal”. In: *Acm Sigplan Notices*. Vol. 47. 8. ACM, pp. 117–128.

Milford, Michael and Ruth Schulz (2014). “Principles of goal-directed spatial robot navigation in biomimetic models”. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 369.1655, p. 20130484.

Milford, Michael J, Gordon F Wyeth, and David Prasser (2004). “RatSLAM: a hippocampal model for simultaneous localization and mapping”. In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA’04. 2004*. Vol. 1. IEEE, pp. 403–408.

Millington, Ian and John Funge (2009). *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann. ISBN: 978-0-12-374731-0.

Mononen, M (2009). *Recast navigation toolkit*.

Muise, Christian, Nir Lipovetzky, and Miquel Ramirez (2015). “MAP-LAPKT: Omnipotent multi-agent planning via compilation to classical planning”. In: *Competition of Distributed and Multi-Agent Planners (CoDMAP-15)* 14.

Nash, Alex (2010). “Theta*: Any-angle path planning for smoother trajectories in continuous environments”. In: *AI Game Dev*.

Nash, Alex and Sven Koenig (2013). “Any-angle path planning”. In: *AI Magazine* 34.4, pp. 85–107.

Nash, Alex, Sven Koenig, and Maxim Likhachev (2009). “Incremental Phi*: Incremental any-angle path planning on grids”. In: *Twenty-First International Joint Conference on Artificial Intelligence*.

Nash, Alex, Kenny Daniel, Sven Koenig, and Ariel Felner (2007). “Theta^*: Any-angle path planning on grids”. In: *AAAI*. Vol. 7, pp. 1177–1183.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron (2008). “Scalable parallel programming with CUDA”. In: *Queue* 6.2, pp. 40–53.

Noreen, Iram, Amna Khan, and Zulfiqar Habib (2016). “Optimal path planning using RRT* based approaches: a survey and future directions”. In: *Int. J. Adv. Comput. Sci. Appl* 7.11, pp. 97–107.

NVIDIA. CUDA (2017). http://www.nvidia.com/object/cuda_home_new.html.

Oh, Shunhao and Hon Wai Leong (2016). “Strict Theta*: Shorter Motion Path Planning Using Taut Paths.” In: *ICAPS*, pp. 253–257.

O’Keefe, John and Jonathan Dostrovsky (1971). “The hippocampus as a spatial map: preliminary evidence from unit activity in the freely-moving rat.” In: *Brain research*.

Oliva, Ramon and Nuria Pelechano (2013). “NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments”. In: *Computers & Graphics* 37.5, pp. 403–412.

Ortega-Arranz, Hector, Yuri Torres, Diego R Llanos, and Arturo Gonzalez-Escribano (2013). “A new GPU-based approach to the shortest path problem”. In: *2013 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, pp. 505–511.

Othman, Mohd Fauzi, Masoud Samadi, and Mehran Halimi Asl (2013). “Simulation of dynamic path planning for real-time vision-base robots”. In: *FIRA RoboWorld Congress*. Springer, pp. 1–10.

Otte, Michael W (2008). “A survey of machine learning approaches to robotic path-planning”. In: *International Journal of Robotics Research* 5.1, pp. 90–98.

Pallottino, Lucia, Vincenzo G Scordio, Antonio Bicchi, and Emilio Frazzoli (2007). “Decentralized cooperative policy for conflict resolution in multivehicle systems”. In: *IEEE Transactions on Robotics* 23.6, pp. 1170–1183.

Pelechano, Nuria, Jan M Allbeck, and Norman I Badler (2007). "Controlling individual agents in high-density crowd simulation". In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, pp. 99–108.

Pelechano, Nuria and Carlos Fuentes (2016a). "Hierarchical path-finding for Navigation Meshes (HNA*)". In: *Computers & Graphics* 59, pp. 68–78.

— (2016b). "Hierarchical path-finding for Navigation Meshes (HNA$*$)". In: *Computers & Graphics* 59, pp. 68–78.

Pelechano, Nuria, Jan M Allbeck, Mubbasir Kapadia, and Norman I Badler (2016). *Simulating heterogeneous crowds with interactive behaviors*. CRC Press.

Rabin, S. (2000a). "A* Speed Optimizations". In: *Game Programming*, 272–278.

Rabin, Steve (2000b). "A* aesthetic optimizations". In: *Game Programming Gems* 1, p. 600.

— (2014). *AI Game programming wisdom 4*. Vol. 4. Nelson Education.

Rahmani, Vahid and Nuria Pelechano (2017). "Improvements to hierarchical pathfinding for navigation meshes". In: *Proceedings of the Tenth International Conference on Motion in Games*. ACM, p. 8.

— (2020). "Multi-agent parallel hierarchical path finding in navigation meshes (MA-HNA*)". In: *Computers & Graphics* 86, pp. 1–14.

"Recast" (2017). In: https://github.com/recastnavigation/recastnavigation.

Rodrigues, Rafael Araújo, Alessandro de Lima Bicho, Marcelo Paravisi, Cláudio Rosito Jung, Léo Pini Magalhães, and Soraia Raupp Musse (2009). "Tree paths: A new model for steering behaviors". In: *International Workshop on Intelligent Virtual Agents*. Springer, pp. 358–371.

Russell, Stuart J and Peter Norvig (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,

Sacerdoti, Earl D. (1974). "Planning in a hierarchy of abstraction spaces". In: *Artificial Intelligence* 5.2, pp. 115 –135. ISSN: 0004-3702.

Samet, Hanan (1988). "An overview of quadtrees, octrees, and related hierarchical data structures". In: *Theoretical Foundations of Computer Graphics and CAD*. Springer, pp. 51–68.

Sanders, Peter and Dominik Schultes (2005). "Highway hierarchies hasten exact shortest path queries". In: *European Symposium on Algorithms*. Springer, pp. 568–579.

Savkin, Andrey V and Chao Wang (2014). “Seeking a path through the crowd: Robot navigation in unknown dynamic environments with moving obstacles based on an integrated environment representation”. In: *Robotics and Autonomous Systems* 62.10, pp. 1568–1580.

Schulman, John et al. (2014). “Motion planning with sequential convex optimization and convex collision checking”. In: *The International Journal of Robotics Research* 33.9, pp. 1251–1270.

Sharon, Guni, Roni Stern, Ariel Felner, and Nathan R Sturtevant (2015a). “Conflict-based search for optimal multi-agent pathfinding”. In: *Artificial Intelligence* 219, pp. 40–66.

Sharon, Guni, Roni Stern, Ariel Felner, and Nathan R. Sturtevant (2015b). “Conflict-based search for optimal multi-agent pathfinding”. In: *Artificial Intelligence* 219, pp. 40 –66. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2014.11.006. URL: http://www.sciencedirect.com/science/article/pii/S0004370214001386.

Sigurdson, Devon, Vadim Bulitko, William Yeoh, Carlos Hernández, and Sven Koenig (2018). “Multi-Agent Pathfinding with Real-Time Heuristic Search”. In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, pp. 1–8.

Silver, David (2005). “Cooperative Pathfinding.” In: *AIIDE* 1, pp. 117–122.

Šišlák, David, Premysl Volf, and Michal Pechoucek (2009). “Accelerated A* trajectory planning: Grid-based path planning comparison”. In: *Proceedings of the 19th International Conference on Automated Planning & Scheduling (ICAPS)*. Citeseer, pp. 74–81.

Snook, Greg (2000). “Simplified 3D movement and pathfinding using navigation meshes”. In: *Game programming gems* 1.1, pp. 288–304.

Stentz, Anthony (1993). *Optimal and efficient path planning for unknown and dynamic environments*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST.

Sterren, W. van der and Champandard (2008). “Building a near Optimal Navigation Mesh”. In: *Game Programming Games 2*. http://aigamedev.com/premium/masterclass/automatedterrain-analysis/.

Štolba, Michal, Daniel Fišer, and Antonın Komenda (2016). “Potential heuristics for multi-agent planning”. In: *Proceedings of the 26th International Conference on Automated Planning and Scheduling, ICAPS*. Vol. 16, pp. 308–316.

Sturtevant, Nathan and Michael Buro (2005). "Partial pathfinding using map abstraction and refinement". In: *AAAI*. Vol. 5, pp. 1392–1397.

Sturtevant, Nathan and Renee Jansen (2007). "An Analysis of Map-Based Abstraction and Refinement". English. In: *Abstraction, Reformulation, and Approximation*. Ed. by Ian Miguel and Wheeler Ruml. Vol. 4612. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 344–358. ISBN: 978-3-540-73579-3.

Sturtevant, Nathan R (2007). "Memory-Efficient Abstractions for Pathfinding." In: *AIIDE* 684, pp. 31–36.

Sturtevant, Nathan R and Michael Buro (2006). "Improving Collaborative Pathfinding Using Map Abstraction." In: *AIIDE*. Marina del Rey, pp. 80–85.

Sturtevant, Nathan R and Robert Geisberger (2010a). "A comparison of high-level approaches for speeding up pathfinding". In: *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Sturtevant, Nathan R. and Robert Geisberger (2010b). "A Comparison of High-Level Approaches for Speeding Up Pathfinding." In: *AIIDE*. Ed. by G. Michael Youngblood and Vadim Bulitko. The AAAI Press.

Sturtevant, Nathan R, Devon Sigurdson, Bjorn Taylor, and Tim Gibson (2019). "Pathfinding and Abstraction with Dynamic Terrain Costs". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 15. 1, pp. 80–86.

Sud, Avneesh, Erik Andersen, Sean Curtis, Ming Lin, and Dinesh Manocha (2008). "Real-time path planning for virtual agents in dynamic environments". In: *ACM SIGGRAPH 2008 classes*. ACM, p. 55.

Thalmann, Daniel and Soraia Raupp Musse (2013). *Crowd Simulation*. Springer.

Tolman, Edward Chace and Charles H Honzik (1930). "Degrees of hunger, reward and non-reward, and maze learning in rats." In: *University of California Publications in Psychology*.

Tominski, Christian, James Abello, and Heidrun Schumann (2009). "CGV—An interactive graph visualization system". In: *Computers & Graphics* 33.6, pp. 660–678.

Torreño, Alejandro, Eva Onaindia, and Oscar Sapena (2014). "FMAP: Distributed cooperative multi-agent planning". In: *Applied Intelligence* 41.2, pp. 606–626.

Toth, Csaba D, Joseph O'Rourke, and Jacob E Goodman (2017). *Handbook of discrete and computational geometry*. Chapman and Hall/CRC.

Tožička, Jan, Jan Jakbuv, and Antonín Komenda (2014). "Generating multi-agent plans by distributed intersection of finite state machines". In: *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, pp. 1111–1112.

Tozour, Paul (2002). "Ai game programming wisdom". In: *Charles River Media*, pp. 541–547.

Uras, Tansel, Sven Koenig, and Carlos Hernández (2013). "Subgoal graphs for optimal pathfinding in eight-neighbor grids". In: *Twenty-Third International Conference on Automated Planning and Scheduling*.

Van Toll, Wouter, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts (2016). "A comparative study of navigation meshes". In: *Proceedings of the 9th International Conference on Motion in Games*. ACM, pp. 91–100.

Vermette, Jonathan (2011). "A Survey of Path-finding Algorithms Employing Automatic Hierarchical Abstraction". In:

Willms, Allan R and Simon X Yang (2006). "An efficient dynamic system for real-time robot-path planning". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 36.4, pp. 755–766.

Wolfe, R, W Fitzgerald, and Franklin Gracer (1981). "Interactive graphics for volume modeling". In: *18th Design Automation Conference*. IEEE, pp. 463–470.

Yap, Peter (2002). "Grid-based path-finding". In: *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer, pp. 44–55.

Yap, Peter Kai Yue, Neil Burch, Robert C Holte, and Jonathan Schaeffer (2011). "Any-angle path planning for computer games". In: *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Yiu, Ying Fung, Jing Du, and Rabi Mahapatra (2019). "Evolutionary Heuristic A* Search: Pathfinding Algorithm with Self-Designed and Optimized Heuristic Function". In: *International Journal of Semantic Computing* 13.01, pp. 5–23.

Yu, Jingjin and Steven M LaValle (2013). "Structure and intractability of optimal multi-robot path planning on graphs". In: *Twenty-Seventh AAAI Conference on Artificial Intelligence*.

Zhang, Yunong and Jun Wang (2004). "Obstacle avoidance for kinematically redundant manipulators using a dual neural network". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34.1, pp. 752–759.

Zhou, Yichao and Jianyang Zeng (2015). "Massively Parallel A* Search on a GPU." In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence.