Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Hierarchical pathfinding in computer games

Bachelor thesis

2017

Jana Harvanová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Hierarchical pathfinding in computer games

Bachelor thesis

Study program:	Applied Informatics
Field of study:	2511 Applied Informatics
Department:	Department of Applied Informatics
Supervisor:	RNDr. Jozef Šiška, PhD

Bratislava, 2017
Jana Harvanová

13016281

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Jana Harvanová
Study programme:	Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Applied Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Hierarchical pathfinding in computer games

Aim: Compare existing approaches to hierarchical pathfinding in computer games, implement and try to improve one of them.

Annotation: Maps in computer games are often too large to allow simple use of A* which led to development of hierarchical approaches which try to find paths using an abstraction of the original map but in doing so can introduce errors. Various approaches (such as HPA* or Contraction hierarchies) have different properties and are applicable on different problems.

Furthermore the problem is harder in games with non-grid maps, where different approaches to map pre-processing and graph generation further influence the properties of the search algorithms (i.e. Delaunay Triangulation).

Keywords: Hierarchical Pathfinding, Computer games, A*

Supervisor:	RNDr. Jozef Šiška, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	13.10.2016
Approved:	17.10.2016

doc. RNDr. Damas Gruska, PhD.
Guarantor of Study Programme

....
Student

....
Supervisor

Acknowledgements

I wish to express my sincere thanks to my supervisor Jozef Šiška for his invaluable advice and guidance when I was writing this thesis. I am also extremely grateful to my family, friends and my partner for their continuous support.

Abstract

Hierarchical pathfinding algorithms such as HPA* and PRA* are often used in computer games when traditional pathfinding approaches such as Dijkstra's algorithm and A* are unable to find a path in sufficient time. Since videogames evolve fast and so do the requirements for performance and pathfinding times, it is essential that an effective algorithm is capable of fulfilling these requirements. The aim of hierarchical approaches to pathfinding is to organize the search space to smaller parts which can be evaluated and later incorporated in the solution efficiently. We research these algorithms, aim to implement PRA* and HPA*, compare them while also enhancing PRA* to be able to operate on dynamic maps.

Keywords: pathfinding, hierarchy, Dijkstra, A*, PRA*, HPA*

Abstrakt

Hierarchické algoritmy ako HPA* a PRA* sa často používajú v počítačových hrách keď tradičné riešenia ako A* alebo Dijkstrov algoritmus nie sú postačujúce, pretože nedokážu nájsť optimálnu cestu v požadovanom čase. Keďže vývoj počítačových hier ide rýchlo dopredu a s ním aj požiadavky na výkon a nájdenie optimálnych ciest v mape, je nevyhnutné hľadať efektívne algoritmy na naplnenie týchto požiadaviek. Cieľom hierarchických prístupov k hľadaniu ciest v mapách je organizácia prehľadávacieho priestoru na menšie časti, ktoré je možné vyhodnotiť a neskôr efektívne spojiť do jedného riešenia. Naším cieľom je skúmať tieto algoritmy, urobiť implementáciu HPA* a PRA*, porovnať ich a PRA* rozšíriť o schopnosť pracovať nad dynamickými mapami.

Kľúčové slová: prehľadávanie, hierarchia, Dijkstra, A*, PRA*, HPA*

Introduction

Computer games have been around entertaining people for many years now, evolving and changing in many ways. With technology improving constantly, many games today are a big step from what their predecessor were. The most significant change that deserves special mention and is highly relevant for this thesis, is the fact that in today's games, game worlds are often very large, offering a lot of exploration and immersion for the players. It is understandable that such vast landscapes would be rather dull if they were empty. What would be the point, offering a great world for the player to experience, only for them to be all alone in such a world, with no means to interact with it or see signs of life around?

That is the reason behind the existence of Non-Player characters (NPCs).
An NPC is a character that exists in the world, but is under no control of the player. They may offer players insight into the story, information concerning an issue in the game world, give them quests - they serve as means through which the player is able to interact with the game environment.

The main requirement for such NPCs to be believable is to provide them with some kind of intelligence - in many, if not all more complex games, a form of Artificial Intelligence (AI). This means the AI is responsible for generating intelligent behavior, simulating human-like intelligence for the NPCs. This is also largely used in NPCs' actions regarding their movement around the map.
In order to make this achievable for the AI, it has to be given an understanding of the game's environment. As games become more complex and the game worlds vast - as already mentioned above - NPCs should be able to identify fundamental properties of the game map, namely paths the character can walk to reach their destination while avoiding obstacles, or paths that are traversable but not satisfiable for other reasons and should better be avoided.

Let's say that the NPC is a merchant on the road from one city to another. There is also an area along one of the roads, said to have been ambushed by bandits recently. In such a case, it would be wise for the merchant to avoid this place and seek a different path to his next stop.

Many computer games send NPCs from their current location to a predetermined - or sometimes user-determined - location on the game map. The paths that progress to the destination may contain multiple obstacles along the way, which the NPC should avoid - just as a real human being would. A real person would also probably try to reach their destination as fast as possible, taking the shortest possible route leading them to their point of interest.
Therefore, it would be reasonable to expect the NPC to also act in such a way. This presents the issue of determining the most clever and efficient way to avoid obstacles along the way, and determine the shortest - or otherwise most satisfiable - path to the character's destination. The approach used to resolve said issue is called pathfinding, which determines the shortest path from one point to another for a computer-controlled character.

Abbreviations

NPC	**N**on **P**layer **C**haracter
AI	**A**rtificial **I**ntelligence
GBFS	**G**reedy **B**est **F**irst **S**earch
RPG	**R**ole **P**laying **G**ame
PRA*	**P**artial **R**efinement **A***(A-star)
HPA*	**H**ierarchical **P**athfinding **A***(A-star)

Contents

Acknowledgements **i**

Introduction **iv**

Abbreviations **vi**

1 Pathfinding overview **1**

1.1 Dijkstra's Algorithm . . . 1
1.2 Greedy Best-First Search Algorithm . . . 2
1.3 A* algorithm . . . 3
1.4 Search-space representations and Optimization . . . 5
1.4.1 Grid . . . 6
1.4.2 Waypoint Graphs . . . 6
1.4.3 Navigation Meshes . . . 6
1.4.4 Hierarchical Pathfinding introduction and HPA* . . . 6
1.5 Hierarchical Pathfinding approaches overview . . . 7
1.5.1 HPA* . . . 8
1.5.2 Partial Refinement A* . . . 9

2 Problem specification **11**

2.0.1 Algorithm specification . . . 11
2.0.2 Map Specification . . . 11
2.0.3 Map Rules . . . 12
2.0.4 Final product . . . 12

3 Implementation **13**

3.1 A* implementation . . . 13
3.1.1 The open and closed lists . . . 13
3.1.2 Heuristics . . . 14
3.1.3 Manhattan distance . . . 14
3.1.4 Diagonal Distance . . . 15
3.2 PRA* implementation . . . 15
3.2.1 Architecture explanation . . . 15
3.2.2 Abstraction Layers . . . 15
3.2.3 Cluster nodes . . . 17
3.2.4 Heuristic distance . . . 18
3.2.5 Cluster node connections . . . 18

3.2.6 Pathfinding process 20
3.2.7 Path Smoothing 21
3.3 PRA* addition: Dynamic pathfinding 22
3.3.1 Dynamic abstraction structures 22
3.3.2 Start and destination location 23
3.3.3 Non-traversable to traversable 23
3.3.4 Traversable to Non-traversable 24
3.3.5 Path replanning 24
3.4 HPA* implementation 25
3.4.1 Architecture explanation 25
3.4.2 Building clusters 26
3.4.3 Cluster nodes, inter-edges and intra-edges 27
3.4.4 Pathfinding process 29

4 Results 30
4.1 PRA* vs HPA* pathfinding 30
4.2 HPA* vs PRA* abstraction building 32
4.3 HPA* cluster transition positioning 33
4.4 Dynamic map tests 33

Conclusion 33

References 36

Chapter 1

Pathfinding overview

Pathfinding has gained popularity over the years, especially recently as gaming industry is on the rise, gaining more and more attention and popularity from the masses. The most well-known algorithms include Dijkstra's algorithm, which influenced the development of other pathfinding approaches. Many accustomed solutions to pathfinding problems that were widely used - Breadth-First Search, Best-First Search, Depth-First Search and others - decreased in fame, overwhelmed by the increased complexity and performance demanding games which required more efficient solutions on complex environments with limited time and resources.

A-Star (A*) has become one of the most popular algorithms for solving pathfinding problems, gaining a huge amount of attention and success. Because of its success, A* has given foundation significant for other approaches that draw their inspiration from it, modifying A* for their needs of optimisation and improvement, hoping to satisfy the ever-changing demands of the game industry. Multiple optimisations of A* have been successfully introduced over the years. Such optimisations mainly consist of heuristic methods improvements, modification of map representation, introduction of new data structures and memory requirements reduction.

1.1 Dijkstra's Algorithm

Dijkstra's algorithm [20] operates over a weighted graph. Initially, it assigns a distance value to every node of the graph - the starting node to zero, and the rest of the nodes to infinity. It also remembers which graph nodes were already visited and which were not.

The algorithm visits graph nodes one by one, running the search from the object's

starting location. For current node, remembers it as visited. Then the algorithm calculates the distance from the current node plus the weight of the edge between the current node and its closest, non-visited neighbour. If the calculated value for said neighbour is less than its current stored distance, the stored distance is replaced with the newly calculated one. This repeats for all non-visited neighbours of the current node.
Then we move to the closest non-visited neighbor of the current node and repeat the process above. This process repeats for each node, until either all nodes have been visited or the current node is the destination node, in which case the search terminates and then the algorithm back-traces its way to the start, remembering the shortest path.

Dijkstra's algorithm unfailingly finds the shortest path in a connected weighted graph from the object's starting point to the goal, as long as the graph contains no edge with a negative value.

1.2 Greedy Best-First Search Algorithm

The Greedy Best-First-Search algorithm (GBFS) also keeps track of a frontier to locate the target[6]. The way the frontier expands compared to Dijkstra's algorithm, however, is significantly different. Compared to Dijkstra's algorithm, GBFS makes use of a heuristic function.

The heuristic function approximately determines the distance between a particular node and the object's destination. The Dijkstra's algorithm always selects the node which distance is closest to the starting point, but in GBFS, the node closest to the goal is selected and given higher priority than nodes that are farther away from the destination. Subsequently, if the object's destination was to the right of the object's starting point.
Greedy Best-First-Search will try and focus on the path which leads to towards the right, the nodes along this path calculated as the most optimal according to the heuristic. This helps the algorithm to capture the target in its frontier very quickly.
The rest of the procedure is the same where it back-traces the pointers to the parent nodes to formulate the path travelled.

Greedy Best-First-Search runs much faster than Dijkstra's algorithm because it uses the heuristic function to estimate the distance to the goal, which helps it filter its paths to save time and resources. However, this algorithm, unlike Dijkstra's, does not guarantee a shortest path. It will lead the NPC to the destination, avoiding all obstacles, but the path chosen may not be the most optimal one.[6]

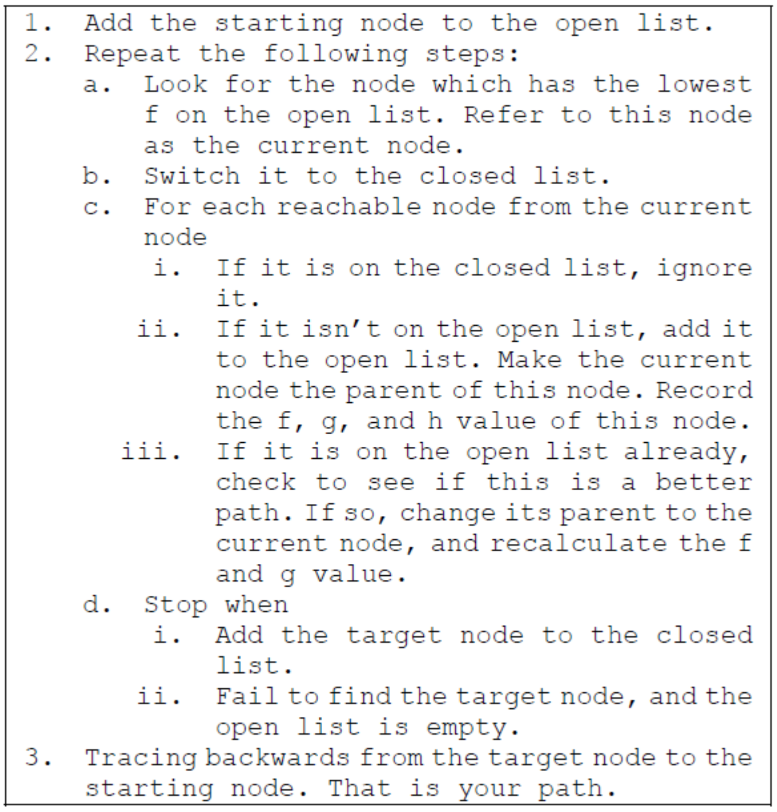
1.3 A* algorithm

The reasoning behind A* being based on Dijkstra's and GBFS is that, like Dijkstra's algorithm, A* always succeeds in finding the shortest path possible on the game map, and like GBFS, it uses a heuristic function to estimate the distance to the object's destination. [6]

It incorporates Dijkstra's exact distance - g(n) - from the starting node to any graph node n and GBFS's heuristic function - h(n) - which evaluates the distance from any node n to the destination node. A* algorithm repeatedly examines the most promising non-visited node it has seen.
When a node is visited, the search terminates if the said visited node is the destination; if it is not, it visits current node's neighbors and explores further.
In other words, the algorithm repeatedly examines the non-visited node n that has the lowest value of f(n), where

$$f(n) = g(n) + h(n)$$

Fig.1 presents a pseudocode of A*, a final summary of how the algorithm works.



```
1.  Add the starting node to the open list.
2.  Repeat the following steps:
    a.  Look for the node which has the lowest
        f on the open list. Refer to this node
        as the current node.
    b.  Switch it to the closed list.
    c.  For each reachable node from the current
        node
         i.  If it is on the closed list, ignore
             it.
        ii.  If it isn't on the open list, add it
             to the open list. Make the current
             node the parent of this node. Record
             the f, g, and h value of this node.
       iii.  If it is on the open list already,
             check to see if this is a better
             path. If so, change its parent to the
             current node, and recalculate the f
             and g value.
    d.  Stop when
         i.  Add the target node to the closed
             list.
        ii.  Fail to find the target node, and the
             open list is empty.
3.  Tracing backwards from the target node to the
    starting node. That is your path.
```

Fig.1 Pseudo-code of A* [6]

The closed list contains nodes that are already resolved. This means that, for each node n in this list, f(n) is equal to the shortest possible path, from the starting point to n and from n to the goal node [10]. The map doesn't contain any shortcut through a different visited node that would influence the g(n) value, nor would expanding other nodes yield a better g(n) score for n. The h(n) value is estimated by the heuristic function and thus doesn't change.
Since the node is resolved, we never expand it again.
Note that the statement of f(n) evaluated as the most optimal path for node n is true only if the heuristic function for computing the h(n) value of the node n is admissible. A heuristic is admissible if it never overestimates the cost of reaching the goal, meaning that the estimated cost of the heuristic to reach the destination is not higher than the lowest possible cost from the current point in the path.[15]
In other words, if the heuristic is not admissible, the path found by A* is not guaranteed to be optimal, but will probably be close.[10]
The open list holds all nodes that are yet to be evaluated. This is the list from which the lowest f(n) node is selected and the list to which neighbors of the current node are added (provided they are traversable and not on the closed list yet).
The nodes contained in the open list can be accessed multiple times - i.e from a different neighbor, where a path through this neighbor could potentially yield a better g(n) value.

Dijkstra's algorithm is very effective in finding the shortest, most optimal path from point A to point B. Its drawbacks lie in search time and resources as it can waste time searching in directions that would never possibly contribute to the best path.
This is especially apparent when working with larger world's maps. GBFS, on the other hand, focuses its search only to directions most likely yielding an effective path, though this approach understandably doesn't guarantee the final path to be truly the shortest, most optimal one. The approach is greedy and consequently, as is known, its heuristic sacrifices accuracy for performance improvement. [6]
The power of A* lies in the fact that it is as fast as GBFS and finds a path as good as Dijkstra's algorithm does, gaining the best from both.

Although A* remains among the most popular choices for computer games' pathfinding, the application of the algorithm largely depends on the nature of the game and how the world is represented internally.
If we had a rectangular gird map of 1000x1000 squares, there are at most 1 million possible squares to search. Finding a path in such a map is not a trivial issue and would take a lot of work. In most cases, it is required to somehow organize the search

space and represent it in a manner that would reduce it, significantly improving search performance. In the next section, we will introduce some of such optimizations.

1.4 Search-space representations and Optimization

As we mentioned above, it is important to implement the search space representation in an effective manner, which mostly attributes to data structures used, minor optimizations in some cases, or using heuristics. This representation of the game's environment is mandatory for the NPCs as any path they plan, they plan based on this information[18].
Choosing the most appropriate data structure influences the pathfinding performance and speed the most. If the search space is simple and well organized while keeping all crucial game world information, the algorithm is going to perform faster and won't need to put in as much work in order to calculate the optimal path.
The next page contains multiple examples of map representations, namely Figure 2.a shows an example map, Figure 2.b shows the map's gird decomposition Figure 2.c shows a waypoint graph on the map, and Figure 2.d shows a triangle decomposition (a type of navigation mesh).

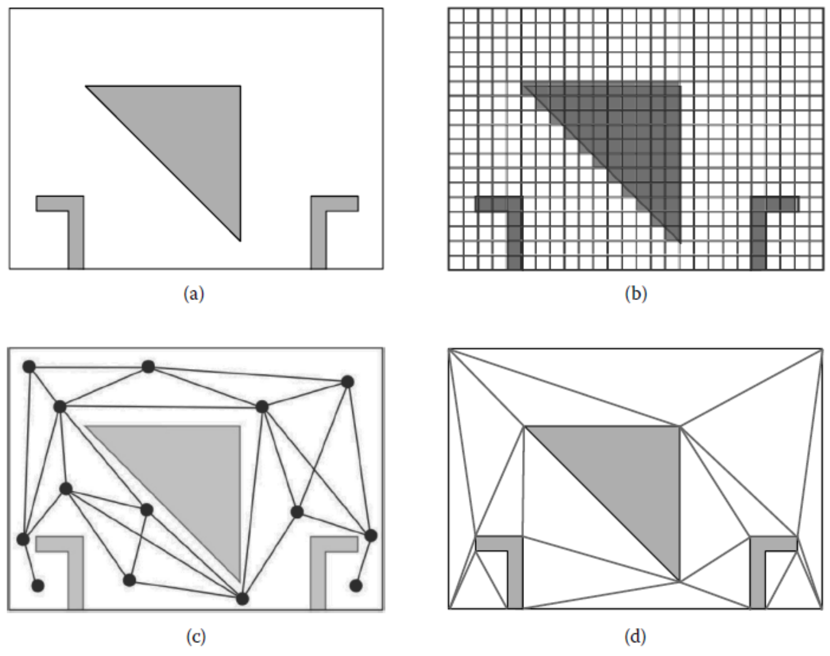


Fig.2 Three common world representations. a) Original map, b) grid decomposition, c) waypoint graph, d) nav mesh [18].

1.4.1 Grid

A very trivial and simple to implement world map representation is a Grid, internally consisting of an array of cells. Depending on the sophistication of the implementation, these cells can either be blocked or unblocked or contain other information, i.e a value indicating the travel effectiveness (possibility representing slopes, hills, woods, type of terrains in general) or other information useful for planning.

1.4.2 Waypoint Graphs

In this representation, the world is portrayed as an abstract graph. An important property of waypoint graph's to note is that there are no explicit mapping between the graph's nodes and the space that is traversable. Waypoint graphs were widely used before the growth in popularity of navigation meshes.
While they have been criticized for their shortcomings, they have also been praised for their strengths [18].

1.4.3 Navigation Meshes

Similar to Waypoint Graphs, navigation meshes represent the world using way-polygons instead of waypoints. Way-polygons define walkable areas where the characters can move, other areas do not need to be considered when computing pathfinding solutions. Because of the way polygons are defined, they can span through a wider map area while potentially using less nodes to do so. Consequently, potentially increasing search performance and requiring less memory to store information regarding these nodes.

Constrained Delaunay triangulations are a special case of navigation meshes [5], the world being represented by triangles in this case. Grids can also be seen as a special case of a navigation mesh, since both use convex polygons. However, their usage is, in practice, significantly different.

1.4.4 Hierarchical Pathfinding introduction and HPA*

Despite the speed and superior performance properties of A*, in some cases even these are insufficient. Hierarchical pathfinding is one of the most widely considered possible solution when it comes to searching optimal paths in enormous search space. The main concept of hierarchical pathfinding is dividing the search space into multiple smaller parts, called clusters or sub-grids. An abstraction layer is built on the map in this way, each cluster maintaining its nodes and connections to other, neighboring

clusters.
Hierarchical path-finding A* is said to be over 10 times faster than the low- level A* algorithm[3].

HPA* is only one of many hierarchical approaches to pathfinding - other hierarchical approaches use slightly different implementations to reach their desired performance (multiple abstraction layers, heuristics, search space compression and so on). In the next section, we will in detail explain the concept of Hierarchical pathfinding in general, introduce multiple algorithms used in hierarchical pathfinding and mention other thesis and publications which are related to the final hierarchical algorithm we are most interested in.

1.5 Hierarchical Pathfinding approaches overview

As we briefly mentioned before, the concept of Hierarchical pathfinding is that the search space can be divided into at least two levels on which the search occurs: one is a high-level, abstract representation of zones (clusters) containing a sub-set of the search space's cells, and a low-level representation of all cells with all information initially provided.

The path is first constructed in less detail on the high-level representation from the cluster containing the start to the cluster containing the end goal (think of rooms in a castle, starting in the foyer and finding a path, from room to room, to the balcony). When this is finished, the search begins on the low-level, finding a path from the starting cluster to the next cluster to be reached (from the standing place in the foyer to the first room on the path). When the second cluster is reached, another low-level, detailed path is formed from this cluster to the third, and so on, right until the last cluster on the path is entered and the goal reached.

We have two concrete examples of games that have used hierarchical pathfinding on their maps, namely the famous RPG Dragon Age: Origins [19] developed by BioWare^{TM} and Company of Heroes [9] developed by Relic Entertainment$^{TM.}$

In Dragon Age: Origins' case, the final architecture used two levels of grid-based abstraction above the low-level grid. In case of Company of Heroes, the high-level search space representation was a hex-grid and the low-level representation was a regular square grid. [14]

Hierarchical abstraction has potential to speed up a search by replacing it with a series of smaller searches. Abstract solutions computed in a higher-level state space representation are gradually refined, possibly through several hierarchical levels, down to the ground level of the original search space.[7]

1.5.1 HPA*

Hierarchical pathfinding A* is an enhanced implementation of A* that uses an abstract representation of the search space in order to increase the speed of the pathfinding. The process of HPA* is in detail described below.

Let's say we have a grid of cells. All cells are the same size and each cell is considered a node. Every cell also has defined edges to its neighbouring cells in the grid - there is an edge between two neighboring cells if it is possible to get from one cell to the other. In other words, the cells must both be traversable or fulfilling other possible requirements for them to be reachable (if the map contained terrain information like chasms, hills and cliffs, the cell on the cliff at a substantial height logically couldn't be linked with its neighbouring cells even if they all were traversable, for instance). [3]

The next step is to divide the grid defined above into smaller clusters. It is mandatory to represent connections between the now divided clusters, since the newly created clusters are disconnected and missing information from the former grid map. Therefore, the entrance between two adjacent clusters is defined as the maximum free space available on the two clusters' bordering cells.

The next step is to construct the Abstract graph from the clusters defined above. We make use of transitions for this purpose. The number of transitions created per entrance depends on the entrance's width. If the width is smaller than a pre-defined constant, only one transition is created approximately in the middle of the entrance. If the width is larger, two transitions are created on each side of the entrance [3]. We define two nodes per transition, each of which is connected via an edge. This edge serves as a connection between the two nodes. If these two nodes are both present in the same cluster, the edge is called an intraedge. If these nodes are not present in the same cluster (meaning each is in a different cluster) the edge between them is called interedge.[3]

The building of the Abstraction graph is finished, thus we insert the starting position S and the destination position D to the corresponding nodes represented in the graph. A* is then used to find the path between the start S and destination D. The pathfinding traces a path from S to the border of the cluster where S is present, then plans an abstract path to the border of the cluster where D is located, and finally searches the path to D in the cluster.

The abstraction in searching the path this way results in even fewer nodes and space search (since the AI operating on such an Abstract graph knows exactly which cluster to look to, avoiding the irrelevant ones for the selected destination)[3] and, consequently and understandably, Hierarchical pathfinding A* is much faster in its approach to appropriate pathfinding and also lowers the memory requirements compared to A*.

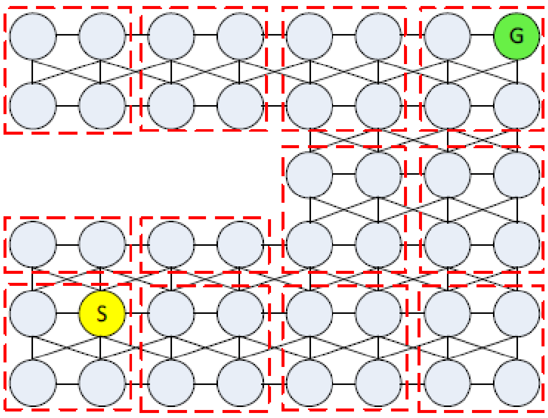
1.5.2 Partial Refinement A*

Partial Refinement A* was first introduced by Nathan Sturtevant in his work [16], where he explains the theory behind the algorithm. The main concept of building map abstractions can be derived from a theoretical definition of a search problem itself.
The usual representation of a search problem is a tuple, $\{S; s; G; O; H\}$, where S is the set of all states, s corresponds to a start state, G holds a set of goal states or represents a goal test function, O is a set of operators, and H is a heuristic function.
A search problem can be represented as a graph whose nodes are states of the problem and edges are operators. An abstract graph would be a reduction of such graph. Each node of the abstract graph's nodes would represent one or more states (nodes) in the lower level graph. An edge connecting two nodes in an abstract graph exists if an operator from O could be applied to any state abstracted by the first node that would lead to a state abstracted by the second node.
The abstractable search problem is defined as a tuple $\{S^0; s; G; O; H^k; A\}$[16]. In an abstractable search problem, a state is associated with an abstraction level, so S^k is a set of nodes abstracted on the level k. Aside from S^0 which has to be defined as a separate sub-problem, all higher levels of abstraction can be defined by an abstraction function $A(s_1^k...s_i^k)$. This function takes i states at the abstraction level k and assigns them to a single node on the abstraction level $k+1$. Each node is abstracted exactly once. Also, the heuristic function must be enhanced to take two states and an abstraction level k, returning the estimated distance of the two states on level k.
Partial Refinement A* (PRA*) is a combination of hierarchical pathfinding and fast partial path refinement, which effectively interleaves the planning of the path with path execution. This is relevant mostly to robotics and videogames [16].

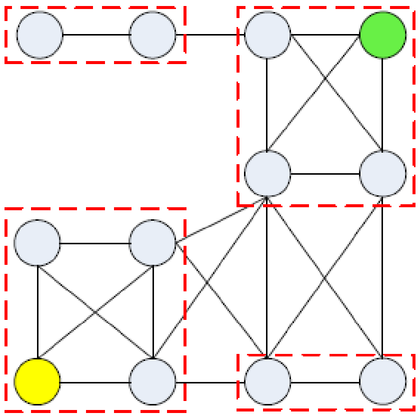
To abstract the base navgraph, PRA* uses a "clique and orphans" based abstraction [1]. A clique is defined as a set of nodes wherein each node is both reachable and at most a single step away from every other node in the set (i.e. each node has an edge to every other node in the set).
PRA* initially builds an abstract hierarchical map representation from the bottom, clustering small connected regions to tiles on another level, preserving connectivity between them. Through this greedy abstraction process, coarser and more unrefined pyramids of map representation are created, up until the moment where only one single node for every connected area of the map remains.
These nodes represent the original connected clusters. In order to find paths according to time and quality constraints, an abstraction layer is chosen to look for abstraction graph's nodes that represent the start and goal on the original map.

The search starts at this level, where PRA*(k) first uses A* to determine the shortest path and in the next step projects first k steps down to the next level of the abstraction of the map. Here, the shortest path is computed by considering only a small corridor around the path determined in the higher level. This continues until the goal node is reached.

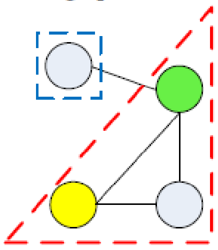
High-quality initial paths on the original map are quickly found through this process, allowing NPCs (or other desired game's objects) to start moving in the right direction quickly.
Cliques are shown in Figure 3 on this page, and are outlined in red. An orphan is a node that is reachable from only a single other node (the blue outlined node in Figure 3.c). PRA* uses a maximum clique size of 4 in abstracting the navgraph.



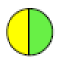
a) The first level of abstraction is created by dividing a navgraph into 4-cliques.



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

Fig. 3 the abstraction of PRA* [1] Later, Sturtevant [19] described a hybrid approach that combines ideas of HPA* and PRA* to decrease memory used for abstraction.

Chapter 2

Problem specification

The goal of this thesis is to make a successful implementation of multiple (at least two) hierarchical approaches, run performance tests of these algorithms on various game maps and present the results.

The last part of this work was to introduce an improvement to one of the implemented algorithms and include the description and details of said improvement.
All pathfinding is additionally visualized in a GUI mode, as well as a simulation of an agent moving along the determined path.

2.0.1 Algorithm specification

The majority of our focus went to PRA* and, to a lesser extend, Hierarchical Pathfinding A* (HPA*). Both of these algorithms were already mentioned and briefly introduced in the first chapter of this work.
We have also implemented A* as a base algorithm, since both PRA* and HPA* use A* and also for performance comparison and testing.

2.0.2 Map Specification

Each map is equivalent to a weighted graph on which the pathfinding from point A to point B is carried out. The map format consists of a square (n x n) or rectangular (n x m) grid layout. The grid consists of blocks, where each block is a square and represents a node in such graph.
A node can represent one of the following states:

Traversable - a terrain through which the agent is able to move

Non-traversable - node is blocked and thus is not accessible by the agent

Start position - specifies the starting point of the agent.

End position - the destination the path should reach - if a path exists.

The starting position is the place from which the pathfinding starts planning the path. Every map can only have a single Start and End position.

2.0.3 Map Rules

Horizontal, vertical and diagonal movements are all allowed on the map. This implies that each grid cell (node) of the grid map has at most 8 neighbors: 4 perpendicular and 4 diagonal.
The distance from a node to a neighbor is dependant on whenever the neighbor is reached by a diagonal or a perpendicular (vertical/horizontal) movement. When we think of a node as a square of 1 x 1 size, if the neighbor is a perpendicular neighbor, the distance is 1. If the neighbor is diagonal, the distance equals to $\sqrt{2}$, where we store the approximated value of 1.4 for simplicity.

2.0.4 Final product

The final product is a GUI application which is capable of loading maps, enables the user to edit them and run multiple search algorithms, namely PRA*, HPA*, A* and Dijkstra. The search visualizes the grid nodes visited, the path computed and runs a simulation of a moving agent from the starting position to the end, along the path, after the search completes.
Another information such as number of nodes expanded, run time (in ms) of the search and the length of the path, are also visible.
Some of the maps come with selectable pre-defined test cases which test -and visualize - the dynamic PRA* path planning.

Chapter 3

Implementation

In this chapter we describe our implementation of A*, HPA* and PRA* as well as the dynamic planning with PRA*.
All algorithms were programmed in C# using .NET 4.5 and WinForms for the GUI.

3.1 A* implementation

We have introduced A* in the first chapter of this work. We know that the algorithm preserves an open list and closed list of nodes, as we can see on Fig. 1.
In this section and following subsection, we address parts of the implementation that influence the performance of A*. We consider this relevant information mainly because both PRA* and HPA* use A* as a part of their search. Therefore, the following implementation influences their performance as well, as will be discussed in their respective sections.

3.1.1 The open and closed lists

The structures chosen for the open list and the closed list have the most influence over the final performance of A*. Since each time, until either the open list is empty or the goal node is reached, we: 1) get the node n with lowest f(n) value and 2) for this node, check if it is present in closed/open lists. We also need to store the g and f values, respectively.
In our A* implementation, we have chosen bool[] array for the closed list and a dictionary for the open lists.
Since our map representation assigns a unique ID to each grid node, values ranging from 0 to $n * m - 1$ (if the map size is n x m grid cells), we can effectively initialize the bool[]

array of the required size of n^2 or n x m, where the starting values are false.
Then, if a node n is added to the open list or the closed list, we simply change the value of $closedList[n.ID] = true$. When checking if a node n is present in a closed list, we index the bool[] array and get the value, in $O(1)$ time. If we need to check if a node is present in the open list, we also make a lookup in constant O(1) time.
Similarly, we used Dictionaries for storing the g and f values of all nodes, where the dictionary key corresponds to the node's unique ID, therefore storing, updating and getting these values for a node n also takes $O(1)$ time.
Getting the node with the lowest f cost itself is done by sorting the dictionary key-value pair by value in descending order and getting the corresponding key of the first pair.

3.1.2 Heuristics

The importance of a heuristic used in A* being admissible was already mentioned in the above subsection. There are various heuristic functions used for A* pathfinding, a good overview and explanation of them presented on this web page [12] hosted by the Computer Science Department of the University of Stanford. For our pathfinding implementation, we chose the Manhattan Distance heuristic and the Diagonal Distance heuristic. Even though Manhattan heuristic is mainly used - and recommended - for 4-way pathfinders (maps where diagonal movement is not possible) and, therefore, in our case can be inadmissible, we decided to use it anyway.
First reason being that, even though inadmissible, it will almost always give the shortest path, and when it doesn't it won't be far off.[10]
The second reason is that since Diagonal Distance is the heuristic most recommended for 8-way pathfinders such as the one we implemented, we can probably see the difference in path quality in some cases and, therefore, see the importance of heuristic admissibility for ourselves.

3.1.3 Manhattan distance

The Manhattan Distance heuristic has the primary advantage of getting to the destination faster than most other alternatives. The value of H can be computed as follows:
$H = D * (abs(currentX - targetX) + abs(currentY - targetY))$

where D is the set distance between two adjacent grid nodes, which in our case equals 1.

3.1.4 Diagonal Distance

The diagonal distance heuristic is one of the most suitable heuristics for grid maps that allow perpendicular as well as diagonal movement. This heuristic estimates how many steps would be required to take if diagonal movement wasn't possible and then subtracts the steps that could be saved by using the diagonal movement. The number of diagonal steps is min(dx, dy), where the cost of each is D2 but saves 2D non-diagonal steps.

$dx = abs(node.x - goal.x)$
$dy = abs(node.y - goal.y)$
$H = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)$

where D represents the value of a diagonal step between two grid nodes, while $D2$, same as in the Manhattan heuristic, is the distance between two adjacent grid nodes.

3.2 PRA* implementation

Similarly to the previous section, in this section we introduce our PRA* implementation. We describe the architecture, how we dynamically build the hierarchy, determine cliques, build cluster nodes, connections between them and the PRA* pathfinding process itself. The definition of abstraction problem has had a significant influence on our implementation of PRA* abstraction layers, clusters and cluster connections.
We discuss this in detail in the next subsections.

3.2.1 Architecture explanation

Our implementation of the PRA* abstraction hierarchy was influenced by the abstraction problem definition that is mentioned and further described in Chapter 1 based on [16]. Below, we apply the theory to practical implementation and provide an explanation of the structure building in greater detail than the original paper did.

3.2.2 Abstraction Layers

The structure holding the most information is an *abstraction level*, also called *abstraction layer* in the terminology used in our implementation.
An *abstraction layer* holds a unique ID which represents the position of the layer in the hierarchy, much like the level k does in the Abstractable search problem. Aside from

the ID, the layer holds a collection (dictionary) of nodes called *cluster nodes*, where each such node on this abstraction is made of i nodes from the lower level abstraction (similarly to the abstraction function $A(s_1^k...s_i^k)$, if this layer would have an ID equal to $k+1$). The important requirement for the $s_1^k...s_i^k$ nodes from the lower level is, in the case of PRA*, that the nodes need to form a clique. As mentioned in chapter 1, a clique is defined as a set of nodes wherein each node is both reachable and at most a single step away from every other node in the set. We do not take into account the value of the distance itself - it is only important that there is an edge from each node to the other nodes in the clique. The fact that the value of a diagonal movement is slightly larger than the value of a perpendicular movement is irrelevant in this case.
We start building the first abstraction layer after the map has loaded. The first layer is built by mapping the grid nodes of the map itself, forming cliques of them. Every time we make a clique, we create a new cluster node and add it to the abstraction layer's cluster node collection. This cluster node is an abstraction of the clique and holds a list of IDs of the grid nodes. We refer to this list as *innerNodes* of a cluster node. The cluster node is also assigned a unique ID by the abstraction layer and for each node from *innerNodes* we assign this ID as an identifier of a cluster to which the grid node belongs. This will be important later in building the cluster connections, in the pathfinding process and in the dynamic properties of PRA* discussed later in this work.
Since we need to abstract each grid node from the grid map exactly once, we maintain a set called *resolvedNodes* during the building process of the abstraction layer. Each time we make a cluster node, we add all the cluster node's *innerNodes* to the *resolvedNodes* list. If a grid node is contained in the *resolvedNodes* list, it cannot belong to another clique and, therefore, to another cluster node. This assures that each cluster node's *innerNodes* are unique from the other cluster node's *innerNodes* and that each grid node is abstracted exactly once.
The layer continues building grid node cliques represented as cluster nodes until all grid nodes are resolved. When all the nodes are resolved, the abstraction layer holds the complete abstract representation of all the map nodes from the lower level.

Higher abstraction layers are built in a similar fashion to the first abstraction layer described above, with a few following differences:
Instead of looking at the grid nodes when making cliques, the new higher layer k looks at the cluster nodes of the layer *k-1.* A set of cluster nodes make a clique if for each cluster node n in this set, we could reach from this cluster node's *innerNodes* to the other cluster nodes' *innerNodes* in a single step.
When we identify a set of cluster nodes $s_1^{k-1}...s_i^{k-1}$ as a clique, we create a new cluster node, assign a unique ID to it and add it to the abstraction layer's cluster node collection. The *innerNodes* of this cluster node are the IDs of the lower level cluster

nodes $s_1^{k-1}...s_i^{k-1}$. Similarly as in the first level abstraction, for each cluster node from *innerNodes* we assign the new cluster node's ID as an identifier of a cluster to which the cluster node from the lower level belongs.
The *resolvedNodes* list has the same purpose as in the building of the first layer, except for the fact that instead of adding grid node's IDs we populate the list with the IDs of the cluster nodes from the $k-1^{th}$ level.
When all nodes are resolved, there are a few conditions that influence the next action. These conditions are:
1) If the current abstraction layer contains only a single cluster node, this means that the map has been as abstracted as is possible. The building of the abstraction hierarchy is complete.
2) If the current abstraction layer contains n cluster nodes, but all these clusters are disconnected, this means that the map has contains n disjoint areas that are as abstracted as possible. The building of the abstraction hierarchy is complete.
3) If the current abstraction layer contains n cluster nodes and at least two of them have an edge connecting them, the map can be abstracted further. We start building a higher level of abstraction.

Therefore, the building of the abstraction hierarchy is finished when either a single cluster node remains, or when multiple disconnected cluster nodes remain, which is correct according to the algorithm explanation in Chapter 1.

3.2.3 Cluster nodes

The previous section has included information regarding the creation of cluster nodes and adding them to the abstraction layer. However, there are questions that remain unanswered at this point, such as how to create connections between cluster nodes effectively, or how we determine a distance from one cluster node to another. We address these questions and more, in detail in the following sections.

In the previous section we have also mentioned that the cluster node has the list of *innerNodes* and that it holds the ID of its parent cluster node on the higher layer - provided a higher layer exists in the abstraction.
The cluster node naturally also preserves the connections to its neighbors - cluster nodes belonging to the same layer of abstraction. This structure is a dictionary, where the keys are the cluster node IDs and the values the respective cluster node neighbors. The cluster node preserves another dictionary which holds the estimated heuristic distances to the neighboring cluster nodes, the key being the cluster node ID and the value the heuristic distance to the cluster node.

3.2.4 Heuristic distance

Every time a cluster node is created, we calculate the position of the cluster node in the abstract space relative to the map.
We achieve this by making a sum of all nodes' X and Y coordinates in the *innerNodes* list and dividing them by the number of nodes, getting the average values of the position.
This is trivial if the cluster node belongs to the first level of abstraction, since in that case the *innerNodes* correspond directly to the grid nodes of the map which posseses concrete X/Y coordinates.
If the abstraction layer is higher in the hierarchy, the X/Y coordinates of the layer's cluster nodes can be built the same way - making a sum of all cluster nodes' positions in the *innerNodes* list and getting the average value. The distance between two cluster nodes is heuristic. We use the heuristic defined in [16] :
$\sqrt{2} * min(|\Delta x|, |\Delta y|) + ||\Delta x| - |\Delta y||$

3.2.5 Cluster node connections

Another issue which arises on building the abstraction hierarchy is effectively creating edges between the cluster nodes on each of the abstraction levels. Since the cluster nodes cannot be checked for neighbors directly like the grid nodes on the base map can, we need a different approach.
One possible solution, which is also the most simple, is loping through all cluster nodes on a layer, then looping through all cluster nodes in an inner loop and checking if nodes from their *innerNodes* are each others neighbors. This process would work, but there are obvious undeniable issues with this solution.
Looping through all the cluster nodes in the nested way mentioned above by itself would take O(n^2) where n is the number of cluster nodes on the layer. We have used this solution in the early stages of our PRA* implementation for the sake of simplicity. Building the abstraction hierarchy on a 512 x 512 grid map took several minutes, which is unacceptable if we want to implement an efficient dynamical maps and dynamic planning with PRA* as well.
For the reasons explained above, we have developed a different, effective solution which we will introduce below.

The algorithm starts by looping through the layer's cluster nodes.

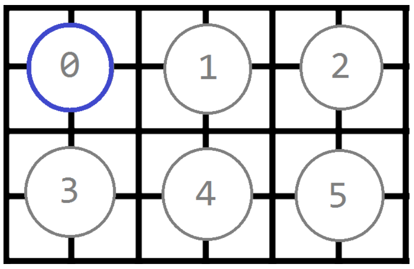


Fig.4 An empty grid map with six 4-clique cluster nodes.

Fig. 4 shows a simple 6 x 4 grid map of which all grid nodes are traversable. The map is abstracted as is shown - we see the first abstraction layer hosting six cluster nodes with assigned IDs from 0 to 5.

Each cluster node overlaps 4 grid nodes, which makes each of these 4 grid nodes a 4-clique and also the *innerNodes* of the cluster node. The cluster node with ID 0 is outlined in blue and is the node currently visited in the loop.

Since we have direct access to all the unique grid node IDs in *innerNodes* of this cluster node, we can look directly to these grid nodes and get their neighboring nodes. We get only the neighbors that do not belong to the *innerNodes*. The next image shows these neighbors of cluster node 0 in light-blue color.

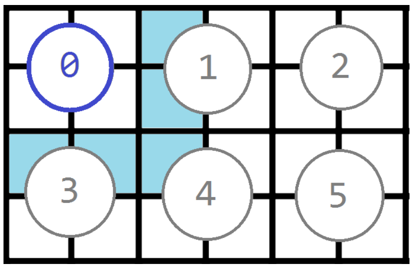


Fig.4.2 The neighbors of the *innerNodes* of the cluster node 0.

The property that each grid node (or cluster node) remembers its parent cluster node becomes an important part of this solution now, as we mentioned in the section 3.2.1.

At this step, we initialize an empty set which will hold all the neighbors after the following step.

Now that we have the IDs of the neighboring nodes, we loop through all of these neighbors. We get the corresponding grid node from the ID and *get the parent cluster ID from it.* We add the ID to the set and continue until the last neighbor reports their parent cluster's ID.

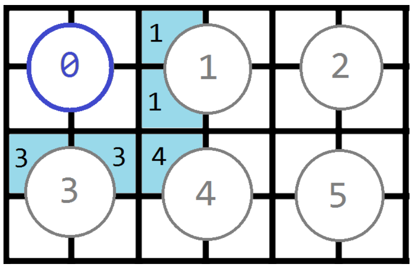


Fig.4.3 The neighbors of the *innerNodes* of the cluster node 0.
At the end of the loop the set of the soon-to-be neighbors contains the values $\{1, 3, 4\}$.
At last, we assign the cluster nodes as each others neighbors an we are done.

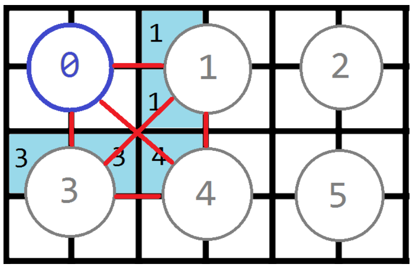


Fig.4.3 The connected neighboring cluster nodes

The exact same algorithm can be applied on any layer of abstraction - the only difference being that, if on a higher layer of abstraction, when looking to the *innerNodes* of a cluster we look to the cluster nodes one layer down, not to the gird nodes directly. Since every cluster node remembers its parent cluster if there is a higher abstraction layer, the algorithm will work the same way.
The algorithm performs only one loop over all the cluster nodes instead of a going through a nested loop, which is $O(n)$. The time required for the *innerNodes* to report their neighbors is $8m$ where m is the size of the clique (or the *innerNodes*), the maximum value of m is equal to 4. The loop through the final neighbors is $O(k)$, $k =$ the number of neighbors (maximum of 8) and adding the neighbors is done in a constant time $O(1)$. The final complexity of the algorithm is $0(n)$, which has brought drastic improvements in the abstraction hierarchy building performance.

3.2.6 Pathfinding process

The theory behind how the pathfinding works, starting on a layer k of the abstraction and then refining the path until the last level, is already mentioned and explained in 1.5.2 , therefore we proceed to the implementation details.
The starting layer we chose is somewhere around the middle of all the abstraction layers, namely $k = m/2$ where m is the number of layers in the hierarchy.

Before we start the pathfinding, we first check whenever a path exists between the set starting point and destination point on the grid map. We can do this easily by getting the cluster nodes of the highest layer in the hierarchy that is the recursive parent to the starting grid node and the ending grid node.
We know the cluster parent of a grid node directly, as we have mentioned in 3.2.2 above. We can then recursively ask the cluster node to return its parent cluster node until we either get the parent on the k^{th} level we wanted to, or when the cluster node has no parent, which means that no higher abstraction layer exists.
We can get the cluster node parent of a grid node on the highest layer in $O(m)$ time, where m is the number of abstraction layers in the hierarchy. If the cluster nodes do not match, at this point we know that there is no way reaching the end position from the start position on the map. Running the search for the path wouldn't yield a solution.
When the cluster node parent of the start node and the end node match on the highest abstraction layer, there is a path connecting the start and end nodes and we can start the pathfinding.
We start a slightly modified A* search on the abstraction layer k. The A* function's arguments now contain a layer k, a set of nodes which represents the search space, a cluster node containing the start position and a cluster node containing the end position. Both of these cluster nodes are nodes from the layer k. Each cluster node of the set also belongs to the k^{th} layer.
When we first build an abstract path, the set of nodes supplied to A* contains all the cluster nodes of the k^{th} level of abstraction. After an abstract path is built, we proceed to refine the path on the $k-1^{th}$ layer. Refining a path means that instead of running the search on the lower level through all its cluster nodes representing the search space, we only search the ones that are the *innerNodes* of the higher abstract path computed, reducing the search space on each layer. We repeat this process until the first layer of abstraction is reached. When we have the abstract path on the first level, the only remaining lower level is the grid map itself. There we compute our final path on the grid base from the abstract path. We achieve this by running the slightly modified A* from one cluster node to the next in the abstract path until the last cluster node is reached. When the last cluster node is reached, we plan the last part of the path to the end node and return.

3.2.7 Path Smoothing

When the path was refined to the lowest layer of abstraction and we started planning the final path from cluster node to cluster node through the underlying grid nodes, we have sometimes gotten paths that contained creases and small diagonal turns

where a straight line would be optimal. This has degraded the path quality, especially in relation to its length. It was a result of a somewhat greedy approach to passing from a cluster node to another in the modified A* search. The determining condition on reaching the next cluster node was only that if *any* neighboring grid node of the current grid node belonged to the next cluster node, we automatically return the part of the path and started the search to the next cluster node in the abstract path.
We have no guarantee that the first neighbor grid node we evaluate as belonging to the next cluster node is also the closest to the end cluster node.
Instead of greedily returning path parts as soon as the next cluster node is entered, we compute all the neighbor's f(n) values and add to the path the one with the lowest f(n) value. This has increased the PRA* path quality by 5-8% in our tests, which will be reflected upon later in the Results.

3.3 PRA* addition: Dynamic pathfinding

There were multiple challenges in coming up with a solution to implementing the dynamic version of PRA*, especially concerning the speed and effectiveness of the solution when applied to larger game maps.
In order to achieve the real-time, dynamic re-planning of the path on the map change, we need to know how to modify the abstraction sructure to correctly reflect the change. We address this problem in the next section.

3.3.1 Dynamic abstraction structures

On every change in the map, we needed to modify the abstraction to correctly reflect this change in its structure. We define a change in a map as a grid node's transition from one state to another. The change can occur for one or multiple grid nodes at a specified time.
The only possible grid node state transitions are the following:

1. a *traversable* grid node becomes the *start* position
2. a *traversable* grid node becomes the *end* position
3. a *non-traversable* grid node becomes *traversable*
4. a *traversable* grid node becomes *non-traversable*

Each transition is handled by manually changing the first layer of abstraction to reflect the change and then rebuilding the higher layers.

Since the first abstraction layer contains the largest amount of cluster nodes of all the layers in the hierarchy, applying the changes here manually without rebuilding the layer saves us time - especially on the larger maps. However, manually changing the higher layers as well proved to be more complex and introduced errors in the hierarchy. Therefore, in our implementation we decided to rebuild them entirely.
We now expand on the specific transitions and how are they handled when a changed grid node g is being resolved.

3.3.2 Start and destination location

There is no impact on the abstraction structures directly in this transition. The only thing we need to do in this case is update the PRA* cluster node parent property of g. We do this by looping through all the cluster nodes on the lowest layer of abstraction. When we reach a cluster node of which *innerNodes* contain the ID of g, we set this cluster node's ID as the parent of g. We do this in $O(n)$ time, where n is the number of cluster nodes on the lowest layer of abstraction.

3.3.3 Non-traversable to traversable

When g becomes traversable, we need to incorporate it to the abstraction layer. We start by getting all its the cluster node neighbors. If there are no neighbors, we create a new orphan node and create an orphan, disconnected node on each layer of abstraction until the last one. This is one of the exceptional cases where no recomputing of the abstraction structure is needed.
Otherwise, if neighbors exist, then:
1) if there is an orphan cluster node among the neighbors, we can trivially assign the node to this cluster node's *innerNodes* and the orphan becomes a 2-clique. We also check if adding this grid node to the cluster node would create new connection(s) between the modified cluster node and the neighboring cluster nodes of g. If it does, we create the edges between g and each of them.
2) If the neighbors do not contain an orphan, we look if any of the neighbors are a 2-clique or 3-clique. If there are, we check if adding g would make the clique a valid $k+1$ clique. We do this by checking if all the *innerNodes* of the clique are neighbors of g. If they are, the clique would be a valid $k+1$ clique, we can assign g to *innerNodes* and check for any new neighbors similar to 1).
3) If no suitable 2-cliques nor 3-clique exist in the neighboring cluster nodes of g, we create a new orphan node and connect it to all neighboring cluster nodes.

In each case we also recalculate the relative position of the cluster node from the positions of its *innerNodes* and assign the PRA* cluster node parent of g.

3.3.4 Traversable to Non-traversable

When g becomes npn-traversable, we need to remove it from the abstraction layer and resolve its past neighbors accordingly.
1) If g was part of a 4-clique, trivially the clique stays valid as a 3-clique without g.
2) If g was part of a 3-clique or a 2-clique, trivially the clique stays valid - since diagonal movement is valid as long as the two nodes are traversable in the case of the 3-clique and a 2-clique becomes an orphan node.
3) If g was an orphan, the node is deleted. We remove the node form all its previous neighbors.
In all cases, we recalculate the relative position of the cluster node from the positions of its *innerNodes* and assign the PRA* cluster node parent of g. We also check all the cluster's neighbors if they still remain neighbors after this change. If no connections were cut off, the rebuilding of the higher levels is unnecessary. Otherwise, we delete the connections on the first abstraction layer and proceed to rebuild the higher levels.

3.3.5 Path replanning

We test the dynamic path replanning of PRA* by making selectable premade test cases assigned to specific maps. Each test case contains:

- the start and end points on the map
- a set *nodeChanges* of grid nodes with assigned states
- a parameter *trigger* (an integer > 0) when the map change should occur

After a path is built by PRA* and the agent starts moving along the path, we start a counter with 0 as its initial value. The counter increases each time the agent moves from one grid node to another. When the counter is equal to *trigger*, each grid node from *nodeChanges* is set to the state it was assigned in the test.
This means that a state transition occurs for each grid node from *nodeChanges* as described in 3.3.1 and the lowest layer of abstraction is changed accordingly.
Note that since on large maps the rebuilding of the abstraction hierarchy could take some time (i.e multiple seconds), it would visibly influence the agent which would freeze in place, wait for the abstraction to rebuild, and only then start the replanning of the

path. The replannning would take additional time and make the agent frozen in place even longer.
In order to prevent this, we create a new abstraction hierarchy in an asynchronous thread while the agent is running along the old path in its own thread.
The new abstraction first copies the lowest abstraction layer from the old hierarchy. Then it applies all the transitions from *nodeChanges* modifying this layer *only*, since rebuilding all the hierarchy after each transition would be time consuming, resource consuming and unnecessary.
After all the *nodeChanges* were performed, only then we start the building of the higher layers of the hierarchy. When the new abstraction hierarchy is finished building, the thread notifies the agent to stop moving. When it does, we replace the old hierarchical structure with the new one and run PRA* search on the modified map, where the start position of the search is the agent's position.
There are two possible behaviors for the agent in this case:

- a new path was found and the agent starts moving along this path
- the end position is now unreachable with the map changed, which causes the agent to stop and report that no path exists.

3.4 HPA* implementation

We have introduced how HPA* works in the first chapter, namely in the subsections 1.4.4 and 1.5.2. The following subsections address the architecture, the building process of the clusters, an efficient solution to creating the cluster nodes that connect the clusters and the pathfinding process itself.

3.4.1 Architecture explanation

There are multiple variables to consider when choosing the value of n - the size of the clusters - especially the map size, as pathfinding on a very huge map might not benefit from the speed up of the HPA* abstraction if the value of n is too low. In that case, the abstraction layer would contain many small clusters and, therefore, more cluster nodes representing the search space.
We have chosen a value of $n = 10$ for our HPA* implementation and testing purposes. Compared to PRA*, our HPA* implementation consists of only a single layer of abstraction over the grid map. The abstraction layer holds all information about the clusters it holds. Each cluster has a unique ID assigned by the abstraction layer and keeps a list

of all the grid node IDs it abstracts - similarly to the *innerNodes* of a cluster node in the PRA* abstraction. For consistency, we will refer to this list of the HPA* cluster's grid nodes as *innerNodes*, too. Similarly to PRA*, each node from the *innerNodes* of a cluster also remembers the ID of the HPA* parent cluster it is abstracted to.
Aside from the *innerNodes*, a cluster also keeps track of all the cluster *nodes* that belong to it. This will be important when creating intra-edges of a cluster later in this chapter, as well as during the pathfinding process.
In order to traverse the abstract graph directly and avoiding the nested structure of looking to clusters and then into their cluster nodes, we make the abstraction layer keep a reference to each cluster node created. This allows the abstraction layer to traverse the graph directly and, when needed, directly look up the cluster to which the cluster node belongs (by getting the cluster node's HPA* parent cluster ID).
In the next section, we explain how we build the clusters and the cluster nodes connecting them.

3.4.2 Building clusters

After loading a map, we start the cluster building process.
We iterate the grid nodes in two nested loops, skipping by 10 grid nodes in each, starting in the upper left corner of the loaded map. We traverse the whole map in 10 x 10 areas, each time creating a cluster. If the map's size is not divisible by 10, we encounter areas of smaller size that do not fit the 10 x 10 size, in which case we get the remaining area and create a smaller cluster. Aside from this occurrence, we periodically create 10 x 10 clusters until all grid nodes are all assigned to a cluster.

When we create a cluster, we assign its *innerNodes* and set this cluster's ID as a HPA* parent cluster to all grid nodes from the *innerNodes*. Each cluster also keeps a structure of *outerNodes*, which is a subset of *innerNodes* such that the grid nodes are located at the borders of the cluster. The *outerNodes* is a Dictionary in structure, preserving a set of grid nodes located on a specific border of the cluster, where the set is the value and key is an identifier of the border.
The direction keys identifying the location of the cluster border are following:

U - upper border of the cluster

D - lower border of the cluster

L - left border of the cluster

R - right border of the cluster

After we finish building the clusters, we look at each cluster's *outerNodes* to all the directions. Since all the sets under the dictionary's direction keys are the unique IDs of the grid nodes from the map, we can directly look up their horizontal neighbors (in the case of the left or right border) or vertical neighbors (in the case of the upper or lower border) and get their HPA* cluster parent ID which was assigned during the building of the clusters.
If the horizontal or vertical neighbors from the *outerNodes* under a specific direction keys don't exist, the cluster doesn't have a neighbor cluster in the said direction. The set under the direction key is empty.
The *outerNodes* also play an important role in creating the cluster nodes on the two borders when tracking entrances connecting two clusters, which we will explain below in the next section.

3.4.3 Cluster nodes, inter-edges and intra-edges

We have explained the terminology of *inter-edges* and *intra-edges* in the first chapter under section 1.6.2 which addresses HPA*.
The last step of fully building the abstraction layer is creating the entrance points between clusters (cluster nodes), then creating edges between the entrances within the same cluster (intra-edges) and the edges connecting two different clusters (inter-edges).
For each cluster c_1, we look at its *outerNodes*. In each direction - if the set under this direction is not empty - we get the corresponding neighbor cluster c_2 and create a (possibly empty) set of entrances connecting c_1 and c_2 in the specified direction.
An entrance is a list of adjacent grid node pairs g_1 and g_2, where the following conditions are met:

- $g_1 \in c_1 \land g_2 \in c_2$
- both *g1* and *g2* are traversable

We start tracking an entrance by going through each grid node g_1 in the *outerNodes* on the border of *c1*. If both g_1 and the adjacent node g_2 from the border of c_2 are traversable, we add the pair $\{g_1, g_2\}$ to the currently tracking entrance.
We continue tracking an entrance this way until either 1) the whole border was examined, or 2) if one (or both) of the grid nodes g_1 and g_2 are non-traversable.
In the case of 1), we finished tracking all the entrances and can proceed to determining the exact places where cluster nodes connecting c_1 and c_2 will be created.
In the case of 2), we finish tracking the entrance and continue checking the border. When we encounter two traversable adjacent grid nodes again, we start tracking an entrance

again and repeat the process above until 1) occurs and the process terminates.
Then, for each entrance from the set we've created we determine the specific positions of the cluster nodes connecting the two clusters in relation to the size of each entrance. In the end, each entrance will be represented by *at least* one pair of cluster nodes, each assigned the cluster ID it belongs to on the abstraction layer and the grid node ID it represents on the grid map level. We also create an *interedge* connecting the two cluster nodes. The value of the edge is 1, since only perpendicular cluster transitions are allowed, not diagonal.
The position and number of cluster node pairs created in relation to entrance size are described below:

- if the entrance length is less than 5, we take one pair positioned on the index *entrance.Length/2* and create two cluster nodes.
- if the entrance length is at least 5 but less than 7, we take two pairs positioned on the start and end of the entrance and create two cluster nodes for each.
- if the entrance is at least 7 and a maximum of 10, we take two pairs - one on the 3rd position from the start and one on the 3rd position from the end. We create two cluster nodes for each.

In Fig. 5 we show a simple 20 x 15 map, consisting of traversable (white), and non-traversable (black) grid nodes with a set start position(red) and end position(yellow). The map is divided into 4 clusters visualized by blue lines, the red perpendicular lines indicating the transitions between clusters.

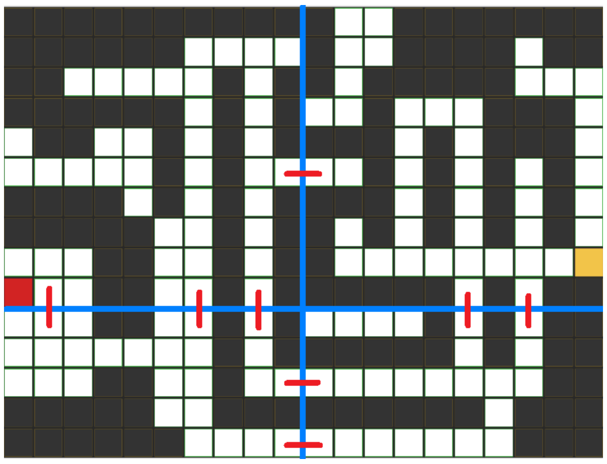


Fig.5 The first process of HPA* map abstraction

By this point, we are finished with building the clusters, cluster nodes and inter-edges. The only remaining part is building the intra-edges within each cluster. Since each cluster knows which cluster nodes belong to it, we simply loop through all the clusters and run A* from each cluster node to the others contained in the cluster. If a path exists between two cluster nodes, we create an edge with the value computed by A*. After we compute the intra-edges for all clusters, the abstraction process is complete. We present the final abstract graph from the map in Fig. 5.1

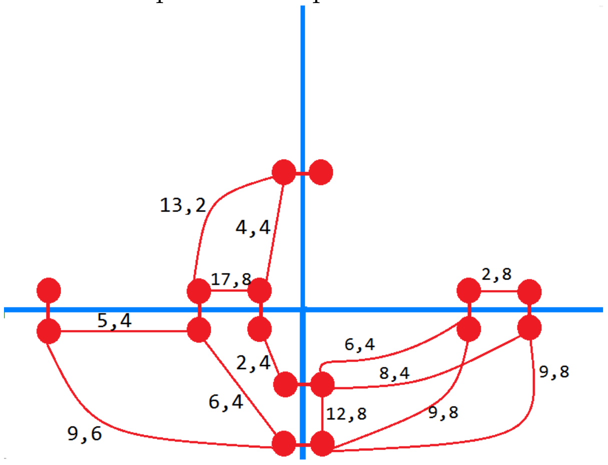


Fig.5.1 The final abstract graph of the map

3.4.4 Pathfinding process

At the start of HPA* pathfinding, we need to incorporate the starting and ending grid nodes into the abstraction hierarchy.
If one or both the start/end grid nodes are the same as a cluster nodes already created, we simply set the cluster node as start/end point in the abstract graph. If it doesn't, we create a temporary cluster node with the grid node ID and get the cluster to which it belongs. We then create all intra-edges from the temporary node to all others in the cluster, connecting it to the abstract graph.
When we have the set start and end cluster nods, we start the A* search on the abstract graph. The A* search function is supplied the start cluster node and the end cluster as arguments. Then a search over the abstract graph is performed, starting at the start cluster node, building the path. When the end cluster is reached, we add the end grid node's ID to the solution and start tracing the path to the start.
After we have built the abstract path, we plan the path on the grid map itself, from cluster node to cluster node, building the final path from the partial solutions.
When the pathfinding is finished, we check if we created a temporary start or end cluster nodes before the search. If we did, we remove them from the abstract graph.

Chapter 4

Results

In this chapter and the following sections, we present the results of our algorithms. All performance tests were done on a laptop with Intel Core i7 4510U 2.1 GHz CPU, 8 GB RAM and the NVIDIA GeForce 840m GPU. Most of the maps on which we performed the tests were downloaded from [13], free for public use and performance testing for developers. Some of the maps were also made by us.

4.1 PRA* vs HPA* pathfinding

This subsection presents the results of the PRA* and HPA* pathfinding. In the following table we show multiple properties of the tests, including map name, the cost of all the algorithm's paths as well as their length (i.e from how many grid nodes the path consists of), times it took to find the paths, the number of nodes expanded in the search and finally, the PQD (Path Quality Degradation) which states to what extend is the path sub-optimal in % A* is always 0% since it succesfully finds the shortest path in each case.

Map	Search	Path Cost	Path length	Time	Expanded	PQD
map02	A*	18	17	4,0254 ms	89	0%
map02	HPA*	18,8	17	5,6288 ms	69	4,4%
map02	PRA*	20,4	22	2,7439 ms	67	13.3%
map03	A*	24,2	22	9,3139 ms	58	0%
map03	HPA*	24,8	23	14,8724 ms	52	2.5 %
map03	PRA*	24,8	23	18,6546 ms	84	2.5 %

Map	Search	Path Cost	Path length	Time	Expanded	PQD
map04	A*	54,2	48	17,2556 ms	946	0%
map04	HPA*	55	49	13,1284 ms	285	1.5%
map04	PRA*	59	22	19,8424 ms	246	8.8%
map05	A*	61,8	56	43,5634 ms	1740	0%
map05	HPA*	65,4	58	17,9081 ms	363	5.8%
map05	PRA*	68	61	15,8277 ms	265	10%
map06	A*	100,4	91	13,4797 ms	527	0%
map06	HPA*	104,4	95	20,6548 ms	363	4%
map06	PRA*	103,8	94	19,7729 ms	359	3.3%
map07	A*	132,4	119	151,0827 ms	4957	0%
map07	HPA*	143,4	128	24,6924 ms	845	8.3%
map07	PRA*	146,4	127	20,1156 ms	549	10.5%
brc101d	A*	597	518	988,1166 ms	25229	0%
brc101d	HPA*	614,6	542	68,1375 ms	4516	3.9%
brc101d	PRA*	682	571	25,8942 ms	2534	11%
brc503d	A*	466,4	399	5038,5642 ms	53950	0%
brc503d	HPA*	479	420	220,5746 ms	3654	2.7%
brc503d	PRA*	518,4	449	43,6796 ms	1992	11%
dwg	A*	497	416	16660,3938 ms	90849	0%
dwg	HPA*	511	441	552,3995 ms	3882	2.8%
dwg	PRA*	552,4	479	54,107 ms	2233	11%
ATC	A*	1244	1053	128559,9131	345259	0%
ATC	HPA*	1281	1108	2653,3554	10113	3%
ATC	PRA*	1445,4	1260	76,3346	5603	16.2%

Maps from map01 to map07 are custom made while brc101d, brc503d, dwg (drywatergulch) and ATC(Across the Cape) are real maps from Dragon Age^{TM}by BioWare, Warcraft III^{TM}and StarCraft^{TM}by Blizzard Entertainment, respectively.

It is apparent that PRA* is by far the fastest algorithm, even compared to HPA* on huge maps. on the other hand, the paths calculated by HPA* are closer to the optimal A* solution, the path degradation very rarely passing 5% and usually staying in the 2-3% range in sub-optimality.

PRA*, though fast, is more susceptible to generating sub-optimal paths, usually between 8-9% and about 16-17% in the worst case. Even in the worst case during the stress test on the ATC 768x768 map however, we can see that PRA* is extremely fast, even compared to HPA*.

4.2 HPA* vs PRA* abstraction building

In the case of HPA*, the search space of A* is defined as each cluster's *innerNodes*, which is at most 100 grid nodes. Despite the small search space for each cluster intra-edge computations, the time to build the HPA* abstraction hierarchy is considerably slower than the PRA* abstraction hierarchy build, especially on larger maps. We present the average build times of both PRA* and HPA* for respective map sizes and also the size of their traversable space. We used the same maps as above, plus additional ones.

Map name	Map size	Traversable space	PRA*	HPA*
map02	20x15	68,6 %	12,2332 ms	45,0591 ms
map03	20x15	75,3 %	13,9237 ms	49,9802 ms
map04	50x50	54 %	18,1181 ms	248,5478 ms
map05	50x50	62,7 %	19,3124 ms	204,4688 ms
map06	50x50	57,88 %	10,7545 ms	181,1809 ms
custom_maze	60x60	55,11 %	11,1528 ms	246,2223 ms
den204d	65x65	65,5 %	16,4566 ms	327,3745 ms
map07	100x100	53,5 %	30,4077 ms	762,2582 ms
map08	100x100	35,6 %	18,9704 ms	766,1990 ms
hrt001	105x110	31,8 %	25,4476 ms	1536,153 ms
map09	200x200	68,8 %	235,4628 ms	2711,9479 ms
custom_maze2	200x200	61,8 %	194,592 ms	2611,7036 ms
brc503d	320x257	81,9 %	568,5586 ms	5809,432 ms
brc101d	640x380	12,2 %	352,5577 ms	42302,855 ms
drywatergulch	512x512	52 %	1121,1691 ms	28462,0827 ms
AcrossTheCape	768x768	66,5 %	3426,1156 ms	49429,3321 ms

While it is apparent that PRA* performs significantly faster than HPA* when it comes to building the abstraction hierarchy, the main advantage of PRA* is that it creates abstraction layers only from traversable nodes. HPA* can sometimes create clusters made entirely by non-traversable nodes, slowing down the process. We can also see this by comparing the PRA* and HPA* results in maps with low percentage of traversable space.

4.3 HPA* cluster transition positioning

Most implementations we've seen described in other works used only one predefined constant, where if an entrance size was lower than this number, only one transition was created in the middle. Otherwise, two were created, one on each end of the entrance.
However, on large maps with many clusters consisting entirely of traversable nodes and therefore having large 10 sized entrances, the transitions were located mainly around the corners of all clusters. We thought that, if the transitions were distributed more evenly, the path might be slightly more optimal. That's why we introduced two constants - 5 and 7 - in our solution which aim to place the transitions accordingly.
Aside from a nicer HPA* cluster/transition visualization, tests have shown that the path cost difference was minimal (ranging from 0 to 3 points in path quality).

4.4 Dynamic map tests

We have run several test cases on various maps.
On smaller maps, the rebuilding of the abstraction hierarchy was faster than the movement speed of the agent, so at the *trigger* step in the test, the abstraction was rebuild and the path recalculated without the agent having enough time to make a step further along the old path. The agent moves from one grid node to another on the path every 500 ms. However, on larger maps, mainly *drywatergulch* from *WarCraft III* and *Accross the cape* from *StarCraft*, the simulation was more interesting.
On the former, the test run made a vertical wall of blocks that blocked the previously calculated path. The hierarchy was rebuild in 1408,1178 ms time in the background, during which the agent took 2 steps in the simulation along the old path. The new path was calculated in 45,1275 ms time, so the fact the agent was stopped during this time couldn't be seen.
On the latter, the test created a traversable corridor through a wall of non-traversable nodes. The hierarchy was rebuild in 3858,2258 ms time and the agent took 7 steps on the old path in the simulation. The new path was calculated in 78,3716 ms time, so as in the previous case, the time the agent was idle couldn't be seen.

Conclusion

We have successfully implemented two algorithms used for hierarchical pathfinding, HPA* and PRA*, the latter also enhanced with a dynamic version of replanning paths when the map is changed.
We have measured their performance and concluded their advantages and drawbacks. The dynamic PRA* successfully rebuilds the hierarchical structures in the background while the moving agent simulation runs along the old path, stopping and replanning the path only after the rebuilding of the structures is finished.

Future Work

In this section we briefly mention several issues and current drawbacks in the implementation that could be addressed and improved in the future.

A* performance

The A* implementation wasn't the most effective one possible as well, though this wasn't our main focus. Though always finding optimal paths, our A* also expanded more nodes than was necessary. This was due to always making an in-place sort of the open list when selecting the lowest f(n) node and getting the first node. However, if there are multiple nodes with the lowest f(n) value, the most recently added would be preferred. Getting the first one from the sorted list doesn't guarantee this, leading to expanding the nodes in directions that might be unnecessary. If a Priority Queue or a Heap was used for the open list, this problem would be fixed and we would also gain an improvement in performance, which would positively influence HPA* and PRA* as well, making them even faster.

PRA* path quality

According to [16], the PRA* should perform better when it comes to how close to optimal the paths should be. Given sufficient time, our PRA* could be enhanced and improved to produce more optimal paths. While refining the path down the levels, instead of searching the *innerNodes* only, maybe checking their neighbors could generate more optimal paths, if for a slight performance drop.

HPA* improvements

There are multiple possible improvements for HPA*, such as:
1) **Cluster nodes -** The HPA* abstraction layer could be improved such that instead of the two cluster nodes that create an inter-edge between clusters, only a single node would be created - with additional information such as having both cluster IDs it connects. Implementing this would reduce the number of HPA* cluster nodes by half, speeding the search on the abstraction layer, especially on larger maps.
2) **Multi-level abstraction -** We could also make HPA* hierarchy contain more abstract levels, speeding the search even more.
3) **Dynamic pathfinding -** Dynamic HPA* could be implemented, where we would be creating or deleting the cluster nodes depending on the changes in the map. We would also need to recalculate the value of intra-edges in some cases when the inner part of a cluster changes.

Closure

Though there is still potential to make many improvements as we have disclosed above, we have acquired a great deal of new theoretical knowledge regarding these algorithms. Through applying the knowledge to make a functional implementation of the algorithms, as well as the visualization and dynamic rebulding of the abstraction hierarchy in asynchronous threads, we gained invaluable practical experience as well.
To our knowledge, this is the first thesis that in detail explains the implementation of PRA*, both on static and dynamic maps. While both have been mentioned in existing papers, the static PRA* was mentioned only in theory and didn't disclose information how to approach issues associated with the implementation (i.e efficiently making connections between cluster nodes on an abstraction level). PRA* capable of operating on dynamic maps was mentioned, but no further information was published.
That is the main contribution of this thesis.

References

[1] Bobby Anguelov. “Video Game Pathfinding and Improvements to Discrete Search on Grid-based Maps”. In: (2011). URL: http://cirg.cs.up.ac.za/thesis/anguelov.pdf.

[2] Amitha Arun. “Pathfinding Algorithms in Multi-Agent Systems”. In: (2014). URL: https://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc15/01Amitha/thesis.pdf.

[3] Adi Botea, Martin Muller, and Jonathan Schaeffer. “Near Optimal Hierarchical Path-Finding”. In: *Journal of Game Development, Volume 1* (2004). URL: https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf.

[4] Adi Botea et al. “Pathfinding in Games”. In: (2013). URL: http://drops.dagstuhl.de/opus/volltexte/2013/4333/pdf/4.pdf.

[5] K. Chen. “Robust Dynamic Constrained Delaunay Triangulation for Pathfinding.” In: (2009).

[6] Xiao Cui and Hao Shi. “A*-based Pathfinding in Modern Computer Games”. In: *IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1* (2011). URL: http://paper.ijcsns.org/07_book/201101/20110119.pdf.

[7] Alex Kring, Alex J. Champandard, and Nick Samarin. “DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds”. In: (2010). URL: http://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/viewFile/2131/2543.

[8] Marc Lanctot, Nicolas Ng Man Sun, and Clark Verbrugge. “Path-finding for Large Scale Multiplayer Computer Games”. In: (2006). URL: http://www.mlanctot.info/files/papers/pathfinding-orbius-2006.pdf.

[9] William Lee and Ramon Lawrence. “Fast Grid-based Path Finding for Video Games”. In: (2013). URL: https://people.ok.ubc.ca/rlawrenc/research/Papers/dba.pdf.

[10] Patric Lester. “Heuristics and A* pathfinding”. In: (2008). URL: http://www.policyalmanac.org/games/heuristics.htm.

[11] Parth Mehta et al. “A Review on Algorithms for Pathfinding in Computer Games”. In: (2016). URL: https://www.researchgate.net/publication/303369993_A_Review_on_Algorithms_for_Pathfinding_in_Computer_Games.

[12] Amit Patel. “Heuristics”. In: (2009). URL: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[13] “Pathfinding Benchmarks”. In: (). URL: http://movingai.com/benchmarks/.

[14] Steve Rabin and Nathan R. Sturtevant. “Pathfinding Architecture Optimizations”. In: (2013). URL: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter17_Pathfinding_Architecture_Optimizations.pdf.

[15] P. Russell S.J.; Norvig. *Artificial Intelligence: A Modern Approach*. 2002.

[16] Nathan Sturtevant and Michael Buro. “Partial Pathfinding Using Map Abstraction and Refinement”. In: (2005). URL: https://www.societyofrobots.com/robottheory/Partial_Pathfinding_Using_Map_Abstraction_and_Refinement.pdf.

[17] Nathan R. Sturtevant. “Benchmarks for Grid-Based Pathfinding”. In: (2012). URL: https://pdfs.semanticscholar.org/82ca/e9f97b7dfa0c244452065916cbc0348438a7.pdf.

[18] Nathan R. Sturtevant. “Choosing a Search Space Representation”. In: (2013). URL: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter18_Choosing_a_Search_Space_Representation.pdf.

[19] Nathan R. Sturtevant. “Memory-Efficient Abstractions for Pathfinding”. In: (2007). URL: https://webdocs.cs.ualberta.ca/~nathanst/papers/mmabstraction.pdf.

[20] Ronald L. Rivest Thomas H. Cormen Charles E. Leiserson and Clifford Stein. “Introduction to Algorithms, Third Edition”. In: *MIT Press* (1990).