Parallel multi-agent path planning in dynamic environments for real-time applications

Alexander Dooms
Parallel multi-agent path planning in dynamic environments for real-time applications

Alexander Dooms

Promotor: prof. dr. ir. Rik Van de Walle


Masterproef ingediend tot het behalen van de academische graad van Master in de ingenieurswetenschappen: computerwetenschappen
Preface

In the first place I want to thank Professor R. Van de Walle for making it possible to do this research for my thesis. In addition, I want to thank my mentors Aljosha Demeulemeester and Jonas El Sayeh Khalil for their assistance, whether it be technical, providing ideas, or pushing me to my limits.

Next I thank my friends that made these five years a fantastic experience. They provided me with valuable feedback on every project, allowed me to discover new technologies, to participate in events, to work together on multiple projects, and simply made my life as a student better than I could imagine.

I thank my parents, for giving me the opportunity to study what I always dreamt of, to support me through the whole experience, and helping me whenever I needed their help. I also thank my brother and sisters, who made sure that living in Ghent was a joy, that I did not lack company, and that I could always count on them.

Last, but certainly not least, I want to thank my partner, Alicia, for standing beside me, for her love, and for the fantastic moments we had and certainly will have in the future. Without her I would not have been where I am now.

Alexander Dooms, June 2013
Permission for usage

“The author gives permission to make this thesis available for consultation and allows parts of the thesis to be copied for personal use. Any other use is subject to the restrictions of copyright, in particular the requirement to clearly mention the source when citing results from this thesis.”

“De auteur geeft de toelating deze scriptie voor consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie.”

Alexander Dooms, june 2013
Parallel Multi-agent Path Planning in Dynamic Environments for Real-time Applications

by

Alexander Dooms

Thesis submitted to obtain the degree of Master of Engineering: Computer Science

Academic Year 2012-2013

University Ghent
Faculty of Engineering and Architecture
Department of Electronics and Information Systems
Head of department: prof. dr. ir. J. Van Campenhout
Promotor: prof. dr. ir. R. Van de Walle
Thesis monitors: Aljosha Demeulemeester, Jonas El Sayeh Khalil

Abstract

Current pathplanning algorithms are not efficient enough to provide optimal pathplanning in dynamic environments for a large number of agents in real time. Furthermore, there are no real-time algorithms that fully use the potential of parallelism. The goal of this thesis is to find a basis for such an algorithm. Based on the literature study, an algorithm is proposed which is based on a Quadtree roadmap. The Quadtree can rapidly be adapted to a dynamic environment, requiring only a couple of nodes to be re-evaluated. In addition, the pathplanning algorithm using the Quadtree takes full advantage of the fact that all nodes (and the zones of the environment they determine) are in the free space. Any path that stays within the zones is a valid path, and determining the optimal path is reduced to calculating with coordinates. A couple of optimizations have also been investigated, of which two stand out in particular. The principal shortcomings of the solution using Quadtrees is that we use a regular grid, with the links having a cost of 1 (or $\sqrt{2}$ for diagonal links), and that although each agent can search his path fully independently of the other agents, there is no parallelism during path search. Even with these shortcomings, results show a great improvement in execution times compared to A*.

Keywords: Global pathplanning, Parallel pathplanning, Quadtree
Parallel multi-agent path planning in dynamic environments for real-time applications

Alexander Dooms

Supervisor(s): prof. dr. ir. Rik Van de Walle, Aljosha Demeulemeester, Jonas El Sayeh Khalil

Abstract—Current pathplanning algorithms are not efficient enough to provide optimal pathplanning in dynamic environments for a large number of agents in real time. Furthermore, there are no real-time algorithms that fully use the potential of parallelism. The goal of this thesis is to find a basis for such an algorithm. Based on the literature study, an algorithm is proposed which is based on a Quadtree roadmap. The Quadtree can rapidly be adapted to a dynamic environment, requiring only a couple of nodes to be re-evaluated. In addition, the pathplanning algorithm using the Quadtree takes full advantage of the fact that all nodes (and the zones of the environment they determine) are in the free space. Any path that stays within the zones is a valid path, and determining the optimal path is reduced to calculating with coordinates. A couple of optimizations have also been investigated, of which two stand out in particular. The principal shortcomings of the solution using Quadtrees is that we use a regular grid, with the links having a cost of $1$ (or $\sqrt{2}$ for diagonal links), and that although each agent can search his path fully independently of the other agents, there is no parallelism during path search. Even with these shortcomings, results show a great improvement in execution times compared to A*.

Keywords—Global pathplanning, Parallel pathplanning, Quadtree

I. INTRODUCTION

PATHPLANNING has been a long-standing problem which has an impact in several areas, such as computer games, robotics and simulations. In these applications the real-time behaviour is an important limitation on the computation time. In these kind of applications we require at least 30 frames per second, which, after displaying the image, only leaves a couple of milliseconds for other calculations such as pathplanning.

The emergence of dual- and quad-core computers has created the possibility for parallel solutions to this problem. However, to this day there are no parallel pathplanners, only pathfinders. The difference is that a pathplanner finds an optimal path between two points, while a pathfinder finds a path as fast as possible, regardless of whether there exists a faster, shorter or better path.

The purpose of this thesis is to design a pathplanner which uses the accessible dual- and quad-core CPUs. By using parallelism, it is possible to carry out much more work within the same timespan.

II. ROADMAPS

A. Roadmaps in general

Roadmaps are graphs which are an abstraction of the virtual environment. The goal of a roadmap is generally to reduce the search space. When using a roadmap, the pathfinding consists of finding a path between its two nodes that are closest to the starting and goal points. The search algorithm hence searches far fewer nodes. Another advantage of a roadmap is that all agents can use the same roadmap once it is constructed.

A disadvantage of a roadmap is that in general there is no specification about how the roadmap should react to changes in the environment. The biggest disadvantage, especially in the context of this thesis, is that paths found using a roadmap are sub-optimal. The reason lies in the fact that the found path uses the nodes of the roadmap, which often forms a detour. Connecting the nodes at the two ends of the path to the start and goal points creates an additional detour.

B. Quadtree

The algorithm developed in this thesis uses, despite the disadvantages highlighted above, a roadmap. We use a Quadtree[1] as roadmap, of which an example is given in Figure 1.

A Quadtree consists of Quadnodes, which determine a square zone. To construct a Quadtree, we start of by considering the whole environment as a single Quadnode. At each step of the construction, a Quadnode in the Quadtree is split into four equally sized squares. If a Quadnode consists solely of free or blocked nodes, then it is finished, and hence will not be split again. If it contains only of free nodes it is added to the Quadtree. In the other case it is removed.

Each Quadnode, as a result of the construction algorithm, consists only of free nodes. The advantage of this is that each path within a Quadnode is a correct path. The same is true for paths which end in neighbouring Quadnodes, if we consider direct paths, i.e., paths that connect two nodes by first moving in a straight line, and then diagonally (see Figure 3). This property comes from the fact that Quadnodes determine square zones which only contain free nodes.

Fig. 1. A Quadtree

Fig. 2. An improved Quadtree

Fig. 3. Some direct paths
III. PATHPLANNER

A. A* pathplanner

The most widely known pathplanner is A*[2]. It is a search algorithm which finds an optimal path between two nodes of a graph. A* is a greedy algorithm, but it makes intelligent choices using a heuristic. The heuristic is a problem-specific function which gives a conservative lower bound of the cost from a node to the goal node. If it gives any other value than a lower bound, the A* algorithm becomes a pathfinder. In the context of pathplanning, an often used heuristic is the euclidean distance, because the distance between two nodes can never be smaller than a straight line.

B. Quad*

Quad* is the algorithm developed in this thesis. It is a variant of A* which uses the properties of a Quadtree. The difference with A* is that Quad* uses direct paths to nodes adjacent to the current Quadnode. This is correct, because of the properties of direct paths in Quadtrees. In addition, there cannot be a shorter path than a direct path.

The result is that Quad* searches far less nodes than A*, because it “jumps” over the zones of the Quadnodes. A small example is given in Figure 4. It is clear to see the difference in the number of searched nodes.

IV. OPTIMIZATIONS

A. Improved Quadtree

The most important optimization of the Quadtree consists of improving its biggest advantage. The Quad* algorithm thanks its greatest performance gains compared to A* to the big zones of the Quadtree. Thus, there is benefit to enlarging these zones. We observe that the advantages of square Quadnodes, especially that direct paths are correct, also apply to rectangular Quadnodes. Hence, the construction of the Quadtree can be adapted such that Quadnodes can be merged, as long as they stay rectangular. This results in less Quadnodes and hence fewer nodes searched by Quad*, and bigger jumps during the search. An example of an improved Quadtree is shown in Figure 2.

B. Grouping of nodes

The second important optimization improves the Quad* algorithm itself. In this optimization we group searched nodes per Quadnode. In each step of the search, the algorithm expands all searched node of the Quadnodes at once. This results in bad nodes being filtered more quickly.

V. RESULTS

The figure below shows the results of the performance of Quad*. We compare A*, Quad* and the optimized Quad* (see Section IV). The results show that Quad* has a performance gain of about 5%, but once optimized this reaches about 175%.

VI. CONCLUSION

The resulting optimized Quad* algorithm is significantly faster than A*. It is fully parallel and scales perfectly with the number of cores. The disadvantage of Quad* is that the effective pathplanner consists of a sequential algorithm. This means that in this respect there is no extra gain to A*. Nevertheless, the use of a Quadtree forms a good basis for a parallel pathplanner.

WORD OF THANKS

I want to thank prof. R. van de Walle for the opportunity to execute this research, as well as Aljosha Demeulemeester and Jonas El Sayeh Khalil for their aid in reaching these results.

REFERENCES

Parallele pad planning van een groot aantal actoren in ware tijd in een dynamische omgeving

Alexander Dooms

Supervisor(s): prof. dr. ir. Rik Van de Walle, Aljosha Demeulemeester, Jonas El Sayeh Khalil

Abstract—Huidige padplanningalgoritmen zijn niet efficiënt genoeg voor optimale padplanning in dynamische omgebings om een groot aantal actoren in ware tijd. Bovendien zijn er geen ware-tijdalgoritmen die parallelleismen ten volle benutten. Het doel van deze thesis is het vinden van een basis voor zo een algoritme. Ons baserend op de literatuurstudie stellen we een algoritme voor dat gebruik maakt van een Quadtree als roadmap. Een voordeel van Quadtrees is dat veranderingen in de omgeving slechts een klein aantal nodes zal beïnvloeden bij het veranderen van de omgeving. Een tweede voordeel is dat het pad plannen gebruik maakt van het feit dat de Quadtree nodes (en de zones gedefinieerd door deze nodes) enkele vrije nodes zijn, wat betekent dat elk pad dat binnen de zones blijft een correct pad. Het determineren van het optimale pad is dan gereduceerd tot het rekenen met coördinaten. Enkele optimalisaties zijn onderzocht, waarvan er twee een bijzondere snelheidsverleng opleveren. De grootste nadeelen zijn dat de Quadtree enkel werkt op een regulier veld met kosten 1 of √2 voor diagonale links), en dat hoewel elke acteur volledig individueel zijn pad kan zoeken, er geen parallelleismis ipe tijdens het padzoek zijn zelf. Desondanks halen we een grote snelheidswinst in vergelijking met A*.

Keywords—Globale padplanning, Parallele padplanning, Quadtree

I. INLEIDING

PADPLANNEN is al een lang bestaand probleem dat een impact heeft in meerdere gebieden, zoals computerspel len, robotica of simulaties. In deze applicaties vormt het ware-tijdgedrag een belangrijke limitering op de rekentijd. In simulaties heeft met nood aan minstens 30 beelden per seconde, hetgeen, na het weergeven van het beeld, maar enkele milliseconden over laat aan andere berekeningen, zoals het padplannen.

Het opkomen van dual- en quad-core computers heeft de mogelijkheid gecreëerd voor parallelle oplossingen voor dit probleem. Nochtans bestaat er nog geen parallelle padplanner, enkel padzoekers. Het verschil ligt erin dat een padplanner een optimaal pad vindt tussen twee punten, terwijl een padzoeker zo snel mogelijk een pad vindt, ongeacht of er een sneller, korter of beter pad bestaat.

De bedoeling van deze masterproef is om een padplanner te ontwerpen die gebruikt maakt van de toegankelijke dual en quad-core CPUs. Met behulp van parallelleismis is het mogelijk veel meer werk te verrichten in dezelfde tijdsspanne.

II. ROADMAPS

A. Roadmaps in het algemeen

Roadmaps zijn grafen die een abstractie vormen van de virtuele omgeving. Het nut van een roadmap bestaat er in het alge meen in de zoekruimte te verkleinen. Indien men over een roadmap beschikt, bestaat het padzoeken uit het vinden van een pad tussen de nodes in de roadmap die het dichtst bij de start en het doel liggen. Het zoekalgoritme verkent dus veel minder nodes. Een ander voordeel van een roadmap is het feit dat alle actoren gebruik kunnen maken van dezelfde roadmap. Eenmaal deze dus opgesteld is, brengt dit een voordeel met zich mee voor alle actoren.

Een nadeel van roadmaps is dat er in het algemeen geen specificatie is over hoe de graaf moet reageren op veranderingen in de omgeving. Het grootste nadeel, vooral in het kader van deze thesis, is dat paden gevonden met behulp van een roadmap suboptimaal zijn. De reden hiervoor is dat de gevonden paden de nodes van de roadmap gebruiken, hetgeen meestal een omweg vormt. Ook is het verbinding van de eindpunten van het pad met de start en het doel meestal een omweg.

B. Quadtree

Het algoritme ontwikkeld in deze thesis maakt, desondanks de hierboven aangelijke nadeel, gebruik van een roadmap. Zie Figuur 1.

Fig. 1. Een Quadtree

Fig. 2. Een verbeterde Quadtree

Een Quadtree bestaat uit Quadnodes, die elk een vierkante zone bepalen. Een Quadtree wordt opgesteld door eerst de volledige omgeving als een Quadnode te beschouwen. Elke Quadnode in de Quadtree wordt dan in 4 gelijke vierkanten opgesplitst. Als een Quadnode enkel vrije of enkel bezette nodes bevat, dan is deze node klaar. In het eerste geval wordt deze dan definitief aan de Quadtree toegevoegd, in het tweede geval wordt deze verwijderd. Elke Quadnode bevat, volgend uit de constructie, enkel vrije nodes. Het voordeel hiervan is dat elk pad dat binnen een Quadnode een correct pad is. Dit geldt ook voor paden die in naburige zones eindigen, indien men spreekt van directe paden, i.e. paden die zo twee nodes verbonden door eerst in een rechte lijn te bewegen en daarna diagonaal (zie Figuur 3). Deze eigenschap ligt hem in het feit dat Quadnodes vierkante zones bepalen die enkel vrije nodes bevatten.

III. PADPLANNER

A. A* pathplanner

De meest gekende padplanner is A*[2]. Dit zoekalgoritme zoekt een optimaal pad tussen twee nodes in een graaf. A* is
een gretig algoritme, maar maakt intelligente keuzes met behulp van een heuristiek. De heuristiek wordt bepaald door een probleem specifieke functie die, voor de correcte werking van A*, een conservatieve ondergrens moet geven van de kost van een node tot de doel node. In het kader van padplannen is een vaak gebruikte heuristiek de euclidische afstand, aangezien de afstand tussen twee punten nooit kleiner kan zijn dan een rechte lijn.

B. Quad*

Quad* is het algoritme dat ontworpen werd in deze thesis. Het is een variant op A* en maakt gebruik van de eigenschappen van de Quadtree. Het verschil met A* bestaat erin dat Quad* telkens directe paden legt naar de nodes grenzend aan de huidige Quadnode. Dit mag, wegens de eigenschappen van directe paden in Quadtrees. Er bestaat ook geen korter pad dan een direct pad.

Het resultaat hiervan is dat Quad* veel minder nodes onderzoekt dan A*, omdat deze "springt" over de zones van de Quadnodes. Een klein voorbeeld hiervan is gegeven in Figuur 4. Men ziet duidelijk het verschil in het aantal onderzochte nodes.

IV. OPTIMALISATIES

A. Verbeterde Quadtree

De belangrijkste optimalisatie van de Quadtree bestaat er in om het grootste voordeel ervan te verbeteren. Het Quad* algoritme dankt namelijk zijn grootste snelheidswinst tegenover A* aan de grote zones in de Quadtree. Er is dus baat bij het vergroten van deze zones. We observeren dat de voordelen van vierkante Quadnodes, namelijk dat directe paden correct zijn, ook gelden op rechthoekige Quadnodes. Het opstellen van de Quadtree kan dan zo aangepast worden dat Quadnodes terug samengevoegd kunnen worden, zolang ze rechthoekig blijven.

We krijgen zo minder Quadnodes en dus minder nodes die onderzocht worden door Quad*, en grotere sprongen tijdens het zoeken. Een voorbeeld van een verbeterde Quadtree is te zien in Figuur 2.

B. Groeperen nodes

De tweede belangrijke optimalisatie verbetert het Quad* algoritme zelf. In deze optimalisatie worden onderzochte nodes gegroepeerd per Quadnode. In elke stap van het algoritme worden alle nodes van een Quadnode in één keer onderzocht. Dit zorgt ervoor dat slechte nodes sneller weg gefilterd worden.

V. RESULTATEN

Onderstaande grafiek toont de resultaten van de performantie van Quad*. We vergelijken A*, Quad* en de geoptimaliseerde Quad* (zie Sectie IV). Uit de resultaten blijkt dat Quad* een performantie winst van slechts 5% behaalt, maar eens geoptimaliseerd stijgt dit tot ongeveer 175%.

VI. CONCLUSIES

Het resulterende geoptimaliseerde Quad* algoritme is beduidend sneller dan A*. Het is volledig parallel en schaalt perfect met het aantal cores. Het nadeel aan Quad* is dat het effectieve padzoeken serieel is, en er dus op dit vlak geen extra winst ten opzichte van A* is. Desondanks vormt het gebruik van een Quadtree een goeie basis voor een parallelle padplanner.

DANKWOORD

Ik wil prof. R. van de Walle danken voor de gelegenheid om dit onderzoek uit te voeren, alsook Aljosha Demeulemeester en Jonas El Sayeh Khalil voor hun steun tot het bereiken van deze resultaten.

REFERENCES


Contents

1 Introduction ................................................................. 1
  1.1 Problem statement .................................................. 2
    1.1.1 Path planning on the GPU ................................... 3
  1.2 Goal setting ............................................................ 4

2 Related work .................................................................... 5
  2.1 Basic graph search ..................................................... 5
  2.2 Sequential CPU pathplanning ......................................... 6
    2.2.1 A* and its adaptations .......................................... 6
    2.2.2 Roadmaps ............................................................ 8
    2.2.3 Potential fields .................................................... 10
    2.2.4 Ant Systems ........................................................ 11
  2.3 Parallel CPU pathplanning ............................................... 13
    2.3.1 A* based ............................................................. 13
    2.3.2 Roadmaps ............................................................ 15
    2.3.3 Potential fields .................................................... 15
    2.3.4 Genetic ............................................................... 16
  2.4 GPU pathplanning ........................................................ 17
    2.4.1 For a single agent ............................................... 17
    2.4.2 For multiple agents sharing a target location ............... 18
    2.4.3 For multiple agents with different target locations .......... 20
  2.5 Conclusion ................................................................. 21

3 Parallel Quadtree ............................................................ 22
  3.1 Quadtree ................................................................. 23
    3.1.1 Construction of the Quadtree .................................. 24
    3.1.2 Handling a dynamic environment .............................. 27
  3.2 Parallelizing the Quadtree ............................................. 30
  3.3 Quadtree on the GPU .................................................... 32
  3.4 Conclusion ................................................................. 32
4 Quadtree pathplanning
  4.1 A* pathplanning .................................................. 33
    4.1.1 Disadvantages .................................................. 35
  4.2 Quad* pathplanning ................................................. 37
    4.2.1 The Quad* algorithm ............................................ 37
    4.2.2 Example execution .............................................. 40
    4.2.3 Clarifications on Quad* ........................................ 41
  4.3 Conclusion ........................................................ 45

5 Optimizations
  5.1 Filter nodes added to the open list ................................ 46
  5.2 Add only best path to Quadnode .................................... 48
  5.3 Grouping open list nodes per Quadnode ............................. 48
  5.4 Merging of Quadnodes ............................................. 49
  5.5 Parallel Quadtree merging ........................................ 51
  5.6 Conclusion ........................................................ 52

6 Results
  6.1 Hardware and software platform .................................. 53
  6.2 Types of tests ..................................................... 54
  6.3 Execution time of A*, Quad* and optimizations .................... 55
    6.3.1 Effect of the number of agents ................................ 55
    6.3.2 Effect of the map size ......................................... 57
    6.3.3 Effect of the block size ....................................... 58
  6.4 Quadtree construction time ....................................... 59
    6.4.1 Effect of the number of threads ................................ 59
    6.4.2 Effect of the block size ....................................... 59
    6.4.3 Effect of the map size ......................................... 59
  6.5 Conclusion ........................................................ 61

7 Conclusion ........................................................... 62

8 Future work .......................................................... 63

A Data structures ..................................................... 65
  A.1 Environment ........................................................ 65
  A.2 Quadtree ............................................................ 65
  A.3 Binairy heap with update functionality ............................. 66

Bibliografie ............................................................ 69
List of Figures

2.1 Depth-first and Breadth-first search examples ...................... 6
2.2 Example of A* ................................................. 7
2.3 Comparision of IDA* and Fringe Search ............................... 8
2.4 A Probabilistic Roadmap ......................................... 9
2.5 Growth of an RTT .............................................. 9
2.6 A small Quadtree ................................................ 10
2.7 Continuum Crowds overview ...................................... 10
2.8 A potential field .................................................. 10
2.9 Behaviour of real ants when faced with an obstacle ............... 12
2.10 Ant System emulation of the same scenario as in Figure 2.9 .... 12
2.11 Parallel Bidirectional Search .................................... 13
2.12 Parallel Continuum Crowds ....................................... 15
2.13 Genetic algorithm ................................................ 16
2.14 The graph after three iterations of R* ............................... 18
2.15 The steps of the CrowdE framework ............................... 19
2.16 Grid Space possible local paths in a block ....................... 19
2.17 Steps of the All-Pairs Shortest-Paths algorithm for a block ... 20

3.1 An example $16 \times 16$ world .................................. 22
3.2 Interconnected nodes with link costs ............................... 23
3.3 An example of a direct path in a Quadnode zone .................. 24
3.4 The steps of dividing Quadnodes ................................ 25
3.5 First check for adjacency ....................................... 26
3.6 Example of second adjacency check ............................... 26
3.7 Checks after adjacency checks ................................ 27
3.8 Adding an obstacle to a Quadtree ............................... 28
3.9 Removing an obstacle to a Quadtree ............................. 29
3.10 Finished Quadtrees when using $8 \times 8$ blocks .................. 30
3.11 The four passes for connecting the Quadnodes between blocks .. 31
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DFS</td>
<td>Distributed Fringe Search</td>
</tr>
<tr>
<td>ERPP</td>
<td>Experience-based Randomized Path Planner</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>IDA*</td>
<td>Iterative Deepening A*</td>
</tr>
<tr>
<td>PBS</td>
<td>Parallel Bidirectional Search</td>
</tr>
<tr>
<td>PHS</td>
<td>Parallel Hierarchic Search</td>
</tr>
<tr>
<td>PRA*</td>
<td>Parallel Retraction A*</td>
</tr>
<tr>
<td>PRM</td>
<td>Probabilistic Roadmap Method</td>
</tr>
<tr>
<td>PRS</td>
<td>Parallel Ripple Search</td>
</tr>
<tr>
<td>RRT</td>
<td>Rapidly-exploring Random Tree</td>
</tr>
<tr>
<td>RTA*</td>
<td>Real Time A*</td>
</tr>
<tr>
<td>RTAA*</td>
<td>Real Time Adaptive A*</td>
</tr>
<tr>
<td>SPACO</td>
<td>Shortest Path Any Colony Optimisation</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Something that comes natural to most people, is finding a path to travel through the environment around us. When we want to go outside, we know that we can go out the room, to the hallway and then out the front door, and this without bumping into the walls or other objects that might be in our way. However, this seemingly easy task has proven to be quite hard for a computer. Indeed, many algorithms have been developed over the years to solve so called pathfinding and pathplanning problem. The difference between the terms is that the latter is about finding the optimal path, often determined by travel time or distance, whereas the former is about finding any path from start to destination.

The problem of pathfinding and pathplanning is one occurring in a multitude of areas. Examples of such are robotics, simulations and gaming. In robotics, a physical robot may try to execute a certain task which requires it to move, and thus the robot needs to search for a way to reach its destination without colliding. A simulation uses virtual entities, called agents, which may represent anything from humans to animals to vehicles, to replicate a scenario in the confines of a virtual world. Simulations are often used to look how a group of people might react when facing a certain situation. An example is finding bottlenecks of a building, such as staircases or doors, in case of an evacuation. In games, a virtual world might be populated with computer controlled enemies, racers, or even armies of hundreds to thousands of soldiers, and each of these must be able to navigate in the game world.
1.1 Problem statement

Finding such a path is in itself fairly easy. The difficulty comes from the constraints imposed by the application. For most applications, such as for games, it is required that the algorithm runs in real time at an acceptable frame rate. If a frame rate of 50 frames per second is desired, all calculations and the rendering have to be finished within a time window of 20 milliseconds for each frame. This real-time constraint can be relaxed by using multi-threading, where the algorithm can run during the whole frame on one or more separate threads. Two significant factors are the number of agents that need to perform pathplanning and the size and complexity of the virtual world. A larger environment will increase the average and worst case computation times, while increasing the number of agents augments the number of paths that need to be planned within the limited amount of time. Other considerations that increase the complexity of the application are dynamic (moving) objects that need to be avoided, paths that may become obstructed and freed at different times, and interaction between agents.

One of the best known algorithms to use for pathplanning is no doubt the A* search algorithm. The idea is to always search in the general direction of the goal, such that the search space is greatly reduced. To achieve this, A* uses best-first search, where a conservative cost estimate heuristic guides the search. However, the A* search algorithm does not scale well for real-time applications when there is a dynamic environment and when there are many agents. There exist a lot of variants improving certain aspects of standard A*, such as reducing the memory requirement to name an example. Another method for finding paths is using a potential field. Here, a vector field is generated in the whole world in order to guide agents from anywhere towards a goal. The advantage here is that multiple agents can use the same vector field if they share a destination, and the vector field is dependent on the world size instead of the number of pathfinding agents. However, this approach is not fitted for multiple goals and large worlds.

To solve the issue of multiple agents requiring pathplanning, we can use parallel algorithms. Indeed, at any point in time, the world in which the agents move is the same for each agent and interaction between agents is minimal. If we assign a thread per agent, little to no synchronization is therefore required between threads, and the problem lends itself well to parallelization. There are multiple parallel algorithms taking advantage of the advent of multi-core CPUs. Some try to improve A*, for example by searching both from the start point and the goal point towards each other using two threads. Others use abstractions of the environment, called roadmaps or high level graphs, to determine a rough path. Multiple path searches can then connect the nodes of the roadmap in the environment, which can run in parallel.
One of the biggest problems that arises when using parallel pathplanning is that the algorithm always ends up finding suboptimal paths. The reason for this is that in order to use parallelism the problem is abstracted or divided into sub problems. When connecting the partial solutions, some sort of sub optimality arises. For example, imagine we connect A to B and B to C, finding a path that goes through A, then B, then C. It can very well be that there is a faster path from A to C than through B, but because of the abstraction, it is not found. Abstraction or division into smaller sub problems often causes the solution not to satisfy the triangle inequality, i.e. \( d(x, z) \leq d(x, y) + d(y, z) \). Other important aspects are the scalability of the algorithm, its memory usage and the load balancing.

1.1.1 Path planning on the GPU

Currently most computers have two or four cores that can be exploited to parallelize pathplanning, but there is a device that has far more cores, namely the Graphical Processing Units (GPU). With hundreds of cores in a mainstream consumer graphic card\(^1\), there is a lot of potential in offloading pathplanning to the GPU.

There are currently several algorithms that are executed using the GPU. However, these algorithms often concentrate on reducing computation time for pathplanning, not on parallelizing multiple pathplanning queries or reducing the number of calculations. Furthermore, because most pathplanning algorithms have divergent memory accesses, they do not lend themselves well to the Single Instruction Multiple Data architecture of GPUs, as cores end up waiting for each other when they need to access memory from different locations. A lot of potential performance is hence lost as core occupancy is greatly reduced. Another approach, which is very well suited to the SIMD architecture of a GPU, is the all-pairs shortest-paths algorithm. It computes the optimal path between each two points in the world. However, execution times make this method only useful for pre-computation and hence only in a static environment. Finally, the previously mentioned potential fields are also adequate for the GPU programming model, where each point re-calculates its potential until either the potential values converge to a stable solution, a set number of iterations has been executed or any other stopping mechanism. While the sheer number of cores on a GPU allows for much larger worlds, this method still uses a single goal.

\(^1\)For example, an NVIDIA GTX 560 Ti has 384 cores
There are also two memory related challenges regarding GPU algorithms. Firstly, in computers, GPUs do not have as much memory as CPUs do. It can be expected to find 4 to 8 GB of memory in a consumer computer, whereas the aforementioned GPU has only 1 GB of memory. This restriction often means that large virtual worlds need to be divided into regions that each need to be processed in turn, while still requiring the final path to be continuous and correct. Secondly, transferring data from and to the GPU is relatively slow. Algorithms need to minimize this lost time by keeping memory transfers to a minimum, and amortizing the cost of a large transfer by high execution efficiency.

1.2 Goal setting

The challenge can thus be summarized as developing a parallel pathplanning algorithm, which can find paths in a large virtual world in real time by using a high level of parallelism. It has to achieve maximal performance and core occupancy while coping with the SIMD architecture.

It is not required that a single search is faster because of parallelism, but that the total number of paths found within the same time frame as an existing algorithm is larger, and that all paths of the agents can be recalculated in a single frame of about 30 ms when the environment changes.
Chapter 2

Related work

In this chapter, an overview is given of some pathfinding and pathplanning algorithms. Note that this overview is in no way exhaustive. The idea of this chapter is to indicate the principal ideas that have led to pathfinding or pathplanning algorithms and to explain in short their advantages and disadvantages. The algorithm proposed in this thesis is then based on specific ideas in this literature study.

First we will describe some world representations and graph search algorithms in Section 2.1. These algorithms can be categorized based on their level of parallelism. Serial algorithms in Section 2.2, parallel algorithms in Section 2.3 and finally GPU algorithms will be discussed in Section 2.4. Further subdivision groups the algorithms that are variants of each other, or use the same basic ideas.

The difference between a pathplanner and a pathfinder is that the latter searches a path without a guarantee whether the resulting path will be the most optimal path, while a pathplanner does find the optimal path.

2.1 Basic graph search

In most applications where pathplanning is needed, the world is represented by either a graph or a grid of cells. The graph representation of the virtual world has travel costs (e.g. time, distance, difficulty) for each edge connecting two nodes. A grid of cells can either be a uniform grid of rectangular cells or a grid of varied cells (such as triangles). In both cases, the grid can be transformed into an equivalent graph. As such, most pathplanning algorithms are graph search algorithms.

The most basic graph search algorithms are Breadth-First Search[11] and Depth-First Search[12]. The first uses a FIFO queue to search nodes, while the latter uses a LIFO stack. Figure 2.1 shows the order in which nodes are searched in a tree. Both algorithms
search the entire graph until the goal node is found, and hence are very inefficient for anything but small graphs. In addition, because the search stops as soon as the goal node is found, this is a pathfinder. Breadth First Search is improved by *Beam Search*, which prunes the search space by only expanding the N best child nodes of a node. Depth First Search is improved by *Hill Climbing*, which uses an estimated remaining distance heuristic to sort the child nodes of a node in order to steer the search towards the solution.

Although all algorithms described here are unsuitable for graphs representing virtual worlds because they are brute force algorithms, the idea of using a heuristic to direct the search towards the solution lies at the base of the best known pathplanning algorithm, namely *A*\(^*\), which is itself based on *Dijkstra*’s algorithm. *Dijkstra*’s algorithm assigns a cost of 0 to the goal node, then it sets the cost of each neighbouring node to the connecting edge’s cost. It then selects the neighbour with the lowest cost, and updates the cost of each neighbour with the cost of the connecting edge plus the cost of the current node, unless the neighbouring node already has a lower cost attributed. At the end of the search, each node indicates the cost to go to the goal node and the path can be followed by moving at each node towards the neighbour with the lowest associated cost.

![Depth-first and Breadth-first search examples](image)

**Figure 2.1: Depth-first and Breadth-first search examples**

### 2.2 Sequential CPU pathplanning

#### 2.2.1 A\(^*\) and its adaptations

The A\(^*\) search algorithm\[17\] is the most widely known algorithm in pathplanning problems. The strategy of the A\(^*\) algorithm is that it searches in the general direction of the goal, thereby greatly reducing the search space. To do so, A\(^*\) uses best-first search, where the best node is the one with the lowest cost. This cost, \(f\), is calculated by adding the effective cost from the start node to the current node, \(g\), to a conservative estimate of the cost from the current node to the goal node, \(h\), such that \(f = g + h\). This estimate is the result of a heuristic function, which effectively pulls the search towards the goal along
the lowest cost path. An example is given in Figure 2.2. While the A* pathplanning algorithm is successfully used in a multitude of real-time applications, especially games, it does not scale well with an increasing number of agents in a real-time environment.

![Figure 2.2: Example of A*. The top left corner is the g cost to the tile, the bottom left corner is the heuristic cost h, the top right corner is the total cost f = h + g](image)

While A* search has proven its use, there are many adaptations that appeared over the years. Real Time A* [26] re-uses previous costs in order to speed up the algorithm. At each node, it calculates for each neighbour the cost $f = g + h$, where $g$ is the edge cost from the current node to the neighbour. It then selects the neighbour with the lowest cost as current node, and sets the $h$ of the previous node to the lowest $f$ of its remaining neighbours. Nodes without a previously computed $h$ have a heuristic cost equal to that of standard A*.

Another variant is Iterative Deepening A* [25]. This variant tries to greatly reduce the memory footprint of A*, which keeps a cost value for each visited node in memory, while continuing to be a pathplanner. IDA* uses iterations and thresholds to achieve this. During each iteration, the algorithm backtracks when it encounters a node with a cost $f$ greater than the threshold. The threshold is initialized as the heuristic cost $h$ of the start node, and at each iteration it is set to the lowest $f$ above the current threshold. The downside of IDA* is that if the path is long, a lot of nodes are searched multiple times, as the search starts anew from the start node at each subsequent iteration.

Fringe Search [4] builds upon IDA* in order to benefit from its advantages while eliminating the disadvantage of repeated work. A comparison is shown in Figure 2.3. At each iteration, Fringe Search stores nodes that have a cost $f$ beyond the current iteration’s threshold in a so called later list. When there are no more nodes to search in the current iteration, instead of starting at the start node, the next iterations starts with the nodes stored in the later list. In other words, it continues where the previous iteration left of. The memory footprint is hence slightly higher than in IDA*, but states are not searched twice. In comparison with A*, Fringe Search searches many more nodes, but as these nodes need not to be sorted, the cost for searching a node is only a small constant.
Real-Time Adaptive A*\cite{24} speeds up the A* algorithm in real-time applications. In these applications it is often the case that an agent moving along a found path does pathplanning to the same goal in each frame while moving along the path. RTAA* takes advantage of this by updating the heuristic costs $h$ after each A* search. For each node in the list of visited nodes, the heuristic cost can be updated such that future searches are more strongly guided to the same goal.

### 2.2.2 Roadmaps

Several pathplanning techniques are based on roadmaps. A roadmap is essentially a high level graph of the virtual world, where only a few nodes of the world graph are used. Nodes connected in the high level graph represent feasible paths. Paths are planned by first locating their start and goal locations in the roadmap and then determining a route between them from the pathways of the roadmap. Once a roadmap is present, finding new paths is fast, and any graph search algorithm may be used.

There are several pathplanning approaches that use roadmaps \cite{3, 21, 16, 22}, however these approaches all concentrate on the behaviour of a group along a path, as the path itself is easily found in the roadmap. All mentioned methods use some force model which is used to pull the group along the path. This creates visually pleasing changes of direction when changing edges in the roadmap. In addition, there are forces around the obstacles which keep the agents away from them, avoiding collisions.
Roadmaps are often pre-computed, using for example Voronoi diagrams to find a graph whose edges are equally spaced between obstacles. One kind of roadmap construction results in a probabilistic roadmap. *Probabilistic roadmaps* are often used in robotics, where the world is randomly sampled, and from these samples a roadmap is constructed. Samples inside obstacles are removed, and the rest are connected by local pathplanning searches. An example of a Probabilistic Roadmap is shown in Figure 2.4.

*RRT-Connect*\[19\] constructs a roadmap using Rapidly-exploring Random Trees for each pathplanning query. RRTs explore a search space randomly but uniformly. The search is biased to unexplored parts of the world. Thus, RRT quickly explores the space relatively far around the initial point, before exploring the space close to the start more in depth. This is shown in Figure 2.5. RRT-Connect uses two RRTs, one at the start node and one at the goal node, and it expands each tree in turn. Once the trees connect, a path is found. In order to steer the RRTs, it biases the searches toward each other using a heuristic, such that the search space becomes smaller, similar to A*.

![Figure 2.4: A Probabilistic Roadmap](image1)

![Figure 2.5: Growth of an RRT](image2)

A *Quadtree*\[18\] is another kind of roadmap. A Quadtree can be used when the environment is represented by a regular grid of cells or nodes. The purpose of a Quadtree is to reduce the number of nodes in the search space by dividing the environment in squares as large as possible. The Quadtree is constructed by dividing the whole environment in four equally sized square regions. Then, for each region, it checks whether it contains both impassable nodes and free nodes. If it does, it is subdivided in four equal sized square regions. If it does not and it is a free region, it forms a node of the Quadtree. A region containing only obstacles in not added to the Quadtree. The algorithm continues until no region needs to be divided any more. The result is a roadmap where each node corresponds to a square region which contains no obstacles, and is as large as possible. An example is shown in Figure 2.6.
2.2.3 Potential fields

*Potential fields* are pathplanning methods that work mostly on a regular grid of cells (such as the Quadtree). In each cell of the grid, a potential value is calculated, which is the cost to the goal cell. Once the potential field established, it suffices to go from the start cell to the neighbouring cell with the lowest potential value, until the goal cell with cost 0 is reached. Two of the biggest advantages of this method are (1) that multiple agents that share the same goal cell share the same potential field and (2) that the calculation is dependent on the world size instead of the number of pathplanning agents. However, when agents have different goals, they also need different potential fields. In addition, complex worlds require a higher resolution tile grid, and hence smaller tiles, in order to allow agents to move around an obstacle, which increases the complexity. The *Continuum Crowds* [30] method is a well know pathplanning method using potential fields. However, Continuum Crowds does not allow for global pathplanning, as the idea is that a person only has information from the immediate surroundings.

![Figure 2.6: A small Quadtree](image)

![Figure 2.7: Continuum Crowds overview](image)

![Figure 2.8: A potential field](image)
Experience-based Randomized Path Planner[8] is another potential field pathplanning algorithm. Each time a local minimum is encountered, it is added as a node to a roadmap. When a local minimum is reached from another local minimum, the corresponding nodes in the roadmap are connected, the path being represented by the path that was used to reach the second minimum from the first in the potential field. During multiple searches, ERPP first tries to find a path in the roadmap, before using the potential field. To use the roadmap, it uses the potential field to connect the start and goal cells to the start and goal nodes in the roadmap. Because local minima will appear in roughly the same locations regardless of the goal location, pathplanning to different goals benefit from the same roadmap.

2.2.4 Ant Systems

Another approach to pathfinding is using Ant Systems. These systems are heavily inspired by the behaviour of ants. The interesting thing about ants is that while each ant is a simple being with limited knowledge about its surroundings, a colony of ants is an efficient ensemble that can find food and bring it to the nest in a cooperative way. Ant Systems try to emulate this behaviour in order to find solutions to a problem. A particular property of ants is that they leave behind a trail of pheromones. These pheromones attract other ants, and evaporate over time. Ants use these pheromones in order to influence the other ants. For example (see Figures 2.9 and 2.10), if we put an obstacle on an ant trail, then the ants will try to find a way around the new obstacle. If an ant, or a group of ants, finds a path around an obstacle, other ants will end up following this path, being attracted by the pheromones. When the ants that go around the obstacle using the long way arrive back at the trail, the pheromones at the start of their detour will be largely evaporated, while the trail laid by the ants choosing the shorter way won’t be, and will in time be more and more used, and hence more and more attractive.

In graph based problems, such as the Travelling Salesman problem[14], a number of ants are placed in the different nodes, which represent the cities. Each ant then randomly chooses a neighbouring node to go to. In order to avoid ants walking in circles and instead make them explore the search space, each ant remembers the nodes it passed, and may never go to a node it already visited. Once all ants have no way to go any more, there is a clean-up phase. All trails that contain all cities are stored, and pheromones are placed along these trails on the edges of the graph. The shorter the trail, the more pheromones are placed. Then, a new iteration starts, starting with the placing of ants, and then the exploring of ants. But this time, each time an ant needs to choose a next node, it will have a higher chance of choosing a node that is connected by an edge with more pheromones. After a predetermined number of iterations, the trail with the most pheromones will be the best found solution.
Based on this work, an algorithm was developed for finding alternate paths between two points\cite{10}. First an A* search is executed to find the optimal path, which is called the \textit{primer ant}. Along this path a trail of pheromones is laid. Then a number of ants are placed on the starting node, and try to find alternate paths to the goal. These are called the \textit{exploration ants}. Their search is guided by the pheromones on the optimal path, but because of the randomness of the decision making process, the ants will end up finding alternate paths. The problem with this algorithm, in the context of this thesis, is that the ants are not used to find an optimal solution, but instead they explicitly find suboptimal solutions.

The \textit{Shortest Path Ant Colony Optimisation}\cite{2}, or SPACO, algorithm, is an adaptation of the Ant System used for the Traveling Salesman problem for the shortest path problem. In SPACO, ants are placed at the start location and try to find the optimal path towards the goal location. Using multiple iterations, the paths of the ants start to converge towards the optimal solution. The ants base their decision based on three parameters. The first is the level of pheromones, a resource placed by previous ants and which diminishes over time. The second is visibility, which pulls the ants towards the goal, similar to the heuristic function in A*. The third is the cost, which indicates the cost to move to a certain position. For example, a high cost might indicate a difficult mountain path, while a low cost might indicate an open flat space.
While SPACO gives competitive results, the number of iterations required, and the random nature of the algorithm, have two side effects. The first is that the computation time can become large very quickly if we need to find longer paths, and it takes more iterations to smooth out the tentative solutions until it forms the optimal solution. The second is that there is no guarantee on the optimality of the found solution.

2.3 Parallel CPU pathplanning

To solve the issue of multiple agents requiring pathplanning, we can use parallel algorithms. Indeed, at any point in time, the world in which the agents move is the same for each agent and interaction between agents is minimal. If we assign a thread per agent, little to no synchronization is therefore required between threads, and the problem lends itself well to parallelization.

2.3.1 A* based

Brand and Bidarra[6] provide an overview of a couple of A* inspired parallel algorithms, and then introduce a novel one.

*Parallel Bidirectional Search* uses two cores that each start a separate A* search, one from the start to the goal node and the other from the goal towards the start node. As soon as the cores explore the same node, a full path can be constructed. This is shown in Figure 2.11. PBS achieves a large speedup and perfect load balancing between the two cores while sacrificing only a little optimality of the path. The downside is that it uses only two cores.

![Parallel Bidirectional Search](image)
Distributed Fringe Search is a parallel version of Fringe Search[4]. At each iteration in the algorithm, the list of nodes with which the iteration starts is distributed equally amongst the cores. While DFS scales well with the number of cores available, it has bad load balancing. This is because most cores will finish quickly as their paths will not lead to the solution. It is also hard to implement and requires a lot of synchronisation moments between the cores. Compared to PBS, it has a lesser speed-up and is less optimal, although the extra cost of the resulting path is acceptable (around 3-4%[6]).

The third parallel algorithm is Parallel Hierarchic Search. PHS uses a high level graph, obtained for example using the probabilistic roadmap method, to find a high level path. Then each node in the high level path is assigned to a core which connects it to the next node in the path. PHS scales well with the number of cores and is fairly easy to implement compared to DFS, it suffers from extremely costly paths (up to 45% worse than the optimal path). The cause of this is that the high level path nodes may be far from the course of the optimal path.

Although PHS achieves the worst speedup and result among the algorithms described above, it serves as a base for the novel Parallel Ripple Search. PRS uses a high level path, but instead of connecting the nodes of the high level path, each core performs A* to flood the search space. As soon as two searches collide a sufficient number of times, a path can be constructed. The result is that although the high level path defines the start points from which the cores perform their search, the path must not pass through the high level path nodes. The start node and goal node are each assigned to a core, and the remaining cores start from near equidistant nodes in the high level path. The worst case scenario is that only the start and goal nodes connect and result in a path, in which case PRS is reduced to PBS. PRS achieves significant speedup when the path becomes longer, and the resulting path is only slightly less optimal than PBS.

Another parallel A* based search algorithm is Parallel Retraction A*[15]. It is a parallel variant of Retraction A*, which is able to retract and re-expand nodes during an A* search if the allocated memory is full. If a node is retracted, its cost is stored in its parent in such a way that there is no loss of information, and this cost can be restored when the parent node is expanded anew. The parallel variant divides the expanded nodes over the cores using a hashing function. A good such function results in a good load balancing. Each core selects at each step of the search the node with the lowest cost. Nodes that must be expanded are signalled to the correct core depending on the hashing function. If a core has no more memory remaining, it will search for a retractable (childless) nodes in its memory and retract it. Because each core searches independently, the first path is not guaranteed to be the optimal path. Hence, whenever a path from the start node to the destination node is found, its cost is signalled to the other cores. Each subsequent found path will either be discarded if its cost is higher, or replace the previous path if the cost
is lower. Also, any node of an incomplete path with a cost higher than that of the best found path can be pruned, further reducing execution time.

2.3.2 Roadmaps

It has been observed that PRM generation is easily parallelizable[1]. The method used is simply dividing the job of connecting the randomly generated nodes amongst the cores, as each connection between nodes can be done independently.

2.3.3 Potential fields

While Dijkstra lies at the base of the A* algorithm and all of its adaptations, it is in itself fairly inefficient, as it attributes costs to the whole graph. However, one of the uses of Dijkstra is the computation of a potential field, where each cell in a world corresponds to a node in a graph. Solka et al.[29] describe how it can be implemented in parallel, and then show an implementation with a turning constraint of 45° in a grid of equivalent square cells. Although the latter is outside of the scope of this thesis, the parallel method for applying Dijkstra is useful for potential fields. Each core gets attributed an equal amount of nodes, and each node is updated in every iteration of the algorithm, until the solution stabilizes.

A technique for parallelizing Continuum Crowds is proposed by Mao et al.[27]. The suggested method uses a division of the world in rooms and corridors, intuitively suited but not restricted to buildings. A district contains any adjacent rooms, and all their adjacent corridors. Corridors are shared between adjacent districts if they connect a room from both districts. The potential field is calculated on a per district base, to guide agents towards their goal inside a district or towards the edge of a district (see Figure 2.12). When an agent needs to travel between districts, a high level path is found using A*, which works on a graph that represents the rooms and corridors.

![Figure 2.12: Parallel Continuum Crowds](image-url)
2.3.4 Genetic

An approach that is rarely used is pathplanning through a genetic algorithm. Although the genetic algorithm described by Tsai et al.[31] is an implementation of a parallel genetic algorithm for robots on a specific hardware platform, there are a few ideas that could be used to optimize pathplanning. A genetic algorithm starts with a set of randomly generated solutions, called the population, which even need not be legal. At each iteration the population is ranked based on a fitness score. The higher the fitness of a particular solution, the better it is. An illegal solution, for example a path going through an obstacle, has a fitness of 0 regardless of other factors, while other solutions have a fitness based on the cost of the path, where a lower path costs results in a higher fitness. Then a new population is constructed by going through a selection mechanism, which may be random, selects the best solutions, or any other selection mechanism. The selected solutions are combined to create new solutions which are added to the new population, and then they are slightly changed as well, such as removing a node, adding a node, or replacing a node (see Figure 2.13). The new population at the end of an iteration has the same number of solutions as the old population. The algorithm can execute any number of iterations, slowly improving the solutions in the population, or it can stop as soon as a solution with a fitness score above a threshold is found. The solution with the highest fitness score is then the found path.

A drawback of this algorithm is that good solutions can be lost during iterations, instead of being improved upon. A possible way out of this is using an elite mechanism, which automatically selects the best solutions of an iteration and keeps them intact for the next iteration. Another drawback is that the solutions may converge to a local optimum. Adding a small population of random solutions at each iteration solves this issue.

Figure 2.13: Genetic algorithm
2.4 GPU pathplanning

Currently most home computers have two or four cores that can be exploited to parallelize pathplanning, but there is a device that has far more cores, namely the Graphical Processing Units (GPU). To be sure, with 384 cores in a mainstream consumer graphic card, there is a lot of potential in offloading pathplanning to the GPU. However, using the GPU to parallelize an algorithm is not as trivial as it might seem. Using the GPU has a couple of memory related drawbacks, such as relatively little memory available, data needs to be transferred between the CPU and the GPU which is relatively slow, and finally, the GPU requires coalesced memory accesses for maximal performance, as memory accesses are costly.

2.4.1 For a single agent

Bleiweiss describes an efficient A* search implementation on the GPU[5]. The challenge of A* is that its memory accesses are highly divergent; hence a lot of performance is lost due to memory accesses. The implementation proposed by A. Bleiweiss focusses on A* searches running on sparse graphs that are stored in adjacency format. He shows certain techniques to use the GPU more efficiently, and achieves up to 55% speedups in execution time between GPU and CPU implementations. However, these techniques show some shortcomings. Firstly, the implementation is designed around sparse graphs, which we can safely say is never the case for graphs representing a virtual world in pathplanning applications, except for roadmaps. Secondly, the implementation is a speed-up of a query from a single start to a single goal location.

\( R^* \)[20] is a pathplanning algorithm originally designed for robots, but is also applicable to 2D spaces. \( R^* \) tries to reduce the execution time of A* by sacrificing the optimality guarantee. It achieves this by reducing the amount of space searched. \( R^* \) works similarly to A*, but when expanding a node, instead of generating all immediate successors of the state, it generates a predefined number of states at some distance \( \delta \) from that state. A partial execution can be seen in Figure 2.14. If the goal state is within the distance \( \delta \) of the current state, it is selected among the generated states, such that it is impossible to “jump” past the goal. Once a path is found, each state among the path is not necessarily connected, and short pathplanning searches need to be executed to connect these unconnected states, resulting in the final path. These short range searches are excellently fitted for massive parallel execution, because they are independent from each other and easy to solve, and hence adhere well to the GPU paradigm. Although \( R^* \) floods significantly less of the world than standard A*, it is suboptimal, suffers from complex, obstacle filled, worlds, and concentrates on a single start and goal locations.
2.4.2 For multiple agents sharing a target location

A step towards multi-agent path planning is formed by algorithms that allow multiple agents to do pathplanning towards the same goal. A possible solution might be found by using a potential field, where each agent traveling to the same goal can use the same field. A naïve implementation on the GPU might consist of assigning a thread to each grid cell, and recalculating its value at each iteration depending on the neighbouring values, until values either converge to a stable solution or until a set number of iterations have been done.

Demeulemeester et al.[13] propose the CrowdE framework, a hybrid pathplanner that uses a potential field to guide agents towards their common goal. A coarse A* solver is used to determine the regions of the world containing the agents, the goal location and where the found paths will pass. This A* search is executed on a pre-computed roadmap. During execution, each agent will determine the node in the roadmap closest to its position. Then, from each such node a path is found to the node closest to the target location using A*. Subsequently a region of interest is constructed around the paths found. Inside this region, a potential field is calculated. An example is shown in Figure 2.15. This ensures that the potential field is only considered where agents will pass, instead of the whole virtual environment. To calculate this field, it is divided in blocks and each block is calculated using a working queue. The goal is to eliminate unnecessary work when iterating after the potential values have converged to a stable state.

Caggianese and Erra[7] propose another multi-agent single target GPU algorithm. The idea is to divide the world in equal sized rectangular blocks of tiles. The pathplanner then selects each tile at the border of a block and each tile that contains an agent, and assigns a thread to it. Each thread can perform A* without interference from the other threads. The A* search is conducted in the direction of the goal tile, but stops on the first tile outside of the block where the search started, or when the goal is reached. This ensures
that the A* runs are very short and quick. If an agent needs to move towards the goal, it will have an A* path in his current block that starts at his tile, and when he leaves the block, he’ll be on a border tile of another block, which will have an A* path ready to continue guiding him towards the goal. Possible paths inside blocks are shown in Figure 2.16. A problem that may arise here is that a path may form a loop between two blocks, in case a path must go around a big obstacle. In that case no path will be found and the pathplanner fails.
2.4.3 For multiple agents with different target locations

The type of pathplanner that this thesis targets is the kind that is able to navigate multiple agents, each towards its individual goal. During our research, we did not encounter such pathplanners that can be used in real-time. We do however present two offline pathplanners.

The first is the *All-Pairs Shortest-Paths algorithm*\[23\]. It computes in a grid all shortest paths that go from each tile to each other tile. To do so, the algorithm first divides the grid in equal sized blocks of tiles. Then it iterates over the blocks diagonally. At each iteration, there are three steps (see Figure 2.17). In the first step, the shortest path from each tile to each other tile is calculated inside the block. In the second step, the path from each tile in the block to each tile in the blocks on the same horizontal or vertical line inside the world is calculated. In the third step, the same is done but to every remaining block. In each step, each path can be calculated independently, and hence each target tile can be assigned to a thread. At the end of the iterations, the lowest cost path between each pair of tiles is known, and a simple lookup is all that remains to be done to guide an agent towards its goal. The two main drawbacks of this algorithm are that the execution time increases exponentially, and it is not applicable to dynamic worlds.

![Figure 2.17: Steps of the All-Pairs Shortest-Paths algorithm for a block](image)

The second planner is the *g-Planner*\[28\]. g-Planner uses a probabilistic roadmap to sample the search space, followed by a graph search for pathplanning. The samples of the PRM can be taken in parallel, as they are independent of each other. A parallel nearest neighbour search is used to connect samples. For querying, a parallel graph search algorithm is used. Depth First Search can be executed in parallel for multiple agents having different goals. Each thread is assigned to an agent query, and can perfectly run concurrently without synchronization requirements. The drawback of g-Planner is that it takes a lot of time to construct the roadmap (up to 100ms for 1000 samples). g-Planner is designed for robots, and hence is logically not optimized for 2D path planning.
2.5 Conclusion

The conclusion of this preliminary research is that there are a lot of different approaches to pathplanning and pathfinding. Each approach tries to tackle the problem from a different viewpoint, using varying techniques and ideas, and creating solutions which have specific advantages and suffer from different drawbacks.

Considering all the work that has been done in the domain, it is clear to see that while there are already parallel pathfinders, there is no parallel pathplanner that works in real-time. However, there are multiple ideas presented above that can be re-used for a new algorithm that considers optimal paths using parallel execution. The next chapters detail which ideas have been re-used in the algorithm that resulted from this thesis, why they have been chosen, and how they tackle the problem.
Chapter 3

Parallel Quadtree

In the previous chapter we gave an overview of developed pathplanners and pathfinders, in addition to methods that improve their performance, such as roadmaps. The algorithm presented in this thesis uses the Quadtree as roadmap (see Section 2.2.2). In this chapter we go into more detail about this datastructure. In Section 3.1 we introduce the Quadtree in more detail. In Section 3.2 we discuss a parallel variant. And finally, in Section 3.3 we briefly talk about a GPU adaptation.

Figure 3.1: An example $16 \times 16$ world. Nodes in a blocked state are black.
3.1 Quadtree

We assume the following world representation. The environment consists of a regular grid of nodes, each node connected to 8 neighbours. Obstacles are represented by nodes that are in a blocked state (see Figure 3.1). Each link connecting two nodes has a cost of 1 for horizontal or vertical links, and $\sqrt{2}$ for diagonal links (see Figure 3.2).

The idea of using a roadmap is to reduce the number of searched nodes. In addition, in the context of a large number of individual agents, a single roadmap can be used by all agents. For this reason the algorithm uses a roadmap to speed up the pathplanning. However, using a roadmap introduces sub-optimality for found solutions, because we pass through the nodes of the roadmap. This results in detours, as the roadmap is an abstraction of the environment.

The Quadtree offers an important advantage over these more traditional roadmaps: any node within the zone of a Quadtree node, henceforth called Quadnode, is part of the free space. The result of this advantage is that when doing pathfinding with a Quadtree, we can be sure that any direct path between two nodes that are part of the same Quadnode is a viable path (i.e. one which passes only through unblocked nodes, see Figure 3.3). The same is true for any direct path from a node in a Quadnode to a node adjacent to the same Quadnode (see Figure 3.4(a)). The importance of this advantage is that the pathplanning algorithm that uses our Quadtree can find optimal paths quickly. This is further explained in Chapter 4.

A formal definition of a direct path is: A direct path is an optimal path between two nodes in a Quadnode or to a node adjacent to the same Quadnode, where the path first goes horizontally or vertically towards the goal and then diagonally. A direct path is always valid.

The reason for going first in a straight line, followed by going diagonally, is that in this way the path will always be valid. If we would first take a diagonal path, in the case of traversing the boundary of Quadnodes, we could exit the free space. An example of this is shown in Figure 3.4(b).
A Quadtree is composed of Quadnodes. Each Quadnode represents a square zone of the environment, containing only free nodes. The Quadnodes are as large as possible, without containing blocked nodes. Each Quadnode has a set of neighbours, which are the Quadnodes who’s square zone shares an edge or corner with this Quadnode’s square zone.

### 3.1.1 Construction of the Quadtree

The construction of a Quadtree is a simple iterative process. At the start of the algorithm, the whole environment is considered as being a single Quadnode. This Quadnode is added to a FIFO queue, which we call the *construction queue*. Resulting Quadnodes that will not be processed any further, are added to an initially empty list of finished Quadnodes.

At each iteration of the algorithm, a Quadnode is popped from the construction queue. Then we iterate over each node within the zone of the Quadnode. As soon as we encounter both a free node and a blocked node, we divide the Quadnode in four equally sized smaller Quadnodes, and push them on the construction queue. If however a Quadnode contains only free nodes, it is added to the list of finished Quadnodes. A Quadnode containing only blocked nodes is removed, it will not be part of the Quadtree, and will hence not be accessible to agents. An example of a Quadtree construction is show in Figure 3.4.
Figure 3.4: The steps of dividing Quadnodes
Once all Quadnodes have been generated, each Quadnode needs to add its neighbours to its set of neighbours. Because all Quadnodes represent square zones, this can be done using the coordinates. First we check if the Quadnodes are adjacent. Two Quadnodes A and B can only be adjacent if A is on top of B, B is on top of A, A is left of B or if B is left of A. Not that this includes corners. A graphical representation can be found in Figure 3.5.

![Figure 3.5: First check for adjacency](image)

(a) A right of B: $A.x + A.size = B.x$
(b) A on top of B: $A.y + A.size = B.y$

Figure 3.5: First check for adjacency

Once we filtered the Quadnodes that cannot be adjacent, we iterate over the candidate Quadnodes. For each of them, we check the size and the other coordinate. This must be done because the first adjacency check only determines if a Quadnodes shares a coordinate. For example, if we determined that B is left of A, then we are going to check its $y$ coordinates. This means that we need to check whether B is too high or too low compared to A. This example is illustrated in Figure 3.6.

![Figure 3.6: Example of second adjacency check](image)

Figure 3.6: Example of second adjacency check. B has a shared $x$ coordinate with A, but their $y$ coordinates clearly determine that they are not adjacent.

A few examples for determining adjacencies are shown in Figure 3.7. A Quadnode that passes these checks is one that needs to be added as a neighbour.
3.1.2 Handling a dynamic environment

An advantage which is important for dynamic environments is that any changes in obstacles only requires some local repair. When an obstacle is moved, removed or added, we need to identify the Quadnodes that are impacted, and either merge or split Quadnodes similarly to the construction algorithm.

If an obstacle is added to a Quadnode, then this node will need to be split until there are only valid Quadnodes remaining. This is handled in three steps. The first step is to break the links to the Quadnode. The next step is to start the construction algorithm with the changed Quadnode as initial Quadnode. Once the construction is complete, we connect the new Quadnodes with each other and the bordering Quadnodes. This is shown in Figure 3.8. If an added obstacle is located on multiple Quadnodes, then each Quadnode must be repaired in the same way. Once all these Quadnodes are repaired, they are connected with each other and the rest of the Quadtree.
If an obstacle is removed from the environment, we determine the smallest legal Quadnode zone that fully encompasses the removed obstacle. Then we replace the contained Quadnodes by a single Quadnode, breaking all links to the now non-existing Quadnodes. If the newly obtained Quadnode contains another obstacle, then we split it in the same way as when an obstacle is added. In the other case, we continue to merge quadruplets of Quadnodes until we have a legal Quadtree again. When done, the Quadnode needs to be connected in order for the merge to be completed.
Figure 3.9: Removing an obstacle to a Quadtree.

If an obstacle changes position, we first handle the nodes that became free as if the obstacle was removed, and then we handle the nodes that became blocked as if the obstacle was added. In some cases, the second step is not required, because removing an obstacle already repairs a larger zone.
3.2 Parallelizing the Quadtree

In order to parallelize the construction algorithm of the Quadtree, only a small change is required. Instead of considering the whole virtual world as a single Quadnode at the start of the algorithm, it is divided in equal sized square blocks. Each block is then a small Quadtree, which can be constructed fully independent of the other blocks. See Figure 3.10 for an example of multiple small Quadtrees. If we assign a core to each block, then the construction of the small Quadtrees scales linearly with the number of cores until it exceeds the number of blocks. For example, using a standard world of $1024 \times 1024$ with blocks of size $16 \times 16$, this would scale up to 4096 cores.

![Figure 3.10: Finished Quadtrees when using 8 x 8 blocks](image)

Once we have a small Quadtree per block, we need to connect the Quadnodes of each block to the neighbouring Quadnodes of the adjacent blocks. We can again assign a core to each block, which will then add links between the Quadnodes of the block and the Quadnodes of the neighbouring blocks. But because adding a link adds the linked Quadnodes to each other’s neighbour set, if two cores were to add links to each other’s block, we get duplicates or worse, crashes because the two cores insert in each other’s set of neighbours. We could add a mutex, but this would add serial segments in the algorithm. The solution to this problem is using four passes over the blocks. At each pass, a core will handle a block on coordinates $(x, y)$. It will connect the Quadnodes of this block with the neighbouring Quadnodes of the blocks at coordinates $(x+1, y)$, $(x, y+1)$ and $(x+1, y+1)$, as well as connecting the Quadnodes of block $(x+1, y)$ with those of block
At the first pass, the blocks that are handled by the cores are those with even coordinates. At the second pass, the blocks that are handled have uneven $x$ coordinates. At the third pass they have uneven $y$ coordinates, and at the final pass they have uneven $x$ and $y$ coordinates. This process is illustrated in Figure 3.11. In the end, each Quadnode is connected to each neighbouring Quadnode, using a connection algorithm that is fully scalable until the number of cores exceeds the number of blocks divided by four, because of the number of passes. For the previous example, the algorithm scales up to 1024 cores.

After the connection of all neighbouring Quadnodes, the construction of the Quadtree is completed.

![Figure 3.11: The four passes for connecting the Quadnodes between blocks. Only the blocks are shown in these figures](image)
3.3 Quadtree on the GPU

The construction algorithm presented above can likely also be implemented on a GPU architecture, using for example CUDA. It would suffice to replace a core in the explanation with thread block. If we use threads blocks with the same size as a block, then each thread is assigned to a node within the block.

During the construction, each thread will signal to its thread block if it has a free node or not. When all threads have signalled this, the first thread will divide the block in four if required, and signal a new iteration. In the end, the thread block will have constructed a full Quadtree for a block.

During the connection of the Quadnodes of the different blocks, we only need a thread per node that lies at the border of a block. These threads then check if they require a new link to a neighbouring Quadnode.

Please note that the algorithm was not implemented on the GPU, so there are no comparisons of efficiency between the parallel CPU and the GPU variant. In addition, this means that the explanation given above is purely based on speculation and our knowledge of CUDA.

3.4 Conclusion

In this chapter we discussed what a Quadtree is. We showed what its interesting properties are. We also detailed the construction algorithm for the Quadtree, extended this to a parallel variant, and finally we mentioned a possibility for porting the algorithm to a GPU.

In the next chapter we will discuss the novel pathplanner which requires the properties of a Quadtree in order to gain an advantage over existing pathplanners.
Chapter 4

Quadtree pathplanning

In Chapter 3 we discussed the Quadtree as a roadmap for a virtual world, which is represented by a regular grid of nodes. We extended the Quadtree to perform better in a parallel environment.

In this chapter we will discuss the pathplanning algorithm that uses this parallel Quadtree. The goal of the pathplanning algorithm is to find **globally optimal** paths for a large number of individual agents. The presented algorithm shows a significant speed-up, compared to a standard A* search, thanks to the overlying Quadtree roadmap and its characteristics regarding open space and direct paths (see page 23).

In Section 4.1 we will describe the A* pathplanning algorithm in detail. In Section 4.2 we introduce the proposed algorithm, which is an adaptation of A*.

### 4.1 A* pathplanning

A* is a depth-first, greedy algorithm, but which is guided by a heuristic in order to dramatically reduce the search space. During the algorithm, there are two lists of nodes, an *open-* and a *closed list*. The open list contains those nodes that have been encountered by the search, but who’s neighbours have not been explored. The closed list contains all nodes who’s neighbours have been explored. A node in the closed list contains the optimal path to it, from the agent’s starting node.

During the A* algorithm, searched nodes get two costs attributed to them. The *g* cost is the cost to reach this node from the start node, and the *h* cost, or heuristic cost, is the cost determined by a heuristic function. The total cost \( f = g + h \) is the total cost of the node. The used heuristic function must return a value that represents a conservative minimal distance between two nodes. The closer the returned value is to the cost of the optimal path between two nodes, the faster the A* search will finish. However, a heuristic
function that returns values greater than the real distance between two nodes causes the A* search to lose its guaranteed optimality. An example heuristic function in the 2D search space is the Euclidean distance between two points. This is a fairly good heuristic, because an optimal path will never be shorter that a straight line.

Initially, the agent’s starting node is the only node in the open list. This node has an attributed $g$ cost of 0, since the agent does not need to move to reach this node.

At each iteration of the search, the best node of the open list is selected, the best node being the one with the lowest $f$ cost, which means that at any time the open list must be sorted. The best node is removed from the open list. Then we iterate over all its neighbouring nodes. If such a node is already in the closed list, then we skip it, because there already exists a better (or equally good) path to that node. In the other case, we calculate the $f$ cost of the neighbour, where the $g$ cost can be derived from the selected node, and the $h$ cost is determined by the heuristic function. Once the $f$ cost is determined, we insert the node into the open list. However, if the node has already been inserted into the open list, then we only keep the one with the lower $f$ cost. In other words, if we found a better path, we replace the node, but if we found a worse path, we ignore it.

As soon as the best node in the open list is the goal node the search ends. We can then reconstruct the optimal path. An example execution of the A* algorithm can be seen in Figure 4.1.
Figure 4.1: An example A* execution, where an agent starts on node A and has node G as goal. Black nodes are obstacles, grey nodes are in the open list and dark grey nodes are in the closed list. The resulting path is A, f, i, k, G.

4.1.1 Disadvantages

While A* pathplanning is widely know and has a lot of variations, it suffers from a few shortcomings. First of all, the worst-case scenario. If there exists no path between the agent and the goal, A* will flood the search space, expanding every node it can, until the open list is empty and the search stops. This means that A* is reduced to a depth-first graph search. If we ensure that no node, or group of nodes, is isolated in our environment, then this will not cause a problem.
Another problem with A* is the behaviour when the goal finds itself behind a large obstacle, such as in Figure 4.2. The result is that A* floods, like water in a cup, in the corner of the obstacle until there is an overflow and a path is found. In the case of a U shaped dead end and equal distance paths around the dead end to the goal, such as in Figure 4.3, A* floods at the same speed at both sides of the obstacle, finding two paths simultaneously.

Finally, A*, like any other globally optimal pathplanning algorithm, needs to inspect each node at the basic level of the graph representation of the environment in order to guarantee the optimality. This means that at each iteration, the search advances by a single node, slowly creeping towards the solution.
4.2 Quad* pathplanning

We propose a variant of the A* algorithm by using a Quadtree as roadmap, and we will refer to our algorithm as Quad*. Quad* tries to reduce the impact of the disadvantages of A* search by using a Quadtree roadmap. The most important property of a Quadtree is that any direct path between two nodes that are part of the same Quadnode or two neighbouring Quadnodes (see Figures 3.3 and 3.4(a)), is a valid and optimal path. This property is used in Quad* such that the search can skip nodes that are in the middle of a Quadnode’s zone, and go directly to the nodes of neighbouring Quadnodes. The end result is that the search “jumps” over nodes.

4.2.1 The Quad* algorithm

Quad* uses an open- and a closed list, with the same purpose as the lists used in A*. The difference lies in how we expand nodes. At each iteration, the best node is selected from the open list as the current node. This node also stores the Quadnode in which it is positioned, the current Quadnode, as well as the node from whence it came. For each neighbour of the current Quadnode, except those through which the path to the current node has already passed, we generate all the nodes that are adjacent to the current Quadnode (see Figure 4.4). We calculate the costs $g$ and $h$ for each of these nodes, and then add them to the open list, unless the node is already present in the closed list. Just as it is done in A*, if a node is already in the open list, we only keep the version with the lower total $f$ cost.

If the current Quadnode contains the goal node, then we can trace a direct path to the goal. We only add the goal to the open list in this case (see Figure 4.4(c)).
In other words, Quad* adds paths to each node neighbouring the current Quadnode. This means that we try every possible path to each neighbouring Quadnode from the current node. It is very possible that there is a faster path to a specific node, and due to the way A* works, using a heuristic function, the path to the node will be updated to the fastest path before it will be popped from the open list.

Figure 4.4: Quad* steps in detail.
Figure 4.5: Example execution of Quad*
4.2.2 Example execution

In this section, we provide a sample execution of Quad* to clarify the previous description. In the following figures, the black nodes are the start and goal nodes. Nodes in the open list are represented with light grey, while nodes in dark grey represent those in the closed list. The arrows pointing to white, unvisited nodes indicate those that are added to the open list. Those that point to a light grey node indicate the nodes that are updated in the open list.

Figure 4.5 shows the first 11 steps of the Quad* algorithm. In the first three frames (a-c), we see that Quad* expands the nodes as fast as possible towards the goal node. In the following five frames (d-h) there is some flooding around the obstacle. Frame i shows how the algorithm progresses quickly towards the goal around the obstacle, in a single step. The frames j-k show some more flooding, where frame k skips a few steps in order to shorten the example. The last frame (l) shows the final step.

The resulting flood zone of the Quad* search is compared to the flood zone using A* in Figure 4.6. As you can see, Quad* expands fewer nodes, but because this is an example on a small path in a small environment, the difference seems small. As you will be able to see in the results of the tests, the “jumps” that Quad* can take by using 16 × 16 Quadnodes are a significant advantage.

![Figure 4.6: Flood zones of Quad* and A*](image)
4.2.3 Clarifications on Quad*

Some clarification might be required on why we don’t do an A*-like) search directly on the Quadtree nodes. The first reason is that it is impossible to set correct costs on the edges connecting the Quadnodes. If we follow a path through a series of Quadnodes, then the cost we should attach to each edge changes depending on the path. Consider the Quadtree in Figure 4.7. Depending on from which node we start choosing from nodes 1 to 4, and to which node we search a path, choosing from nodes 5 to 8, we get different costs between the Quadnodes. For example, a path from node 1 to node 5 gives us the edge costs $c(A, C) = 1$ and $c(C, D) = 4$, while a path from node 2 to node 5 gives us $c(A, C) = \sqrt{2}$ (see Figure 4.8). In addition, a path from node 4 to node 5 gives us $c(C, D) = 2 + 2\sqrt{2}$ (see Figure 4.9). As you can see, all cases would require different edge costs between the Quadnodes.

Figure 4.7: Example situation of a Quadtree roadmap
Figure 4.8: Problems assigning Quadtree edge costs.

Figure 4.9: Problems assigning Quadtree edge costs.
One could argue to set the edge cost to the distance between the centres of the Quadnodes. This is not a good idea, because this gives non-optimal paths. Consider the situation in Figure 4.10. A path needs to be found from node 1 to node 2. By sight we can see that the optimal path should go to the right around the obstacle, following the path $J \rightarrow K \rightarrow H \rightarrow D \rightarrow A$ or a variation (going through Quadnodes $I$, $G$ and/or $C$).

![Diagram of problem situation and optimal path](image)

(a) A problematic situation.
(b) Optimal path between nodes 1 and 2, with different variations possible.

Figure 4.10: Optimal path in problematic situation.

We now assign the centre to centre cost to each edge, which gives us the graph shown in Figure 4.11(a). If we would use Dijkstra on this graph, with edge costs given by the distance between the centres of the Quadnodes, then the path would go around the left (see Figures 4.11(b) and 4.11(c)). As you can see, this does not give us an optimal path.
Figure 4.11: Non-optimal path in problematic situation using Dijkstra.

(a) Graph with edge costs the centre to centre distance.

(b) Dijkstra on the previous graph.

(c) Resulting sub-optimal path.
The second reason why we don’t perform a graph search directly on the Quadtree, is because even if the costs of the edges between the nodes of the roadmap were correct (for example when using a Probabilistic Roadmap), the resulting path would still be sub-optimal. This is because we need to join a node on the roadmap from the start node, follow a path using the roadmap, and then go form the last node of the roadmap to the real goal. This will nearly always create a detour, and hence a sub-optimal path. An example is given in Figure 4.12.

![Suboptimal path](image)

Figure 4.12: Example of a detour because of a Roadmap.

### 4.3 Conclusion

In this chapter we discussed the novel Quad* pathplanner. Because it is an A* variant, we first described in detail how A* works, and we mentioned a few drawbacks. Then we explained how Quad* works, and how it can take advantage of the properties of a Quadtree, which is explained in Chapter 3.

In the next chapter we go over a few optimizations for the basic Quad* and Quadtree construction algorithms. We will show what the idea behind each optimization is and what advantages and disadvantages they have.
Chapter 5

Optimizations

In Chapter 4 we defined the basic Quad* algorithm. In this chapter we discuss a couple of possible optimization. These optimizations represent additional ideas we got during the development of Quad*, and which have been implemented in order to test them. The results are covered in Chapter 6.

5.1 Filter nodes added to the open list

The first optimization tries to reduce the number of nodes we add during the expansion of a node. It is often the case that too many nodes are added, based on the following observation: We expand a node $X$ with cost $f_X = g_X + h_X$, and we add nodes A, B, C and D for one of the neighbouring Quadnodes (see Figure 5.1).

If we calculate the costs we get:

\[
\begin{align*}
g_A &= g_X + 4, h_A = 2 + 2\sqrt{2} \Rightarrow f_A = g_X + 6 + 2\sqrt{2} \\
g_B &= g_X + 3 + \sqrt{2}, h_B = 3 + \sqrt{2} \Rightarrow f_B = g_X + 6 + 2\sqrt{2} \\
g_C &= g_X + 2 + 2\sqrt{2}, h_C = 4 \Rightarrow f_C = g_X + 6 + 2\sqrt{2} \\
g_D &= g_X + 1 + 3\sqrt{2}, h_D = 3 + \sqrt{2} \Rightarrow f_D = g_X + 4 + 4\sqrt{2}
\end{align*}
\]
You can see that the nodes A, B and C have the same $f$ cost. The meaning of this is that it does not matter whether we go through A, B or C to reach the goal, because at the expansion of either of these nodes, the cost to goal will be the same (because of the exact heuristic function, the cost to the goal is $f_G = g_X + 6 + 2\sqrt{2} = f_A = f_B = f_C$).

The reason why this happens is that it does not matter if we first go in a straight line and then in a diagonal line, the other way around, or even mix it up. The important thing is that we do not go too far. Figure 5.2 shows the different optimal paths in our example. Going through node D would be a case of going too far.

![Figure 5.2: Different optimal paths](image)

While the optimization looks good on paper, when we run tests we quickly find that it does not work in practice. An example of a problem situation is shown in Figure 5.3. The error stems from the fact that while at a certain point the $f$ costs are equal between some nodes, the rest of the path is still strongly dependant on the chosen node.

![Figure 5.3: Filtering nodes can lead to sub-optimal paths](image)

Although the resulting paths are sub-optimal, it is interesting to look at how much sub-optimal the paths are, versus the time gained.
5.2 Add only best path to Quadnode

This optimization is inspired by the previous one, where we try to build upon its increased speed. Here, we don’t try to achieve optimality, but we do try to get a maximum speed-up. The general idea is to make the algorithm more greedy.

When adding a node of an adjacent Quadnode during the expansion of the current node, this optimization only adds the node with the lowest \( f \) cost (see Figure 5.4). This reduces the number of nodes to a minimum, namely 1 per neighbouring Quadnode. The advantage is of course the high speed up, but the disadvantage lies within the greedy execution, and hence the sub-optimality.

![Figure 5.4: Adding only the best node of the neighbouring Quadnode](image)

5.3 Grouping open list nodes per Quadnode

In the following optimization, we group the open nodes per Quadnode. This means that the open list now does not contain nodes, but groups of nodes, depending on the Quadnode in which they are contained. These groups are sorted in the open list depending on the best cost of their contained nodes. This way, when we pop the front element of the open list, we are guaranteed of having the best node included in the group (which is a requirement for the algorithm to work).

Once we have popped the best group of nodes from the open list, we expand all of these at once. To do so we loop over each node \( n_i \) adjacent to the current Quadnode. We calculate its heuristic cost, and select the node from the group \( gn_j \) as parent which adheres to the following criteria:

- The Quadnode containing \( n_i \) is not part of the fastest path to \( gn_j \).
- The cost \( g(gn_j) + distance(gn_j, n_i) \) is the minimum cost to \( n_i \) from any node from the open group.
The reason for this optimization is that it allows the algorithm to do less double work. Take the example execution in Figure 5.5. At step 3 (Figure 5.5(c)), instead of expanding node 3 and then node 4, where we replace the path to node 9 that goes through node 3 with a path that goes through node 4, we immediately find the fastest paths to each node neighbouring Quadnode C. This means we have less open list lookups and changes of priority of elements already in the open list.

However, we also do some useless work using this method. Take step 2, where we expand Quadnode B, for example (Figure 5.5(b)). Here we expand node 1, although in the case of the base algorithm it would never have been expanded.

![Figure 5.5: Example execution when grouping nodes per Quadnode](image)

### 5.4 Merging of Quadnodes

The Quadtree lies at the base of the Quad* search. The reason we use a Quadtree as overlying roadmap is to allow for the pathplanner to “jump” over groups of nodes, reducing the number of searched nodes and hence the time in which a path is found. The following optimization is meant to increase the effectiveness of the structure in order to allow bigger “jumps”.
It is important that this improved Quadtree still has the same attributes as a standard Quadtree regarding *direct paths*. As long as we keep the Quadnodes rectangular, then this property is kept.

In order to allow for bigger “jumps” in Quad*, the improved Quadtree merges Quadnodes. It does this in such a way that all Quadnodes stay rectangular. Two Quadnodes may only be merged if they are adjacent and if their share the same size on the shared side. Figure 5.6 shows which Quadnodes may be merged, and which may not.

![Figure 5.6: Example of Quadnodes that may and may not be merged: Quadnode A may be merged with B or C, because they are adjacent and their size is equivalent on the shared edge. It may not be merged with D, because it is not adjacent to A. It may also not be merged with neither E nor F, because both are of a different size on the shared edge.](image)

During the construction of the Quadtree, each time we have a finished Quadnode (one that contains only free nodes), we check for merge-able nodes before adding it to the tree. When we find a Quadnode with which the new one can merge, we merge the nodes and we check again. Only when the new Quadnode can’t merge with an existing node do we add it to the tree. In Figure 5.7 you can see both the results of the basic and the merged construction algorithms on the example world given in Figure 3.1. As you can see, there are far fewer Quadnodes (42 vs 12), and each Quadnode has far fewer neighbours (maximum 8 vs 4).
5.5 Parallel Quadtree merging

For the parallel variant of the Quadtree, the construction using merged Quadnodes is slightly more complicated. Because we subdivide the environment in blocks and build a Quadtree per block, there are Quadnodes that are not merged because they are not part of the same block. Using the above example map and $8 \times 8$ blocks, we get the Quadtree shown in Figure 5.8.

![Figure 5.8: Parallel Merged Quadtree. The indicated edges show borders that could be merged, but aren’t because of the subdivision into blocks.](image-url)
To merge these Quadnodes, which are part of different blocks, we change the step in the construction algorithm where we connect Quadnodes from different blocks (see page 30). During each pass, each core has access to up to four blocks arranged in a $2 \times 2$ grid (see Figure 3.11). In the case of this optimization, each Quadnode which is adjacent to and merge-able with a Quadnode from another block is merged instead of connected. One of the Quadnodes is destroyed, while the other copies its links and takes up its space.

However, this algorithm will most likely fail. The reason is that a Quadnode can cover more than 2 blocks, and using the 4 passes as described on page 30 does not protect against this. As in theory a merged Quadnode can stretch to any size, we are forced to either work with mutexes or use only a single thread. In the implementation used in the following tests, we have chosen for a single thread. We did so based on the observation that because of the merging inside the blocks, there are only few cases where we need to merge through the border of blocks. In addition, the Quadtree is not meant to be built fully from the ground up during execution of the program.

5.6 Conclusion

In this chapter we discussed a couple of optimizations for the basic Quad* algorithm, described in Chapter 4, and for the Quadtree it uses, which is described in Chapter 3.

In the next chapter we test the Quad* algorithm by comparing it to A*. We also test the different optimizations presented here, and show which ones are the most advantageous.
Chapter 6

Results

The previous chapters detailed the novel Quad* algorithm and a number of optimizations for it. In this chapter we put the algorithm and the optimizations to the test.

In Section 6.1 we define the hardware and software platform on which the tests have been executed. Then we explain the type of tests that were performed in Section 6.2. In Section 6.3 we compare $A^*$ with Quad* and its optimizations. Finally, in Section 6.4 we investigate the Quadtree construction performance.

6.1 Hardware and software platform

All the test are run on an Intel® Core™ i5-3750K, which is a CPU with 4 physical cores and clocks at 3.40 GHz (and up to 3.80 GHz using Turbo boost).

The software platform is a 64-bit console application running on the Windows 8 OS. It is written in C/C++ and uses Win32 native threads. The application allows to generate random maps of any size, randomly place any number of agents, display agent paths, and run different pathplanning algorithms. For this it uses an $A^*$ and a Quad* implementation. The latter has several options for optimizations, which are used in Chapter 5.

The application can either be run in command mode, as shown in Figure 6.1, or using command line parameters. In the latter case, the program executes a single search given a set seed, number of cores, map size, number of agents, block size (for Parallel Quadtrees) and optimization options. In the first case, it is possible to set up specific searches and scenarios. This mode is used for debugging during the development of the algorithms and their optimizations.
6.2 Types of tests

In command mode, the application gives us for each agent the time in milliseconds it required and the length of its path. In both modes, the global metrics give the maximum, minimum and average path length and time required per agent. It also calculates the number of distance units per ms. This metric is obtained by summing over all the path lengths, and dividing by the total time. The more distance units per millisecond an algorithm achieves, the faster the algorithm. Also, because Quad* uses a Quadtree, the total construction time for the Quadtree is given.

In summary, the available metrics are:

- maximum, minimum and average path lengths;
- maximum, minimum and average time required per agent;
- path length units per millisecond calculated;
- quadtree construction time.

We execute both A* and Quad* on the same scenarios, which are generated using the same seed which is set in the command line parameters. The random map generator places a number of obstacles according to the following formula: \( \text{ObstacleCount} = \text{width} \times \text{height} \times 0.005 \). For a map of 1024 \( \times \) 1024 nodes, this corresponds to 5242 obstacles. Each obstacle has a random footprint between 1 \( \times \) 1 and 10 \( \times \) 10 nodes. This means that the number of blocked nodes on a map ranges from \( \text{width} \times \text{height} \times 0.005 \) to \( \text{width} \times \text{height} \times 0.5 \), or from 5242 to 524288 in the case of a 1024 \( \times \) 1024 map.

Because both A* and Quad* are sequential algorithms and the only parallelism comes from the construction of the Quadtree, we do not measure their execution times with respect to the number of cores. All tests hence use 4 threads (the maximum for the used hardware platform), where the pathplanners pop agents without a path from a stack, achieving near linear speed-up with respect to the number of threads.
The following aspects of the algorithms are tested:

- execution time with respect to map size;
- execution time with respect to number of agents;
- execution time with respect to number of block size (not applicable to A*);
- quadtree construction time with respect to number of threads;
- quadtree construction time with respect to map size;
- quadtree construction time with respect to block size.

### 6.3 Execution time of A*, Quad* and optimizations

This section investigates the execution time of A*, Quad* and its optimizations with respect to the map size and the number of agents. All tests have been run using 4 threads which are executed on the 4 physical cores of the CPU, and the Quadtree construction algorithm uses $16^2$ blocks.

#### 6.3.1 Effect of the number of agents

First we investigate the execution time with respect to the number of agents. Each result indicates the average over 50 tests, each test using a different seed to generate the random environment and the placement of the agents. The environment is a $1024^2$ map. The resulting execution times are shown in Figure 6.2. For reference, the average path length is around 565 units (see Figure 3.2).

These results show that Quad* generally achieves a 5% speed-up in this setting. While this is certainly not much, one can see that all optimizations, except Filter, achieve better speed-ups. The optimizations that have the best speed-up are Best node only and Merge Quadtree. Both achieve a speed-up of around 40-45% compared to A*.

In addition, the execution time grows linearly with the number of agents. This is expected behaviour, because both search algorithms are inherently sequential. Agents are simply popped from a common stack, their path is searched, and the next agent is popped. This means that if we had run the tests with only 2 threads, then the execution time would double. And the other way around, if we double the number of threads, then the execution time would be halved.
Figure 6.2: Execution times of A*, Quad* and its optimizations with respect to the number of agents.

Because these results show that both Group nodes and Merge nodes optimizations achieve the best execution times, we combined both and ran the tests again. This is possible because Group nodes changed the Quad* algorithm and Merge nodes only changes the structure of the Quadtree. In Figure 6.3 the results of these tests are shown. It also shows the speed-up of the combined optimizations compared to the A* search. As can be seen, this speed-up is around 175%.

Figure 6.3: Execution time best optimizations with respect to the number of agents. The speed-up is compared to A*
Figure 6.4: Execution times of A*, Quad* and its optimizations with respect to the map size. The line indicates the average path length over all 500 agents and 50 test samples.

6.3.2 Effect of the map size

Next we investigate the execution time with respect to the map size. Again, each result indicates the average among 50 tests, each using a different seed. The tests are run using 500 agents. For reference, the average path length is given in the same figure. Just as the execution time, it increases exponentially.

The resulting execution times are show in Figure 6.4. Because, as before, the *Merge nodes* optimization achieves the best execution time, and *Group nodes* is still better than basic Quad*, we combined them and tested them separately. Figure 6.5 compares A* to Quad* and the combined optimizations, and the speed-up of the latter two compared to A*. The results show that Quad* can be up to 100% faster than A*, and the fully optimized version up to 225%. However, we also see that Quad*, nor its optimizations, scale well with map size.
6.3.3 Effect of the block size

A third test explains why we use a block size of 16² for all other tests. The following tests are run using different block sizes on a map with size 1024² and containing 500 agents.

As you can see in Figure 6.6, the optimal block size is indeed 16². Both *Merged Quadtree* and *Heavy merged Quadtree* have an optimal block size of between 16² and 32². If we use smaller blocks, then the Quadnodes become too small and any advantage over A* is lost (i.e. covering large distances in a single step).
6.4 Quadtree construction time

This section investigates the Quadtree construction time with respect to the number of threads, the block size and the map size. Unless stated otherwise, all tests are run using a $1024^2$ world, $16^2$ blocks and 4 threads. Again, each test is done 50 times with different seeds, and the values given here are the averages.

6.4.1 Effect of the number of threads

The first test we did was checking the construction time with respect to the number of threads. The results can be seen in Figure 6.7. The blocks that are the root of multiple Quadtrees are on a queue from which the threads pop the next block. It is hence no surprise that the test results show that doubling the number of threads halves the construction time.

![Figure 6.7: Quadtree construction times with respect to the number of threads](image)

6.4.2 Effect of the block size

The second test increases the block size used. The results can be seen in Figure 6.8. The construction time increases exponentially with the block size. It shows that a block size of $8^2$ or $16^2$ is optimal.

6.4.3 Effect of the map size

The last test investigates the construction time when the map size changes. The results are shown in Figure 6.9. Again, the construction time increases nearly exponentially. Note that the merged Quadtree is constructed much faster than the standard Quadtree (about...
half the time using a $5120^2$ world). The reason is that connecting Quadnodes that are in different blocks is more time consuming than merging Quadnodes per block. When we use merged Quadnodes, there are less Quadnodes that need connecting to the neighbouring blocks, and hence execution is much faster. Using a heavily merged Quadtree, where we merge Quadnodes through blocks, the construction time is worse than the simple merged variant, but still faster than the standard version.

Note the excellent construction times, even with a map as large as $5120^2$. This might seem a lot, but remember that the Quadtree is only constructed once, and is only adapted during the running of the application, depending on changing obstacles (see Section 3.1.2).
6.5 Conclusion

In this chapter we tested the Quad* pathplanner and the Quadtree construction algorithm, as well as their different optimizations. We showed that the basic Quad* search achieves varying speed-ups of between 5 and 45% compared to A*. However, once we combine the grouping of nodes and the merging of Quadnodes in the Quadtree, we achieve speed-ups of up to 225%.
Chapter 7

Conclusion

The initial goal of this thesis was to create the base for an algorithm which is able to find optimal paths for a large number of agents in a real-time application, and in a dynamic environment. In order to do so, the algorithm was going to use as much parallelism as possible, extracting the most of what modern, common, dual- and quad-core CPUs have to offer, as well as the power of GPUs.

For this purpose, a new algorithm, Quad*, has been developed. It is an A* variant that is heavily dependent on a roadmap, which must be a Quadtree. We note that the initial goal of the thesis has not fully been reached. Although we did present a parallel algorithm for the construction of the Quadtree, the Quad* search in itself remains sequential.

In addition, there is no implementation which takes advantage of the massive parallelism offered by a GPU. We can only make an educated guess, based on the literature study and the CPU implementation, on the efficiency of the proposed algorithm if offloaded to the GPU.

Our test results show that the basic Quad* algorithm does not offer any significant speed-up compared to A*. The tests show that Quad* reduces execution time between 5 and 45% compared to A*.

However, once we take into consideration the optimizations consisting of merging the Quadnodes of the Quadtree into rectangular nodes and grouping the explored nodes per Quadnode, we do achieve a high speed-up compared to A*. This speed-up varies from 150 to 225%. This leads us to believe that there is a lot of potential in the road taken here to solve the problem of massive optimal pathplanning. We also refer to Chapter 8 for ideas about how to improve both the Quadtree and Quad* for which we did not have the time in this thesis.
Chapter 8

Future work

Parallelizing Quad*  The most important task left to investigate is the parallelization of the Quad* algorithm. At its current state, the algorithm is inherently sequential, but we believe there are a couple of ways to parallelize it, inspired by attempts by other researchers to parallelize A*.

One could imagine using a bi-directional search, where we start two Quad* searches per agent. The second search would look for a path from the goal to the start. Once the two searches meet, for example when they expand a node which is contained in a Quadnode which contains a node already expanded by the other search, we get a full path. We could go a step further, inspired by PRS, and start multiple searches starting in specific Quadnodes. In this case, we would need to search a path in the Quadtree itself first.

Another possibility consists in using a queue of Quadnodes. Each thread then pops a Quadnode from the queue and expands any nodes marked as open by any agent, and performs the expansion step of the node for each agent which marked it. New open nodes are marked and their Quadnodes are pushed on the queue. This would however require a lot of extra memory to store all the markings, and agents must be accessible by all threads. The latter is an important constraint when implementing on the GPU, which has a limited amount of space.

Using the Quadtree as standard roadmap  We discussed in detail why we do not use the Quadtree directly as we would any roadmap in Section 4.2.3. However, the test result show that the Quad* optimizations which should lead to non-optimal paths have the same average path lengths as the optimal paths. This leads us to believe that it might be interesting to investigate the amount of sub-optimality when using the Quadtree directly, instead of using the Quad* algorithm.
Increasing performance on large maps  The test results have shown that both Quad* and its fully optimized version do not scale well in the map size. If larger maps are a necessity, there is need for optimization of the algorithm in this area. As can be seen from the results, the *Best only* optimization achieves much better scaling in the map size, but it does not guarantee optimal paths.

Adding map abstraction  The presented Quad* algorithm performs at its best using a merged Quadtree, because its speed is highly dependant on the structure of the roadmap. It is therefore useful to increase the intelligence behind this structure. A possible optimization is adding map abstraction. If we would add some extra information to the Quadnode that describe the environment and/or reachability of the node, then the Quad* search might use this information to gain an additional speed-up. For example, we might analyse the Quadtree and mark in a Quadnode which neighbours lead only to a dead end, and which coordinates are hence reachable through these neighbours. This way, Quad* may skip entire corridors that lead to a dead end, when the goal location is not included.

Adding layers to the Quadtree  Another possibility in order to improve the structure of the overlaying roadmap is to add layers. What we understand by this is that we add a roadmap to the roadmap, using one or more levels, or layers. Each roadmap is an abstraction of the roadmap below, and greatly reduces the number of nodes that need to be searched. These abstractions will be able to quickly determine the reachable areas, and might even indicate among which paths of Quadnodes the fastest path runs. Higher levels make this harder, but improve the speed of the reachability tests. When combined with extra map abstractions, this might prove to be a powerful tool.

Reducing the number of searched nodes  During the discussion of Quad*’s optimizations, we used *Group nodes* in conjunction with *Merge nodes* for the best results. However, the *Best only* optimizations performs better than *Group nodes*. We did not choose to test the *Best only* optimization any further, because it does not guarantee optimality. However, we imagine that a similar idea can be used in order to increase the performance even further than *Group nodes*. For example, when we consider a node during the pathplanning, the algorithm could investigate the nodes of the previous block and execute some path smoothing.
Appendix A

Data structures

This appendix details the data structures used in the implementation used for the tests. All code has been implemented in C++.

A.1 Environment

The environment is represented by a graph. The graph is built using two classes, namely Node and Link. A Node contains its coordinates, a boolean that indicates whether it is part of an obstacle, and a list of Links. A Link contains the two Nodes to it connects, as well as the cost of the link. An example is given in Figure A.1.

A.2 Quadtree

The Quadtree uses the same structure as the environment (see above). However, a Quadnode inherits from Node by adding the size of the zone.

For the parallel version of the Quadtree, the environment is divided in blocks before building multiple small Quadtrees. These blocks are represented by the class SmallQuadTree, and contain all the Quadnodes inside of its sub-Quadtree.

Figure A.1: Interconnected nodes with link costs. The white nodes are free, the dark node represents (part of) an obstacle, and is hence blocked
A.3 Binairy heap with update functionality

For the open list, there is need for a data structure that can do the following operations as fast as possible:

- Get the element with the lowest priority and remove it.
- Know whether an element is contained.
- Insert a new element.
- Change the priority of an element.

We use a binairy min-heap to implement this functionality. The heap is implemented using a vector. Using a binary heap provides us with $O(\log n)$ insert and delete times, and keeps the elements sorted at all times. However, this does not allow us to perform the other two operations faster than $O(n)$.

We add, next to the vector, a map. The map takes as key an element, and has as value its index in the vector representation of the heap. The map is kept in sync at each operation of the heap. It allows us to implement the missing operations.

The “contains” operation uses the map.find function for $O(1)$ lookup of the element.

The insert operation first checks if the inserted element is already present using the map in $O(1)$ time. If it is contained, then the map immediately gives us the index in the vector. We change the priority of the element, and then rearrange it similarly to a standard insert, in $O(\log n)$ time.

The resulting data structure performs better than any standard data structure that C++ has to offer, because it is tailored specifically for the function as an open list.

![Diagram](image)

Figure A.2: An example Binairy heap with update functionality with characters as elements
Bibliography


