Master of Science in Engineering: Game and Software Engineering
June 2019

Combining Influence Maps and Potential Fields for AI Pathfinding

Filip Pentikäinen
Albin Sahlbom

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

Contact Information:
Author(s):
Filip Pentikäinen
E-mail: fipe14@student.bth.se

Albin Sahlbom
E-mail: alsk14@student.bth.se

University advisor:
Håkan Grahn
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet	:	www.bth.se
Phone	:	+46 455 38 50 00
Fax	:	+46 455 38 50 57

Abstract

Background. This thesis explores the combination of *influence maps* and *potential fields* in two novel pathfinding algorithms, *IM+PF* and *IM/PF*, that allows AI agents to intelligently navigate an environment. The novel algorithms are compared to two established pathfinding algorithms, *A** and *A*+PF*, in the real-time strategy (RTS) game *StarCraft 2*.
Objectives. The main focus of the thesis is to evaluate the pathfinding capabilities and real-time performance of the novel algorithms in comparison to the established pathfinding algorithms. Based on the results of the evaluation, general use cases of the novel algorithms are presented, as well as an assessment if the novel algorithms can be used in modern games.
Methods. The novel algorithms' pathfinding capabilities, as well as performance scalability, are compared to established pathfinding algorithms to evaluate the viability of the novel solutions. Several experiments are created, using *StarCraft 2*'s base game as a benchmarking tool, where various aspects of the algorithms are tested. The creation of *influence maps* and *potential fields* in real-time are highly parallelizable, and are therefore done in a GPGPU solution, to accurately assess all algorithms' real-time performance in a game environment.
Results. The experiments yield mixed results, showing better pathfinding and scalability performance by the novel algorithms in certain situations. Since the algorithms utilizing *potential fields* enable agents to inherently avoid and engage units in the environment, they have an advantage in experiments where such qualities are assessed. Similarly, *influence maps* enables agents to traverse the map more efficiently than simple A*, giving agents inherent advantages.
Conclusions. In certain use cases, where multiple agents require pathfinding to the same destination, creating a single *influence map* is more beneficial than generating separate A* paths for each agent. The main benefits of generating the *influence map*, compared to A*-based solutions, being the lower total compute time, more precise pathfinding and the possibility of pre-calculating the map.

Keywords: Pathfinding, Real-time performance, Influence map, Potential field, GPGPU

Sammanfattning

Bakgrund. Denna rapport utforskar kombinationen av *influence maps* och *potential fields* med två nya pathfinding algoritmer, *IM+PF* och *IM/PF*, som möjliggör intelligent navigation av AI agenter. De nya algoritmerna jämförs med två existerande pathfinding algoritmer, *A** och *A*+PF*, i realtidsstrategispelet *StarCraft 2*.
Syfte. Rapportens fokus är att utvärdera de nya algoritmernas pathfindingförmåga samt realtidsprestanda i förhållande till de två existerande algoritmerna, i sex olika experiment. Baserat på resultaten av experimenten kommer generella användningsområden för algoritmerna presenteras tillsammans med en bedömning om algoritmerna kan användas i moderna spel.
Metod. De fyra pathfindingalgoritmerna implementeras för att jämföra pathfindingförmåga och realtidsprestanda, för att dra slutsatser angående de nya algoritmernas livsduglighet. Med användningen av *StarCraft 2* som ett benchmarkingvertyg skapas sex experiment där olika aspekter av algoritmerna testas. Genereringen av *influence maps* och *potential fields* i realtid är ett arbete som kan parallelliseras, och därför implementeras en GPGPU-lösning för att få en meningsfull representation av realtidsprestandan av algoritmerna i en spelmiljö.
Resultat. Experimenten visar att de nya algoritmerna presterar bättre i både pathfindingförmåga och skalbarhet under vissa förhållanden. Algoritmerna som använder *potential fields* har en stor fördel gentemot simpel *A**, då agenterna kan naturligt undvika eller konfrontera enheter i miljön, vilket ger de algoritmerna stora fördelar i experiment där sådana förmågor utvärderas. *Influence maps* ger likväl egna fördelar gentemot *A**, då agenter som utnyttjar *influence maps* kan traversera världen mer effektivt.
Slutsatser. Under förhållanden då flera AI agenter ska traversera en värld till samma mål kan det vara förmånligt att skapa en *influence map*, jämfört med att generera individuella *A**-vägar till varje agent. De huvudsakliga fördelarna för de *influence map*-baserade algoritmerna är att de kräver lägre total beräkningstid och ger en mer exakt pathfinding, samt möjligheten att förberäkna *influence map*-texturen.

Nyckelord: Pathfinding, Realtidsprestanda, Influence map, Potential field, GPGPU

Acknowledgments

Thanks to Håkan Grahn for guiding us during this project and providing invaluable feedback to the project and report. Thanks to Johan Hagelbäck for support and guidance in the beginning of the project. Thanks to our classmates that have given suggestions and feedback on how we could improve upon our work. Thanks to Hans Tap for providing access to the room we have been using during the project.

We also want to thank Galgamarken Indian Food ”Pappa curry” for the good food and Nick Wedzinga for organizing ”Vindaloo Friday”. Lastly we want to thank Fritjof Cavallin for organizing fika every Friday.

Contents

Chapter 1
Introduction

One of the great challenges when designing artificial intelligence (AI) for games is navigation. AI units in most games need to traverse the game world and to do that a path needs to be plotted with the use of a *pathfinding algorithm*[16]. Pathfinding is a core component of most games today and provides many challenges for game AI developers.

A common approach for handling AI controlled units in games is letting each unit act as an individual *agent*. An agent is a system that acts autonomously and reacts to data given by its various environment sensors and input[14]. Agents can perform actions and react to changes in their environment based on the information gathered. Since the computational power of a computer dwarf those of a human, an AI controlled player can direct each unit on the map individually for each of the game's update cycles, by letting each unit acts as its own AI agent that decides its own movement based on a set goal, the unit's characteristics and the proximity of friendly and hostile units. The difficulty of this approach is determining where each unit should go to execute a command. There are many variations of pathfinding algorithms used in modern games, with their individual pros and cons[9] that make them suitable for different environments.

This thesis presents two novel pathfinding solutions that build upon two existing techniques, called *influence map* (*IM*) and *potential field* (*PF*). The two novel solutions are called, *IM+PF*[8][28] and *IM/PF*, which combine the use of influence maps and potential fields in novel algorithms. The two novel solutions are compared to two existing pathfinding algorithms, *A**[23] and *A*+PF*[20].

The aim of the thesis is to evaluate the novel algorithms compared to two established pathfinding solutions and to study how well the novel solutions performs in terms of pathfinding capabilities as well as performance scalability. Two research questions are formulated that reflect these goals:

RQ1 How does the *IM+PF* and *IM/PF* algorithms perform in terms of agent combat, movement and enemy avoidance compared with established algorithms?

RQ2 How does the *IM+PF* and *IM/PF* algorithms scale compared with established algorithms, in terms of frame time and memory utilization, when increasing the unit count?

The algorithms are evaluated in various experiments that test their basic functionality, such as finding the shortest path or avoiding obstacles. Variations of these experiments are used with a varying number of AI agents to evaluate the real-time performance and scalability of the algorithms.

The pathfinding solutions are implemented and tested through the use of *Blizzard Entertainment's open source StarCraft 2 toolset* C++ API (s2client-api)[6]. The SC2 API is chosen for its versatility and accessibility. However, the objective is not to create an algorithm specific to StarCraft 2.

The aim is to evaluate the algorithm's viability compared to established pathfinding algorithms in a general real-time strategy (RTS) game environment. Therefore some StarCraft 2 specific problems and mechanics are excluded from the experiments. The experiments don't measure performance with flying units, the interaction between air and ground units or terrain height difference. Some aspects of the various pathfinding algorithms are GPU-accelerated, through the use of *GPGPU* (*General-purpose computing on graphics processing units*), using the parallel computing platform *CUDA*. CUDA places restrictions on what hardware the solution can run on, limiting the solution's compatibility to Nvidia GPUs.

To evaluate the novel solution, six experiments are performed where the implemented pathfinding algorithms are evaluated against each other. The experiments are performed in various environments and in different conditions to assess the different aspects of the pathfinding algorithms. The different environments are constructed so that the algorithms can, in a clear and controlled way, demonstrate their capabilities. To evaluate the performance of the algorithms, measurements are taken. Metrics such as "distance travelled" and "damage taken" are evaluate and compared between the algorithms to measure pathfinding capabilities. Frame time and memory utilization is measured to evaluate how the algorithms impact the game when utilized.

The results show a large discrepancy in both pathfinding performance and system utilization, depending on the experiment. It becomes clear that the proposed solution has use cases where it heavily outperforms the other algorithms, both in terms of pathfinding capability and compute time performance, but falls short in other areas.

Chapter 2

Background

The novel pathfinding algorithm presented in this thesis work is based on several existing pathfinding solutions and implemented and tested in the *real-time strategy* (RTS) game StarCraft 2.

2.1 StarCraft 2

StarCraft 2 is the second installment in the *StarCraft* RTS game series created by Blizzard Entertainment. The first installment was released in 1998 and was named *StarCraft*. *StarCraft 2: Wings of Liberty* is the first game in the second installment of the game series, and was released in 2010. Later in 2013 and 2015, there were two expansion packs released for the game, named *StarCraft II: Heart of the Swarm* and *StarCraft II: Legacy of the Void*. The base game *StarCraft 2: Wings of Liberty* became free to play on the 14Th of November 2017[5].

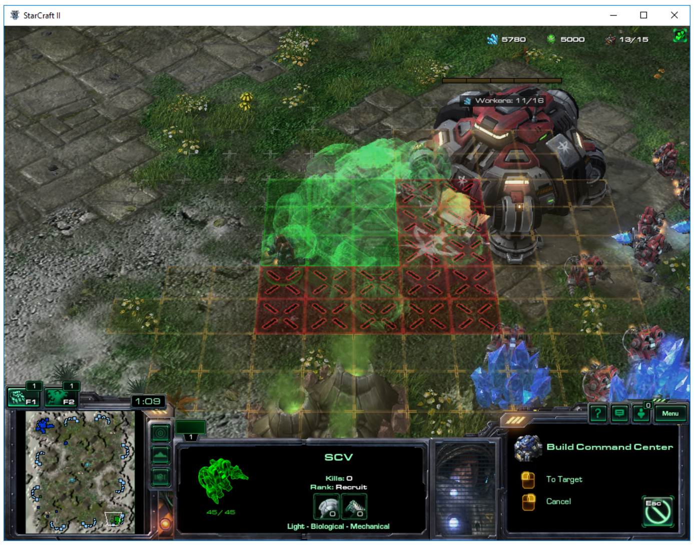


Figure 2.1: Build grid representation in *StarCraft 2*, showing tiles as green, red or neutral depending on occupancy and player interaction.

StarCraft 2 is an RTS game, where the aim is to build an army, composed of various units, and destroy an opponent's base. There are many different unit types available for a player to build, all with different abilities, weapons, movement speed and more. There are also "melee" units in the game, who can only attack other units in arms reach to them. The game uses a grid system, see figure 2.1, to indicate space availability and for constructing buildings. A similar approach is implemented for the units in the game. They use a grid of triangles that is much smaller than the build grid to navigate through the game world.

In August of 2017, Blizzard Entertainment released the StarCraft 2 Client API (s2client-api)[6]. The Client API allows for custom, offline matches between human players or computer-controlled bots. The API enables a controlled environment where game properties, map attributes and run-time variables can be customized to create experiments where various aspects of the pathfinding algorithms can be evaluated.

2.2 A*

The search algorithm *A**[23] is a commonly used pathfinding algorithm. The algorithm traverses nodes in a best-first search manner, meaning that the algorithm chooses to expand an available node that has the shortest (straight-line) distance to the goal node (h(n)) added together with the cost from the start node to the node that is desired to expand (g(n)). Resulting in an expansion of the node with the lowest score when adding h(n) and g(n) together:

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

This behavior gives *A** the ability to stop exploring a path down the ”line” when the f(n) for that node is no longer the lowest or if the path followed results in a dead end. *A** is very popular because of its effective search algorithm when traversing a grid world, which most strategy games' maps are built of. Figure 2.2 gives a visual representation of how the path of the algorithm can look.

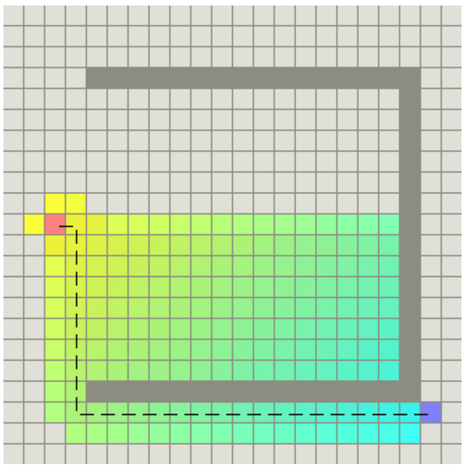


Figure 2.2: An illustration of the explored nodes and the final solution to *A** pathfinding in a grid from the start node (red) to the destination (blue)[7]. The dark grids represent a wall that cannot be traversed.

The A* algorithms mainly work with two lists, the ”open” and ”closed” list. The closed list contains all the nodes that have been expanded and can possibly create a path to the goal node while the open list contains all the nodes that can be expanded.

In each step a node from the open list is expanded based on the value $f(n)$, the node's valid traversable neighbours are placed in the open list, and the expanded node is moved to the closed list. In figure 2.3, the yellow nodes are in the closed list and their adjacent nodes that have not been expanded are green and they are in the open list.

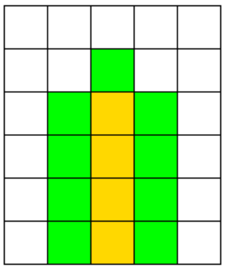


Figure 2.3: An illustration of the available (green) nodes and closed (yellow) nodes.

Several high performing custom StarCraft AI bots[21] use $A*$ exclusively as a pathfinder for their units, but several developers have chosen to integrate the use of *potential fields* into their pathfinding algorithms.

2.3 Potential Field

Potential fields (PF)[19][28] are used to determine if and how interesting an object in the game world is. This is done by giving it a positive or negative charge which influence the world around it. The origin of this technique comes from how magnetic fields are formed around electrons and protons, forming a field uniformly around the object. With this method, things like obstacles or dangers can be given a negative charge to indicate that agents should not go near them and an enemy agent can be given a positive charge to indicate that agents should go towards them. A charge's field will expand outwards depending on how strong the charge is.

One disadvantage of potential fields is the local optima problem, where an agent might reach a point where the grids neighbors have worse values than the current grid, but the goal has not been reached, see figure 2.4. The figure shows that the agent (represented in green) can get stuck behind the brown obstacle when traveling to E, and not find a path around it, due to the local optima problem inherently present in PF solutions. There are several optimizations that can be implemented to limit the issues created by local optima, such as *pheromone trails*[22], but none can completely solve the problem.

2.4 Influence Map

An *influence map* (IM)[8][35] works similarly to a PF. In an influence map grid, charges are placed which allows the grid coordinates to acquire values based on their distance to the charge, but unlike the PF approach, blocking objects in the grid (such as walls) hinders the field (see figure 2.5). This effect is achieved by calculating each

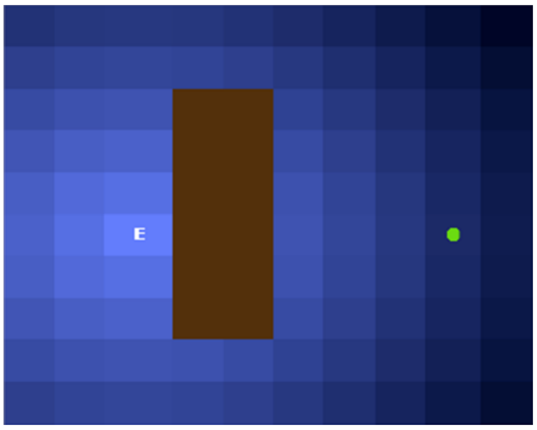


Figure 2.4: Potential Field, illustration of how a single charge affects a grid.[19] Notice how the wall does not affect the field generated by the charge. A low value in the PF corresponds to a more light area.

grid closest ”walking path” distance to the charge source, through the use of the A* pathfinding algorithm (see section 2.2)

This means that potential fields and influence maps are similar, but can be used in very different ways. IMs are typically used for pathfinding toward or away from specific goals, because of their ability to reliably pathfind through environments with impassable terrain. A PF could not be used in such an environment since the algorithm can’t reliably navigate an environment with blocking obstacles. PFs are instead typically used when representing objects to attract or repel agents, such as a game’s bonus points or enemy units.

Generating IMs is computationally expensive since every node in a grid needs to perform its own pathfinding to the destination through the terrain. For calculating each path in the IM to a destination, *A** is traditionally used, due to its performance and accuracy[36]. PFs, on the other hand, is computationally cheap to generate since every node simply calculates the distance from the node to the charge (with no regard to terrain), making them ideal to represent dynamic objects in an environment that needs to be updated often.

2.5 Parallel Computing

General-purpose computing on graphics processing units (GPGPU) is the use of a GPU, also called *device*, to perform general compute tasks typically performed by a CPU, called the *host*. A GPU has many more cores than a CPU, but they are less powerful, and as such the GPU excels at large workloads which can be done in parallel[37]. The GPU also utilizes a ”throughput oriented design”, compared to the CPU’s ”latency oriented design”. This means that the GPU is designed to handle large amounts of data, but at a higher latency compared to the CPU. The CPU achieves low latency by utilizing several layers of cache memory to access data at a high rate, minimizing the time the CPU is forced to wait for data being fetched in memory[27]. Modern CPUs also have multiple cores, all acting as individual processing units that

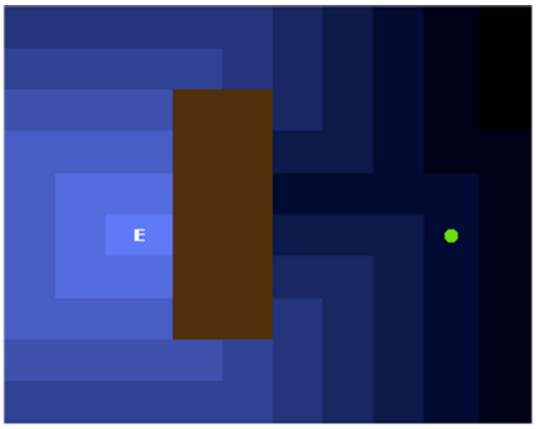


Figure 2.5: Influence Map, illustration of how a charge affects a grid.[19] Notice how the wall hinders the field's expansion. A low value in the IM corresponds to a more light area.

can handle different tasks. Modern consumer grade CPUs often have 4-6 cores[25]. This makes the CPU ideal for computing small, varied and sequential tasks.

To compensate for the lack of speed in the GPU, it's equipped with many more cores[31] and a memory fetching technique designed to transfer large sequential memory chunks simultaneously. Modern consumer grade GPUs have around 2000 cores, divided into groups of 32, that excel at performing similar tasks with varying data input and output location. This behaviour is named *Single Instruction Multiple Data* (SIMD), which allows the GPU to effectively execute parallelizable tasks. Each group of 32 cores is part of a *streaming multiprocessor* (SMP), and perform the same tasks in parallel. These GPU cores are much less powerful compared to their CPU counterparts, but when working together will exhibit much greater compute performance when utilized correctly in appropriate workloads.

The programming language *CUDA* is used to apply parallel computing techniques in the implementation. CUDA enables low-level GPU programming, host-device intercommunication and device code execution, and is run asynchronously during runtime along with the host process.

2.6 Related Work

The main objective of this thesis is to combine IM and PF to create a novel pathfinding solution. Due to the novelty of the solution, there is little to no work done on the subject. However, there are many research papers that address the subject of pathfinding exclusively[26][30][11], both with and without charge-based maps, but which we feel have little relevance to our work that is not covered by the related work listed later in this section. Similarly, there is a lot of work done in the field of GPGPU that has some relevance to our work[3][12][24][17], but most works, besides those listed later in this section, have little relevance to the combination or implementation of IMs or PFs on the GPU.

The paper "A*-based Pathfinding in Modern Computer Games" by Cui and Shi[9] evaluates the A* algorithm and how the game world can be represented for the A* algorithm. The paper also discusses different heuristic functions and their impact on how the nodes are explored to generate a path. The paper evaluates the A* algorithm in two games, Age of Empires II and Civilization V, both games used A* for unit pathfinding. They concluded that A* is one of the best pathfinding algorithms for fast and precis pathfinding in games, but they also mention that when the number of units using A* reaches in the hundreds and thousands of units, it can become hard to provide optimal paths in real time for all the units. The article provides a good base and a few suggestions of how A* can be implemented and optimized, and how the world can be represented to suit the needs of the game.

One of the articles that have implemented A* in combination with PFs (*A*+PF*) is one by Hagelbäck[20]. Hagelbäck uses traditional A* pathfinding but switches to a PF-based solution when enemies are nearby. When the agents are utilizing the PF they will either avoid or attack enemies represented in the field, depending on its "state" (offensive, defensive), and properties (can/can't attack). The *A*+PF* pathfinding solution showed to yield a great benefit over StarCraft's original pathfinding implementation.

Hagelbäck has also published an article where the combining of *A** and *Boids algorithm*[21] is presented. Boids is an algorithm that allows agents to recreate the flocking behaviour of birds and relies on 3 core rules to function; *separation*, *alignment* and *cohesion*. Hagelbäck's implementation also included a *goal* rule and several variations of *separation*-functions to adjust to the StarCraft gameplay. An approach similar to the previous implementation is used, where *A** handles movement until enemies are detected, in which case the Boids algorithm is solely utilized. The article then compares the implementation with his previous *A*+PF* solution, and concludes that *A*+Boids* has a 45% win rate when fighting against the *A*+PF* bot. However, the *A*+Boids* bot requires on average less than 2% of the compute time of the *A*+PF* algorithm.

An article written by Gomes and Amador describes the use of potential fields (incorrectly referred to as *influence maps* in their article, they are using PFs) and *A** in a single pathfinding solution[1]. They call the algorithm *Xtrek*, and it is used to calculate the optimal path through attracting and repelling fields in a node graph. The algorithm excels at finding the optimal pathfinding solution for individual units through an environment but struggles if many dynamic objects are present in the environment. Since the algorithm is not adapted to handle many updates to the potential field map, it is not suitable for RTS games, like StarCraft 2, where there are many units moving around and resulting in many updates to the potential field map in a short time.

Bergsma and Spronck[18] utilize IMs for adaptive spacial reasoning in turn-based strategy (RTS) games. Bergsma and Spronck propose a game AI architecture named ADAPTA (Allocation and Decompostion Architecture for Performing Tactical AI). They use machine learning to train their AIs combat behaviour. For the combat behaviour they implore the use of IMs to represent the state of the game so it can be used for spatial reasoning. The IMs are used to determine if the units under the AIs command are supposed to move or attack. The IMs are calculated depending on two functions, a distance function and a propagation function. The distance function as described in their paper can be a general distance metric, Euclidean or Manhattan distance, or it can be a domain-specific function. Their conclusion states that the AI was able to learn and counter different tactics and play equally well or better than AI that utilize static strategies.

In an project by Elmir[10], generation of PFs are done in a GPGPU solution, and then compared to a generic CPU implementation in terms of real-time performance. Two versions of PF generation are implemented on the CPU, one naive version and another with small optimizations to speed up the generation. Three GPGPU PF generation versions are created, the first being a simple implementation with no performance optimizations, second with a basic *memory bursting* optimization, and a third version utilizing *memory bursting* as well as *shared memory* to store numerous potential field entities to minimize global memory reads. The various algorithms are tested in a benchmarking environment where the texture size, PF entity charge radius and number of PF entities are varied. The results show the optimized CPU version having the lowest frame time in small maps with low amounts of PF entities, but as the workload increases, the third GPU implementation with both optimization techniques outperforms the CPU version significantly.

The previous studies presented have shown the advantages and disadvantages of both potential fields and influence maps. The proposed solution is theorized to take the advantages from both while countering some of their disadvantages.

Chapter 3

Proposed Solution, IM+PF

The proposed solution is a combination of the existing *influence map* and *potential field* algorithms. The algorithms work in tandem to compensate for each others shortcomings while utilizing their advantages.

3.1 Algorithm Description

The proposed solution is a combination of *influence maps* and *potential fields* (IM+PF) to create a novel pathfinding algorithm that does not require a hard switch between different algorithms, but retains the advantages of both *A** and PF. The IM is used as a pre-calculated *A** to a destination that can be used by any agent traveling to that position. Several potential fields are generated that represent offensive and defensive positioning for agents. These PFs are not used for pathfinding to a set destination, but rather to attract or repel agents from a position. The agents always use the IM to determine their path through the map and can, depending on the ”state” the agent is in, add the PFs to its evaluation of the path. The agents can be in a passive, defensive or aggressive state. Since the agents have access to the path to their destination, enemy range zones and its own attack vectors simultaneously, they can attack or avoid enemy units while simultaneously moving towards their goal.

The *IM+PF* algorithm can be altered to perform a ”hards switch” from IM to PF during combat, disregarding the values from the IM to only use the PF. This variant is called *IM/PF*. The *IM/PF* solution is more proficient at avoiding enemies due to the fact that it does not attempt to move agents towards the destination when an enemy is close enough, resulting in a more ”aggressive” avoidance. The same principle applies when attacking, the agents doesn’t attempt to reach the goal when attacking enemy units, and will therefore be more attracted to the designated range of the PF. The ”hard switch” does not impact performance because the only change to the *IM+PF* algorithm is a simple switch to the agent’s logic that will check if the PF is above or below a certain threshold, triggering the switch.

A big advantage of the proposed pathfinding solutions is that the computationally heavy calculations, the generation of IM and PF maps, are highly parallelizable, and can therefore be done by a GPGPU solution[4]. Generating the PFs and IMs is fully synchronous, meaning all computations done per grid node (or map coordinate) can be done individually and in parallel with no need to synchronize between nodes. Because game maps can be quite large, utilizing a GPGPU solution to calculate the IMs and PFs requires less compute time than if the generations are done on the CPU.

The agents themselves perform very simple operations. Because of this, the algorithm only suffers a slight performance reduction when more agents are added to the game. Instead, as long as the agents can share PFs (if they are the same unit type), the only additional computation is a minimal increase to the generation of the PFs. Since all agents share IMs, more units walking to the same destination has a minimal impact on performance. When a new destination is required that no other agent has as its destination, a new IM is generated that is accessible by other agents. Additionally, when using an IM the game world can be traversed diagonally, because every point on the map have been calculated with a distance to the goal.

The hypothesis is that the *IM+PF* pathfinding algorithm will be a viable option among established pathfinding solutions. The most significant advantage of the *IM+PF* algorithm is that the agents always have access to the closest route to their respective destination, through the use of IMs, while also having a map of any close dangers, through the use of PFs. This allows the agents to intelligently avoid dynamic objects in the environment while simultaneously moving towards a goal. A visual representation of the data available to the agents utilizing *IM+PF* and *IM/PF* can be seen in figure 3.1.

Advantages of IM+PF:

1. Generating the IMs and PFs can be performed on the GPU. Utilizing the heterogeneous computing between the GPU and CPU enables the CPU to perform other tasks while the GPU generates the IM and PF maps[29].

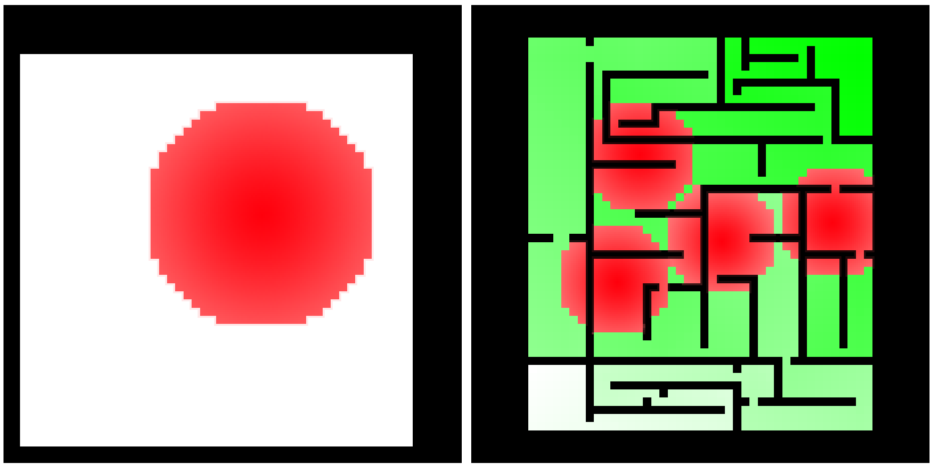
2. Multiple agents can utilize the same IM and PF for pathfinding. Sharing resources between agents reduces redundant computation, reduces system load and allows the *IM+PF* algorithm to scale better when increasing the number of agents in the same environment.

3. *IM+PF* handles dynamic objects in real-time. Enabling agents to avoid or go to objects of interest, such as enemies or friends, while traveling to a destination. Other pathfinding algorithms, such as *A**, need to constantly set new exact destinations when encountering dynamic objects that need to be avoided.

4. There is no hard switch between different pathfinding algorithms depending on the agent's situation. Agents using *IM+PF* can handle pathfinding and avoidance or combat simultaneously.

Evaluating IM and PF separately in terms of their pathfinding capabilities, their respective problems and limitations become clear. Only using an IM has some of the same drawbacks as *A**. IM, like *A**, has to update, or completely recalculate, an IM if a dynamic object is introduced to the map or if parts of the terrain moves. This causes performance degradation, due to the time-consuming calculations required to create an IM.

Only using PF results in a pathfinding algorithm that is unable to handle an environment that isn't simple, due to how PFs are designed. PFs, as described earlier, won't take static terrain into consideration when creating the field, as illustrated in figure 2.4. Some changes can be made to how PFs are generated, such as giving charges to static objects in the environment, such as walls, but the result still won't be able to handle complex terrains such as mazes or similar environments.



(a) The repelling *PF* (red circle) generated by a marine. (b) IM with PFs, Labyrinth Hard.

Figure 3.1: Visualizations of PF and PF on IM.

3.2 Compute Time Optimizations

Several optimization techniques are implemented to the GPGPU solution to increase compute time performance and memory utilization.

3.2.1 Shared Memory Used as Local Intermediate Buffers

When generating the IMs on the GPU, the search algorithm A* is used. Since A* of each thread in a thread group can choose to expand vastly different nodes in the map, because of difference in start position, map terrain and node prioritization, there is not much use in sharing node data in a group[33][32]. So instead the *shared memory* (memory accessible from all threads in a work group) is divided into chunks and allocated to each thread to be used as an intermediate buffer to global memory.

When performing the A* search algorithm on the GPU, the intermediate buffer is used to store elements in the closed and open list. These elements can then be moved simultaneously on all threads in a work group to global memory (where the majority of the open and closed list is stored) at specific intervals. This frees up the SMP to handle other work groups during the larger transfer and promotes *memory bursting*.

3.2.2 Minimizing Shared Memory Bank Conflicts

Shared memory is divided into separate memory banks[32]. Accessing multiple variables in separate banks results in a simultaneous transfer of the variables. However, accessing multiple variables in the same bank results in a serialized retrieval operation. Because of this the local intermediate buffers of shared memory (described in section 3.2.1) are allocated to span multiple banks, interleaving other work groups' memory in the same SMP. This does require more processing during memory allocation, but every shared-to-global memory transfer benefits from simultaneous transfers to some degree.

3.2.3 Device Lookup Table

Transferring data between VRAM and RAM is quite slow, because of the limitations of the PCI-bus[32]. Because of this it is most often important to limit the data being transferred between the host and device. Static information about unit types, such as range and movement speed, does not need to be transferred for each unit. Instead, a lookup table is created on the device, which is stored in constant memory, that is accessible with the same speed as registers (after the first access). This means that less data needs to be transferred over the PCI-bus, only the unit type, position and faction owner are sent to the GPU per-frame, and the rest of the required information is readily available for quick access in the device.

3.2.4 Changing Terrain Map In-device

As mentioned in section 3.2.3, transferring data between the host and device is slow. Because of this it is not effective to transfer an entirely new terrain map to the device when a small portion of the map is altered. Instead, a kernel launch is invoked that changes the map in device memory. This has low impact in small environments, but has a great performance impact on larger maps.

3.2.5 Device Job Queue System

When multiple kernel launches are executed in succession in the same *stream*, the launches are placed in a queue[33][32]. Each device job is divided into multiple sub-tasks that each SMP can execute, based on group size. These sub-tasks are not guaranteed to execute in order, or even, before a subsequent job. After an SMP is finished with a sub-task, and there are no more sub-tasks of the same device job available, the SMP will be assigned a sub-task of another device job, even though other SMPs on the device are still working on the original job. Because of this, the order of device jobs heavily impacts the performance of the solution.

Since *IM+PF* and *IM/PF* both create IMs, which are quite expensive to compute, it is important that other device jobs can execute in between the generation of multiple IMs. Since the experiment is running in a game environment, the game's *frame* (the pictures presented to the screen) must be rendered on the GPU to allow the game to progress. Because of this, a queue system is implemented to allow an intelligent sequence of kernel launches. There are 2 queues, one with PFs and one with IMs. For each frame, a maximum of 1 IM, 1 repelling PF and 5 attracting PFs are allowed to be generated per game loop iteration before the game frame output is rendered. The ”magic number” of 5 attracting PFs per game loop is chosen because they produce optimal frame time performance on the *System under test* listed later in section 4.2.

3.2.6 Asynchronous Runtime

The CPU and GPU can perform jobs in parallel, and to maximize runtime performance both compute units should be fully utilized simultaneously. It is important that as many jobs as possible are queued up to be run over the game loop iteration without impacting the frame rendering.

To achieve this, the CUDA program is implemented to run fully asynchronously, with no forced device-and-host barrier synchronization. A fixed number of device jobs are queued up to be run over a game loop, as described in section 3.2.5, and the results from the jobs are retrieved (if the job has complete) when control is returned to the queue system. Simultaneously as the transfer of data from the device to the host is ongoing, the next frame's device jobs are started, which constantly keeps the device occupied.

Chapter 4
Method

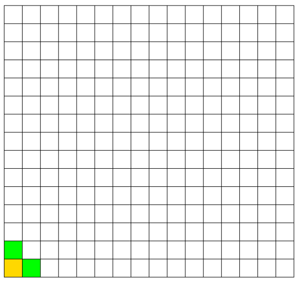
Blizzard Entertainment's open source StarCraft 2 toolset C++ API (s2client-api)[6] is used to gather map- and unit information from the game world. The data is used to generate influence maps and potential fields for the agents. To generate the influence maps and potential fields, *CUDA* is used because the workload can be parallelized on the GPU. The StarCraft 2 client API can be run in two different modes, asynchronously or sequentially. If the solution is run asynchronously, it means that the game and the bot are run separately, and that the bot requests updates to the game at it's own pace. Running the solution sequentially means that the game and the bot client are never run at the same time, halting the game while the bot is performing it's operation and vice versa. For all experiments the system is set to run sequentially. Even though the game is stepped forward by the API each frame, the game is not deterministic and the same experiment setup can result in different outcomes every run. This is probably caused by fluctuations in frame time and GPU floating point inaccuracy. Because of this all experiments are executed multiple times and averaged to get a better evaluation of the experiment.

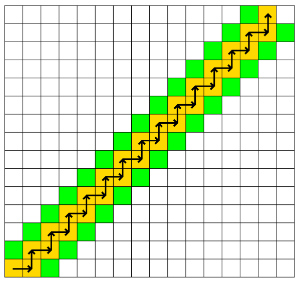
During some of the experiments distance traveled is measured. Due to StarCraft not defining exactly how long 1 distance is except for that the tiles in the game is 1 distance wide or high. When measuring the units walk distance the result can therefore be quantified to how many tiles or parts of a tile they have walked.

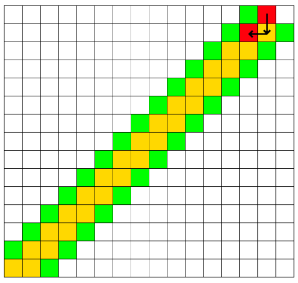
4.1 A* Implementation

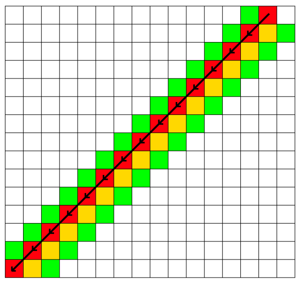
The A* algorithm can be modified when implemented to suit specific needs of the game. StarCraft 2s' units are able to traverse the world diagonally, but the basic implementation of A* does not handle diagonal traversal and can only walk to adjacent nodes that are above, blow, left or right. However, the behavior of moving to diagonally adjacent nodes can be achieved with a slight modification of the algorithm.

The modification is done to the backtracking step. The expanding of the nodes and putting them in the open list is still the same as the standard implementations as visualized in figure 4.1b. In this figure the start is in the bottom left corner and the goal is in the top right. The algorithm starts by putting the first node in the closed list, represented by yellow, and then expands the adjacent nodes and puts them in the open list, represented by green, the node above and to the right, see figure 4.1a. This will be repeated until the goal is reached. When the goal node is reached, backtracking will be performed and is where the algorithm is modified to enable diagonal movement. The finished path can be observed in figure 4.1d, as the red path.


(a) Expansion of the first node.


(b) A* expanding nodes and reaching the goal.


(c) Finding the second node to be backtracked to.


(d) A* performing back-track to create the best path possible, among the expanded nodes.

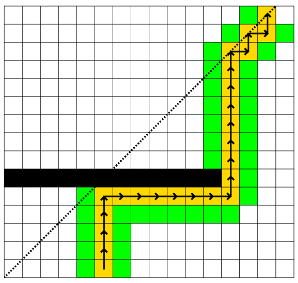

(e) A* pathfinding with a wall on the map.

Figure 4.1: Illustration of how A* works with expanding the nodes and backtracking to create a path.

The first step of the backtracking can be seen in figure 4.1c. The backtracking starts by taking the goal node and puts it in a list that will create the final path, the nodes in this list is represented by red in this figure. Then the algorithm looks at the parent of the current node, the arrow going from the red node to the yellow node. The algorithm then looks at the parent of that node, the arrow going from the yellow node to the red, and evaluates if that node is diagonally adjacent from the first red node. If it is, the node is placed in the list representing the final path. Then these steps are repeated until a path has been created from the goal node to the start.

If the algorithm tries to find a path from a point below or above the map's diagonal, as seen in figure 4.1e. The algorithm will not be able to find a diagonal path, because comparing the node above and to the right of the current node yields a slightly smaller value for the node above, in the case where the nodes are blow the diagonal of the map. The last steps show what happens when the algorithm makes it to the diagonal.

4.2 System Under Test (SUT)

All experiments are performed on the same system.

- Intel i7-8700K 6 Core @ 3.900 GHz w. Hyper-threading
- EVGA GeForce GTX 1080 @ 1830 MHz
- 32 GB DDR4 RAM @ 3000 MHz
- Windows 10 64 bit, version 1809
- CUDA 10.1
- The C++17 standard, compiled using Visual C++ 2017
- StarCraft II patch 4.8.3, version 72282
  - Ultra graphics preset
  - Windowed mode, 800x600 resolution

4.3 Pathfinding Experiments

The quantitative study consists of three experiments where quantitative values are measured. The values are described and explained under each experiment in this section. The three experiments that are conducted relates to:

1. Combat
2. Moving from point A to point B
3. Avoiding enemy units

These three experiments are used to answer RQ1. The experiments are performed on custom maps created in the StarCraft 2 Editor, to ensure a controlled environment.

4.3.1 Experiment Maps

A total of four maps are used for the experiments.

1. Empty50, figure A.1 in appendix A
2. Labyrinth easy, figure A.2 in appendix A
3. Labyrinth medium, figure A.3 in appendix A
4. Labyrinth hard, figure A.4 in appendix A

All maps are created by hand, however they are designed to be as generic as possible, to limit any advantage an algorithm may have in specific terrain. The maps are created by hand and not by a random generation program, because the maps must be solvable. Due to experiment 3 this can't be guaranteed if the maps are randomly generated. Empty50 is an empty map with no obstacles and with a width and height of 50 units. The labyrinth maps are classified with easy, medium and hard, to indicate the level of complexity. The complexity is reflected in the potential distance of the longest path.

4.3.2 Units Used In Experiments

Different units are used in experiment 4, 3 ranged units and 1 melee unit. The 4 units and their information are listed in table 4.1. The different units are chosen because of their diversity, to represent different scenarios. The marine is an all around average ranged unit that represents the common use case. The ghost has a long range, high speed and low damage. The roach is a unit with many hitpoints, high damage and low speed. Lastly, the zealot is a melee unit with high damage and hitpoints.

	Marine	Ghost	Roach	Zealot
Hitpoints	45	100	145	100
Shield	0	0	0	50
Armor:	0	0	1	1
Damage:	6	10	16	16
Weapon Cooldown:	0.61	1.07	1.43	1.26
Weapon Range:	9	11	4	0.1
Speed:	3.15	4.34	3.15	3.15
Sight:	9	11	9	9

Table 4.1: Tables of statistics for StarCraft II ranged and melee units which are utilized in the experiments.

4.3.3 Experiment 1, Combat

Experiment 1 is only performed on map Empty50. A fixed number of units are created for two factions. The agents then attack the units of the opposing faction while only utilize the pathfinding algorithm in combat. One faction utilizes *A**, *A*+PF*, *IM+PF* or *IM/PF* while the opposing units utilize StarCraft 2's built in pathfinding algorithm. This is done to evaluate the performance of each listed pathfinding algorithm against StarCraft 2's built in pathfinding algorithm. The experiment evaluates the algorithms in combat between different types of units, especially units with different attack ranges but also units with different movement speed. Units that can attack other units from a distance are called ranged units and units that can only attack units that are in extremely close proximity (arms reach) are called melee units. The specific setups that are performed for this experiment are:

(The units listed on the left side are the agents that utilize the implemented pathfinding algorithms and the units listed on the right use StarCraft 2's built-in pathfinding)

1. Ranged vs Ranged
   (a) Marine vs Marine
   Same range and movement speed.
   (b) Ghost vs Roach
   Range and movement speed in favor of Ghost over Roach.
   (c) Roach vs Ghost
   Range and movement speed in favor of Ghost over Roach.

2. Ranged vs Melee
    (a) Marine vs Zealot
    Range in favor for Marine, same movement speed.
3. Melee vs Melee
    (a) Zealot vs Zealot
    Same range and movement speed.

The metrics that are evaluated for this experiment are:

1. Damage done
2. Units killed
3. Damage taken
4. Units died

Only the number of units killed in the experiment is needed to calculate how many times the algorithms won or lost. The four metrics are taken to give a more precis evaluation. If two or more algorithms get the same amount of wins, the average damage done can be evaluated. The resulting graphs shows how many times each algorithm won in every setup.

The ”Ranged vs Ranged” case is performed three times to cover all of the basic cases; units have same range (marine vs marine) and units have longer or shorter range (ghost vs roach and roach vs ghost). This is done so that the tested algorithms have to perform the experiment for the different unit range cases and so that a more in-depth analysis can be made about combat performance.

The low number of tests for ”Ranged vs Melee” is due to the low number of melee units in the game and performing more tests with that type of unit won't yield varied results. The same reasoning and conditions apply to ”Melee vs Melee”. The only difference that would be seen is that different units have different attack damage, and the pathfinding algorithms would not affect the outcome.

The test case ”Melee vs Ranged” is not evaluated due to that case would not yield any valuable results. The melee agents would be able to utilize the PFs to be able to avoid friendly units and split up when attacking. But these behaviour are already displayed in the ”Melee vs Melee” test case.

4.3.4 Experiment 2, Pathfinding

Experiment 2 is performed on all four maps. This experiment is exclusively testing the various algorithms' ability to find a path to the given destination. Therefore the distance traveled between a point A and a point B is the only metric in this experiment.

On all four maps, an agent is spawned in the bottom left corner on the maps and is tasked to find the way to the top right corner of the maps. No other units, friendly or hostile, are present on the map, so the only obstacles are the map's walls or borders.

The *marine* unit is used in the experiment. Only one type of unit is required because the only difference between units is their speed and size. A unit's size can impact it's ability to traverse the map, and because of that, one of the game's smaller units were chosen to perform the experiment. Performing the experiment with larger units would not change the performance of the algorithm, it would only introduce potential problems specific to the StarCraft 2 game that could impact the results.

4.3.5 Experiment 3, Enemy Avoidance

Experiment 3 is performed on all four maps. This experiment is similar in setup and conditions as experiment 2, except enemy units are present on the map that the agents should avoid. The agents are still starting in the bottom left corner and have a destination in the top right corner. The metrics that are evaluated for this experiment are:

1. Damage taken
2. Distance

However, the metrics must be analyzed as a whole. For example, an agent using an algorithm that receives no damage but travels ten times longer than an agent using another algorithm that only took a small amount of damage is not considered better. A low score in damage taken and distance traveled is a desirable result.

4.4 Scalability Experiments

To test the performance and scalability of the algorithms, and thereby answer RQ2, similar experiments as those listed in section 4.3 are performed with varying number of agents while measuring frame time and memory usage. RAM and VRAM are measured separately. From this data, performance trends for each experiment with varying number of entities are generated. The trends from each algorithm's experiment are then evaluated to determine the difference in real-time performance and scaling between the pathfinding algorithms.

All algorithms utilize StarCraft specific functionality (such as moving across the map) in the same way, so the difference in result will be as close to the tangible difference in real-time performance and scalability between the algorithms as possible.

During the initial frame(s) of the experiments, the algorithms calculate their individual pathfinding. Because of this, all frame time data generated from the experiments are segregated into "*path calculation frames*" and "*average frame time*". This is done to highlight the algorithms' significant frame time increase during the calculation of the paths and because presenting the data as only one averaged value, that is not segregated, would not yield any data that could be properly analyzed. The *path calculation frames* are also excluded from the *average frame time*, due to the large impact it would have on the *average frame time*.

Since *IM+PF* and *IM/PF* both utilize the same algorithms, perform the same workload and have access to the same data, the only difference is a small modification in the agents' logic, only *IM+PF* is tested. All results relating to *IM+PF* is applicable to *IM/PF*.

4.4.1 Experiment 4, Combat, Single Unit Type

In experiment 4, varying number of agents advance across an empty map to engage hostile units. A total of 1000 frames are recorded per experiment run. Each experiment iteration increases the number of agents by 10 and the experiment reaches a maximum of 250 units. This experiment is used to determine the cost of calculating a path over a simple environment, as well as the cost of generating in-combat resources (e.g. attracting potential fields).

4.4.2 Experiment 5, Pathfinding, Single Destination

Varying number of units traverse the Labyrinth Hard map to a single destination. An experiment iteration is run until 1 unit reaches the destination. The closest unit to the goal in each experiment iteration always start the same distance away from the goal, meaning the experiment will be run *roughly* the same number of frames. The experiment is run 40 times, each iteration increasing the number of units by 10. The experiment is designed to test the different algorithms' ability to supply agents with paths to the same destination.

4.4.3 Experiment 6, Pathfinding, Multiple Destinations

Varying number of units traverse the Empty map to unique destinations. An experiment iteration is run 1000 frames. The start position and destination for each unit in each iteration are identical. Each iteration increases the number of units by 10 and the test is run 10 times. The experiment is designed to test the different algorithms' ability to generate multiple different paths.

Chapter 5

Results and Analysis

The results from the six experiments are presented and analyzed below. The first 3 experiments except experiment 2 are performed 100 times. Experiment 2 is only performed 10 times, due to pathfinding resulting in the same result every time. The first 3 experiments present results that evaluate the pathfinding algorithms ability to find the shortest path, avoid enemies and attack enemies. The last 3 experiments present results that evaluate each algorithms performance with an increasing load.

5.1 Experiment 1, Combat

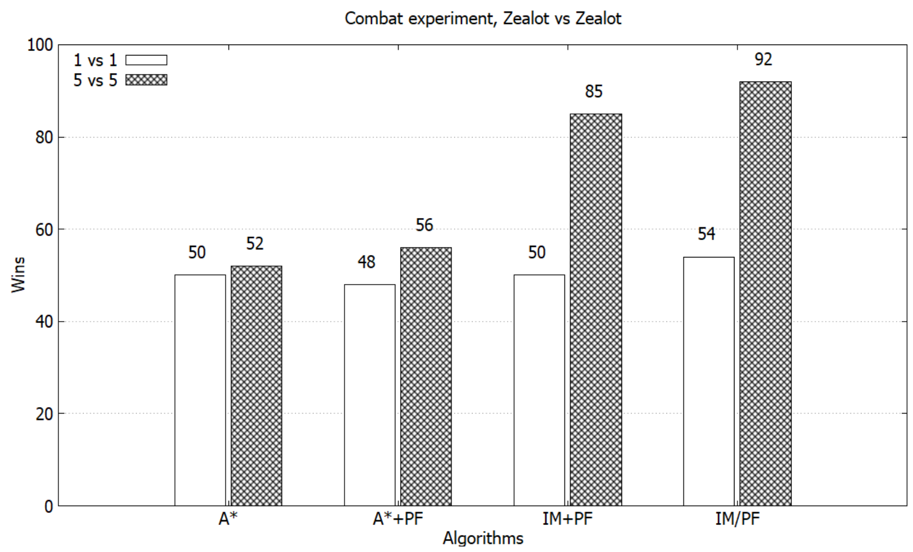


Figure 5.1: Combat experiment for Zealot vs Zealot, 1v1 and 5v5. Higher is better.

The results gained from experiment 1 show that in ”Melee vs Melee” combat, figure 5.1, when there are only two units attacking each other (1v1) all four algorithms have around 50% win rate. In the same figure, when the experiment is run in a 5v5 setup, a small increase can be observed for *A** and *A*+PF* where they have an increase of 2 and 8 percentages respectively. IM/PF demonstrates the largest increase in win rate between 1v1 and 5v5. With an increase of 38 win percentage from 1v1 to 5v5, resulting in 92% win rate in 5v5. A performance increase almost as large as IM/PF can be seen for IM+PF, with an increase of 35 win percentage between 1v1 and 5v5, resulting in a 85% win rate in 5v5. Both *A** and *A*+PF* perform as expected when the same unit is fighting on both sides, the win rate is around 50%.

IM+PF and *IM/PF* performs as expected for the 1v1 case but when the algorithms runs with 5v5 an increase can be observed. This big increase is likely caused by the Zealots using the *IM+PF* or *IM/PF* algorithm keep trying to go to the goal, to some extent, resulting in some Zealots trying to go around and therefore don't get stuck behind friendly units and can then attack the enemy units from the sides. It can also be caused by the PF used by the units, which also has the friendly units in the field as small charges, so that they can avoid each other. This could result in units that would choose to go around and attack from the sides. If this is the case, *A*+PF* should see some increase to its win rate when performing the experiment 5v5. It has a win rate of 56%, but this is not high enough to indicate a significant change.

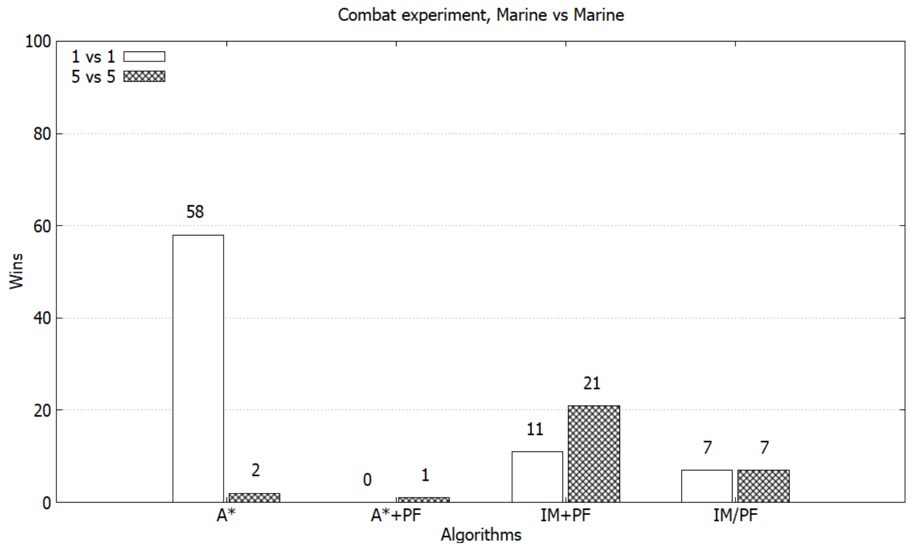


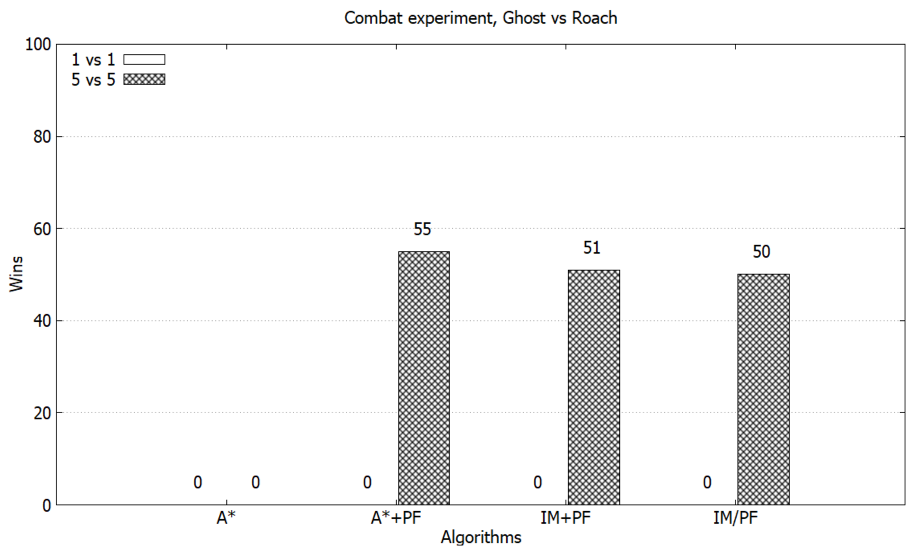
Figure 5.2: Combat experiment for Marine vs Marine, 1v1 and 5v5. Higher is better.

The presented results in figure 5.2 are difficult to analyze due to the variations between the algorithms win rate. Only *A** performed as predicted when identical units fight. This is not the case for any other setup of the experiment. When performing 5v5, the *A** algorithm only managed a 2% win rate, this is caused by all units "filing up" to walk in a straight line to the goal. Which results in enemy units that are waiting at the destination to have time to attack the first unit before the other units can come close enough to attack, effectively causing the fight to be 4v5 because of the bad positioning.

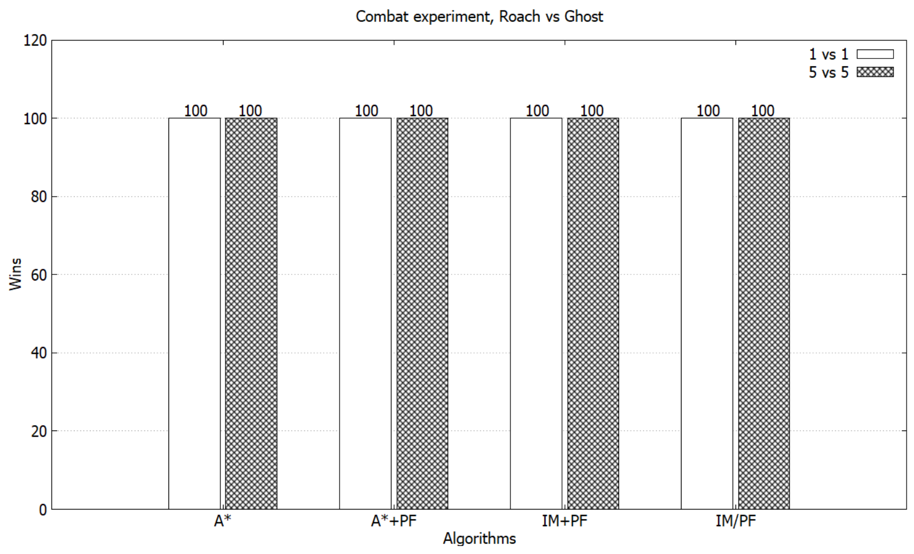
For *A*+PF*, *IM+PF* and *IM/PF*, the low win rate is caused by the units having the same attack range, causing the PF to force the agents to stay at their max attack range, resulting in them walking from side to side, trying to go alongside the range. Because of this, fewer attacks against the enemy are executed, which caused the low win rate.

The results gained from the Ghost vs Roach experiment, figure 5.3a, shows that all four algorithms lost every time in 1v1 combat. The reason that the ghost can't kill the roach in 1v1 is because of the difference in damage and hit points for the units. The ghost has 10 attack damage and 100 health points while the roach has 16 attack damage and 145 health points, as presented in table 4.1.

When the algorithms performed the experiment with 5 agents (5v5), all algorithms



(a) Combat experiment for Ghost vs Roach, 1v1 and 5v5.



(b) Combat experiment for Roach vs Ghost, 1v1 and 5v5.

Figure 5.3: Enemy combat experiment. Higher is better.

except *A** had an increase of around 50% win rate, 55% for *A*+PF*, 51% for *IM+PF* and 50% for *IM/PF*. The increase in win rate can be contributed to the use of PF because the *A** pathfinder did not see an increase in win rate. The use of PF enables the units to attack and back away when the enemy comes to close. Switching the setup so the algorithms control the roach instead of the ghost, figure 5.3b, results in a 100% win rate for all four algorithms. Due to the mentioned difference in damage and hit points.

The results for the "Ranged vs Melee" experiment, figure 5.4, shows all four algorithms lost every 1v1 battle in the experiment. A slight improvement can be seen for *A*+PF* and *IM/PF* when the experiment is performed with 5v5 units, yielding an 27% and 11% win rate improvement for the algorithms respectively. While both *A** and *IM+PF* kept their 0% win rate.

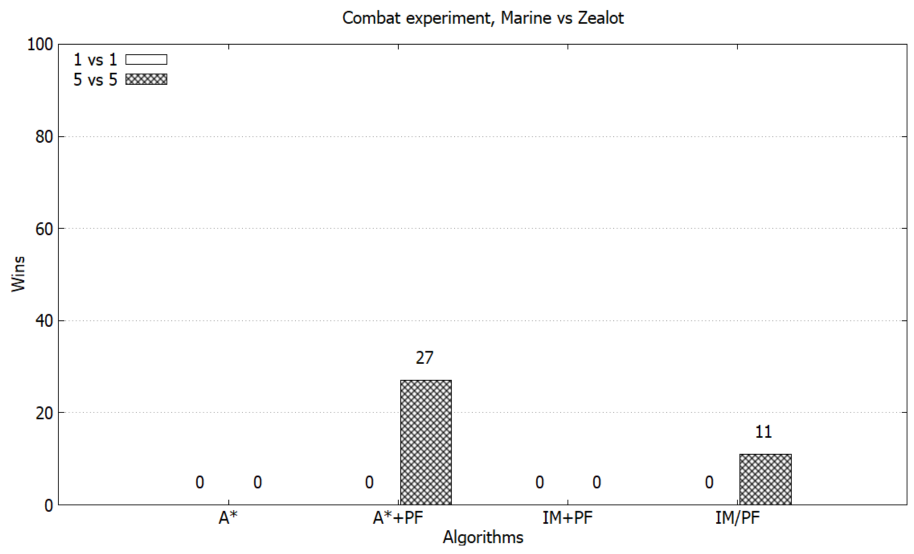


Figure 5.4: Combat experiment for Marine vs Zealot, 1v1 and 5v5. Higher is better.

A+PF* achieved the highest win rate thanks to the hard switch to PF as soon as the agent is in range of the enemy, resulting in agents that no longer tries to make its way to the specified goal. This is also why *IM/PF* is able to achieve a win rate of 11%, it also makes a hard switch from IM to PF, but not as soon as it is in range of the enemy, it waits until the PF have achieved a specific value and then it switches to only use the PF. If the PF value drops under the specific value the algorithm starts using the IM again and thus tries to walk towards the destination. This is probably why the *IM/PF* have a lower win rate than *A*+PF*.

IM+PF didn't achieve any wins due to it trying to walk towards the goal, the PF will make it so that the units try to walk around the enemies, but due to the higher damage and hit points of the zealots, the marines are not able to survive and kill them, while moving towards the goal.

5.2 Experiment 2, Pathfinding

Experiment 2 only tests the pathfinding capabilities of the four algorithms, resulting in *A** and *A*+PF* walking the same distance, and *IM+PF* and *IM/PF* also walking the same distance, as shown in figure 5.5. Because of this, only *A** and *IM* parts of the algorithms affect the distance when no objects that emit a *PF* is present on the map. Because of this, *A*+PF* won't find the diagonal paths, visualized in figure 5.6a, and *IM+PF* will find the diagonal paths, visualized in figure 5.6b. And as seen, the results for both *A** and *A*+PF* indicates a longer distance.

The same behavior is present on all three labyrinth maps giving the result of a longer distance walked for *A** and *A*+PF*. This behavior is not present on map Empty50 because there are no walls or objects that hinders the pathfinding of *A**. That *A**, *A*+PF* and that *IM+PF* and *IM/PF* perform the same is logical since the experiment only tests the pathfinding capabilities of the algorithms. The PF part of the algorithms are not used. The different results between *A** and *IM* is only present because *A** won't find the paths that is diagonal, as shown in figure 5.6a and

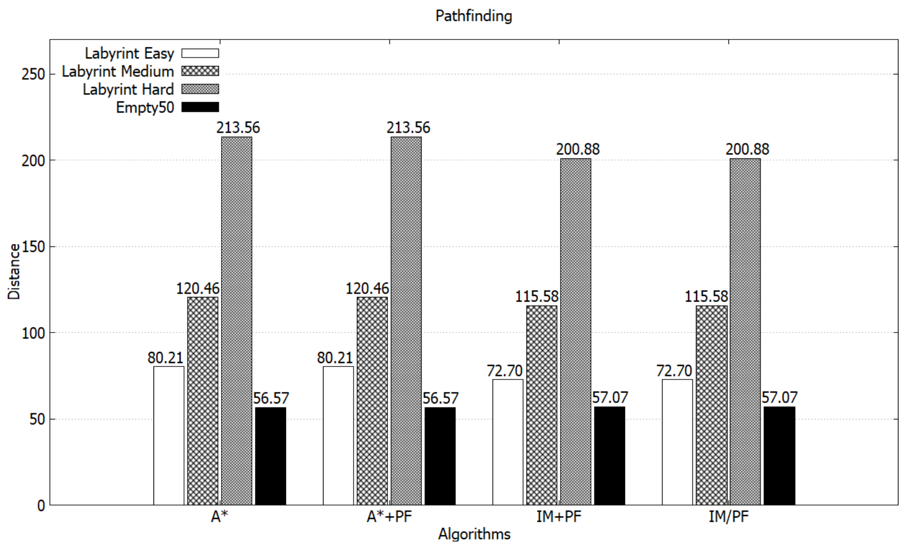
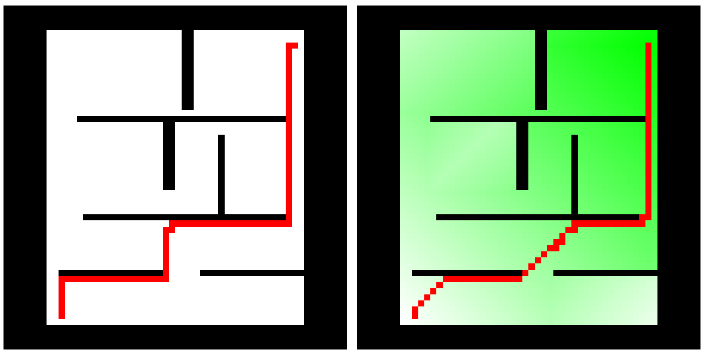


Figure 5.5: Pathfinding experiment, distance traveled. Shorter is better.



(a) Pathfinding experiment, walked path (red).

(b) Pathfinding experiment, walked path (red). Gradient of green indicates desired destination.

Figure 5.6: Walked path for *A*+PF* and *IM+PF* on Labyrinth Easy. Start point, bottom left corner. End point top right corner.

5.6b, due to the mentioned problem in section 4.1, where the path is found below the diagonal of the map.

5.3 Experiment 3, Enemy Avoidance

The results presented for experiment 3 is the average performance of 100 executions of experiment 3. For some of the runs the agent that tries to avoid the enemies sustain enough damage to die. When this happens the unit gets the remaining A*-walk distance to the goal added to the total distance walked. This is only an issue for the *A** and *A*+PF* algorithms. The agents utilizing *IM+PF* and *IM/PF* never receive enough damage to die.

A+PF* is the algorithm that obtains the longest walking distance out of the four algorithms, with varying 6-24 tile width difference compared to the IM-based algorithms. The shortest travel paths on the Labyrinth Hard and Empty50 maps are achieved by *A**. On Labyrinth Easy and Medium, the distance walked by *A** is 5-6 tile width longer than the IM-based algorithms, which has the shortest distance traveled on those maps.

In figure 5.7b the results for the amount of damage each algorithm sustains on all four maps can be seen. *A** receives the greatest amount of damage and *IM+PF* and *IM/PF* receives the least amount of damage over all, except on map Labyrinth Easy where *A*+PF* sustains 0 damage. The IM-based algorithms are the only algorithms that doesn't sustain enough damage to die on any of the maps while both *A** and *A*+PF* suffered from this. *A** has test iterations where the unit dies on all four maps except on map Empty50, this can be observed in the high average damage in figure 5.7b. The same can be observed for *A*+PF*, but only for maps Labyrinth Medium and Hard.

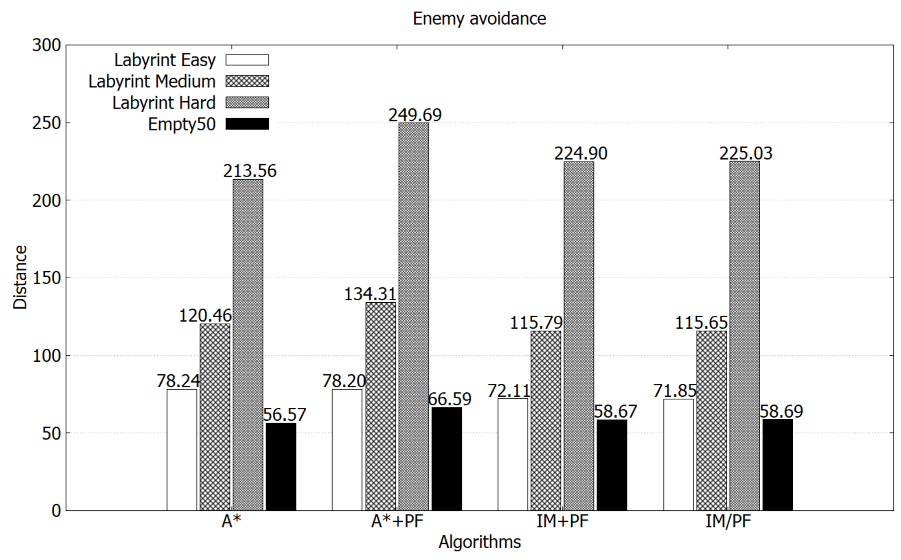
In figure 3.1b an example of how the map Labyrinth Hard would be visualized can be seen. This visualization is how the IM-based algorithms observes the map. The red circles are enemy units that should be avoided. The background goes from white to green and the unit will go towards a more intense green, which represents a lower value in the IM. The red circles are added together with the value obtained from the IM resulting in a higher value. In figure 3.1a a more clear view can be obtained of the circle. On the edge of the circle, the lowest value can be obtained where the red is more saturated, and higher values can be obtained closer to the middle. When added together, the red circles creates areas on the IM that the unit tries to avoid.

Because the *A** algorithm does not use a PF, agents utilizing the algorithm never try to avoid enemies and therefore sustain a greater amount of damage than the other algorithms. One factor that contributes to why *A*+PF* doesn't perform as well as *IM+PF* is the "hard switch" from A* to PF. The switch causes the agent to walk back the way it came in most cases, and when the A* path is recreated there is a big risk that the unit walks straight into to the enemy again. This behavior is not as prominent for *IM+PF* because to that agents tries to avoid the enemies while simultaniously being influenced by the goal, and can therefore walk in the red circles because it is the best path to take.

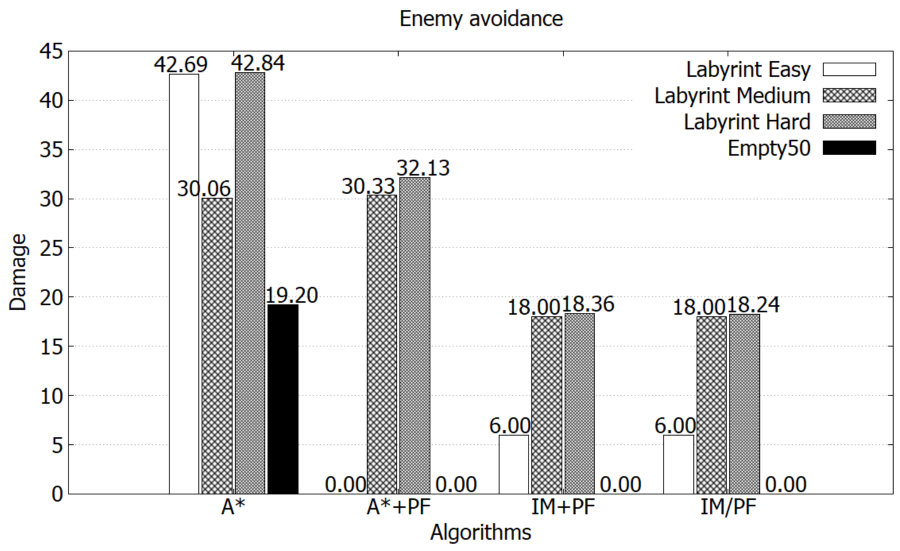
IM/PF does not suffer from the problems of *A*+PF*, because the threshold to make a hard switch from the IM to the PF is not reached, resulting in same or similar performance as *IM+PF*.

5.4 Baseline Test

The baseline test displayed in figure 5.8 shows a steady increase in frame time as more units are added to the scene. These are expected results, since more units active on the map results in an increased system load, because of the increased number of entities that are rendered and animated. The fluctuations in frame time around 1000-1600 and 2000-2100 units are most likely caused by internal deviations in the StarCraft 2 engine, or other factors in the SUT (such as Windows background tasks).



(a) Distance traveled for each algorithm. Lower is better.



(b) Damage taken for each algorithm. Lower is better.

Figure 5.7: Enemy avoidance experiment.

5.5 Experiment 4, Combat, Single Unit Type

The results from experiment 4 show that the average frame time in combat is very similar for all 3 algorithms (see figure 5.9a). However the initial frames, when the paths are generated, scale differently (see figure 5.9b).

There is no discernible difference of the performance trend between the algorithms in the experiments' average frame time. This is interesting since *IM+PF* and *A*+PF* generate potential fields each frame. Since there is no noticeable scaling impact of this workload, it can be concluded that the asynchronous PF generation on the GPU and StarCraft 2's frame rendering is able to complete without impacting the frame time for any experiment iteration. The agents utilizing *A*+PF* have a slightly increased frame time. The agents utilizing *A*+PF* need to recreate an *A** path when they are

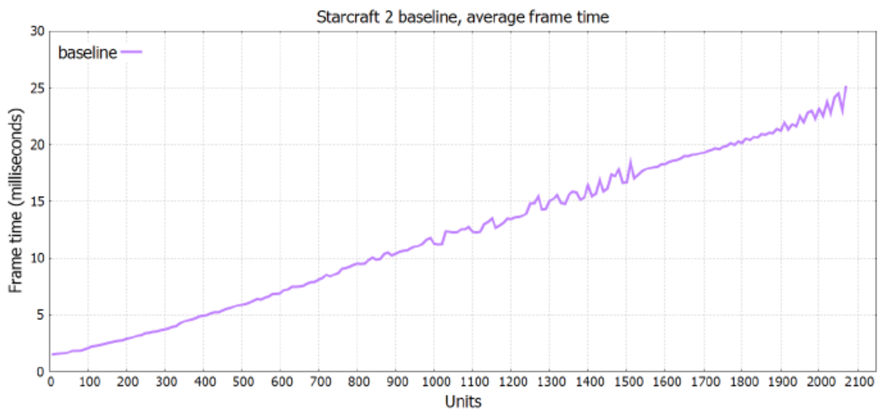


Figure 5.8: Average frame time baseline, 1000 frames. Lower is better.

no longer influenced by a PF. So *A*+PF* generates paths on the fly, not just the first few frames like the other algorithms, which does contributes to the discrepancy in performance per frame. *A*+PF* also require increased per-agent computing to utilize A* and PFs intelligently on the CPU, which also may contribute to the increased frame time, but this is uncertain.

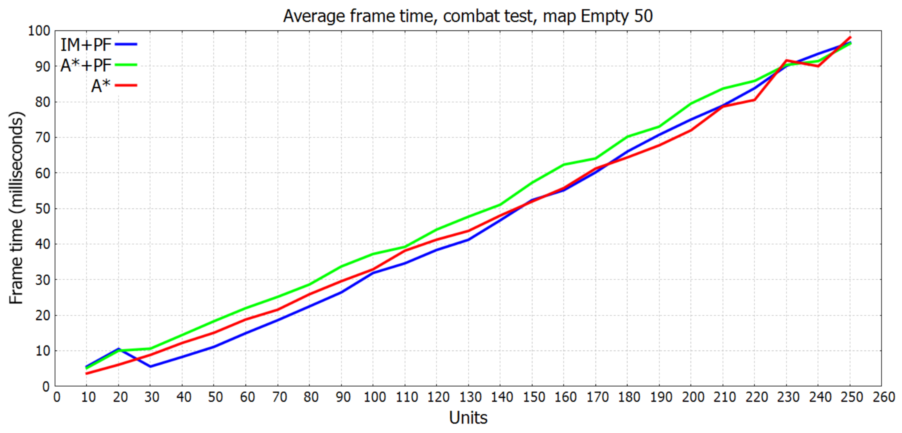
The pathfinding algorithms' path calculation frames do show great discrepancy. The generation of the IM starts of quite slow compared to A*, requiring 4.7x more compute time when the algorithms generate paths for 10 agents, but does exhibit improved performance compared to *A** and *A*+PF* when 60 or more agents requires a new path at the same time. This is expected since only 1 IM is created that is used by all agents. The workload does not increase based on the number agents that require pathfinding to the same destination.

The *IM+PF* path calculation frame time does however increase with experiment iteration, but this is not due to increased path generation workload. In the path calculation frame time the spawning of the units are included, causing the increasing workload. *A** and *A*+PF* appear to scale linearly when looking at the path calculation frames, averaging an increase of 143ms every experiment iteration (each increase of 10 units).

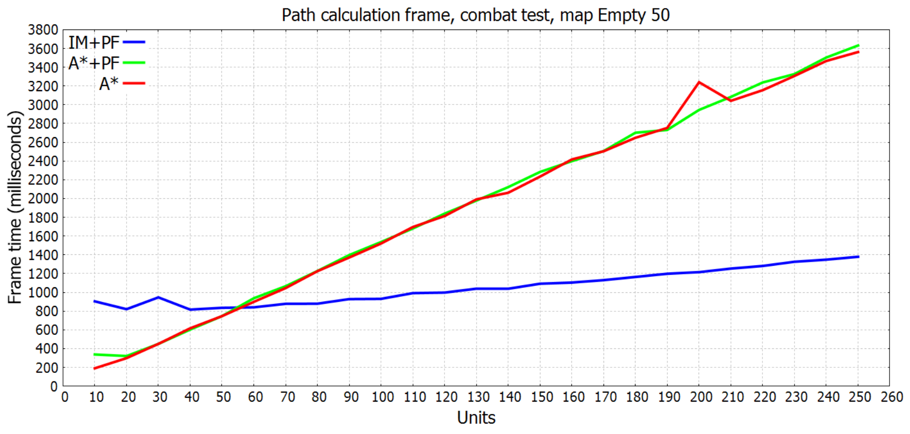
Total RAM allocated for each algorithm in experiment 4 (see figure 5.9c) is quite even between *IM+PF* and *A*+PF*, allocating 82.3MB on average, while *A** averages 15.3MB (15.6% as much memory as the other algorithms). The increased memory usage of *IM+PF* and *A*+PF* is caused by the PFs and IMs allocated in RAM. The internal storage allocators pre-allocates memory blocks for the 2D-maps, which is included in the measurement of memory utilization. This means that even though *A*+PF* does not utilize any IMs, the memory is still allocated and is included in the result data. RAM utilization increases linearly with each experiment iteration. This is expected since several arrays store information about active units in the environment.

VRAM utilization is quite low overall (see figure 5.9c), maxing out at a constant 157KB for the *IM+PF* algorithm, while *A** does not utilize GPU memory at all. It should be noted however that the measurements are done after the CUDA kernel launches and the agent's pathfinding logic function, meaning that the memory usage of the fast PF generation *may* not be captured by the VRAM measurement tool. The PFs would account for an additional 12 16KB (depending on memory pitch and storage alignment).

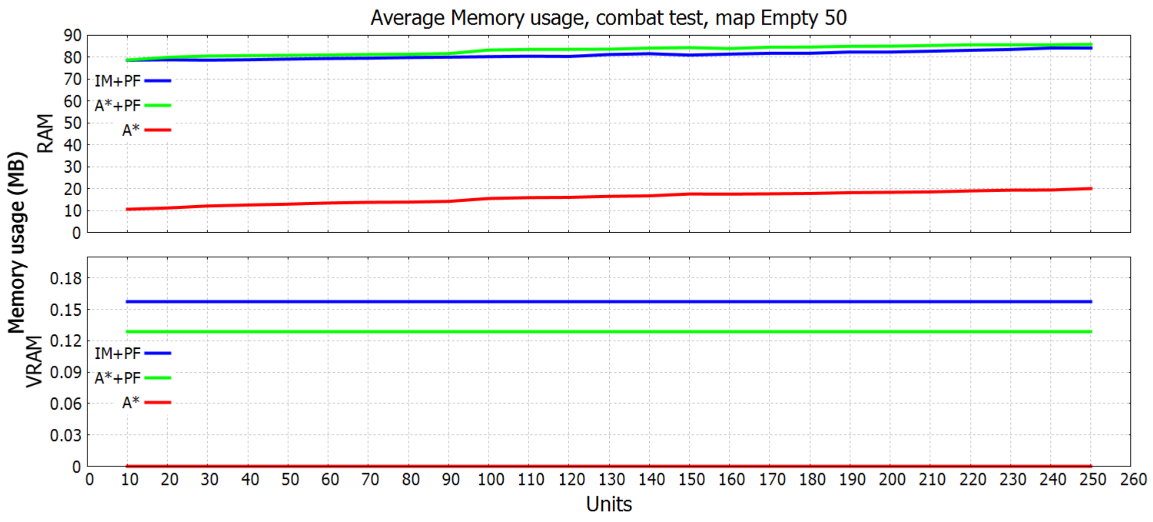
The *IM+PF* algorithm allocates 28.6KB more VRAM than *A*+PF*, contributed to the allocation of the IM. The remaining device memory allocated by both *IM+PF* and *A*+PF* is utilized by the list of units, unit lookup table, terrain map and internal arrays. Similar to the RAM, most of these memory allocations are block-allocated, meaning they most likely occupy more space than they utilize. Because of this it is hard to get an accurate measurement of VRAM utilization. There is no increase in VRAM allocation with experiment iteration because of this.



(a) Average frame time.



(b) Frame time of the path calculation frame.



(c) Average memory usage, RAM and VRAM.

Figure 5.9: Results from experiment 4. Agents generate paths to enemy units and engage in combat. Lower frame time and memory usage is better.

5.6 Experiment 5, Pathfinding, Single Destination

Similar to experiment 4 (see section 5.5) the average frame time remains consistent between algorithms in experiment 5 (see figure 5.10a) even though PFs are generated each frame. The pathfinding algorithms' path calculation frames (see figure 5.10) also show similar results as experiment 4, small differences can however be seen. The generation of the IM outperforms the A* pathfinding once 80 or more units requests paths, instead of 60 as in experiment 4. This is most likely caused by the increased complexity of the map. The workload of generating an IM on the GPU increases more drastically than creating an A* path on the CPU when the environment's complexity increases, because of the GPU's poor ability to cache memory and handle random access in linear lists.

The performance trend of frame time for all algorithms in experiment 5 are improved compared to experiment 4. The path calculation frame time of the IM generation only increases 1/4 as fast per iteration as the previous experiment, probably contributed to the lack of hostile units in experiment 5. The trend of the *IM+PF* path calculation has a linear increase with small fluctuations, however it's not constant, which would be expected by the generation of an IM. This can most likely be contributed to the creation of the units by the StarCraft 2 engine, which increases with each iteration of the experiment. *A** and *A*+PF* appears to scale linearly in regard to the path calculation frames, averaging an increase of 91ms every experiment iteration (increase of 10 units).

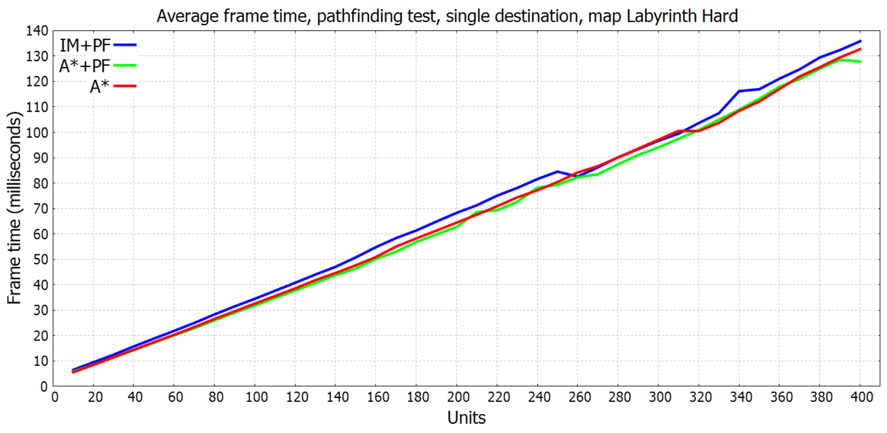
The RAM utilization in experiment 5 (see figure 5.10c) is very similar to the RAM utilization of experiment 4 (see Figure 5.9c). The only difference being that experiment 5 requires less memory per iteration, since there are no enemy units that needs to be stored. The VRAM is also utilized slightly less by *IM+PF* and *A*+PF* than in experiment 4, which can also be contributed to the lack of enemy units that needs to be stored on the device.

5.7 Experiment 6, Pathfinding, Multiple Destinations

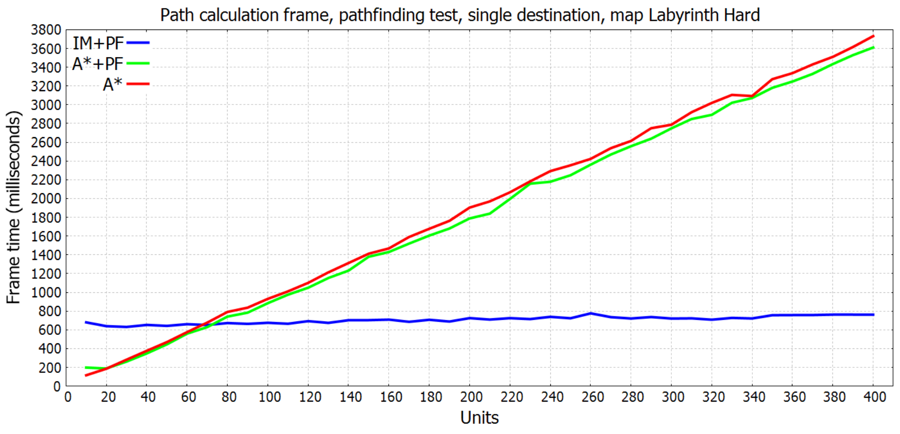
Analyzing the average frame time (see figure 5.11a) of experiment 6, it can be seen that *IM+PF* has a slightly steeper performance trend than *A** and *A*+PF*. Since the experiment requires the *IM+PF* algorithm to create paths to many different destinations, each unit is allocated in its own large 2D array. Because of this the *IM+PF* solution is much less effective at utilizing the CPU cache, and therefore takes a performance hit. In contrast, the agents using the *A** and *A*+PF* algorithms always read chunks of sequential memory, and can use the cache more effectively.

Looking at the difference in compute time of the path calculation frames, see figure 5.11b, it can be observed that *IM+PF* performs horrendously in this experiment. Generating IMs is quite demanding, and the entire benefit of using IMs is lost if only a single unit uses the IM during run time. 8.1 seconds is required for *IM+PF* to create 10 different paths, which is a 47.1x longer compute time compared to *A** and *A*+PF*, scaling linearly to 56.0x longer compute time when creating 100 different paths, totaling 64.9 seconds for *IM+PF*.

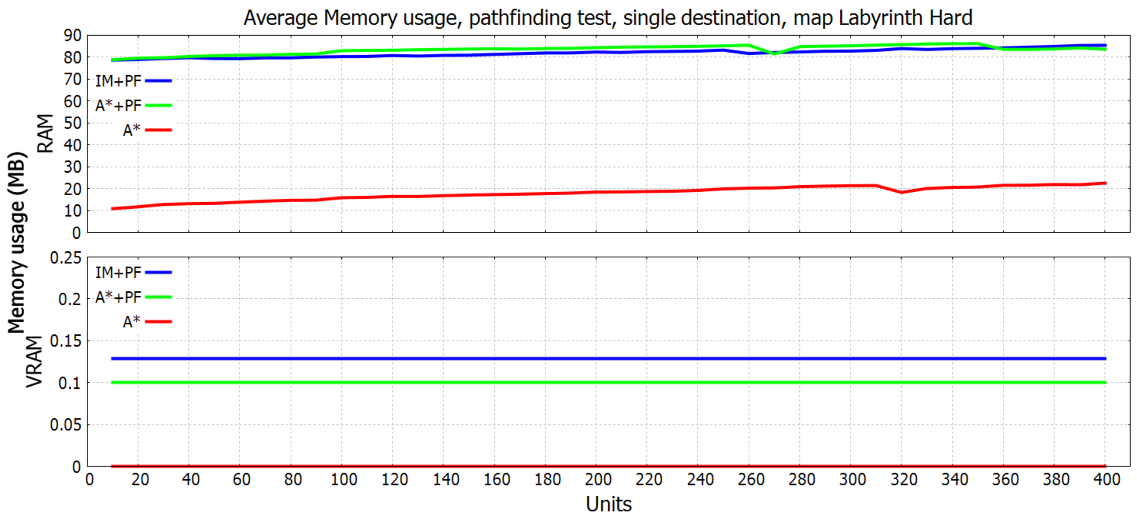
The utilization of RAM resembles those in experiment 4 & 5. There is a slight



(a) Average frame time.



(b) Frame time of the path calculation frame.

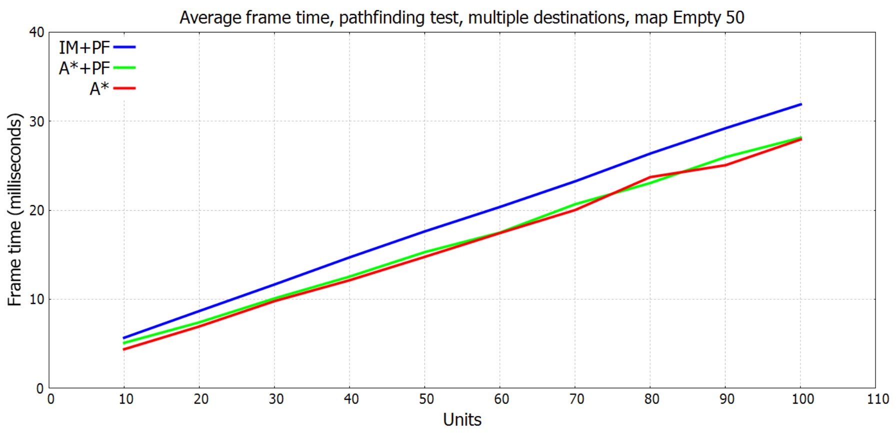


(c) Average memory usage, RAM and VRAM.

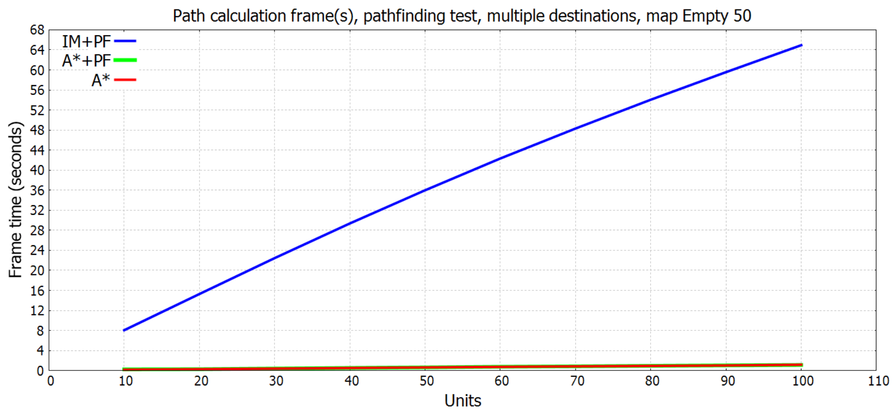
Figure 5.10: Results from experiment 5. Agents pathfind to a single destination through a labyrinth. Lower frame time and memory usage is better.

increase in memory usage (see figure 5.11c) for the *IM+PF* algorithm, 1.2MB when running the experiment with 100 units, that can be contributed to the storage of the IMs in RAM. The *A*+PF* and *A** algorithms utilize less memory than experiment 5, since the A* paths are much shorter on the empty map used in experiment 6, compared to the long paths generated by the labyrinth in experiment 5.

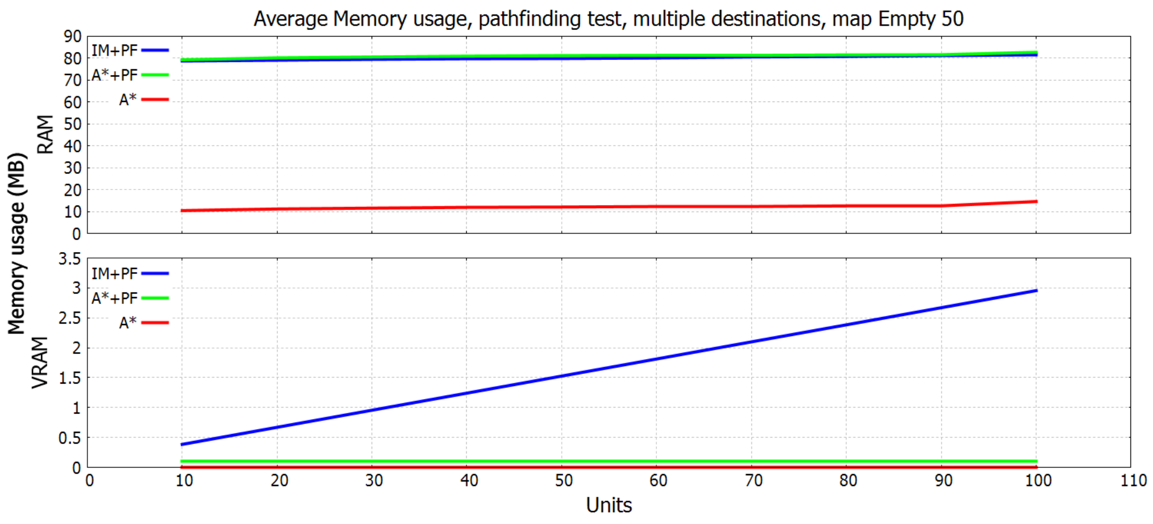
IM+PF does not perform well when measuring VRAM in experiment 6, as can be seen in figure 5.11c. The algorithm has a gradual increase of average device memory utilization from 385KB to 2.96MB over the experiment duration. This is caused by the large number of IMs that are queued for generation by the device. *A*+PF* shows the same results as experiment 5, since there is no changes to the usage of PFs between the experiments.



(a) Average frame time.



(b) Frame time of the path calculation frame(s).



(c) Average memory usage, RAM and VRAM.

Figure 5.11: Results from experiment 6. Agents pathfind to different destinations in an empty map. Lower frame time and memory usage is better.

Chapter 6

Discussion

The results presented in chapter 5 are both expected and unexpected. Some experiments performed differently than what were expected of that type of setup.

6.1 Unexpected Results

As stated in section 5.1, some results are unexpected. These results are mainly tied to the use of potential fields, and how the fields impact the performance of the algorithms.

The low win rate for *IM+PF* and *IM/PF* is caused by the PF, the agent is forced to stay at its maximum range and attack from that distance, because the agent and the enemy marine has the same exact range. This causes a behaviour where the agent tries to constantly move from side to side at its maximum range, because the agent can't find a single optimal grid coordinate with the lowest charge value. When the agent walks back and forth along the edge of its range, it requires a rotation towards the enemy to be able to attack. This gives the enemy marine the opportunity to attack more times than it got attacked, resulting in the enemy killing the agent faster. This problem emerges because PFs can generate local optimas, as described in section 2.3. As mentioned previously, a technique that can mitigate the local optima problem is *pheromone trails*. Pheromone trails discourage agents from walking to nodes that has been previously visited. This would hinder agents from walking back the the way they came, and push it forwards out of the local optima and towards the goal.

IM+PF is able to achieve a win rate of 21% in marine vs marines combat with the 5v5 setup, which indicates that it is better than the other four algorithms in this test.

6.2 Kiting

The most interesting setup is the case when the capabilities of the PF is displayed in the ghost vs roach setup, figure 5.3a. And results in a behavior called *kiting*. As mentioned in section 5.1, when performing the 1v1 test, all algorithms loses and when switching to 5v5 all algorithms except *A** has an increase in win rate. This increase is contributed to the use of PF. The PF enables the ghosts to attack the roaches and then back away to keep out of their attack range when they start to move towards the ghosts to attack, resulting in a kiting behavior. Due to this behaviour, the agents need to turn and face the target they are going to attack, this enabled the roaches to get close and attack a few times before the ghosts have turned away and started retreating again. Because of this, the enemy roaches are able to kill the ghosts a couple of times. If the agents didn't need to face the targets to attack they would most likely achieve a 100% win rate.

When the setup is flipped to agents using roaches against enemy ghosts, figure 5.3b, a 100% win rate is achieved by all algorithms. This setup is done to highlight the superior nature of the roach over the ghost and to indicate that if the ghost's longer range could be utilized they could win over the roaches.

6.3 Expected Results

For experiment 2 all four algorithms performs equally well, they all find the best path capable for their respective implementation. As seen in figure 5.6a and 5.6b, the A*-based algorithm doesn't find a diagonal path. The agents utilizing IM-based algorithms finds the diagonal path due to the algorithm performing A* for all points in the map. If the A* implementation was changed to always find a diagonal path if one is possible, it would result in the algorithm needing more time to calculate the path due to the many more nodes that would be expanded. Overall this experiment performed as expected and didn't generate any unforeseen data.

All four algorithms performed as expected in Experiment 3. It is no surprise that *A** fails and dies on the majority of the runs, because *A** can't inherently avoid enemies. The results where expected due to the related work by Hagelbäck, described in section 2.6, where he shows that A*+PF outperforms A* in his experiments. The better performance is contributed to the use of PF and due to the algorithms in this thesis except for one algorithm uses PFs they also shows a performance increase over just A*.

6.4 Performance and Scalability

Moving on to the real-time performance and scalability experiments, it's quite interesting that none of the experiment shows that generating the PFs has an impact on the average frame time. Even experiment 5 (figure 5.9) where 400 units are present in the environment simultaniously show any scalability impact. It should however be noted that no scalability experiment uses more than 1 unit type. Since the repelling PF is universal, but the attracting PF is unique for each unit type, increasing the number of unit types in the environment increases the workload of generating PFs.

As expected, *IM+PF* scales much better than A*-based solutions when A* is forced to create paths for multiple units to a single destination, with a break-even point being reached at 60 and 80 units in experiment 4 and 5 respectively. No experiments are however done on varying sizes of maps or with varying hardware, since those are two very important factors to the performance of the IM creation. The results from experiment 6 are also very expected, with the IM-based solution struggling when forced to create multiple IMs in a short time.

These results are consistent with the previously mentioned results presented by Elmir[10]. Performing small repetitive tasks are at first more optimally done on a CPU because of their low-latency design, but as the workload increases, the throughput oriented GPU handles the workload more efficiently.

The average frame time difference for all algorithms in experiment 4 and 5 show discrepancy in scaling. Since roughly the same workload is done in the initial frames in both experiments, it can be assumed that the difference is caused by the increased number of entities in the environment. It's clear that the number of units in the test environment heavily impacts the result of the experiment. This claim is further supported by comparing the results of experiment 4 and 5 with the general baseline of sustaining units in the environment, see figure 5.8. It can be observed that there is a huge discrepancy in frame time performance between the baseline and the experiments.

The discrepancy is likely caused by the multiple commands issued to the units each frame. For each frame, every agent in the environment is either issued a move-command or an attack-command. In normal gameplay, a unit would be issued a command, and once complete, or if new circumstances arise, a new command will be issued. The StarCraft 2 system is not designed to be issuing multiple commands per frame to various units, and the frame time performance is heavily suffering because of it.

It is unfortunate that memory utilization is not measured accurately, as the implementation allocates memory in large blocks. Because of this, conclusions regarding memory scalability can not be drawn.

Chapter 7

Conclusions

This thesis evaluates four algorithms and compared them against each other. The evaluations are done on four different maps in six different experiments that evaluates different properties of the algorithms. Three of the six experiments are used to measure and evaluate the algorithms pathfinding capabilities, and the other three experiments evaluates the real-time performance of the algorithms.

Based on the results collected from the experiments, it can be concluded that the proposed pathfinding solution *IM+PF* and its variant *IM/PF* do outperform *A** and *A*+PF* in both pathfinding performance and scalability in certain situations. In environments where multiple units require pathfinding to a single destination, creating an IM can be beneficial compared to creating separate A* paths for each agent. This is especially true if agents require alteration to their path, for instance to avoid enemies, forcing them to create new paths later to that same destination. The inherent avoidance and combat capabilities of *IM+PF* and *IM/PF* make them well suited in environments where agents should excel in movement micro management.

There are clear benefits of using an IM for pathfinding compared to A*. The ability to traverse the map diagonally at all times benefits *IM+PF* and *IM/PF* greatly. However, creating an IM requires more compute time than creating an A* path, so the IMs must be utilized by multiple agents to be a viable replacement to A*.

The IM-based algorithms suffer when multiple paths to different destinations are required at the same time, causing noticeable frame time reduction during the generation of the IMs. The IMs also require more memory to store the 2D maps, and if few agents access the same map during a frame, there is a high risk of causing cache misses. Also, since the GPU is utilized to generate the IMs and PFs, certain inherent problems arise. In most modern high fidelity games, the GPU is the bottleneck during the game's update loop[2], so adding more work to the GPU may impact the game's overall frame time.

Based on these pros and cons, the novel algorithms, *IM+PF* and *IM/PF*, are best utilized in environments where multiple agents require paths to common static goals, and where dynamic entities in the environment should be avoided or engaged. Solutions where the influence maps can be pre-calculated or where compute time is of less importance are ideal because of the significant time required to generate the maps. Games such as "They are billions"[15], where "hordes" of hostile agents are tasked with attacking a base controlled by the player, are good solutions for the novel algorithms. Similarly, games of the *tower defence* genre where the player builds structures on the map to hinder enemies from reaching a shared goal is an ideal use case for *IM+PF* and *IM/PF*.

Chapter 8

Future Work

During the planning and implementation of the project some interesting suggestions and ideas were brought up that could not be incorporated in the project's time frame. Therefor these ideas are proposed as future work for anyone looking to further study the topic.

8.1 Comparisons

Performing the experiments with more algorithms, comparing a greater number of pathfinding solutions. Comparing the novel solutions to more varied algorithms, such as Hagelbäck's *A*+Boids*[21] or NavMesh pathfinding, would increase the validity of the research and to the novel algorithms.

8.2 Comparisons in a More Suitable Environment

As seen in the discrepancy in frame time between the experiments (figure 5.9, 5.10 and 5.11) and the StarCraft 2 baseline (figure 5.8), StarCraft 2 struggles to issue multiple commands to multiple units per frame. Therefore, the comparison should be performed in other games or benchmarking environments where performance will not be reduced by changing the walking direction of multiple units per frame.

8.3 Combining PFs With Pheromone Trails

The local optima problem described in section 2.3 can be negated to some extent using a technique called *pheromone trails*, described by Schwab[34]. Pheromone trails are indications on a map that a unit has traversed the location before. This technique can be used in several ways; to mark a path for future traversal, to deter agents from walking the same path as other agents, and more.

Agents utilizing IM+PF can get stuck in terrain when avoiding enemy units, because of local optima. Implementing the use of local pheromone trails would reduce the issue. Each agent would have access to a private trail, so agents are not affected by other agents' paths, to repel the agent from positions that have been previously traversed. This would "force" the agent out of the local optima to resume normal traversal, but would normally incentivize the agent to always keep moving out of its current location. This negative behaviour can however be reduced, depending on implementation.

8.4 Sort Units in a Quadtree Before PF Generation

The generation of the PFs, as implemented in this project, scales linearly with the number of units in the entire environment, even though only a small portion of those units affect each thread. A culling technique, performed before the kernel launch, can be used to reduce unnecessary load to the generation of the PFs. This could be achieved by sorting units on the map into a *quad tree*[13] and assigning thread groups various quad tree nodes depending on their position in the world. Such an implementation will reduce the number of unnecessary distance checks between units and threads, and improve the usage of the intermediate buffers.

8.5 Host Mapped Memory

Mapped memory[32] allows the GPU to write data directly to system memory. As expected, such an operation is slow, but does not lock an SMP during the transfer. Because of this, all results from the IM and PF generations can be converted to output directly to RAM instead of requiring a separate VRAM-to-RAM transfer after the entire GPU job is complete. A potential drawback to the constant and slow transfer of data during the generation of PFs and IMs is the occupancy of bandwidth. It is unknown if the low constant bandwidth utilization may cause performance degradation.

References

[1] G. Amador A. Gomes. xtrek: An influence-aware technique for dijkstra's and a pathfinders. *International Journal of Computer Games Technology*, 2018(5184605):19, 2018.

[2] Talha Amjad. How much does your cpu matter when gaming in 2018? https://segmentnext.com/2018/05/29/cpu-matter-when-gaming/, May 2018. retreived Feb-2019.

[3] Johan Anderdahl and Alice Darner. *Particle Systems Using 3D Vector Fields*. PhD thesis, Blekinge Institute of Technology, 2014.

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[5] Blizzard. Starcraft ii official game site. https://www.blizzard.com/en-us/. retreived Apr-2019.

[6] Blizzard. Blizzard starcraft 2 client api. https://github.com/Blizzard/s2client-api, Jan 2019.

[7] Brilliant.org. A* search. https://brilliant.org/wiki/a-star-search/, 2016. retreived Jan-2019.

[8] Alex Champandard. The mechanics of influence mapping: Representation, algorithm & parameters. *Top 10 Most Influential AI Games | AiGameDev.com*, 2011.

[9] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, Jan 2011.

[10] Hassan Elmir. *Using Multicore Programming on the GPU to Improve Creation of Potential Fields*. PhD thesis, Blekinge Institute of Technology, 2013.

[11] Samuel Erdtman and Johan Fylling. *Pathfinding with Hard Constraints - Mobile Systems and Real Time Strategy Games Combined*. PhD thesis, Blekinge Institute of Technology, 2008.

[12] Jesper Hansson Falkenby. *Physically-based fluid-particle system using DirectCompute for use in real-time games.* PhD thesis, Blekinge Institute of Technology, 2014.

[13] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, Mar 1974.

[14] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. *Intelligent Agents III Agent Theories, Architectures, and Languages Lecture Notes in Computer Science*, page 21–35, 1997.

[15] Numantian Games. They are billions. (digital distribution), 2017.

[16] Ross Graham, Hugh McCabe, and Stephen Sheridan. Pathfinding in computer games. *The ITB Journal*, 4(2):57–81, Dec 2003.

[17] Tobias Grundel. *Particle Systems: A GPGPU based approach.* PhD thesis, Blekinge Institute of Technology, 2015.

[18] Maurice H. J. Bergsma and Pieter Spronck. Adaptive spatial reasoning for turn-based strategy games. *Bijdragen*, Jan 2008.

[19] Johan Hagelbäck. Using potential fields in a real-time strategy game scenario. *Top 10 Most Influential AI Games | AiGameDev.com*, 2009.

[20] Johan Hagelbäck. Potential-field based navigation in starcraft. *IEEE Conference on Computational Intelligence and Games*, pages 388–393, 2012.

[21] Johan Hagelbäck. Hybrid pathfinding in starcraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(4):319–324, 2016.

[22] Johan Hagelbäck and Stefan J. Johansson. The rise of potential fields in real time strategy bots, Jan 2008.

[23] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[24] Yiyang He. *A Physically Based Pipeline for Real-Time Simulation and Rendering of Realistic Fire and Smoke.* PhD thesis, KTH Royal Institute of Technology, 2018.

[25] Intel. Product brief, 7th gen intel core desktop processors, accelerate performance, May 2017.

[26] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Autonomous Robot Vehicles*, page 396–404, 1986.

[27] David Kirk and Wen-mei W. Hwu. *Programming massively parallel processors: a hands-on approach.* Elsevier, 2017.

[28] Yoram Koren and Johann Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, 1991.

[29] Kenli Li, Lipo Wang, and Yong Liu. Applications in heterogeneous parallel and distributed environment. *Concurrency and Computation: Practice and Experience*, 29(20), 2017.

[30] Victor Martell and Aron Sandberg. *Performance Evaluation of A* Algorithms.* PhD thesis, Blekinge Institute of Technology, 2016.

[31] NVIDIA. Desktop GPUs & Gaming PCs, https://www.geforce.com/hardware/compare-buy-gpus, GeForce product page, May, 2019.

[32] NVIDIA. Cuda c best practices guide. NVIDIA Developer Documentation, Oct 2018.

[33] NVIDIA. Cuda c programming guide. NVIDIA Developer Documentation, Oct 2018.

[34] Brian Schwab. *AI game engine programming.* Course Technology, 2 edition, 2009.

[35] Paul Tozour. Influence mapping. *Game Programming Gems 2*, pages 287–297, 2001.

[36] W. Zeng and R. L. Church. Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009.

[37] Hong Zhang, Da-Fang Zhang, and Xia-An Bi. Comparison and analysis of gpgpu and parallel computing on multi-core cpu. *International Journal of Information and Education Technology*, page 185–187, 2012.

Appendix A

Representations of the Used Maps

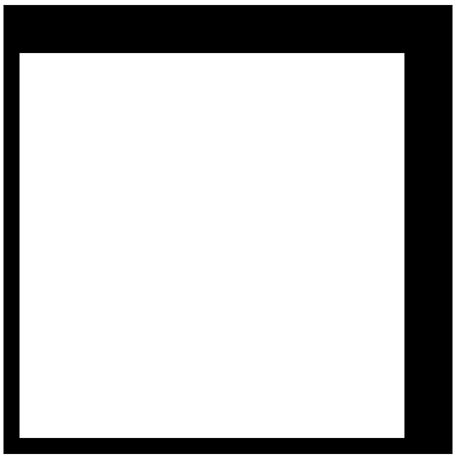


Figure A.1: Representation of map Empty50.

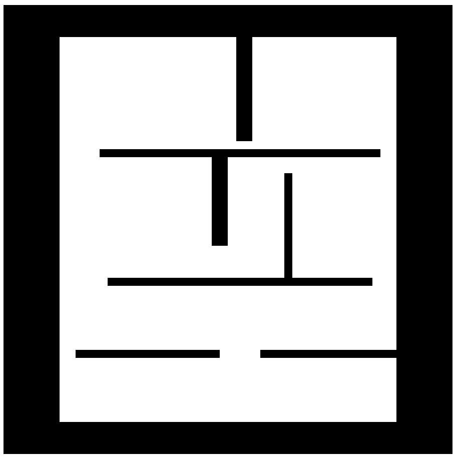


Figure A.2: Representation of map Labyrinth Easy.

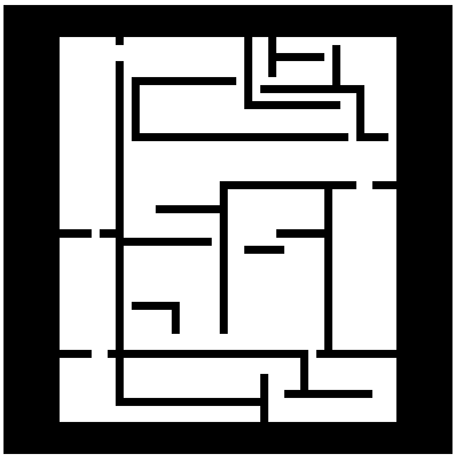


Figure A.3: Representation of map Labyrinth Medium.

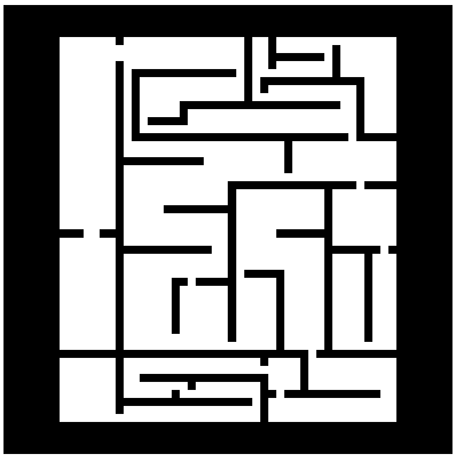


Figure A.4: Representation of map Labyrinth Hard.