UTILIZING SWARM INTELLIGENCE ALGORITHMS FOR PATHFINDING IN GAMES

Bachelor Degree Project in Informatics

30 ECTS
Spring term 2017

Alexander Kelman

Supervisor: Mikael Thieme
Examiner: Sanny Syberfeldt
Abstract

The Ant Colony Optimization and Particle Swarm Optimization are two Swarm Intelligence algorithms often utilized for optimization. Swarm Intelligence relies on agents that possess fragmented knowledge, a concept not often utilized in games. The aim of this study is to research whether there are any benefits to using these Swarm Intelligence algorithms in comparison to standard algorithms such as A* for pathfinding in a game.

Games often consist of dynamic environments with mobile agents, as such all experiments were conducted with dynamic destinations. Algorithms were measured on the length of their path and the time taken to calculate that path.

The algorithms were implemented with minor modifications to allow them to better function in a grid based environment. The Ant Colony Optimization was modified in regards to how pheromone was distributed in the dynamic environment to better allow the algorithm to path towards a mobile target. Whereas the Particle Swarm Optimization was given set start positions and velocity in order to increase initial search space and modifications to increase particle diversity.

The results obtained from the experimentation showcased that the Swarm Intelligence algorithms were capable of performing to great results in terms of calculation speed, they were however not able to obtain the same path optimality as A*. The algorithms’ implementation can be improved but show potential to be useful in games.

**Keywords:** Swarm Intelligence, Pathfinding, Ant Colony Optimization, Particle Swarm Optimization, A*
Table of Contents

1 Introduction.................................................................................................................. 1

2 Background.................................................................................................................. 2
  2.1 Pathfinding.................................................................................................................. 3
     2.1.1 A* ....................................................................................................................... 4
  2.2 Swarm Intelligence..................................................................................................... 5
     2.2.1 Ant Colony Optimization ................................................................................... 5
     2.2.2 Particle Swarm Optimization .............................................................................. 8

3 Problem ......................................................................................................................... 11
  3.1 Question ..................................................................................................................... 11
  3.2 Method....................................................................................................................... 12
     3.2.1 The Environment ................................................................................................. 12
     3.2.2 The Experiment .................................................................................................. 13
     3.2.3 Results & Validity ............................................................................................... 14

4 Implementation .............................................................................................................. 15
  4.1 Experiment Environment ......................................................................................... 15
  4.2 The Seeker and The Target ....................................................................................... 17
  4.3 Implementation of Ant Colony Optimization .......................................................... 18
  4.4 Implementation of Particle Swarm Optimization ....................................................... 20
  4.5 Implementation of A* ............................................................................................... 22
  4.6 Pilot Experiment ........................................................................................................ 23

5 Evaluation ..................................................................................................................... 27
  5.1 The Study .................................................................................................................. 27
     5.1.1 Battleground ...................................................................................................... 31
     5.1.2 Caldera ............................................................................................................... 33
     5.1.3 Crescentmoon ..................................................................................................... 37
     5.1.4 Scorchedbasin .................................................................................................... 39
     5.1.5 Stromguarde ....................................................................................................... 41
  5.2 Analysis ...................................................................................................................... 42
     5.2.1 Calculation Time .................................................................................................. 42
     5.2.2 Path Length ........................................................................................................ 43
     5.2.3 Game Completion Time ...................................................................................... 43
     5.2.4 Behaviour .......................................................................................................... 44
  5.3 Conclusions ................................................................................................................. 46

6 Concluding Remarks ..................................................................................................... 48
  6.1 Summary .................................................................................................................... 48
  6.2 Discussion .................................................................................................................. 49
     6.2.1 Garbage Collection ............................................................................................. 50
     6.2.2 Real World Applications .................................................................................... 50
  6.3 Future Work ................................................................................................................ 51

References ......................................................................................................................... 53
1 Introduction

Games today often utilize artificial intelligence in order to enhance the player's experience. There are many different functionalities that can be implemented in the form of artificial intelligence, one of the most common being pathfinding. Pathfinding is utilized in a great number of fields including games. One of the most common pathfinding algorithms in games is A*, which is always capable of performing to a great standard. However one of the drawbacks of using A* is the fact that A* expects an environment to remain static when determining a path. In today's fast paced games it is rare for an environment to remain static.

This paper focuses on researching the possibility of using swarm intelligence algorithms for pathfinding in games. The two algorithms tested are Ant Colony Optimization and Particle Swarm Optimization. Both algorithms' possess qualities that make them suitable for implementation in dynamic applications. However their use within games is still scarce.

Ant Colony Optimization is based on the natural behaviour of ants and their ability to path from their nest to a source of food. Ants do not communicate directly when trailing a path but rather by placing a pheromone trail. When another ant detects the trail it can either decide to follow the trail, thus reinforcing it, or not. The stronger the pheromone trail is the more likely an ant is to follow the trail. As pheromone decays from abandoned paths, the ants all converge on the shortest path.

Particle Swarm Optimization is a optimization algorithm based on social behaviour. By iteratively improving a candidate solution the algorithm hopes to achieve better solutions for the entirety of the swarm. Problems are solved by having 'particles' move about a search space with their own velocity controlled by simple formulas. Each particle's velocity is affected by the position of the global best fitness and the best fitness achieved by the particle itself.

Tests to measure the algorithms' capabilities for pathfinding in games have been detailed. As well as what aspects of the algorithms are recorded in order to measure the results. The results that were expected before experimentation took place are noted in this paper, with evidence to suggest that these actions were plausible.

The implementation of the algorithms is documented with regards to design decisions taken to further improve the performance of the algorithm. Also how issues regarding the algorithm were solved, or what workarounds were used. There is also a pilot study showcasing that the study can be evaluated in the current setup.
2 Background

Games today often utilize artificial intelligence (AI), a term first coined by John McCarthy in 1956 (McCarthy, 1956). AI refers to intelligence exhibited by machines, where an intelligent machine would ideally be capable of calculating an approach that offers the greatest chance of success. The AI often encountered in video games is referred to as game AI as its main objective is to enhance a player's experience. Bourg & Seemann (2004) define game AI as anything that gives the illusion of intelligence to an appropriate level, thus making the game more immersive, challenging and fun. Game AI often consists of a set of algorithms that control the behaviour of an agent (Russell & Norvig, 2003). Among these algorithms there is pathfinding that controls the movement of an agent.

Pathfinding is the task of plotting an uninterrupted path between two points. It is generally utilized in any field where movement takes place such as video games. Game AI is often dependent on good pathfinding in order to function in a desirable manner. In games the desired path is often not the best path or the shortest path in terms of required movement, it is rather the best path obtainable in the shortest amount of time possible. If enemies such as those present in Left 4 Dead 2 (Valve Corporation, 2009), see Figure 1, were to pause their movement when calculating a path towards the player then the player's immersion would be negatively affected. Being able to quickly calculate a new path in a dynamic environment is one of the challenges that pathfinding faces in games today.

One of the most renowned pathfinding algorithms is A*, capable of always determining the best path available, if there is one. However A* expects values to remain static during its calculations which is seldom the case in games today that consist of vast dynamic environments. Swarm intelligence (SI) algorithms Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO) are both based on emergent behaviour from interaction between simple agents. Utilizing intelligence born from interaction they possess qualities...
that make them suitable for real-time pathfinding. This chapter provides required information regarding pathfinding and the SI algorithms researched in this paper.

## 2.1 Pathfinding

Pathfinding is the task of calculating an uninterrupted path from one point to another. In games paths are generally found on terrains that agents can traverse. Depending on how the terrain is divided different paths can be found. One common method for dividing the terrain in games is the use of a grid. Tiles, the smallest unit in a grid, can take any form that allows for repetitive tiling. Using different tile-representations gives different results when searching for a path (Björnsson, Enzenberrger, Holte, Schaeffer & Yap, 2003). Different variants such as a square grid and a hexagonal grid can be seen in Age of Empires 2 HD (SkyBox Labs, 2013) and Civilization V (Firaxis Games, 2010) respectively, see Figure 2.

![Figure 2](image)

**Figure 2**  Two types of grid based games. Age of Empires 2 HD (left) and Civilization V (right) with a square grid and hexagonal grid respectively.

There are traditionally two methods of prioritization in pathfinding, the mathematical approach and the heuristic approach (Hart, Nilsson & Raphael, 1968). The mathematical approach focuses on determining the best possible path available, it is not concerned with other affecting factors such as cost. Whereas the heuristic approach focuses on determining the best possible path for as cheap as possible. The heuristic method does not always generate the best path available but rather the best path available within a price limit.

In video games pathfinding is an important aspect to create more engaging environments for the player. Games are often dynamic in the sense that a map might change as a wall is destroyed to open up a new path (Graham, McCabe & Sheridan, 2005). Environments are not the only dynamic aspects in games, often units are told to move to the player’s position. It is very unlikely that the player will remain in the same position the entirety of the game, thus a new path must be calculated repeatedly as the player moves. Therefore, in real-time games especially, a sub-optimal path found using the heuristic approach is often preferred. While it is possible to create pre-processed maps, containing world geometry, in order to reduce search times these solutions do not allow for dynamic environments.

The most common pathfinding algorithm is A* (chapter 2.1.1). However for pathing to a moving target’s position A* requires a lot of unnecessary time, as the path must be recalculated from the current position each time the target moves. Whereas A* is unable to adapt to new destinations, ACO (chapter 2.2.1) is well suited for dynamic applications such as pathing towards a moving target.
Games also struggle with creating realistic movement. Paths are often determined in straight lines causing unrealistic movement. As it is often better to have fewer nodes for faster search times this problem arises as more nodes would have allowed for smoother pathing. There are implementation methods available in order to create more realistic movement in games by controlling the agents steering between different nodes in order to create more curved movement.

2.1.1 A*

A* is one of the most renowned search algorithms due to its performance and accuracy. It is an informed search algorithm. This means that it estimates the distance from its current position to the goal and selects paths based on these estimates in order to find the shortest path with the lowest cost. It was first introduced in 1968 by Hart, Nilsson & Raphael (1968) as an improvement of Dijkstra’s algorithm (Dijkstra, 1959).

In order to determine a path, A* searches through a graph by looking at nodes. At each node it has access to information such as the node’s neighbours and the estimated cost of travelling via those neighbours. The algorithm begins by initializing two lists, an open list and a closed list. These lists are used to determine nodes that have yet to be examined and ones that have respectively. The starting node is always placed in the open list first. The search algorithm will continue as long as the open list is not empty or until the goal is reached. The algorithm can be described using the pseudocode in Figure 3.

```
initialize open list
initialize closed list
add starting node to open list

Do
    locate node with lowest F cost in open list - current node
    remove current node from open list
    generate current nodes successors, set current node as their parents
    for each successors
        if successor is goal, break
        successor.g = current.g + distance between current and successor
        successor.h = distance from goal to successor
        successor.f = successor.g + successor.h
        if successor is in open list with lower F cost
            skip successor
        else if successor is in closed list
            skip successor
        else
            add successor to open list
    End
    add current to closed list
While open list is not empty
```

Figure 3  Pseudocode for A* search algorithm
Each iteration the algorithm will select a new node to investigate, this node should be the
node with the current lowest estimated cost out of all nodes in the open list. In order to
estimate the cost the following formula is used:

\[ F = G + H \]

F represents the sum of G and H. G is the cost to get to the current node from the starting
node. H is the estimated cost to reach the goal from the current node, this if often called the
heuristic. In grid based systems using square tiles the estimated distance is usually the
displacement in tiles vertically and horizontally, this is also called the Manhattan distance
(Buckland, 2005).

When a new node is selected from the open list, its neighbours are checked. If a neighbour is
already in the closed list or a blocked node, not navigable, then the neighbour is ignored.
Otherwise the current F cost for the neighbour is calculated from the current path, if the
node is not already in the open list it is added. If the neighbour already exists in the open list
then its previous F cost from when it was added is compared to its current F cost. If the
current cost is lower than the previous it is overwritten, otherwise it is ignored.

The current node is then removed from the open list and added to the closed list. The closed
list contains all nodes that have already been investigated and is there to prevent nodes from
being searched more than once. Next iteration then begins by taking a new node from the
open list, the one with the lowest F cost, and checking whether this node is the destination or
not. This process then repeats until the destination is reached.

2.2 Swarm Intelligence

A swarm is a group of agents that communicate with each other, either directly or indirectly,
by acting on the local environment (Engelbrecht, 2007). The emergent collective intelligence
of groups of simple agents, relying too large extent on fragmented knowledge, is what is
defined as swarm intelligence. There is no central entity that controls the behaviour of other
agents, rather each agent acts individually in response to information gathered from other
agents. These interactions create intelligent behaviour (Bonabeau, Dorigo & Theraulaz,
1999). Much of the study on swarm intelligence often takes inspiration from nature, such as
ant colonies (Bonabeau & Theraulaz, 2000). One of the most renowned artificial life
programs created using swarm intelligence is Craig Reynolds's Boids (1987), meant to
simulate the flocking behaviour of birds.

The term Swarm Intelligence is one that was first introduced by Gerardo Beni and Jing
Wang (1989) in the regards to cellular robotics. Beni and Wang discuss in their paper what
they define as "intelligent" behaviour and its difference to robotic intelligence, stating that
swarm intelligence is an unpredictable intelligence born from intractability. Today the term
swarm intelligence is more specifically related to a general set of algorithms that control
agent's behaviour, separating the application of swarm principles to robots as swarm
robotics.

2.2.1 Ant Colony Optimization

Ant Colony Optimization is a population based approach to solving computational problems
first proposed by Marco Dorigo in 1992 (Dorigo, 1992). The algorithm was initially
introduced to search for an optimal path in a graph, inspired by the behaviour of ants. Ants
ability to find the shortest path between their nest and a source of food is the core concept utilized in the algorithm.

Ant societies resemble that of human societies in several fashions, they have division of labour, communication between individuals and the ability to solve complex problems. Despite most ants having poor eyesight they are still able to travel long distances in order to obtain food sources. Ants use pheromone trails in order to communicate information regarding paths. When an individual ant travels it does so seemingly randomly, however it will lay some pheromone on the ground to marking the trail it followed. When another ant encounters this trail it can determine to follow this trail thus reinforcing the trail with its own pheromone. The pheromone placed by an ant will eventually begin to disappear if not reinforced indicating that another path may be more attractive. This can be seen in Figure 4. As more ants use the same path thus increasing the pheromone count on the path, the more attractive the path becomes to other ants (Bonabeau, Dorigo & Theraulaz, 1999).

![Figure 4](image_url)

**Figure 4** Pathing of ants from nest A to food source E. a) Ant walking on a path between points A and E. b) An obstacle appears, forcing ants to path around. c) Ants steadily start choosing the shorter path.

The fundamental process of ACO is iteratively having agents in a population construct more optimal solutions. While there are many variants of ACO the one in focus for this assignment will be the Ant Colony System (Dorigo & Gambardella, 1997) which acts as an improvement on Dorigo, Maniezzo & Colorni's Ant System (1996).
The ACO algorithm created by Dorigo & Gambardella (1997) allows solutions to be found by using each ant build a possible solution to the problem by moving through a finite sequence of neighbour states. An ant’s next move is selected by a state transition rule. This state transition rule is directed by problem specific information as well as previous pheromone. After each has transitioned once the pheromone for all positions will update. This process is then repeated until every ant has built a path to the destination. Once each path has been built the best solution, determined by a fitness value, will be updated and pheromone for the paths will be updated. The process repeats until some specified criteria is met such as a singular path being determined. The algorithm can be summarized according to the following psuedocode seen in Figure 5.

Figure 5  Pseudocode for Ant Colony Optimization algorithm

In order to determine the next node to move towards an ant uses the pseudo-random-proportional action choice rule. This rule shares many similar qualities with the pseudo-random action choice rule often used in reinforcement learning (Watkins & Dayan, 1992). When applied to the travelling salesman problem (Dantzig, Fulkerson & Johnson, 1954) the rule shows bias towards shorter edges, this is balanced with the locally updating pheromone trail that encourages exploration. Equation 1 allows ant \( k \) in node \( i \) to select a node \( j \) to move to using the following:

\[
P_{ij}^k(t) = \begin{cases} 
\frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}(t)]^\beta}{\sum_{u \in J^k(i)} ([\tau_{ij}(t)]^\alpha [\eta_{ij}(t)]^\beta)} & \text{if } j \in J^k(i) \\
0 & \text{if } j \not\in J^k(i) 
\end{cases}
\]  

Equation 1

\( \eta_{ij}(t) \) is information regarding the problem, this being the value of path \( ij \) at the time \( t \). \( \tau_{ij}(t) \) is the total pheromone present at path \( ij \) at time \( t \). \( J^k(i) \) are all the available moves for ant \( k \) from node \( i \). \( \alpha \) and \( \beta \) are parameters that determine importance of the pheromone trail. Equation 2 is the next node \( j \) that ant \( k \) chooses to move:

\[
J = \begin{cases} 
\max_{u \in J^k(i)} \left\{ [\tau_{ij}(t)]^\alpha [\eta_{ij}(t)]^\beta \right\} & \text{if } q \leq q_0 \\
J & \text{if } q > q_0 
\end{cases}
\]  

Equation 2
q is a random number uniformly distributed between 0 and 1. q0 is a parameter value, (0 <= q0 <= 1). j ∈ f^k(i) is a node randomly selected using the pseudo-random-proportional action choice rule.

This next movement determination is run during each iteration until each ant has completed their path. However after moving from node i to j the pheromone value of the corresponding transition will be updated locally. By updating locally new search space may be explored as premature convergence of the solution can be avoided. Locally updating is especially important in problems where the starting position is fixed. Equation 3 determines the local update of pheromone:

\[
\tau_{ij}(t) \leftarrow (1 - \varphi) \cdot \tau_{ij}(t) + \varphi \cdot \omega
\]  

φ a parameter (0 <= φ <= 1). ω is the initial value of pheromone. After each ant has managed to build a path from the starting position to the end then the pheromone trail of the best paths is increased. This increase is to make the path more attractive as the algorithm is run until a criteria, dependent on the problem, is met. The global update of pheromone is done according to the following:

\[
\tau_{ij}(t) \leftarrow (1 - p) \cdot \tau_{ij}(t) + p \cdot \Delta \tau_{ij}
\]  
P ∈ [0, 1] is a parameter for pheromone decay. Δ τ_{ij} is the increment in pheromone trail according to Equation 5:

\[
\Delta \tau_{ij}(t) = \begin{cases} 
\frac{1}{F_{gb}} & \text{if } (i,j) \in \text{global best tour} \\
0 & \text{otherwise}
\end{cases}
\]  

F_{gb} is a fitness value corresponding to the global best tour within all past iterations. After a path has been built by the ants it can easily be expanded to include additional nodes. As pheromone placed by ants on paths toward the previous destination remains, then the only calculation required is to path from the previous destination to the new target destination. Compared to A* which has to restart its calculations to determine a path, ACO is better suited for dynamic pathfinding.

Previous work conducted has proven that ACO is capable of adapting for dynamic environments. The time taken for a ACO to adapt is shorter than the time taken to start over from scratch (Angus & Hendtlass, 2005). As such there is much work on adapting ACO in dynamic travel path (Kponyo, Kuang & Zhang, 2014) and dynamic vehicle routing (Montemanni, Gambardella, Rizzoli & Donati, 2005).

### 2.2.2 Particle Swarm Optimization

The Particle Swarm Optimization algorithm (Kennedy & Eberhart, 1995) is related to both swarm theory and evolutionary computation, where each "particle" in a swarm can be represented by an agent. The algorithm was first intended to simulate social behaviour. PSO works by optimizing a problem by iteratively trying to improve a candidate solution given certain circumstances. It solves a problem by having 'particles' move around a search space according to simple formulas controlling the particles position and velocity. Each particle in
the search space is influenced by its own best known position as well as the best known position in the search space. The effect of the global best and local best can be seen in \textbf{Figure 6}. By having each particle be affected by the best known position it hopes to achieve a better solution for the entirety of the swarm.

\textbf{Figure 6} \hspace{1cm} The effect of the personal best (pbest) and global best (gbest) on a particle's velocity

While capable of searching very large spaces, PSO still possesses several qualities differentiating it from other algorithms such as ACO. The difference between PSO and other optimization metaheuristics is that PSO never guarantees an optimal solution is ever found. PSO is very much like a genetic algorithm, an algorithm for solving optimization problems based on natural selection, where a population of individual solutions is repeatedly modified. In the PSO algorithm a population of particles is initialized with a velocity in order to be flown through the search space, in hopes of locating a solution. Compared to other algorithms PSO is a fairly easy to implemented and understand.

The particle swarm optimization algorithm was created in order to simulate social behaviour as such the algorithm works by utilizing knowledge of others' current performance. Particles adjust their movement according the best performance in the group hoping to achieve similar results. In a pathfinding scenario each particle would be informed of their distance to the goal, the group would then start randomly searching for the goal from their position. Each iteration the each particle would adjust its movement to move towards the particle with the current best position and the best position the particle has achieved itself throughout all iterations. Pseudocode for PSO can be seen in \textbf{Figure 7}.
PSO begins by initializing several agents dubbed 'particles' that then search for a solution. Each particle is updated every iteration by using two "best" values, its personal best (pbest) and the global best (gbest). The pbest of a particle is stored by itself while the PSO algorithm stores the gbest achieved throughout any iteration. Each particle then updates their velocity and positions using two equations to determine their new movements:

\[ v[i] = v[i] + c1 \times \text{rand} \times (\text{pbest}[i] - \text{present}[i]) + c2 \times \text{rand} \times (\text{gbest}[i] - \text{present}[i]) \]  
\[ \text{present}[i] = \text{present}[i] + v[i] \]  

In comparison to other pathfinding algorithms, PSO does not determine a path by evaluating nodes. Instead PSO utilizes distances and fitness values of particles to move the entirety of the swarm towards a better solution. In the best possible scenario this would result in faster pathfinding than A*, as no neighbouring nodes would have to be investigated. During the worst possible scenario there is the possibility that the algorithm is unable to determine a single path. In order to implement PSO for pathfinding the fitness calculations were modified to reward particles in a suitable manner.

A strength of the PSO algorithm is its capability to determine optimal or near-optimal solutions in large search spaces quickly. When applied to dynamic environments or destinations the algorithms would have to be written in a manner where the global best is reset every so often when a new path is required. Following a previous best answer would result in outdated information. Carlisle and Dozier (2000) discuss in their article *Adapting Particle Swarm Optimization to Dynamic Environments* different time periods to reset the global best in order to achieve the best overall performance. There has also been other work completed on the study of PSO’s adaptation to dynamic applications with relative success (Liu, Wang & Yang, 2010; Blackwell, 2007).
3 Problem

This chapter will discuss the basis of this work and its relevance. It will also state the question that this work hopes to answer. Afterwards there will be a description of the implementation environment and algorithms that will be constructed in order to answer the aforementioned question. There will also be a portion discussing the required measurements to ensure that the results gathered maintain their validity.

3.1 Question

The aim of this project is to research the benefits of swarm intelligence algorithms, ACO and PSO, as pathfinding algorithms for use in video games. There are two methods for prioritizing pathfinding alternatives, the mathematical approach and heuristic approach, see chapter 2.1. This study focuses on the heuristic approach of calculating the best possible path for the lowest cost possible, where cost refers to the time taken to calculate the path. As such the algorithms are judged on their path optimality, also known as the path length, and cost. To ensure that the algorithms are applicable to video games, the algorithms were run in environments constructed using map data taken from the games Starcraft (Blizzard Entertainment, 1998) and Warcraft 3 (Blizzard Entertainment, 2002). These algorithms have already been utilized in their own respective fields (Kponyo, Kuang & Zhang, 2014; Blackwell, 2007), however their use within video games is still scarce if present at all.

Games today are often set in large dynamic environments with mobile units, as such there are requirements that the pathfinding system must fulfil. Current pathfinding methods such as A* are capable of meeting the desired criteria, however the need to recalculate paths from scratch as destinations and environments change takes an unnecessary amount of time. Determining a path should be done in a manner that allows NPCs to maintain their level of intelligence in the eyes of the player. Any pause an NPC experiences in order to calculate a path negatively affects how the player perceives the NPC.

This experiment aims to evaluate the effectiveness of ACO and PSO when utilized for pathfinding in games. The effectiveness in terms of games is the ability to quickly determine optimal paths for agents to traverse. For the experiment the measurements recorded are time and quality. Quality can be seen as path optimality. These measurements were used as they have been utilized in previous experimentations (Angus & Hendtlass, 2005), and have provided sufficient information to draw a conclusion regarding the effectiveness of path planning algorithms in dynamic environments.

The time is recorded as a measurement of efficiency. There are three different time values recorded: time taken to calculate the initial path, average time taken to calculate each subsequent path and the total time taken to complete the experiment. Due to how the ACO algorithm functions is will be important to note the differences in time to calculate the initial path and each subsequent path, as such they are measured separately.

In terms of path optimality games often have few demands. As long as a path isn't viewed as peculiar or moronic then it is often acceptable. However the ability to calculate a higher quality path in a shorter amount of time is preferable. The path length will be a measurement recorded to compare the correlation between time and quality. As A* is
capable of always determining the shortest path available, path lengths are compared to the \( A^* \) path length.

As done by Montemanni, Gambardella, Rizzoli and Donati (2005) in their experiment, on dynamic vehicle routing, it is also valuable to recorded the minimum and maximum values achieved by each algorithm for the experiment. This relates to the shortest and longest times taken by each algorithm and the shortest and longest paths throughout all runs of the experiment. It is important to be aware of the worst possible outcome of an algorithm as the average is not guaranteed to be achieved each run.

As \( A^* \) always determines the best path possible, if any, the time taken to calculate a path can sometimes be greater than what is desired, especially if this path has to be recalculated numerous times as the destination moves. Sometimes routes that are faster to calculate are preferred in games despite not being the best. PSO is capable of efficiently finding optimal or sub-optimal solutions in large search spaces (Carlisle & Dozier, 2000). It has already been established that PSO is functional in both dynamic and static situations, as such it has the potential to be functional in games. As PSO does not have to determine the best path possible it may be capable of reducing search times which is a desirable feature in games. Sometimes more obscure paths, within bounds, lead to more human-like behaviour.

ACO’s ability to work in dynamic applications makes it suitable for pathfinding in games. After having determined the first path it is possible that the time taken to extend the path, as the target moves, may be less than the time taken for \( A^* \) to calculate a new path. This can summarized with the hypothesis - If ACO is implemented for pathfinding towards a mobile destination then it will perform better than \( A^* \) after the initial path has been determined. There is no evidence to support that ACO would be capable of determining a path at the speed of \( A^* \), but there is evidence to suggest that ACO may be capable of updating the path dynamically faster than \( A^* \) can recalculate the path each time the destination moves (Angus & Hendtlass, 2005).

### 3.2 Method

The method chapter has been split into three parts according to the tasks that had to be completed. In order to test the swarm intelligence algorithms’ pathfinding abilities a suitable environment had to be created, this environment reflects an environment found in games. The experiment is then be discussed as well as the different aspects that are observed during experimentation. Lastly how the results are obtained as well as their validity will be discussed.

#### 3.2.1 The Environment

The first task for this experiment was to create a suitable environment to conduct the pathfinding experiments. As different methods of navigating an environment lead to different paths in a game it is important to apply a suitable method. As previous pathfinding studies by Sturtevant and Buro (2005) have utilized a grid consisting of square tiles the same was then used in this experiment.

In order to use environments that accurately represent environments found in games, map data from the games Starcraft (Blizzard Entertainment, 1998) and Warcraft 3 (Blizzard Entertainment, 2002) are used. Although the original maps may have been created with different block width and height settings, they were scaled to a size of 512 x 512 tiles. This
size is utilized as it is used in other similar studies (Sturtevant & Buro, 2005). By using actual game maps the experiment is able to accurately represent environments found in games. The size of the map also allows for greater distinction between the algorithms, using smaller maps generates results that are more difficult to measure.

The testing environment was created using the game engine Unity (Unity Technologies, 2017) using its inbuilt 2D perspective. A 2D perspective allowed for the results to be easily observed by looking at the environment from above. By seeing the entirety of the environment at once it becomes easier to grasp the possible paths algorithms can find as well as compare the paths taken by different algorithms.

### 3.2.2 The Experiment

In order to path around the obstacle the algorithm is able to create a path by traversing in one of eight directions. Each tile the algorithm wishes to traverse to can either be traversable or part of an obstacle. If the tile is part of an obstacle then the algorithm will be unable to create its path using that tile. The algorithm will have completed a path when it can determine an uninterrupted path from the start to the goal using traversable tiles.

Games often force enemy AI to path towards the player, since the player is a mobile unit the path often has to be re-determined as the player moves. As the goal of this experiment is to test the capabilities of the algorithms for pathfinding in games, a mobile agent is used to act as the destination, this agent will be referred to as the target. The pathfinding algorithms will determine a path for another agent, the seeker, from its current position to the target’s position.

Both the target and the seeker move at a constant speed, although the seeker moves slightly faster in order to allow the seeker to reach the target. As the agents move, their centre are used to determine what tile they currently occupy. Twice every second the seeker will have to re-determine a path to reach the target. During this period the seeker will be at a standstill until a new or updated path is completed. Should the seeker and the target occupy the same tile, the experiment is considered complete and the results are recorded.

The target’s movement is determined beforehand using waypoints. Once the target arrives at one waypoint it will begin traversing towards the next, utilizing the A* algorithm to create its path. Once all waypoints are reached it will begin from the first waypoint once again, thus allowing it to maintain its constant movement. These waypoints are manually placed. The target’s movement has it traversing the environment in a manner similar to a player exploring the map to obtain information.

In the terms of the ACO and PSO implementations there are variables that are set during the implementation that then remain constant throughout their runtime. These parameters are for the ACO: the amount of ants, $\alpha$, $\beta$, pheromone decay, and $p_0$ which are all described back in chapter 2.2.1. For PSO the parameters include the $c_1$ and $c_2$ parameters, where $c_1$ and $c_2$ control the importance of the global and personal best position respectively. Depending on how these variables are set the results achieved can be affected. In order to give an accurate representation of the algorithms’ capabilities test were run before the experiment to determine values for these parameters. The tests conducted allowed the algorithm to run on a map not utilized for the actual experimentation, with different combinations of parameter values. The combination of values that allowed for the algorithm to complete with the fastest search time were then used during the actual experimentation.
3.2.3 Results & Validity

The aim of this experiment was to test the effectiveness of the ACO and PSO algorithms for pathfinding in games. Actual map data from games has been utilized to set up the experiment environments in order to determine the algorithms' use for games. However all the map data used is taken from real-time strategy games, this means that the results gathered will reflect upon this specific genre. It is possible that different results may be obtained if using map data from other game genres where environments may have been built with different intentions. Also the map data is taken only from two games leading to a small sample size, as such results could possibly still vary within the same genre.

The experiments are also conducted in a grid based environment, consisting of square tiles, as such the results are not relevant for all studies of pathfinding. Despite results gathered from this experiment it would still be possible that another outcome could be achieved in a different environment. Grid based systems often have large search spaces in comparison to other navigation environments such as navigation meshes (Snook, 2000).

When the experiment is conducted each algorithm runs in each environment several times. As it is not guaranteed that the swarm intelligence algorithms will determine the same path each time the program is run, several tests must be conducted to determine an average. It is also entirely possible that one of the algorithms fails to determine a path at all, by conducting several experiments on the same environment the probability of this failure can be determined.

One of the aims of the experiment was to determine the algorithms' efficiency. This is done by measuring the time taken to calculate the initial path, each subsequent path and the total time for the experiment. All these time measurements were be recorded using Unity's (Unity Technologies, 2017) inbuilt timer.

The experiment could have been conducted differently by instead using qualitative research with testers. By conducting an experiment where the testers would be allowed to observe the pathfinding algorithm's movement, and then asked to answer a questionnaire or interview, a different perspective could have been obtained. As games are created with the player in mind it is important that the player's immersion does not break due to something such as bad pathfinding. Earlier chapters of this report even mention games wanting to avoid certain AI movement as it is deemed unrealistic, see chapter 2.1, yet this characteristics is not measured in this study. Rather than considering the player's experience or immersion, regarding the pathfinding, this study solely focuses on the technical aspect. This methodology was utilized due to its previous use by researchers conducting research on similar topics (Angus & Hendtlass, 2005).
4 Implementation

This chapter discusses the implementation process of the study. There is a part discussing the experiment environment created to run the pathing algorithms. There is also a discussion on the implementation of each algorithm that brings to light implementation decisions made in order for the algorithms to better function in the desired context. There is also a pilot study that showcases how parameter values were assigned for the SI algorithms.

4.1 Experiment Environment

The focus of this project was to analyse the capabilities of the swarm intelligence algorithms for pathfinding in games. As such the majority of the project, in the manner it is conducted, relies on drawing conclusions from the results obtained during experimentation. Therefore the experimentation environments graphical aspect was not prioritized. In order to easily create an experiment environment, the environment was implemented in the C# programming language using the Unity game engine (Unity Technologies, 2017). The Unity engine provides tools to easily create different scenes for different experiments allowing more time to be focused on the actual implementation of the algorithms.

All experiment environments were built using actual map data taken from the games Starcraft (Blizzard Entertainment, 1998) and Warcraft 3 (Blizzard Entertainment, 2002). A range of maps of different maps were chosen based on different values in regards to the amount of traversable tiles present on the map and the longest path possible from one point to another, when taking the shortest path. Using maps with different characteristics and values allowed for the algorithms to be tested in different scenarios. By using actual map data, rather than self-created maps, it will be easier to tie in the algorithms' capabilities corresponding to video games. Map data and information regarding map benchmarks are taken from Nathan Sturtevant's (2012) repositories.

The environments were created with a two-dimensional array consisting of 512 tiles, or nodes, in both the x and y axis. As not all maps from the Starcraft and Warcraft 3 games are originally in the 512 x 512 format, some have been scaled to fit this grid size. The 512 x 512 tile size was chosen due to its previous use in similar studies where this grid size provided satisfactory search space to accurately test the pathing algorithms (Sturtevant & Buro, 2005).

The environment consist of two types of terrain, ground and obstacle. Each tile will be designated one terrain type represented by a bool value, with true equating to ground and false to obstacle. The seeker and target will only be able to traverse tiles of the type ground and will have to path around obstacles. The map data contains several types of terrain such as tree, swamp, water, ground, obstacle and out of bounds. For this study the goal is to be able to produce a path using traversable tiles. Therefore the environment was only concerned with reproducing the game map using the ground and obstacle terrain types. Terrain types that require specific units or interactions in order to be traversed, such as water and tree, are therefore counted as obstacles, the effect of this decision can be seen in Figure 8.
Figure 8  Conversion of map data to the game map, white tiles are traversable ground terrain while black tiles are obstacles.

The graphical aspect of the experiment was created using simple sprites. Sprites are used to illustrate tiles as they have low requirements in order to be rendered. As this study focuses on the algorithms, no time will be spent on optimizing the visualization of the grid, therefore using a cheap solution is desirable. At each node’s location a sprite was placed with its colour depending on the sprite’s terrain type. A white sprite represents a traversable tile while a red sprite is an obstacle. After the seeker has reached the target’s position, all tiles the seeker took to reach the target will be changed to a different colour depending on what algorithm was used. An example of a completed run can be seen in Figure 9.

Figure 9  A completed A* run. Blue tiles are the path taken by the seeker to reach the target.
4.2 The Seeker and The Target

The seeker and target are both capable of moving to any of the eight neighbouring tiles, granted that they are traversable. Horizontal and vertical movements will both have a cost of 1, whereas diagonal movements will have a cost of 1.4, the approximate value of $\sqrt{2}$. This is the same octile distance heuristic used by Hernandez and Baier (2011) and many others.

Pathfinding is not a cheap resource, therefore games only want to update a path when required. Generally pathfinding is updated based on events, in this experiment the event would have been when the target changed position. However this would require far too many updates and calculations due to the large size of the grid. While it would have been possible to have the path recalculate after the target had travelled a certain amount, such as 10 or so tiles, it was instead decided to update the path based on a time frequency. As the time taken for the target to travel 10 tiles may not remain constant due to diagonal movements, the frequency allows for the path to always update at a constant pace. The path will update twice every second, or every thirty frames. Due to the larger scale of the map half a second seemed as a adequate amount of time for the algorithm to determine a path and perform some movement.

The experiments were run until the seeker managed collide with the target. To ensure that the seeker could reach the target, the seeker was given twice the movement speed than that of the target. This is the same principal used by Moldenhauer and Sturtevant (2009) in their article regarding the classical game of cops and robbers. By granting the seeker a greater movement speed than that of the target it prevents a stalemate scenario where the seeker chases the target, but is never able to catch it despite optimal pathing. For the experiments the seeker was given a movement speed of 30 units per second and the target a movement speed of 15 units per second.

In Starcraft, and many other games the map is often covered by a gray 'fog' deemed the fog of war. Fog of war refers to the lack of vision and information on areas of the map that do not contain a friendly unit. Explored areas that have since been abandoned show terrain and buildings in their last known state. The fog of war often encourages players to explore the map in order to gain information, the target's movement will be based on an AI whose objective is to gain information by exploring the fog of war. By having the target's movement resemble possible actions taken by either real players or AI it further increases the validity of the experiments connection to video games.

Waypoints will be placed on the map for the target to traverse between, there will be enough waypoints placed for the target to explore all key points on the map. After the target has reached the last waypoint it will once again return to the first waypoint and then repeat the path. The waypoints for each environment will always be traversed between in the same order for each experiment run. The target's path, between two waypoints, will be calculated using the A* algorithm. Possible pathing for the target can be seen in Figure 10.
Regarding the starting position of the seeker and the target, these positions will be pre-defined for each map. Utilizing random spawns allows for the performance of the algorithm to be tested in several scenarios (Moldenhauer & Sturtevant, 2009), however the swarm intelligence algorithms are not guaranteed to execute the same performance each time the experiment is run. Therefore the experiment will instead be repeated several times with constant values, regarding the starting positions, to measure their best and worst as well as average performance. This allows for a more accurate representation of the algorithms’ performance to be achieved.

4.3 Implementation of Ant Colony Optimization

The ACO algorithm was implemented using the pseudocode and algorithms showcased in chapter 2.2.1. Other sources such as an ACO implementation for the travelling salesperson (Brownlee, 2011) and a article discussing the algorithms’ complexity in each step (Dorigo, Maniezzo & Colomii, 1991) were also used to gain a better understanding of the algorithm. The algorithm has however been modified to work in dynamic environments by normalising the amount of pheromone on a tile.

In order to avoid having several arrays for different values relating to one ant there was a separate class created to store information regarding a single ant. The information stored in this class includes the current node the ant occupies, its previous node, its path so far and the ant’s fitness. An array will then store all ants in the colony.
There are also two lists utilized in the implementation to keep track of the ants. One of the main portions of the ACO algorithm is iteratively having ants chose a new position to move towards until they reach their target destination. Once an ant reaches the destination, that ant will then be marked as complete and will no longer have to be included in the loop. As such there is a list to keep track of which ants have yet to complete their path and which ones have. The list keeping track of completed ants is merely used to track what ants to remove from the prior list as this cannot be done during runtime. The complete list is cleared once all ants have been removed from the not complete list. As the size of these lists will change dynamically during runtime an array seemed unfitting as the size of an array is declared upon initialization.

When deciding upon what node to explore next the ACO algorithm utilizes problem specific information. When applied to the travelling salesperson problem the problem specific information becomes distance between cities (Dorigo & Gambardella, 1997). Similar to the travelling salesperson problem, the distance between a neighbouring tile and the targets location will be used for the problem specific information. The distance used will be the distance of the tile positions in the grid and not the world position, as such the movement cost put in place can be utilized.

An ant decides which neighbouring tile to travel to by observing the neighbouring tiles distance towards the target as well as the amount of pheromone on the tile, see algorithm 1 (chapter 2.2.1). This greedy selection often favours tiles with the best combination of distance and pheromone levels. In order to balance out the greedy selection of the algorithm there is always a possibility that an ant will travel towards a random neighbouring tile, see algorithm 2 (chapter 2.2.1). This random movement by an ant promotes greater exploration during the earlier stages where concrete best paths have yet to be found allowing for other solutions to be attempted. However the amount of random movements executed by an ant is lowered during later cycles of the algorithm where more pheromone has been placed on tiles. While exploration is useful during earlier stages in order to explore more possibilities it does slow down the algorithm by having an ant run about randomly when a traversing a path that has already been established.

One issue encountered with the ants movement was where an ant would get 'stuck' between two tiles. In these scenarios an ant would move from tile \( i \) to tile \( j \) and then from \( j \) to \( i \) repeatedly until the pheromone on the tiles lowered enough to make other tiles more attractive. The occurrence was caused by tiles sometimes having far greater pheromone levels than that of neighbouring tiles, due to being perceived as part of the best path for several path calculations, see Figure 11. This repeated movement between two tiles increased the ant's path drastically in some cases. In order to counteract this type of movement any node which the seeker passed over would have their pheromone level reset to the initial pheromone level. By resetting the pheromone level on tiles that the seeker had traversed the ants would begin to explore other tiles and continue building a path reaching the target.
One of the big changes for the ACO implementation in comparison to standard ACO implementations is how pheromone is handled. In a static environment the algorithm would run until some criteria was met and a single path was found. However in a dynamic environment the algorithm may be run several times leaving rather high values of pheromone on several tiles. This increase in pheromone may be prevent certain solutions to be found in later stages. Due to this, pheromone levels are normalised (Angus and Hendtlass, 2005). Normally the $j$ pheromone level of tile $i$ would be calculated by $\tau_{ij}$, this is then replaced by $\tau_{ij}/\tau_{i_{max}}$ where $\tau_{i_{max}}$ is the maximum level of pheromone on any of tile $i$'s neighbours.

4.4 Implementation of Particle Swarm Optimization

The implementation of the PSO algorithm was implemented using the psuedocode and algorithms shown in chapter 2.2.2. Other psuedocode implementations (Blackwell, 2007) were also studied in order to gain a better understanding of the algorithm and possible alterations. The PSO algorithm has been modified to include diversification of the swarm to allow for more solutions to be explored.

Similar to the ants, particles have their own class in order to store information such as position and velocity. For the study there will be 8 particles in a swarm as this allows for a particle to be initialized in each of the neighbouring tiles. The starting velocity of a particle will be the required velocity to traverse from the seeker's position to the particle's determined starting position. Should a tile not be traversable then that particle will be initialized on the seeker's position, this particle would then be given a starting velocity of 0 and have to wait until the next velocity update in order to begin traversing the grid. By spreading the starting positions of the particles a greater diversity should be achieved in comparison to having all particles start on the same position. As the goal of the PSO algorithm is to move the swarm as a whole towards a better solution having the particles
converge is expected, but having the particles only converge once reaching the destination is preferred. While it would have been possible to have a randomized starting velocity this would lead to far too inconsistent results.

There are two factors to keep particles from converging: fitness and distance from other particles (Liu, Wang & Yang, 2010). In their study Liu, Wang & Yang (2010) describe a possible solution to prevent particles from converging by having the current particle move then check for nearby particles and then move them by the same amount, see Figure 12. In a case where two particles facing each other have opposite velocities the particles would in the end remain at the same position. Once achieved a scenario such as this will continue until a new global best is achieved thus altering the paths of the particles or a particle's positions is altered by a third particle.

This solution works fairly well when applied to standard PSO implementations and is utilized for this study in order to delay convergence. However as this study uses tiles where the algorithm will not be able to traverse, convergence still occurs when a particle is unable to diversify. As such a second measure is also implemented; When a particle attempts to move into a obstacle tile then their fitness will penalized by adding an additional sum to the fitness calculation. Generally fitness is calculated by determining the distance from the current position to the destination. This in turn should make the current position more attractive and strive to have the particle explore other areas further away from the obstacle.

![Figure 12](image_url)

Figure 12  Particle position adjustment: Black, Green and Blue are all particles. Step 1: Black calculates a new velocity. Step 2: Black moves to the new position. Step 3: Green's position is adjusted according to Black's velocity.

The velocity of a particle is updated according to the algorithms in chapter 2.2.2, using the global best position and the personal best position. After a particle's new velocity has been determined that velocity will then be applied to determine the particle's new position. First a check is run to determine whether the new desired position is ground terrain, or traversable, to ensure that movement to the position is possible. Should the desired position not be traversable, a new random position will be chosen instead. This random position will be a neighbouring tile to the current position and must be traversable.
One of the challenges of using PSO for dynamic environments is controlling the global best. As the environment is dynamic the global best for one iteration of particles may not be optimal ten iterations later, and if no better best has been achieved then that poses a problem. A possible solution proposed by Carlisle and Dozier (2000) is resetting the global best value after 'X' amount of iterations to ensure that the current global best accurately reflects the current particles. Blackwell (2007) discusses that the algorithm must either be aware when the current global best value no longer represents a suitable value for the swarm or it must be capable of detecting change in the environment forcing change. This project will utilize the latter solution where an update in the environment forces change. To implement this the global best will be updated whenever the path is re-calculated.

One of the biggest weaknesses of having the algorithm reset the global best position each time the path is recalculated, is the increased difficulty for the algorithm to path out of a blind alley. The spread out initial positioning of the particles does aid the algorithm in pathing out of the blind alley however additional aid is often required if the target's position is not favourable. When stuck in an blind alley the algorithm will often lead the seeker to traverse all the tiles in the alley several times. As such another feature was implemented to aid the algorithm where each tile has a counter measuring how many times the seeker has traversed that tile. This counter value will then be included in the fitness value of each tile that the algorithm observes. The more a tile is stepped on the less attractive it will become and the algorithm will begin to path outwards as the global best value appears further and further out of the alley due to those tiles presenting more attractive alternatives.

4.5 Implementation of A*

A* is one of the most well renowned search algorithms, therefore there is plenty of implementation examples and psuedocode available online, the algorithm also has plenty of different variants, however the core concept remains the same. The implementation of the A* algorithm was done using the psuedocode in chapter 2.1.1 and Mat Buckland’s book Programming Game AI by Example (2005).

While the actual implementation of an A* algorithm is fairly straightforward the real challenge of an A* implementation is the optimization of the algorithm. One problem with implementations of A* when using the Manhattan distance in octile graphs is the use of overestimating heuristic. An overestimating heuristic is one that overestimates the cost from the current node to the goal. The Manhattan distance returns the sum of the vertical and horizontal deltas however when used in an octile graph, where diagonal movement is possible, an overestimation occurs. An inflation arises in the H cost decreasing the value of the G cost. This causes the algorithm to explore lower H values over lower G values. In order to use a more accurate representation the octile distance is used. The difference in the distance calculation between the methods can be seen in the equation below:

\[ \text{abs} \left( x_1 - x_2 \right) + \text{abs} \left( y_1 - y_2 \right) \]  \hspace{1cm} (1)

\[ \min(dx, dy) \times Dc + \left( \max(dx, dy) - \min(dx, dy) \right) \]  \hspace{1cm} (2)

Equation 1 above is the standard Manhattan distance calculation whereas Equation 2 is the octile distance. For the octile distance the smaller value out of the two deltas is multiplied by the diagonal cost, Dc. That is then added to the distance of the smaller delta subtracted from the greater delta. The differences in H cost calculations between the distance calculations can be seen in Figure 13. While it would have been possible to use standard operations
involving the square root operator, the square root operator increases the calculation cost. As the majority of the A* algorithm computational time is spent in the heuristic function it is critical that the function is as lightweight as possible. The octile distance equation allows for cheaper calculations which preferable as the function is called upon repeatedly during execution.

![Manhattan Calculation](image1.png) ![Octile Calculation](image2.png)

**Difference in X coordinate** = 7  
**Difference in Y coordinate** = 3

Distance = $7 + 3 = 10$  
Distance = $3 \times 1.4 + (7 - 3) = 8.2$

Figure 13  Difference in H cost calculations using Manhattan and the modified calculation.

### 4.6 Pilot Experiment

In order to test whether it is possible to evaluate the algorithms, a pilot experiment has been run. This pilot experiment also served to find optimal parameter values for the ACO and PSO algorithms. In this pilot study the Warcraft 3 map - "Frost sabre", was constructed for the experiment environment. This map was chosen for possessing fairly average values in terms of longest possible map length and percentage of traversable tiles on the map. The map can be seen in Figure 14.
Figure 14  Warcraft 3 map - Frost sabre. White tiles are ground terrain while red tiles are obstacles.

The first algorithm tested was the ACO algorithm, which needed to determine values for the $\alpha$, $\beta$, pheromone decay, $p_0$ and the number of ants. In a previous study by Gaertner and Clark (2005) optimal parameter values were determined by running several tests on different parameter combinations. In their experiment each parameter was given a lowest and highest value to be tested and then a step in which the parameter value was increased or decreased. Each possible combination of parameters was then tested taking several weeks to complete all tests so that the best combination could be determined. Due to the time constraint of this study parameters were instead given three values to be tested: a low value, an average value and a high value. A previous study by Montemanni, Gambardella, Rizzoli & Donati (2005) was used for reference in order to determine adequate values as that study focused on similar goals of achieving fast calculation times. Each combination of parameter values was then run with the combination garnering the best result being used in the study going forward. After having completed the pilot testing of the ACO algorithm the best parameter combination was determined to be: $\alpha = 0.3, \beta = 0.7$, pheromone decay = 0.1, $p_0 = 0.02$ and using 3 ants.

The study indicated that the number of ants was the greatest influence on the performance of the ACO algorithm. The pilot study completed experimentation using three different amounts of ants: 3, 5 & 8. When using more ants the time to calculate a path and update each subsequent path increased however the change in path optimality was often very little.
In order to increase the path optimality even more ants would most likely be required. As this study focuses on having cost effective paths the ACO algorithm will use 3 ants. While this result indicates that the time may be improved even more by decreasing the number to only using 1 or 2 ants. Using 1 ant would completely remove all forms of exploration from the algorithm whereas using 2 ants would only decrease calculation time minimally in comparison to the lost path optimality when only two options are explored.

Thereafter the PSO algorithm was tested. For the PSO algorithm the values of the c1 and c2 parameters must be determined. C1 and c2 are both constants that are used to adjust the influence of the personal best position and global best position respectively when determining the new velocity. As there are only two parameters to be tested in the PSO algorithm, more values were tested. Both c1 and c2 were tested in the range of 0 to 2, with an increment of 0.4. These ranges were used due to inspiration from previous studies (Liu, Wang & Yang, 2010). The same concept was utilized on PSO’s tests where every combination was tested and the best performing combination was utilized. For PSO the best parameter combination was C1 = C2 = 0.4.

Results from the pilot experiment for all three algorithms, using the determined best parameter value combinations for ACO and PSO, can be seen in Figure 15.

![Figure 15](image_url)  
**Figure 15** Results from pilot study: blue path is A*, green path is ACO and purple path is PSO, yellow tiles are where paths overlap. The image is zoomed in order to easily see the paths.

The results seen in Figure 15 above illustrate how the algorithms do not find the same path when pathing towards the target. Due to the algorithms reaching the target during different points in time, they have different end destinations. Despite how many times the A* algorithm is run, it will always determine the same path and reach the target at the same point. The ACO algorithm will determine a path that is fairly similar each time with minor adjustments. The PSO algorithm however will most likely determine a different path each time, this is due to the different influences of the personal and global best as they can be adjusted randomly, see chapter 4.4.
One apparent weakness in the ACO algorithm is its inability to path away from tiles with already well established pheromone counts. Currently there is a solution in place to reduce the influence of extreme pheromone counts on tiles, however values such as the starting pheromone and maximum amount of pheromone may have to be adjusted in order to allow for more dynamic pathing.

During this pilot study measurements of initial time to calculate a path, subsequent time to calculate a path and time to complete the experiment can be seen in the Unity editor. As these measurements are currently only seen via a debug log in the console they can be disabled to increase performance when testing the pathing. The path length of the seeker can be obtained visually by observing the completed path however this information is also available by checking the count of the list containing the seeker’s path. For the actual experiment all values will be stored in a text file once an experiment is complete.

In chapter 3 several values that will be measured are mentioned, these being what the algorithms will be evaluated on, based on the pilot experiment conducted so far all those values are obtainable using the current experiment format. Thus using that obtainable data this pilot study indicates that it is possible to evaluate the effectiveness of the ACO and PSO algorithms for pathfinding in games.
5 Evaluation

This chapter will list the results obtained through the experimentation method described in previous chapters. The study's measurements will be presented in an easily read table as well as graphs illustrating the difference in results. Figures highlighting pathing differences will also be presented. There is then an analysis of the obtained results discussing any correlation between them. Finally there is concluding statement regarding the issue presented in this study.

5.1 The Study

For this study there are 5 different maps from the games Starcraft (Blizzard Entertainment, 1998) and Warcraft 3 (Blizzard Entertainment, 2002) used as experiment environments. Maps were chosen based on two characteristics: percentage of the map that is traversable and the longest path possible from one point to another on the map. These statistics and map data were taken from Sturtevant's repositories (2012). Maps are chosen with varying values in order to cover a greater range of game scenarios. This eliminates the possibility of an algorithm only being run in an advantageous environment and allows a more general understanding of the algorithm’s performance to be obtained. The maps chosen and their statistics can be seen in Table 1.

<table>
<thead>
<tr>
<th>Map</th>
<th>Traversable Tiles</th>
<th>Largest Single Path</th>
<th>Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Battleground</td>
<td>90166</td>
<td>595</td>
<td>Warcraft 3</td>
</tr>
<tr>
<td>Caldera</td>
<td>164789</td>
<td>786</td>
<td>Starcraft</td>
</tr>
<tr>
<td>Crescentmoon</td>
<td>122296</td>
<td>792</td>
<td>Starcraft</td>
</tr>
<tr>
<td>Scorchedbasin</td>
<td>80848</td>
<td>609</td>
<td>Warcraft 3</td>
</tr>
<tr>
<td>Stromguarde</td>
<td>76556</td>
<td>567</td>
<td>Warcraft 3</td>
</tr>
</tbody>
</table>

In order to ensure that the results obtained accurately reflected the algorithms' performance each algorithm was run on each experiment environment 20 times. After each experiment there were three cost values recorded as well as the path length. The three cost values were game completion time (ms), initial path calculation time (ms) and average path calculation time (ms). As it was hypothesised in chapter 3.1 that the ACO algorithm would be capable of updating its path at a greater speed than A* algorithm, the initial path calculated must be separated from the subsequent path calculations as the initial calculation is expected to take a greater amount of time and would only serve as an outline in the results. The game completion time is recorded in order to measure correlations between the path length and path calculation times, as it is possible that an algorithm may have a shorter path yet a longer calculation time leading to a longer game completion time. Once all experimentation was complete the results were recorded graphed using box plots seen in Figure 16, Figure 17, Figure 18 and Figure 19.
Figure 16  Results for the Initial Time (ms) to calculate a path.

Figure 17  Results for the Average Time Taken for each Subsequent Calculation (ms).
Due to certain values in the box plots making the graphs unreadable new versions of the graphs were created without these outliers. For the time calculations, initial time and subsequent time, A*'s results on Scorchedbasin have been removed, see Figure 20 and Figure 21. Whereas for path length and game completion time the maximum time and path length of PSO on Caldera is a onetime occurrence outliner. These graphs have also been
remade without that outliner, see Figure 22 and Figure 23.

Figure 20  Results for the Initial Time (ms) without A*'s Scorchedbasin results.

Figure 21  Results for the Average Time Taken for each Subsequent Calculation (ms) without A*'s Scorchedbasin results.
5.1.1 Battleground

Battleground is the largest Warcraft 3 map used in the study and is the middle map, out of all five, in terms of the amount of traversable tiles. The map is composed of several small to medium sized obstacles, the map boundary is jagged and consists of several dead-alleys. The Battleground map can be seen in Figure 24.
On the Battleground map both the ACO and PSO algorithms outperformed A* when determining the initial path. The A* algorithm always determines the shortest path possible at any given moment, and due to that characteristic it often has to investigate several different alternatives. Whereas the ACO and PSO algorithms simply determine a single path, regardless of that path’s optimality. As the Battleground presents several obstacles for the algorithms to path around the A* algorithm requires a greater amount of calculation time than the SI algorithms.

The SI algorithms both also manage to outperform the A* algorithm in subsequent path calculation speed. While it was hypothesised that ACO would be able to outperform A* in subsequent calculation speed, PSO was also capable of doing so calculating each subsequent path at an average of 10ms compared to A*’s 14ms. ACO achieved the fastest subsequent calculation time with 3ms. As the ACO algorithm, to some extent, merely extends the previously determined path it is able to achieve a lower calculation time.

While the ACO algorithm was unable to achieve a path length less than that of the A* algorithm, the PSO algorithm was able to do just that. When calculating a path A* will always determine the shortest path at that moment as mentioned before, the PSO algorithm will however always determine a path based on the influence of the swarm’s particles. A factor not included in either algorithms calculation is the target’s movement. On the Battleground map PSO is able to achieve better results than A* due to the target’s movement. While A* determines the shortest path at a certain point in time to be pathing around a obstacle, PSO
instead manages to charge towards the target thus achieving better results. An average representation of the path taken by each algorithm can be seen in Figure 25.

5.1.2 Caldera
Caldera is the largest map present in the study having a total of 164789 traversable tiles. The map contains six medium sized obstacles and one large obstacle in the middle splitting the
map. Edges are fairly smoothed in comparison to other maps. The map can be seen in Figure 26.

![Starcraft map: Caldera](image)

**Figure 26**  Starcraft map: Caldera

On the Caldera map A* achieved the worst results on average in all categories. As mentioned in chapter 5.1.1 this is due to the A* algorithm determining the shortest path at a given moment in time. The SI algorithms were able to perform the initial path calculation 5ms faster than that of the A* algorithm. ACO then managed to perform each subsequent calculation at 5ms compared to the 13.5ms of A*.

While ACO and A* produced consistent results in terms of path length and game time on the Caldera map, the PSO algorithm proved highly inconsistent. Despite the algorithm being capable of achieving a shorter path length than either of the other algorithms, it was also able to achieve a path length almost 175% longer. While the A* and ACO algorithms have rather linear movement, the PSO algorithm often travels in a diagonal manner. When colliding with an obstacle the algorithm will then often have to reflect of the obstacle in order to advance. Sometimes the algorithm will collide with an obstacle numerous time before being able to advance. PSO's movement is controlled by the personal best of a particle and the global best, see chapter 2.2.2, which are altered by a random value that can either increase or decrease their influence. Due to this randomness PSO can achieve different results when attempting to path around an obstacle depending on the alterations.
The PSO algorithm was able to at best achieve a path length of 296 tiles, an average of 332 and a longest path of 776 tiles. This is in comparison to A*’s path of 503 tiles. On the Caldera map the ACO algorithm was also able to determine a shorter path than that of the A* algorithm. When determining a path the ACO algorithm often simply functions by expanding upon a previously determined path by following pheromone trails, see chapter 2.2.1. However as the target moves further away from its starting location, the less accurate the path becomes in terms of optimality. As such ACO’s path will slowly start to explore new path alternatives along the current established pheromone path. In this specific scenario the ACO is able to path around an obstacle in a different manner than A*. This is due to the late update of the path, thus allowing the algorithm to achieve a shorter path length. The difference in the path taken by the algorithms can be seen in Figure 27.
Figure 27  Visualization of average path taken by each algorithm on the Caldera map. The blue path is A*, green is ACO and purple is PSO.
5.1.3 Crescentmoon

Crescentmoon is the second Starcraft map and the second largest map used during the study. The map has the greatest single path length out of all the test environments with 792 tiles for a single path. Crescentmoon contains very few obstacles instead having obstacles of greater size, that are for the most part attached to the maps edges. The map can be seen in Figure 28.

The PSO algorithm had its worst performance on the Crescentmoon map. The decreased performance of the PSO algorithm indicate that it has difficulty pathing around larger obstacles. The reasoning for the PSO’s decrease in performance was mentioned in chapter 5.1.2. The Crescentmoon map consists of one large obstacle in the middle, therefore there are no shortcuts for the PSO algorithm to utilize in order to obtain better results than the other algorithms.

A* showcased a greater performance on this map in comparison to the two previous as there were less obstacle on the map, leading to less pathing alternatives when calculating the optimal path. Therefore A* achieved a average subsequent calculation time of 3ms, however this time was still beaten by ACO’s 2ms for average subsequent calculation time. Difference in the average path taken by each algorithm can be seen in Figure 29.
Figure 29  Visualization of average path taken by each algorithm on the Crescentmoon map. The blue path is A*, green is ACO and purple is PSO.
5.1.4 Scorchedbasin

Scorchedbasin is the second smallest map used during the study but despite that it still has a greater single path length than Battleground which has just below ten thousand more traversable tiles. The map consists of several dead-alleys around all edges of the map and several medium sized obstacles in the middle. As four of these obstacles are U shaped an algorithms' calculation time would increase greatly should they attempt to path into these obstacles. The map can be seen in Figure 30.

![Figure 30 Warcraft 3 map: Scorchedbasin](image)

Due to the amount of obstacles on the map the A* algorithm required a great deal of time in order to calculate a path. As such A*, despite having a shorter path length than ACO, required a greater amount of time to complete the experiment. This environment showcases the strengths of the SI algorithms as both ACO and PSO were able to have a average calculation speed of roughly 10ms whereas A* averaged 184ms.

As this map consists of smaller to medium sized obstacles, the PSO algorithm is able to achieve greater results. Any collision the algorithm has on this environment is quickly resolved allowing the algorithm to progress to open areas where it is able to path at a great speed. The amount of traversable tiles on the Scorchedbasin map is less than the other environments causing a smaller deviation in results among the algorithms. The average path taken by each algorithm can be seen in Figure 31.
Figure 31  Visualization of average path taken by each algorithm on the Scorchedbasin map. The blue path is A*, green is ACO and purple is PSO.
5.1.5 Stromguarde
The last map used as an experiment environment was the Warcraft 3 map Stromguarde, the smallest out of all the maps with only 76556 traversable tiles. Out of all maps Stromguarde has the most obstacles present on the map, however the obstacles are smaller than the obstacles on other maps. There are also several wide open areas on the map. Compared to other maps, Stromguarde presents a more clean look with fewer obstacles being constructed in a diagonal manner. The map can be seen in Figure 32.

Figure 32  Warcraft 3 map : Stromguarde

Despite achieving a shorter initial path calculation time and subsequent calculation time than the A* and PSO algorithms, the ACO algorithm had the longest game time and path length. This is again a cause of the delayed path update that ACO experiences due to following a already well established pheromone trail.

The PSO algorithm remained fairly consistent on all previous maps, in regards to the subsequent path calculation time, however on this environment the average time required increased. This indicates that the PSO algorithm had difficulty pathing around the increased amount of smaller sized obstacles. While A* required the most time for the initial calculation it still managed to achieve faster subsequent calculations than PSO. The paths taken by the algorithms can be seen in Figure 33.
5.2 Analysis

5.2.1 Calculation Time
The results seen in chapter 5.1 demonstrate that, in terms of calculation speed, the ACO and PSO algorithms performed better than the A* algorithm in the dynamic test environments. This was more apparent on maps with a greater amount of small to medium sized obstacles. Whereas on maps with less obstacles and more space the calculation speeds were more equivalent. As the A* algorithm always determines the shortest path available the amount of tiles inspected by the A* algorithm is greater than the amount of tiles inspected by either the ACO or PSO algorithm resulting in a greater search time. In open environments or environments with few obstacles this is less apparent.

The PSO algorithm showcased a rather identical performance on all environments except Stromguarde indicating that as the amount of obstacles increased, despite their decrease in size, so did the time taken for PSO’s calculations. For the ACO algorithm there was a slight time increase on the Scorchedbasin map, same as the A* algorithm, while all other maps maintain a fairly similar time. The results for initial path calculation time and subsequent path calculation time can be seen in Table 2.
Table 2 Results for Initial Time (ms) and Average Subsequent Time (ms) after 20 experiment runs on each map for each algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Initial Time (ms)</th>
<th>Average Subsequent Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A*</td>
<td>ACO</td>
</tr>
<tr>
<td>Battleground</td>
<td>82</td>
<td>12.35</td>
</tr>
<tr>
<td>Caldera</td>
<td>17.5</td>
<td>12.1</td>
</tr>
<tr>
<td>Crescentmoon</td>
<td>22</td>
<td>13.1</td>
</tr>
<tr>
<td>Scorchedbasin</td>
<td>504</td>
<td>23</td>
</tr>
<tr>
<td>Stromguarde</td>
<td>118</td>
<td>14</td>
</tr>
</tbody>
</table>

5.2.2 Path Length
When comparing the path lengths of the algorithms A* shows to be the most consistent in achieving the shortest path. While both the A* and ACO algorithms tend to always achieve a similar path length each execution, the PSO algorithm has a great deal of inconsistency. The PSO’s path length inconsistency tends to be greater on maps with larger obstacles, thus being most present on the two Starcraft maps.

There was only one map that the ACO algorithm was unable to determine the same path consistently, Stromguarde. These results showcase how ants handle exploration versus exploitation. When there are obstacles that are smaller in size ants have a greater amount of exploration resulting in different path lengths. This is because there is a smaller gap in the distance from the altering paths to the targets location allowing ants to explore. Whereas larger obstacles often make the ants prone to consistently take the same path as the gap in distance between two paths may be far too great. On maps with larger obstacles ants can be seen as favouring exploitation despite not achieving path lengths equal to that of A*. A detailed view of the path length results are shown in Table 3 below.

Table 3 Results for Path Length, in tiles, after 20 experiment runs on each map for each algorithm.

<table>
<thead>
<tr>
<th></th>
<th>A*</th>
<th>ACO</th>
<th>PSO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Average</td>
<td>Max</td>
</tr>
<tr>
<td>Battleground</td>
<td>258</td>
<td>258</td>
<td>258</td>
</tr>
<tr>
<td>Caldera</td>
<td>503</td>
<td>503</td>
<td>503</td>
</tr>
<tr>
<td>Crescentmoon</td>
<td>236</td>
<td>236</td>
<td>236</td>
</tr>
<tr>
<td>Scorchedbasin</td>
<td>148</td>
<td>148</td>
<td>148</td>
</tr>
<tr>
<td>Stromguarde</td>
<td>162</td>
<td>162</td>
<td>162</td>
</tr>
</tbody>
</table>

5.2.3 Game Completion Time
Despite previous results where A* was outperformed in terms of calculation speed, it still manages to achieve the shortest game completion times due to the shorter path length. As with the path length, the ACO and A* algorithms achieve a fairly constant game completion
time with only a few milliseconds difference. However the PSO algorithm once again has a greater inconsistency in the time required to complete an experiment. The PSO algorithm also manages to outperform the A* algorithm on several maps due to different path choices resulting in shorter game times. For the most part the game completion times correlate to the path lengths. Recorded game time results can be seen in Table 4.

Table 4 Results for Game Time (seconds) after 20 experiment runs on each map for each algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Battleground</th>
<th>Caldera</th>
<th>Crescentmoon</th>
<th>Scorchedbasin</th>
<th>Stromguarde</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td>Min</td>
<td>Average</td>
<td>Max</td>
<td>Min</td>
<td>Average</td>
</tr>
<tr>
<td>Min</td>
<td>19,46</td>
<td>19,47</td>
<td>19,48</td>
<td>35,78</td>
<td>36,30</td>
</tr>
<tr>
<td>Average</td>
<td>22,14</td>
<td>22,15</td>
<td>22,16</td>
<td>31,93</td>
<td>31,94</td>
</tr>
<tr>
<td>Max</td>
<td>22,16</td>
<td>14,72</td>
<td>15,85</td>
<td>25,36</td>
<td>28,94</td>
</tr>
</tbody>
</table>

5.2.4 Behaviour

Whilst A* movement is based on the shortest possible path, it determines a path that is capable of moving in a single direction for as long as possible thus decreasing the required movement cost. ACO movement however tends to reach an obstacle and then travel along the edge of the obstacle until reaching an open area. This causes the ACO algorithm to take an extended path when exploring indents in an obstacle instead of taking the shorter path across. It is possible that with an increase amount of ants that a path ignoring the indent would have been found, however as showcased in chapter 4.6 this lead to an increase in calculation time that was not desired. This type of movement can be seen in Figure 34.

Figure 34 Illustration of different movement between A* (left, blue) and ACO (right, green).

Another movement habit of ACO is the delayed change in direction due to the algorithm following the already established path. ACO is capable of updating its path dynamically in a fast manner due to placing pheromone on established paths. However as the target changes position the algorithm continues to first path towards the target's old position then adding
the additional tiles required to reach the target's new position. As the target begins moving further and further away from its original position new paths begin to be found altering the seeker's movement. This is because as the target moves further away from its original position the distance between the tiles of the previous path and the target become too great to steer the target on pheromone alone, thus the path is updated. As the A* algorithm recalculates its path from the beginning each update this problem does not occur. A comparison between the ACO's path update and A*'s can be seen in Figure 35.

![Figure 35](image)

**Figure 35** Difference in path update between A* (left, blue) and ACO (right, green).

While the A* and ACO algorithms tend to have rather linear motion when possible, the PSO algorithm often moves in a more diagonal manner. In order to path around an obstacle the algorithm takes into account the amount of times a tile has been traversed, and as such tries to avoid tiles that have been traversed numerous times as they do not yield better results. However instead of then following along the obstacles edge like the ACO algorithm, the PSO algorithm instead reflects its movement of the obstacle and continues. The algorithm can sometimes collide with the same obstacle several times as it continues to attempt to move towards the target. This behaviour of PSO can be seen in Figure 36. This is where the algorithm proves to provide the most inconsistent results as sometimes the algorithm may not be able to path past the obstacle in the same manner as previous iterations. As the influence of the personal and global best are affected by a random value that may increase or decrease either's influence different results are expected. The greatest difference due to this random increase or decrease of either the personal or global best has been seen on maps with larger obstacles that take a greater deal of time to path around. The reason this inconsistency is seen to a greater extent on maps with larger obstacles is due to the fact that a slight alteration when pathing past smaller obstacles may have a minimal effect, however around larger obstacles this can cause a greater delay as more iterations are spent pathing around the obstacle.
Due to the different approach of path determination the PSO algorithm is able to outperform the A* algorithm in terms of path length in some cases. As the target moves the algorithms determine a path to the target’s position. A* will determine the shortest path possible from the seeker's current position at that moment. In certain scenarios this will lead the seeker to chase the target following it path, whereas PSO may instead opt to take a different path meeting the target head on. A*’s movement can be seen as robotic always determining a method based on the likelihood of achieving the best result at that moment which in some cases prevents it from determining paths that may result in shorter paths as a whole. A scenario where A*’s path leads to this type of behaviour can be seen in Figure 37. Had the target been pathing away from the seeker then it is more likely that the A* algorithm would have had a shorter path, however in games target’s do not always path away from a seeker.

5.3 Conclusions

In chapter 3.1 it is stated that the aim of this study is to research the benefits of the swarm intelligence algorithms, ACO and PSO, as pathfinding algorithms for use in video games. It is also stated that this study uses the heuristic approach to prioritize pathfinding alternatives, thus judging them based on path optimality and cost. Therefore an algorithm is judged on its
ability to quickly calculate a path while still maintaining a short path length. The focus of this study is for pathfinding within games therefore a greater emphasis is placed on calculation speed over path optimality. As the experiments were conducted in a dynamic environment the resulting path taken to reach the target destination was used for path length and the average time taken to calculate each path.

In chapter 5.1 the experiment was conducted according to the method documented in chapter 3.2. The results, which are analysed in chapter 5.2, showcase that the SI algorithms do in fact have attributes suitable for pathfinding in games and can in certain scenarios measure up against the utilized implementation of the A* algorithm.

One of the two characteristics judged in this study was the cost of calculating the path. In almost all scenarios the SI algorithms were able to calculate a path at a greater speed than A*. It was theorized in chapter 3.1 that the ACO algorithm would be capable of dynamically updating its path at a greater speed than A* would be able to recalculate the path, while the results indicate this to be true, for this experiment, it was also shown that the ACO algorithm determined the initial path at a greater speed as well. The difference in time taken to calculate a path was shown to be greater on maps occupied by a greater amount of small to medium sized objects.

The second characteristic judged in this study was the path length. On average the A* algorithm achieved shorter paths in comparison to both of the SI algorithms. There were however some scenarios where the SI algorithms achieved better results in terms of path length compared to the A* algorithm. In chapter 3.1 it was stated that games often have few demands in terms of path optimality, that the paths were merely required to not be viewed as peculiar or moronic. Therefore a slightly longer path that is calculated within a shorter timeframe is more desirable than the reverse. While neither path managed to consistently outperform A* in terms of path length their paths were still relatively close to the path length of the A* algorithm.

After having conducted the study the benefits of using the SI algorithm for pathfinding in games were determined for this study. The ACO algorithm is reliable, consistently being able to achieve the same result, and has the benefit of having low calculation times when pathing from one target to another. On maps with small to medium sized obstacles the algorithm proved the strongest when compared to A*, in terms of calculation speed. The algorithm did however tend to take longer routes than A* and these deviations would have to be considered during implementation. For this study a 512 x 512 grid size was used, in larger environments it is likely that these deviations, in path length, would increase.

While the PSO algorithm showcased itself to be less predictable, achieving a greater range of best and worst outcomes, it still provided information showcasing a benefit of using it to also be calculation speed. Despite being able to achieve shorter path lengths than A*, due to environment circumstances, the PSO’s unpredictability makes it less desirable to implement. However this study aimed to determine the benefits of the algorithms and the speed of the algorithm is just that.
6 Concluding Remarks

This chapter includes a summary of the research presented in this study. There is also a discussion regarding the results obtained and their possible effect on other fields. Finally there is a part discussing how it would be possible to expand on this study in future works.

6.1 Summary

The aim of this study was to determine whether there are any benefits to using either the ACO or PSO algorithm for pathfinding in games in comparison to well established algorithms such as A*. As pathfinding in games often involves determining a path towards a mobile target this study focuses on the dynamic aspect of pathfinding in games, where a path needs to constantly be updated.

In order to determine whether there are any benefits to using the algorithms, five experiment environments were set up. Out of the five environments, three were constructed using map data from the game Warcraft 3 and the other two using map data from the game Starcraft. Using the experiment environments each algorithm was tested by determining a path for a character, deemed the seeker, to reach another character, the target. After a path was determined the seeker would begin moving towards the target using the determined path. Meanwhile the target would continue to explore the map using a waypoint system. Twice every second the seeker would be required to update its path. The experiment would continue until the seeker reached the target's location. After all experiments had been conducted the algorithms would be judged on two characteristics; the path length and the average time to calculate a path.

Results of the experimentation indicated that the SI algorithms both were capable of determining a path at a greater speed than the A* algorithm. The ACO algorithm was capable of determining an initial path at a greater speed than the A* algorithm. This is a result of the fact that A* opts to determine the shortest path whereas ACO merely determines a path. After having calculated an initial path the ACO algorithm was proven to be capable of dynamically updating its path at a greater speed than the A* algorithm could recalculate a path. The difference in path calculation times proved greater on maps with a greater amount of smaller sized obstacles.

In terms of path length the ACO algorithm performed worse than the A* algorithm, whereas the PSO algorithm achieved both shorter and longer paths than the A* algorithm on the same map. The PSO algorithm performed with a great deal of inconsistency, where maps that contained larger obstacles often proved to have an increase in inconsistency. Part of the PSO algorithm is a random value that either increases or decreases the influence of the personal or global best value which are utilized for velocity adjustment. When pathing around smaller obstacle a small adjustment often proves to have minimal impact however when pathing around larger obstacles this slight change can have a greater effect on the time taken to path around the obstacle.

While neither SI algorithm is capable of calculating a path with the same path optimality as the A* algorithm, they are still able to calculate each path at a greater speed than the A* algorithm. The ACO algorithm seemed well adjusted to be utilized in dynamic environments where a path must be continually updated or calculated. The PSO algorithm, while
possessing the desired trait of fast calculation times proved inconsistent and harder to predict making it more difficult to recommend in implementations where a consistent performance is required. Therefore the benefit of using the SI algorithms, ACO and PSO, for pathfinding in games is the reduced time taken to calculate a path.

6.2 Discussion

When applied to a dynamic version of the travelling salesperson, Angus and Hendtlass (2005) found that ACO was capable of finding the best path only 18 times out of 10,000 while finding the second best path 2,974 times. However despite the results achieved in terms of path length, the speed of the algorithm was determined to be remarkable. These results are similar to the results determined in this study. The ACO algorithm implemented for the pathfinding was able to achieve outstanding results in terms of speed, but was lacking in terms of path optimality.

In their paper on optimal parameters for the ACO algorithm, Gaertner and Clark (2005) obtained results that indicated that in the case of the travelling salesperson there is no optimal set of parameters for all instances. Due to time constraints this study uses static parameters for the ACO algorithm. Therefore the utilized parameters were chosen as the most promising combination based on results from a pilot study, see chapter 4.6. It is possible that different parameter combinations may have yielded different results. This can also be said for the PSO algorithm which determined parameter values in the same manner as the ACO algorithm, using the pilot study.

Another implementation of the ACO algorithm, by Mocholi, Jaen & Catala (2010), utilizing 32 ants was able to determine an initial path on a similar grid with an average time of 4148 milliseconds. Only three ants were used in this implementation opting to favour exploitation over exploration. This showcases how the different parameter setting affect the results achieved.

Related studies regarding the PSO algorithm by Carlisle & Dozier (2000) showcase that the PSO's inconsistency is affected by the rate at which the personal best and global best are updated. For this study the best values were reset each time a new path was calculated whereas their study achieved the best results when values were reset after 16 iterations. The fact that their study showcased better results when resetting after 16 iterations does not mean that better results would be achieved in this study using the same format, however neither does it exclude the possibility that a different method for resetting the values could have increase performance. Whereas this study based results on cost and path optimality theirs were based on the ability to simply complete the experiment within 4000 generations. It is difficult to compare the results of the PSO algorithm because despite being applied to dynamic environments the other did not focus on the same measurements (Liu, Wang & Yang, 2010; Blackwell, 2007).

As public documentation of utilization of the two SI algorithms for video games is scarce, it is difficult to determine what method of implementation is the most prevalent. Compared to A* which has a great deal of public documentation and research papers, the SI algorithms and their use within this field is lacking. While it is possible that some games may utilize these SI algorithms for pathfinding, public documentation of such uses is not available. However as both algorithms were modified in one manner or another, see chapter 4, it is safe to say that all the algorithms could have been implemented in a different manner and as such have
received different results. Therefore the results gathered in this study can only be used to
draw conclusions for the utilized implementation method. As such the implemented version
of ACO and PSO do not necessarily reflect the original algorithms by Dorigo (1992) and
Kennedy & Eberhart (1995) respectively.

6.2.1 Garbage Collection

Other factors that may also have affected the results is the built in garbage collection for the
C# programming language and Unity (Unity Technologies, 2017). As a game runs it uses
memory to store data. When that memory is no longer needed for storing data, for example if
a parameter is no longer accessible, then the memory storing that data is freed up so it can
be reused. The garbage collector identifies and de-allocates unused memory. Garbage
collection is run periodically meaning sometimes there may be more memory to clean than
other depending on the amount of unused memory. Despite requiring a great deal of CPU
time, garbage collection is automatic and invisible to the programmer. When the garbage
collector runs the additional overhead there may be a drop in frame rate and performance.

Algorithms that store more information have a greater risk of being affected by the garbage
collector, it is possible that the garbage collector had to free up more memory during their
data. The algorithms most likely affected by this is the A* algorithm and possibly the ACO algorithm due them storing information. While it is possible to code in a manner that minimizes the impact of the garbage collector, by using
memory more efficiently, none of the current implementations are implemented with that in
mind. Had they been then it is possible that their results may have been different from the
current results.

6.2.2 Real World Applications

When travelling it is common to use a GPS, global positioning system, in order to determine
a what roads to take. However travelling by GPS often only determines the shortest route in
order to travel from one point to another. For truck drivers and other workers there are often
other things to also consider when determining a path such as multiple destinations and
resources to be transported. The vehicle routing problem is a combinatorial optimization
problem where a fleet of vehicles, with limited capacity, must be routed in order to visit
customers at a minimum cost, similar to the routing decisions made by companies when
delivering supplies. There is a dynamic version of the vehicle routing problem where new
orders arrive throughout the day requiring the algorithm to adjust routes.

Montemanni, Gambardella, Rizzoli & Donati (2005) have utilized the ACO algorithm for
dynamic vehicle routing problem where the algorithm is capable of achieving good results in
artificial situations. Due to the qualities displayed by the ACO algorithm, it suggests that the
algorithm may function well if applied to realistic routing problems, thus being useful for
larger companies that must manage the delivery of their products. This could then be
implemented in a drivers GPS as to allow their paths to be updated without too much
difficulty. The current implementation of ACO was modified in order to function in a game
environment, likewise the algorithm would have to be modified to function real world
pathing.

If the algorithm would ever be implemented in such a manner then one would have to
consider how optimal the path is, the implementation used in this study is able to achieve a
path at such a fast pace at the cost of optimality. Higher optimality would most likely lead to
greater path calculation times. If a path is being recalculated for a truck driving on the road then it must be completed before the driver approaches a turn in order for the truck to be able to take the turn in a safe manner. Should calculation times become too great for the higher optimality then it may prove harmful for a human operating a vehicle following the algorithms’ instructions. Likewise if the optimality was not focused and the speed was instead favoured then the algorithm may lead a driver to take a longer path than required, increasing the time spent on that particular delivery. Such situations may add unnecessary stress and frustration for the driver.

The aim of this study was to research the benefits of the SI algorithms for pathfinding in games. With the results achieved, the current implementation, of the algorithms have the benefit of having fast calculation times. Despite their somewhat downgraded path optimality in comparison to A*, their speed is a favourable trait. This means that the algorithms may, if not already, be applied to games for pathfinding.

Based on the movement pattern of the ACO and PSO, see chapter 5.2.4, both algorithms display somewhat unique movement. As their movement can almost on its own be considered a unique behaviour it may cause less stress on the system to implement these algorithms for pathing instead of having a dedicated movement state aimed to achieve the same movement pattern. With the ACO’s somewhat delayed change in direction, due to pheromone update, it could presumable be used for a slower character that a player may be able to outwit by constantly moving. Whereas the PSO’s zigzag style movement may be applicable to an agent meant to dodge the player's attacks. Their unique behaviour may lead to increase a player's enjoyment of a game.

### 6.3 Future Work

If there had been more time available for this study then there are several part of the work that would have been completed differently in order to hopefully have obtained better results. As mentioned in chapter 6.2 all parameters are currently static throughout all experiment environments, if possible it would have been desirable to have tests run on each experiment environment in order to first determine the best parameter combination for that environment. This would possibly have improved the path optimality of the algorithms allowing them to achieve results more in line with the A* algorithm.

When implementing the PSO algorithm there are multiple variants that can be implemented (Rini, Shamsuddin & Yuhaniz, 2011). For this study a basic variant of the PSO algorithm, with slight modifications to suit pathfinding, was used. If possible other modifications could have been made to the PSO algorithm in order to determine what modifications led to greater results. This can also be expanded to include other SI algorithms in the study. Currently this study only focuses on two algorithms whereas there are multiple other algorithms that possess the potential to be utilized within this field.

Something that would have proven to be interesting would have been to test the algorithm on maps from more recent games. As both Warcraft 3 and Starcraft were released more than 15 years ago it is possible that game design may have altered with the introduction of new technology allowing for more complexity. The results from the new maps could then have been compared with the old. In such scenarios the algorithms could even be tested in larger environments. While the algorithms are still expected to perform to a certain standard on newer maps it would still have proven to be an interesting study.
This study purely bases results based on measurements obtained from the experiment. When playing a game the path optimality is not the most important aspect of the pathfinding algorithm and therefore algorithms are often modified to suite the game. Another manner in which this experiment could have been conducted is by allowing each pathfinding algorithm to run then having a test persons comment on the determined path. The survey could then have been altered to instead focus on what algorithm provides a player with the most rewarding player experience in comparison to the statistic approach taken.
References


Age of Empires 2 HD. (2013). SkyBox Labs.


