University of St Andrews

Postgraduate Thesis

Swarm Artificial Intelligence in Hive

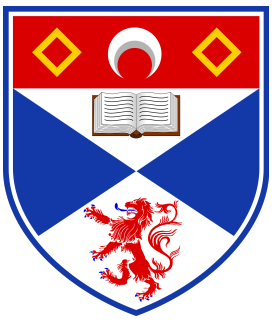
A thesis submitted in fulfilment of the requirements

for the degree of Master of Science in Artificial Intelligence

in the

School of Computer Science

August 2020



Word count: 14,519

UNIVERSITY OF ST ANDREWS

Abstract

School of Computer Science

Master of Science in Artificial Intelligence

Swarm Artificial Intelligence in Hive

by Connor Michael McGuile

Swarm AI is a subset of AI inspired by nature, notably birds flocking and ants foraging for food, which attempts to replicate the observed emergent intelligence of these collective processes. Swarm AI applied to strategy games has rarely been researched, and a game-playing agent using Swarm AI for the insect-based board game, *Hive*, has not yet been documented. This report summarises the research, design, and implementation of a Swarm AI for *Hive*, noting that Swarm AI is comparable in strength to a depth-limited Monte-Carlo Search Tree. Swarming as a strategy for game playing appears to be an ideal, collaborative form of attack and is likely to be suitable for games where coordinated movement of units is required providing a suitable defence mechanism is also incorporated.

Acknowledgements

I would first like to extend my sincere thanks to my dissertation supervisors, Dr. Stephen Linton and Dr. Michael Torpey, whom without I would likely be submitting a lesser dissertation. Your support and guidance throughout my research and writing has been excellent, and I thank you for enabling me to choose a topic which has seldom been researched, providing new and interesting exploration into the field of Swarm AI.

Importantly, I am eternally grateful for my best friend and girlfriend, Dorsa, who happily supported me day in and day out in both my undergraduate and postgraduate degrees.

Last, but certainly not least, I am indebted to my family whom have given their all to ensure I could focus on achieving my best.

Declaration of Authorship

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 14,519 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Date: 14th August 2020

Connor Michael McGuile

Contents

List of Figures

List of Tables

Abbreviations

AI	**A**rtificial **I**ntelligence
PSO	**P**article **S**warm **O**ptimisation
ACO	**A**nt **C**olony **O**ptimisation
RL	**R**einforcement **L**earning
MCTS	**M**onte **C**arlo **T**ree **S**earch
UI	**U**ser **I**nterface

Chapter 1

Introduction

1.1 Overview of Research

Artificial Intelligence (AI) has surged in research and development in recent years due the advancements in computing technology and algorithm design, and its ability solve problems in any mathematically represented domain. Consequently, AI has grown from initial attempts to emulate human behaviours like conversation (Weizenbaum et al., 1996; Colby, 1981) to highly-specialised applications such has tumour detection and classification (Abd-Ellah et al., 2018) and the production of autonomous, self-driving cars (Badue et al., 2019).

Swarm AI, the focus of this research in the game-playing domain, is a subset of AI inspired by biological processes in nature, notably fish in schools or birds flocking, which attempts to replicate the observed collective intelligence of these processes (Kennedy, 2006, p. 187). In flocking, this transcendent intelligence is emergent from simple individual instincts, and no one individual encompasses this intelligence alone. By programming mathematical representations of individuals with their own simple characteristics, Swarm AI hopes to generate similarly emergent behaviour in an attempt to optimise a solution to a problem in its domain. Swarm AI has seen tremendous success in fields such as robotics, with small drones communicating to manoeuvre around each other (Brambilla, 2013), and Human Swarming, where a

group of people answer questions and generalise well enough to find better answers than individuals acting alone (Ryan, 2018). However, Swarm AI has seen little application to games and has never been applied to the game of *Hive*.

Game playing as an optimisation problem has fascinated AI researchers since the 1950s. Armed with advancements in game theory, symbolic AI applications based largely on hash tables limited-depth search trees were developed in an attempt at solving the games of Checkers, (Samuel, 1959), and Chess, (Greenblatt et al., 1988; Condon and Thompson, 1982). More recently, in part due to the realisation that exhaustive exploration of tree nodes is intractable and inefficient, researchers have turned to machine learning with great success. In the past five years, researchers have accomplished building an AI capable of superhuman performance in the game of *Go*; a game thought to be the pinnacle of two-player strategy board games in terms of complexity (Silver et al., 2017).

It is reasonable to question why, then, if such AI exists, there should be research into Swarm AI for game playing, specifically in *Hive*. Aside from the connection between swarms in nature and the insects in *Hive*, it is important to investigate Swarm AI in games due to its rarity in the game playing domain and the gap in the knowledge of its applicability. The great success of other AIs in this domain have pushed aside any other methods which may have potential. It is unlikely that Swarm AI can compete with the best AI available, though it can provide insight into whether Swarm AI is a viable solution to use in *Hive* and other games, and if any new knowledge can be learned from the behaviour of the AI itself.

1.2 Research Methodology

To provide a platform for a Swarm AI to be researched and developed, a software product with a user interface (UI) for the game will be built in Python. Concurrently, research into existing AI systems for game-playing and swarming algorithms will be conducted. As it would be interesting to visualise swarming behaviour applied to the

movement of the pieces in the game, preference will be given to AI which can emulate this behaviour, such as bird flocking, bee colony, and ant colony algorithms.

Chapter 2 will outline current applications of these swarming algorithms and existing AI implementations for the game of *Hive*. Chapter 3 will discuss the design and implementation of the game, including the custom algorithms to provide the insect movements, the opponents the Swarm AI will play against, and the Swarm AI itself. Chapter 4 will analyse and evaluate the success of the AI. Finally, Chapter 5 will summarise the research and provide suggestions for future work.

Chapter 2

Background

2.1 Motivation

Researching a possible implementation of Swarm Intelligence in the game of *Hive* is motivated by the novelty of the application and the relationship between insect swarms and insects in *Hive*. The research goal is not to determine if an appropriate AI can be made for *Hive*, as conventional tree-based implementations already exist, rather the aim is to gather an understanding of whether biologically inspired Swarm AI can be applied to such a game and if any further knowledge on swarm behaviour can be extracted from this research.

2.2 Particle Swarm Optimisation

Particle Swarm Optimisation (PSO), introduced by Kennedy and Eberhart in 1995, is a numerical optimisation algorithm representative of birds flocking to produce emergent behaviour. The problem bird flocking in nature attempts to optimise is minimising the amount of energy used by the birds as a group to fly to their destination. When implemented in computing, the approach is to effectively

simulate this flocking behaviour, whilst demonstrating emergent complexity using simple rules.

Each bird is defined by a position vector and velocity vector in D-dimensional space. Each bird has complete awareness of all events occurring in its local vicinity, set by some radius around the bird, and has zero awareness of any events outside of this vicinity. Each particle determines its own *fitness* in the search space based on a fitness function and, via swarm characteristics of separation, alignment, cohesion, and convergence towards to the best particle in its vicinity, will steer effectively as a swarm.

This type of PSO is known as *local best* or *lbest* PSO as each bird, herein referred to as particle, is steered towards the best particle in its vicinity. The original approach to PSO was to use a *global best* or *gbest* approach, where each particle would be aware of the best particle in the entire swarm and steer towards it. This approach is typically only works well in unimodal optimisation problems due to its fast convergence to global minima. A survey conducted in 2013 suggested that "if the objective is to find the most optimal algorithm for a specific optimization problem, then the neighborhood topology used has to be included as one of the parameters tuned" (Engelbrecht, 2013).

Separation

The concept of separation amongst particles is to ensure that the particles do not collide or overlap with one another while traversing the search space. To do so, at each time step, the particles will steer away from the nearest particles.

Alignment

Alignment is concerned with ensuring the particles in a neighbourhood steer towards the average heading of their neighbours. The purpose is to create the swarming motion towards goal displayed by flocks of birds.

Cohesion

Likely the most important characteristic of PSO, cohesion ensures that the particles do not drift away from their neighbours, and again creates the swarming motion desired.

2.2.1 Global PSO Formula

As mentioned earlier, the implementation of PSO requires that particles are modelled and have respective position and velocity vectors. The updating of these vectors follow a simple but effective formula which utilises hyperparameters. The hyperparameters are often the source of refinement when applying PSO to a specific domain.

To update the position of a particle, a newly found velocity is added to the current position. Therefore, the velocity vector is represented by $[\Delta row, \Delta col]$. The formula for determining this new velocity for global, or local vicinity, PSO is given by:

$$v_{ij}(t+1) = w \cdot v_{ij}(t) + c_1 \cdot r_1 \cdot (y_{ij}(t) - x_{ij}(t)) + c_2 \cdot r_2 \cdot (z_j(t) - x_{ij}(t)) \tag{2.1}$$

where $v_{ij}(t)$ is the current velocity of particle i, $y_{ih}(t)$ is particle i's best known position so far, $x_{ij}(t)$ is particle i's current position, and $z_{ij}(t)$ is the global, or local vicinity, best known position; all in dimension $j = 1, ..., D$ at time step t.

This formula, equation 2.1, includes an inertia component, a social component, and a cognitive component. The inertia factor, w, is a fine-tuned constant which allows smoother control of velocities of particles and results in better performance overall (Shi and Eberhart, 1998). The cognitive term is often referred to as the *experiential* term, meaning it reflects the knowledge it has over previously seen positions, and it is proportional to the distance of its current position to the best known position it has been. The social term, so called due to its dependence on the global best or vicinity best position known, is proportional to the distance between that position and its current position.

Acceleration constants, c_1 and c_2 are fine-tuned hyperparameters which affect the trade-off between exploration and exploitation of the algorithm, where exploration is concerned with visiting new, unseen space and exploitation references converging towards the best known position. The values, r_1 and r_2, are random numbers in the range [0, 1], sampled from a uniform distribution, which scale the acceleration parameters to introduce a stochastic element to the algorithm.

2.2.2 Summary

As can be observed from PSO's simplicity, the algorithm is straightforward to implement among other merits including its effectiveness in global search, insensitivity to scaling of design variables, and fast, but sometimes premature, convergence. As a result, has been successfully applied to many domains such as machine learning, image processing, power systems, and networking.

Conversely, PSO comes with potential limitations in that typically a large population of particles is required for sufficient exploration and exploitation of the search space. In *Hive*, the maximum number of particles possible is 11. Piotrowski et al. (2020) conducted experiments on 60 artificial benchmarks and 22 real-world applications to analyse the effect of the number of particles used for multiple PSO variants. Of eight PSO variants tested, the minimum number of particles required was found to be 20-50 in two of the variants, whilst typically 70-500 for the remaining variants. However, there are methods to overcome these limitations; for instance, setting a balance of exploration and exploitation throughout the process depending on the evaluation of positions.

2.3 Ant Colony Optimisation

Ant Colony Optimisation (ACO), similar to PSO, is inspired by nature, specifically the way in which ants work together to source food and materials and communicate with others in the colony (Dorigo, 1992).

ACO consists of four main parts: the ant, pheromones left by ants, daemon actions, and decentralised control. An ant is a representation of a particle-like agent in the system which traverses the search space, emulating exploitation and exploration that ants perform. As ants travel, they leave a pheromone trail signifying the route they have taken which evaporates with respect to time. Other ants will follow the strongest pheromone trails and eventually the weakest trails are eliminated. The same is true for simulated ants - a mechanism of pheromone emission is implemented as means of communicating with the colony. Daemon actions are optional centralised actions which are executed by some daemon with global knowledge of the problem and search space. They are used when an action cannot be conducted by a single ant and often increase pheromones of certain trails to direct the colony. Finally, decentralised control is an important feature to ensure the algorithms success by providing no single ant more influence than another. In the event an ant is trapped in a loop, or lost completely, the colony is not adversely affected.

ACO is typically useful for solving graph-like problems where cyclical paths are required, such as the travelling salesman problem, as the pheromone placement leads to emergent behaviour of determining the shortest path through the graph.

ACO may not be entirely appropriate for a *Hive* AI due to the unlikelihood of cycles being an effective winning strategy for multiple objectives. Of interest, however, is the means of communication ACO provides with pheromone trails. The ants, of course, do not directly communicate with each other, but indirectly suggest to whomever discovers the trail that the path is optimal. If a means of indirect communication could be combined with particle swarm behaviour, an appropriate swarm-based *Hive* AI may emerge.

2.4 Particle Swarm Optimisation in Games

Particle Swarm Optimisation (PSO) has been shown to be useful as a technique to enable agents in a game to play effectively as a swarm. Liu and Wang (2008)

demonstrated that all aspects of PSO can be mapped to strategy-based games such that each particle is an agent, each action is contained in a list of actions, or strategy, representing the search space, the fitness function is a risk-reward payoff function based on the agents' goal(s), and the global best particle represents the most stable evolutionary strategy. Therefore, a swarm's best action can be said to be the one which results in the maximum payoff.

As PSO is a swarm convergence algorithm, it may useful as a tool for *Hive* for goal(s) where a cell, or cluster of cells, are the target for the swarm to reach. The fitness function for each particle's global reference position may need to be different according to each agent's movement abilities, grouped perhaps by the type of the agent, and might be affected by external factors referencing the opposing agent positions, as the agent may need to be offensive or defensive. Each successive move would require PSO to be repeated should the goal change; for instance, if the opponent queen moves and the goal is to surround the queen.

Duro and Valente de Oliveira (2008) proved PSO's worth in strategy-based games by utilising it in the perhaps most famous board game in the world: *Chess*. However, their implementation of PSO differs from Liu and Wang's solution in that PSO is used as a tool to determine appropriate weights for a heuristic evaluation (fitness) function for successive game board states provided by a minimax algorithm with alpha-beta pruning, rather than a global tool for all agent actions.

The heuristic evaluation function consists of multiple evaluation functions for a game state in *Chess*, notably including Material Balance, where the effective worth of each piece on the board is summed; King Safety, consisting of various factors prudent to the game's continuation; and Attack Lines, giving a value for the number of positions each piece is able to control. A final heuristic evaluation function combines a weighted sum of these smaller functions, and it is these weights which PSO is used to estimate.

The solution proposed by Duro and Valente de Oliveira is an effective one for *Chess* as the smaller function values are easily established. In the case of Material Balance, there is sufficient documentation on the worth of each *Chess* piece. However, for

Hive there is no such worth assigned, rather a ranking without scale, and therefore it cannot be assessed whether an Ant is worth twice as much as a Beetle, for example. Furthermore, pieces in *Hive* are not removed from the board during play, meaning this metric could only be used during early-stage play. Nevertheless, the other metrics proposed do indeed suit *Hive*, and some other fitness functions could be considered.

2.5 Neural Networks with Particle Swarm Optimisation

Messerschmidt and Engelbrecht (2004) demonstrated the use of PSO as means to define appropriate weights of a neural network. Game states of *TicTacToe* were represented by a decision tree, and the neural network was designed to model a heuristic function to evaluate these states.

The same method was applied to the African game of Bao two years later (Conradie and Engelbrecht, 2006). Instead of solving for weight factors applied to heuristic functions, PSO was used to house a swarm of particles consisting of feed-forward neural networks. Each neural network consists of ten inputs, representing the nine cells on a board and an input indicating the player, and a sigmoidal output in the range zero to one indicating the strength of the belief in that game state.

Each particle in the swarm competes against a small sample of other particles, with the worst 15 particles removed from the swarm and the best 15 particles selected for breeding of new particles. The process of breeding refers to selecting 2 of the strongest particles, combining features of each particle into a new particle, typically at random, to produce a child particle, and, with some probability, mutating the features of that child to ensure diversity. This process is adapted from the biologically-inspired "genetic algorithm".

Limitations of the utilising a neural network to establish an evaluation function for *Hive* are that the game board is not constrained as agents can continually move in

any direction, and thus inputs to the neural network are either unknown or must be significantly sized to accommodate any position of agents with the majority of inputs zeroed. Additionally, this leads to increased difficulty in training the neural network as exact game state replicas may exist anywhere on the board in various rotations and translations. A partial solution could be to assign a reference piece, such as the Queen Bee, and have a spatial system built around that point as inputs to the neural network. Due to feed-forward neural networks inability to be spatially invariant, if this solution was not successful, the number of game states required to be learned would increase exponentially. This gives rise to the possibility of using Convolutional Neural Networks (CNN) to analyse game states as CNNs are spatially invariant. As consideration for future work, using Swarm AI in CNNs would likely adopt the approach taken by Messerschmidt and Engelbrecht (2004) of using PSO to optimise the weights of the network.

2.6 AI Implementations for Hive

AI solutions to *Hive* are limited due to the modernity of the game, although no published solution exists utilising Swarm Intelligence. Solutions found include Monte Carlo Tree Search (MCTS) (Konz, n.d.) and Reinforcement Learning (RL) (Blixt and Ye, 2013); both methods are popular approaches to game playing AI.

One reason why these implementations are of interest is they provide a naive method of benchmarking a swarm-based AI implementation alongside our own evaluation. Not only can they be compared by the win-draw-loss statistics against random-guess AI but also on the decision-making speed, memory requirements, and other performance metrics.

2.6.1 Reinforcement Learning

Blixt and Ye (2013) were not able to create a viable RL solution to *Hive* due in part to RL requiring that the same actionable states are seen multiple times to establish

an effective reward or punishment depending on the action taken. The game state space is ultimately too large and the time taken to process each move was not optimal. Blixt and Ye note that of 318,000 game states seen, only 0.49% of those had been seen before. Essentially, this meant that two RL AIs competing against each other were not given sufficient opportunity to learn from their actions and thus were as good as random-guessing.

This aspect of the enormity of number of game states, and learning which actions to take for each, is relevant to this research as Swarm AI does not require any learning capabilities. Swarm particle parameters are updated in real-time based upon evaluation functions of their position. Therefore, this extreme limitation to developing an AI for *Hive* would not need to be considered.

2.6.2 Monte Carlo Tree Search

As mentioned earlier, Konz (n.d.) implemented an AI solution to *Hive* by using MCTS. Their reasoning for this approach is justified by noting that other decision tree algorithms, such as Minimax, and machine learning approaches, such as RL, require a positional evaluation of game states in order to determine appropriate actions. MCTS does not require any such function. MCTS, made famous by researchers at Google for beating professionals at the game of *Go*, utilises randomness to solve deterministic problems by "playing out" the remainder of games with random selections of nodes, or successive game states (Silver et al., 2016). The result of each playout is used to order the nodes such that those which led to the most victories are higher in the tree. When faced with a turn to move, the same number of playouts are applied and the playout with the most victories is chosen as the action.

Konz's implementation of MCTS was more successful than random-guessing, winning 81.1% of games, but "compared to a human player, quite poor". Nevertheless, the interest in this research lies as inspiration for a possible candidate opponent for a Swarm AI to compete and compare against. If MCTS can be implemented as a

weak AI opponent, it would remove the necessity to define a heuristic function for an opponent, and it may also validate the heuristic function defined for the Swarm AI.

2.7 Centralised vs Decentralised Swarm AI

To take an advantage of emergent behaviour of swarms, Cunha (2015) proposes a solution to turn-based strategy games utilising a *decentralised*, non-cyclical approach with Swarm AI concepts. Cunha states "there are generally two ways of implementing an AI-Player for a Strategy Game - using a *centralized* or a *decentralized* approach."

A centralised approach can be modelled by playable agents having some form of intelligence as individuals, creating a multi-agent system. Centralised systems often have one of the agents playing the role of a commander of all other agents, in that it has the final decision on moves agents can perform and often has more knowledge than any of the other agents. This commander would be able to instruct other AI agents to perform actions via its own "intentions" and, because of this, all other agents must have a form of lower intelligence so that their intentions can be overruled.

Cunha suggests this form of AI is most often used by game developers as the commander can be modelled based on human intelligence whereas the other agents can have equal intelligence for both human and AI players, and this can create a game with varying levels of difficulty simply by altering the commander's intelligence.

On the other hand, a decentralised system allows each agent in the game to have their own complex thought processes and convey their own intentions to each other, rather than being directly controlled. A agent is able to continue on its own path, determined by its intentions based on its local domain knowledge, or follow that of another agent. This freedom to act alone is what Cunha suggests to be a potential downfall of a decentralised approach for games in which coordinated moves are required, such as surrounding an opponent with multiple agents. As each agent can merely decide its own fate, it can choose to aide its ally, or follow its own desires.

As a possible rectification, Cunha hypothesises agents to have levels of selfishness programmed and to have a priority level for any intentions. Additionally, agents may have sub-goals adjacent to their overall goal and these sub-goals can differ from agent to agent. For example, a group of agents may act as defenders of their team while others may act as attackers, all the while the main objective of winning, by surrounding the queen in the case of *Hive*, is still at the forefront of all agents' intentions. Finally, Cunha suggests that a mixture of centralised and decentralised AI could be appropriate depending on the game.

The design of Cunha's algorithm is a two-phase system, shown in Figure 2.1. The first phase is the "selfish phase" in which every agent analyses its environment and determines its next action based on the maximum reward it believes it will attain across all actions. This reward could be a heuristic made of many factors not exclusive to the primary goal of the swarm.

The second phase is called the "negotiation phase" in which every agent communicates its predetermined action, or intention, during the selfish phase to each other so that they can be evaluated. The purpose of this phase is so that all agents can then re-evaluate their own prior decision compared against what may be a better proposal by other agents and once again decide an action to take. Cunha states that in order for the algorithm loop to terminate, all agents must have reconsidered their intentions, settled on a final decision, and, if another agents intentions are followed, that agent must still have those intentions. In the case of turn-based strategy games like *Hive*, this would require all but one agents to yield to another to make its turn.

Not shown in the Figure 2.1, Cunha incorporates a theory introduced by Matzinger (2002) called *Danger Theory* into their heuristic. Conceptually, Danger Theory is the notion that the human immune system triggers a biological response to danger, not solely on the fact that it detects foreign bodies. This idea can be applied to strategy game AI as the swarm may detect danger when multiple enemy agents are approaching but perhaps be undeterred by a single enemy agent. Following on from this, Cunha implemented a *help* intention to signal to the swarm that the agent was fearful of its position. All of these heuristic values are scaled by severity.

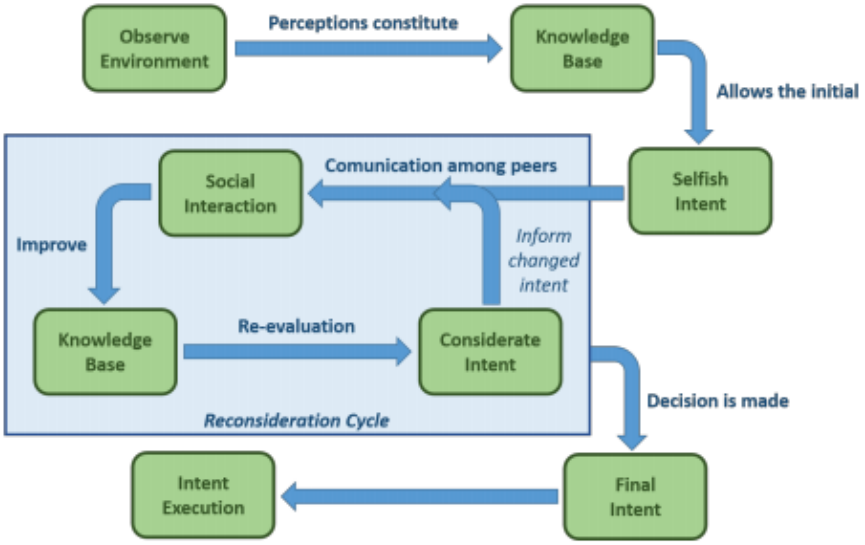


FIGURE 2.1: Conceptual design of two-phase algorithm, Cunha (2015)

2.8 Multi-Swarm AI

A singular swarm with particles declaring intentions and deciding an appropriate action, or particles which update their trajectory based on information gleaned from their neighbours, may both be effective forms of achieving a primary goal or optimising a function, though an alternative of multi-swarm AI should also be considered. Multi-swarm AI, as expected, can have two or more swarms with the intention of solving unimodal or multimodal objectives.

Niu et al. (2007) introduced a multi-swarm variant modelled upon a manager-worker ideology in that there is one central managing swarm and several worker swarms. The worker swarms act independently using a form of PSO whilst the managing swarm evolves based on the knowledge shared by the worker swarms. In many ways, this is reflected in nature, and may be applicable to *Hive* as the queen bee is the essential piece to protect. However, the vital difference between the *Hive* problem and the optimisation problem Niu et al. were solving is that *Hive* is a multi-objective problem. Where multiple swarms may attempt to find an optimal solution to a problem where one or more solutions exist, a multi-objective problem means that all objectives must be solved simultaneously, or at least in order of priority.

A player in *Hive* may have objectives of attacking the opponent's queen, pinning or covering units, and/or defending their own queen. Each of these objectives may be optimised separately, however an ideal solution for all objectives is seldom found due to the likelihood that the objectives conflict with each other, especially with the small number of playable units. Therefore, a selection of possible non-dominated solutions to the multi-objective optimisation problem need to be determined such that no objective solution can be improved without worsening another objective solution, otherwise known as Pareto optimality.

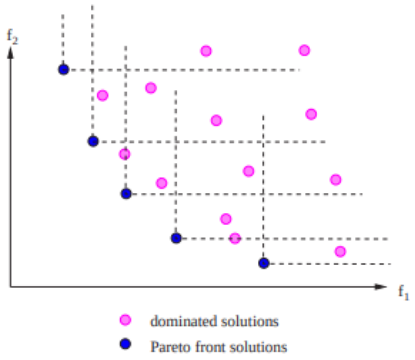


FIGURE 2.2: The Pareto front: a set of non-dominated solutions in two-objective space (Reyes and Coello, 2006)

Early studies proved successful for multi-objective optimisation using genetic algorithms, (Fonseca and Fleming, 1995; Zitzler, 1999), and later multi-objective PSO was explored, (Moore and Chapman, 1999; Hu and Eberhart, 2002). In a survey on the state-of-the-art solutions to multi-objective optimisation with PSO in the mid-2000s, the authors outlined various approaches including "aggregation", "lexicographical ordering", "sub-populations", and "hybrid approaches" (Reyes-Sierra and Coello, 2006).

2.8.1 Aggregation

This category considers methods that aggregate all objectives in one, representing the multi-objective problem as a single-objective problem. Parsopoulos and Vrahatis (2002) demonstrate three types of linear aggregation functions: one consisting of a statically weighted sum of evaluation functions; another, a dynamically weighted sum

of evaluation functions, where weights change throughout the run; and a final method called "bang bang" aggregation where weights are static until sharply adjusted at some point during the run.

Baumgartner et al. (2004) also use linear aggregation but instead applied to the objectives of sub-swarms which have their own weights and a leader to follow, decided by the particle with the best objective function value.

2.8.2 Lexicographical Ordering

The process of lexicographical ordering is to rank objective functions in order of priority. Once ranked, the most important functions are optimised for first and once completed, or in a state where its importance has fallen below the threshold of the second function, the next functions are optimised. Coello (1999) states this method is likely only appropriate when up to three objective functions exist due to the likelihood that one or more objectives and their priority may cause the swarm to converge to one particular part of the Pareto front (the set of non-dominated solutions).

2.8.3 Sub-Populations

Working with sub-populations, or sub-swarms in this manner, infers each have a single objective to optimise for and they communicate their progress to other sub-populations, perhaps even combining swarms if evaluation of strengths and weaknesses of doing so deems it justifiable (Chow and Tsui, 2004).

Chapter 3

Design and Implementation

3.1 Proposal

This chapter begins with a discussion on the infrastructure and algorithms built to accommodate *Hive* game play. The design of these algorithms is not the focus of research, though it should be stated that sufficient optimisation has occurred to ensure that the processing time for the AI is minimised. Section 3.4 outlines the reasoning and implementation of opponent AI for the product of this research, a Swarm AI for *Hive*, to play against. Finally, in Section 3.5, the Swarm AI will be designed and implemented such that it integrates Particle Swarm Optimisation to manoeuvre the insects in the game.

In Chapter 2, it was noted that there are alternatives that incorporate swarming behaviour, such as using PSO algorithms as means to provide weights for a neural network. Whilst these alternatives are certainly reasonable considerations, the movement of the pieces themselves may not simulate swarming behaviour as a result. Not only is PSO a viable method of mobilising a player's pieces due to a direct mapping of particles to insects, but it may also provide an interesting visualisation of the effectiveness of swarming behaviour in two-player games. If simulated swarming behaviour using PSO in *Hive* can be considered strong, it is

plausible that other strategy games may adopt PSO or other swarming algorithms to execute movements of playable units.

As PSO has no means of communication among particles, and *Hive* has a minimum of two objectives of protecting and attacking respective Queen Bees, it is proposed that communication measures will be implemented and will enable a decentralised system. The development of the Swarm AI will be iterative in that the success of all design decisions will be considered with respect to previous iterations, and positive factors will be inherited in newer generations.

3.2 Hive Rules and Gameplay

Hive, published in 2001, is a two-player board game in which there is no board. The hexagonal pieces themselves connect together to form the playing surface of a single, unbroken hive. The objective of the game is to capture the opponent's Queen Bee by surrounding it on all edges by any combination of either player's pieces, whilst avoiding having your own Queen Bee surrounded.

3.2.1 Gameplay

Hive is an abstract strategy game comparable to *Chess* in that different pieces have different abilities of movement. In the original game, each player begins with 11 insect-based pieces in their rack (or hand) ready to be played, where the composition of these pieces are as follows:

- 1 Queen-Bee (yellow)
- 2 Spiders (brown)
- 2 Beetles (purple)
- 3 Grasshoppers (green)

- 3 Ants (blue)

Any piece can be played from the rack at any time, providing it is able and it is the player's turn. The pieces on each player's rack are visible to the other player. As *Hive* is a perfect-information game, it serves no advantage to conceal the pieces available in a player's rack as the contents can easily be deduced from the hive itself.

On each turn, a player may move a piece from the rack onto the board adjacent to one of their own pieces, provided that the piece does not touch an opponent piece. The exception to this rule is when the game begins, and the player to act first has placed their piece, as the second player must play their piece adjacent to their opponents piece. The player may be allowed to move a board piece instead if they have already placed their Queen Bee. If a player has not placed their Queen Bee by the third move, they must play it on their fourth move.

3.2.2 Rules

All moves the player makes must conform to two rules: the One Hive Rule and the Freedom to Move Rule. The One Hive Rule ensures that no piece can be moved or placed such that it is separated from the rest of the hive. This includes when the piece is in transit. The Freedom to Move Rule states that a piece can only move from or into a position should it be able to slide into it without lifting the piece. Due to the move mechanics, the exceptions to this rule are Grasshopper moves and the Beetle stacking and dropping moves which will be explained later.

3.2.3 Movement

As there is no board in *Hive*, the movement of pieces is managed by the imaginary infinite plane of tessellated, flat-top hexagons. A movement of one space is equivalent to moving to a hexagon connected to any face of the current hexagon.

The movement of each insect piece is as follows:

- Queen Bee - extremely limited in movement and can only move in any direction one space, providing the space is empty.
- Spider - able to move three spaces and only three spaces without backtracking.
- Beetle - able to move one space in any direction, but can also climb on top of any piece creating a stack and trapping the piece underneath, and can drop down or walk on top of the hive at will.
- Grasshopper - has the ability to move an infinite distance in a straight line across the faces of hexagons by jumping over connected pieces; however, it must stop jumping at the first empty space.
- Ant - able to move around the hive any number of spaces desired.

3.3 Infrastructure

The game logic is written entirely in Python, utilising the PyGame framework for the UI. The game is built to be as flexible as possible with regards to the types of players implemented into the game, the UI itself, and the capabilities for testing and evaluation.

As Figure 3.1 shows, the game is initialised with the racks placed face-up at the top and bottom of the screen and the playing surface empty. The implementation with a UI allows for manual control of a play-by-play game such that each move is executed by the user's control. This implementation also allows for a full game to be played without a human player - for example, if MCTS AI plays Swarm AI - to log results faster than the UI implementation.

Figures 3.2 and 3.3 illustrates the help provided when a player manually selects a piece to move. Many validation algorithms occur to ensure that the available moves shown to the player are legal.

The game board itself is represented by a 2-dimensional matrix and the shifting of hexagons up or down for the UI is handled by the code. The coordinate system

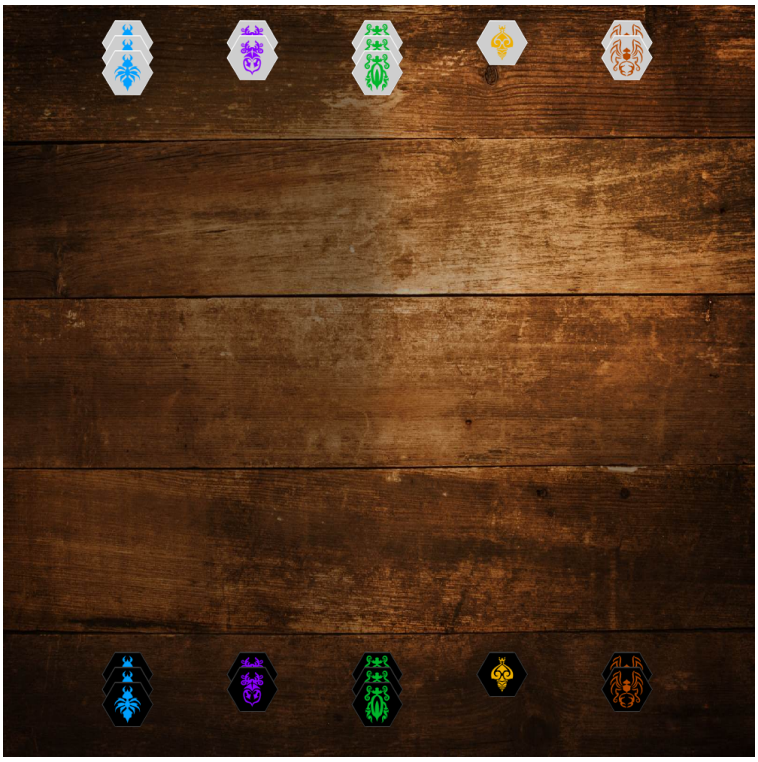


FIGURE 3.1: Hive initial state

implemented is represented as (y, x), written as (row, col) for readability, rather than the Cartesian system (x, y) due to the representation of the board as a 2D-matrix. If Cartesian (x, y) coordinates are given, where x is the column and y is the row, mapping of these coordinates to a matrix would require swapping to access the correct element. For example, Cartesian coordinates $(1, 0)$ would represent $board[0][1]$, where *board* is the matrix. This system is used throughout for all algorithms.

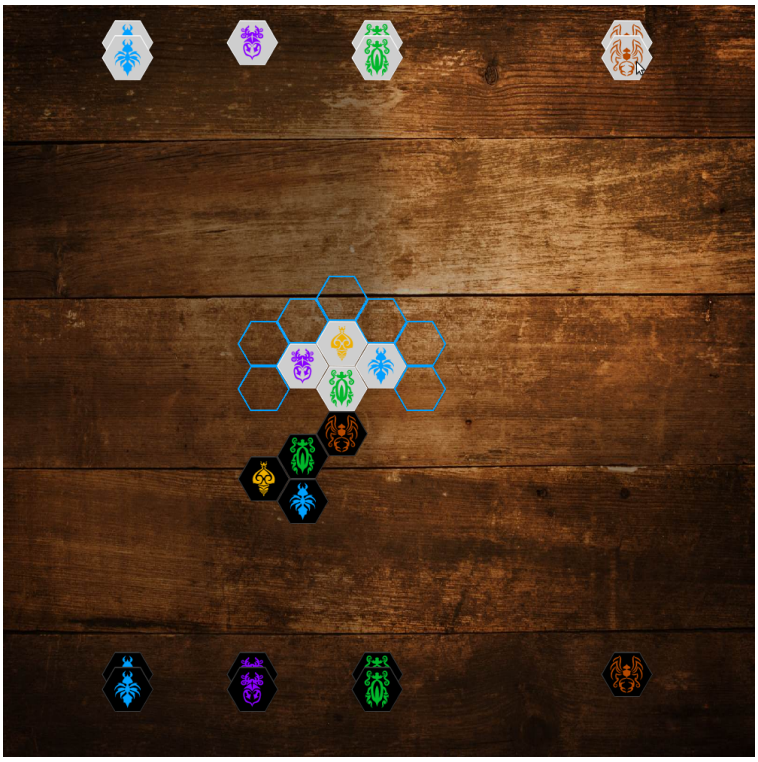


FIGURE 3.2: Selecting a piece from the rack, the UI provides indicators on possible moves.

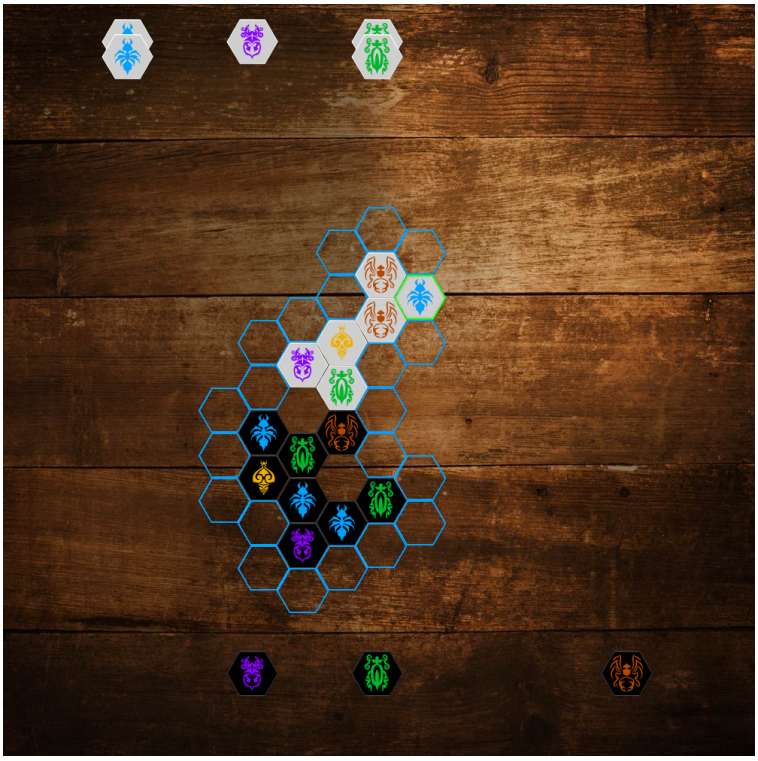


FIGURE 3.3: UI indicators when selecting a piece from the board. Also shown is the prevention of breaking the Freedom to Move rule.

3.3.1 Insect Move Algorithms

Algorithm 1: Get Possible Moves for Bee

input : Game State

output: Set of possible moves

```
1  possible_moves ← ∅;
2  (rf, cf) ← pos_selected;
3  neighbours, can_move ← GetNeighbours(rf, cf, state);
4  if can_move then
5      if WontBreakHive(state) then
6          foreach (r, c) in neighbours do
7              if piece_at_rc = Blank then
8                  if BreaksFreedomToMove(r, c, rf, cf, state) then
9                      continue;
10                 end
11                 neighbours_of_neighbours ← GetNeighbours(r, c, state);
12                 foreach (r1, c1) in neighbours_of_neighbours do
13                     if piece_at_r1c1 ≠ Blank and
14                     (r1, c1) ≠ (rf, cf) and
15                     (r1, c1) ∉ possible_moves and
16                     (r1, c1) ∈ neighbours then
17                         AppendToPossibleMoves((r, c));
18                     end
19                 end
20             end
21         end
22     end
23 end
24 return possible_moves
```

Algorithm 2: Get Possible Moves for Spider

input : Game State
output: Set of possible moves

```
1  possible_moves ← ∅;
2  (rf, cf) ← pos_selected;
3  foreach (r1, c1) in GetPossibleBeeMoves(state) do
4      pos_selected ← (r1, c1);
5      foreach (r2, c2) in GetPossibleBeeMoves(state) do
6          if (r2, c2) ≠ (rf, cf) then
7              pos_selected ← (r2, c2);
8              foreach (r3, c3) in GetPossibleBeeMoves(state) do
9                  if (r3, c3) ≠ (r1, c1) and (r3, c3) ≠ (rf, cf) then
10                     AppendToPossibleMoves((r3, c3));
11                 end
12             end
13         end
14     end
15 end
16 return possible_moves
```

Algorithm 3: Get Possible Moves for Beetle

```
input  : Game State
output: Set of possible moves
possible_moves ← ∅;
(rf, cf) ← pos_selected;
neighbours, _ ← GetNeighbours(rf, cf, state);
if piece_at_rfcf = Stack then
    return neighbours;
end
if WontBreakHive(state) then
    foreach (r, c) in neighbours do
        if piece_at_rc = Blank then
            if BreaksFreedomToMove(r, c, rf, cf, state) then
                continue;
            end
            neighbours_of_neighbours ← GetNeighbours(r, c, state);
            foreach (r1, c1) in neighbours_of_neighbours do
                if piece_at_r1c1 ≠ Blank and
                (r1, c1) ≠ (rf, cf) and
                (r1, c1) ∈ neighbours then
                    AppendToPossibleMoves((r, c));
                end
            end
        else
            AppendToPossibleMoves((r, c));
        end
    end
end
return possible_moves
```

Algorithm 4: Get Possible Moves for Grasshopper

```
input  : Game State
output: Set of possible moves
possible_moves ← ∅;
(rf, cf) ← pos_selected;
(n, s, ne, se, sw, nw) = GetHexasInStraightLine((rf, cf), WIDTH,
 HEIGHT);
if WontBreakHive(state) then
    foreach direction in (n, s, ne, se, sw, nw) do
        found_bug_to_jump ← false;
        foreach (r, c in direction do
            if 0 <= r < HEIGHT AND 0 <= C < WIDTH then
                if found_bug_to_jump = false then
                    if piece_at_rc = Blank then
                        break;
                    else
                        found_bug_to_jump ← true;
                        continue;
                    end
                end
                neighbours_of_neighbours ← GetNeighbours(r, c, state);
                if piece_at_rc = Blank and
                 NotAllNeighboursAreBlank(neighbours_of_neighbours,
                 rf, cf, state) then
                    AppendToPossibleMoves((r, c));
                    break;
                end
            end
        end
    end
end
return possible_moves
```

Algorithm 5: Get Possible Moves for Ant

```
input  : Game State
output: Set of possible moves
1 if WontBreakHive(state) = false then
2 |   return ∅;
3 end
4 possible_moves ← ∅;
5 (rf, cf) ← pos_selected;
6 moves = GetPossibleBeeMoves(state);
7 RecursivelyExecuteBee(moves, possible_moves);
8 RemoveFromPossibleMoves((rf, cf));
9 return possible_moves
```

Algorithm 6: Recursively Execute Bee Move

```
input  : Moves to explore, Known possible moves
output: Set of possible moves
1 foreach (r, c) in moves do
2 |   if (r, c) ∉ possible_moves then
3 |   |   AppendToPossibleMoves((r, c));
4 |   |   pos_selected ← (r, c);
5 |   |   moves ← GetPossibleBeeMoves(state);
6 |   |   RecursivelyExecuteBee(moves)
7 |   end
8 end
9 return possible_moves
```

3.3.2 Rule Algorithms

The One-Hive Rule is implemented by iterating over the entire grid of hexagons and executing a recursive depth-first search from the first hexagon that contains an insect to determine if the hive is in two pieces. The search considers the neighbours of the insect, and repeatedly executes depth-first search for each neighbour. If the depth-first search breaks, resorting back to the original iteration over the grid, and starts again, the hive must be considered broken. Therefore, to determine if an insect's move is legal, the move is temporarily made and the One-Hive Rule algorithm is then called upon to determine if only one hive still exists and, if so, the move is added to the set of possible moves.

whilst the insect is in transit, the Hive should not be allowed to be broken. This logic is handled by the insect movement algorithms as each movement step will always result in at least one adjacent insect.

A wrapper algorithm is also implemented to prevent needless recursive depth-first search over the entire set of hexagons and thus optimise performance. This algorithm first checks if the piece being moved is a Beetle stacked on top of another piece, as this would never result in a broken hive when moved. The second check is to determine if there is only one insect neighbour. In this case, the movement of the selected piece would not break the hive as it forms no connection to the rest of the hive; only the position it moves to may be considered a second hive if not connected to the first hive. However, this cannot occur due to the implementation of the insects' movements. Figure 3.4 shows these two examples.

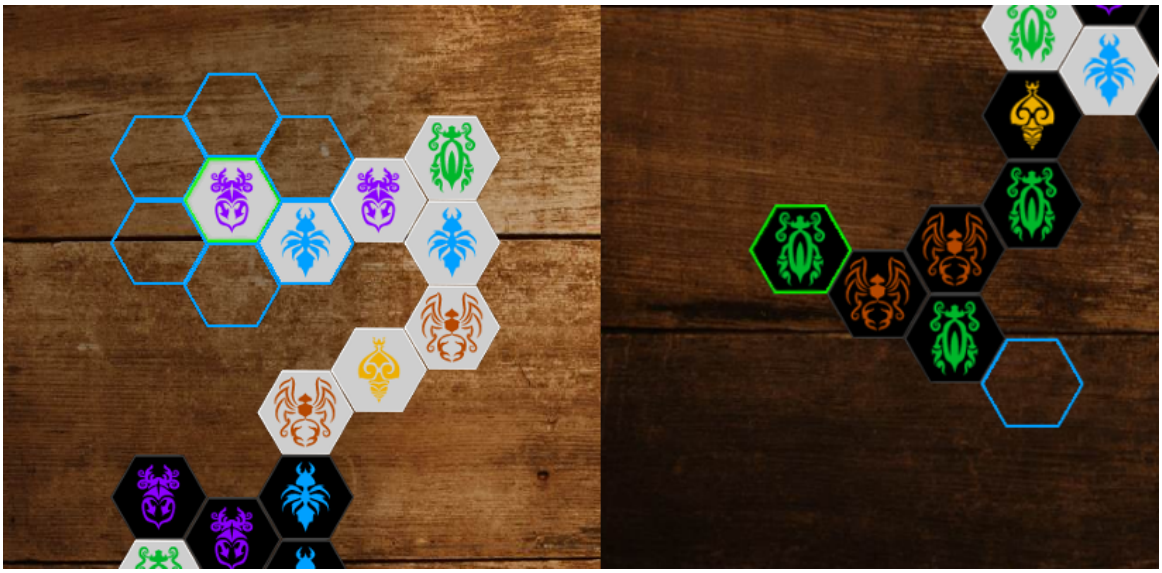


FIGURE 3.4: Examples of optimisation situations to prevent exhaustive depth-first search. Left: Beetle stacked - can move any direction. Right: Grasshopper hanging - will never break Hive with valid move.

The Freedom to Move Rule considers the position an insect is moving to, the position it is moving from, and the state of the game with regards to piece placement. The algorithm itself is not complex. The direction that the piece is attempting to move to is first calculated, followed by a deduction of whether the two possible positions which can block this movement hold placed insects. Figure 3.3 shows the possible outcomes of blocked and non-blocked desired moves.

3.4 Opponent AI

For the purposes of validating the proposed solution to building a swarm AI in *Hive*, there must be an opponent for the AI to play against. The solution will be compared to a "dumb" opponent such that the opponent randomly chooses its moves from a list of all possible moves. Next, and most importantly, the solution will be compared against an opponent utilising a different form of AI. The decision for choosing to build these opponents instead of playing against the author or other human players is twofold. First, building automated opponents allows for multiple games to be played until completion, building up an empirical, reproducible study on the effectiveness of the solution. Second, deciding against using human players removes any bias that may occur during play.

The two opponent AIs utilise random move selection and depth-limited Monte Carlo Tree Search, respectively; the details of which will be discussed here.

3.4.1 Random Move Selection

The first opponent will be entirely based upon random move selection, whilst all moves adhere to the rules of the game. A set of all generated moves, both from the player's rack and from the hive, will be chosen from.

As random move generation does not look ahead further than one move, the speed to make a decision is notably fast; much quicker than a human action. This will enable testing of the swarm AI against this opponent to be thorough.

3.4.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm notably utilised in strategy-based games with intractable numbers of possible successive game states, such as *Go*. As MCTS is a tree-based search algorithm, the typical requirement for application of this method is that there is perfect information of the current state of

the game. The position of all player units must be known at all times in order for a tree of successive moves to be built. The premise of MCTS is to randomly explore various paths in the tree, determine a reward value at a terminal node, and back-propagate these rewards up the tree. The summation of the rewards leading up to the root node of the tree indicate the most optimal move to take at that game state.

3.4.2.1 Alternative Opponent AI

As MCTS needs to be executed numerous times to converge to an optimal solution, one may reason that Minimax is a faster alternative to providing opponent AI with equal, if not more consistent, capabilities. Minimax, another tree-based search, expands an entire tree of successive game states and can be depth-limited like MCTS. Notably, Minimax will always converge to an optimal solution for zero-sum games if expanded completely, also known as the Nash Equilibrium (Russell and Norvig, 2002). MCTS has also been proven to converge to an optimal solution but requires a large enough sample of random path traversals (Kocsis et al., 2006).

The argument against Minimax, however, is the algorithm requires a heuristic function to determine the maximum or minimum value of a game state. This heuristic function must be an accurate evaluation of the game state and determining this heuristic for a multi-objective game is non-trivial. *Hive* has a main objective of surrounding the opponent Queen; however, to do so, the player must be able to move freely and the player must not have their own Queen surrounded. A certain weighting would need to be applied for each of these sub-objectives to play optimally.

Conversely, MCTS does not require a heuristic function. Instead, it relies on terminal states - typically, where a player has won - to assign a reward for a player reaching this state. If White, for example, reaches a state in the tree where it surrounds the Black Queen Bee, a positive award would be assigned to this state. On the contrary, if White were to reach a state where its own Queen Bee was surrounded, a negative reward is given. Finally, if a terminal state is reached due to depth limits only, zero

reward is given. Utilising this approach removes the requirement of domain knowledge or strategy for determining the heuristic function.

3.4.2.2 Exploration and Exploitation

Pure Monte-Carlo Tree Search will continually extend children of leaf nodes until a terminal state is reached. Expansion of the nodes and their children are random, though the final selection is usually via some heuristic to favour or balance exploration of moves with few simulations and/or exploitation of moves with positive rewards. Kocsis and Szepesvari (2006) introduced a method called Upper Confidence Bound (UCT), which proves balancing of the two metrics. Given by equation 3.1, this approach selects the node with the highest value, and is the method chosen for this implementation.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \tag{3.1}$$

- w_i is the number of positive rewards for the node after the i-th move
- n_i is the number of simulations of the node after the i-th move
- N_i is the total number of simulations after the i-th move
- c_i is the exploration constant - typically equal to $\sqrt{2}$

3.4.2.3 Depth-Limitation

Due to the vast array of possible moves available to a player, regardless of whether it is turn four or forty, a positive or negative reward which MCTS uses to determine the best action may not always be found. That is to say a terminal state, where the game has been won by a player, was not reached during the expansion of the tree. Not all paths through the tree end in a reward state due to loops of repeating moves, and those which do are so deep that the processing time required to find that terminal state is infeasible for real-time gameplay. As such, a depth-limitation is introduced.

Depth-limiting the tree involves allows a specified number of moves to be executed for each branch of the tree. Even if a game-over state is not found, the state is considered terminal and a reward of zero for that state is returned. Depth-limited MCTS helps to speed up decision making with the inevitable, but necessary, trade-off of an inexhaustive search and less than optimal decision on average.

3.4.2.4 Root Parallelisation

As the processing time for MCTS to determine the best action is large, optimisation techniques to speed up the process are considered. Root parallelisation is one of these techniques. Cazenave and Jouandeau (2007) introduced the idea as the "Single-run parallel algorithm", among other parallelisation techniques, and it has since been adopted illustratively as root parallelisation.

Root parallelisation consists of building multiple trees in parallel beginning with the same root node and child nodes, representing a state and a set of possible actions, respectively. The randomness of the paths generated thereafter ensure each tree has different rewards associated with each child node. When all trees have completed their simultaneous expansions, the trees are merged by the root. Overlapping child node rewards are summed to form one reward and those which do not overlap are left standing alone. The best action is then chosen by MCTS as normal. Figure 3.5 provides insight into this approach.

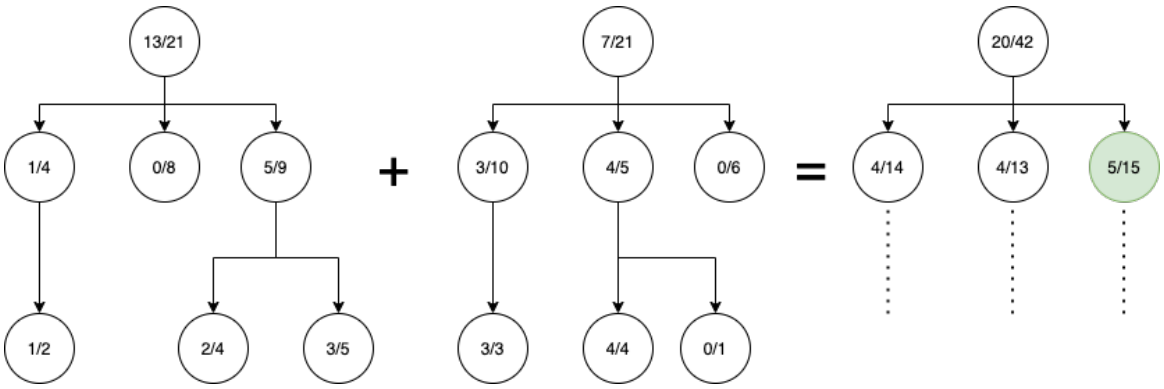


FIGURE 3.5: Root parallelisation reward summation

The benefit of root parallelisation is apparent as it allows a greater exhaustive search which therefore leads to a closer-to-optimal action chosen. The current implementation allows for as many parallel trees as the CPU can handle; typically twice the number of CPU cores available.

It should be noted that parallelisation of MCTS does not speed up the process of finding the best action. In fact, the speed may even be fractionally slower due to the parallelisation framework initialisation and the combination of trees. However, there are alternative adjustments which can be made to decrease processing time. One of which includes reducing the number of iterations the search executes; essentially trading vertical scaling for horizontal scaling by lowering the iteration count but increasing the number of trees. A second adjustment includes reducing the depth limitation of the trees further. In doing so, any paths which would have resulted in finding a goal state at a depth one or more greater than the new depth would not be found; nevertheless, the benefit is that more actions and paths are explored, thus providing greater opportunity to discover shallow goal states.

3.5 Swarm AI

Swarm intelligence is a problem-solving methodology derived collaboratively via the minute interactions of unsophisticated information processors, largely noted in nature. The idea of a swarm itself implies randomness, stochasticity, and multiplicity, and the intelligence is gleaned from the complex, emergent behaviour that arises as the individuals interact with each other. Complexity itself does not equate usefulness in problem solving, as often the simplest solutions are the most appropriate. However, the natural complexity seen in biological systems is almost always useful. Complexity through evolution is extremely difficult to achieve and maintain, and complex behaviours with little-to-no benefit developed via evolution are quickly discarded, therefore those that remain are often strong emergent behaviours which solve problems effectively.

It makes theoretical sense, then, that building algorithms inspired by these natural processes would achieve similar emergent behaviour. As such, the past three decades have been saturated with attempts at modelling any and all kinds of nature-based system to solve complex problems. Of the emergence-related swarm algorithms, the most effective and most widely used today are Particle Swarm Optimisation (PSO), Ant Colony Optimisation (ACO), and the bees algorithm (not to be confused with Artificial Bee Colony).

Ant Colony Optimisation, built upon the simplification that ants will take the path with the strongest pheromone left by previous ants, or a random path if no pheromone exists, is only successful in cases the ants should return to the starting position, and thus is more suitable for route planning and path finding (Dorigo, 1992). The bees algorithm is similar in that bees search for food and then return, although no pheromone trail is left. Instead, the bee returns to the hive and "waggle dances" for a period of time proportional to how good the food source is. This in turn recruits other bees to go to that food source (Pham, 2005).

In *Hive*, manoeuvring the insects to find the goal and return back to its original position, or a central "hive", would be a vast waste of resources, where resources are the number of turns taken by a player. Therefore, the most applicable swarming algorithm for *Hive* is PSO as the insects only converge towards a goal without intentions to return.

3.5.1 Design Process for Swarm AI

Building the most effective Swarm AI for *Hive* is an iterative process due to the number of considerations in terms of hyperparameters and communication amongst particles. As PSO has no form of communication other than awareness of best global particles, certain communication approaches need to be implemented to ensure that, whilst swarming, the player's own Queen Bee does not become surrounded. Additionally, PSO and other swarm-based methods have no consideration of turns, and thus all particles update their vectors simultaneously.

This factor means a decision making process needs to occur to determine which particle will have their position updated each turn.

Importantly, to begin developing a swarm algorithm, there must at least be particles to swarm with. In *Hive*, however, the game begins with an empty board, and the success of trialling swarming methods with few movable pieces would be difficult to ascertain. Consequently, to build and evaluate a swarm model against opponents, the game is initialised with all of the playable pieces on the board in random positions via legal moves.

3.5.2 Fitness Function

As PSO requires a fitness function to establish a particle's proximity to an optimal solution, one must be derived from the rules of *Hive*. Naturally, if the goal is to surround the opponent's Queen Bee, the goal would be the distance between the particle's current position and the position of the opponent's Queen Bee. As all but one insect, the Beetle, can be a minimum distance of one hexagon to the Queen Bee, this distance is the optimisation target. The Beetle, therefore, when climbing on top of the Queen Bee, does not actually contribute to the surrounding of the piece. It does, however, lock the Queen Bee in position to prevent it moving, which could be a useful strategy for the rest of the swarm to catch up should the Beetle be one of the first to reach it. As a result, any distance of zero will be converted to a distance of one such that a Beetle has equal desire to either surround the opponent Queen Bee or climb on top of it.

3.5.3 Global PSO Only

Global PSO is an appropriate first approach to consider as *Hive* can be naively reduced to a unimodal objective game. To trial the implementation described above, all pieces except the Queen Bees are swapped for Ants as they are able to move in all directions around the hive infinitely. This will demonstrate the progressive movement of the

swarm. Secondly, as PSO is typically used for simultaneous updating of positions, this will also be applied to the game by only allowing one player to move infinitely whilst the other is immobile.

As the new velocity elements calculated will not be an absolute value of integer length for either $[\Delta row, \Delta col]$, and the insects can only move by integer units across or around the hive, the new calculated velocities need to be converted appropriately. To do so, the velocity vector is converted to a new vector using an estimated threshold. If an element of the velocity vector has a mantissa with an absolute value greater than 0.25, the number is rounded up (or down depending on sign) to the nearest integer. This threshold is lower than the expected 0.5 to encourage movement and could be considered a hyperparameter for optimisation.

Furthermore, the direction in which the new velocity intends the insect to move may not be, and likely is not, a valid move. This is due to the inability to move to the suggested position due to the space being occupied or the limitations of the insect's movement. To resolve this, the nearest available move to the target position is used. If there are no available moves, no move is made. This process ensures all insects that could move at the initial state do so, though it can produce some anti-swarm behaviour where an insect may move away from the swarm as it is the only/best move possible. This process does not consider insects which are able to move only *after* another insect has freed them by moving themselves away due to the iteration through the list of insects.

Figure 3.6 provides a rough visualisation of how effective the swarming behaviour can work in this domain. Figure 3.7 shows what the swarming behaviour looks like with all insects.

Due to the iteration through the set of insects, where possible moves are determined and the nearest move to the desired velocity is selected, it is possible that a Beetle may be moved twice in one complete swarming move should the Beetle move to a position covering another one of the player's pieces. Therefore, the underlying insect will not be able to perform it's desired move and instead the Beetle performs their

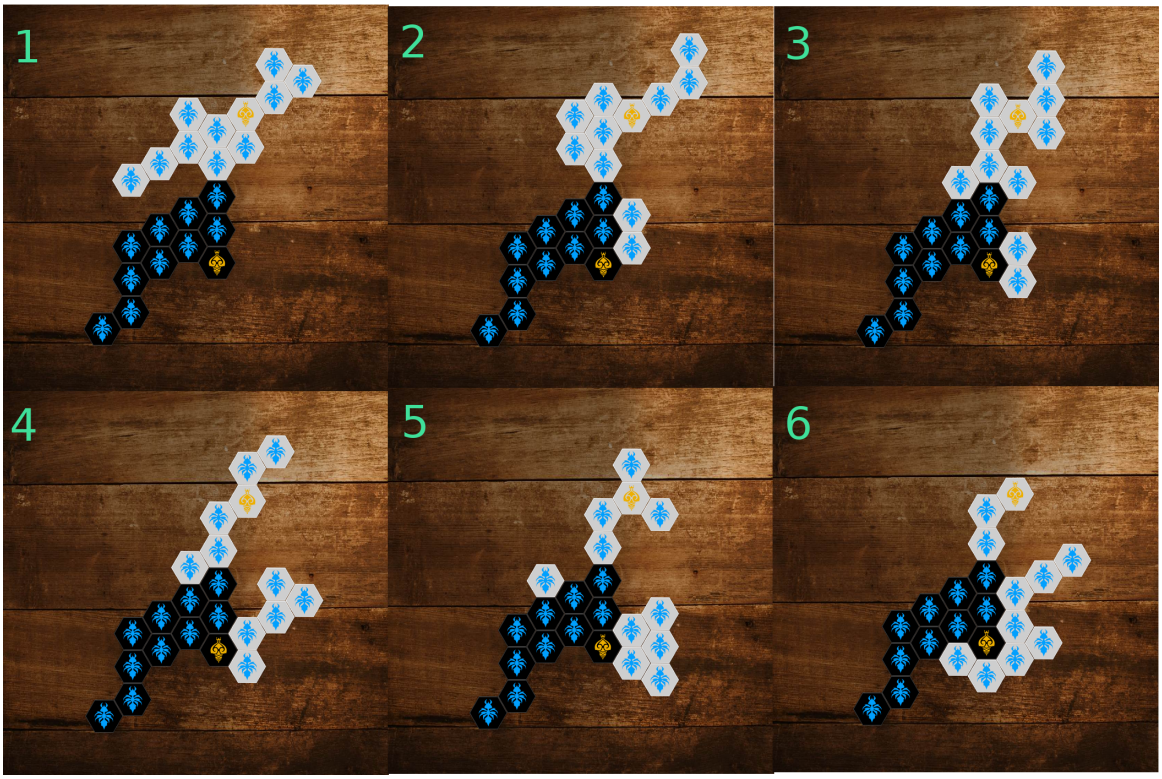


FIGURE 3.6: Global PSO only with Ants and infinite White player moves (Random seed of 9)

move. However, all possible moves made are still valid according to the rules as the set of possible moves are determined during each iteration, not during swarming. This may slightly affect the results of comparisons of different PSO methods.

Trials were conducted to view the impact if all possible moves were calculated prior to any swarming move being made. Findings were that insects may move to positions which are are illegal due to other insects moving away from their original position of which their move was based upon.

Nevertheless, these caveats are not a concern for when the swarming behaviour is implemented in the final version as only the single best move will be chosen via criteria discussed in Section 3.5.5.

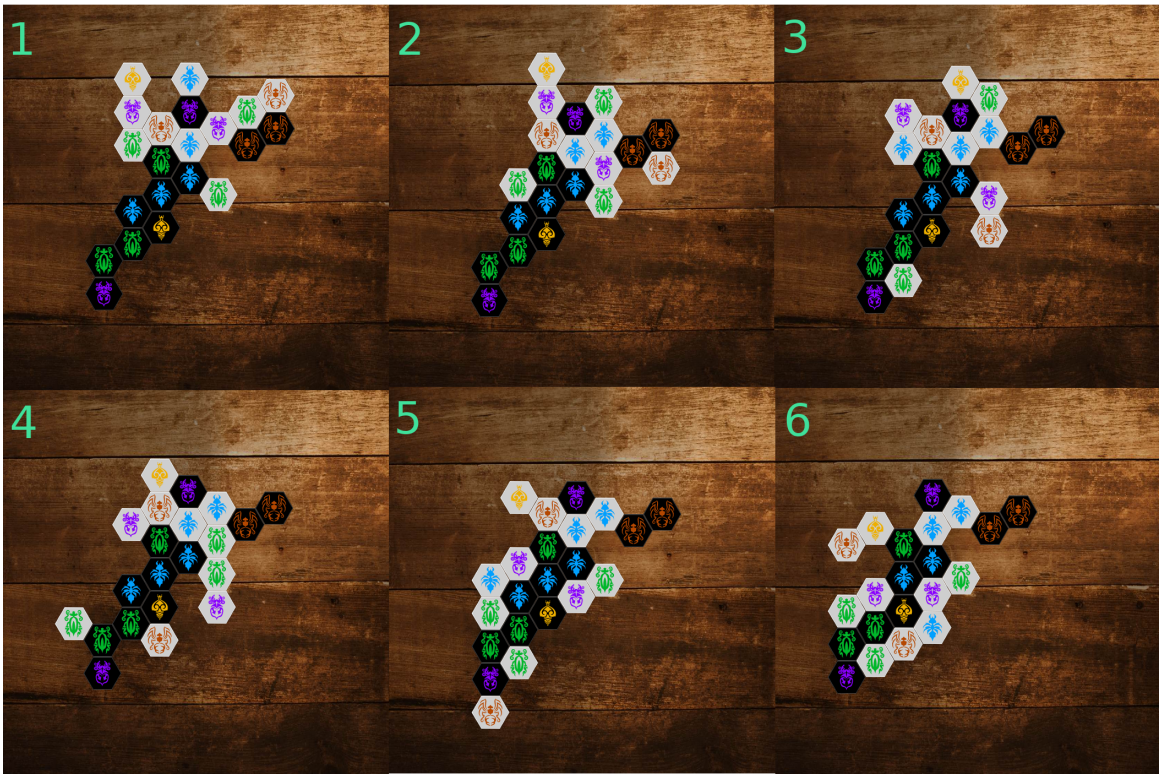


FIGURE 3.7: Global PSO with all insects and infinite White player moves

3.5.4 Local Vicinity PSO

To test the local vicinity PSO against global PSO, random boards are generated and the swarm is allowed to move infinitely until it converges on the opponent Queen Bee or until a maximum number of 100 moves have occurred. A log of these results are taken for varying vicinity radii.

To eliminate any bias from the starting position of the randomly generated board, where the player's pieces are already clustered together, 20 random moves per player are executed once the board is generated. Some games may terminate due to random game-over states. Moving randomly initially may allow the local vicinity PSO to have some vicinities with particles unique to them to differentiate from Global PSO.

Figure 3.8 shows the results when only Ants are used and when all insects are used, respectively. As can be seen from the results, the games are inevitably won more often when all insects are used, though, with Ants, it takes fewer moves to win due to their mobility. They win fewer games as they cannot slide into certain spaces to surround

the Queen Bee because of the Freedom to Move Rule. Few games overall are actually won, partly due to the turn limit but mainly due to the likelihood many pieces are pinned in place as a consequence of the immobile opponent. The difference between global and local vicinities win rate is small, with local vicinities generally performing better. For the purposes of developing Swarm AI further using communication, local vicinities with a vicinity radii of two will be used herein.

```
Testing swarming with Global Best PSO...        Testing swarming with Global Best PSO...
200 games simulated...                          200 games simulated...
400 games simulated...                          400 games simulated...
600 games simulated...                          600 games simulated...
800 games simulated...                          800 games simulated...
1000 games simulated...                         1000 games simulated...
White won 70 times with avg plies of 20         White won 132 times with avg plies of 32
Black won 0 times                               Black won 0 times
.                                               .
Testing swarming with Local Vicinity Best PSO...Testing swarming with Local Vicinity Best PSO...
Vicinity Radius = 1                             Vicinity Radius = 1
200 games simulated...                          200 games simulated...
400 games simulated...                          400 games simulated...
600 games simulated...                          600 games simulated...
800 games simulated...                          800 games simulated...
1000 games simulated...                         1000 games simulated...
White won 59 times with avg plies of 23         White won 138 times with avg plies of 34
Black won 0 times                               Black won 0 times
.                                               .
Testing swarming with Local Vicinity Best PSO...Testing swarming with Local Vicinity Best PSO...
Vicinity Radius = 2                             Vicinity Radius = 2
200 games simulated...                          200 games simulated...
400 games simulated...                          400 games simulated...
600 games simulated...                          600 games simulated...
800 games simulated...                          800 games simulated...
1000 games simulated...                         1000 games simulated...
White won 68 times with avg plies of 21         White won 143 times with avg plies of 32
Black won 0 times                               Black won 0 times
.                                               .
Testing swarming with Local Vicinity Best PSO...Testing swarming with Local Vicinity Best PSO...
Vicinity Radius = 3                             Vicinity Radius = 3
200 games simulated...                          200 games simulated...
400 games simulated...                          400 games simulated...
600 games simulated...                          600 games simulated...
800 games simulated...                          800 games simulated...
1000 games simulated...                         1000 games simulated...
White won 72 times with avg plies of 20         White won 129 times with avg plies of 33
Black won 0 times                               Black won 0 times
.                                               .
-----------------------------------------------------------------------------------------
Ran 4 tests in 3634.396s                        Ran 4 tests in 1634.480s
```

FIGURE 3.8: Comparison of Global PSO vs Local Vicinity PSO - Left: Ants only, Right: All insects

3.5.5 Communication

As only one move is allowed per player in the real game, a decision on how to select this move using PSO must be made. Communication among the insects on the board may allow for a better choice of move. Communication in Swarm AI for *Hive* is

inspired by ACO pheromones and Cunha's (2015) proposed AI with *intentions* to determine the next appropriate action. A model for building an insect's intention is established by considering the factors that contribute to it and their weighting to produce an overall score.

3.5.5.1 Intention Factors

Matzinger's (2002) Danger Theory will be a primary factor of the intention score. Danger Theory implies an awareness of current or approaching danger and reacting accordingly. There is no more obvious danger in *Hive* than that of having your own Queen Bee surrounded. If a particle realises it is close to its own Queen Bee and she is about to be surrounded, perhaps if there are four or five insects adjacent to her, the particle may strongly desire to move away. The Queen Bee itself will implement its own form of Danger Theory and will have a stronger intention to move if it senses it is being surrounded. To keep the AI decentralised, only particles which have the Queen Bee in their vicinity will have a level of desire to move away from the Queen Bee.

One other possible factor is to simply select the move with the greatest velocity, implying that the desire to move to be with the rest of the vicinity is very strong. Another factor could be to give strong weighting to a move where the distance from the desired position to the closest valid move to that position is minimised - or, in other words, an accuracy of move factor. Although, this too has its problems, as will be demonstrated later. For consideration could also be the particle's fitness itself. If a particle believes itself to be currently unfit, it should intend to move.

When the intention formula is decided via experiments in Section 4.2, each vicinity will have its own *best intention*, determined simply as the particle with the maximum intention score. From these best intentions, only one insect can be selected to move, and this is where communication among the vicinities applies. It is quite possible that, due to the intention scoring, multiple insects from different vicinities may have equal highest scores. When this is the case, there is a second selection phase that

occurs where the insect's movement ability is used as a tie-breaker. Much like *Chess*, some pieces are inevitably ranked stronger than others. The ability of an insect refers to its ability to move in and around the hive, ordered from best to worse as: Ant, Beetle, Grasshopper, Spider, Bee. Using ability as a factor in the first phase will likely result in games where only Ants and Beetles play most of the game. Experiments conducted in Section 4.2 will determine the vicinity factors formula.

Chapter 4

Experiments and Evaluation

4.1 Overview

This chapter discusses the experiments conducted using various forms of Swarm AI against various opponents. Many variations in intention factors are introduced and analysis of their playing style against immobile and randomly playing opponents is provided. Upon deciding the appropriate Swarm AI intention factors, the Swarm AI is then tested against a randomly playing opponent and a MCTS opponent for 100 games each. The MCTS opponent will also play against the randomly playing opponent for fair comparison of results.

All experiments conducted are logged by denoting every move made by each player in a text file representing one completed game. Logged games are able to be replayed via the UI; instructions of how to do so are provided in Appendix A. Additionally, some games in this chapter may be referred to by video screen captures of a replayed game.

4.2 Communication Experiments

To determine an appropriate intention for each insect, factors mentioned earlier are trialled. The five factors to consider are as follows:

- **Danger Theory** - proximity to own Queen Bee and number of insects surrounding her
- **Highest Velocity** - the particle with the greatest velocity implies it is far from the swarm
- **Highest Accuracy** - the particle that can move closest to where PSO intends
- **Worst Fitness** - the particle furthest from the goal of surrounding the opponent Queen Bee
- **Combinations** - some factors may work together to form better gameplay

Both manual and automated tests are conducted to determine which factors are appropriate. The manual tests are to visually note the behaviour of the insects, whereas the automated tests are to empirically show how well it performs. All factors below include Danger Theory as an addition to the intention score. Again, these tests are without any opponent movement. Tests where the opponent moves will occur when an appropriate Swarm AI is decided.

Logs of the games using this intentions listed accompany this report and can be replayed with UI via the program.

4.2.1 Danger Theory

As mentioned earlier, Danger Theory will be incorporated into all intentions. The increase in intention score is chosen to have an impact proportional to the number of opponents surrounding the player's own Queen Bee. The Danger Theory score is chosen such that when five insects surround the Bee it will always have a greater

intention than others, and when four surround the Bee the addition to the score may or may not create a maximum vicinity intention score.

For example, a velocity intention is typically an integer from [0..5]. If five surround the Queen Bee, an addition of 6 is given, making it always a greater intention. If four surround the Queen Bee, an addition of 3 is given, permitting other insects with stronger intentions to act. The Queen Bee's own intentions for the same situations are 8 and 4, respectively. The Queen Bee has greater intention to move because its own movement is likely to result in equal or fewer insects around her than one other insect moving away.

4.2.2 Highest Velocity

That with the highest velocity shall win in its vicinity. As the method of actually moving an insect is to choose a valid move closest to the desired move, selecting the insect with the greatest velocity may result in a move which is further in distance to a desired position than moves other insects could make. This results in deviations from the natural swarming behaviour. Nevertheless, using the highest velocity factor demonstrated promising convergence towards the goal with the outer particles moving inwards; though, not necessarily the further particle from goal at all times.

A disadvantage of this approach, again noted visually, is that the particles do not remain at goal when they have reached it. This is most due to the inertia component of the PSO velocity formula and partly due to the particle's desire to move in the direction of the vicinity's best particle. As such, when close to surrounding the opponent, repeating patterns of moves can result.

Figure 4.1 illustrates the convergence towards the goal. Note, this execution uses a vicinity radius of two. All moves executed are generally headed towards the vicinity best, not necessarily the goal. This figure also highlights a few key points.

Notably, in Panel 2, the Beetles and a Grasshopper move south as particles south in their vicinity have a greater fitness. The white Queen Bee is far from the goal,

though this distance has no effect on its velocity, as it typically would in Global PSO, as those in its local vicinity actually have a worse fitness than the Bee. Additionally, the Ant to the left of the Panel remains stationary, and this due to having no other particles in its vicinity. The PSO formula accounts for this with its inertia property. However, in this experiment, all velocities of the particles were initialised to zero, and thus the Ant could have no inertia and could not explore the board to either join a vicinity or find the goal. This suggests velocities should be initialised randomly, even if it results in some insects moving in the "wrong" direction for their first turn.

From Panel 3, it can be deduced that the Grasshopper finishing nearest the opponent Bee made three of the four moves until that state. This is purely because in each turn it is the furthest insect from goal which is able to move in a positive manner without breaking the hive. During Panel 3's turns, the Grasshopper moves into a position two spaces away from the left Ant, and thus that Ant's velocity becomes non-zero and is brought into play during Panel 4. To combat the effect of having no other particles in a vicinity, it may be reasonable to consider increasing the vicinity radius.

Panels 4 through 6 show further convergence of those able to move towards its local best particle. Note that the Spider left of Panel 4 is only one space West of optimal fitness and so attempts to move East around the Hive into a space. Although this is desirable behaviour, the outcome is that the Spider's limited movement prevented it from landing exactly in the gap and thus its inertia overshoots it north. These types of moves often repeat and is why so many moves take place when just one or two moves could win the game.

4.2.3 Highest Accuracy

Unlike the velocity factor where the score given is the velocity itself, the accuracy, or the distance between the desired move and the nearest valid move, cannot be assigned as it would be inversely proportional to this distance. As the velocity is typically in range [0..5], the accuracy score is also constrained to this range by:

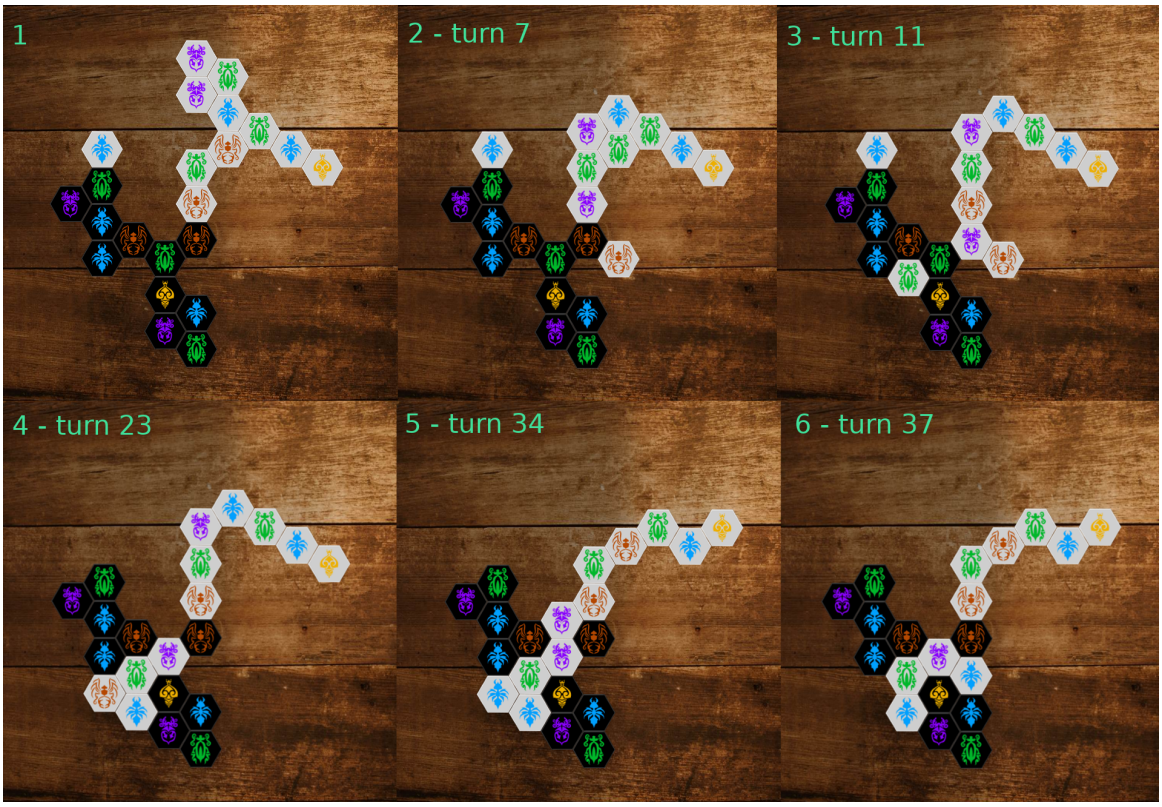


FIGURE 4.1: Progression of swarm using Velocity intention with infinite moves

$$accuracy = max(0, 5 - dist(p_1, p_2)) \tag{4.1}$$

where p_1 and p_2 are the desired move and the nearest valid move.

When the highest accuracy is the intention factor, it is evident that the movement ability of the insects is critical. In all stages of play, the Ant will often be the choice insect to move due to its unlimited range. The Beetle will also often be a primary choice when the velocity of the insect intends to move it one position as it can freely climb over the hive. Once a Beetle begins to move towards goal it continues until it reaches it. When it does so it will often be the choice insect to move and will repeat its movements.

The repetition of Ant and Beetle movements is occasionally broken by outer insects intending to move inwards. Their velocity in this scenario has been established by an insect moving into its vicinity with a better fitness. Though, even in cases where a velocity has been established, the insect does not necessarily execute a move due to

the inaccuracy of its desired move against its valid moves. This can result in large numbers of turns executed with repeated circling of the opponent Queen Bee and interspersed variance in insect movements.

Very often this intention factor will not result in a terminal state as not enough insects are collectively moved. Those which do terminate likely have more opponent pieces surrounding the opponent Queen Bee.

Figure 4.2 illustrates the typical repeating states of the accuracy intention. There is clearly a winning terminal state in one move, yet PSO has no understanding of approaching winning states. Instead, as the circled Beetle has velocity from moving north into its current position, its inertia will push it further north, followed by it correcting itself southwards, and again its inertia carrying it an additional space south. This suggests implementing a lookahead for empty surrounding spaces may prove beneficial.

FIGURE 4.2: Repeating Beetle movement near terminal state using Accuracy intention

4.2.4 Worst Fitness

Insects with the worst fitness, or those furthest from goal, are those chosen to move when using this factor. Unlike the previous two factors, this approach will *always* move the insect furthest from goal if possible moves are available. Figure 4.3 demonstrates why this approach often fails to converge - the Spider pictured continuously attempts to move towards goal but repeats moves around the hive. No matter where this Spider moves to, it continues to have the worst fitness of any insect which can move, and thus a loop occurs.

This factor works well in clustering the insects together but has the evident major flaw that it often fails to progress when the outermost insect is blocked.

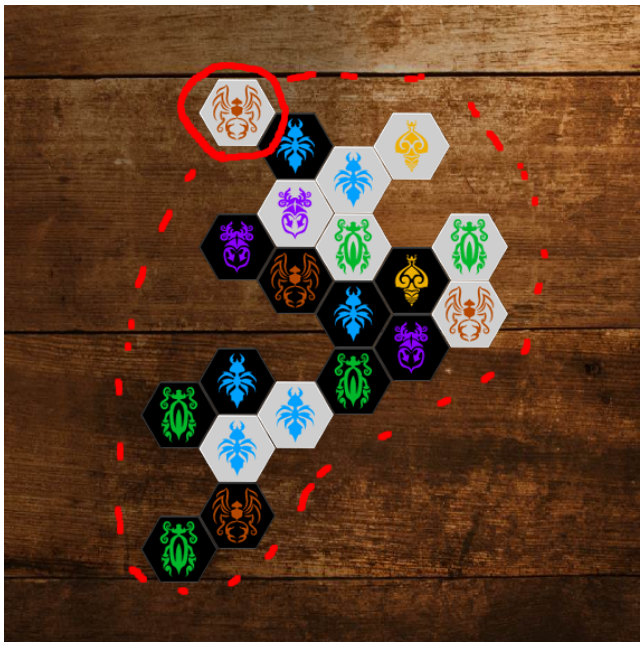


FIGURE 4.3: Outermost insect repeatedly attempting to converge inwards using Worst Fitness intention

4.2.5 Velocity with Accuracy

This summation of velocity and accuracy intentions produces interesting results where the properties exhibited by the previous two intentions work together relatively well. The outermost insects often move inwards, like they do with the velocity intention,

and te clustering effect found in the accuracy intention is also evident. However, once clustered, the algorithm often repeats moves of the Beetle. Both the velocity and accuracy intentions are the catalyst behind this as the Beetle almost always scores highly in both when near an optimal position.

4.2.6 Velocity or Accuracy

This combination intention utilises velocity when the insect does not have the opponent Queen Bee within two spaces of its position and accuracy when it does. The idea for this intention would be that the velocity of an insect could bring the insect towards the centre and the accuracy intention would finalise its position close to the objective. Consequently, what resulted is the two factors working quite literally independently, where both the advantages and disadvantages of both are present.

In Figure 4.4, the Spider highlighted repeats its moves east and west due to its velocity carrying it towards its vicinity's best, and the highlighted Ant repeatedly moves north and south due to the Ant's ability to accurately land anywhere it wants providing the space is free and it does not break the Freedom to Move Rule. This experiment assures that the accuracy intention is not an appropriate candidate to continue with.

4.2.7 Velocity or Direct to Goal

Particle Swarm Optimisation has been proven to work well in situations where concurrent updating of all particle velocities and positions is allowed, though it is evident that it may not be the best approach in its application to turn-based games. Most of the above intentions converge towards goal but fail to surround the Queen Bee effectively.

Fitness functions in PSO are typically used as methods of evaluating closeness to the optimisation target when the exact location of the target in D-dimensional space is unknown. However, *Hive* is a perfect-information game - the location of the target is

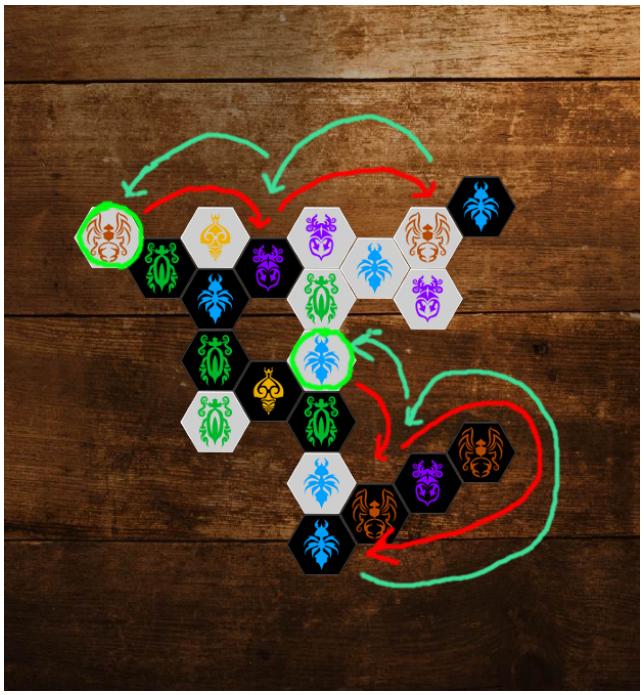


FIGURE 4.4: Spider repeating due to Velocity intention, whilst Beetle repeating due to Accuracy intention

always known - so it may be beneficial to consider an alternative to PSO in the later stages of play where the majority of movable insects are near the goal.

A custom swarming approach, specifically for the game of *Hive*, would be to apply moves which minimise the distance to the Queen Bee when the particle has the Queen Bee in its vicinity. If any particle has the Queen Bee in its vicinity, the particle "knows" where the Queen Bee is, can determine all possible empty spaces around her, and will intent to move to an empty space if the insect's movements allow it. Once a particle moves beside the Queen Bee, it will not move thereafter unless the Queen moves, as it will in real games.

The slight difference is that the particles swarm towards the global/vicinity best particle in PSO, whereas the insects swarm directly towards goal in this approach when the goal is in the vicinity. This approach is a hybridisation of the best factor noted, velocity, and a method to overcome repetition in late stages whilst keeping swarming via vicinity behaviour.

4.2.8 Results of Intentions

Intention Formula	With Tiebreaker	Immobile Wins	Swarm AI Wins	Avg Moves to Win
Velocity + DT	N	0	90	21
	Y	0	114	21
Accuracy + DT	N	0	47	19
	Y	0	56	22
Fitness + DT	N	0	63	19
	Y	0	63	19
Velocity + Accuracy + DT	*N*	0	108	20
	Y	0	112	20
Velocity OR Accuracy + DT	N	0	51	20
	Y	0	71	22
Velocity + DT OR DTG if fitness <= V radius	N	0	**162**	16
	Y	0	**166**	15

TABLE 4.1: Intention factors results of 300 games each against an immobile opponent. Maximum of 40 moves. All intentions incorporate Danger Theory.

The results shown in Table 4.1 indicate that the hybrid method of Velocity and Danger Theory when the insect cannot "see" the Queen Bee, and DGT when it can, is the strongest when facing an immobile opponent.

Clearly evident throughout the results in the table, the tiebreaker appears to have a beneficial effect on the outcome of the games. Therefore, the tiebreaker will remain a form of additional intention in the final Swarm AI product.

Table 4.2 reveals how these intentions perform when the opponent is able to make random moves. Interestingly, slightly more wins were accrued for the strongest intention combination, *Velocity or Direct to Goal*, when the opponent was able to move randomly. This is likely due to positions randomly opening up providing the swarm with freedom to move.

Intention Formula	Random Wins	Swarm AI Wins	Avg Moves to Win
Velocity + DT	0	101	21
Accuracy + DT	2	69	22
Fitness + DT	2	96	22
Velocity + Accuracy + DT	5	111	19
Velocity OR Accuracy + DT	7	67	20
Velocity + DT OR DTG if fitness <= V radius	6	**184**	17

TABLE 4.2: Intention factors results of 300 games each against randomly playing opponent. Maximum of 40 moves. All intentions incorporate Danger Theory and tiebreaker.

4.2.9 Frequency of Final Intention Factors

The factors of velocity, Danger Theory, DTG, and movement ability tiebreaker are evidently the appropriate combination to form a communication and decision-making process within the swarm. Of interest, however, may be the frequency in which the velocity factor is used versus DTG, as only one can be used per intention assignment.

The Swarm AI played against a randomly playing opponent for 100 games with a limit of 40 moves after the randomisation of the pieces on the board. The total number of intentions assigned was **12,922**, where **63.5%** of those intentions were velocity with Danger Theory and **36.5%** were DTG intentions.

This distribution of intentions is as expected, implying that approximately two thirds of the time the AI utilises PSO to converge in a swarm-like manner towards the objective, with the remaining time utilised to attempt to move as close to the objective as possible. This is an additional hyperparameter which could be investigated in future work. If the preference was to execute a greater percentage of swarm moves, the vicinity radius of a particle used for the DTG intention could be reduced.

4.3 Testing Final Swarm AI

To test the final product, the Swarm AI is pitted against a randomly playing opponent, seen earlier, and a MCTS opponent. MCTS also plays against the randomly playing opponent. Each test is conducted with 100 games with 22 initial random moves, followed by 20 random moves per player, concluding with AI moves thereafter until termination or until 40 AI moves are executed by any player. The results of these tests are shown in Table 4.3.

4.3.1 MCTS Analysis

Analysis of MCTS as an opponent is important as it provides insight into the strength of the Swarm AI by comparison. The decision to use a Monte Carlo approach as a

Player 1 vs Player 2	**Player 1 Wins**	**Player 2 Wins**	**Draws**
Swarm AI vs Random	55	0	45
MCTS vs Random	53	0	47
Swarm AI vs MCTS	41	38	21

TABLE 4.3: Final results of Random, MCTS, and Swarm AI playing head-to-head in 100 games. MCTS is depth-limited to 13 with 500 iterations.

suitable opponent for a newly developed AI was twofold: the algorithm had been proven to succeed against a randomly playing opponent in *Hive* in previous research (Konz, n.d.), and, secondly, the algorithm is proven to converge to an optimal solution given infinite resources without the requirement of any strategy or heuristic.

In hindsight, the choice of MCTS is theoretically sound, but the demand for resources required to converge to an optimal solution is far greater than is reasonable for real-time play. With the test constraints mentioned earlier of 40 AI moves, a single game of *Hive* with MCTS playing against an opponent which reacts almost instantaneously can last up to 45 minutes, or a time of 1 minute per move. In physical games this would be infeasible, but in terms of computing time its simply unacceptable.

The MCTS AI used in this final test is depth-limited to 13 moves with 500 iterations and root parallelisation across 11 CPU threads, set for the purpose of gathering results without an inordinate amount of resources used. *Hive*, like *Chess*, has an almost-intractable number of paths in the state space to be explored, and unfortunately this constraint limits the likelihood of finding non-zero reward states. When only non-zero reward states are found, the AI essentially plays randomly.

Nevertheless, MCTS does appear to frequently win games. It performs best when there are one or two moves to execute to secure a win, for example when the opponent Queen Bee has four or five insects around her, or when its own Queen Bee is close to being surrounded and it requires a move to safety.

4.3.1.1 Video Log Example

Video Log: *8_RANDOM_MCTS_5*

For **AI Move 10**, it be can be seen that there is a possibility of winning in three moves, yet MCTS executes a move that appears to be random as it moves away from an attacking position. However, for **AI Move 18**, MCTS clearly has two valid moves to advance an attack and it does choose one of them, despite this move not actually touching the opponent Bee. This advance continues in the next move for MCTS, is rewarded by a chance blunder by the randomly moving opponent moving next to its own Queen Bee, and then MCTS surrounds her in its next move. This video proves MCTS is a worthy opponent in some cases.

4.3.2 Swarm AI Analysis

From Table 4.3, it is understood that Swarm AI performs similarly to MCTS, winning 52% of games which did not end in a draw. The results alone may be considered satisfactory, though video log analysis of the factors which lead the AI to win, lose, or draw may provide insight into how the AI could be improved, and how it may be translated into other games. Many videos will be referenced in this section to understand the interesting properties that arise.

4.3.2.1 MCTS Win Example 1

Video Log: *1_SWARM_MCTS_5*

From the very beginning of the AI moves, this replay shows many of the properties programmed into the AI. **AI Move 0** shows MCTS executing a strong move towards the Queen to have five insects surround her. **AI Move 3** shows Swarm AI utilising Danger Theory by moving the only movable piece surrounding its own Queen. This defensive strategy explains the reason why Swarm AI never lost against a randomly playing opponent, and it is likely one of the few game theory optimal strategies that exist in these situations.

With MCTS now having to execute a minimum of three moves to win, MCTS struggled to find a terminal state and executed an apparently random move. **AI**

Move 5 shows Swarm AI executing a *velocity* intention by bringing an outermost Beetle inwards towards it's vicinity's best known position (Grasshopper). MCTS wisely chooses to move its Queen to safety in **AI Move 6**. Swarm AI then continues with the same Beetle inwards due to a new vicinity best known position (Ant) and the inertia it gained from the previous move. Once again, MCTS appears to execute a random move whilst Swarm AI then continues with the same Beetle towards the objective. In doing so, the Swarm AI has placed the Beetle in an adjacent position to its own Queen Bee, leaving only one space free. MCTS is certain to execute a winning move if only one move is required, and does so.

The fact that the Beetle moved into a position which endangered the Queen is not something that was considered during the intention factor design; only when pieces were already in such a position would they desire to move away. If this was a considered factor, however, the issue arises that the insect then would have knowledge of the environment it was going to move to. This is reasonable if the new position is within its vicinity, perhaps, yet further foresight deviates from traditional biologically-inspired algorithms as they are typically based on natural instinct and localised communication.

4.3.2.2 MCTS Win Example 2

Video Log: *2_SWARM_MCTS_5*

This replay demonstrates the *velocity* intention being the primary intention factor for movement in the first six moves by Swarm AI. All pieces moved during these six moves do so as they strongly desire to gravitate towards their vicinity's best position. Some moves may not be considered optimal, though they certainly advance an attack, and others, such as **AI Move 7** where the Grasshopper jumps adjacent to the opponent Bee, are clearly optimal.

Interestingly, after **AI Move 10** the Swarm AI has an option to move a Spider adjacent to the opponent Bee but instead chooses to move its outermost insect inwards. The *Direct to Goal* intention would certainly have been applied to the

Spider, as it was three spaces from the Bee, giving it a vicinity intention of five. Typically, this score would be highest overall. However, the outermost insect, the Bee, has a *velocity* with a greater value due to the distance between its current position and its desired position. It may seem counter-intuitive that an insect may wish to move further than its vicinity radius as its target is in its vicinity. On the contrary, PSO can create velocities that may pull particles towards and past its target, partly due to inertia. As the particles have initial random velocities, they have inertia even if they have not yet moved.

This occurs once more in the Swarm AI's next move, thus it is evident that this may be an implementation detail worth revisiting. As the *Direct to Goal* intention should likely take precedent the majority of the time, the assigned values for the intentions could be fine-tuned.

As the game advances, the Swarm AI is in a stronger position to win. This holds true until the Swarm AI moves its Bee inwards, accidentally putting itself into a dangerous position, similar to the first example, where MCTS takes advantage of the mistake.

4.3.2.3 Draw Examples

Video Log: *5_SWARM_MCTS_5*

For the majority of this game, the opponent has pinned the Swarm AI such that the Swarm can only move three or four of its pieces. The Swarm AI in this game repeats similar moves with the Spiders for many turns until MCTS eventually frees more pieces. Once they are freed, the swarming process begins but is consequently too late before terminating as a draw.

The repetition of moves seen in this scenario is not a fault of the algorithm itself, as it performs as expected. One possible solution would be to establish recall for the insects such that they are able to trace previous moves executed and favour other non-repeating moves.

Video Log: *11_SWARM_MCTS_5*

This replay is characterised by inability to fully surround the opponent when at a near-terminal state due to the Freedom to Move Rule. Initially, the insects move in a swarm-like manner using *velocity* and *Direct to Goal* intentions. Later, many of the pieces are well-established beside the Queen or are pinned by the opponent and cannot move, resulting in many feeble movements by the outermost insect, the Swarm's own Queen. Finally, when there is only one free space beside the opponent's Queen after **AI Move 40**, no insect can move into it due to the Freedom to Move Rule, if there were possible moves to do so.

What would need to occur in this scenario is movement away from the opponent's Queen to facilitate an opening for ally insects. Alternatively, a piece could move into a position carrying the weight of the hive which allows a Beetle to walk over the hive and drop down. Both scenarios are likely beyond the capabilities of a simple Swarm AI as complex communication would be required.

4.3.2.4 Swarm AI Win Example

Video Log: *20_SWARM_MCTS_5*

In this replay, the Swarm AI executes its first two moves using *velocity* intention and its third move using a *Direct to Goal* intention. It then continues swarming using *velocity* intention until **AI Move 18** where it again goes directly adjacent to the opponent Queen. This move is essentially the hive equivalent of Checkmate as the Swarm positioning has either secured itself adjacent to the Queen or MCTS has pinned itself for the subsequent turn.

Noted in this example and the examples where draws occurred, it is clear that the Grasshopper and Beetles are strong in their own ability to manoeuvre around the hive and enter positions which break the Freedom to Move for other insects. As such, the insects' ability to move could be considered more than just an intention tiebreaker in future work and could be situation dependent.

Chapter 5

Conclusions and Future Work

5.1 Summary

The primary focus of this paper was to research, investigate, and understand if Swarm AI is an appropriate methodology to apply to the two-player strategy board game of *Hive*. Swarm AI has been rarely used as a solution to game-playing as advances in computational resources have accommodated greater success in tree-based search algorithms and machine learning. However, the emergent properties that arise from biologically-inspired swarming algorithms may be beneficial if applied to the appropriate domain.

Expectations prior to undertaking this research were that the Swarm AI built would not be as direct and efficient as other AI tools available, and may even struggle to surround the opponent at all, blocking itself in the process. The findings proved that Particle Swarm Optimisation alone would have indeed produced these results, but the adaptations made to include communications produced an AI which successfully stands equal against MCTS.

Taking inspiration from ACO, where communication occurs via pheromones, it can be said that use of communication via intentions was highly advantageous, with Danger Theory likely being the most promising form of communication to implement. Even

without PSO, Danger Theory could be translated to any two-player strategy game with capturing as part of the objective, such as *Chess*.

The concept of swarming itself looks to be an ideal form of attack, so long as a defensive mechanism is incorporated, and is likely to be suitable for games where coordinated movement of units is required.

As a result of this research, a software product has been built in Python to play *Hive* with the option to choose among players of either human, random, MCTS, or Swarm AI. Furthermore, all games are logged and can be replayed via the UI. Ultimately, Swarm AI itself has been developed and proven to perform well with some limiting factors. Nonetheless, in its current form it is unlikely to beat a capable human player.

5.2 Suggestions for Future Work

The results overall suggest that the Swarm AI could benefit from refinement of the intention strategies for communication. A focus on the Ant and Spider abilities, for example, could encourage exploration of the search space, whilst a focus on the Beetle could exploit the Freedom to Move Rule. The Grasshopper is powerful in that it is capable of both exploration and exploitation in *Hive* and could be modelled accordingly.

Perhaps of most importance is the ability to lookahead and determine if a move would be detrimental to the Queen Bee's safety. Knowledge of the target environment would be required to allow this behaviour, which deviates from PSO and swarming in general. One could argue that, as the hive is fully connected, all pieces can share information globally, rather than just within their vicinity, and thus acquisition of target environment knowledge would be perfectly valid. Alternatively, a centralised system could be built where the Queen Bee is all-knowing and can sense if movement towards her is dangerous. She would then command the potential ally to choose an alternative move. One would need to be careful in development of a centralised system

to ensure the swarming behaviour still exists for the majority of the time, so as to still be called Swarm AI.

Finally, the design and implementation of the software allows players of any kind to be introduced. A recommendation for future work, therefore, would be to introduce an entirely new Swarm AI which takes advantageous of one of the many other existing swarming algorithms. Particle Swarm Optimisation was chosen as it is well-established, non-cyclical, and has visually appealing swarming properties, yet there is certainly opportunity for lesser-known algorithms to be experimented with the game of *Hive*.

References

Abd-Ellah, M. K., Awad, A. I., Khalaf, A. A., & Hamed, H. F. (2018). Two-phase multi-model automatic brain tumour diagnosis system from magnetic resonance images using convolutional neural networks. *EURASIP Journal on Image and Video Processing*, *2018*(1), 97.

al Rifaie, M. M., Bishop, J. M., & Caines, S. (2012). Creativity and autonomy in swarm intelligence systems. *Cognitive computation*, *4*(3), 320–331.

Badue, C., Guidolini, R., Carneiro, R. V., Azevedo, P., Cardoso, V. B., Forechi, A., ... others (2019). Self-driving cars: A survey. *arXiv preprint arXiv:1901.04407*.

Bedau, M. A. (1997). Weak emergence. *Noûs*, *31*, 375–399.

Blixt, R., & Ye, A. (2013). Reinforcement learning ai to hive. *KTH Royal Institute of Technology, Stockholm, Sweden*.

Brambilla, M., Ferrante, E., Birattari, M., & Dorigo, M. (2013). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, *7*(1), 1–41.

Cazenave, T., & Jouandeau, N. (2007). On the parallelization of uct.

Chow, C.-k., & Tsui, H.-t. (2004). Autonomous agent response learning by a multi-species particle swarm optimization. *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, *1*, 778–785.

Coello, C. A. C. (1999). A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information systems*, *1*(3), 269–308.

Colby, K. M. (1981). Parrying. *Behavioral and Brain Sciences*, *4*(4), 550–560.

Condon, J. H., & Thompson, K. (1982). Belle chess hardware. In *Advances in computer chess* (pp. 45–54). Elsevier.

Conradie, J., & Engelbrecht, A. P. (2006). Training bao game-playing agents using coevolutionary particle swarm optimization. *2006 IEEE Symposium on Computational Intelligence and Games*, 67-74.

Cunha, A. A. (2015). Swarm intelligence in strategy games. *Instituto Superior Tecnico, Lisboa, Portugal*.

Del Valle, Y., Venayagamoorthy, G. K., Mohagheghi, S., Hernandez, J.-C., & Harley, R. G. (2008). Particle swarm optimization: basic concepts, variants and applications in power systems. *IEEE Transactions on evolutionary computation*, *12*(2), 171–195.

Dorigo, M. (1992). Optimization, learning and natural algorithms. *PhD Thesis, Politecnico di Milano*.

Eberhart, R., & Kennedy, J. (1995). Particle swarm optimization. *Proceedings of the IEEE international conference on neural networks*, *4*, 1942–1948.

Engelbrecht, A. P. (2013). Particle swarm optimization: Global best or local best? *2013 BRICS congress on computational intelligence and 11th Brazilian congress on computational intelligence*, 124–135.

Fonseca, C. M., & Fleming, P. J. (1995). An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, *3*(1), 1-16.

Greenblatt, R. D., Eastlake, D. E., & Crocker, S. D. (1988). The greenblatt chess program. In *Computer chess compendium* (pp. 56–66). Springer.

Kennedy, J. (2006). Swarm intelligence. In *Handbook of nature-inspired and innovative computing* (pp. 187–219). Springer.

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. *European conference on machine learning*, 282–293.

Kocsis, L., Szepesvári, C., & Willemson, J. (2006). Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*.

Konz, B. U. (n.d.). Applying monte carlo tree search to the strategic game hive. *Technische Universität Berlin, Berlin, Germany*.

Liu, W.-B., & Wang, X.-J. (2008). An evolutionary game based particle swarm optimization algorithm. *Journal of computational and Applied Mathematics*, *214*(1), 30–35.

Messerschmidt, L., & Engelbrecht, A. P. (2004). Learning to play games using a pso-based competitive learning approach. *IEEE Transactions on Evolutionary Computation*, *8*(3), 280-288.

Parsopoulos, K. E., & Vrahatis, M. N. (2002). Particle swarm optimization method in multiobjective problems. *Proceedings of the 2002 ACM symposium on Applied*

computing, 603–607.

Pham, D., Ghanbarzadeh, A., Koc, E., Otri, S., Rahim, S., & Zaidi, M. (2005). The bees algorithm. *Technical Note, Manufacturing Engineering Centre, Cardiff University, UK*.

Piotrowski, A. P., Napiorkowski, J. J., & Piotrowska, A. E. (2020). Population size in particle swarm optimization. *Swarm and Evolutionary Computation*, 100718.

Reyes-Sierra, M., & Coello, C. C. (2006). Multi-objective particle swarm optimizers: A survey of the state-of-the-art. *International journal of computational intelligence research*, *2*(3), 287–308.

Rheingold, H. (2002). *Smart mobs: The next social revolution*. Basic books.

Russell, S., & Norvig, P. (2002). Artificial intelligence: a modern approach.

Ryan, K. J. (2018, Jun). *This startup correctly predicted the oscars, world series, and super bowl. here's what it's doing next.* Retrieved June 7, 2020, from https://www.inc.com/kevin-j-ryan/unanimous-ai-swarm-intelligence-makes-startlingly-accurate-predictions.html

Samuel, A. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*(3), 210–29.

Shi, Y., & Eberhart, R. C. (1998). Parameter selection in particle swarm optimization. *International conference on evolutionary programming*, 591–600.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... others (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Weizenbaum, J., et al. (1966). Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *9*(1), 36–45.

Zitzler, E. (1999). Evolutionary algorithms for multiobjective optimization: methods and applications. *PhD Thesis, Shaker Verlag*.

Appendix A

Repository of Code and Logs

The following GitHub repository link provides the complete codebase, the logs of all tests, accompanying videos, and a README establishing modes of execution of the software:

www.github.com/mcguile/thesis