USING INFLUENCE MAPS WITH HEURISTIC SEARCH TO CRAFT SNEAK-ATTACKS IN STARCRAFT

by © Lucas Critch A Thesis submitted

to the School of Graduate Studies in partial fulfillment of the

requirements for the degree of

Master of Science, Department of Computer Science, Faculty of Science

Memorial University of Newfoundland

September, 2021

St. John's Newfoundland and Labrador

Abstract

Real-Time Strategy (RTS) games have consistently been popular among AI researchers over the past couple of decades due to their complexity and difficulty to play for both humans and AI. A popular strategy in RTS games is a "Sneak-Attack," where one player tries to maneuver some of their units into the base of their enemy without being seen for as long as possible to surprise their enemy and deal massive damage to their economy. This thesis introduces a novel method for finding Sneak-Attack paths in StarCraft: Brood War by combining influence maps with heuristic search. The combined system creates paths that can guide units effectively - and automatically - into the enemy's base, by avoiding enemy unit vision and minimizing both travel distance and unit damage. For StarCraft, this involves guiding a loaded transport ship to the enemy's base to drop off units for attack. Our results show that the new system performs better than direct paths across a variety of maps in terms of total transport deaths, total damage taken, as well as the total time spent by the transport within enemy vision. We then utilize this new system to demonstrate an alternate use: a proof of concept for calculating building placements to defend against enemy sneak-attacks.

Acknowledgments

This thesis was created under the supervision and guidance of Dr. David Churchill, who has been an extraordinary help and mentor to me throughout all facets of this research experience. I would also like to thank my parents for supporting and encouraging me to pursue my interests. And lastly, my partner, Kelsi, whose patience and support has meant the world to me.

Thank you to you all.

Contents

List of Tables

List of Figures

List of Abbreviations

2D 2 dimensional. 10

AI artificial intelligence. 1

BWAPI Brood War Application Programming Interface. 31

EA evolutionary algorithm. 21

PF potential field. 11

RTS real-time strategy. 1

SC StarCraft. 1

SC: BW StarCraft: Brood War. 1

Chapter 1

Introduction

Since they were first brought to consumers in the early 1990s, real-time strategy (RTS) video games have been well known as one of the most complex video game genres. Using an interesting combination of player strategy, game knowledge, and quick reflexes, these types of games are extremely captivating to watch and play. RTS games have also been consistently popular among artificial intelligence (AI) researchers over the past couple of decades due to the aforementioned complexity of play. StarCraft: Brood War (SC: BW) in particular is the game of choice for AI researchers, with examples such as Deepmind's AlphaStar [37], and a multitude of StarCraft (SC) AI tournaments such as AIIDE and SSCAIT [3], [15], [12].

A popular strategy in RTS games is a "Sneak-Attack", where one player tries to maneuver some of their units into the base of their enemy without being seen for as long as possible, in order to surprise their enemy and do as much damage as they can. In this thesis, we will discuss in detail the specifics of RTS games, how they are related to the AI field, and what a "Sneak-Attack" is and why they are important. We also will describe influence maps and heuristic search and their relation to AI and RTS games. Finally, we discuss an AI system used to perform sneak-attacks in a game of StarCraft using influence maps and A*, along with all of the background, experimental details, and results that go with it.

Our novel system creates paths that can guide units effectively - and automatically - into the enemy's base, by avoiding enemy unit vision and minimizing both travel distance and unit damage. For StarCraft, this involves guiding a loaded transport ship to the enemy's base to drop off units for attack. Our experimental results show improvements across a variety of maps in total transport deaths, remaining health after transporting, when the transport was first seen, and the total of time the transport was seen by the enemy.

We also implement a system designed to defend against our sneak-attack system. This system uses generated sneak-attack paths to find optimal building placements that maximize the path length that an enemy transport would take to reach our base. What this aims to do is create a defense against sneak-attacks, where there are little to no areas that the enemy can sneak into the base undetected - allowing the player time to react appropriately.

1.1 Motivation

AI navigation remains a popular topic for researchers and game developers alike. There are a variety of pathfinding algorithms used for AI units to traverse their environments, each with their own pros and cons [19].

Video games are often criticized by players for either being too easy or too difficult. A portion of this criticism is sometimes related to the AI in the game. RTS game AI is often implemented using scripted techniques that can easily be countered by players; or on the other hand, some RTS AI are created with the ability to cheat in the game, creating unfair advantages against the player. The point being: players want a fair AI that provides an acceptable amount of challenge.

Allowing game developers more options for their AI navigation would beneficial, as A* is the standard for most pathfinding in video games. By applying the discussed systems to an AI in video games, it will allow developers to have flexibility in pathfinding for the agents

they create.

This system could also be applied to other domains, such as robotics navigation, general obstacle avoidance, or in other video game scenarios.By having influence come from objects of interest in the domain space, and using a similar custom Cost function, avoidance pathfinding behaviors can be created.

1.2 Problem Statement

The main objective of this thesis is to create and evaluate a sneak-attack system in StarCraft that combines influence maps and heuristic search. We hope that it acts similarly to what a human player would do when performing a sneak-attack in similar scenarios, and performs better than a standard shortest path.

1.3 Thesis Outline

Chapter 2 will discuss RTS games in more detail, as well as some relevant RTS AI research. The RTS genre is broken up into many sub-categories that define what exactly an RTS game is. RTS AI research has over time dictated several sub-problems within RTS AI, and so each of these sub-problems will be explained. Also, chapter 2 touches on backgrounds of topics more related to our experiment; namely what a "sneak-attack" is in an RTS game, the history and explanations of influence maps and A*. Chapter 3 is a short literature review of some the work related to the contents of this thesis. We will go into detail on work related to StarCraft and RTS AI, RTS pathfinding, and various uses of influence maps in RTS games. Chapter 4 describes the methodology of our influence map-search system; from each individual piece - such as the influence maps and A* - to the combination of the systems and how they interact. Chapter 5 explains the sneak-attack defense algorithm; why it is done

and how it works. Chapter 6 covers the experiments done with the sneak-attack and defense systems. We describe the StarCraft maps used, the games and settings used, comparisons made to A*, the data recorded, and how it was used. Chapter 7 is our conclusions, and short discussion on where this work could lead next.

Chapter 2

Background

2.1 Real-Time Strategy Games

Strategy games are a genre of game that emphasizes critical thinking, tactical decisions, and logistical challenges. Usually, the player has a bird's eye view of the game, and is given the role of a commander, who must up some economy (collect resources), decide where to build various buildings, decide units to create, and be able to control those units. The economy that is earned throughout the game is expended on creating units and buildings.

Real-time strategy (RTS) games are a sub-genre of strategy games. They are played in real-time, meaning there are no turns, like some other strategy games such as Civilization. This distinction may seem small, but it is not. More importantly, this means that players are acting simultaneously, and must think on their feet, and have quick reaction times in order to balance the multitude of objectives available to them at any given point in the game - or run the risk of falling behind their enemy. Contrasting RTS games with turn-based strategy games, in turn-based games each player has a set amount of time to do various objectives during each turn, and the game does not proceed until every player indicates they are ready.

In general, the main objective in an RTS is to destroy the opponent player's base in some

way. This is accomplished by building up your own base, accumulating buildings and units, all while attacking the enemy units and buildings to try to defeat them. There are countless strategies that players can employ to accomplish these actions, and the sheer number of strategies, buildings, and units is part of what makes RTS games so complex.

The three main types of strategies in RTS games are rushing, defending, and expanding. These strategies are detailed further in Subsection 2.1.3.

2.1.1 Properties of RTS games

There are some main components of RTS games that make them so interesting to AI researchers. The below terms are the main properties, and explanations of how they relate to RTS games.

- **Real-Time** means that games are played in real-time, meaning the game progresses even if a player is not taking actions for whatever reason. Players can do as many actions as possible within the time frame that the game is executed (usually between 24 and 60 frames per second). For example, a game that executes at 30 frames per second, will accept player input every 33.3ms. Comparatively, Chess is a turn-based game, where players take a turn, and then wait for their opponent to do a turn.

- **Multi-Unit Control** is the idea that players can control many units at a time, ranging from tens to hundreds, or even thousands in certain games. In StarCraft, players can control up to 200 units at a time. Every unit also has individual actions, so at any given point in the game, there are an exponential number of combinations of actions that a player can perform, relative to the number of units under their control.

- **Simultaneous Moves** means that each player can perform an action at the same time step of the game.

- **Imperfect Information** means that a player cannot see their enemy unless their own unit is nearby said enemy. Scouting is a term in RTS games which means controlling a unit in order to get close to the enemy to see what they are doing in order to plan a counter.

- **Non-Determinism**, or randomness. Some actions in RTS games are non-deterministic, like units in StarCraft may have a chance to miss their attack based on some criteria. For example, a ranged unit's attacks have lower chances to hit if their target is under certain objects like trees or a sign. This chance is similarly altered when the attacking unit has high ground and the target is on low ground. The chance for a ranged attack to hit the target if either of these example cases is true is 53.125% [2].

- **Complexity** is a characteristic of RTS games often discussed when compared to other video game genres. RTS games have huge state and action spaces, defined by the large number of actions that can be taken at a single time step, and the number of actions needed to reach the end of a game. As an example of this, Chess is known to have about 10^{50} possible states, while GO is estimated at 10^{170} states, whereas StarCraft has been shown to be about 10^{1685} [30].

2.1.2 RTS Games for AI Testing

There are many properties of RTS games that make them very appealing to AI researchers, as brought to attention by Buro in 2003 [10]. They are: (i) Real-time Planning, (ii) Collaboration, (iii) Uncertain Decision Making, (iv) Opponent Modeling, (v) Spatial and Temporal Reasoning, (vi) and Resource Management. As well, through research over the years, additional challenges have been identified [30], as such: (i) Uncertainty, (ii) Learning (Prior, In-game, Inter-game), (iii) Domain Knowledge Exploitation, (iv) and Task Decomposition.

The reason these properties are so important is because they are also the same properties

of the real world. Thus solving these problems in RTS games can help to solve the real world problems. Some examples include: robotic navigation, road traffic, military positioning, weather forecasting, and financing.

2.1.3 StarCraft

StarCraft is a popular real-time strategy game released by Blizzard Entertainment in March 1998, with its expansion pack, StarCraft: Brood War releasing later that same year in December. The game is based in science-fiction, with players choosing one of three playable races - Protoss (expensive, powerful unit types), Zerg (cheap, weaker unit types), or Terran (mix of expensive and cheap unit types). Each race - along with their associated units - has its own pros and cons, but overall these races are incredibly balanced against one another, meaning their win and loss rates are very similar against every match-up.

Over the years of balance patching and updates, StarCraft has become known to have very good game balance. Game balance is a concept use in game design that describes how fair and balanced a game is when given multiple strategic choices. Every unit can be countered by another, and for each strategy there is another that counters it. And as mentioned above, each race is balanced against the others. In high level competitive play, each race match-up has close to a 50% win and 50% loss ratio against each other race.

A video game with good game balance tends to be highly competitive, while also being entertaining to watch. StarCraft's impeccable game balance allowed it quickly become an extremely popular game among players and eSports alike around the world. Millions of copies were sold, and South Korea's formation of the Korean eSports association for professional StarCraft play in the early 2000s allowed StarCraft to explode in global popularity. StarCraft was by far the most popular eSport for many years, and paved the way for modern eSports and game competition today, with millions of dollars being awarded from tournaments every

year [1]. Its popularity and game balance helped StarCraft rise to become the de-facto testbed for AI research.

At a high level, there are 3 main strategies that players can perform in a game of StarCraft [2]. They are:

- **Rush**: strategy where a player very quickly amasses some attacking units, and rushes to the enemy to attempt to destroy them before they have time to build up defenses and economy. This strategy results in the player having little defenses and technology built up, and so is often referred to as an "All-in" strategy - if it fails, they will likely lose the game.

- **Defend**: Sometimes called turtling. Where a player spends most of their time and resources to build up very strong defenses, resulting in enemy attacking having little to no chance of succeeding, but often the player has very little attacking power. This strategy is commonly used to stall games when the player needs to catch up to their opponent.

- **Expand**: strategy that involves the player expanding to create multiple base locations. Expanding to multiple base locations gives the player access to more resources, and thus they can build more units and technology.

Despite the seemingly simple explanations of these strategies, each of them have a multitude of intricacies of decision making processes and reaction times.

2.1.4 Sneak-Attack Strategy

An effective way to catch an opponent offhand and gain an upper hand is to surprise them in some way, so that they are unprepared and cannot counter appropriately. One example of this is a sneak-attack. A sneak-attack is when a player's units approach the enemy base

a specific way in order to avoid being seen by the enemy for as long as possible, with the intention of attacking. By doing so, the player can surprise the enemy and proceed to quickly attack their base before they have a chance to prepare or react properly. One effective sneak-attack strategy that a player can execute is called a drop strategy, where the player constructs a flying unit known as a transport which carries attacking units over the environment and base defenses in order to unload them deep into the enemy base to attempt to destroy enemy worker units [30].

2.1.4.1 Transport

A transport is a flying unit in StarCraft that can carry a specific amount of other friendly units from one location to another. It has a max capacity (8) that is based on the specific unit it is trying to carry. Some units take up 1 capacity, while others can take 2 or more. Most commonly, the transport takes 4 units. Because transports are flying units, they can fly over buildings and environmental geometry such as walls. This helps with avoiding enemies as most units in the game can only travel on the ground, and thus transports can travel areas that ground units cannot.

2.1.5 Countering Sneak-Attacks

A player would want to defend against enemy sneak-attacks if possible, in order to protect their own base and resources, else they risk falling behind in the game or losing. One common way players defend against sneak-attacks is to place buildings in such a way that it is difficult for any units to sneak into the base without being seen. As buildings give vision, spreading buildings around a base maximizes vision to minimize the possibility an enemy unit can sneak into the base.

2.2 Influence Maps

Influence maps were created from work done by Zobrist on spatial reasoning in the game GO [39], and have continued being used for spatial problems since [35]. They have also been used in some retail video games, like Age of Empires 2 [33], as well as AI research with games [38]. Influence maps are data structures that are used for computing and storing influence within a given environment. Influence values are typically used to either attract or repel from a given region within an environment, based on the intuitive notion behind what those values represent. For example, an enemy may be given a repulsive value as the player wants to avoid them, while a weapon power-up may be given an attractive value, as they want to increase their damage.

Influence maps are often implemented as 2 dimensional (2D) grids/arrays, with each grid position storing the influence value located at the equivalent environmental space. Grid representations are common since they can be thought of as an overlapping layer on a 2D environment for ease of use. A given point of interest will be assigned an influence value based on some influential object in the environment, and that value will radiate outward in the field based on the user-defined influence equation of the field. An example equation can be seen in Equation 4.1, and an example of influence radiating from a unit can be seen in Figure 2.1. The values decrease as they expand from the origin point, and can continue decreasing to zero, or until they reach some threshold - such as a distance from origin or a specific value.

There is no one specific implementation on an influence map, as it relies greatly on the specific tactical needs as well as the design of the environment (2D, 3D, grid, hex, etc). Overlapping influence values can be summed, or combined together in another way to represent a more influential area.

These maps allow computers access to spatial and tactical information of their given

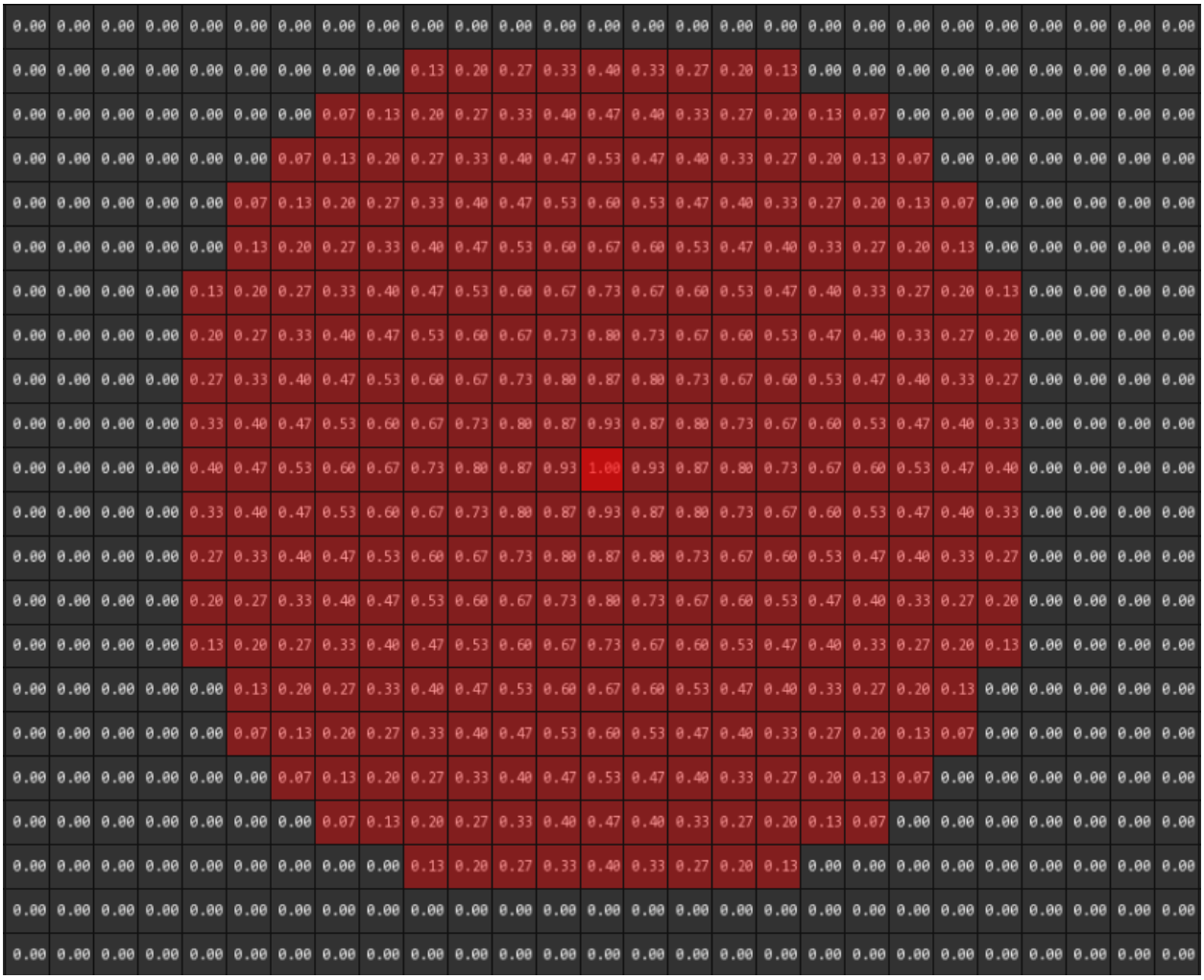


Figure 2.1: Influence map showing a single enemy's vision radius

environment. In our 2D grid example, an influence map could allow for information such as enemy locations and vision, path data, item pickups, etc. Using influence map data, an agent can maneuver in the environment by moving towards or away from positive or negative areas respectively.

2.2.1 Potential Fields

Similarly to an influence map, a potential field (PF) is used for navigation [26] [21] [25]. The main difference between influence maps and potential fields is that influence maps are

scalar fields of influence, and potential fields are vector fields of force. Potential fields give interesting objects in the world positive or negative charges, and the field uses those charges to navigate.

2.3 Grid Representation

StarCraft uses a 2D grid representation, and so it makes perfect sense to use a grid for our representations as well. The real world has arbitrary directional movement, but in video games this movement is often abstracted to be specific directions, commonly in a 2D grid to represent movement along the horizontal and vertical axes. Another similar representation is a graph structure, but we will be focused on grids. Grids have the benefit of being simple to visualize and implement.

A grid representation divides the world into equally sized cells. A simple example of a grid structure is shown in Figure 2.2. Then, directional movement and actions are restricted to adjacent cells instead of arbitrary directions like the real world, greatly simplifying movement problems by restricting the movement we need to consider. Figure 2.3 shows possible movement directions in a 2D grid, along with associated costs that will be explained below in Section 2.4. All movement and pathfinding problems are then computed on this grid instead of the actual problem space. Despite computing movement on the grid, actual entity movement still uses the world, so translating from grid to world positions becomes a necessity. Because of these properties, grid are very often used for pathfinding, which is explained in detail in the following Section.

Figure 2.2: 2D grid positions example, where the representation is (x, y)

2.4 Heuristic Search

2.4.1 A*

A* is a popular search best-first search algorithm that attempts to minimize a given cost function by expanding nodes on a graph in a priority queue until either it finds a goal, or

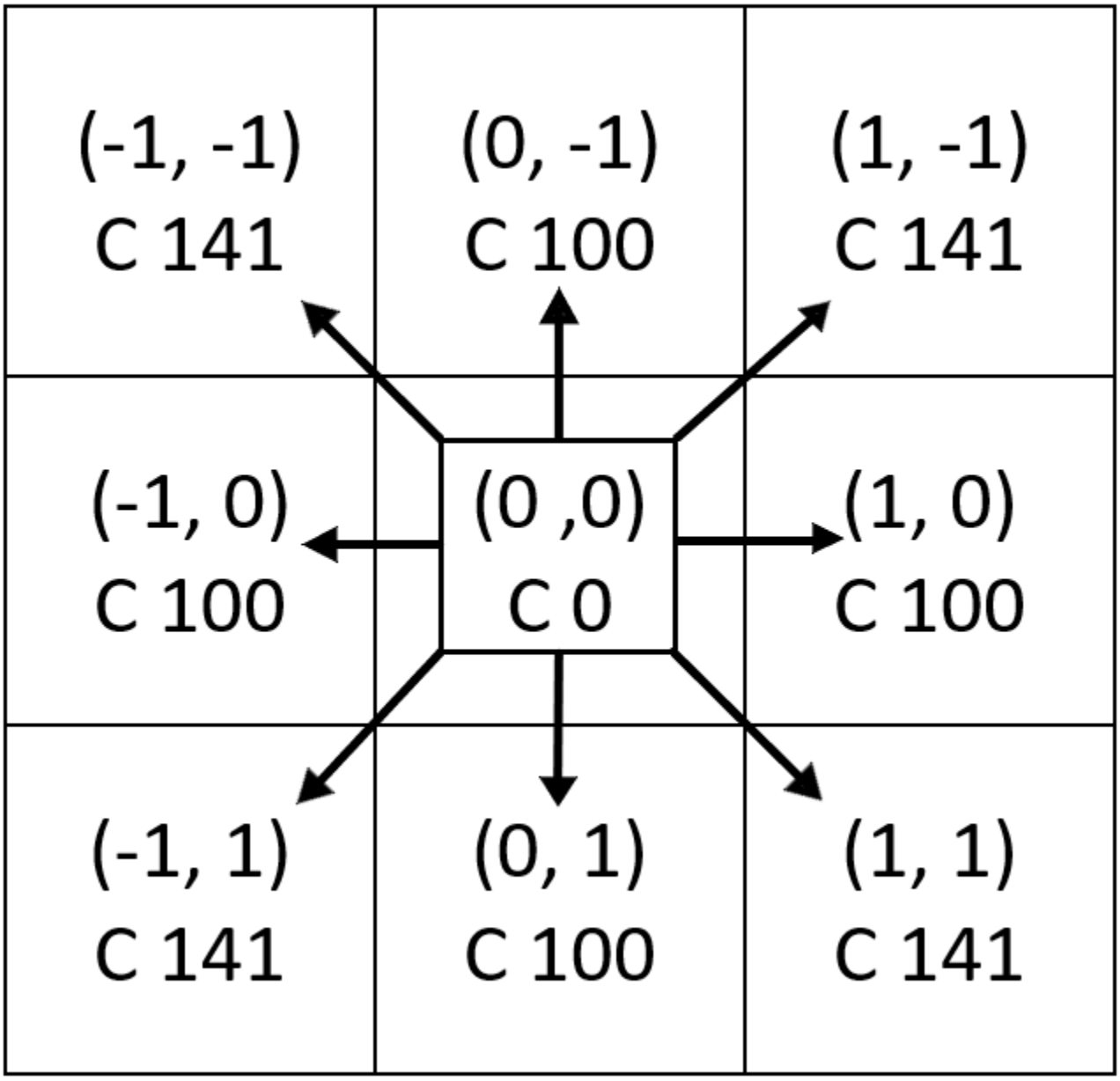


Figure 2.3: Directions and associated costs example

the graph has been entirely searched and no solution found. This priority queue is sorted as a function of search node n, denoted $f(n)$, with:

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

where $g(n)$ is the sum of costs $c(n)$ of the path so far to node n and $h(n)$ is the heuristic estimate of the path cost from the current node to the goal node [23]. Hart et al. defined 3 properties of A*:

- If there exists a path, A* will find it
- A* is optimal if $h(n)$ is admissible
- A* expands fewest possible nodes with its given heuristic $h(n)$

The algorithm in pseudocode is detailed in Algorithm 1.

If the cost $c(n)$ is a distance function, then A* will find the shortest path between two locations. The cost function can be modified to minimize other useful quantities such as fuel costs, time to goal, or frames in vision. The A* algorithm is widely used in industry RTS games [30], and its use of a cost-based heuristic function pairs perfectly with our intended integration with influence maps. Thus, if we want to minimize another cost function such as minimizing enemy vision along a path, we must use a more custom cost function. A* was first made to be 4-directional, but has since been altered to be capable of 8-directional traversal, as seen in Figure 2.4. As well, example costs for grid-based movement can be in Figure 2.3.

2.4.2 Heuristics

An admissible heuristic is a heuristic that never overestimates its distance to the goal node. An optimal heuristic is a heuristic where the resulting solution has the lowest path cost of all possible paths.

In the context of A* pathfinding, a heuristic is an estimation of distance between two points in space. More specifically, heuristics are used in pathfinding algorithms to guess the distance from a given node to a goal node, in order to measure its perceived 'closeness' to

Algorithm 1 A* Algorithm

```
1: openList ← priorityQueue(f(n) ← g(n) + h(n))
2: closedList ← list of explored nodes
3: add starting node to openList
4: while openList is not empty do
5:     currentNode ← minimum f value node from openList
6:     if currentNode is goal node then
7:         create path by following parent nodes backward
8:         return
9:     add currentNode to closedList
10:    for nodes adjacent to currentNode do
11:        if adjacent node in closedList then
12:            continue
13:        if adjacentNode f value is less than currentNode.parent f value then
14:            currentNode.parent ← adjacentNode
15:        add adjacentNode to openList
16: return
```

the goal. Heuristics are used to guide the search toward a goal, and to ideally result in less node expansions than other methods. Three commonly used heuristics [31] are described below:

2.4.2.1 Manhattan Distance

Manhattan distance is commonly used when dealing with 4-directional (quartile) traversal, meaning movement can be horizontal or vertical only. If using this heuristic while diagonal movement is possible, it can no longer guarantee shortest paths, as it will sometimes overestimate. The equation for Manhattan distance is:

$$h(n) = abs(node.x - goal.x) + abs(node.y - goal.y) \tag{2.2}$$

2.4.2.2 Diagonal Distance

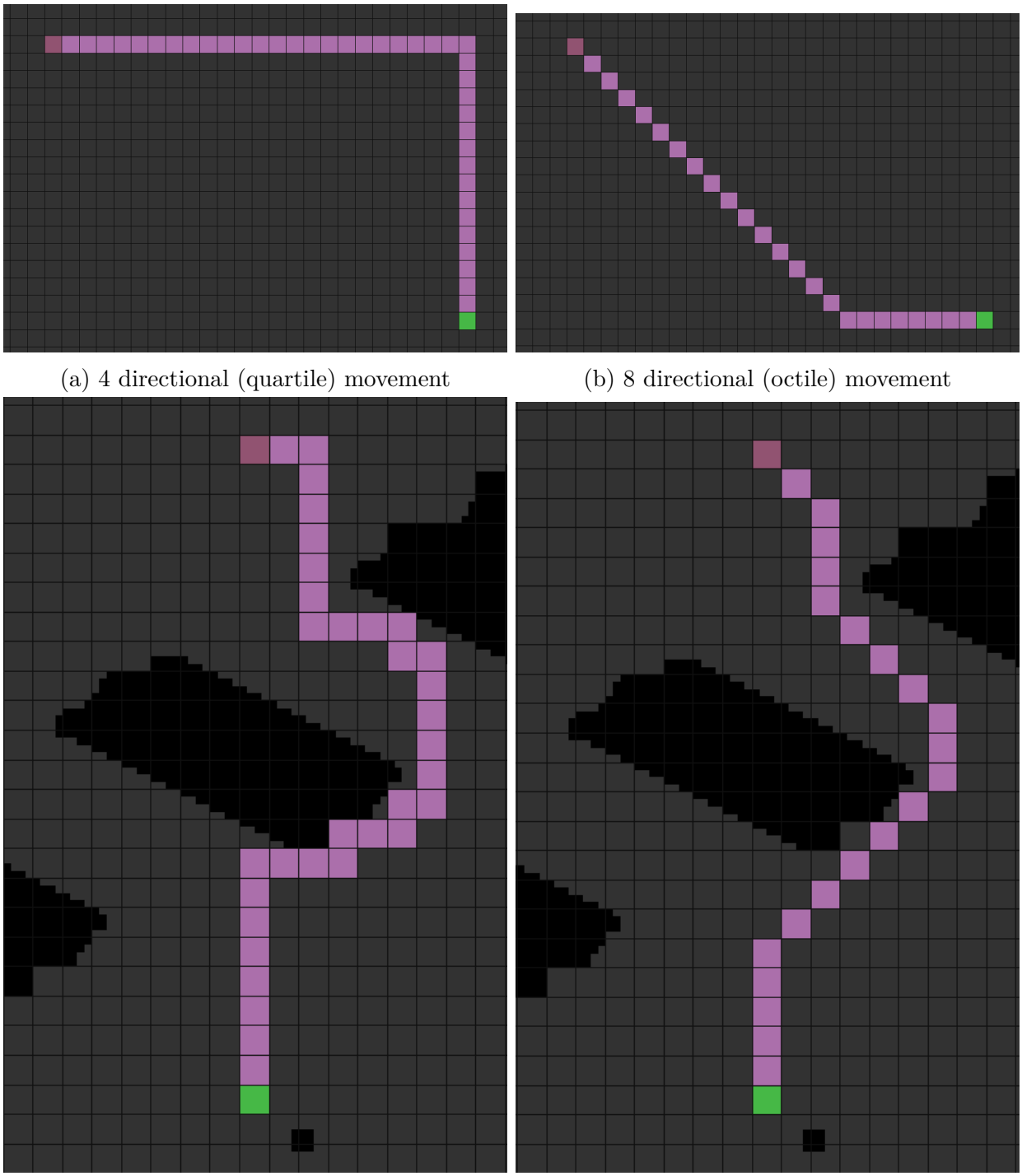
This heuristic is sometimes seen as an 8-directional (octile) Manhattan Distance because of its similarities, however it incorporates diagonal movements in its calculation. This heuristic should be used when diagonal movements are possible. Generally, diagonal moves have a higher cost than horizontal or vertical moves, based on the equation below:

$$\begin{aligned} h\_diagonal &= min(abs(node.x - goal.x), abs(node.y - goal.y)) \\ h\_straight &= abs(node.x - goal.x) + abs(node.y - goal.y) \\ h(n) &= sqrt(2) * h\_diagonal + (h\_straight - 2 * h\_diagonal) \end{aligned} \tag{2.3}$$

2.4.2.3 Euclidean Distance

Euclidean distance is simply a straight line between two nodes - the shortest possible path.

$$\begin{aligned} dx &= abs(node.x - goal.x) \\ dy &= abs(node.y - goal.y) \\ h(n) &= \sqrt{dx^2 + dy^2} \end{aligned} \tag{2.4}$$



(a) 4 directional (quartile) movement

(b) 8 directional (octile) movement

(c) Quartile movement around an obstacle

(d) Octile movement around an obstacle

Figure 2.4: Example of 4-directional versus 8-directional movement heuristics

Chapter 3

Related Work

The main objective of this thesis is creating and evaluating a search-influence map combination system in StarCraft that performs better than direct paths, and similarly to a human player performing a sneak-attack in StarCraft. As such, we will touch on some works relevant to the topics of search, influence maps, and potential fields (due to their similarities to influence maps), all in regards to StarCraft, or other RTS environments.

Martell and Sandberg [29] found A* to be the most performant compared to other pathfinding algorithms when starting with no knowledge of the environment. They compared it against Theta*, HPA*, IDA*, and Dijkstra. Erdtman and Fylling [18] did similar work, aiming to find a pathfinding solution for RTS games designed for systems with limited processing and storage.

Influence maps were used by Bergsma and Spronck [8] on their work adaptive spatial reasoning in turn-based strategy games. They used machine learning for training AI combat behaviour, and influence maps to represent the state of their game, such as attack or move. They layered multiple influence maps using a neural network. Their AI was able to learn and counter tactics and perform equally, or better, than AI using static strategies. They also showed that influence maps could be used to directly influence AI behaviour. Machine

learning - specifically Q-learning - was also used by Liu and Li [27] along with potential fields for multi-agent cooperative learning for a defensive strategy in the 2D simulation game RoboCup.

Hagelbäck [20] wrote about potential field navigation in StarCraft in 2012. They combined A* with potential fields - using standard A* until an enemy unit was nearby, and then switched to using potential fields. This allowed units to avoid or attack enemies depending on whether the unit could attack or not, and the state (attack or defend). This algorithm showed a large improvement over StarCraft's built-in pathfinding AI.

Similarly, potential fields were used for staying at a maximum shooting distance (MSD) in ORTS [22] [9]. Their work has some ideas about maintaining distance from enemies by using fields that may prove useful.

In 2014, Adaixo and Gomes [11] discussed Influence Map-Based Pathfinding Algorithms in Video Games. Specifically, their section on A* with influence maps proposed adding the influence value to the A* equation that corresponds to the current node n. They showed that their algorithm was an improvement over A* in terms of memory consumption and search times, with more or less computation needed depending on how often the influence maps need to be recomputed.

Related to A* with influence maps, Danielsiek et al. [17] used flocking with influence maps and A* to compare unit movement using flocking, and using IMs or not. They found that using both combined resulted in the safest navigation, with units losing less health.

Cui and Shi [16] evaluated the A* algorithm in detail, along with game world representations for A*. They discuss some A* optimizations such as HPA* (Hierarchical Pathfinding A*) - a technique for speeding up pathfinding - and NavMesh - a set of polygons that define traversable terrain in a 3D environment. A NavMesh can allow for near optimal paths in much less data. They conclude that A* is very popular in game AI and it is hard to find a better algorithm for pathfinding.

Xtrek, an algorithm that combines influence maps with normal pathfinding by Amador and Gomes [4] led to an influence aware pathfinder that can avoid or converge to desired areas of the search space during path generation. It combines traditional A* with an influence map with negative values for attractors and positive values for respulsors. When close to an attractor, the $g(n)$ part of A* equation 2.1 is not used in order to assure convergence to the center of the attractor. Otherwise, the algorithm works intuitively, combining the influences with the cost portion of A*, resulting in avoidance behaviors on repulsors, and normal A* behavior the rest of the time. Xtrek results in less memory and time on its pathfinding calculations.

Uriarte and Ontañón [36] successfully controlled multiple units at a time in StarCraft using influence maps. Specifically, they used influence maps to create a kiting behavior in the controlled units. They incorporated this technique into the NOVA bot in the 2010 StarCraft AI Competition. The maps were used to keep track of dangerous areas, and when attacking with a unit, would attack and then retreat to a safe location using the influence maps.

Pentikäinen and Sahlbom [32] used influence maps and potential fields for navigation in StarCraft 2. The influence maps were used for unit navigation, with the potential fields acting as repulsive forces during the navigation. The algorithm could be altered to disregard the influence and just use the potential fields. Tests were done on combat, navigation, and enemy avoidance. They concluded that their algorithm IM+PF outperforms A* in performance and scalability in some scenarios, especially when path alteration may be required.

In 2009, Avery et al. [7] used an evolutionary algorithm (EA) that evolves a set of influence maps that specifies a unit's spatial objectives in a naval RTS game of capture the flag. The tactics displayed in their research were where individual units should move, and who to attack. The evolution was evaluated based on which team captured the flag, if time ran out, or if all friendly players were destroyed. Each unit had its own IM initialized to zeroes. The evolved map values were set to the locations of the units, with a radius of their

weapon range. The IMs were used in combination with A*, with the goal being the lowest IM value on the map. Their results showed they can evolve IMs to create adaptive, tactical and competent plans.

Smith et al. [34] continued from the above work, as well other work using co-evolved influence maps [5] [6]. They created a co-evolutionary algorithm to generate spatially oriented tactics to see if players could learn better when playing against co-evolved opponents or hard-coded opponents. Influence maps were used to designate points of interest in their capture the flag game, whose weights were evolved over generations through the evolutionary algorithm.

All of this work helped to build into our current work, especially the work with influence maps. Some research has been done for navigation with influence maps and pathfinding algorithms, however that work would not exactly do what we propose. Nothing in the current work does anything with sneak-attacks or transport unit navigation, and so we found an area where we could contribute and created our own system that builds upon all of the great work mentioned in this section.

Chapter 4

Methodology

Before getting into the specifics of our system, we will first discuss the roles that both influence maps and A* play within it. We will then discuss how we integrated the system into StarCraft and how each piece ties together to create our sneak-attack system.

4.1 Influence Map Implementation

To use influence maps with StarCraft environment data, the influence maps will store influence values which will be used to implement behaviors relevant to our pathfinding task. Specifically, we will be using influence maps to compute areas of repulsion that we wish to avoid while computing sneak-attack paths, namely: around the vision of enemy units, range of enemy weapons, and within commonly traveled areas. The specific algorithms used for computing each influence map can be seen in their respective sections in Algorithm 2. StarCraft maps are represented as 2D grids, with a typical map being approximately 128x128 game tiles in size, with each tile being 32x32 game pixels. Because of the map 2D grid representation, we can easily construct influence maps as 2D arrays using the same StarCraft tile map dimensions, and have it be easily readable and comparable to the in-game map.

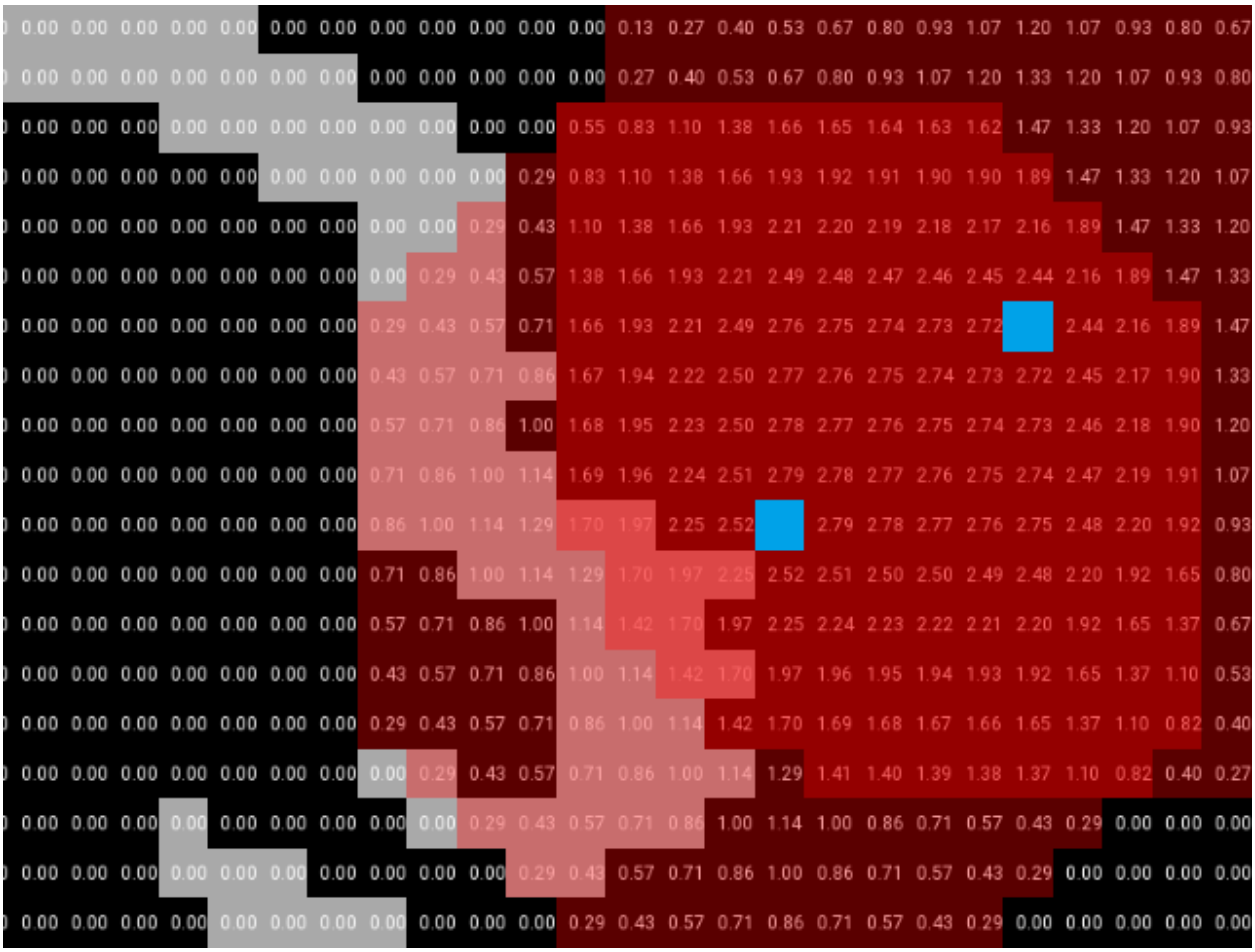


Figure 4.1: Example influence map showing 2 enemy units (blue), their influence radius (red), and their overlapping influence areas (brighter red). The gray sections are walls.

Each cell in the influence map grid will represent one build-tile cell in the StarCraft map at the same (x, y) location. This representation also benefits from easily being able to fit into the memory of most modern computers, with each influence map taking up about 0.2MB or less of memory for even the largest maps. Each cell of influence map grid stores an influence value as an 8 byte double, so for a 128x128 map we have 131072 bytes, or 0.13MB. Enemy units will be points of interest, with influence radiating up to a hard cutoff at the edge of their vision/attack radius. Figure 4.1 shows an example of such construction: showing two enemy units radiating their influence based on their vision radius. We will then use these values as part of the cost function for the A* algorithm so that we minimize the amount of time spent within these areas of vision.

To create an effective influence map for completely avoiding enemy vision as long as possible, we combined the effects of three separate influence maps, each for a different influence metric. These maps are as follows:

- **Vision Map**: Stores influence of visible and last seen enemy unit vision radius. This map helps to avoid areas that are seen by the enemy. Influence values radiate outwards from the enemy unit's position, and stop at the edge of its vision radius, which is determined by the StarCraft game engine.

- **Damage Map**: Stores influence based on enemy unit damage potential. Influence values radiate outwards from enemy position, deceasing based on distance from enemy, and stop at edge of weapon/damage range. Since we must eventually enter enemy vision to attack, we prefer to travel the areas that deal the least amount of damage.

- **Common Path Map**: Stores influence values based on the shortest ground paths between the player's starting base location and every possible enemy base starting location, which are often travelled by enemy units / scouts when moving to or from

bases. By avoiding these commonly traversed paths, we further decrease the chances of being seen by the enemy.

Individual influence values are calculated from the following equation presented in [28]:

$$m_{\mathrm{x,y}} = p - (p * (\frac{dist}{d}))^4 \tag{4.1}$$

where $m_{\mathrm{x,y}}$ is the influence at position (x, y) of map m, p is the max propagation value, and d is the maximum distance away from the center of radius v. An example of influence calculated by this formula, as well as showing each influence map, can be seen in Figure 4.2.

In more detail, each influence map is calculated using the following information:

- The common path influence map is created at the beginning of each game, initialized to zeroes. Both the player and enemy bases are found using UAlbertaBot, and a ground path is calculated from each possible enemy base to the player base. Influence is then calculated at each position of these paths if as there were an enemy at the cell location. In other words, vision influence radiates outwards from each position of these newly created paths. Influence is not overlapped here, as we are not simulating one enemy at each position, we are just creating a sort of barrier around this path in order to give it a repulsive influence value.

- Enemy vision influence map requires recalculation every frame, because if there are many enemies on screen, their vision will be moving with them as they all move around. If enemies are seen in UalbertaBot's vision area, influence values propagate starting at each enemy grid position, moving outwards with a hard cutoff at the edge of their vision radius. This map does have overlapping values, as locations that multiple enemies can see are likely to be more dangerous, and thus should be avoided with more tenacity.

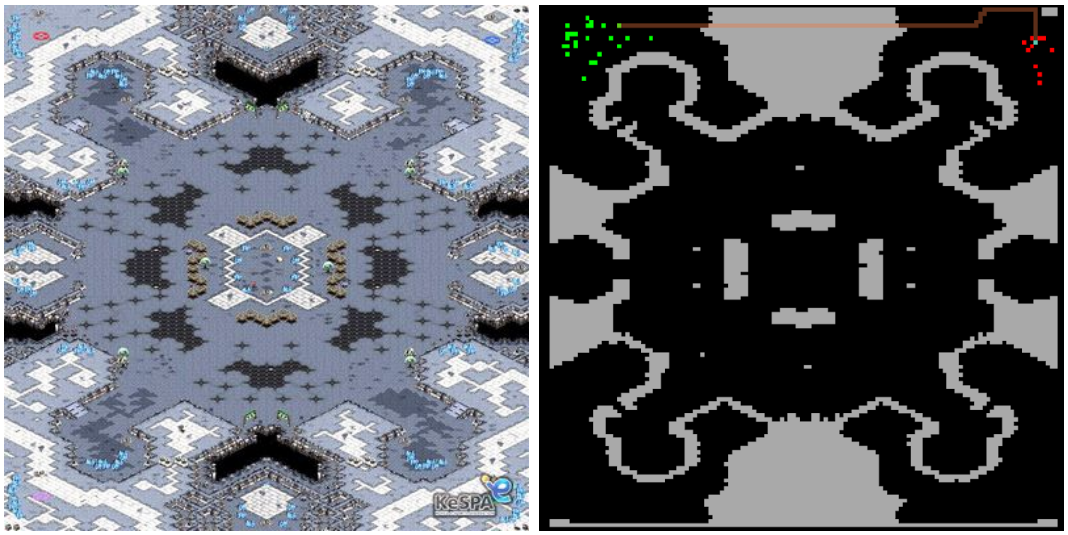
- Similarly to the vision map, the enemy weapon damage influence map does almost the same thing - with two key differences.
  - It uses enemy air weapon range instead of enemy vision radius. Some enemies carry weapons, and some can only attack units on the ground, while other can attack both air and ground units. We only care about the air weapon damage since our plan is to use a flying transport unit. This is easily modified to accommodate other weapon types.
  - The influence values are added with a static weapon damage number, based on the weapon of each enemy's specific weapon. This is done because these weapon range influence areas are much more deadly, so this is done to further repulse from these areas. As well, since damage is not affected by range, these values should represent the damage a unit would take when in the area.

Then, with these influence maps, a path from the player's base to the enemy base is calculated using A*. The maps are used in the cost function of A* in order to guide the path creation to avoid the influence areas as much as possible, as discussed in Section 4.2.

4.2 Influence Maps as Cost in A*

Our implementation combines multiple influence maps to be used to guide A* as the cost and heuristic function. At a given node n, the custom cost function $c(n)$ will now consider the following values:

- $dist$ = cost of traveling a given distance on the map
- vis = influence cost of entering enemy unit vision
- dam = influence cost of entering enemy unit damage area



(a) Map terrain view

(b) Map with no influence fields visible

(c) Map with vision field visible

(d) Map with all influence fields visible

Figure 4.2: Three views of the StarCraft map "Andromeda" with different influence maps visible. (a) shows the terrain view of an example 4-player StarCraft map: Andromeda, in which potential base locations can be seen in the four corners of the map. (b) shows our custom visualization tool representation of the same map during a game, with two bases occupied by units for our player (green) and the enemy (red).

Algorithm 2 Influence Calculations for vision, damage, and path influence maps.
A single unit influence calculation run-time is O(w * h) where w is map width and h is map height.
Vision & damage map calculation run-times are O(w * h * u) where u is # of enemy units.
Common path map calculation run-time is O(w * h * b * p) where b is # of possible enemy bases and p is path length.

$(sx, sy) \leftarrow$ size of StarCraft map
$visionMap[][] \leftarrow$ zeros(sx, sy)
$damageMap[][] \leftarrow$ zeros(sx, sy)
$pathMap[][] \leftarrow$ zeros(sx, sy)
$p \leftarrow 1.0$
$r \leftarrow$ StarCraft unit vision radius
$d \leftarrow$ max distance away from center of r (is $u_{\text{x,y}}$)
$pBase \leftarrow$ Player base
// calculate vision and damage influence maps
for StarCraft unit $u \in$ enemyUnits **do**
    $r \leftarrow u$'s vision radius
    $d \leftarrow$ max distance away from center of r (is $u_{\text{x,y}}$)
    for (x, y) from $(0, 0)$ to (sx, sy) **do**
        $dist \leftarrow$ distance from (x, y) to $u_{\text{x,y}}$
        if $dist < r$ **then**
            $i \leftarrow p - (p \times \frac{dist}{d})^4$ // influence
            $visionMap[x][y]$ += i
            $damageMap[x][y] \leftarrow p$
// calculate common path influence map
for *base* not $pBase$ **do**
    $path \leftarrow$ path from *base* to $pBase$
    for $pos_{\text{x,y}}$ in $path$ **do**
        for (x, y) from $(0, 0)$ to (sx, sy) **do**
            $dist \leftarrow$ distance from (x, y) to $pos_{\text{x,y}}$
            if $dist < r$ **then**
                $i \leftarrow p - (p \times \frac{dist}{d})^4$ // influence
                $pathsMap[x][y]$ += i

- $path$ = influence of entering a common path tile

The cost is computed as a tuple, $c(n) = \{vis, dam, path, dist\}$, which is sorted based on the following priority, in order of highest amount of repulsive influence: $vis > dam > path > dist$. Intuitively, this means that we will prioritize avoiding enemy vision first, avoiding enemy

damage next, then avoiding common paths, followed finally by attempting to minimize the total distance to the enemy base. When we use this new cost function within the A* search algorithm, its priority queue will sort nodes based $f(n) = g(n) + h(n)$, with $g(n)$ being equal to the sum of node costs to that node so far in the search tree. A simple example showing influence avoidance can be seen in Figure 4.3.

4.3 Countering Sneak-Attacks

Now that we have a system in place for calculating sneak- attack paths, we can also use it for a defensive purpose: calculating the positions for placing buildings within our own base to prevent enemy sneak-attacks. As explained in Subsection 2.1.5, we use buildings to maximize the amount of vision in our base. This is precisely what our proposed solution to this sneak-attack defense problem is. If we place buildings in a way that creates the longest path created with the sneak-attack system, it gives the defending player more opportunity to see a sneak-attack coming. This is what our proposed algorithm below in Algorithm 3 does.

4.3.1 Building Information and Placement Details

Buildings in StarCraft come in a variety of sizes. Some buildings are just 1x1 in-game tiles, whereas others can be (but not limited to) 2x2 or 4x3, for example. Most bases have a central entrance area which serves as a choke point, so it is strategic to guard that area. Because of this strategic placement, we limit our building placements to inside the base's general choke point area. An example base showing valid building placements can be seen in Figure 4.4. Intuitively, buildings cannot overlap, and also cannot rotate, so for every building that is placed, we have to check that it is not overlapping anything, and that it is within our base boundaries.

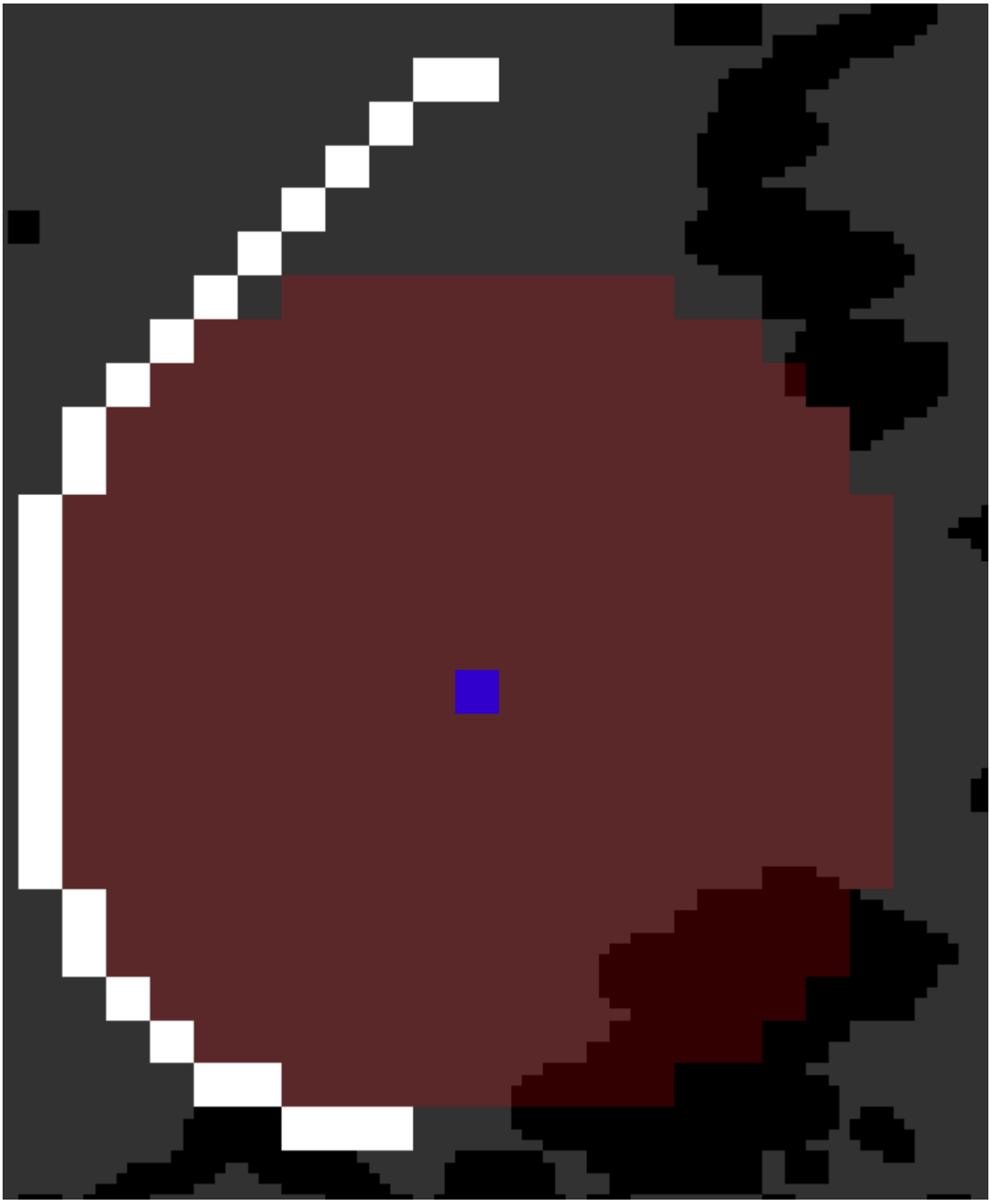


Figure 4.3: White path avoiding unit vision influence (red)

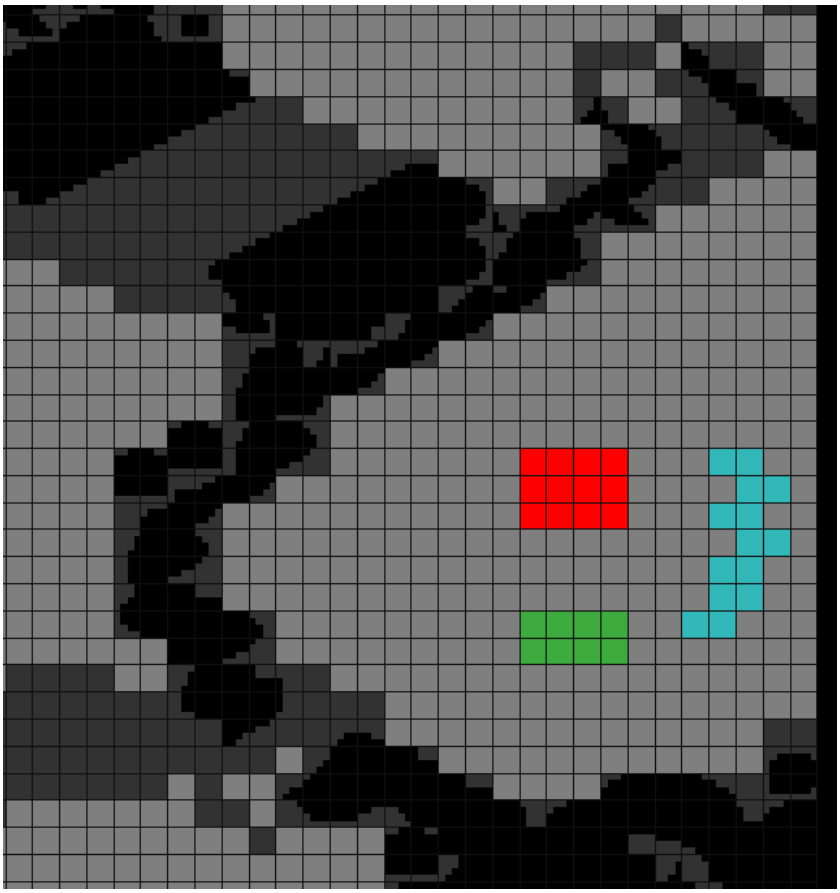


Figure 4.4: A typical base, centered around the red main building, with blue mineral line and green gas resources. The light gray area around this center zone is where buildings can possibly be placed.

4.3.2 Algorithm

The algorithm we chose to implement this was quite intuitively simple: iterate over all legal building positions for the first n buildings in our build order, and for each of those possible placements we run the sneak-attack path system to find the best path to our base from the enemy base. The building positioning that results in the longest possible path to our base is chosen as the position for our buildings. One advantage of this method is that it is an any-time algorithm, which can be given a time limit if desired and return the best solution found so far. Figure 4.5 shows different building placement results from different time limit cutoffs. We use a 'skip size' to designate the number of valid building positions to skip over on the current iteration. This results in checking less positions overall, but covering a greater area overall in the base in an allotted time. The reasoning for this is because we do not want to put every building right next to each other, so we can skip close positions. We want buildings to be reasonably spread out for more base coverage, so we skip over positions in the base iteration to cover more ground. The algorithm is detailed in Algorithm 3.

4.4 StarCraft Implementation

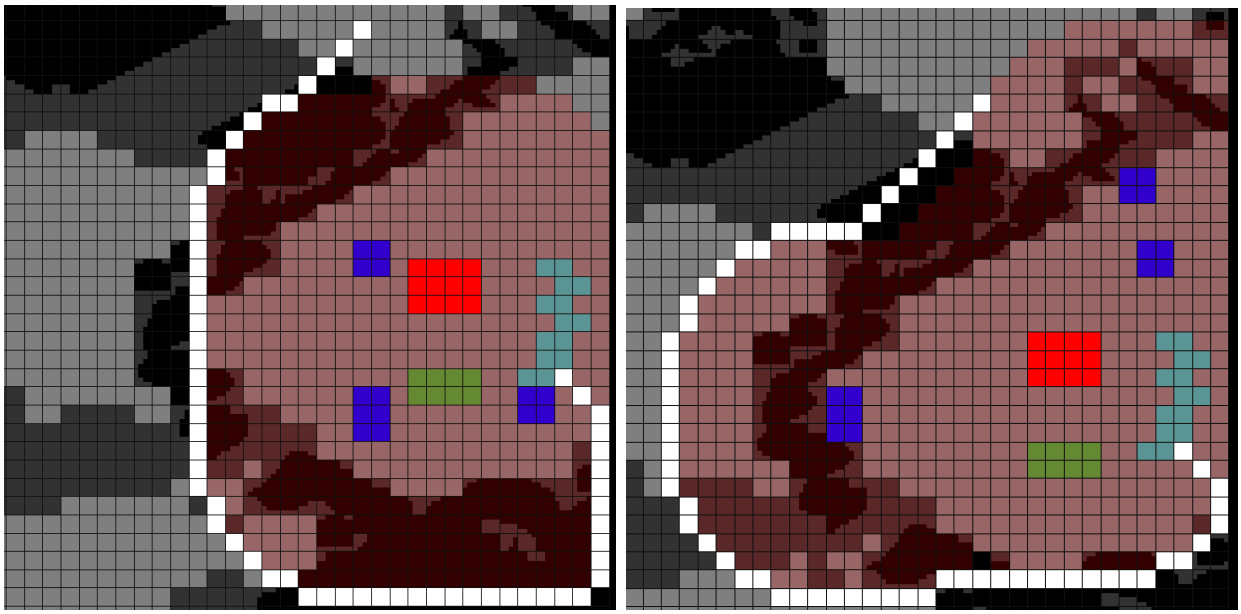
To utilize this new system in StarCraft, we required a way to interface with the StarCraft game engine. To do this, we used the Brood War Application Programming Interface (BWAPI) [24], which allows us full control over the StarCraft game engine to query game data and issue game commands. We also used UAlbertaBot [14], an existing StarCraft AI competition bot that comes with pre-built modular systems for playing the entire game, from gathering resources, to building an army and attacking. UAlbertaBot's modular architecture allows us to add a new strategic module to the bot which performs a drop-based sneak-attack strategy, for which the pathfinding is controlled by our new system, and the rest of the game mechanics are performed by the existing bot logic. We also use the StarCraft map visualizer

Algorithm 3 Recursive Building Placement.
For the recursive function 'recursivePlacement' the run-time is O($n!$ / (n - r)!) where n is number of building placement positions in base, and r is number of buildings to place. This function is called O(log(*m-skipValue*)) times, from the while loop.

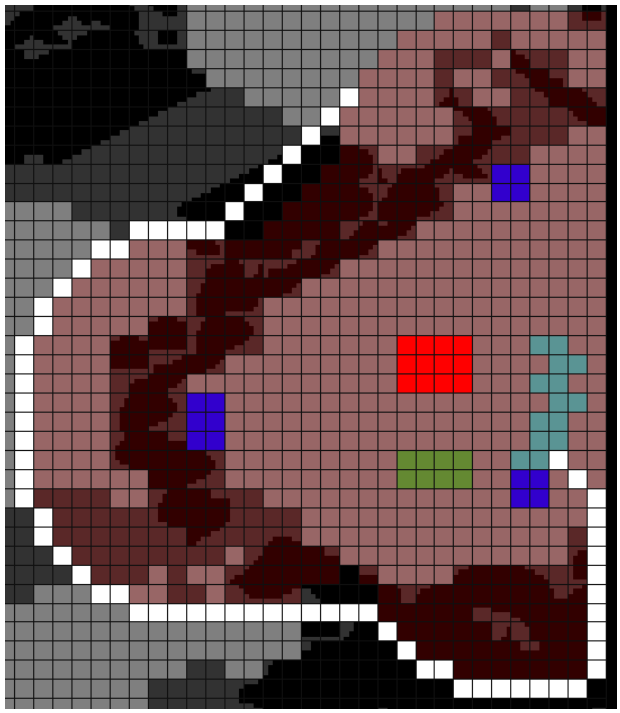
```
// Setup
m-bestBuildings ← []
m-currentBuildings ← []
m-buildingsToPlace ← config.buildings // buildings currently defined from config file
m-bestPath ← []
m-skipValue ← config.skip // starting size of tiles to skip in placement algorithm
while m-skipValue >= 1 do
    recursivePlacements(0, (0, 0))
    m-skipValue /= 2
// Building placement function
function RECURSIVEPLACEMENTS(buildingNum, prevPosition)
    if buildingNum > m-buildingsToPlace.size then
        Return
    building ← m-buildingsToPlace[buildingNum]
    for dy ← prevPosition.y; dy < baseHeight; dy += m-currentSkip do
        for dx ← prevPosition.x; dx < baseWidth; dx += m-currentSkip do
            position ← (dx, dy)
            if !inEnemyBase(position) OR !canPlaceBuilding(building, position) then
                continue
            addBuilding(building)
            if m-currentBuildings.size == config.buildings.size then
                for all building in m-currentBuildings do
                    doInfluence(building)
                search ← runSneakAttackSearch()
                path ← search.results.path
                if path.size > m-bestPath.size then
                    m-bestPath ← path
                    m-bestBuildings ← m-currentBuildings
            recursivePlacements(buildingNum+1, (dx+1,dy))
            removeBuilding(building)
```

tool StarDraft [13], which allows us to perform map analysis and pathfinding offline, makes for easier debugging, and was used to create most of the visualizations for this thesis.

This integrated system reads relevant influence map information from StarCraft every



(a) 100ms (b) 250ms



(c) 500ms

Figure 4.5: Building placements for two 2x2 buildings and one 2x3 building with varying time limits.

game frame and uses it to update the influence values. To calculate influence values from real StarCraft data, we use the following game information: obstructed game map tiles (walls, un-walkable terrain), walkable grid cells, and enemy unit information (vision radius, damage radius, location). Every frame of the game, the system takes new enemy information and updates the vision and damage maps. The common path map does not require updates, as the information stored is static for each map, so we can just compute it once at the start of every game.

The specific algorithms for each influence map can be seen in their sections in Algorithm 2, however we will also provide a more intuitive explanation. The common path influence map is created at the beginning of each game, initialized to zeroes. We then calculate the shortest paths using A* from the player's starting base to each other possible base location, and at each cell of the paths, vision influence is calculated as if an enemy unit was at that position. In other words, influence radiates outwards from each cell of the newly created path, creating a repulsive force away from commonly used paths and bottlenecks in the environment. Unlike common path influence, the enemy vision influence map requires recalculation every frame, since enemy units may constantly be moving around the map. Each frame, the last known position of each enemy unit is used as the center of influence, with repulsion radiating outward. This uses overlapping values, as locations that multiple enemies can see are likely to be more dangerous and should be avoided at all costs. The enemy damage influence map behaves the same as the vision influence map with two differences: it uses enemy weapon range instead of enemy vision radius, and the influence values are based on the damage of each enemy's specific weapon. The idea here is to create high values of repulsion around enemy units that can inflict the most damage.

Then, with these influence maps as a heuristic evaluation, a path from the player's base to the enemy base is calculated using A*. The maps are used in the cost function of A* in order to guide the path creation to avoid the influence areas as much as possible, as discussed

in Section 4.2. Finally, our system has several parameters which are considered, which allow it to perform reliably within a real-time strategy AI agent, which are as follows:

- Recalculation Time: How often the paths are re-calculated. For example, we may only need to recalculate paths once per second instead of once every game frame.
- Drop Distance: How close to the enemy's mineral line (our intended target) do we have to be in order to drop our units and begin the attack.
- Recalculation Distance: Once the player's transport has gotten sufficiently close to the enemy base, there are diminishing returns on recalculating the sneak-attack path. For example, if the transport is already within enemy vision and almost within the Drop Distance threshold, the movement of a single enemy unit may cause the influence to change and a new path to be calculated. If we follow this new path, we may end up spending too much time executing the drop, allowing the enemy to amass its defenses. Within a specific distance, we want the transport to execute the current path to completion, rather than continue to re-plan.

The above properties were tested experimentally, and we decided to use: a recalculation time of 1 second, a distance of 100 pixels as the drop distance, and a distance of 900 pixels as the recalculation distance. These values result in similar sneak-attack behaviour to expert human players in similar situations, based on observation from professional game replays. More specifically, when performing a sneak-attack expert human players navigate their sneaking unit in such a way to avoid enemy units as much as possible. When approaching their opponent's base, they enter in a low defense area, and target enemy mineral line units to disrupt economy production. They continue to monitor their sneaking unit to re-evaluate their path if necessary, and when they enter their enemy's base they continue the planned path and drop their units close to the target. This behaviour is what our system is observed doing when paired with these specific values.

The reasoning for using these specific values are as follows:

- Recalculation time of 1 second, since units are constantly moving in-game, our paths need to be updated very often in order to account for moving enemy units. Having a larger recalculation time, such as 5 or 10 seconds, would result in the paths guiding our unit into enemies, as the path does not update fast enough to account for new enemies appearing on screen. On the other hand, having a smaller recalculation time, such as 0.2 seconds, very little is changing in StarCraft in such a short amount of time. Additionally, a shorter time would result in far too many calculations - and because of that, slowdown.

- The drop distance was set to 100 pixels as that is about the size of a Command Center, and we know that once we are near the Command Center, we know the drop will be within the mineral line - the location we want to disrupt or attack. Setting this to a higher value will result in the drop happening from further away from the designated drop location. Setting to 1000 for example will drop around the entrance to the enemy base or before getting to the base, depending on the map.

- Recalculation distance was set to 900 pixels as that is roughly the size of the enemy base, so the path will not be recalculated once our transport enters the base. While in the enemy base, new units will usually be found, but we do not want to recalculate our path then, as 1) we are already in the base, and 2) the enemy already knows we are there. Setting a larger recalculation distance, such as 5000 pixels, would halt any path recalculations once we are 5000 pixels or closer to the enemy base.

4.5 Testing Environments

4.5.1 StarDraft

StarDraft[1] is a standalone tool used for StarCraft map visualization. It can load any StarCraft map and display it in a simple, easy to use user interface. Some example screenshots can be seen in Figure 4.2. StarDraft allows us to perform map analysis and pathfinding without having to launch the full StarCraft game client, makes for easier debugging, and was used to create most of the visualizations for this thesis.

StarDraft was used for prototyping our sneak-path pathing system. We first implemented A*, a way to input start and end positions for the search, and additional visualization for the generated path. This allowed us to easily test paths to ensure they were working properly, and later test the new sneak-attack paths alongside the A* paths. It was much easier to prototype path generation in StarDraft as opposed to in StarCraft, as we could ensure they were correct before implementing the much more daunting task of implementing everything in StarCraft.

4.5.2 UAlbertaBot

UAlbertaBot [14] is a StarCraft AI competition bot which comes with pre-built modular systems for playing the entire game; from gathering resources, to building an army and attacking. A view of the modular design can be seen in Figure 4.6. UAlbertaBot's modular architecture allows us to add a new strategic module to the bot which performs a drop-based sneak-attack strategy, for which the pathfinding is controlled by our new system, and the rest of the game mechanics are performed by the existing bot logic.

Specifically, we introduce a Pathfinding Manager, as well as heavily modifying the exist-

[1] https://github.com/davechurchill/stardraft

ing Transport Manager in order to utilize the new generated paths. We also introduce an Influence Map Manager that is used in the Pathfinding Manager for organizing and calculating the influence maps.

The Transport Manager module was responsible for moving the transport. The default module just patrolled the perimeter of the map. We had to implement picking up of units, following paths generated by the Pathfinding Manager, and returning to the player's base after dropping off units.

The Pathfinding Manager was responsible for creating and managing the various influence maps and path generations. It creates each specific influence map, and calculates the common paths at the start of each StarCraft game. Then every n frames - decided by the recalculation time described above in Subsection 4.4 - the damage and vision influence maps are recalculated using the Influence Map Manager based on the enemy units currently visible to UAlbertaBot. After these influence maps are recalculated, a new sneak-attack path is generated, to be used by the Transport Manager.

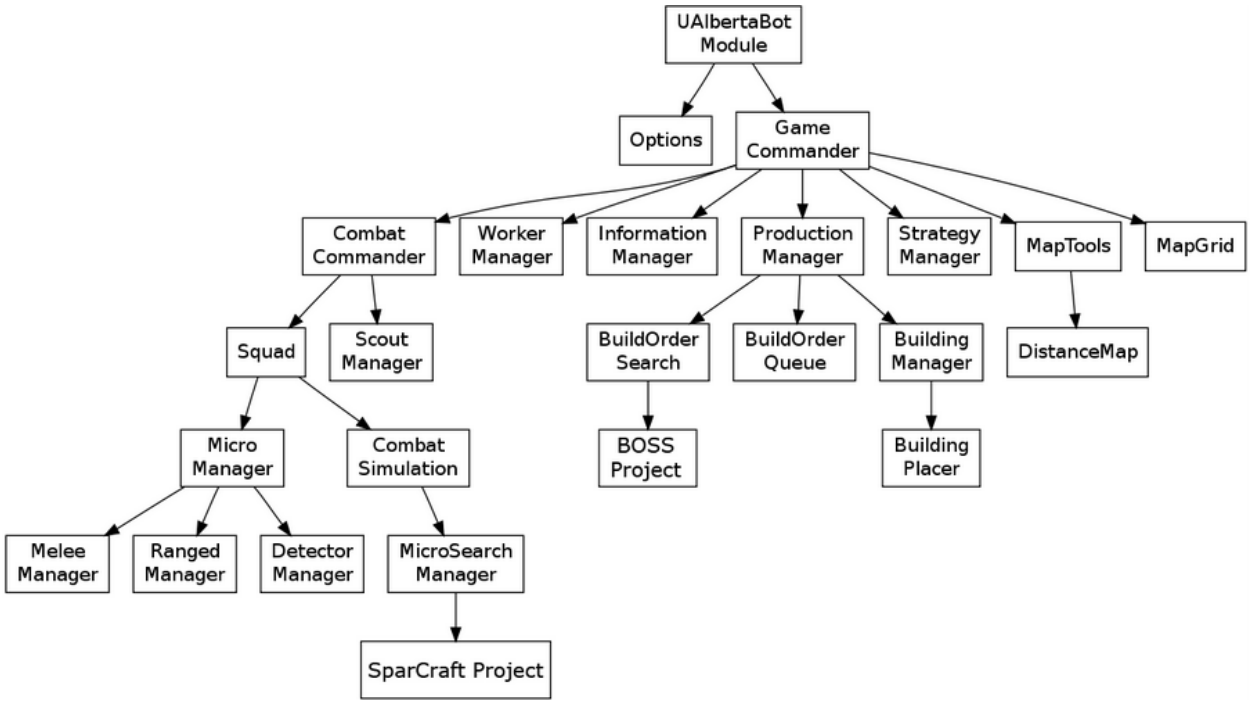


Figure 4.6: UAlbertaBot's modular design

Chapter 5

Experiments & Results

This section details the specific experiments we did in order to test both the sneak-attack path and the direct A* path systems, as well as the results of these experiments. All tests were performed single-threaded on a system representative of a mid-level modern gaming computer: an Intel(R) Core(TM) i5-6500 at 3.2GHz with 16GB DDR3 RAM and a GTX 1060 6GB.

5.1 Maps

In order to perform experiments in a setting as close to a competition environment as possible, we used all 10 maps from the 2020 AIIDE StarCraft AI Competition[1]. Game maps have properties which can affect the results of our system, such as map size and total number of player starting positions, which can be seen in Table 5.1. For all experiments, paths were calculated in real-time in well under the 50ms allowed per frame by the AI competitions.

[1]https://www.cs.mun.ca/ dchurchill/starcraftaicomp/

Map Name	Players	Size (width x height)
Benzene	2	128 x 112
Destination	2	96 x 128
Heartbreak Ridge	2	128 x 96
Aztec	3	128 x 128
Tau Cross	3	128 x 128
Andromeda	4	128 x 128
Circuit Breaker	4	128 x 128
Empire of the Sun	4	128 x 128
Fortress	4	128 x 128
Python	4	128 x 128

Table 5.1: Map names and information

5.2 Recorded Data

Many variables were recorded in these games in order to evaluate the effectiveness of our sneak-attack system. These variables - along with their explanations - are listed in *Table* 5.2.

5.3 Influence & Sneak-Attack Path Generation

The first experiment we performed was to qualitatively test whether or not the system-generated paths for sneak-attacks were intuitively similar to those generated by human players. A sample generated path be seen in Figure 5.1, which is generated for the flying transport unit, which is able to fly over un-walkable terrain. On the left we can see a path generated via standard A* which minimizes distance to the enemy base (direct path), which flies directly through the main base of the enemy. On the right, we see the path generated by our system, which avoids enemy vision by flying around to the right and behind the enemy base, similar to a path generated by a human expert, which is what we expected to

Recorded Stat	Data Type	Explanation
Sneak	bool	Whether transport is using Sneak path or Direct path
Time	int	Time game lasted in seconds
Frames In Vision	int	Number of frames transport was in enemy vision
Frame First Seen	int	Frame transport was first seen by the enemy
Transport Created	int	Frame transport was created
Transport Moved	int	Frame transport loaded and started path to enemy base
Transport Start Drop	int	Frame transport started its drop
Transport End Drop	int	Frame transport ended its drop
Health Start Drop	int	Transport health at start of its drop
Shields Start Drop	int	Transport shields at start of its drop
Dists Start Drop	int	Transport distance from goal at start of its drop
Health End Drop	int	Transport health at end of its drop
Shields End Drop	int	Transport shields at end of its drop
Dists End Drop	int	Transport distance from goal at end of its drop
Deaths	bool	Whether transport was destroyed
Map File Name	string	Name of map file
Map Width	int	Map width in StarCraft Tile units
Map Height	int	Map height in StarCraft Tile units
Players	int	Number of possible players in map. Also denotes number of possible starting locations
Enemy Race	string	Race of enemy
Percent Seen	double	Percentage of path transport was in enemy vision

Table 5.2: Recorded Stats with explanations

see. A transport following this path would avoid the area of enemy vision (red) and remain unseen by the enemy for as long as possible, allowing it to drop its units directly onto the resources of its opponents and attack its workers. In Figure 4.2d we can see an example StarCraft map: Andromeda, along with our system's calculated vision influence values (red and green) and common path influence values (light red). These qualitative tests were done on the variety of maps listed in Table 5.1 - with varying base locations as well - to ensure everything was working as expected.

These tests were quite successful, as the generated paths were performing just as we'd hoped: they were avoiding enemy influence for as long as possible, and then entering the influence at the lowest values to reach the destination. The paths were also quite similar to what a typical player might do with a transport unit under similar circumstances: a player would try to enter the enemy base where there were not many enemy units, instead of going straight in as they would always be spotted and destroyed.

5.4 Comparison of Direct & Sneak-Attack Paths

The next experiment we performed was to objectively compare our sneak-attack paths to direct (shortest distance) paths to the enemy base. For this experiment we played 1000 games on each of the 10 maps (5.1) for both the sneak-attack path and the direct path, for a total of 1000 x 2 x 10 = 20,000 games. For our experiments we (arbitrarily) chose the Protoss race, played against the built-in StarCraft AI using a randomly chosen race, and implemented a strategy which attempted to drop 4 Zealot units into the enemy mineral line with the Protoss Shuttle (transport) unit as fast as possible. This is a common early game strategy implemented by human players. As soon as the Shuttle unit was created, it was loaded with 4 Zealot units and immediately flown along the generated path (direct or sneak attack) to the goal position, which was given as the BWAPI enemy base location,

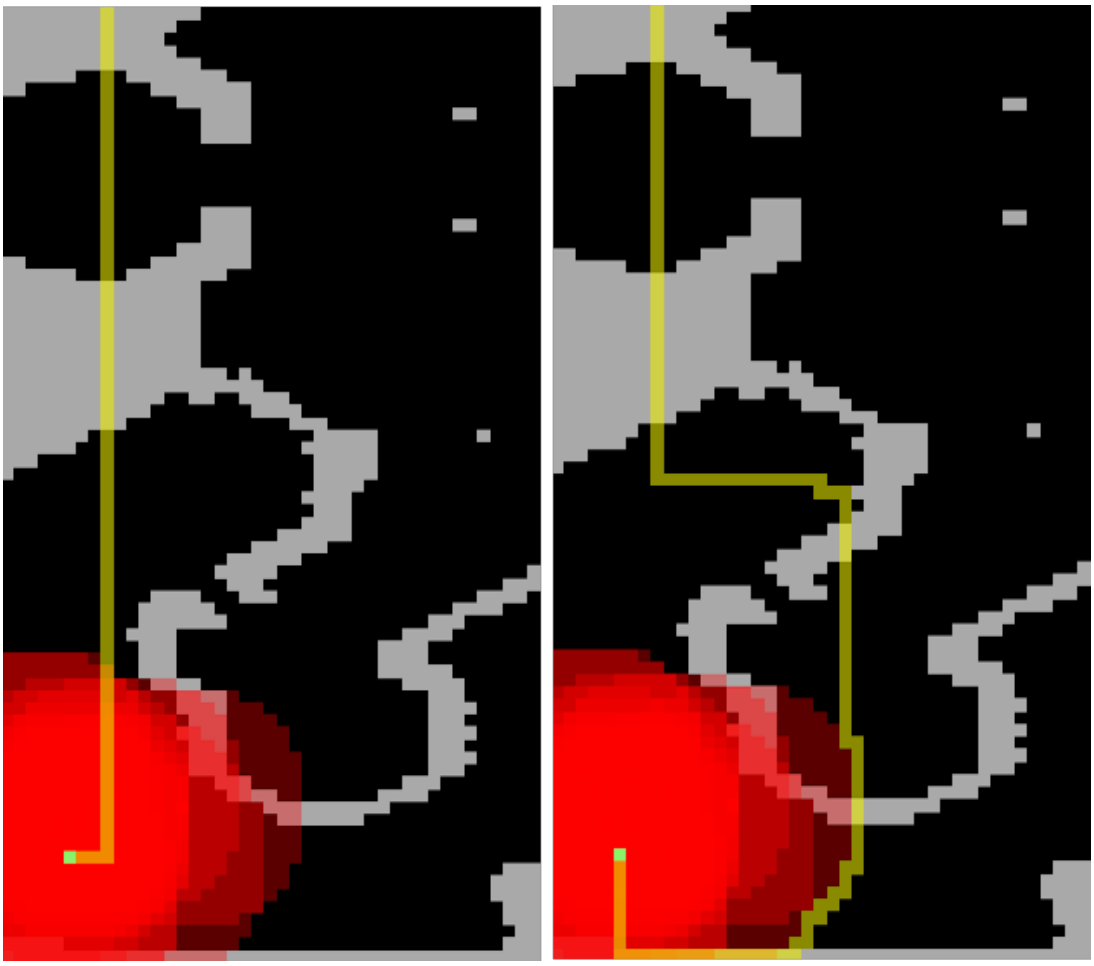


Figure 5.1: Comparison of direct path (left) and enemy vision avoidance path (right). The right path shows how the enemy vision influence is being avoided for as long as possible while going towards the goal.

near the enemy minerals. As the Shuttle flies, it explores more of the map and may discover additional enemy units, and so the influence maps and Sneak-attack paths were recalculated every 1 second to make use of this newly discovered information. Once the Shuttle reached 900 pixels from its destination, sneak-attack paths were no longer re-calculated, since at that distance it is probably usually within vision of the enemy base. Once the Shuttle was within 100 pixels of its goal position, it unloaded all of the Zealots which began to attack the enemy base. Our goal for these experiments was to evaluate the effectiveness of the sneak-attack paths only, not the effectiveness of the drop strategy itself, and so once the Shuttle was unloaded the game was ended and various statistics were gathered about the generated paths.

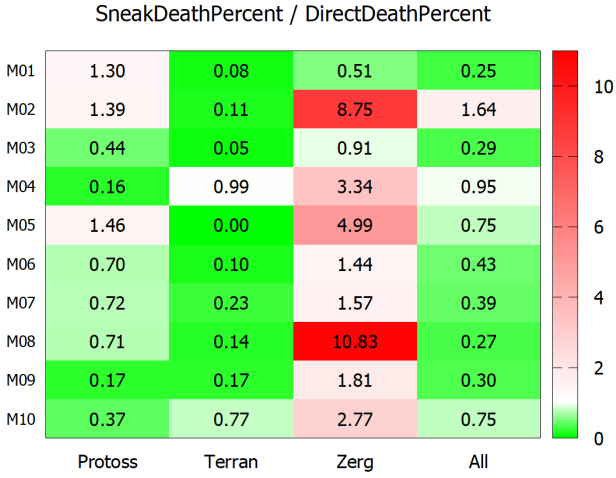
Our main research question was: if an AI agent chose to implement a drop strategy in any given game, would it be more effective to use our sneak-attack paths, or a direct path? In order to answer this question, a number of statistics related to the two different pathfinding methods were gathered for each game, which can be seen in Table 5.2. The most important pieces of data collected to compare the two paths were: how many times the Shuttle died trying to reach the enemy base, how many hit points it had remaining when it finished unloading, how long it took the Shuttle to reach the enemy base, and what percentage of the time did it spend within enemy vision. Intuitively, if our system created paths which when compared to direct paths resulted in the Shuttles dying less often and being seen for less time overall, then we consider this to be a success. Naturally, we expect that our sneak-attack paths will deviate from the direct paths in most cases, and result in a longer arrival time; however if it avoids enemy vision by doing so then this is preferred.

5.4.1 Results

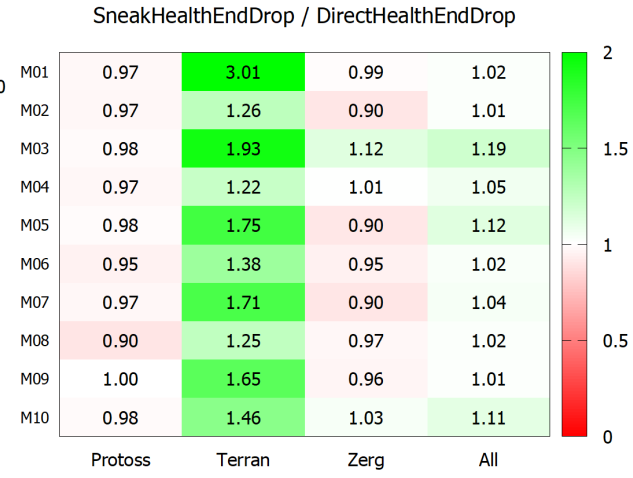
The results of this experiment(5.4) can be seen in Figure 5.2, which displays the mean values some recorded data, for each map played, against each race, with totals for all enemy races. Due to the different properties of each map such as size and number of player starting locations, and the different properties of each enemy race, it is helpful to break down the results in this way for a more detailed analysis. To aid in visualizing these results, each of the cells have been coloured green if the sneak-attack path result performed better, or red if the direct path performed better. For example, if we look at the top-right cell of Figure 5.2a, we see that the ratio of sneak-attack path Shuttle deaths to direct path shuttle deaths was 0.25, meaning that Shuttles using the direct path died 4 times as often than when using our sneak-attack paths, which we count as a success.

Overall, the statistics show that sneak-attack paths outperform direct paths in 3 of the 4 metrics, while losing to direct paths in overall path duration, which was both expected and unavoidable. In the top-right of Figure 5.2a we see a value of 0.25, meaning that Shuttles using direct paths died four times as much as those using our sneak-attack paths on map M01. Shuttle deaths are caused by taking damage from enemy units that can attack air units, such as cannon-like static defenses, or mobile units like Marines or Hydralisks - who have powerful ranged attacks that destroy our Shuttle quickly. Of note in these results is the vastly superior performance against the Terran race, which is explained by the fact that Terran are the only race whose first military unit can attack air units - the Marine. Marines are cheap air attacking units that can be created very early in the game by Terran. When the built-in AI controls the Terran race, it often creates many Marines which can easily kill incoming Shuttles if they are using the direct path, which our sneak-attack path helps to avoid, resulting in far fewer deaths. We can also see from this table that our sneak-attack paths appear to perform worse when playing against the Zerg race, which can be explained

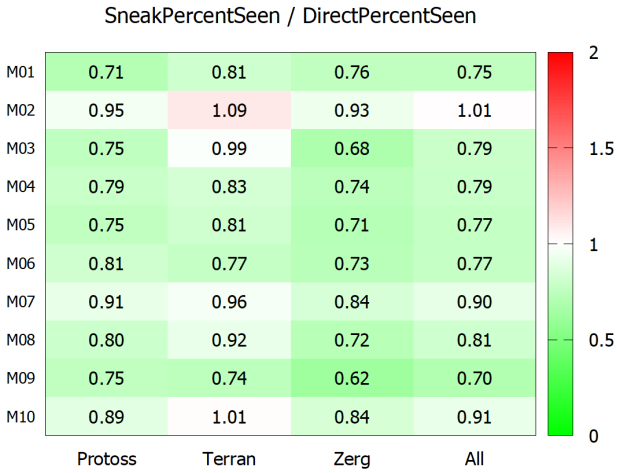
by the choice of build-order used by the hard-coded built-in AI. One of the build-orders the built-in Zerg AI can choose to carry out involves the creation of many Hydralisk units. Hydralisks are mid to late game ranged attacking units often used against Protoss that have powerful anti-air weapons that are capable of killing Shuttles quite quickly. In an unfortunate coincidence, the built-in AI creates these units just seconds before our Shuttle arrives, and gathers them up precisely on top of our chosen goal location on these specific maps (Destination, EmpireOfTheSun, TauCross). The direct path uses a shorter path to the enemy base, and arrives about 20 seconds earlier before the Hydralisks are completed, and therefore does not die to them in these specific cases. Since our experiment did not include any logic to cancel the drop when it detected this mass of Hydralisk units near the goal location, it resulted in more deaths for these specific scenarios. This was a fascinating edge-case that we were not expecting, and it uncovered a weakness in our hard-coded build-order timing for our sneak-attack, showing just how important it is to choose a new goal location for the drop, or to cancel a drop altogether if this scenario presents itself in a real game. In Figure 5.2b we can see a comparison of the resulting hit points remaining for the Shuttle unit once the drop has been completed for the two paths. Due to Terran's easy access to air defenses, we see that the sneak-attack path avoids more damage vs. the Terran race, and overall against all races. In Figure 5.2c we see a comparison of the ratio of the percentage of the duration of the path that the Shuttle was seen for both path types. We can clearly see from these results that the sneak-attack paths result in our Shuttle being hidden from the enemy's view for a longer amount of time while executing the sneak-attack, in nearly every tested situation. Overall, we believe that this new method outperforms direct paths for the purposes of executing sneak-attacks.



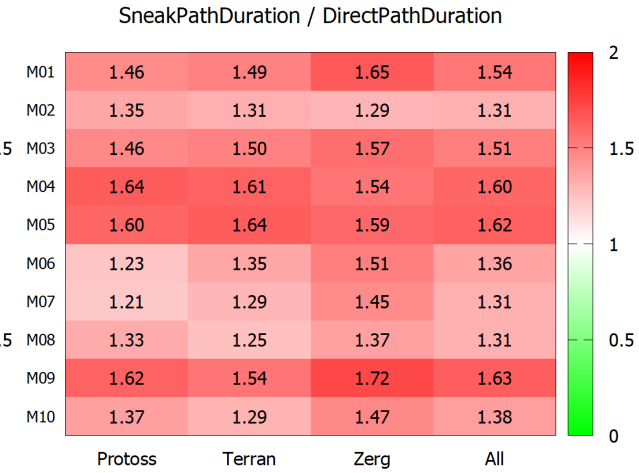
(a) Ratio of the percentage of total games for the Shuttle unit
died when following both path types.



(b) Ratio of Shuttle average health remaining after drop completed. Games where the Shuttle died were not counted here.



(c) Ratio of the average percentage total path duration the Shuttle was in vision of enemy when following both path types.



(d) Ratio of the average duration the Shuttle took to reach the enemy base when following both path types.

Figure 5.2: Results of the four main statistics recorded for Experiment 2. A green shade indicates our sneak-attack system outperforming direct paths for the given metric, while a red shade indicates underperforming.

5.5 Building Placement for Defending Sneak-Attacks

For this last experiment, we wanted to showcase another use for the sneak-attack system - by using it for defense instead of offense. We do this by testing whether the buildings placed using Algorithm 3 mentioned in Section 4.3.2 were placed in such a way that would make it very difficult for an enemy to perform a sneak-attack at the player's base. We do not have any objective way to measure success, since our bot played against the built-in AI which does not execute sneak-attack strategies. However, we did generate several sample building placements and compared them to those that professional StarCraft players implement when defending against sneak-attacks, and found that they have the same properties as expert human players. An example calculated placement of buildings can be seen in Figure 5.3, in which the buildings have been spaced as far apart as possible in order to maximize the vision radius of the 3 buildings constructed within our base. If we suspected our enemy is wanting to execute a sneak-attack, it makes intuitive sense that we would want to cover as much ground within our base with vision in order to detect when they may be trying to attack, as well as maximizing the distance they would have to travel to sneak into our base. We can see from our example that the best possible sneak-attack path found by our system would require the enemy to travel below and around the right-hand side of our base, which is far longer than that direct path.

Figure 5.3: An example calculated placement of 3 buildings (dark blue) which results in an optimal configuration for maximizing the length of an incoming sneak-attack path (white) from the starting position (yellow) to the goal position (purple) near our base's resources (teal/green). Shown in transparent light red is the vision influence map for the buildings used to perform the sneak-attack pathfinding.

Chapter 6

Conclusion and Future Work

In this thesis, we presented a novel method for constructing sneak-attack paths for RTS games by combining the ideas of influence maps for vision, damage, and commonly traveled areas with the pathfinding ability of the A* algorithm. The combined system allowed for the influence maps to serve as a guide for A*, resulting in paths exhibiting avoidance behaviors. We implemented our system in a modular fashion within UAlbertaBot using BWAPI, and compared our sneak-attack paths to direct paths created via a shortest distance implementation of the A* algorithm in a real-time competitive setting against the built-in AI in StarCraft. Our results showed that our paths resulted in fewer transport deaths, less overall damage taken, and less time spent within enemy vision range than the direct paths, which we view as a successful measure for sneak-attack pathfinding. Our results also showed that these paths resulted in overall longer times to arrive at the enemy base, and identified an edge case on 3 specific maps against the Zerg race in which these results were quite disastrous - however in an actual competitive environment we suspect the AI agent would simply modify its goal position or cancel the sneak-attack in those cases. Overall we feel like this new system was a success for calculating sneak-attack paths in StarCraft, and could be easily adapted to calculating other strategic paths for other games.

We then used this system in a defensive manner as by calculating building positions to help prevent a sneak-attack on our own base, and demonstrated an example building placement which was calculated by our system, and resembled those of expert human players. While we do not yet have an objective measure for determining success for this defensive system we feel that it intuitively shows much promise, and part of our future work will be testing this in a competitive setting in which more objective measures can be recorded. In this thesis, we presented the following major contributions: 1) a combined influence map and A* pathfinding system for avoiding influential areas, 2) unit guidance in StarCraft for performing sneak-attacks, and 3) a building placement system for defending against sneak-attacks.

In the future, we would like to test whether or not comparable results would be possible to implement similar behavior to our sneak-attack system by exclusively using influence maps, and without the use of a separate pathfinding algorithm. If we include another influence map using the distance from enemy base as the influence, we could influence our paths to go towards the enemy base, while still being influenced by the other maps by just navigating toward the areas of highest influence - saving time and memory by removing A* search. We also hope to test this system in a more competitive setting as soon as possible by entering the overall agent into a future StarCraft AI Competition, to defeat enemies with our novel sneak-attack superiority.

Bibliography

[1] e-sports earnings. URL: https://www.esportsearnings.com/.

[2] The starcraft encyclopedia. URL: https://liquipedia.net/starcraft/.

[3] Starcraft AI competitions, Bots, and Tournament Manager Software. *IEEE Transactions on Games*, 11(3):227–237, 2019. doi:10.1109/TG.2018.2883499.

[4] Gonçalo P. Amador and Abel J.P. Gomes. XTrek: An Influence-Aware Technique for Dijkstra's and A Pathfinders. *International Journal of Computer Games Technology*, 2018, 2018. doi:10.1155/2018/5184605.

[5] Phillipa Avery and Sushil Louis. Coevolving influence maps for spatial team tactics in a RTS game. *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO '10*, pages 783–790, 2010. doi:10.1145/1830483.1830621.

[6] Phillipa Avery and Sushil Louis. Coevolving team tactics for a real-time strategy game. *2010 IEEE World Congress on Computational Intelligence, WCCI 2010 - 2010 IEEE Congress on Evolutionary Computation, CEC 2010*, (Im), 2010. doi:10.1109/CEC.2010.5586211.

[7] Phillipa Avery, Sushil Louis, and Benjamin Avery. *Evolving Coordinated Spatial Tactics for Autonomous Entities using Infuence Maps*. IEEE, 2009.

[8] Maurice Bergsma and Pieter Spronck. Adaptive spatial reasoning for turn-based strategy games. *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, pages 161–166, 2008.

[9] Michael Buro. Orts - a free software rts game engine. URL: `https://skatgame.net/mburo/orts/`.

[10] Michael Buro. Real-time strategy gaines: A new AI research challenge. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1534–1535, 2003.

[11] Michaël Carlos Gonçalves Adaixo and Doutor Abel João Padrão Gomes. Influence Map-Based Pathfinding Algorithms in Video Games. Technical report, 2014.

[12] Michal Čertický and David Churchill. The Current State of StarCraft AI Competitions and Bots. *Proceedings of the AIIDE 2017 Workshop on Artificial Intelligence for Strategy Games*, pages 2–7, 2017.

[13] David Churchill. Stardraft: Starcraft map visualization tool, 2020. https://github.com/davechurchill/stardraft.

[14] David Churchill and Michael Buro. Build order optimization in StarCraft. In *Proceedings of the 7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2011*, 2011.

[15] David Churchill, Mike Preuss, Florian Richoux, Gabriel Synnaeve, Alberto Uriarte, Santiago Ontañón, and Micha Certick. StarCraft Bots and Competitions. 2015.

[16] Xiao Cui and Hao Shi. A * -based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[17] Holger Danielsiek, Raphael Stuer, Andreas Thom, Nicola Beume, Boris Naujoks, and Mike Preuss. Intelligent moving of groups in real-time strategy games. *2008 IEEE*

Symposium on Computational Intelligence and Games, CIG 2008, pages 71–78, 2008. doi:10.1109/CIG.2008.5035623.

[18] Samuel Erdtman and Johan Fylling. Pathfinding with Hard Constraints - Mobile Systems and Real Time Strategy Games Combined. 2008.

[19] Ross Graham, Hugh Mccabe, and Stephen Sheridan. Pathfinding in computer games. *ITB Journal*, 4(8):57–81, 2003. doi:10.21427/D7ZQ9J.

[20] Johan Hagelbäck. Potential-field based navigation in StarCraft. *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, pages 388–393, 2012. doi:10.1109/CIG.2012.6374181.

[21] Johan Hagelbäck and Stefan J Johansson. The Rise of Potential Fields in Real Time Strategy Bots. Technical report. URL: www.aaai.org.

[22] Johan Hagelbäck and Stefan J. Johansson. A multiagent potential field-based bot for real-rime strategy games. *International Journal of Computer Games Technology*, 2009(1), 2009. doi:10.1155/2009/910819.

[23] Peter E Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics*, 4(2):100–107, 1968.

[24] Adam Heinermann. Broodwar API. https://github.com/bwapi/bwapi, 2013. URL: https://github.com/bwapi/bwapi.

[25] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 500–505, 1985. doi:10.1109/ROBOT.1985.1087247.

[26] Yoram Koren and Johann Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. 1991.

[27] Liu Liang and Li Longshu. Regional cooperative multi-agent Q-learning based on potential field. *Proceedings - 4th International Conference on Natural Computation, ICNC 2008*, 6:535–539, 2008. doi:10.1109/ICNC.2008.173.

[28] Dave Mark. *Modular tactical influence maps*. 2015. doi:10.1201/b18373.

[29] Victor Martell and Aron Sandberg. Performance Evaluation of A * Algorithms. 2016.

[30] Santiago Ontañon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, Mike Preuss, and Santiago Ontañón. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. Technical Report 4, 2013. URL: https://hal.archives-ouvertes.fr/hal-00871001.

[31] Amit Patel. Heuristics. URL: https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[32] Filip Pentikäinen and Albin Sahlbom. Combining Influence Maps and Potential Fields for AI Pathfinding. Technical report, 2019. URL: www.bth.se.

[33] Dave C Pottinger. Terrain Analysis in Realtime Strategy Games. *Proceedings of Computer Game Developers Conference*, 2000.

[34] Greg Smith, Phillipa Avery, Ramona Houmanfar, and Sushil Louis. Using co-evolved RTS opponents to teach spatial tactics. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010*, pages 146–153, 2010. doi:10.1109/ITW.2010.5593359.

[35] Paul Tozour. Influence mapping. In Mark DeLoura, editor, *Game Programming Gems 2*, pages 287–297. Charles River Media, 2001.

[36] Alberto Uriarte and Santiago Ontã. Kiting in RTS Games Using Influence Maps. Technical report, 2012. URL: www.aaai.org.

[37] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[38] Nathan Wirth and Marcus Gallagher. An influence map model for playing Ms. Pac-Man. *2008 IEEE Symposium on Computational Intelligence and Games, CIG 2008*, pages 228–233, 2008. doi:10.1109/CIG.2008.5035644.

[39] A. L. Zobrist. A model of visual organization for the game of go. In *Managing Requirements Knowledge, International Workshop on*, volume 1, page 103, Los Alamitos, CA, USA, may 1969. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/AFIPS.1969.10, doi:10.1109/AFIPS.1969.10.