Sneak-Attacks in StarCraft using Influence Maps with Heuristic Search

Lucas Critch
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada
lrc374@mun.ca

David Churchill
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada
dave.churchill@gmail.com

***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 paper introduces a novel method for finding sneak-attack paths in StarCraft 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. Our results show that our 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 a proof of concept for calculating building placements to defend against enemy sneak-attacks.**

I. Introduction

StarCraft: Brood War is a Real-Time Strategy (RTS) game that continues to be popular for conducting AI research due to its complex game theoretical properties, large state and action spaces, and difficulty of play for humans and computers. Some of the main sub-problems that exist in the RTS genre include real-time planning, collaboration, path-finding, uncertain decision making, enemy modeling, spatial and temporal reasoning, and resource management [1].

In StarCraft, players gather resources, build buildings, and construct armies to do battle with their opponent. Players can gain competitive advantages in various ways in RTS games, and one common way is to execute what is known as a *sneak-attack*: in which a player guides its units to flank an opponent's base while attempting to remain unseen until the attack commences. By doing so, the player can surprise the enemy and proceed to quickly attack the base before they have a chance to defend appropriately. One effective sneak-attack strategy that a player can execute is called a *drop*, wherein the player constructs a flying unit known as a *transport* which carries attacking units over the environment and base defenses in order to shuttle them deep into the enemy base to attempt to destroy vital strategic units, such as workers [2].

978-1-6654-3886-5/21/$31.00 ©2021 IEEE

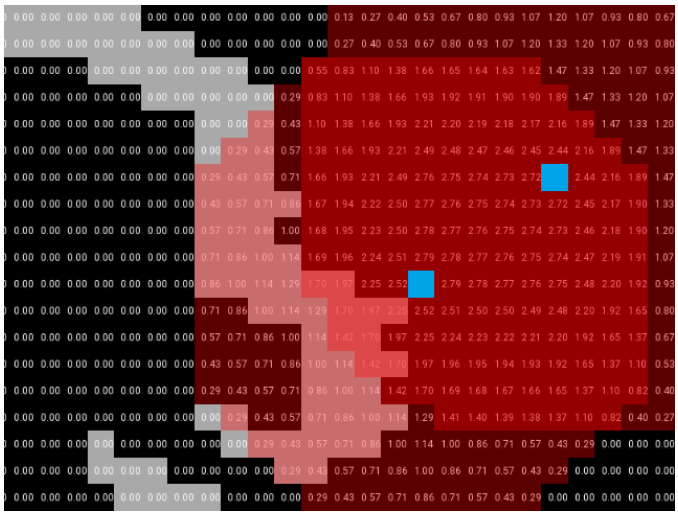


Fig. 1: Example influence map showing 2 enemy units (blue) and their influence radius (red). Overlapping areas have higher values, denoting regions of higher repulsion.

This paper will tackle two main problems related to sneak-attacks. We will first create a path-finding system for determining the best routes to follow for executing sneak-attacks by combining the efficiency of heuristic search with the intuitive representation of influence maps. We will then use this system as an analysis tool to show a proof of concept to determine an optimal positioning of our own base buildings to prevent sneak-attacks from enemy units.

To construct a sneak-attack path, we will pair the idea of enemy avoidance with a shortest path path-finding algorithm. Inspired by similar work from [3] and [4], we use the A* path-finding algorithm, as well as influence maps that build areas of influence from enemy vision and damage. The influence map is used to construct the cost function g(n) and heuristic evaluation h(n) for A*, resulting in the construction of paths to the enemy base, which balance minimizing enemy detection as well as the total distance to the enemy base. An example influence map of enemy unit vision can be seen in Figure 1, with higher values indicating areas of higher repulsion.

Following this section, the paper is broken down into

logical sections. First, in Section II, we will talk about the background of the algorithms used in our system, namely influence maps and A* search. Subsection II-C breaks down some works related to the current work, such as research done on StarCraft or other RTS games using influence maps or search techniques. Then in Section III, we will get into our system's methodology; how influence maps and A* are involved, how they are used within StarCraft, and what we have done to create the system. In Section IV, the experiment we conducted is talked about in great detail. Results and discussion are contained in Section IV as well - complete with recorded data, tables, and figures. Finally, our conclusions in Section V: what we made of our experiment and findings, and where this work could lead in the future.

II. Background and Related Work

A. Influence Maps

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 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 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.

B. A* Search Algorithm

A* is a popular search best-first search algorithm that expands nodes on a graph in a priority queue until either it finds a goal, or the graph has been entirely searched and no solution is found. This priority queue is sorted as a function of search node n, denoted $f(n)$, with $f(n) = g(n) + h(n)$, 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 [5]. The A* algorithm is widely used in industry RTS games [2], and its use of a cost-based heuristic function pairs perfectly with our intended integration with influence maps.

C. Related Work

Influence maps were built on work done by Zobrist on the game GO in 1969 [6], and have been widely used for both academic and industry game purposes such as in another popular RTS game, Age of Empires 2 [7]. In this section, we will discuss related work in the area that we will use as inspiration for our work.

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.

Hagelbäck [9] 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 path-finding AI.

In 2014, Adaixo and Gomes [10] 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.

Xtrek [4], an algorithm that combines influence maps with normal path-finding by Amador and Gomes 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 using positive and negative numbers for attractors and repulsors, respectively. When close to an attractor, the $g(n)$ part of the A* equation listed in Subsection II-B is not used in order to assure convergence to the centre 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 path-finding calculations.

Pentikäinen and Sahlbom [11] 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.

Building on these ideas, the following sections will explain how we will combine the use of influence maps and the A* algorithm for executing and defending against sneak-attacks in StarCraft.

III. Methodology

Before getting into the specifics of our system, we will first discuss the roles that both influence maps and A* have within

Algorithm 1 Influence Calculation

$(sx, sy) \leftarrow$ 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 $dist < v$ **then**
            Influence $i \leftarrow p - (p \times \frac{dist}{d})^4$
            $visionMap[x][y]$ += 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 $dist < v$ **then**
                $i \leftarrow p - (p \times \frac{dist}{d})^4$ // influence
                $pathsMap[x][y]$ += i

it. Followed by that, we discuss how we integrated the system into StarCraft and how each piece ties together to create our sneak-attack system.

A. Influence Map Implementation

To use influence maps with StarCraft environment data, the influence maps will store values that will be used to implement behaviors relevant to our path-finding 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 travelled areas. 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 this, we can easily construct influence maps as 2D arrays using the same StarCraft tile map dimensions, and have them be easily readable and comparable to the in-game map. 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, taking up less than 1MB of memory for even the largest maps. Enemy units will be points of interest, with influence radiating up to a hard cutoff at the edge of their vision/attack radius. Figure 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 the cost function for the A* algorithm to minimize the amount of time spent near these areas.

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

- **Vision Map**: Stores influence of visible 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 points radiate outwards from the enemy position, deceasing based on distance from enemy, and stop at the 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 based on all-pairs shortest paths between all starting base locations on the map, which are often travelled by enemy units/scouts when moving to or from bases. By avoiding these commonly walked paths, we further decrease the chances of being seen by the enemy.

Individual influence values are calculated based on the following equation presented in [3]:

$$m_{\text{x,y}} = p - (p * (\frac{dist}{d}))^4 \tag{1}$$

where $m_{\text{x,y}}$ 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 can be seen in Figure 3.

B. 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:

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

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.

C. StarCraft Implementation

To implement this new system in StarCraft, we require a way to interface with the StarCraft game engine. To do this, we used the BroodWar API programming interface (BWAPI) [12], which allows us full control over the StarCraft game engine to query game data and issue game commands. We also used UAlbertaBot [13], 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 path-finding is controlled by our new system, and the rest of the game mechanics are performed by the existing bot logic, such as scouting the map to find the enemy base location. We also use the StarCraft map visualizer tool StarDraft[1], which allows us to perform map analysis and path-finding offline, makes for easier debugging, and was used to create most of the visualizations for this paper.

This integrated system reads relevant influence map information from StarCraft every 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, unwalkable 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 1, 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 paths 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 III-B. Finally, our system has several additional factors 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 recalculated. 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 dropship has gotten sufficiently close to the enemy base, there are diminishing returns on recalculating the sneak-attack path. For example, if the dropship 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 dropship 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 behaviour to expert human players in similar situations, based on observations from professional games and replays.

D. Sneak-Attack Defense Building Positioning

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. It should be noted that this is a proof of concept - it is simply shown as an example of how the sneak-attack system could be used defensively. The algorithm we chose to implement this was quite intuitively simple: iterate over all legal building positions for the first $n = 3$ 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.

IV. Experiments and Results

In this section, we will describe all of the experiments we performed to test the performance of our new sneak-attack path system. 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. In order to perform experiments

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

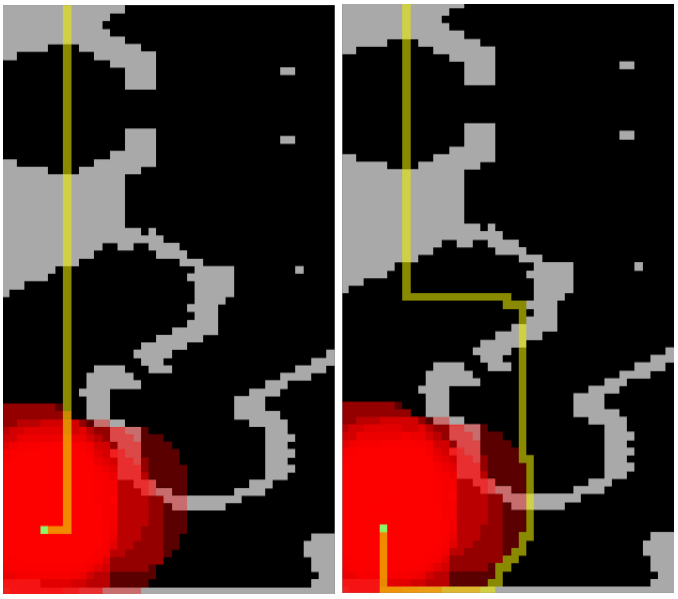


Fig. 2: Comparison of A* shortest path (direct path) to enemy base (left) and our system sneak-attack path to enemy base (right). Paths are shown in yellow, with enemy influence shown in red. Black indicates the area of the map which is walkable, while gray is un-walkable (walls).

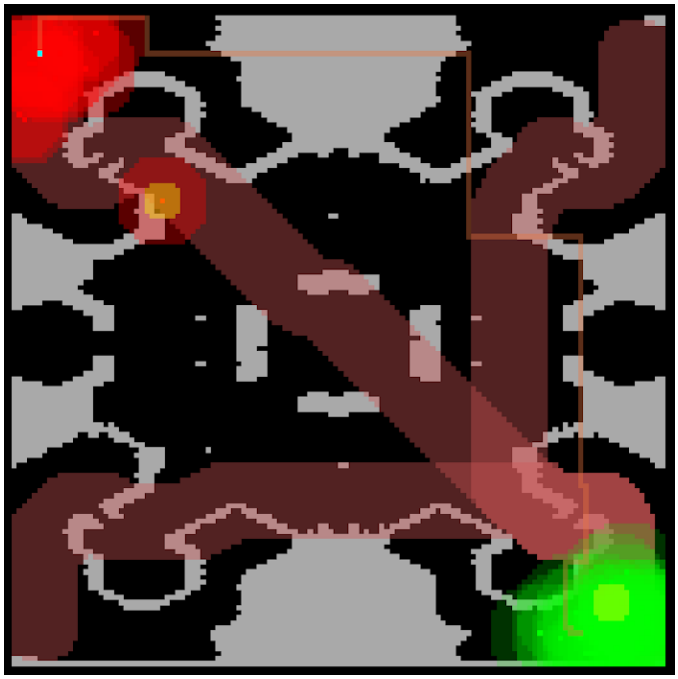


Fig. 3: StarCraft map with influence maps visible: enemy vision (red), our vision (green), and common paths (light red).

in a setting as close to a competition environment as possible, we used all 10 maps from the 2020 AIIDE StarCraft AI Competition[2]. 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 I. For all experiments, paths were calculated in real-time in well under the 50ms allowed per frame by the AI competitions.

A. Influence and 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 2, which is generated for the flying Dropship 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 see. A Dropship 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 3 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). In the next experiment, we will give detailed numerical results comparing these two paths.

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

B. 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 for both the sneak-attack path and the direct path, for a total of 1000 x 2 x 10 = 20,000 games. For each game 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 (dropship) unit as fast as possible. 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, 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

Map ID	Map Name	Players	Size (width x height)
M01	Benzene	2	128 x 112
M02	Destination	2	96 x 128
M03	Heartbreak Ridge	2	128 x 96
M04	Aztec	3	128 x 128
M05	Tau Cross	3	128 x 128
M06	Andromeda	4	128 x 128
M07	Circuit Breaker	4	128 x 128
M08	Empire of the Sun	4	128 x 128
M09	Fortress	4	128 x 128
M10	Python	4	128 x 128

TABLE I: Map names and information

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

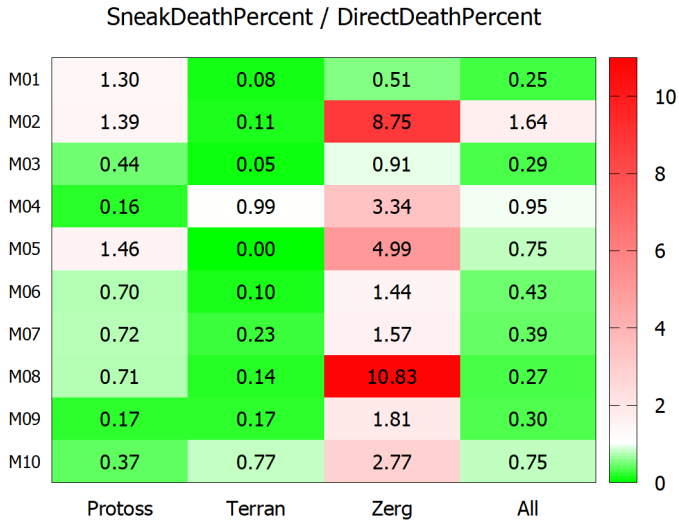
TABLE II: Recorded Stats with explanations

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 path-finding methods were gathered for each game, which can be seen in Table II. 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.

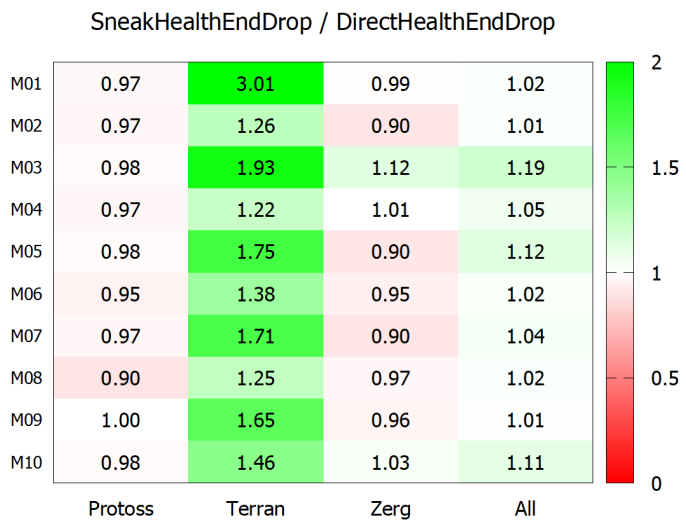
The results of this experiment can be seen in Figure 4, which displays the mean values for each experiment, for each map played, against each race for each AI opponent, along with the totals for all enemy races. Due to the different properties of each map such as size and number of player starting locations, as well as 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 4a, 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 as when using our sneak-attack paths, which we count as a great 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 4a 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, Dragoons, or Hydralisks. 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. 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, which have powerful anti-air weapons 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.

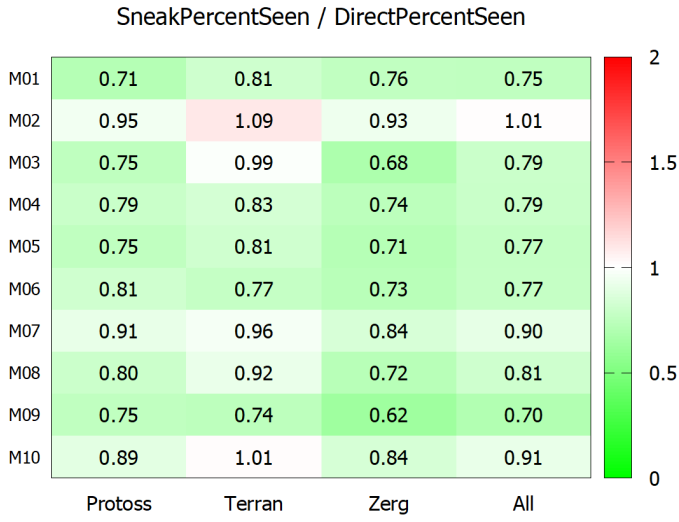
In Figure 4b we can see a comparison of the resulting hit points remaining for the Shuttle unit once the drop has been completed for the two paths. Due to Terran's easy access to air defenses, we see that the sneak-attack path avoids more damage vs. the Terran race, and overall against all races.



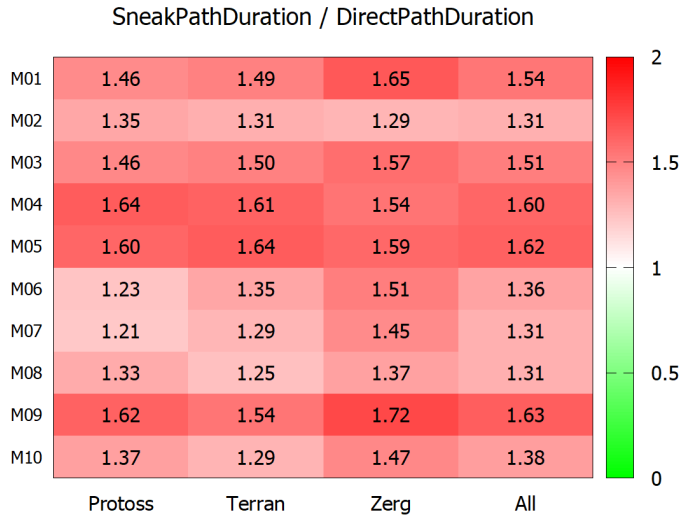
(a) Ratio of the percentage of total games for the Shuttle unit died when following both path types.



(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.

Fig. 4: 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.

In Figure 4c 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.

C. Building Placement for Defending Sneak-Attacks

Now that we have a system in place for calculating sneak-attack paths, we can use this system in a defensive manner by planning our own building positions to best prevent sneak-attacks. For this experiment 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, 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 of 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.

Fig. 5: An example calculated placement of 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 path-finding.

V. Conclusions and Future Work

In this paper, we presented a novel method for constructing sneak-attack paths for RTS games by combining the ideas of influence maps for vision and damage with the path-finding ability of the A* algorithm. 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 dropship 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 path-finding. 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 the future, we would like to test whether or not comparable results would be possible by exclusively using influence maps for guiding path-finding, without needing to combine them with a separate path-finding algorithm like A*. 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.

References

[1] M. Buro and T. M. Furtak, "RTS games and real-time AI research," *Proceedings of the Behavior Representation in Modeling and Simulation Conference*, 2004.

[2] S. Ontañon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, M. Preuss, and S. Ontañón, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," Tech. Rep., 2013. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00871001

[3] D. Mark, *Modular tactical influence maps*, 2015.

[4] G. P. Amador and A. J. Gomes, "XTrek: An Influence-Aware Technique for Dijkstra's and A Pathfinders," *International Journal of Computer Games Technology*, vol. 2018, 2018.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[6] A. L. Zobrist, "A model of visual organization for the game of go," in *Managing Requirements Knowledge, International Workshop on*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, may 1969, p. 103. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/AFIPS.1969.10

[7] D. C. Pottinger, "Terrain analysis for real-time strategy games," in *Proceedings of Game Developers Conference*, 2000.

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

[9] J. Hagelbäck, "Potential-field based navigation in StarCraft," *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, pp. 388–393, 2012.

[10] M. Carlos Gonçalves Adaixo and D. Abel João Padrão Gomes, "Influence Map-Based Pathfinding Algorithms in Video Games," Tech. Rep., 2014.

[11] F. Pentikäinen and A. Sahlbom, "Combining Influence Maps and Potential Fields for AI Pathfinding," Tech. Rep., 2019. [Online]. Available: www.bth.se

[12] A. Heinermann, "Broodwar API," https://github.com/bwapi/bwapi, 2013. [Online]. Available: https://github.com/bwapi/bwapi

[13] D. Churchill and M. Buro, "Build order optimization in StarCraft," in *Proceedings of the 7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2011*, 2011.