Combining Influence Maps with Heuristic Search for Executing Sneak-Attacks in RTS Games

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 become a popular domain for AI research due to their large state and action spaces, as well as complex sub-problems. One popular strategy in RTS games is the idea of a "Sneak-Attack", in which one player attempts to sneak enemy units into the base of their opponent without being seen, in order to gain the element of surprise. In this paper we will present initial results on combining influence maps with heuristic search to produce a path-finding system which allows us to guide StarCraft drop ships in order to execute a sneak attack. Our preliminary results show that by combining these two techniques, we can efficiently and automatically produce paths that guide our drop ships in a stealthy manner toward the enemy base, minimizing distance traveled and avoiding enemy vision of our army.**

I. Introduction

Starcraft is a Real-time Strategy (RTS) game that has become popular for conducting AI research due to its complex game theoretical properties, and large state and action spaces. Some of the main sub-problems that exist in the RTS genre include: real-time planning, collaboration, pathfinding, uncertain decision making, opponent modeling, spatial and temporal reasoning, and resource management [1]. In Starcraft, a player wins the game by gathering resources, constructing an army, and destroying the units of their enemy, with the player acting as a general, issuing commands to each unit in the game to carry out those specific tasks. When designing an AI system for Starcraft, that system must perform all decision making related to those tasks, as well as carrying out the specific commands required to execute them.

One of the ways in which players can gain a competitive advantage in an RTS game is to execute a *sneak-attack*, in which the players units approach the enemy base in such a way that they avoid being seen by the opposing army. By doing this, the player's army can surprise the enemy and proceed to deal damage to the enemy base before they have a chance to prepare or react properly. One way that a player can execute an effective sneak-attack strategy is by constructing a flying unit known as a *drop ship* which can carry and transport units over environment geometry and base defenses and deposit them deep into the enemy base to attempt to kill the worker units of the enemy, which is known as a *drop strategy*. This paper attempts to solve the problem of constructing these sneak-attack paths by combining the efficiency of heuristic search with the intuitive representation of influence maps.

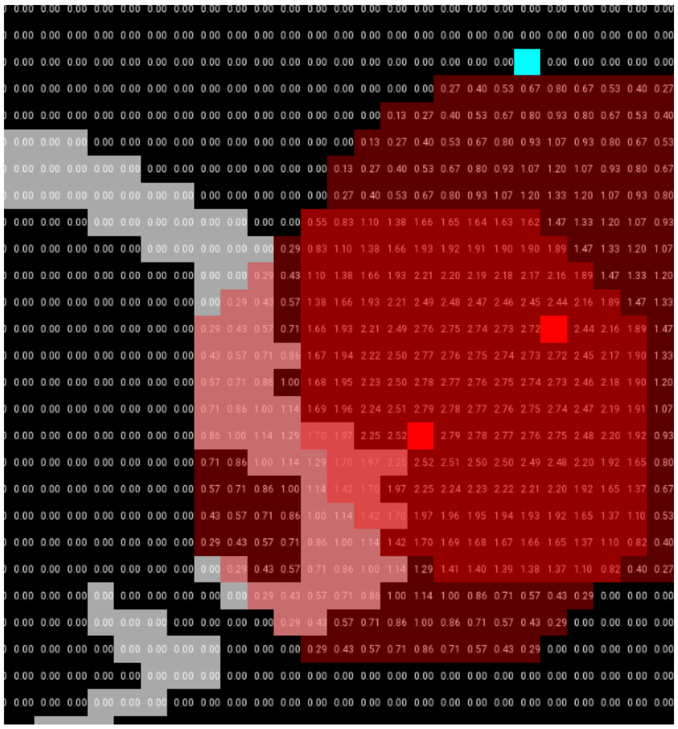


Fig. 1: An example influence map of around 2 enemy units with vision radius's 8 and 9. Brightest red tiles are the enemy units. Each unit has a red radius showcasing their vision influence. The overlapping areas have higher values, denoting regions of higher repulsion.

In order to construct these paths, we must perform a shortest path path-finding method in conjunction with the concept of avoiding enemy vision as we approach the goal. Inspired by similar work in [2], we combine the A* path-finding algorithm with an influence map that builds up areas of repulsive influence around enemy vision and damage. The influence map is used to construct the cost and heuristic functions for A*, resulting in the construction of paths to the enemy base which balance minimizing enemy detection as well as distance to the enemy base. An example influence map of enemy unit vision

Algorithm 1 Influence Calculation

```
1: $(sx, sy) \leftarrow$ size of Starcraft map
2: $visionMap[][] \leftarrow$ zeros$(sx, sy)$
3: $damageMap[][] \leftarrow$ zeros$(sx, sy)$
4: $pathMap[][] \leftarrow$ zeros$(sx, sy)$
5: $p \leftarrow 1.0$
6: $r \leftarrow$ Starcraft unit vision radius
7: $d \leftarrow$ max distance away from center of $r$ (is $u_{\text{x,y}}$)
8: $pBase \leftarrow$ Player base
9: // calculate vision and damage influence maps
10: for Starcraft unit $u \in$ enemyUnits do
11:     $r \leftarrow u$'s vision radius
12:     $d \leftarrow$ max distance away from center of $r$ (is $u_{\text{x,y}}$)
13:     for $(x, y)$ from $(0, 0)$ to $(sx, sy)$ do
14:         $dist \leftarrow$ distance from $(x, y)$ to $u_{\text{x,y}}$
15:         if $dist < v$ then
16:             Influence $i \leftarrow p - (p \times \frac{dist}{d})^4$
17:             $visionMap[x][y]$ += $i$
18:             $damageMap[x][y] \leftarrow p$
19: // calculate common path influence map
20: for $base$ not $pBase$ do
21:     $path \leftarrow$ path from $base$ to $pBase$
22:     for $pos_{\text{x,y}}$ in $path$ do
23:         for $(x, y)$ from $(0, 0)$ to $(sx, sy)$ do
24:             $dist \leftarrow$ distance from $(x, y)$ to $pos_{\text{x,y}}$
25:             if $dist < v$ then
26:                 $i \leftarrow p - (p \times \frac{dist}{d})^4$ // influence
27:                 $pathsMap[x][y]$ += $i$
```

can be seen in Figure 1, with high positive numbers denoting areas of higher repulsion (cost) when path-finding.

II. Influence Maps

Influence maps are data structures which are used to compute and store influence: a value that is typically used to either attract or repel specific behaviors to or from a given region within an environment. For example, a section of lava that damages a player may be given a repulsive value, while a health pickup may be given an attractive value. In our case, we wish to construct our influence map such that our paths will avoid areas of the environment that can be seen by enemy units, or reached by the range of the enemy unit weapons. We will then use these values as the cost function for our path-finding algorithm such that our units will minimize the amount of time that they spend near these areas.

Influence maps are commonly implemented as 2-dimensional grids/arrays which store the influence values located at that associated point of the environment. A point of interest will have some influence value, and that value propagates outward in the field. The values gradually fade to zero, or go to some cutoff - such as a value threshold or a distance. Influence is used to attract or repel inside the space, based on the values (positive repel, negative attract for example). Grid representations can be intuitively overlapped on a grid based game environment, as well as overlap with other influence maps to combine different influence information.

The the case of Starcraft, the build-tile resolution of a typical map is approximately 128 by 128 units in size, up to a maximum of 256 by 256. This means that we can construct an influence map as a 2-dimensional array with the same dimensions, and have it easily fit into the memory of a typical modern computer, with each cell in the influence map representing a build-tile region within the Starcraft map. An example of such a construction can be seen in Figure 1, which shows two enemy units imparting influence based on their vision radius.

A. Constructing the Influence Map

In order to construct the best influence map for totally avoiding enemy vision, we decided to combine three difference influence maps, each calculating influence for different metrics. These maps are as follows:

- **Vision Map**: Stores influence of seen (or previously seen) enemy vision radius. This map will help us avoid areas that can be seen by the enemy, giving us the best chance to surprise the enemy upon reaching the goal.
- **Damage Map**: Stores influence based on the damage that enemy units can deal to specific areas of the map. Given that we must eventually enter an area of enemy vision, we would prefer to avoid the areas that can be attacked by enemy units.
- **Common Path Map**: Stores influence based on paths calculated between bases on the map, which are often patrolled by enemy scouts. By avoiding these commonly walked paths, we may further avoid detection.

Individual influence values are calculated based on the following equation presented in [2]:

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

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.

III. A* Search Algorithm

A* is a popular search best-first search algorithm that continuously expands nodes in a priority queue until a goal is found, or everything in the graph has been seen. This priority queue is sorted as a function of a search node n, denoted $f(n)$, with:

$$f(n) = g(n) + h(n) \tag{2}$$

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. Given a consistent heuristic $h(n)$, A* is guaranteed to find a path that minimizes the total path cost. If the cost is a distance function only, then it will find the shortest path between two locations. However, 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. *Influence Maps as Cost*

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 compute the following values: $dist$ = the cost of traveling a given distance on the map, vis the influence cost of entering enemy unit vision, dam = influence cost of taking enemy damage, and $path$ = influence of traversing onto one of the map's common path tiles. The cost is the computed as a tuple, $c(n) = \{dist, vis, dam, path\}$, which is sorted based on the following priority, in order of higher amount of repulsive influence: $vis > dam > path > dist$.

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. This means that the resulting paths will be constructed with avoiding the enemy vision as the highest priority, and path distance being the lowest priority. The resulting paths will attempt to minimize travel time / distance (influence costs at that location are zero), while being sure to avoid areas where the drop ship may be seen or damaged by the enemy army (influence costs at that location are non-zero).

IV. Methodology

In order to use these influence maps with Starcraft, we built a system that integrates with a live Starcraft game being played by UAlbertaBot [3] using BWAPI [4]. The program reads relevant information from Starcraft and UAlbertaBot every frame, and uses that information to fill out the influence maps. For visual reference, it also has a visualizer which draws the game environment as a 2D grid, which can be seen in many Figures in this paper. Everything that is relevant to navigation is stored, such as walls, walkable tiles, player units, enemy units, and last known enemy unit locations. Each frame of the game, the influence maps are recalculated based on the updated positions of the units in the game.

The algorithm for each influence map can be seen their respective commented section in Algorithm 1.

- The common paths are found at the start of a game and stored, as these do not change throughout the game. The paths are found with a simple A* search from the player's base location to every possible enemy base location. The paths influences are calculated by propagating influence from each node along the paths, using the vision radius of a worker unit as the radius.
- The enemy vision influence map requires recalculations often as enemy units are constantly moving. For every enemy unit that has been seen by the player, influence propagates from their positions until reaching the end of their vision radius.
- Similar to the vision map, but checks enemy unit weapon ranges instead of vision radius, and the influence is a static number based on the damage of the air weapon.

V. Experiment

We tested our system by performing a drop strategy using UAlbertaBot, with the path taken by the drop ship being calculated using the proposed method. An example sneak-attack drop ship path can be seen on the right hand side of Figure 2, with a more standard distance minimzing path being seen on the left. You can see that this path (in yellow) avoids enemy vision by turning outside the vision radius of enemy units, and proceeding to sneak around to the back of the base where the enemy is most vulnerable. Another example of this can be seen in Figure 3, where the generated path also avoids commonly used paths in the game between the bases. So far, this system has worked in generating paths that are similar to those produced by humans when attempting similar strategies.

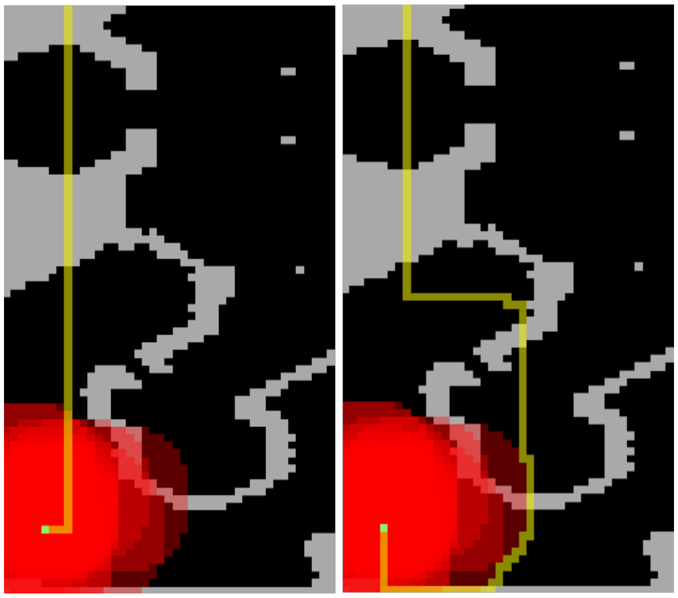
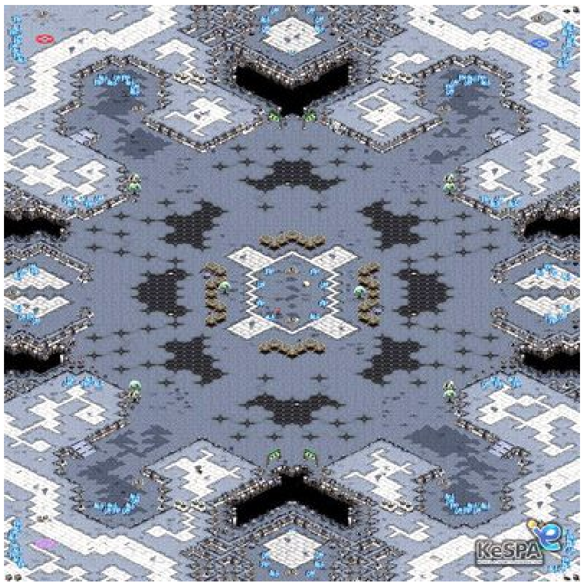


Fig. 2: Comparison of normal path and avoiding enemy vision path

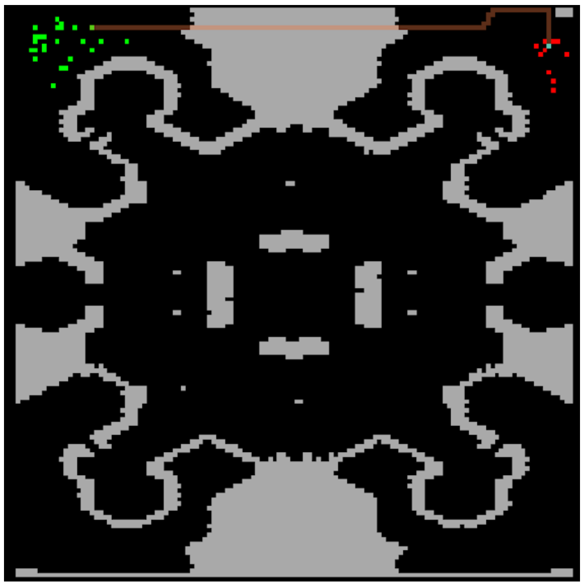
VI. Future Work

As this research is still in progress, we plan on making several improvements to the current system, as well as performing experiments to demonstrate the effectiveness of these new paths over those which already exist within UAlbertaBot. The main experiment we wish to run is to test whether the inclusion of this new system can improve the performance of existing agents (such as UAlbertaBot) in a competition setting. We wish to re-run the previous year's AIIDE Starcraft AI Competition with both the previous version of UAlbertaBot, and new version which includes the proposed sneak-attack system. We can then tell whether or not this system helps improve the overall performance of the bot in the competition by comparing its performance the the previous version.

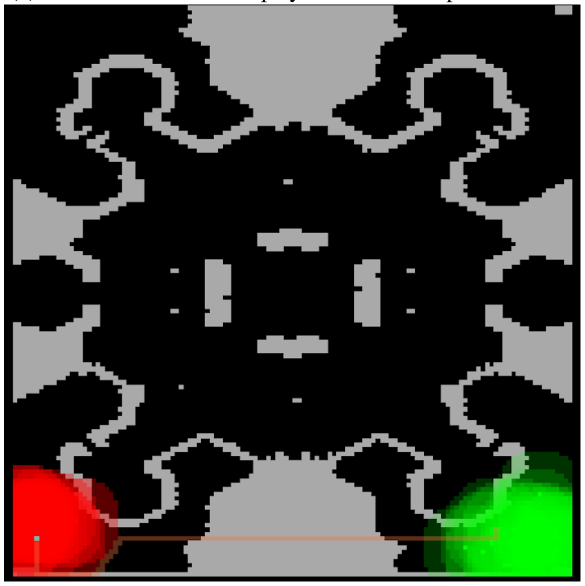
We also wish to test whether influence maps can be used exclusively, without the need for using another algorithm such as A*. By including distance as an influencing factor, we may be able to add an attractive influence toward the enemy base, and simply navigate toward the areas of highest influence. This method however has not yet been implemented, and remains for future testing.



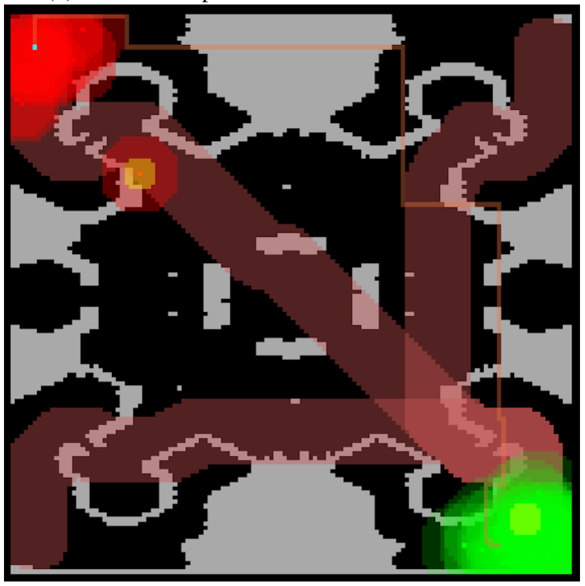
(a) Terrain view of the 4-player Starcraft map: Andromeda



(b) Starcraft map with no influence fields visible



(c) Starcraft map with vision field visible



(d) Starcraft map with all influence fields visible

Fig. 3: Three views of Starcraft maps 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).

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

[3] 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.

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