Chokepoint Detection and Wall-In Building Placement in StarCraft

Collin Riggs

Department of Computer Science

Memorial University

Supervised by Dr. David Churchill

A dissertation submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Science (Honours) in Computer Science.

April 2023

Abstract

A common concept in RTS games is a “chokepoint”, a narrow unavoidable section of the game world. In StarCraft, chokepoints are obstacles that restrict the flow of units as they travel throughout the map. A fundamental component of defense in StarCraft is blocking these chokepoints near your own resources. Given the small traversable area inside a chokepoint, it can be more efficient to use the surrounding natural untraversable geometry of the world to protect you, and reinforce only these chokepoints with defenses, creating the same level of defense that blocking open space would while consuming fewer resources. We attempt to locate the most vital chokepoints near a player’s base, and then determine a strategy for placing defensive structures to efficiently restrict enemy access through this chokepoint. After calculating optimal building placements, we can show that our solution greatly restricts enemy access to the player’s base with minimal resources, increasing safety of the players resources, while also accommodating specific strategies that will allow the player to continue moving their own units through the defended chokepoint.

Acknowledgements

Thank you firstly to my supervisor Dr. David Churchill, for his aid along the way when I could see no solution. His courses I took earlier in my studies were the inspiration for me wanting to focus my project on game AI.

Thanks must be given to my friends, who were also my peers throughout the completion of my degree. While not working on my project with me, they always offered insight wherever they could and listened to my frustrations when I could not solve a problem.

Lastly, I could not have finished this without my family. Their continued support even throughout the times when I thought I would surely not complete my work kept me pushing and without them, I am sure you would not be reading this now.

Table of Contents

List of Figures

List of Algorithms

Chapter 1:

Introduction

Real Time Strategy (RTS) games are difficult to solve with artificial intelligence because they are often too complex to have an explicit solution. Without a guaranteed method through which a computer could play the game, the best we can create is a good solution based on the strategy we have learned from the experience of real human players. StarCraft [1] is no different. While certain strategies may be popular, there is no perfect way to ensure you win every time.

In this instance, we decided to focus on a narrow aspect of StarCraft strategy. Defending a 'base' in StarCraft, where the players' main resources and infrastructure are located, is a primary goal, as an enemy's goal is to destroy your base so they can win the game. While it would be possible to surround one's base in defensive structures, doing so would hamper a player's progression by using valuable time and resources to build defenses, reducing their chance of winning. It is the theory then that the natural terrain of the StarCraft map can be used to aid with defending a player's base. Finding a way to use the natural terrain to a player's advantage is our goal, which can be done by finding chokepoints within the map's terrain and then efficiently blocking them. By utilizing this strategy effectively, this could be adapted into competitive AI programs for new or existing StarCraft computer bots, making this solution worthy of investigation.

In chapter 2, we describe the relevant details of StarCraft, as well as the visualization program used as a medium to demonstrate our solution. Chapter 3 describes the implementation of StarDraft as a visualization tool and how it is used to power our stepwise algorithm. Chapter 4 contains the results of our algorithm in various real StarCraft maps, as well as other strategies used

during development of this algorithm. In Chapter 5 we summarize the results of this algorithm and present possibilities for future applications of this algorithm.

Chapter 2

Background

2.1 StarCraft

StarCraft is a video game produced by Blizzard Entertainment released initially in 1998. Sequels have been made since, but the core game concepts have remained the same. StarCraft falls under the real time strategy genre, where players perform actions in the game simultaneously, as opposed to a game like chess where players take turns playing. At the beginning of the game, players have their base, and some units. Units are movable entities that can perform various tasks at the player's command, such as gathering resources or attacking the enemy. Throughout the game, each player will build more units, construct more building structures in their base, and attack the enemy with their units. The game world, also referred to as the map, is filled with terrain that disallows moving and building. Players will navigate their units through the map to travel to the base of the enemy players, and then destroy the enemy base, securing a victory in the game. To allow a computer to act as a player in a game of StarCraft, developers write ‘bots’, a computer-controlled player which interfaces with the game throughout play, allowing the bot to understand the game world and to take actions inside it.

2.2 StarDraft

StarDraft [2] is a visualization tool developed by Dr. David Churchill, which can display real StarCraft maps in simple two-dimensional top-down graphics.

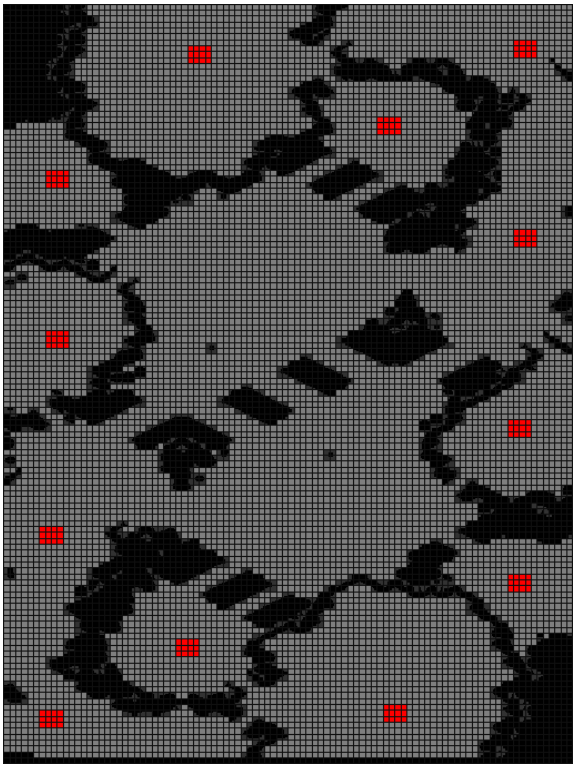

Figure 2-1: StarDraft program. Blocked tiles shown in black, bases shown in red

Eliminating the extra interfacing between the developed algorithm and the base StarCraft game, this visualization tool makes comparing results from the algorithm easier and reduces complexity of the system. While the applications of our algorithm will function identically when directly implemented into a StarCraft bot, StarDraft is used for simplicity and generalization.

2.3 Chokepoints

A chokepoint is a narrow route which allows movement from one region of space to another.

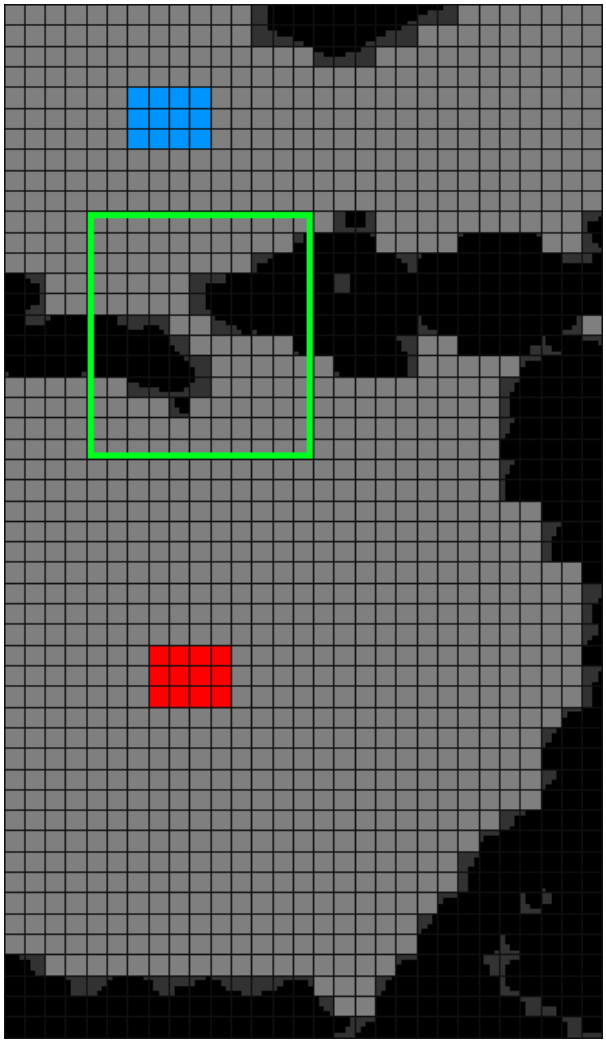

Figure 2-2: Chokepoint in StarDraft. Player's base shown in red, other base shown in blue, chokepoint area shown in green

They are prevalent across StarCraft maps and restrict the flow of units between areas. On most maps, each player's base will have a singular chokepoint as the access point to the base. This will limit the speed that a player can send their own units out of their base, but the more relevant aspect in this case is it limits the speed enemies can send units towards a player's base when they are attacking.

2.4 Specific Strategy

The specific strategy we have implemented is walling-in, where the natural terrain is supplemented by constructed defenses to create a complete barrier. To do so efficiently, we have designed an algorithm that will find the chokepoint around a player's base that the enemy must enter through during an attack. With the location of this chokepoint, then critical locations around the chokepoint are found, locations where creating buildings would block the enemy from entering the player's base.

By limiting the necessary space that needs to be blocked, the fewest number of buildings can be used to offer the same amount of protection as building an entire wall around a base. This minimalistic approach to defense allows resources to be allocated towards other goals in the game, creating a better chance for victory.

Chapter 3:

Methodology

3.1 StarDraft Initialization

As previously stated, the StarDraft program was used as the medium for testing this algorithm, as StarDraft mirrors the StarCraft map without unnecessary graphics and three-dimensional projections. When a user in StarDraft selects a map, after it has been loaded our program is run. After the program has finished, the results are displayed on top of StarDraft's standard interface. The locations for each player's starting base, as well as a map of the StarCraft world which denotes which areas are traversable and not are provided by the existing StarDraft library. StarDraft segments the game map into a two-dimensional tile grid, where each tile is marked as not traversable if there are any movement-blocking objects, such as buildings or natural terrain, inside the area that tile represents.

3.2 Chokepoint Detection

The overall strategy used to detect chokepoints in our tile map is to find areas in the map where a square object of some size would not be able to fit due to blocked tiles. Since the size of chokepoints can vary between maps, the size of the square object is not always constant, and using the wrong size could result in the chokepoint not being found or false positive results. To combat this, our algorithm looks specifically for the chokepoint nearest a player's base.

The beginning of the algorithm starts with a breadth-first search around the player's base. Expanding from the player's base, vacant tiles are explored, the search not continuing into or past

blocked tiles. At each step of the search, a list of tile positions at the current maximum depth of the search is stored. These positions are the furthest locations one could possibly reach from the player's base.

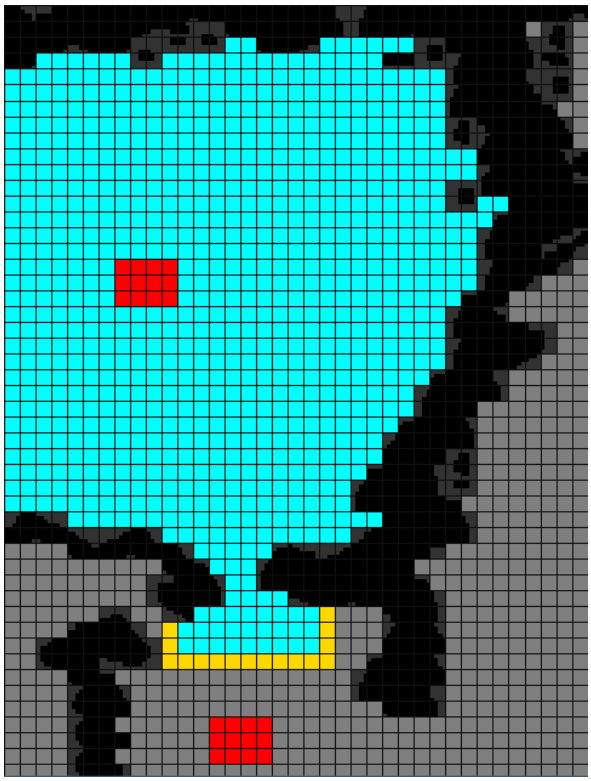

Figure 3-1: Maximum search distance. Searched tiles in blue, maximum depth tiles in yellow

Once the depth limit for the search has been reached, any of the tile positions at the maximum search depth is selected and a path is traced back from that tile to the player's base. The resulting path is one from the player's base to the furthest distance a unit could possibly reach from the player's base. Along this path must be the chokepoint, because to exit the player's base, a unit must travel through the base's chokepoint.

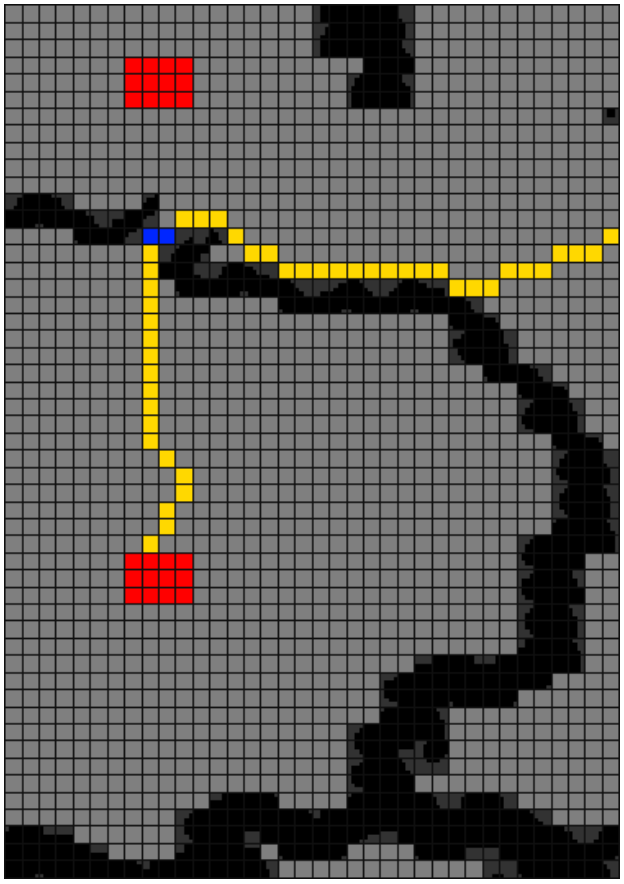

Figure 3-2: Calculated path. Path shown in yellow, chokepoint in blue, bases in red

To find where along the path the chokepoint is, the path is traversed repeatedly, testing collisions with objects of increasing size. The fact that a path was found through this chokepoint means that any object with the size of one can pass through, so the search begins with a square object of size three increasing by two at each iteration. For each tile along this path, if the object collides with blocked tiles on multiple sides, then it is where the chokepoint is located. The blocked tiles must be on opposite sides, because in instances where the path travels along a wall, a false chokepoint will be detected next to walls. Checking for opposite side collisions removes walls being detected as false positives. In larger chokepoints, a small object will not detect the presence of a chokepoint, so the size of the collision-detecting object is incremented until a chokepoint is found.

With a tile inside of the chokepoint found, a breath-first fill function is used to mark the rest of the chokepoint, connecting any adjacent tiles where the collision-detecting object finds blocked tiles within its bounds. The path that was used to detect the chokepoint may not travel through the centre of the chokepoint. There are cases where it can travel along the side, so finding the full size of the chokepoint is imperative to being able to completely block it. An alternate method to detect chokepoints is discussed in section 4.1.

Algorithm 1 Chokepoint Detection

```
(x, y) ← Player base location
(sx, sy) ← Size of StarCraft map
distanceMap[][] ← map[][] of size (sx, sy) each value is distance from (x, y) calculated with BFS search
maxDepthLocation ← select any maximum tile from distanceMap
path[] ← empty
pos ← maxDepthLocation
do
    pos ← adjacent tile with minimum value in distanceMap
    append pos to path
while pos is not (x, y)
reverse order of path
chokepointPosition ← (-1, -1)
for r ← 1..3 do
    for pos in path do
        if square with side length 2r+1 overlaps with blocked tiles on opposing sides then
            chokepointPosition ← pos
            break
```

3.3 Critical Locations

To block a chokepoint with the minimum number of buildings and resources, the number of tiles that need to be blocked should be minimal as well. A breadth-first search function is used again, as in section 3.2, but this time will not expand into the area of the chokepoint. This search will find the tiles around the chokepoint near the player's base. These tiles are then added to a list for storage, knowing that if buildings are placed such that each of these tiles is blocked, then units will not be able to move through this chokepoint.

Algorithm 2 Critical Blocking Position Detection

```
(x, y) ← Player base location
(sx, sy) ← Size of StarCraft map
distanceMap[][] ← fill((sx, sy), -1)
stack[] ← [(x, y)]
distanceMap[x][y] ← 0
toBlock ← []
while stack is not empty do
    tile ← remove bottom element from stack
    distance ← distanceMap[tile]
    for each direction in cardinal directions do
        nextTile ← tile + direction
        if nextTile is valid and not blocked then
            n ← distanceMap[nextTile]
            if nextTile is in a chokepoint then
                append tile to toBlock
            else
                append nextTile to stack
                distanceMap[next] ← distance + 1
```

3.4 Building Configurations

To find the final building configurations needed to wall-in the chokepoint, the user feeds in a series of building dimensions for structures currently available to be built. The simplest solution would be to brute-force the most optimal way to place each building to block the chokepoint, ignoring the critical locations located in section 3.3. However, the computation time for a realistic set of building options would have been great, as well as an evaluation method for determining if a chokepoint was completely blocked.

The solution found was to iterate over each building option and find the building that could block the greatest number of critical tiles. The best option is placed, and the iteration is repeated until all the critical tiles have been covered by any of the placed buildings. This greedy solution minimizes the number of buildings needed to cover tiles, as at each step the building which would cover the most tiles possible will be selected. With these building positions located, the locations of where to place each building can be returned to the user, whether that be player or computer, and used to optimally wall-in their base.

Algorithm 3 Block Chokepoint

toBlock[] ← list from Algorithm 2

buildings[] ← program input

buildingInstructions[] ← empty

for each *building* **in** *buildings* **do**

    bestPos ← *-1*

    maxTilesBlocked ← *-1*

    for each *tile* **in** *toBlock* **do**

        if *tile* **is not blocked then**

            pos ← position in grid where building dimensions block *tile* and as many more unblocked tiles in *toBlock* as possible

            if # of tiles blocked in *pos* > *maxTilesBlocked* **then**

                bestPos ← *pos*

                maxTilesBlocked ← # of tiles blocked in *pos*

    append *(building, bestPos)* to *buildingInstructions*

    if all tiles in *toBlock* are covered by buildings **then**

        break

Chapter 4:

Results and Discussion

When executing our algorithm, the output produced is consistently a configuration with the most optimal solution. With most scenarios, there are a multitude of configurations that would provide the same result and efficiency, with minor differences in building locations, and each time we detect one of these optimal configurations. A complete evaluation of the usefulness of this strategy is not able to be calculated in the context of this project, but as discussed in section 5.1, this strategy could be implemented into a complete StarCraft bot to test its effectiveness. After running the algorithm on 36 maps, it correctly determined every chokepoint on 75% of them. The primary reason for failure was excessively large chokepoints, where concave sections of terrain would be detected as a chokepoint due to the large search radius during chokepoint detection.

While the execution time of the algorithm will be system dependent, the time to compute is insignificant on modern computers, with the largest maps requiring a fraction of a second to complete. The time was found to directly correlate to the size of the map, which aligns with segments of the algorithm which perform calculations over the entire map, thus explaining the time difference.

In each figure displaying results of the algorithm in various StarCraft maps, open tiles are represented in grey while blocked tiles are black. The chokepoints detected are represented in red, the critical defense locations are represented in white, and the building placements are represented in borders around the tiles the buildings would occupy. For these examples, an arbitrary set of building options was used as input, but in practical scenarios, the dimensions of structures a player

could build would be used as input. Computation time was measured on an Intel i7-9700k processor at stock clock speeds, with 3200MHz RAM.

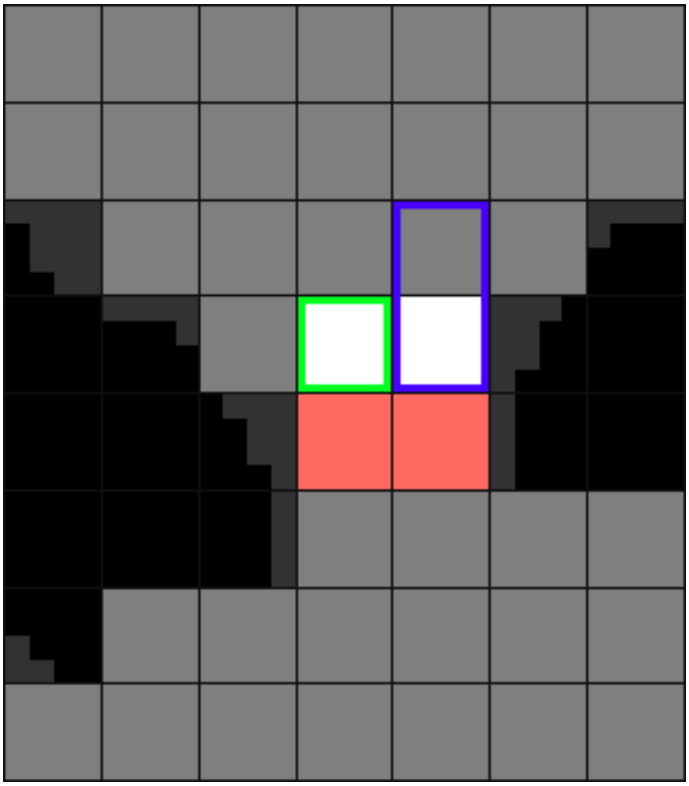

Figure 4-1: Result on ‘Heartbreak Ridge’ map. Computation time: 10053µs

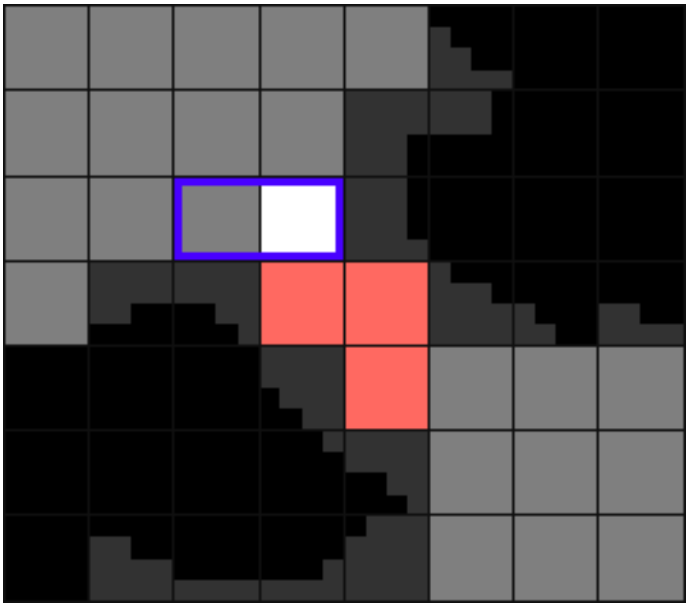

Figure 4-2: Result on ‘Destination’ map. Computation time: 10852µs

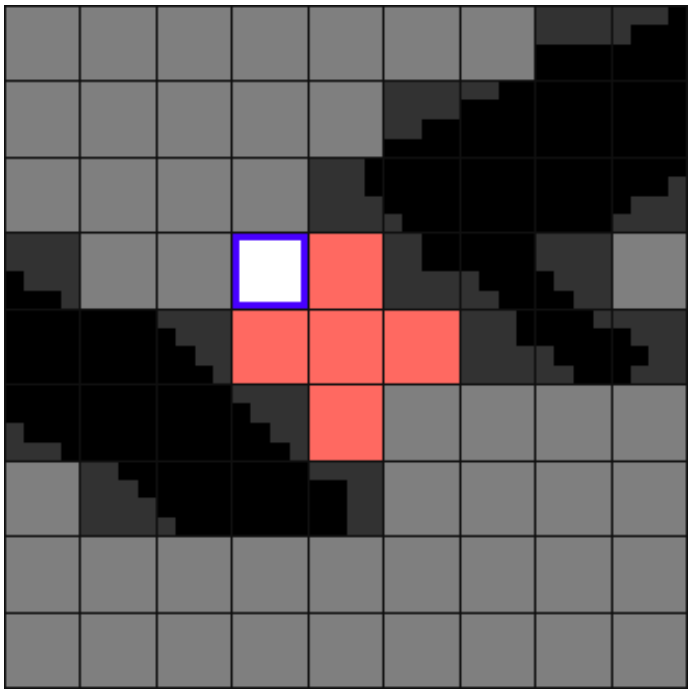

Figure 4-3: Result on ‘Circuit Breaker’ map. Computation time: 23791µs

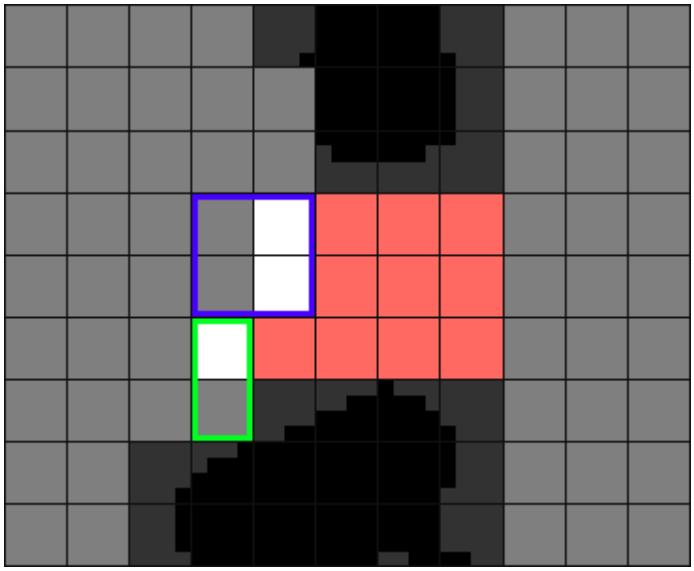

Figure 4-4: Result on ‘Empire of the Sun’ map. Computation time: 72406µs

The computation time in results from Figure 4-1 and Figure 4-2 differed so greatly from the results in Figure 4-3 and Figure 4-4 because the latter figures were four-player maps instead of the two-player maps in the lesser figures. As well, in the second pair of results the maps were an additional 33% larger, thus requiring more time to search through.

4.1 Alternative Chokepoint Detection

A strategy initially considered to perform chokepoint detection was rooted in the concept of ray casting, inspired by a similar implementation found in a StarCraft bot developed by Guillaume Docquier [3]. To detect chokepoints, rays would be cast from every tile on the grid in all directions. The rays would end when they collided with blocked tiles on the map. By comparing the distance values between perpendicular rays, chokepoints were found on tiles where the difference was large. Areas outside of chokepoints would have perpendicular distance difference values near zero. These results would create a rough estimate of where all chokepoints on the map were located.

This strategy was not selected for two reasons. Firstly, the need to make many ray calculations at every tile would require significant computation, compared to our final solution which searches for the nearest, and only relevant, chokepoints. While time efficiency was not a priority in designing this algorithm, future iterations of this project could be hampered by the volume of ray calculations. Secondly, the rest of the project uses breadth-first search methods as the primary tool for finding solutions, so reducing the number of technologies within the solution raised comprehension and readability.

4.2 Ballooning Strategy

The ‘ballooning’ moniker was given to this original concept due to the primary idea behind the solution, that by expanding the size of blocked tiles, chokepoints would become closed off. Such as a balloon would, the first step was to find every blocked tile in the map and ‘inflate’ it, increasing the radius of tiles that were blocked. The walls of the map terrain would expand into the area of chokepoints, meaning that bases were now completely enclosed areas. Using a breadth-first search method like in the real solution, the area of the reduced-bounds base was detected and marked. The map would then be ‘deflated’, where all the previously expanded blocked tiles would be

assigned to the adjacent base. By performing a breadth-first search from the base again on the original map, any tiles that were found in the search that did not belong to the base found in the first search would be the location of the chokepoint.

While this solution worked for some maps, the ‘inflation’ step of this approach was very susceptible to losing map information, by accidentally completely enclosing areas or completely smothering the player’s base in inflated tiles. This strategy proved promising at the beginning of development but was quickly considered to have poor future performance and thus was discarded.

Chapter 5:

Conclusion

In this project, we aimed to design an algorithm that would achieve the walling-in strategy in StarCraft. We described each step necessary to perform this task: how we can detect chokepoints near a player's base, find the critical defensive locations necessary to achieve the wall-in strategy, and find an optimal building strategy to cover these critical locations with minimal resources. In the scope of this project, it was not possible to calculate the exact effectiveness of using this strategy in a real game of StarCraft. We did prove that if this strategy is accepted as powerful, we are able to determine an efficient implementation of the strategy.

5.1 Future Work

This project implements the chokepoint detection and wall-in building placement algorithms but does not apply the strategy to real StarCraft AI bots. The next step in this project is to implement the algorithm into a playable bot to evaluate practical effectiveness of the strategy.

5.1.1 Multiple Chokepoint Detection

One drawback of the current solution is that it only works with bases that have a singular chokepoint. The few StarCraft maps that have multiple chokepoints per player base will find one of the chokepoints detected and blocked, while the other chokepoint will remain undetected and therefore undefended. An additional feature to be implemented in this project is the ability to detect and block multiple chokepoints.

5.1.2 Building Placement Goals

Without the capability to test the wall-in strategy in real games using this algorithm, it is uncertain whether the current greedy building placement optimization is the optimal strategy. Other goals, such as maximizing the number of buildings so the enemy must invest more resources into destroying all the buildings to clear the chokepoint, may provide different results, and needs to be tested in a full implementation of this algorithm in a bot.

Bibliography

[1] “StarCraft” (1997). Irvine, CA: Blizzard Entertainment.

[2] Churchill, D (2020) StarDraft [Source code]. https://github.com/davechurchill/stardraft

[3] Docquier, G (2022) Sajuuk-SC2 [Source code]. https://github.com/Guillaume-Docquier/Sajuuk-SC2