

## Describing your approach

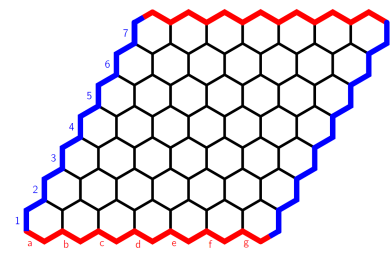
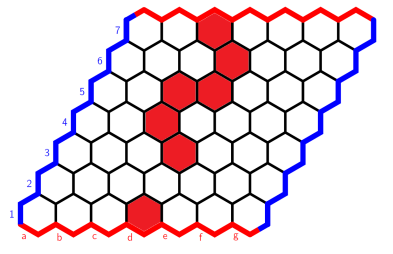
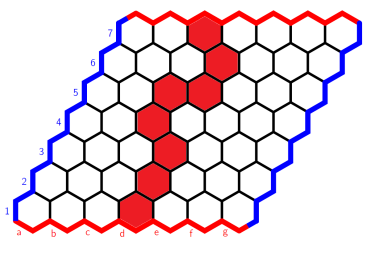
Our approach for our player was a mixture of the following goals:

1. Minimising the amount of pieces needed to be placed to win (win distance)
2. Maximising the other players win distance
3. Preferring stable states over unstable states (triangles, diamonds)
4. Expanding only nodes that would likely minimise win distances

### Minimising win distance

Win distance is the distance that a player wants to minimise. Initially it starts at  $n$  and the game ends when it reaches zero. We defined the win distance to be the minimum distance between any of the opposite nodes (top and bottom for red and left to right for blue)

**Table 1: minimum distances**

Min win distance = $n$	Min win distance = 1	Min win distance = 0
		

We discovered that the win distance is very computationally expensive to run, and so many different algorithms and arrangements were attempted to optimise.

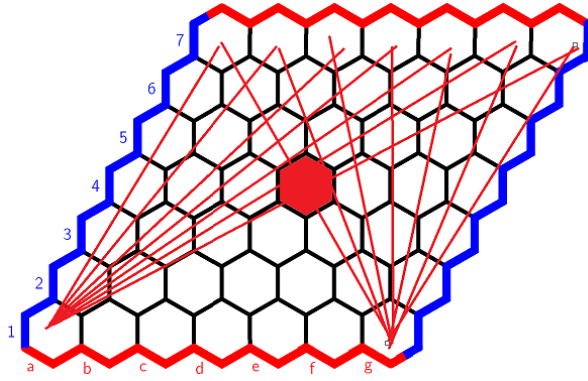
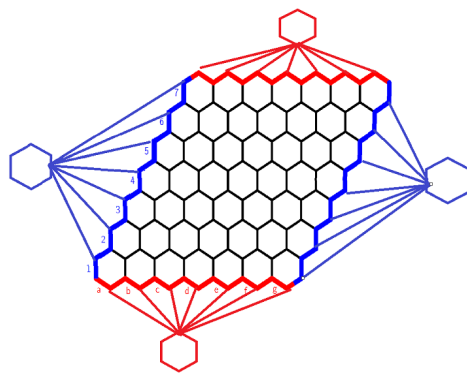
Originally whenever the minimum distance was needed we tried to compute it within a double for loop(see table 2 below), and then select the minimum distance and path.

This proved to be very inefficient as  $A^*$  has a runtime complexity of  $O(b^d)$ , which is then repeated  $b^2$  times, leading to a time complexity of  $O(b^{2d})$ . Caching was attempted, but the time complexity was still too long, around 1 minute to make a move for a 15x15 board.

To improve on this the graph was changed to include start and end nodes that artificially neighbour the bottom and top nodes respectively(table 2).

This meant that the same minimum result could be achieved in a single call to A\* with time complexity of  $O(b^d)$ , and furthermore results can be cached, such that a minimum distance is cached per colour so when the next action is taken, the cache only needs to be updated if the changed node is in the minimum path. Weighted A\* was attempted, but was ineffective at finding the desired nodes with the artificial neighbours method of finding the minimum distance.

**Table 2: A\* Optimisations**

Double for loop A* algorithm	A* artificial neighbours
	

The following is the benchmarking results for each of the algorithms tested. This is not strictly related to the gameplay performance, but the search algorithm optimization played a big part in the success of our implementation.

**Table 3: Benchmarking results for different search algorithms  
first 10 moves, n = 10, search depth=3, profiling enabled**

Search algorithm used	Time taken	Number of Calls
Floyd Warshall	> 5m	> 0
A* double for loop	233.53s	78495
A* optimised graph	54.31s	2146
A* optimised graph cached	29s	977

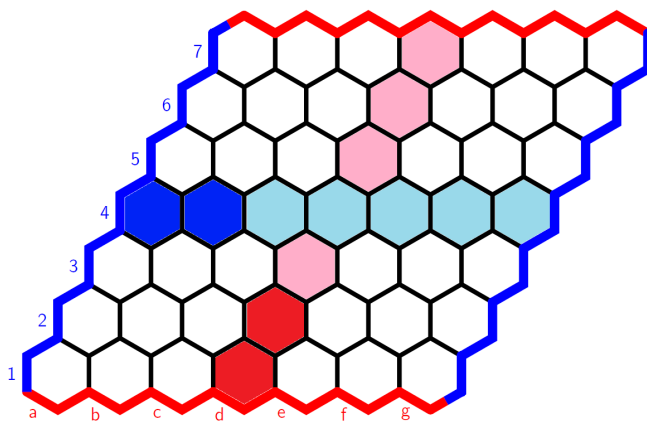
As can be seen in table 3 the search algorithm optimization took up a huge portion of the runtime of our program. Minor optimisations were made to other functions (removing slow list comprehensions, minor caching etc) but none were comparable to the efficiency gained in optimising the search algorithm.

### Maximising the other players win distance

To maximise the other players' win distance, and to minimise our own the minimax algorithm was used. Notably it only became reasonable to run the minimax algorithm on board sizes over 4 once the A\* optimisations above were added.

Minimax was implemented with alpha beta pruning, and the only nodes to be expanded are the ones that lie on the path of minimum win distance path for each player. The nodes expanded first are the ones that lie on the minimum win distance path as can be seen below in image 1.

**Image 1: Priority given to nodes along minimum distance**



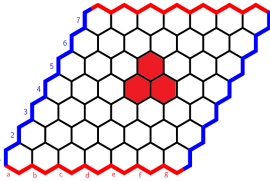
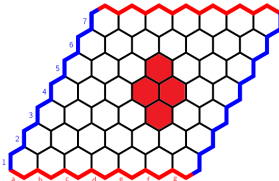
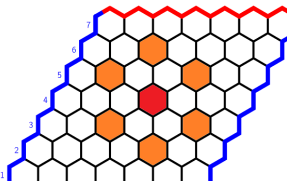
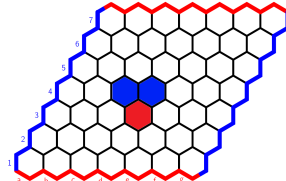
This algorithm works well because it gives preference to the board states that are more likely to occur. Even though this algorithm is sub optimal, as it disregards nodes that are not in either player's minimum win distance path, in practice the depth of the search is never enough for this to matter too much. In this case our player is sub optimal; if an unusual play is played against our player that takes a longer than usual path from one side to another our player will only be able to play defensively when the minimum win distance is that path. This problem is mitigated by the fact that any path is too long is not worth exploring anyway because either 1. Our player wins, or 2. The other player needs to play defensively which means that the need to explore many other possible paths is decreased.

## Preferred states

These states are local patterns that were researched and found to be beneficial for either side. Each one of these patterns is counted in our evaluation function, and then the weights were manually added when comparing different versions of the player.

Future optimisations could be applied to adjust these weights, possibly using gradient descent.

**Table 4: Preferred states**

Triangle	Diamonds	Double paths	Captures
Cannot be captured	Cannot be captured	Paths from red to orange can only be blocked with two or more opposite pieces	Red can be captured if blue places one piece
			
Weight: 0.2	Not added in final version	Weight: 0.05	Weight: 0.4

The outcome of this is that with distance being equal, states that are more stable will be preferred to than ones without stable states. In the final version diamonds were removed because the algorithm used was too computationally expensive. More optimisations could be done to add diamonds efficiently.

The weights of *Capture* were the highest because it led to more favourable outcomes; if we can capture a piece we extend our path at the cost of the other player. Triangles were an effective measure of how stable states were. Double paths seemed to be the least important of the patterns tested.

### The evaluation function

Once the number of preferred states are calculated, and the minimum distance is calculated the evaluation function is calculated as follows:

$$evaluation = triangles \times 0.2 + Double\_Paths \times 0.05 + Captures \times 0.4 + 10 \times \frac{1}{distance}$$

**Table 5: Evaluation function examples**

Distance	Triangles	Double paths	Captures	Eval function
10 = 1/10 * 10	2 * 0.1 = .2	2 * 0.05 = .1	2 * 0.4 = .8	2.1
1 = 1 * 10	2 * 0.1 = .2	2 * 0.05 = .1	2 * 0.4 = .8	11.1

## Performance evaluation

To evaluate our performance a couple of comparisons were made on a 9x9 board:

**Table 6: Evaluation of performance for game-playing program**

	Red Player	Blue Player	Expected	Winner
0	depth = 1	random=True	red	red
1	random=True	depth = 1	blue	blue
2	depth = 1	depth = 2	blue	red
3	depth = 2	depth = 1	red	red
4	depth=1	depth=3	blue	blue
5	depth=3	depth=1	red	red
6	distance	distance + patterns	blue	blue
7	distance + patterns	distance	red	red

### **Player vs Random (0, 1)**

Our player consistently outperformed a random player, this is because it has the ability to follow a path, as well as block opposite players

### **Depth (2, 3, 4, 5)**

Initially it seemed like our player of depth 2 does not always beat a player of depth 1 as is the case of N=2. This is rectified in N=3 as when the depth=2 goes first it wins against depth=1.

When the depths are 1 and 3 respectively (N = 4, 5), the depth=3 wins no matter what order the game is played.

Unfortunately more optimisation needs to be done for it to be efficient to run at more than 1 depth at n greater than 9.

### **Evaluation functions (6, 7)**

This shows that a distance heuristic to place pieces is worse than something that takes local patterns into account. Future optimisations to this could make this difference even more, especially with densely populated graphs.

Of these evaluations the one that was the most interesting to watch was the `capture` evaluation, which systematically disrupts the opposite sides pieces when possible.

## **Other Aspects**

Using gradient descent was explored in order to tune the weights referred to in table 4. This would have the effect of being able to see which attributes in our evaluation function are more or less important, as currently we manually tuned it in, but with 4 different attributes this is likely to be far from the true most optimal solution.

The tensorflow library was chosen to start implementation, and the ``tensorflow.keras.optimizers.SGD`` was going to be used, but for the scope of this project the resources required to run gradient descent were deemed too impractical, and there are other optimisations that could be made, namely around how the nodes get expanded, and minimising how often the A\* algorithm gets called.

Other improvements would come in the form of caching. Limited caching and memoisation was used in the implementation of this player, on a 15x15 Board only 0.25mb is used after 25 turns and this has the ability to greatly improve performance if board states were cached.

## **Supporting work**

Prior to working on the game-playing program, research was done on the game that Cachex was adapted from, Hex. Some strategies that were adopted into the game-playing program were things such as double bridges, the reason being these formations meant double threats which would allow a player to force his opponents hand into either using one move to block one side of the bridge and allowing you to resume the connection on the other side(Seymour).

For the A\* search, to find the shortest path, the algorithm was adapted from Hexagonal Grids.

## **Appendix**

1. "Hexagonal Grids." *Red Blob Games*, Oct. 2021,  
<https://www.redblobgames.com/grids/hexagons/#distances>.
2. Seymour, Matthew. "Basic Concepts - Hex: A Strategy Guide - MSEYMOUR.CA." *Hex: A Strategy Guide*, 2020, [http://www.mseymour.ca/hex\\_book/hexstrat1.html](http://www.mseymour.ca/hex_book/hexstrat1.html).
3. Yang, Jing, et al. *On a Decomposition Method for Finding Winning Strategy in Hex Game*.