COMP30024 Assignment 2

Tran Dao Le, Ohan Ra

May 11, 2022

1 Approach for adversarial search

The adversarial search strategy used in this agent is the orthodox mini-max search algorithms with alpha-beta pruning discussed in the lecture and cutoff-test at the depth of 2 on average. There were few modifications done on this algorithm to make the search more efficient.

1.1 Ordering of successor nodes

The successors at each level of the search tree is ordered in a specific way to optimise the number of pruned nodes. For max-player (our agent) the successors would be explored in order of decreasing evaluation function estimation. For min-player, the successors would be ordered in increasing order of evaluation function estimation. The estimation of evaluation is set to be the new Hex's distance from the path connecting two sides with the lowest cost to reduce computation time. This corresponds to exploring the nodes in order of best to worst for each player in the game. This way, the algorithm can ignore the rest of the successors once a evaluation value of successor for min/max player of a node at depth i is smaller/greater than the currently stored value of min/max player at depth i-2. In the best case, this is expected to prune about $O(n^d)$ nodes. On average it should be able to prune about $O(n^d/2)$ nodes

1.2 Restricting search space

From our own game experience of the game, we have derived that a threshold of 1 is the most effective, as this would prevent the game states being subject to having bridges (Shown in Figure 1). Bridges [Hex] can be considered one of the worst Hex placement in Cachex as diamond capture cannot be prevented. When we create the bridge, the opponent can place a piece between the bridge, and for any placement of us after that, the opponent can choose to capture our pieces.

For the game of Cachex, it maybe disadvantageous to place hexes far away from the cluster of occupied cells as this would reduce chance of diamond capture to prevent opponent from winning. Hence, the agent would only consider placing a hex within a threshold distance from an occupied piece as shown in Figure 2.

The number of successors that needs to be explored is now proportional to the number of occupied hex on the board, and can expect to reduce the search space by $O(n^2)$

1.3 Reducing successor state space for initial moves

The effect of initial move maybe minimal in Cachex, as long as the player is able to secure the first hex on the board. The agent would account for this factor by having a fixed move in the case of its first turn. If the agent is "red", it would place the first piece on a predefined neutral position to minimise the advantage of the opponent player stealing the hex. If the agent is "blue" it would always steal the opponent's hex that is within the range shown in figure 3. Fixing the initial moves of the agent helps to significantly reduce the search space and help increase the cutoff depth for early game by average of 1.

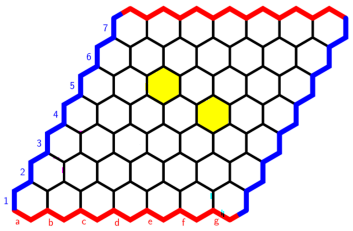

Figure 1: Example of bridge

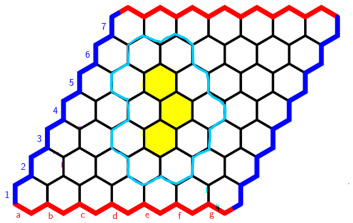

Figure 2: Yellow = the occupied cells, Blue Border = The range for possible successors

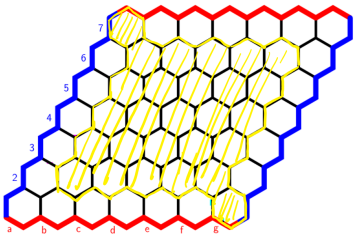

Figure 3: Range for steal action

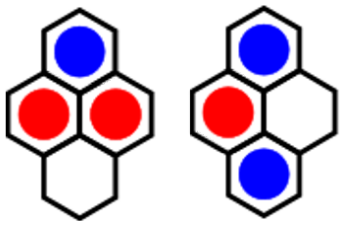

Figure 4: Patterns that can be captured

1.4 Changing depth according to game progress

Through various observation of game played by AI agents. It was observed that with the technique discussed in section 1.2, the number of successor states required for exploration increases as the game progresses and decreases again in late game.

The agent utilises this property, and is able to change the cutoff depth accordingly as the game progress. We have predefined the default depth for each board sizes based on our observations of the time required by agent to complete a game, and have defined the following function to change search depth accordingly.

- If n > 10 and the number of occupied cells < 6 or > n^2 - 6, increase depth to the default depth for board of size n+1
- Otherwise, stick to the default depth

This allows greater depth for early game and possibly late game, which increases the chance for agent to place itself in an advantageous position early, and possibly have enough depth when it becomes more important in the late game.

2 Evaluation Function

The evaluation function consists of following features and each of them are weighed according to its significance in estimating the value of each successor state

2.1 Difference in minimum number of pieces required to connect two sides

The minimum number of pieces required to have a connected path from one side to another is derived using A* search. The A* search uses the heuristic (introduced in our Part A Report):

$$h(hex_1, hex_2, ourOccupiedCells) = |r_1 - r_2| - occupiedRowsBetween(hex_1, hex_2, ourOccupiedCells)$$

This ensures that the search algorithm would pick the lowest cost path even if the path may be a detour. The starting will always be the first row/column of the board according to the player colour, and the goal is set to the last row/column of the board. All occupied cells of player's colour is set to have 0 cost in A* search and all unoccupied cells would have a cost of 1. This guarantees to be optimal, and would return the cost of lowest cost path in joining the two opposite sides. The agent would calculate this for both players, and take the difference to be one of the features for its evaluation function. If the agent is able to have a win state for the successors in the next step, we set the value of this function to n^3 to ensure that evaluation function is the greatest.

This feature was chosen due to its likelihood of being representative of which player is closer to their winning state. The bigger the value, we can naively determine that the agent has more advantage in the game than the opponent, and vise versa. This also somewhat accounts for possible diamond captures on the opponent's lowest cost path, since reducing the opponents piece by 2 would increase the difference more than the agent simply placing a piece on the lowest cost path.

The weight of this feature is set to 1.

2.2 Number of Own pieces - Number of opponent pieces on board

Number of occupied pieces on the board is counted for each player. And the difference is taken as second feature of the evaluation function. This feature was chosen because it is often advantageous for the player to have greater amount of pieces than the opponent, as it reduces the effect of being captured, and increases the effect of capture.

This feature favors states where the agent is able to capture the opponent and avoid states where the agent is being captured, as the value would be more significant. Therefore, this single feature can be portrayed as the combination of features that corresponds to:

1. The number of opponent pieces that can be captured
2. The number of own pieces that are prone to capture

Combining these two features may increase the precision of the evaluation function in determining the how advantageous a state is, with lower cost of computation required for calculation in comparison to having two features.

The weight of this feature is set to 1.

2.3 Number of patterns on board for potential diamond capture

Due cutoff depth having an average depth of 2, there is need for a feature that helps to evaluate states beyond the reach of the adversarial search. The features discussed in section 2.1 and 2.2 may ignore the patterns shown in Figure 4 if we stop at the leaf with these patterns. This feature helps the agent to find possible captures or prevent the captures other than those immediate ones to increase flexibility of the agent.

The calculation involves the finding of all diamonds for each of the occupied cells which has 3 pieces already and can form a capture.

The weight of this feature is set to +2 for each diamond we can capture and -2 for each we can lose the pieces.

3 Performance Evaluation

We test our game on dimefox against 3 agents: a random agent, an agent with the same evaluate function but just running at depth 1, our own agent. The result is shown in Figure 5.

For the random agent, as the behaviour is random, the pieces can be sparse, and increase our search tree branches. Therefore, the runtime and number of moves really varies. We managed to win in all the matches versus the random players.

For the greedy player, we managed to win most of the time. For larger board (n ¿ 10), our agent also play at depth 1, which causing the draw to happen.

When playing against our own agent, most of the time, the blue player is winning. Since we always steal the first move (which are defined to the agent by hand), this is a sign that our strategy for the first move is making an advantage when playing with a player at the same level. For larger board, the draws are caused by the same reason as playing with the greedy agent.

All the depth for each board size is set by hand after testing against the above agents to make sure the game can happen in the time constraint.

All the agents used above are provided in the module along with this report: `random_1` for the random agent, `we_have_no_idea_1` for the greedy agent with depth 1 and our own agent `we_have_no_idea`

4 Other aspects of optimisation

4.1 Heap queue for A* search (Section 2.1)

Our agent only need to find one path with the lowest cost on the board. Heap queue would reduce the computation time required for the search since the data structure is similar to stacking where the most recent node would be returned in the case of equal priority. This often would prioritise the exploration of the node that is closest to the goal, and hence fasten the time required for search. Furthermore, in the case when there are many pieces on the board, the performance of A* search will depreciate to those of uninformed search, with all priority in the queue being equal. Stacking would allow the algorithm to mimic the behaviour of DFS in the worst case scenario. Since BFS would explore all the cells on the board, DFS may requires less time to reach the goal.

References

[Hex] HexWiki. Basic (strategy guide).

Random

	3	4	5	6	7	8	9	10	11	12	13	14	15
Red	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win
Red time	0.189	2.265	4.568	3.233	19.429	9.666	14.357	4.159	10.873	22.376	55.89	99.828	85.214
Turns	5	7	9	17	31	19	25	23	29	43	61	69	55
Blue	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win	Win
Blue time	0.157	1.304	7.373	5.237	7.273	15.364	45.227	4.175	13.877	46.842	18.97	25.346	55.014
Turns	5	8	18	18	20	26	38	22	40	60	36	38	54

Greedy

	3	4	5	6	7	8	9	10	11	12	13	14	15
Red	Lose	Win	Win	Win	Win	Win	Win	Lose	Draw	Draw	Draw	Draw	Draw
Red time	0.747	2.215	3.043	5.347	7.48	18.268	17.758	10.659	23.076	28.769	25.419	44.96	54.522
Turns	8	9	11	39	43	51	55	78	108	108	108	108	108
Blue	Win	Win	Win	Win	Win	Lose	Win	Win	Draw	Draw	Draw	Draw	Draw
Blue time	0.914	3.41	2.173	3.213	5.451	43.566	10.967	10.838	23.399	31.126	38.48	49.344	60.332
Turns	10	10	12	24	32	77	34	78	90	90	90	90	90

Auto

	3	4	5	6	7	8	9	10	11	12	13	14	15
Winner	Blue	Blue	Red	Blue	Blue	Red	Blue	Blue	Draw	Draw	Draw	Draw	Draw
Red time	0.977	4.52	5.245	7.24	31.195	9.7	49.413	10.603	22.613	29.09	37.307	45.902	57.262
Blue time	0.486	3.756	3.391	7.25	31.612	11.699	33.938	10.758	23.302	30.016	38.886	48.096	60.041
Turns	6	18	13	42	94	27	110	78	90	90	90	90	90

Figure 5: Test result