Einstein Würfelt Nicht Agent

CSIE5138 Theory of Computer Games Final Project

R11922047 朱紹瑜

1. Introduction

Einstein Würfelt Nicht (EWN) is a two-player board game with complete and perfect information. Players take turns to move their cubes based on the dicing results with the goal of reaching the opposite corner or capture all the cubes owned by the other player. Unlike deterministic games, the dice introduce randomness into EWN, which makes the typical mini-max inapplicable for solving EWN.

In this project, we built an agent of Einstein Würfelt Nicht. To deal with the randomness, we introduced priori chance nodes into our search algorithm. The search is done with the Star1 algorithm with NegaScout. Other techniques include bitboard, transposition table, and iterative deepening aspiration search. Details can be found in the following sections.

2. Algorithms and Heuristics

2.1 Evaluation Function

Our evaluation function refers to the offensive and defensive view introduced in An offensive and defensive expect minimax algorithm in EinStein Würfelt Nicht! (2015 CCDC). **Our evaluation function consists of three parts: *Attack*, *Block*, and *Threat*.

Attack

First, we define the ***mobility*** of an existing cube to be six times the probability that we can move it after the next dice roll. If a cube has been captured, its mobility is 0. Take the board configuration in Figure 1 for example. Assume that it's the red player's turn. The mobility of red cube with index 3 is 3 since we can move it if the outcome of the dice roll is 3, 4, or 5. The mobility of red cube with index 1 is 0 since it has been captured.

Secondly, we define the ***positional value*** of an existing cube to be 2^{5-d}, where d is the shortest distance between its current position and the opposite corner. If a cube has been captured, its mobility is 0. For example, the red cube with index 3 in Figure 1 can reach the opposite corner in two moves by going diagonal then vertical. Therefore, its positional value is $2^3 = 8$.

Let ATT_p denote the player p's Attack. And let $M_{p,i}$ and $P_{p,i}$ denote the mobility and the positional value of the player p's cube with index i, respectively. The formula of ATT_p is as follows:

$$ATT_p = \sum_{i=1}^{6}(M_{p,i} \times P_{p,i})$$

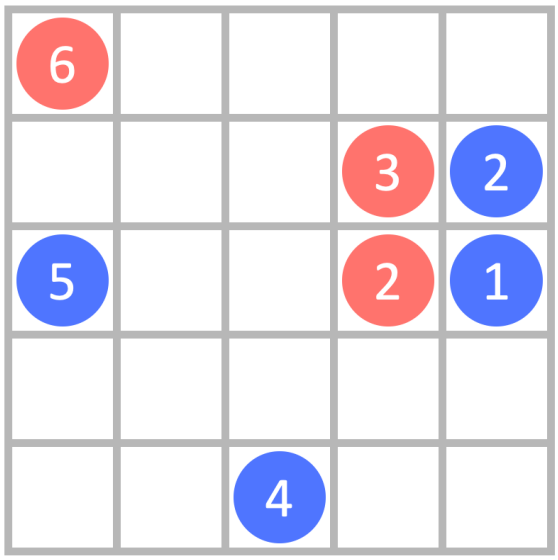


Figure 1: A board configuration example.

Block

A player's Block equals to the his/her opponent's Attack.

Let BLK_p denote the player p's Block. And let $\neg p$ denote the player p's opponent. The formula of BLK_p is as follows:

$$BLK_p = ATT_{\neg p} = \sum_{i=1}^{6}(M_{\neg p,i} \times P_{\neg p,i})$$

Threat

First, we define the ***maximum threatening value*** of an existing cube c as the maximum positional value among all its opponent's cube that c can capture. If a cube has been captured or cannot capture any opponent's cube in the next move, its maximum threatening value is 0.

For example, the blue cube with index 1 in Figure 1 can capture the red cubes with index 2, 3, and the blue cube with index 2. Here we ignore the blue cube with index 2 since it is also owned by the blue player. The maximum threatening value of the blue cube with index 1 is this $\max 2^2, 2^3 = 8$.

Let THT_p denote the player p's Threat. And let $MT_{p,i}$ denote the maximum threatening value of the player p's cube with index i. The formula of THT_p is as follows:

$$THT_p = \sum_{i=1}^{6}(M_{\neg p,i} \times MT_{\neg p,i})$$

Combine Attack, Block, Threat

Let w_1, w_2, and w_3 be the coefficient of Attack, Block, and Threat. The evaluation function from the player p's view is as follows:

$$f(p) = w_1 \cdot ATT_p + w_2 \cdot BLK_p + w_3 \cdot THT_p$$

In our implementation, we have $w_1 = 2$, $w_2 = 1$, and $w_3 = 0.05$.

2.2 Transposition Table

During the NegaScout game tree search, we may reach the same position by more than one path. By using a transposition table, we can leverage the result of previous search on the same position. To ensure an efficient table lookup, we use hash tables to store the transposition table. For each entry, the key blends the information of the board configuration, the next player, and the dice outcome with the Zobrist's hash function. The value is a tuple of the form $(depth, alpha, beta, value)$.

Zobrist's Hashing

First, we come up with three sets of random numbers as follows:

1. Board configuration: 12×25 random numbers in the format of $B[cube][location]$
2. Next player: 2 random numbers in the format of $P[player]$
3. Dice outcome: 6 random numbers in the format of $D[num]$

Given a position pos with the next player p, dice outcome d, and x cubes, where q_i is the ith cube at location l_i. The hash key of the position pos is as follows:

$$hash(pos) = P[p] \otimes D[d] \otimes B[q_1][l_1] \otimes \ldots \otimes B[q_x][l_x]$$

Using Information From the Transposition Table

It is not always the case that we can reuse the value retrieved from the transposition table. The following pseudocode illustrates how we leverage the retrieved value when there is a hash hit.

Let `t_depth`, `t_alpha`, `t_beta`, `t_value` be the search depth, alpha value, beta value, and the search result of the previous search. And let `depth`, `alpha`, and `beta` be the search depth, alpha value, and beta value of the current search.

```
if t_depth < depth:
    ignore the retrieved value
else:
    if t_alpha < t_value < t_beta:
        it should be an exact value, use it directly
    else if t_value >= t_beta:
        this introduces a lower bound, set alpha = max(alpha, t_value)
    else if t_value <= t_alpha:
        this introduces an upper bound, set beta = min(beta, t_value)
```

2.3 Iterative Deepening Aspiration Search and Time Control

It is seldom the case that the evaluation value largely increase or decrease with 1 or 2 additional depth. Therefore, we can iteratively deepen our search and use the evaluation value of the shallow ones to modify our alpha/beta bounds. The process is as follows:

Assume that there are x legal moves.

Let `score[i]` be the evaluation value of the board after applying move i.

Let `MIN_DEPTH`, `MAX_DEPTH` be the minimum and search depth, respectively. And let `THRESH` be the window size.

```
for i = 1 to x:
    score[i] = Star1(move[i], -inf, inf, MIN_DEPTH)

next_depth = MIN_DEPTH + 2
while the remaining time is enough for searching the next depth:
    new_score = Star1(move[i], score[i] - THRESH, score[i] + THRESH,
next_depth)
    if new_score <= score[i] - THRESH:
        new_score = Star1(move[i], -inf, new_score, next_depth)
    else if new_score >= score[i] + THRESH:
        new_score = Star1(move[i], new_score, inf, next_depth)
    score[i] = new_score
    next_depth = next_depth + 2
return move[j] where move[j] is the largest among all move[i]
```

Moreover, we set `MIN_DEPTH` and `MAX_DEPTH` to be different for each game stage. For the open game, we set `MIN_DEPTH = 2` and `MAX_DEPTH = 4`. For middle game and end game, we set `MIN_DEPTH = 2` and `MAX_DEPTH = 6`.

3. Experiments and Results

3.1 Search Depth

We compare the performance and time usage of each fixed depth agent. Each agent play against the Depth 0 agent for 20 games. The results are show in the following table.

	Depth = 0	**Depth = 1**	**Depth = 2**	**Depth = 3**	**Depth = 4**	**Depth = 5**	**Depth = 6**
Win Rate	0.45	0.7	0.45	0.65	0.75	0.7	0.8
average time usage (sec)	0.00	0.01	0.05	0.29	2.06	16.19	134.02

From the table, we can see that the average time spent for a complete round increases exponentially as the search depth increases. As for the win rate, agents that search deeper usually performs better than those that search shallower. However, the win rate is not strictly increasing.

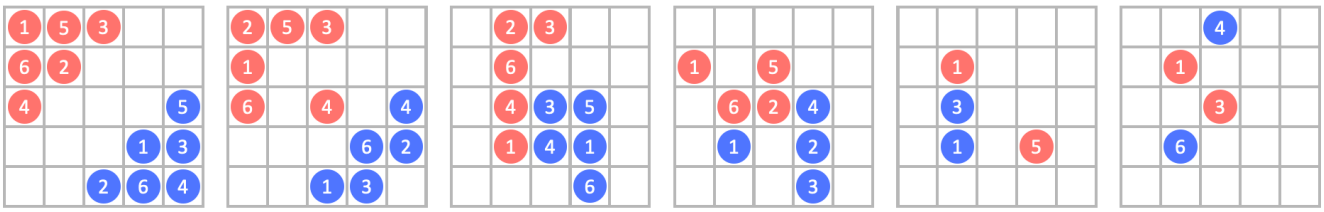
3.2 Star0 vs. Star0.5 vs. Star1

In this project, we implemented 3 different chance node search algorithms as follows:

- Star0: chance node search with exhaustive enumeration without cuts
- Star0.5: chance node search with cuts in between choices

- Star1: chance node search with cuts inside choices using bounds from an arbitrary move ordering

To compare the performance of each algorithm, we take 2 open game board configuration, 2 middle game ones, and 2 end game ones. We set the depth to be fixed at 4, disable the transposition table, and compare the leaf nodes in the search tree.



Open game 1 | Open game 2 | Middle game 1 | Middle game 2 | End game 1 | End game 2

	Open game 1	Open game 2	Middle game 1	Middle game 2	End game 1	End game 2
Star0	4655928 (0%)	6031164 (0%)	16156128 (0%)	7547142 (0%)	2320152 (0%)	477828 (0%)
Star0.5	4654885 (-0.02%)	6026691 (-0.07%)	16097175 (-0.36%)	7023990 (-6.93%)	1674930 (27.81%)	343747 (-28.06%)
Star1	4134750 (-11.19%)	5230117 (-13.28%)	14032449 (-13.14%)	6090856 (-19.30%)	1617891 (-30.27%)	273630 (-42.73%)

Comparing the three algorithms, we can see that the number of leaf nodes of Star0.5 is less than Star0 and that of Star1 is even further less. It is also worth mentioning that the improvement is much more significant in end game configurations.

3.3 Transposition Table Performance

To evaluate the performance enhancement of the transposition table, we compare the leaf nodes in the search tree with two agents. One with the transposition table enabled, and the other disabled. The experiments are run against the 6 board configurations introduced in the previous section.

	Open game 1	Open game 2	Middle game 1	Middle game 2	End game 1	End game 2
TP disabled	4134750 (0%)	5230117 (0%)	14032449 (0%)	6090856 (0%)	1617891 (0%)	273630 (0%)
TP enabled	1240975 (-69.99%)	899432 (-82.80%)	1239950 (-91.16%)	409648 (-93.27%)	22024 (-98.64%)	11558 (-95.78%)

The experiment results show that the transposition table does efficiently reuse the search result of previous searches. The impact is especially significant on end game configurations. A possible reason is that a lot of the cubes have been captured, thus multiple dice outcomes may lead to the same game tree search. By utilizing the transposition table, we need not do duplicated search.

4. Discussion

In our experiment, we found that it is usually the case that searching with a fixed depth 6 can be made within the 240-second time limit while setting any search to be depth 8 may take more than 10 minutes for a single move. Due to this finding, we set our `MAX_DEPTH` to be 6 for both the middle game and the end game.

However, during the tournament, we found that some of the moves appear to be far inferior to others after a shallow search is done. Therefore, a further idea is to use statistical analysis to cluster moves based on their scores attained from the shallow search. Then we apply deeper search on the moves that belongs to the cluster with the highest scores only. This may be an idea for further improvement.

5. How to Compile the code

Run the `make` command to compile the code into `main.out`. No argument is required to execute the agent.