# Theory of Computer Games

## Final Project Report (Fall 2023)

學生：許家銓；學號：r12922080

---

## 0. Complilation and Execution

- Compile

```
$ cd r12922080
$ make
```

The generated execution file is called **agent (or agent.exe)**.

---

## 1. NegaScout, Star1 and Transposition Table

這次project中，我主要implement的演算法為**NegaScout**，**Star1**和**Transposition Table**。而接下來為了方便解釋，我會先從我的資料結構開始解釋，後說明程式的整體架構。

### (1) Data Structure

我這次的資料結構直接沿用過去作業中的class EWN，而只是將原本class中會儲存的board改成了全域變數BOARD，但實作完後實際上也只會有一個EWN的object，因此實際上並沒有影響。架構則如下：

```
class EWN {
public:
    int pos[MAX_CUBES * 2];  // red cubes: 0~5, blue cubes: 6~11
    int num_cubes[2];
    int next;  // next player

    int history[60];
    int n_plies;

        // functions
        // ...
};
```

另外我也沒有使用原先架構中儲存Dice順序的dice_seq，那是因為我們的骰子是隨機的，且因為move會被記錄在history的array中，我們實際上不需要儲存，因此我選擇不去使用它。

## (2) NegaScout and Star1 (Search Algorithm)

關於NegaScout與Star1的詳細算法我基本上與課本的內容沒有出入，因此以下我會以我如何結合兩者與一些modified的細節進行說明。

總體來說，Star1負責擲骰子，NegaScout則是負責evaluate跟search。為了處理EWN中骰子的隨機性，原本NegaScout的recursive call就不能回傳對於下一步的精確預估值，而是在考慮骰子的隨機性後回傳的期望值。因此在每次進行Test跟Search時，都要先經過Star1演算法去計算期望值，再由Star1去call下一層的NegaScout。

因此實際NegaScout會變成如下：

```
double NegaScout(EWN &agent, int dice, double alpha, double beta, int dept
{
        ...
    double m = -INF_DIST;
    double n = beta;

    for (int i = 0; i < count; i++) {
        agent.do_move(moves[i]);

        double t = (-1) * star1(agent, -n, -max(alpha, m), depth-1);
        // double t = (-1) * NegaScout(agent, dice, -n, -max(alpha, m), de
        if (t > m)
            if (n == beta || depth < 3 || t >= beta)
                m = t;
            else
                m = (-1) * star1(agent, -beta, -t, depth-1);
                // m = (-1) * NegaScout(agent, dice, -beta, -t, depth-1);

        agent.undo();

        if (m >= beta || m == INF_DIST)
            return m;

        n = max(alpha, m) + 1;
    }
        ...
    return m;
}
```

Star1則是採用uniform case，假設每個骰子的機率相同，並對每個骰子都去計算下一層的NegaScout。具體演算法大致如下：

```
double star1(EWN &agent, double alpha, double beta
                            , int depth)
{
  double A_prev = 6 * (alpha - WIN) + WIN;
  double B_prev = 6 * (beta - LOSE) + LOSE;

  double m_prev = LOSE, M_prev = WIN;

  double vsum = 0;

  for (int dice = 0; dice < 6; dice++)
  {
    double t = NegaScout(agent, dice, max(A_prev
, LOSE), min(B_prev, WIN), depth);
    m_prev += (t - LOSE) / 6;
    M_prev += (t - WIN) / 6;

    if (t >= B_prev)
        return m_prev;
    if (t <= A_prev)
        return M_prev;

    vsum += t;
    A_prev += (WIN - t);
    B_prev += (LOSE - t);
  }
  return vsum / 6;
}
```

跟講義比較不同的是，我們不需要去紀錄每個dice的value A和B，因此我就直接改成同一個參數存。而vmax和vmin都是基於我的evaluation回傳的最大值下去設定，c=6則是骰子的面數。

## (3) Transposition Table

對於每一個State，我去hash它每個棋子的位置與確定骰子之後當下所有可能的走步，也就是最多有12(所有棋子) + 6(兩個棋子的移動方向)=18個數字要hash。這麼做的好處是能夠減少搜尋與建立entry的次數。

假設我方現在盤面只剩下1號與6號棋，那實際上這個盤面我只會產生三種結果：

- 骰面1：動1
- 骰面2-5：動1或6
- 骰面6：動6

此時如果我去hash各個骰面就會對應產生6個entry，而如果只hash能動的棋子跟move則能夠只產生3個hash，並增加3次hash table hit的情況，大幅減少計算的時間。

而hash的方法則是參照C++本身的hash公式來hash：

$$hash = (hash + (324723947 + value))\ xor\ 93485734985;$$

這樣的方法好處是高速且容易執行，但壞處就是除非transposition table有紀錄原始值，否則在發生hash collision的時候難以判斷是否為同一個盤面，可能導致hash crash，但考慮執行的速度，我認為這個犧牲是可以被接受的。

Transposition table的entry有3個value，分別是depth，bound跟type。Depth是紀錄當下的深度，bound是當下的return value，而type則是記錄前面的bound為exact value, upper bound或lower bound。

每次NegaScout在開始搜尋之前，會先check out hash table。沒有就正常做完並存進hash table中，有且深度比現在還深的話就會取值並進行如下的判斷：

- exact value：直接回傳值
- upper bound：將value和現在的alpha比較取maximum，如果取完大於等於beta就直接cut off return。
- lower bound：將value和現在的beta比較取minimum，如果取完小於等於alpha就直接cut off return。

若沒有不是Exact value或不能直接cut，則會去判斷並更新hash的內容。

- 如果m在過程中大於beta且不為極值，則紀錄m為upper bound
- 如果m在完成所有move的搜尋後小於等於alpha，則紀錄m為lower bound
- 剩下則紀錄為exact value

hash table的實作則是利用unorder map去紀錄hash value對應到的index key，並用一個10萬的int array去紀錄每個index對應到的content。

具體實作如下：

```
int transition_table_count = 0;
int transition_table[TT_SIZE][3];   // [depth, value, bound type]
                                    // bound type: 1: upper, 0: exact,
                                                                      //
unordered_map<size_t, int> hash_table;
...
```

```
double NegaScout(EWN &agent, int dice, double alpha
            , double beta, int depth)
{
  ...
  // checkout transition table
  bool is_find_in_hash_table = false;
  size_t hash_value = return_hash_value(agent.pos
, moves, count);

  if (hash_table.find(hash_value) !=
             hash_table.end())
  {
    // printf("hit!\n\n");
    is_find_in_hash_table = true;
    if (transition_table[hash_table[hash_value]][0]
                >= depth)
    {
    switch (
                transition_table[hash_table[hash_value]][2])
    {
    case EXACT:
        return
transition_table[hash_table[hash_value]][1];
    break;
    case UPPER:
        alpha = max(alpha
, transition_table[hash_table[hash_value]][1]);
        if (alpha >= beta)
            return alpha;
    break;
    case LOWER:
        beta = min(beta
, transition_table[hash_table[hash_value]][1]);
        if (beta <= alpha)
            return beta;
    }
    }
  }
    // NegaScout Search
  for (int i = 0; i < count; i++)
  {
```

```
      ...
      if (m >= beta || m == INF_DIST) {
          int tt_status;
          if (m == INF_DIST)
              tt_status = EXACT;
          else
              tt_status = UPPER;

          if (is_find_in_hash_table) {
              // ... update hash table
          }
          else {
              store_result_into_hash(hash_value, depth, m
, tt_status);
          }
          return m;
      }
            ...
  }

  int tt_status;
  if (m > alpha)
      tt_status = EXACT;
  else
      tt_status = LOWER;

  if (is_find_in_hash_table) {
      // ... update hash table
  }
  else {
      store_result_into_hash(hash_value, depth, m, tt_status);
  }
  return m;
}
```

## 2. Heuristic / Evaluation Function

當search到的state為teminal state，則NegaScout會回傳當下state的evaluated value。而此次project中我嘗試了以下兩種不同的Evaluation Function。

### (1) Shortest Distance

一個很直觀的判斷是去計算現在離終點最近的棋子還要幾步才會到終點，以及對手最快還有幾步就到終點。而在一些不同條件的嘗試下，若我方最短距離為L0，對手最短距離為L1，我最終回傳的evaluation function為：

$$(L1 * 2 - L0) * 6$$

這個function可以很簡單的看做是敵方距離減掉我方距離，越大代表我越有優勢。後面乘6是為了避免在star1平均時產生的小數點影響negascout的null window而設定的default scaling。

### (2) Weighted Distance

這個做法是對每個棋子的距離根據選到的機率去做weight。對於每顆存在的棋子，去計算兩側被吃掉的棋子數量並以此為權重乘上距離得到一個分數。

假設只有2跟4號棋，距離終點分別為1跟3步，則因為1, 3都不存在，2的權重就為3；因為3, 5, 6都不存在，4的權重為4。因此分數為[(2 * 3) + (4 * 4)]。

並最後乘上6回傳它的負數，因為這個數字越小代表它的距離越小，越有優勢。而乘上6的理由跟前面相同，只是稍微緩解在double上的精度處理。

---

## 3. Experiments

以下實驗以random作為baseline打100場進行測試，而時間則是另外打5場並計算每場剩餘的時間平均，因此時間的部分越多代表跑得越快。

### (1) Depth Limit Test

這邊就深度來觀察performance的差別:

| Depth | 4 | 6 | 8 | 10 |
|---|---|---|---|---|
| win (/100) | 72 | 75 | 80 | 79 |
| Left Time | 59.23 | 59.20 | 58.73 | 59.01 |

從Performance可以發現在增加深度能夠在一定程度上確實增加勝率，但當深度超過8層以後勝率就沒有特別的變化，停留在8成左右。

時間的部分則是因為都低於1秒鐘，看不太出來變化，且多數變化來自於盤面的情況導致搜尋的速度不同。

### (2) Transposition Table加速實驗

為了實測transposition table為程式碼帶來的加速，我透過不同的深度進行測試，並得到實際上transposition table的勝率與時間的差距。

以深度4與深度6進行測試，至於深度8因為沒有transtion table跑不完因此沒有測試：

| Depth 4 | win (/100) | Left Time |
|---|---|---|
| without TT | 72 | 59.41 |
| with TT | 70 | 59.62 |
| Depth 6 | win (/100) | Left Time |
| without TT | 84 | 41.23 |
| with TT | 86 | 59.64 |

比較深度的話可以發現雖然在深度4並沒有太多變化，但深度為6的時候，加深的兩層所造成的loading是接近36倍，因此放大後時間的performace有相當程度的進步，且勝率沒有影響，可見TT的實作確實能夠帶來足夠的效益。另外經過實測目前在使用Transposition Table後能夠用2-4秒的時間搜尋完8層的深度。

## (3) Shortest Distance Evaluation Experiment

為了調整shortest distance的參數，以下記錄了實際嘗試的公式與結果：

| formula | win (/100) | formula | win (/100) | formula | win (/100) |
|---|---|---|---|---|---|
| (L1 - L0) | 75 | (L1 - L0 * 2) * 6 | 80 | (L1 * 2- L0) * 6 | 84 |
| (L1 - L0) * 6 | 82 | (L1 - L0 * 4) * 6 | 78 | (L1 * 4 - L0) * 6 | 79 |
| (L1 - L0) * 12 | 72 | (L1 - L0 * 8) * 6 | 78 | (L1 * 8 - L0) * 6 | 80 |

首先是關於對實際分數的scaling，在經過測試後可以發現scale為6的時候performace表現比較好，因此之後的測試都以scale=6進行。

而接下來我測試透過增加我方最短距離的scale去激勵agent前進，但實際的結果發現並沒有太大的變化，因此基於好奇我反過來測試去增加敵方最短距離的scaling，反而發現效果有增加。在研究盤面的變化後，我推測可能是因為敵方接近的同時我方也突破對方的防線，因此導致勝率上升。

## (4) Weighted Distance Evaluation Experiment

這邊我並沒有進行太多的實驗，因此就直接與Shortest Distance中最好的performace進行比較：

| formula | win (/100) |
|---|---|
| (L1 * 2- L0) * 6 | 84 |
| weighted distance | 80 |

兩者的performance比較下來可以發現實際上還是shortest distance的performance比較好，這我認為可能還是因為shortest distance中的加權導致agent的performance上升。

## 4. Conclusion and Discussion

這次的Final Project的遊戲我認為運氣成分還是偏高，雖然上面已經以100場作為測試基準，但成績的浮動性依舊很大。實際上我最終使用的程式勝率依舊在79%到85%的幅度之間搖擺，成績並不穩定。

而實作上我認為transposition table的實作偏難，最一開始我嘗試了使用自己的hash function與hash map，但除了搜尋速度不夠快，也同時會產生不少hash crash的情況。因此最後回頭參考實際上hash map的實作進行hash，才有明顯的進步。

至於performance的部分，我認為依舊是限制於我的evaluation function太過簡易，在經過比賽的討論後，我或許可以透過precomputed table與determinancy的計算去增進performance。

最後，謝謝教授與助教的努力，這是堂很棒的課 (雖然我兩次作業都吃了wrong format QQ)。

一個小建議是可以稍微改變zip的敘述，因為有些壓縮程式對資料夾壓縮後並不會多一層資料夾，可以對這個情況有更好的描述，避免未來有更多像我這樣的蠢材犯下這種錯誤。