Theory of Computer Games

Homework 1 Report (Fall 2023)

學生：許家銓　學號：R12922080

作業檔案內容：

```
r12922080
├── Makefile
├── report.pdf
└── src
    ├── ewn.cpp
    ├── ewn.h
    ├── main.cpp
    └── verifier.cpp
```

1. Compilation and Execution

(1) **編譯方法**

```
$ ls
Makefile  report.pdf  src/
$ make
g++ -std=c++11 -O3 -Wall -Wextra src/main.cpp -o agent
g++ -std=c++11 -O2 -Wall -Wextra src/verifier.cpp src/ewn.cpp -o verifier
$ ls
agent*  Makefile  report.pdf  src/  verifier*
```

編譯完後會生成兩個執行檔，agent為作業所要求的程式，verifier則是用來測試agent輸出合法性的程式。

(2) **執行方法**

```
# 單獨執行解題的agent
$ ./agent < [input data]
# 使用verifier執行agent 測試是否合法
$ ./verifier ./agent [input data]
```

2. A* Algorithm and Implementation

這次作業我所使用的演算法為**A* Algorithm**。為了方便說明，我們先了解題目內容與使用的components。

作業題目要求程式找出一款單人愛因斯坦棋的最佳解，因此我們將遊戲過程分解。除了初始盤面，每次遊戲多下一步就會形成一個新的state(盤面)，而一個state會包含三個重要資訊：**棋子的位置(pos)**、**目前的**

步數(n_plies)以及**盤面分數(state value)**，這些都是我們在使用A*時相當重要的資訊。

在使用A*時我們主要會用到兩個Container：

- **Hash Table**：用來記錄state pos的hash value是否已經被尋訪過，並被用來對應到這個pos中的最佳state。
- **Priority Queue (pq)**：用來排序並儲存要計算與處理的state，insert的值為state pos的hash value，並會根據hash table中的最佳state的state value進行排序。

在A*的過程中，每次我都會從pq中找到目前盤面分數最高(state value最小)的state S，接著對每一個S的下一步S'做檢查：

- 如果S'抵達終點，那就紀錄下來作為目前的最佳解
- 如果S'還未抵達終點，就先確認S'的pos有沒有在Hash Table裡，沒有的話就做為新的state將S'放進pq
- 如果是，則會去確認Hash Table中的state與S'的n_plies誰比較小，若S'比較小就將Hash Table中的state更新為S'

因此大致上的流程如下：

```
pq.push(hash value of initial state pos);
hash_table[hash value of initial state pos] = initial state;

while (pq is not empty)
{
  cur_hash = pq.top()          // pq中分數最高的hash_value
  cur = hash_table[hash_value] // hash_value對應到的最佳state

  for cur能下的每一步move {
    cur' = cur + move

    if cur'到終點 且比目前記錄的解還好 {
      將cur'紀錄為最佳解
    }
    else if cur'的位置已經出現在hash_table了 {
      if cur'的盤面分數比較高：
        _cur_hash_value = cur'位置的hash_value
        hash_table[_cur_hash_value] = cur';  // 更新hash_table的最佳解
    }
    else if cur'的位置沒有出現在hash_table {
      _cur_hash_value = cur'位置的hash_value
      hash_table[_cur_hash_value] = cur';    // 新增hash_table的最佳解
      pq.insert(_cur_hash_value);            // 放入pq運算
    }
  }
}
```

3. Tricks, Heuristic, Experiments and Findings

這段我會說明我為了優化程式的執行所使用的tricks和heuristic functions，並最終結合出前面所述的state value。

(1) Moves Generated

由於終點很明確的在棋盤的右下方，因此在大多數情況下向右下方走對盤面比較有利。

於是我稍微調動了function中回傳可能步數的順序，讓A*在執行時會用以下順序優先執行可以到達的state：向右下走→向右/下→向左下/向右上→向左/上→向左上走。

以下我實驗了兩種不同走法順序的時間：

單位:ns	11.in	21.in	22.in	31.in	32.in
上述順序	**176900**	**2608000**	**13458800**	**770174400**	5011800
將順序反過來	278200	9690400	33394000	776471500	**3226700**

可以發現大多時候右下優先策略會優於左上優先策略，且值得一提的是在21.in與22.in兩筆資料中，右下優先策略在第一時間就找到最佳解，而左上優先策略則是先找到了次優解，而後才找到最佳解。

(2) Step Need

假設現在的state走了n_plies步，那最少還要幾步才會走到終點呢？一個顯而易見的lower bound就是目標棋到終點的最短距離，若沒有目標棋就是離終點最近的棋到終點的最短距離。而我在這邊給這個最短距離命名為Step Need。

對於每個state，他最終的答案都不會小於n_plies加上step_need，兩者的和越小，他到達最佳解的可能性就越大。因此我先定義state_value = n_plies + step_need，以此做為我最初的state_value。另外當目標棋已經被吃掉，就不存在step_need，此時的step_need就會設為是MAX_MOVES+1，作為最大的upper bound。

為了證明這個方法在多數時候有用，我跑了下面兩種state_value的比較：

state_value	11.in	21.in	22.in	31.in	32.in
n_plies	605600	1568794800	1433454800	3803519400	1619734300
n_plies+step_need	**173800**	**90007800**	**69588200**	**701071500**	**67291400**

從這邊就可以發現，在使用step_need作為lower bound的參考後，跑的速度有明顯變快許多。

(3) Valid Moves

接下來要考慮到state_value中最後一個參數，valid_move。根據愛因斯坦棋會吃棋的特性，吃得越多，給予其他棋子移動的機會就越大，而一個簡單的參考依據就是目前目標棋在sequence中能夠移動的步數有多少，也就是valid move。若目標棋能夠移動的次數越多，那相對來說就更容易到達最佳解。而因為valid_move是越多越好，因此在state_value中為負數，結合前面的Step Need可得：

state_value = n_plies + step_need - valid_move

那一樣我們跑了valid_move的一些實驗來觀察valid_move的影響

state_value	11.in	21.in	22.in	31.in	32.in

n_plies+step_need	173800	90007800	69588200	**701071500**	67291400
n_plies+valid_move	435500	TLE	TLE	3119874100	TLE
All	**162800**	**2413700**	**16095700**	753630300	**6181500**

這邊先註明，11.in與31.in並沒有goal piece，因此兩者的valid_move只會回傳0。

而將11與31移除的話可以發現，step_need的影響比valid_move大很多，若沒有step_need的bound，多出來的部分運算量甚至會導致在部分測資上TLE。但在有step_need的前提下，valid_move在那些有goal piece的測資上所展現的優化是非常可觀的，有些測資甚至快了10倍找到最佳解，因此這是一個相當有用的heuristic value。

至此，state_value已經被定型為n_plies + step_need - valid_move了。而接下來的優化會著重在程式與演算法本身的優化。

(4) Hash Value of State

在前面的演算法中有提到，我在pq裡面放的是hash value，但最一開始我是直接將整個state的class放進去裡面，也就是：

- **priority_queue<EWN, vector<EWN>, game_cmp> pq;**
- **unordered_set<EWN, ewnHash, ewnHashEqual> vis;**

這當中unordered_set作為hash table，會將state的position跟n_plies都一起hash進去，公式如下：

```
class ewnHash {
public:
    size_t operator()(const EWN& cur)const {
        std::hash<int> hasher;
        size_t seed = 0;
        seed ^= hasher(cur.pos[1]) + 0x9e3779b9 + (seed<<6) + (seed >> 2);
        seed ^= hasher(cur.pos[2]) + 0x9e3779b9 + (seed<<6) + (seed >> 2);
        seed ^= hasher(cur.pos[3]) + 0x9e3779b9 + (seed<<6) + (seed >> 2);
        seed ^= hasher(cur.pos[4]) + 0x9e3779b9 + (seed<<6) + (seed >> 2);
        seed ^= hasher(cur.pos[5]) + 0x9e3779b9 + (seed<<6) + (seed >> 2);
        seed ^= hasher(cur.pos[6]) + 0x9e3779b9 + (seed<<6) + (seed >> 2);
        // TODO: remove n_plies from hash
        seed ^= hasher(cur.n_plies) + 0x9e3779b9 + (seed<<6) + (seed >> 2);
        return seed;
    }
};
```

這樣的好處是不會經過太多hashing的處理，但會需要在pq中大量的request memory space導致overhead。因此此時的agent表現也相當不理想：

11.in	21.in	22.in	31.in	32.in
262200	7101900	131255400	TLE	72877600

多數測資除了跑得相對慢，在那些沒有valid_move優化的測資(31.in)也會TLE。因此我之後先嘗試了將priority queue裡面的內容換成hash_value，並同時為了減少memory overhead，我在最一開始先建立了一個global的buffer避免大量的request空間，而為了對應hash value與buffer中state的index，我也將unordered_set改成了unordered_map。因此結構變成了：

- **priority_queue<size_t, vector<size_t>, game_cmp> pq;**
- **unordered_map<size_t, int> vis;**：key為hash value，value為在buffer中的index

而與前面原本的方法比較之下：

pq	11.in	21.in	22.in	31.in	32.in
class	**262200**	**7101900**	131255400	TLE	72877600
hash_value	329700	92791700	**63392300**	**3078959000**	**36420400**

可以發現雖然在某些測資下表現並不會變好，但當測資大小變大，優化的效果就會變得更明顯。最顯著的變化也就是31.in能夠跑進5秒鐘內了。

(5) Hash Value of Position

而雖然前者說能夠跑進5秒鐘，但在多次的測試之下依舊是相當不穩定，有時候甚至會需要跑到5.1秒。因此我持續優化目前的演算法。

在向同樣有修此課的同學**郭恩銘**討論過後，我得到了一個非常特別的想法，就是將n_plies的內容從hash_value中移除，hash_value只存目前的盤面。這樣的做法可以減少state的數量，進而增加每個container運算的速度，而這樣看似直接的解方，其實會遇到很多問題。

首先，在愛因斯坦棋中，同樣的盤面不代表同樣的state，因為能夠下的棋子不一定會一樣，隨著n_plies的不同，兩個state的變化也會不一樣，因此這必須要納入考量。同時原先一個state只會對應到一個hash_value，現在由於只存position，同樣的hash_value會對應到多個state，hash table中key與value的對應就要重新handle，並判斷現在的hash_value應該存哪個state。

我最後實際的做法是讓hash table只存score最好的state，如果之後遇到同樣的position就比較互相的score，並讓hash table去儲存score最好的state。因此可以發現在前面的演算法中，我已經有handle遇到visited的hash_value時要更新hash_table。而實際上的表現如下：

time	11.in	21.in	22.in	31.in	32.in
hash pos and n_plies	329700	92791700	63392300	3078959000	36420400
hash pos only	**206000**	**2687300**	**13195400**	**688868400**	**5246300**

state num	11.in	21.in	22.in	31.in	32.in
hash pos and n_plies	**1488677**	**2537418**	**2852379**	**2891835**	2672616
hash pos only	133627	2370941	2284875	2296214	**2792595**

可以發現，只hash position的情況下速度會有驚人的提升，而若比較state的數量，在多數情況下也有變少。

雖然結果是好的，但只取最好的score也有可能導致一些最佳解無法被得到。

假設存在一條最佳解會經過同樣的position兩次，令第一個state為A，第二個為B，則當我們遇到B state的時候，由於position相同，B的n_plies又比A大，因此B的hash value會被認作是次佳解，前面的演算法就不會將其放入pq中，連帶導致這條最佳解無法被探索到。

由於這個問題的存在，使得我一開始卡在hash_table的更新很久，到此刻也沒有想到能夠避免此狀況發生的方法。但換個角度思考，就算發生了，他對performance的傷害會有多少?

假設A與B中間多走了k步，是否有可能換個順序不經過B走到存在的解？若這k步造成的分布不影響A與B中間特定子的關係，或許存在同樣長的路線也能夠到達最佳解。而若A與B中間每顆棋子的位置都必須固定，那也可以透過多出至多一個period的方式將棋子的位置平移得到新的state。最重要的是雖然得到的是次佳解，但此次作業的評分標準中，避免TLE才是最重要的事情，因此我選擇去犧牲掉這種可能性的最佳解。

4. Final Result

以下是我最終在每筆測資上得到的結果：

data	Final Solution Solving Time	Steps
11.in	203600	5
12.in	407500	9
13.in	2206100	9
14.in	2780900	10
21.in	2235900	13
22.in	13604100	13
23.in	398270700	12
31.in	706438100	13
32.in	4898900	14
33.in	18695400	16