R12922167 郭恩銘

一、如何編譯與執行我的程式:

我的 zip 解壓縮完後應該長這樣:

R12922167

|------- makefile

|------- report.pdf

|------- src

|------ewn.h

|------astar.cpp

直接在 R12922167 的資料夾裡輸入 make 就可以編譯產生 test 的執行檔

之後輸入./test 可以執行 (或是 ./test < 測資.in 直接可以直接導入測資輸出結果)

二、程式結構與一點實作細節:

雖然助教有在 ewn.cpp 實作方便 IDA* 演算法的函數,但我考量到用基於 DFS 的方法會有 do 跟 undo 的 overhead,故選擇用 A* 演算法來實作,heuristic 的計算我寫在下章

以下為幾點實作細節: (方便第三節說明,助教覺得篇幅太長可以跳過)

(1). 由於骰子的順序是已知的,所以我會先推算 "假設之後沒再發生其他棋子死亡" 的情況下,2^6 種棋子存活的可能性(每個棋子不是死就是活,有六個棋子所以是 2 的六次方)中,每個棋子要動幾步需要投幾次骰子。

舉例來說假如骰子週期是 1 2 3 5 這四個,先把他擴展成 100 ply 的週期 1 2 3 5 1 2 3 5 1 2.... ,考慮現在在第七 ply,骰子擲出來的數字是 3

故只有 1 ,3 號棋子活著的時候 :

1 號棋子要再 3 ply 才能動 1 次、4 ply 動 2 次 、 7 ply 動 3 次 ...

3 號棋子再 1 ply 可以動 1 次、2 ply 動 2 次 、 5 ply 動 3 次 ...

2,4,5,6 號棋子之後都動不了

而只有 1 號棋子活著的時候 :

1 號棋子再 1 ply 可以動 1 次、2 ply 可以動 2 次 ...

2,3,4,5,6 號棋子之後都動不了

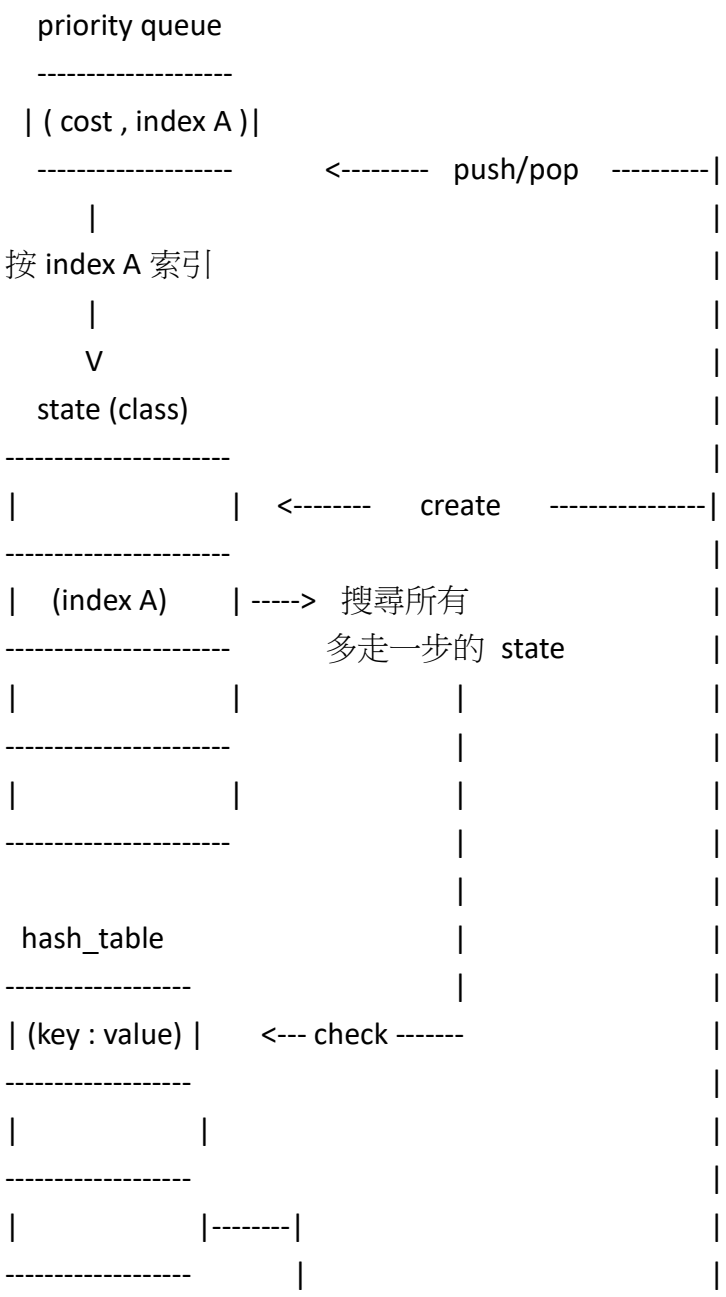
如此,最一開始先花時間與空間建立好有 64 * 6 個 entry 的幾個輔助陣列,之後如果 heuristic 使用到這種還要擲幾次骰(幾個 ply )的估算就可以直接查表快速解決,但缺點是 "假設之後沒再發生其他棋子死亡" 這個前提會造成高估棋子移動需要再幾個 ply,這點第三節 test log 會說明

(2). A* 演算法需要一個 global 的 priority queue,考慮 c++ 的 priority queue 沒有實作 decrease key,且記錄每一個 state space 的 state 都需要記錄版面和是從哪個 state 找到他的,我的 priority queue 裡面放的是 ( cost , state index ) ,已節省空間

cost 就是 current cost + heuristic cost

state index 是一個全域陣列的 index，指向我在一開始額外開的超大儲存這些必須資訊的 class 陣列的某 entry，而因為程式只能跑 5 秒，所以我開 1000000 的大小基本上就已經夠記錄搜尋到的所有 state 了，但我還是怕爆掉所以開到 5000000，實際監測 memory 不會長超過 1GB。此外，因為在 A* 演算法中需要查找一個 state 之前有沒有搜尋過，我還用了一個 c++ hash table ( unordered_map )來實作這件事

整體流程圖如下:




key 直接存 state class 的實例並實作 c++ class 的 hash function 與重載 == operator，value 存該 state 的 index

如果是空的，代表是新 state，create 新 state 並 push ( 該 state cost，該 state 的 index ) 到 priority queue 中

如果有找到，按 value 找回原 state，如果新的 state current cost 比原 state 少才新建一個 state 並 push 到 priority queue 中，並用點小技巧 pop 掉在 PQ 的原 state

hash_table 這邊可能有點 memory waste 的問題，但如果轉成 int 做為 key 可能會有 hash collision 的問題，用 tuple 的儲存方式至少也要 6 個 int 的空間不划算所以我才直接做在原本 class 上

(3). 兩 state 相等的定義與取捨 :

當要查詢一個 state 之前有沒有遇過，除了看"六個棋子位置相等與否"，這種決定性的愛因斯坦棋還要考慮當前到第幾個 ply 了，就常理來看同樣棋子位置，不同 ply 對應到的 state 應該要不一樣，但我直接假設他們會是一樣的，比如考慮兩個在不同 ply 的但同棋子位置的 state

我使用了" ply 數小的比 ply 數大的多出的幾個 ply 數只會造成結果相等或變好"這個假設，因為棋子有 8 個方向可以動所以多出的幾個 ply 有很多種無傷走法，如果是要被吃的棋子不得不動可以選擇繞著吃他的棋子走，比如說:

有兩個 state 有相同的棋子位置，但兩者的 ply 數差了 1。假設 ply 數大的 state 是其中一最佳解會經過的 state，輪到他時最佳走法是讓 4 號棋子被 1 號吃，但我在 ply 數小的 state 多出了一步擲到 4 號棋子的 move ，按規則要動完他才能照著 ply 數大的走法讓 1 號動

那麼 4 號可以選擇繞著 1 號走，兩條路讓程式自己選傷害最小的位置移動

Ply 數小的走法:

```
1 0        1 0        0 0
0 4 ->     4 0   ->   1 0
      (waste move)
```

Ply 數大的走法(最佳解):

```
1 0       0 0
0 4 ->    0 1
```

兩者差別只在吃完後棋子的位置差了一格。若在最壞的情況這種假設可能會導致需要再多幾步才能到終點，但我暫時想不到最嚴重的例子，說不定還會變好所以我就選用這種策略了

這樣 priority 只需要考慮那些 ply 數小的版面就好，ply 數大的可以當成不是最優解丟出 priority queue 外，減少記憶體使用的 state

( 雖然找到的可能不是最佳解，但我覺得加入這個假設後程式很快就能結束應該會比不加好，可能犧牲最佳解至少確定不會 TLE )

後面有 test log

三、test log 與發現:

(1). 9/29 完成主程式架構並確定可以輸出一組解，不過只會輸出找到的第一個解

使用的演算法為 A*，每個 state 的
current cost 為 "目前已經走的步數"
heuristic cost 為 "所有可以到終點的棋子與終點的最短距離中，最短的那個"

因為有"右下"這個方向可以走，所以與終點的最短距離可以直接寫成
MAX (棋子與右邊邊界的距離，棋子與下面邊界的距離 ) -> 以下簡稱 D

可以看出 current cost + heuristic cost 必小於經過他到終點的 actual cost ，因為我們最快一定要擲至少 D 次骰子(即 D 個 ply)，而且要剛剛好都能動那個離終點最近的可 goal 棋子。因此這個 heuristic function 是 admissible 的，找到的第一個解即是一個最佳解，使用這個 heuristic function 助教提供的所有測資都能在 5 秒內跑完
下面表格中稱呼他為"原版本"，所有的實驗版本比較對象都是他

(2). 10/10 修改程式架構變成 4.5 秒內盡可能找解，方便測試其他 heuristic function

(3). 10/10 額外測試一種 heuristic function
heuristic cost 為 "假設之後沒有其他棋子死亡，在所有可以到終點的棋子直直走到終點所需投的骰子數裡，最少的那個 "，也就是我在 二之(1) 裡預先算的那個東西
由於可能會發生走不到終點的情況 (好比骰子週期 1 2 3 4 時若每個棋子都存活，6 號棋子就不可能走的到終點 )，因此我會把這個 heuristic cost 上限設成走 10 步的值，
具體的方式是先把原始 heuristic cost 值大於 50 的都壓到 50 ，然後再一律除以 5 得來上限 10 步，也就是 0-5 映射到 0 cost，6 – 10 映射到 1 cost，此外除以 2 就是上限 25 步

下面放上比上面初版本的比較的表格(10/11 更新)
**表內為找最終解花的時間與結果(括號為找出第一個解的時間與對應解，如果有的話)**

| | 原版本 time | 原版答案 | 新 /5 版本 | /5 版答案 | 新 /2 版本 | /2 版答案 |
|---|---|---|---|---|---|---|
| 31.in | 1.509 | 13 | X 找不到 | X 找不到 | 0.187 | 13 |
| 32.in | 0.066 | 14 | 0.07 | 14 | 0.006 (0.0004) | 14 (15) |
| 33.in | 0.3499 | 16 | 0.051 | 16 | 2.093 (0.0002) | 16 (20) |
| 21.in | 0.112 | 12 | 0.057 | 12 | 0.032 (0.0016) | 12 (13) |
| 22.in | 0.336 | 13 | 1.9238 | 13 | 0.1161 | 13 |
| 23.in | 0.608 | 12 | X 找不到 | X 找不到 | 0.0025 | 12 |

看得出來找到第一組解的時間可以到很短 (31.in 的測資上獲得了很大的進步)，但也很容易什麼都找不到(<-當除太多的時候) 或是花很多時間在錯誤的 state 上搜索。
除太多 prune 太少 ，除太少有高機率 over estimate heuristic cost

我推測應該和 heuristic 沒有鼓勵去吃棋子，舉例來說當 1,2,3,4,5 號棋子都存活時，如果目標要讓 5 號去 goal 。當吃棋的那步沒有辦法立刻給 5 號多的移動機會，就不會得到高的 score 所以最佳解反而會沉到 priority queue 的深處

(4).10/10 額外測試一種 heuristic function

heuristic cost 為

"假設之後沒有其他棋子死亡，在所有可以到終點的棋子直直走到終點所需投的骰子數裡，最少的那個" 的除以 2 版本

多加上 "假設之後沒有其他棋子死亡，盤面任兩個棋子能相遇 (即吃子) 所需投的骰子數，最少的那個"，壓到 25 以下後，那個值再除以 5 (即上限為 5 步)。也有測除以 2 的版本(即上限 12 步)

一樣放上比較表格(10/11 更新)

| | 原版本 time | 原版答案 | 新 /5 版本 | /5 版答案 | 新 /2 版本 | /2 版答案 |
|---|---|---|---|---|---|---|
| 31.in | 1.509 | 13 | 0.173 | 13 | 0.181 | 13 |
| 32.in | 0.066 | 14 | 0.008 (0.0011) | 14 (16) | 0.008 (0.0010) | 14 (15) |
| 33.in | 0.3499 | 16 | 2.077 (0.0004) | 16 (17) | 2.345 (0.0004) | 16 (17) |
| 21.in | 0.112 | 12 | 0.005 (0.0016) | 12 (13) | 0.016 (0.0006) | 12 (13) |
| 22.in | 0.336 | 13 | 0.102 | 13 | 0.0901 | 13 |
| 23.in | 0.608 | 12 | 0.055 | 12 | 0.6572(0.3538) | 12(13) |

看起來 33.in 那筆測資 並沒有好的進步，但 23.in 獲得了很大的進步。不過多加的那一項上限不能開太大，不然 heuristic cost 總和會太大，就超容易 over estimate

(5). 10/11 更新: 找到一個重大 bug，重新修正程式架構與上面表格結果

bug 如下: 我比較 state 相等的規則是只看 "六個棋子的位置" ，不看當前 current step 到哪，為了節省記憶體我每個 state 又只會對應到一個 now step 和一個 parent，想當然我是存最小的 current step。

但這會有一個問題，我是用類似 link list 的方式儲存答案的搜索串列的，如果今天有個 goal state 找到答案後(其實只要他在 PQ 內就會出 bug )，其他 state 動到了找到的解前面的路徑，意思是說剛好他能夠比較早到達某個 state，但如果他之後的步數都在耍廢無法產生步數更短的最佳解或是遇到 timeout 被中斷，會導致我儲存的解 ply 對不上變成 illegal move 。因此我改成當找到一個到比原 state 還好的步數時新建一個 state，並且把原 state 從 priority queue pop 掉

(6). 10/12 合併以上三種 heuristic，做出我的最終版本

即 1."所有可以到終點的棋子與終點的最短距離中，最短的那個"，不做任何調整

2."假設之後沒有其他棋子死亡，在所有可以到終點的棋子直直走到終點所需投的骰子數裡，最少的那個"，壓到 50 後除以 5 (上限 10)

3."假設之後沒有其他棋子死亡，盤面任兩個棋子能相遇(即吃子)所需投的骰子數，最少的那個"，壓到 25 後除以 5 (上限 5)

得到的結果如下:

| | 原版本 time | 原版答案 | 新版 time | 新版答案 |
|---|---|---|---|---|
| 31.in | 1.509 | 13 | 0.332527 | 13 |
| 32.in | 0.066 | 14 | 0.002885(0.002680) | 14 (15) |
| 33.in | 0.3499 | 16 | 0.204444(0.000683) | 16 (17) |
| 21.in | 0.112 | 12 | 0.001697 | 12 |
| 22.in | 0.336 | 13 | 0.065740 | 13 |
| 23.in | 0.608 | 12 | 0.069030 | 12 |

可以看到 31.in 已經有顯著的進步了，因為他有 1. 這個穩定的 heuristic，同時又能包含 2. 與 3. 鼓勵吃子與往終點走的兩個蠻直覺的小要素

這些小要素的上限又不需要設太大，因此我決定拿他當我的最終版本，當然也可以考慮按存活棋子數給 2. 3. 一個加權，但我怕變太複雜就沒去修了。