Santorini AI Implementation

一、 策略邏輯

以下分成幾個要點陳述我的策略邏輯

1. 資訊的使用：

我使用了 chessColor.txt、chessStructure.txt 去實作策略，沒有使用 stepLog.txt 的原因是因為目前我的程式還無法針對雙方下棋「順序」進行判定，惟利用目前現有工人位置及建築物分布決定下一步棋子的走向。

2. 策略的決定：

本程式以類 Monte Carlo Method 的方式，以大量隨機下棋的數據，輔以一些 rules 校正部分數據勝率，來決定最終走向。(第二部分有較完整的說明，已歸類在策略決定的部分)

二、 策略架構

以下將依據程式運行順序，陳述我的策略程式架構。

首先我先以 fopen()、fscanf()等函式讀取目前棋盤的分布，將工人位置與建築物分布寫入 now_worker_pos[5][5]及 now_building_pos[5][5]兩個 int 的 two-dimensional array 中。(使用 array 的優點：取值容易、元素數量有限；缺點：維度高時較難操作)

接著創建雙方的 struct worker[2]，記錄雙方各兩位工人的位置、所站

位置樓層數、天神卡、是否能移動及建造、顏色等等。以傳入的 argv[1] parameter 決定雙方棋子顏色並初始化所有數據後，若此時是放棋階段，先手的話我預先準備了 3 個 case 以備不時之需，將放在分散且對稱的對角或兩側上，比較好兼顧到整張地圖的範圍；後手的話我會放在對手所放的棋子附近，避免對方在我視線範圍外快速蓋出 3 樓。

接下來便是重頭戲，我參考了 machine learning 中的 Monte Carlo method，以十幾萬次的迴圈重複模擬從現在的地圖開始的各種可能的局。其中每局以我方與對方皆輪流隨機放置下一步棋(也隨機使用或不使用天神)，直到勝負分曉並記錄此局我先以哪位工人移動到哪裡與建造在哪裡。最終我將這十幾萬局的資料中，理論上將選擇勝率最高的那步棋去動作。但後來實際測試發現需要經過一些 rule base 處理，因為對手大多不是隨機下棋的。若完全假設對方會隨機下，那很有可能對方下步要贏了我還在蓋其他位置而輸局。因此我在篩選最高勝率的組合時，會去調整發生以下情況的勝率：

1. 現在是否必須防守：當對方人在二樓，他可以走上三樓，且我的棋子經過移動後蓋得到他即將走上的位置。若必須防守，大幅降低不去防守的樣本勝率，幾乎確保我會守到這步棋。

2. 現在這樣下是否會讓對手能直接上三樓：若我現在蓋的位置能讓對手在下一步的時候直接上三樓，那基本上應當作我必輸無疑，因此也大幅降低去蓋此格的勝率，幾乎確保我不會送頭。(除非我只剩這步能走)

3. 我方棋子互相靠太近/我方棋子靠近對方棋子：由於視角有限，為了避免我看不到對方棋子而讓對方自己蓋到贏，略為降低我方兩個棋子距離小於 2 的勝率，調高我方棋子靠近對方棋子的勝率。(得確保不會發生我下一步就能贏卻不走的情況，得加入此回合不會贏的條件)

此外，由於移動有侷限性，若我方棋子和對方棋子放太多回合，導致我的棋子被卡住無法防守到對方而輸局，紀錄樣本時我會以模擬局開始到結束的回合數作加權，越快結束的佔比越高，以免看得太長遠而發生上述情況。

最後我的棋子將依照經過以上數據處理後，勝率最高的那步棋去移動。

本算法的優點在於避免 case by case 的走法，除了開發者本身得先匯完遊戲且走法較容易被別人識破外，也不易推演回合數較長的結果。更重要的是可能會有各種 corner case 導致違規。

而缺點在於本算法其實離真實的 Monte Carlo Method 有段不小的差距，除了運用大量數據與機率這點相似外，由於程式無法依據先前模擬過的局數加以修正或加權，勝率不一定高，維護上也相對無從下手(因為只看得到最終數據結果)。

＊註：以上會使用到幾個重要的 function：

1. void find_worker(building)_valid_pos(struct worker[2], int)

//尋找工人在模擬時所站的位置下，是否能移動(建造)

→若找到可以行動的點，增加該 worker 的.canmove(build)數值。

2. int sim_one_action(struct worker[2], char*[])

//模擬一玩家進行一回合的步驟

→先判定兩位工人能不能移動，若有其中一位可移動，則以 while(1)搭配 rand()取得隨機的格子點直到取得合法的移動點，反之則回傳移動方輸了這局。若可移動，再判定這位移動的工人是否能建築在他的周圍，若可以建造，回傳此局尚未結束讓對手行動，反之一樣回傳移動方輸。

3. void sim_one_game(struct worker[2], struct worker[2], char*[]) //模擬一局

→先讓我方移動一次，取得此時移動與建造的位置作為紀錄樣本的依據。再依序讓對手與我輪流隨機行動，直到勝負分曉，依據回合數進行樣本數加權後加入 mytried/mywin[5][5][5][5][5]裡。

4. int verify_oppcan_threefloor_after_mybuild(struct worker[2], int, int, int, int); //判斷對方是否會在我建造完後直接上三樓

→在篩選數據時進行，若此樣本所建造的位置會讓對手能從二樓站上三樓，則此樣本的勝率需降低。

5. int *verify_must_defend(struct worker[2], struct worker[2]); //判斷我方現在是否一定要防守對手上三樓

→若我的某位工人能移動，且能蓋在對手即將站上三樓的位置，則此

樣本以外的樣本勝率須大幅降低，讓我方會去防守。

6. int verify_nowcan_win(struct worker[2]);

//判斷我現在是否能直接站上三樓獲勝

→由於在篩選數據時會讓「使我方棋子靠近對方棋子」的樣本勝率增加，而可能在我方快站上三樓時反而移去對方棋子附近。故需要此function確保我方能把握獲勝機會站上三樓。

三、其他

1. 須以UTF-8格式開啟程式原始碼，若使用不相符的編碼開啟，可能會出現亂碼。