Theory of Computer Games HW1

B02902071 陳柏堯

1. How to Run?

(1) Set Python Environment

alias python as /usr/bin/python2.7 and install numpy

{./code} $ source make.sh

EX:

```
b02902071@linux8 {~/Course/TCG/tcg_hw1/b02902071/code} [W0] source make.sh
Requirement already satisfied (use --upgrade to upgrade): numpy in /usr/lib/python3.5/site-packages
You are using pip version 8.1.2, however version 9.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

(2) Run Code

{./code} > python run.py $QuestionsFilePath $Answers_File_Path $Method $ProblemID

Note:

$QuestionsFilePath is the path to file “tcga2016-question.txt”

$Answers_File_Path is the path you would like to output the result

$Method is the method short name you want to try

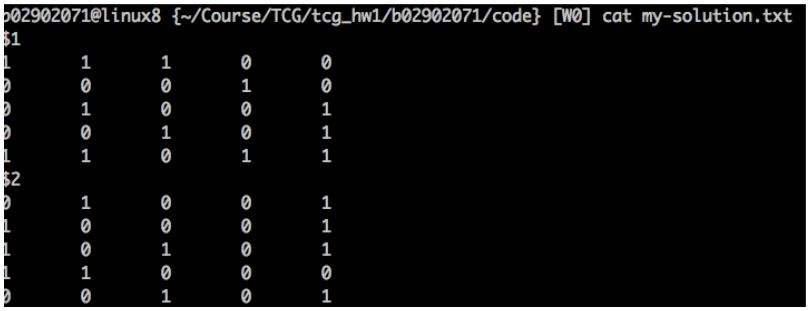
5 methods to use: { “bruteforce” / “dfs” / “dfs_rowbyrow” / “dfs_RL” / “bfs_heuristic_RL” }

$ProblemID is the id number the problem you want to solve.

(“0” represent all problems)

EX:

```
b02902071@linux8 {~/Course/TCG/tcg_hw1/b02902071/code} [W0] python run.py tcga2016-question.txt my-solution.txt dfs_rowbyrow 0
$1
[[1 1 1 0 0]
 [0 0 0 1 0]
 [0 1 0 0 1]
 [0 0 1 0 1]
 [1 1 0 1 1]]
0:00:00.008819
```



2. Implemented Algorithm

(1) Brute Force Search (Method: “brute_force”)

BruteForce 搜索嘗試所有的 state 並且檢查是否符合 hint 的條件，符合的則回傳。

(2) DFS (Method: “dfs”)

從左上角到右下角一格一格填，填 1 或者填 0 是每一格的 branch。分別檢查 row_hint 和 col_hint 是否有可能滿足目前填過的格子，不滿足的話則 traceback 回去到另一個分支下去繼續搜索。

(3) DFS Row by Row (Method: “dfs_rowbyrow”)

將每一橫列的排列組合可能性都列出來，然後從最上面一列到最後一列一列一列下去分支作 DFS。每一列下去都去檢查 col_hint 是否有可能滿足目前填過或沒填過的格子，不滿足的話就 traceback 回去到另一個分支找。

(4) DFS Rules with Logic (Method: “dfs_RL”)

經過很多版本修改之後，變成是(3)的規則改進版。

在每一橫列往下搜索之前，都用一套“規則流程”去推理盤面，將能填的格子先填上去，可能會填入白格也可能填入黑格，也可能留下 unknown 格。規則流程結束后檢查盤面是否全部填完：若全部填完則檢查所有 hint，符合的話就回傳結果，否則就 traceback 到前面一列；若沒有填完就繼續 branch 搜索某一列的所有組合。選擇最少組合的一列作為目前分支 branch 的一列。

也就是在(3)的演算法下加入兩個主要的改進：

a. 每次分支前都要使用“規則”來推理
b. 優先選擇排列組合數最少的一列來作為目前的搜索分支。

(5) BFS with Heuristic and Rules with Logic (Method: “bfs_heuristic_RL”)

該演算法是(4)的 A*改版。選擇最少排列的橫列，並且分支之後，進行“規則流程”來推理盤面。然後計算盤面的 unknown 格子數，unknown

格子數為盤面的估計函數，利用一個 priority，unknown 格子數少的盤面先搜索。

補充：關於 Rules With Logic (RL) “規則流程”，主要參考和改進自 (Chen, 2009) 和 (Wikipedia Nonogram, 2016)。大致如下：

Initial run range estimation

Let the size of each row with k black runs be n and the cells in the row with index $(0, \ldots, n-1)$, we can use the following formula to determine the initial range of each black run.

$$
\begin{aligned}
r_{1s} &= 0, \\
r_{js} &= \sum_{i=1}^{j-1} (LB_i + 1), \quad \forall j = 2, \ldots, k \\
r_{je} &= (n-1) - \sum_{i=j+1}^{k} (LB_i + 1), \quad \forall j = 1, \ldots, k-1 \\
r_{ke} &= n - 1
\end{aligned}
\tag{1}
$$

where LB_i is the length of black run i.

Using formula (1), we can get the initial ranges of the three black runs in Fig. 6(a), which are (0, 2), (2, 6), and (6, 9), respectively.

每一個 hint 都會對應到一個 run，根據 run 的範圍和重疊進行推理，將一定為黑格的部分填入黑格，一定為白格的填入白格，並且改變 run 的範圍。

總計完成了 Rule1.1~Rule3.3 (11 條 Rules) ＋ 自己添加的一條 rule。

Rule1.1: 觀察每一列一定為黑的就填入黑的

Rule1.2: 觀察每一列一定為白的就填入白的

Rule1.3: 觀察每一列，若有一個格子沒有 run 覆蓋就是白的

Rule1.4: 觀察每一列已經填黑格子，如何中間空一格沒填，假設填入會超出 hint 長度的話就填白

Rule1.5: 觀察每一列已填的黑格子，假設該格子旁邊一格再塗黑一下，如果剛好他們所在的 run 都相同長度也符合這個長度，那麼這串黑的兩旁塗白。

Rule2.1: 保證前一個 run 的“頭”一定在後面 run 的“頭”前面。

Rule2.2: 觀察每一列的 run，若 run 的第一格前面塗黑了，就將 run 第一格的範圍往後移動一格。（尾巴同理）

Rule2.4: 若 run 的頭已經塗白了，就將 run 第一格往後推。（尾巴同理）

Rule2.3: 如果一串黑格的長度大於目前的 run 所對應的 hint，就去看這串黑格屬於前面的 run 還是後面的 run，改變 run 範圍

Rule3.1:觀察每一列的 run，如果 run 內部和前面的 run 重疊部分中間有黑的，就把黑色之前的 unknown 也塗黑

Rule3.2 觀察每一列的 run，如果 run 內好幾個白格夾著的 unknown 格長度比 hint 少的話就將 unknown 格子塗白，如果在頭尾就縮短 run。
Rule3.3 觀察每一列的 run，如果 Run 內有有一個黑色並且前後距離不遠有白格就可以根據這個白格好 hint 長度去判斷有一些黑格附近也要塗黑。(等之類的變化規則)

每次進行這套規則流程，都可以將目前 nonograms 的盤面作 transpose、水平翻轉 flipr 從而繼續推理更多的格子出來。

這套流程寫完之後，容易有很多 bug 需要反復檢查，若可以推理的盤面可以很快求出解，若不能推理的盤面，該規則提升的速度不大(可能因為規則繁冗，反而增加了搜索的時間)。

3. Experiment

Experiment On CSIE Workstation Linux2 & Linux8

boardgen.py 5 10 0.5 0.3 12345 → 5x5
boardgen.py 10 10 0.5 0.3 12345 → 10x10

Method/n*n	Total Time (s)	Total Time (s)	Average Time (s)	Average Time (s)
	5x5	10x10	5x5	10*10
brute_force	13949	/	1394.9	/
dfs	0.517	/	0.0517	/
dfs_rowbyrow	0.0223	61.9	0.00223	6.19
dfs_RL	0.913	24.1	0.0913	2.41
bfs_heuristic_RL	0.5837	51.2	0.05837	5.12

boardgen.py 15 10 0.5 0.3 12345 → 15x15

15x15	dfs_RL (s)	bfs_heuristic_RL (s)
Case1	2.101	2.09
Case2	5.796	6.703
Case3	8.558	13.95
Case4	4.174	19.406
Case5	23524	28323

Case6	505	310
Case7	269	3599
Case8	5929	63.84
Case9	>10000	>10000
Case10	181	464.57

4. Game Complexity

- BruteForce: $O(2^{n^2})$ (n is the length of the nonogram’ s length)
- dfs: $O(2^{n^2}$ ） (Worst Case)
- dfs_rowbyrow: $O((n+1)!)$
- dfs_RL: $O((n+1)!)$ (Worst Case)
- bfs_heuristic_RL $O((n+1)!)$ (Worst Case)

5. Factor Affect the algorithm

- BruteForce:

  由於該 BruteForce 是從 00…00, 00…01 到 11…11，後面到前面往前面填，所以若 nonograms 的解答前面 0 比較多則較快找到解答。
- dfs:

  由於該 dfs 是從第一個格子為 1 開始往下填，若解答前面 0 比較多則會計算比較久。
- dfs_rowbyrow

  a. 每一橫列排列的組合可能性，若可能性越多，複雜度會越高，時間越久。

  b. prunning 很重要，只要往下沒有可能有結果，就要剪掉。若沒有剪掉，搜索時間會更久。
- dfs_RL

若初始盤面可以快速推理出來，使用“規則”可以非常快地解開題目，否則時間複雜度會接近 dfs_rowbyrow。例如 15＊15 的 case1~4。

- bfs_heuristic_RL

  “越接近填完的盤面先做”的 heuristic 效果上不是很好，相對比較 dfs_RL 而言對於 Case8 比較有效，主要是可以避免被 dfs 搜索到很深的地方卻沒有結果。若盤面填完的很少，推理效果不大時，priority queue 資料一大（Python 的 Priority Queue 內儲存超過 400~500 個 object）就會運算非常緩慢。

6. Other Observations

- Nonograms 最困難的部分在於生成 boardgame 時，機率在 0.3~0.7 中間所生成的盤面都會很難解，而且很有可能會有多種解。若機率範圍為 0.8 以上或 0.2 以下，推理規則在解決這類問題上會非常有用。
- Python 的運算效率較慢（而我的程式運行時間也比其他寫 python 的人慢很多），可能之後要用 c++來寫下次作業。

Reference:

＊ChenYu · Hui-Lung Lee · Ling-HweiChiung-Hsueh. (2009). An efficient algorithm for solving nonograms . Springer Science+Business Media, LLC 2009.

＊Wikipedia Nonogram. (2016 年 9 月 10 日). 擷取自 維基百科: https://zh.wikipedia.org/wiki/Nonogram