**Theory of Computer Games HW1**

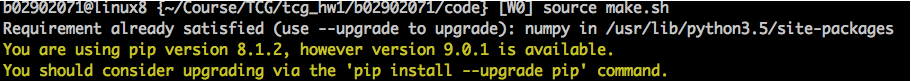
B02902071 陳柏堯

1. **How to Run?**
2. Set Python Environment

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

{./code} $ source make.sh

EX:



1. 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

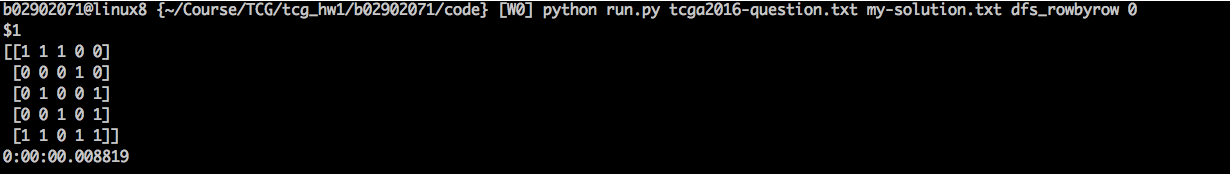
$Method is the method short name you want to try

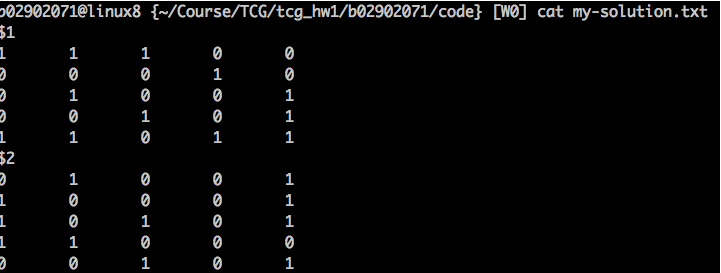
5 methods to use: {“bruteforce”/“dfs”/“dfs\_rowbyrow”/“dfs\_RL”/“bfs\_heuristic\_RL”}

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

( “0” represent all problems)

EX:





1. **Implemented Algorithm**
2. 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。

1. 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。

1. 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。

1. 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。

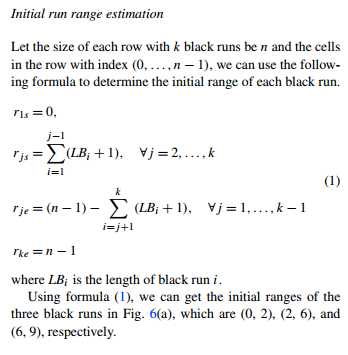
在每一橫列往下搜索之前，都用一套“規則流程”去推理盤面，將能填的格子先填上去，可能會填入白格也可能填入黑格，也可能留下unknown格。規則流程結束后檢查盤面是否全部填完：若全部填完則檢查所有hint，符合的話就回傳結果，否則就traceback到前面一列；若沒有填完就繼續branch搜索某一列的所有組合。選擇最少組合的一列作為目前分支branch的一列。

也就是在(3)的演算法下加入兩個主要的改進：

1. 每次分支前都要使用“規則”來推理
2. 優先選擇排列組合數最少的一列來作為目前的搜索分支。
3. 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：



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)。

1. **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 |

1. **Game Complexity**

* BruteForce: (n is the length of the nonogram’s length)
* dfs: (Worst Case)
* dfs\_rowbyrow:
* dfs\_RL: (Worst Case)
* bfs\_heuristic\_RL (Worst Case)

1. **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

1. 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。
2. 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。

1. **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