Computer Gaming Theory Homework 1

B02902105 資工四 廖瑋中

I. Environment Settings

The machine I use to run my program is NTU CSIE workstation (linux7.csie.ntu.edu.tw). Its operating system is Arch Linux.

The program language I use is Matlab R2016b [1], which is available on NTU CSIE workstation. I do not use any third-party tools. All the functions I use are included in Matlab.

In this paper, boardgen.py is used to generate Nonogram problems. The random seed I use in the experiments are all 12345. The max probabilities are all 0.5 and the min probabilities are all 0.35.

II. How to Run the Program

First, change directory to the *code/* directory. Subsequently, execute Matlab. One can use the function *nonogram_solver* to run the program. Here is its interface.

nonogram_solver(boardsize, boardnum, method) solves *boardnum* *boardsize*boardsize* Nonogram problems. *method* indicates the method used to solve the Nonogram problems. *method* can be a string of a method name, or a integer between 1 and 12, represented ID of different methods. The relations between the method IDs and method names are as below.

Method ID	Method Name
1	‘bruteforce’
2	‘bruteforce_rulecut’
3	‘bruteforce_wiki’
4	‘bruteforce_rulecut_wiki’
5	‘DFS’
6	‘DFS_rulecut’
7	‘DFS_wiki’
8	‘DFS_rulecut_wiki’
8	‘DFS_possiblecut’
10	‘DFS_possiblecut_wiki’
11	‘recursive_intersect’
12	‘recursive_intersect_greedy’

For example, if one wants to run brute-force search to solve 10 5x5 Nonogram problems, one can call *nonogram_solver(10, 5, ‘bruteforce’)* or *nonogram_solver(10, 5, 1).*

After executing the function *nonogram_solver*, a file named “solution.txt” will be generated, which records the solutions of the question file “tcga2016-question.txt”. Elapsed time and correctness of each sub-question will also be printed on the screen.

III. Methods

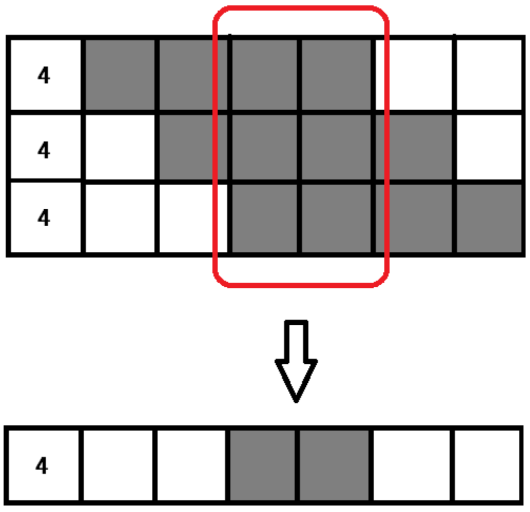
The methods I implement can majorly be classified into 3 categories, burte-force search, deep-first search (DFS), and intersection method. In the IV to VI section, I will discuss each method and their variances. In the Experiment section, I will demonstrate the results of the major methods and discuss their complexity in Discussion section.

IV. Brute-force Search

Brute-force search is the easiest and intuitive method to solve the puzzle. Nevertheless, it is really slow since the complexity exponentially grows. Therefore, I use two cutting method to accelerate it.

First, I stop my search if I am sure that the state is impossible to be a solution. To elaborate, every time I reach a new state, I check if the current state follows the rule. If the current state does not follow the rule, it is impossible to be a solution. Thus, I do not have to keep searching for the unknown part.

Secondly, I use the techniques to find simple boxes and simple spaces in the beginning. The techniques are written in Wikipedia. [2] Its idea is like this. If all the possibilities of a row or a column have a box in block i, then block i must be a box. Similarly, if all the possibilities of a row or a column have a space in block i, then block i must be a space. For example, if the rule indicates that there is a 4-length box in a 6-length row, then it implies that the middle 2 blocks must be boxes since the middle 2 blocks in all possibilities are boxes. Therefore, they must be boxes.



After I fix those determined boxes and spaces in the beginning, I start brute-force searching. However, this time I can skip those fixed boxes and spaces. Therefore, the searching process will become faster.

The average time they spend to solve a problem are as below.

Method	5 x 5
Brute-force	134.88 sec

Brute-force + rule cutting	0.39 sec
Brute-force + Wiki techniques	3.18 sec
Brute-focce + rule cutting + Wiki techniques	0.05 sec

One can see that brute-force with rule cutting and Wikipedia techniques is the fastest.

More discussion about complexity will be discussed in the Discussion section.

V. Deep-first Search (DFS)

DFS is one of the most popular methods in puzzle games. However, if one just naively searches for next step, it will be like brute-force method. To make DFS efficient, first I find out all possibilities of each row that follow the rules. [3][4] Then, I start DFS. In each row, I randomly pick one possibility and search the next row. Since I find all legal possibilities at first, I can avoid many impossible searches in brute-force method.

To accelerate DFS, I also apply rule cutting and techniques form Wikipedia. The rule cutting is a little different from brute-force search. In brute-force search, I check rules of the row and the column in each step. Nonetheless, in DFS I have found all possibilities that follow rules of rows. Therefore, I do not have to check them. All I need to do is to check the column rules.

Nevertheless, I am not satisfied with the rule cutting since it only returns when the current state breaks the rule. To provide a more efficient pruning method, I propose a new method that can prune possibilities that follow the rules in current state but will break the rule in the future.

At first, find out all possibilities of each row and column, which form a state space. Subsequently, I start DFS. Each time I randomly pick a possible row from the state space and put it in the board. Then, I check each column and remove some columns in the state space that does not fit the board. Then, go to the next row and repeat these steps again. [5]

This method is theoretically faster than DFS with rule cutting since each time when a row is put in the board, some column possibilities are removed from the state space. Therefore, the possibilities will become fewer and fewer. That is to say, fewer possibilities need to be checked, compared to DFS with rule cutting. As a result, it is faster than DFS with rule cutting.

The average time is as below.

Method	5 x 5	7 x 7
DFS	0.03 sec	21.04 sec

DFS + rule cutting	1.28 sec	> 20 min
DFS + Wiki techniques	0.01 sec	1.81 sec
DFS + rule cutting + Wiki techniques	0.61 sec	> 20 min
DFS + state space cutting	0.04 sec	104.53 sec
DFS + state space cutting + Wiki techniques	0.01 sec	4.22 sec

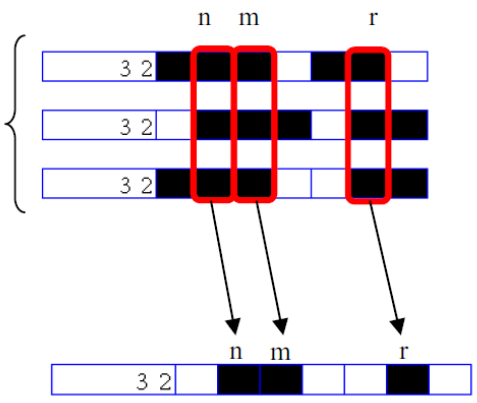
It is really surprising that DFS with rule cutting and state space cutting are even worse than DFS. I think this is because when one does rule checking or state space checking for each column, it has to scan over the whole row. Therefore, they act like brute-force with rule checking. DFS with rule cutting is even slower than brute-force search with rule cutting because it has to find out all possibilities of each row in the beginning. Therefore, rule cutting and state space cutting are not useful in DFS. However, the Wikipedia techniques are still useful. DFS with Wikipedia techniques is the fastest among all.

More discussion about complexity will be discussed in the Discussion section.

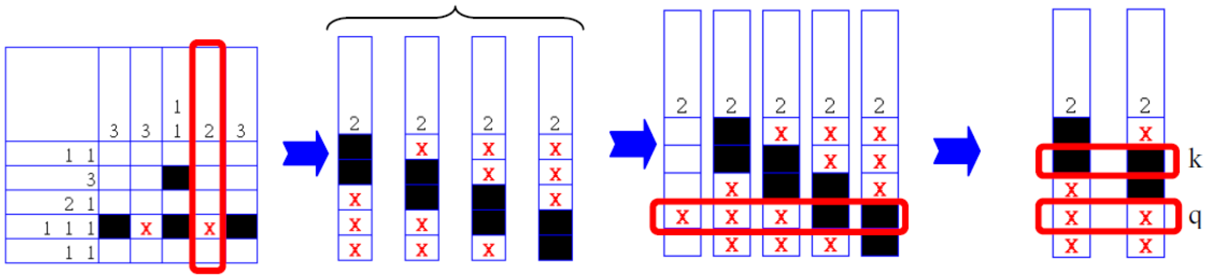
VI. Intersection Method

This is a method addressed by Yen et al. [6] The main idea of this method is similar to the Wikipedia techniques. The algorithm is as below.

At first, find out all possibilities of each row and column. Then, use the Wikipedia techniques, which are called intersection method in this method, to find out all the determined boxes and spaces in rows.



After determining all boxes and spaces in rows, remove impossible columns in the state space. After it, use the intersection method to find out all the determined boxes and spaces in columns, just as below.



After determining all boxes and spaces in columns, remove impossible rows in the state space. Then, repeat the steps. Determine boxes and spaces in rows; remove impossible columns; determine boxes and spaces in columns; remove impossible rows.

This method is extremely fast since it solves the puzzle in polynomial time, which is different from other searching methods, like brute-force search or DFS. Nevertheless, Wang et al. [7] point out that this method has a serious problem. The algorithm will converge even if some blocks are still undetermined. This usually happens because there are multiple solutions in the puzzle. Therefore, the algorithm cannot determine boxes and spaces for the blocks which may have multiple possibilities.

To solve this problem, I propose two methods. First, randomly assign an undetermined block to be a box. Then do the intersection method again. Since I force a block into a box, some possibilities may be removed due to the update. Then, the algorithm can keep going. If it eventually finds out that there is no solution, back to the situation we force a block to be a box. This time, it is sure that the block will definitely be a space. Then, the algorithm can keep going.

The second method I propose is similar to the first one. Nevertheless, instead of randomly assigning a box to an undetermined block, this time I compute the probabilities that each undetermined block can be a box (or a space). The probability is computed as following. If block i is in row r and column c and denote $f(i)$ to be a function that returns 1 if block i is a box and 0 if block i is a space, then

$$P(f(i)=1)=\frac{1}{2}\times\frac{\#\text{ of } f(i)=1\text{ in }R}{\#\text{ of } f(i)=1\text{ in }R+\#\text{ of } f(i)=0\text{ in }R}$$
$$+\frac{1}{2}\times\frac{\#\text{ of } f(i)=1\text{ in }C}{\#\text{ of } f(i)=1\text{ in }C+\#\text{ of } f(i)=0\text{ in }C}$$

where R is the collection of all possibilities of row r and C is the collection of all possibilities of column c. Similarly,

$$P(f(i)=0)=\frac{1}{2}\times\frac{\#\text{ of } f(i)=0\text{ in }R}{\#\text{ of } f(i)=1\text{ in }R+\#\text{ of } f(i)=0\text{ in }R}$$
$$+\frac{1}{2}\times\frac{\#\text{ of } f(i)=0\text{ in }C}{\#\text{ of } f(i)=1\text{ in }C+\#\text{ of } f(i)=0\text{ in }C}$$

After finding out all probabilities, assign the block i to be a box if $P(f(i)=1)$ has the highest probability among all, or assign the block i to be a space if $P(f(i)=0)$ has the highest probability among all. That is to say,

$$i_{opt}\,,b_{opt}=\underset{i,c}{\operatorname{argmax}}\big(P(f(i)=b)\big),\quad i\in\{\text{index of undetermined blocks}\}, b\in\{0,1\}$$

Then, assign b_{opt} into block i_{opt}. This is a greedy method since I try to find the most possible solution of next step.

The average, maximum, and minimum time of solving ten 15x15 puzzles are as below.

Method	Average time	Maximum time	Minimum time
Intersection	28.23 sec	257.50 sec	0.55 sec
Intersection + greedy	5.45 sec	24.43 sec	0.32 sec

One can see that the minimum time is near. However, the greedy one largely improves the maximum time. Therefore, greedy method is useful.

More discussion about complexity will be discussed in the Discussion section.

VII. Experiment Among the Three Methods

In this section, I will demonstrate experiment results between the 3 major methods. For each algorithm, I choose the best variance to represent it. That is to say, for brute-force search, I choose brute-force + rule cutting + Wikipedia techniques as representation. For DFS, I choose DFS + Wikipedia techniques as representation. For intersection method, I choose intersection + greedy as representation.

For the parameters of *boardgen.py*, the max probability I use is 0.5. The min probability I use is 0.35. The random seed I use is 12345. The average times each method spends solving different size of problems are shown as below.

Method	5 x 5	7x7	10 x 10	15 x 15	25 x 25
Brute-force	0.05 sec	> 20 min	> 20 min	> 20 min	> 20 min
DFS	0.01 sec	1.81 sec	> 20 min	> 20 min	> 20 min
Intersection	0.01 sec	0.03 sec	0.28 sec	5.45 sec	17 min

One can see that intersection surpasses brute-force and DFS very much. When it comes to 7x7, brute-force method becomes unable to handle the question. When it comes to 10x10, DFS becomes unable to handle the question, either. Nevertheless, even if it comes to 25x25, intersection method can still solve the question in 20 minutes. Thus, intersection method is largely surpasses other methods.

For brute-force search and DFS, one can see that DFS is better than brute-force search. When it comes to 7x7, brute-force is unable to solve it, but DFS can still solve it in 2 seconds.

In conclusion, intersection method is the fastest, followed by DFS, and brute-force search is the last.

VIII. Discussion

Denote the question is a $n \times n$ puzzle.

First, for brute-force algorithm, its time complexity is $O(2^{n^2})$ since it has to scan over all blocks. If k blocks are determined by the Wikipedia techniques, the time complexity will reduce to $O\left(2^{n^2-k}\right)$. However, it still grows exponentially, so brute-force search is tremendously slow. Its space complexity is $O(n^2 \times n^2) = O(n^4)$ since every step it needs to copy a board to the next step.

Secondly, for DFS, if there are k groups of spaces, m spaces in total in every row, then there are H_m^k possibilities for each row. The time complexity is $O(nH_m^k) = O(nC_m^{k+m-1}) \approx O(n(k+m)^m) < O(n(2n)^n) = O(n2^n n^n)$. The time complexity of DFS grows exponential, while the time complexity of brute-force search $O\left(2^{n^2}\right) = O((2^n)^n)$ grows double exponentially. Therefore, DFS is faster than brute-force search. The space complexity of state space is $O(nH_m^k) = O(nC_m^{k+m-1}) \approx O(n(k+m)^m) < O(n(2n)^n) = O(n2^n n^n)$, and the space complexity of current board is $O(n \times n^2) = O(n^3)$ since for each step, one copy of the board is needed. Therefore, the total space complexity is $O(n2^n n^n + n^3) = O(n2^n n^n)$.

Finally, for intersection method, assume that each time the possibilities that one has to check in a row or a column is t. Then, the time complexity of solving a puzzle is $O(n^2 \times 2n \times t) = O(2n^3 t)$ if at least one block is determined every time it checks for all possible rows and columns. t seems to be H_m^k due to the previous discussion. Nevertheless, every time when some blocks are determined, some possibilities are removed from the state space, which leads to a decrease in t. Therefore, the average t is actually much smaller than H_m^k. Nevertheless, in the previous section, one also knows that sometimes the intersection method converges before all blocks are determined. Assume there are g guesses during the algorithm. Then the time complexity is $\mathrm{O}((2n^3t)^g)$, which is still smaller than the ones of brute-force search and DFS. The space complexity of state space is $\mathrm{O}(2nH_m^k)$ if there is no guess during the algorithm. If there are g guesses during the algorithm, the space complexity of state space becomes $\mathrm{O}(2gnH_m^k) < \mathrm{O}(gn2^nn^n)$. The space complexity of current board is $\mathrm{O}(g \times n^2) = \mathrm{O}(gn^2)$. Therefore, the total space complexity is $\mathrm{O}(gn2^nn^n + gn^2) = \mathrm{O}(gn2^nn^n)$.

In conclusion, for time complexity, intersection method is the fastest, followed by DFS, and brute-force search is the last. For space complexity, brute-force search saves the space the most, followed by DFS, and the intersection method is the last.

Next, I am going to discuss factors that affect the performance of algorithms and difficulty of Nonogram.

For factors that affect the performance of algorithms, I think there are two. The first is the way to prune possibilities. In brute-force search, only determined blocks are skipped. In DFS, I find out all possibilities of each row at first, which also means that some impossible rows and columns are also pruned in the beginning. For the intersection method, I keep pruning the state space during the algorithm. Therefore, the more you prune the state space, the faster the algorithm will be. The second one is how you decide your next step. In intersection method, one can see that the performance of bad cases and average time improve a lot if greedy step is used. If one can decide the next step more accurately, the algorithm will become much faster since it can avoid many less possible searches, and therefore the expected time decreases.

For factors that affect the difficulty of Nonogram, I think there are three factors. The first is the size of the puzzle. This is obvious. If the size of the puzzle is larger, more time has to be spent solving the question. The second is the number of solution. If there is only one solution, many heuristic methods are easily applied to find determined blocks. Nevertheless, if there are multiple solutions, many blocks will become hard to determine. Therefore, more solutions lead to higher difficulty. The third is the probability that boxes appear in the question. If there are many boxes in the question, then it will be easier to determine boxes in the question. Similarly, if there are many spaces in the question, then it will be easier to determine spaces in the question. Nevertheless, if the probability of boxes and spaces are similar, it will become difficult to determine them. Therefore, if the probability of boxes is too high or too low, it will become easier to solve the puzzle.

References

[1] Matlab - MathWorks

https://www.mathworks.com/products/matlab/

[2] Nonogram - Wikipedia

https://en.wikipedia.org/wiki/Nonogram

[3] The idea of implementation of finding all possibilities of a row comes from B02902071 Po-Yao Chen（陳柏堯）.

[4] Uniquely generate all permutations of three digits that sum to a particular value? - MathOverflow

http://mathoverflow.net/questions/9477/uniquely-generate-all-permutations-of-three-digits-that-sum-to-a-particular-valu

[5] The idea comes from B02902011 Hung-Yen Huang（黃泓硯）.

[6] Yen, S. J., Su, T. C., Chiu, S. Y., & Chen, J. C. (2010). Optimization of Nonogram’s Solver by using an Efficient Algorithm. *Conference on Technologies and Applications of Artificial Intelligence (TAAI).* doi: 10.1109/TAAI.2010.95

[7] Wang, W. L., & Tang, M. H. (2014). Simulated Annealing Approach to Solve Nonogram Puzzles with Multiple Solutions. *Procedia Computer Science, 36*, 541-548. doi: 10.1016/j.procs.2014.09.052