SUTD 50.004 2D 2SAT ALGORITHM ANALYSIS

Presented by :

Li Yanzhang 1003932
Luo Yifan 1002975
Tang Xiaoyue 1002968
Wang Tianduo 1002963
Zhu Bo 1002856

Table of Contents

1. Abstract

This report will show an implementation of 2-SAT problem consisting of four parts: explanation to algorithms, design and implementation in java, result and performance analysis and finally conclusion.

2. Explanation to algorithms

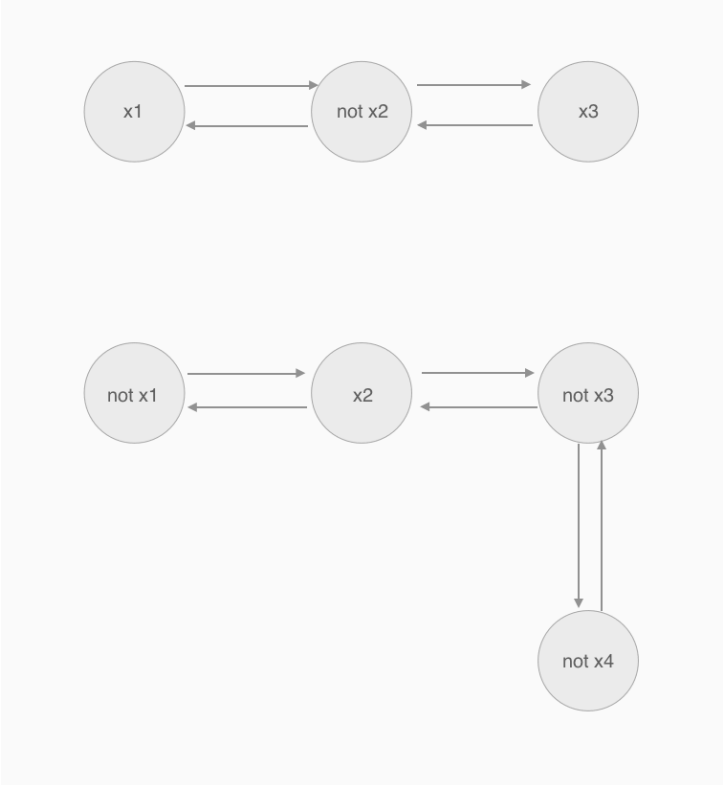
2.1 2-Satisfiability (2-SAT) Problem

2-SAT is a special case of Boolean Satisfiability Problem and can be solved in polynomial time in the number of variables and clauses and it limits the problem of SAT to only those boolean formula which are expressed as a CNF with every clause having only 2 terms (also called 2-CNF).

2.2 Approach for 2-SAT Problem

To solve a 2-SAT problem, one approach is to express the Conjunctive Normal Form (CNF) as an Implication by creating an Implication Graph in which every clause has two edges.

An example Implication Graph of a boolean formula (x1 or x2) and (not x2 or not x3) and (x3 or not x4) will be:



Consider the following cases:

CASE 1: If edge(x —> not x) exists in the graph. This means if x = TRUE, not x = TRUE, which is a contradiction. But if x = FALSE, there are no implication constraints. Thus, x = FALSE

CASE 2: If edge(not x —> x) exists in the graph. This means if not x = TRUE, x = TRUE, which is a contradiction. But if not x = FALSE, there are no implication constraints. Thus, x = TRUE

CASE 3: If both exist in the graph. x needs to be TRUE and at the same time FALSE. Thus, is it impossible

Therefore, if x and not x are on a cycle then the CNF is unsatisfiable. Otherwise, there is a possible assignment and the CNF is satisfiable.

When using implication: If both x and not x lie in the same SCC (Strongly Connected Component), the CNF is unsatisfiable. (SCC: A graph is said to be strongly connected or diconnected if every vertex is reachable from every other vertex.) Otherwise, it is satisfiable.

Thus, to solve 2-SAT problem is to determine the SCC and group vertices that are along the same path.

2.2.1 Kosaraju's Algorithm & Depth-First-Search (DFS)

In order to find all strongly connected components, one way is using Kosaraju's algorithm.

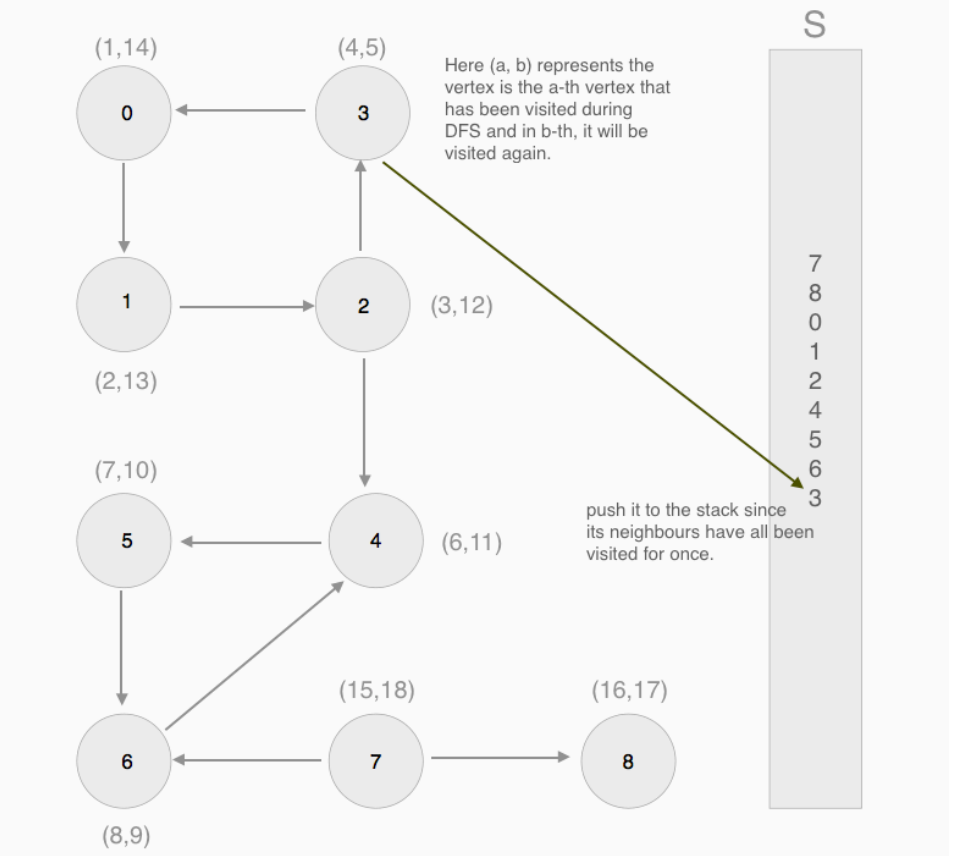
1) Create an empty stack 'S'

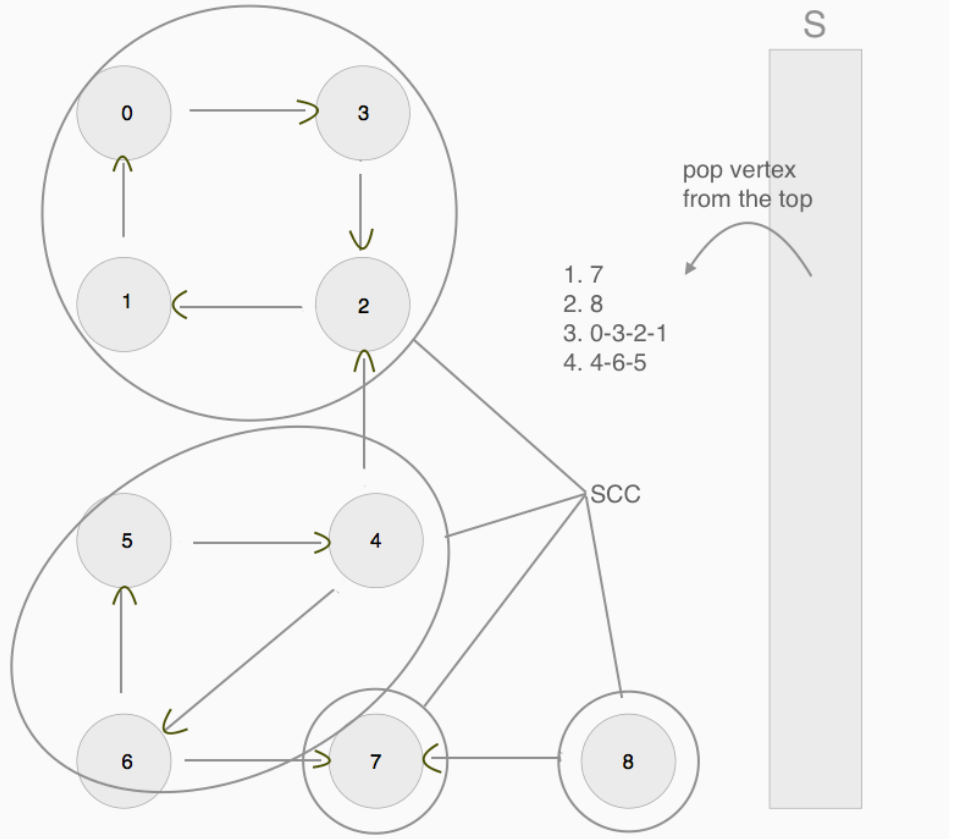
2) Do a DFS traversal of the graph and push the vertex to the top of the stack 'S'.

3) Reverse directions of all arcs to obtain the transpose graph and again do a DFS traversal.

4) One by one pop a vertex from 'S' while it is not empty. Let the popped vertex be the source to do a DFS. Store the vertices within the same SCC together.

Example on how this process will be done:


S
(1,14)
(4,5)
Here (a, b) represents the vertex is the a-th vertex that has been visited during DFS and in b-th, it will be visited again.
0
3
1
2
(3,12)
(2,13)
7
8
0
1
2
4
5
6
3
(7,10)
5
4
(6,11)
push it to the stack since its neighbours have all been visited for once.
(15,18)
(16,17)
6
7
8
(8,9)


S
0
3
pop vertex from the top
1
2
1. 7
2. 8
3. 0-3-2-1
4. 4-6-5
SCC
5
4
6
7
8

2.2.1 Time Complexity

1. Do a DFS traversal for each vertex takes O(V + E) where V stands for the number of vertices and E stands for the number of edges.
2. Computing G transpose takes O(V + E)
3. Do a DFS traversal for each vertex for G transpose takes O(V+E)
4. Output Vertices as separate SCC's.

Therefore, Overall Time Complexity = O (V+E)

2.2.2 Question: Why this algorithm is not working for 3-SAT but only for 2-SAT?

Our approach is to express the CNF as an Implication by creating an Implication Graph which has 2 edges for every clause of the CNF. The reason to use this method is because for 2-SAT problem, the relationship between two literals inside the same clause is confirmed as long as the boolean formula is given. For example, for (not x1 or x2), when x1=true, x2 must be true and when x2=false, x1 must be false too. Therefore it is possible to draw Implication Graph. However, for 3-SAT problem, the relationship between literals inside the same clause can not be confirmed, e.g. (not x1 or x2 or x3), when x1 is false, either x2 or x3 can be true. Thus, it can not be done by Implication Graph. That is why the algorithm only works for 2-SAT and not 3-SAT.

3. Design & implementation in Java

The files that are needed to run the 2-SAT Solver are as follows:

1. ReadFile.java
2. ImplicationGraph.java
3. SearchSCC.java
4. TwoSATSolver.java

3.1 File Processing

ReadFile.java will take in files in cnf format and read it line by line. It stores the number of variables inside a var called variables and number of total clauses inside a var called clauses. Furthermore, a clause who has a form of (L1, L2) will be stored separately as two ArrayList Literal1 and Literal2 in which the index indicates which clause it belongs to.

3.2 Plot Implication Graph

Two HashMaps are created to represent the graph and the reverse graph. Inside each HashMap<Int1, List<Int2s>>, the In1 represents the source of the edge, and all the Int2s inside the List are these destinations Int1 points to. For example, HashMap<x1, [x2,x3]> means Edge(x1—>x2) and Edge(x1—>x3) exist in the graph.

3.3 Determine Strongly Connected Components (SCC)

SearchSCC.java has two DFSs and one satisfiability check. The first DFS will start from an input vertex source and mark all the following vertices. After marked all the vertices that are reachable from the source, it will push the vertex whose neighbourhoods are all visited to the top of the stack. The second DFS will do the same thing to a reverse graph except no pushing vertices to stack but instead it will store which SCC the vertices belong to. Satisfiability will check whether a x and not x are inside the same SCC, if it is, return false because if x and not x are on a cycle then the CNF is unsatisfiable. Otherwise return true.

3.4 Result presentation and time recording

TwoSATSolver.java will combine all the methods above to do a testing and return the result as well as the time spent.

4. Bonus — Randomising algorithm

4.1 Random Walk Algorithm

There are 3 steps to perform a random walk algorithm in our analysis:

1. Set all the literals to be true.
2. Test the satisfiability of CNF.
3. If the CNF is unsatisfied: Flip the boolean value of a random literal and go back to step 2. Otherwise it is satisfied.

Note: In the Random.java using for preform a random walk algorithm, in order to avoid running forever in the case of unsatisfiable, it will stop automatically after square of number of variables times of flipping.

4.2 Time Complexity

The time complexity of random walk algorithm depends on the number of literals whose boolean value have been flipped. We can check that the average amount of steps to walk (randomly) from 0 to n is just n^2 . On average, we will find a solution in at most n^2 steps.

Therefore, the time complexity of a random walk algorithm is $O(n^2)$ on average where n is the number of variables.

5. Results & Performance analysis

5.1 Performance of the two algorithms on similar inputs

Satisfiability	Kosaraju's Algorithm	Random Flip
SA testing1	4.585869 ms	0.556265 ms
UNSA testing1	0.362814 ms	1.509554 ms
SA testing2	2.090335 ms	0.105522 ms
UNSA testing2	4.16692 ms	254.349462 ms

5.2 Time complexity analysis

From the result, a general trend can be seen:

WHEN IT IS SATISFIABLE:	Time taken by Kosaraju's Algorithm > Random Flip
WHEN IT IS UNSATISFIABLE:	Time taken by Kosaraju's Algorithm < Random Flip

From Time Complexity point of view, when total number of literals is n, time taken by Kosaraju's Algorithm is around O(n), while Random Flip is $O(n^2)$ on average. Therefore, if dealing with large number of CNFs containing unknown satisfiability, Kosaraju's Algorithm is preferable while dealing with small number of satisfied CNFs, random algorithm might be applicable.

6. Conclusions

6.1 Summary

This report has talked about an implementation of 2-SAT problem using Kosaraju's Algorithm and its design and implementation in java as well as result and performance analysis comparing with Random Flip Algorithm. The result shows when the CNF is satisfiable, Random Flip Algorithm will usually be faster while when the formula is unsatisfiable, Kosaraju's Algorithm will be quicker.

6.2 Question: Is the randomising algorithm a practical substitute for the deterministic one?

It depends on two factors :
1. the number of the CNFs/the size of each CNF.
2. the satisfiability if the CNF

When dealing with large number of cases, time complexity is applicable. Hence, Kosaraju's Algorithm with time complexity O(n) is generally faster. Also in a more intuitive way, Random walk will waste a lot of time on unsatisfiable cases. However, if dealing with CNFs that most of them are satisfiable and each of them has a small size, Ransom Algorithm will be preferable since Kosaraju's Algorithm will waste time on building Implication Graph and searching for SCCs.
Conclusion: only when dealing with CNFs that most of them are satisfiable and each of them has a small size will randomising algorithm be a practical substitute.

7. References

https://sites.fas.harvard.edu/~libcs124/CS/lec15.pdf

https://en.wikipedia.org/wiki/2-satisfiability

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

https://www.geeksforgeeks.org/strongly-connected-components/

https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/