Term4 2D PROJECT

50.004: Algorithms

Group 8

STUDENT ID	NAME
1005401	Erick Chandra
1004885	Guo Yuchen
1005254	Visshal Natarajan
1004866	Wang Yueheng
1004875	Xiang Siqi
1002141	Zach Lim

1. Description of Problem

A Boolean Satisfiability Problem, or SAT, is the problem of assigning Boolean values to a set of variables in order to satisfy a set of Boolean equations. A 2-SAT Problem is defined either as one where each clause in a given formula has exactly 2 literals, or one where each clause has at most 2 literals. Despite this difference, the two definitions will be shown to be equivalent, in that both can be solved in Polynomial Time.

2. Overview of Algorithm

Our implementation of a 2-SAT solver uses 2 main ideas: Implications arising from each clause in a 2-SAT problem, and Kosaraju's Algorithm for Strongly Connected Components (SCC) which utilizes Depth-First Search (DFS). The algorithm is written in Python(see README.md in the zip), and can be divided into 3 sections, which we will cover in more detail below.

Section 1. Parsing the CNF File

Our Algorithm is designed to work with problems input using the CNF Format as given.

First, it creates an empty list of clauses and a variable denoting the number of the highest literal (initialised as 0). It then reads the CNF file and appends it line by line to a list, creating a list of strings. It ignores lines beginning with "c" or "p", which signify comment lines and problem description lines respectively.

Then, for each string in the list, the string is parsed character by character: If the character is "0", an empty clause is appended to the clause list. Else, that character is appended as a literal to the last clause in the list of clauses. The results are the number of literals, number of clauses, and a list of clauses that can then be input into the rest of the algorithm.

Section 2. Determining Satisfiability

The second part of our algorithm involves converting the list of clauses into a Graph through the Implications that arise from each clause in the problem. For example, given a clause $(a \vee b)$, we can see that if either a or b are False, then in order for the clause to be satisfiable, the other is *implied* to be true. In other words, the clause above can also be written as $\neg a \Rightarrow b \wedge \neg b \Rightarrow a$, where the double right arrow represents an implication. Note that $a \Rightarrow b$ **does not** mean that $b \Rightarrow a$. Thus, we can now create a graph where each literal and its negation are separate vertices, and the implications are the directed edges between them.

Once the graph has been generated, we apply Kosaraju's Algorithm to find Strongly Connected Components (SCC), which is defined as a sub-graph composed of vertices that can all reach every other vertex in that component. Single nodes are by definition SCCs, with just that node as a member. This is done using DFS twice, first on the original graph, then on its Transpose (The Transpose of a directed graph is defined as another directed graph with the same set of vertices, but with all edges reversed). In our algorithm, the two graphs are generated at the same time by giving each vertex two sets of *children* and *visited* attributes: one for the original graph and one for the transposed graph. A stack is also utilized to determine the order in which to run DFS on the transposed graph.

During the first DFS traversal, nodes are pushed into the stack in the order that they are finished i.e. when a node has no more unvisited children. The second DFS is then run on the transposed graph, in the order that nodes are popped from the stack. The nodes are again appended to a list in their finishing order, with a marker node denoted by "SEP" appended between iterations (after each backtrack). We see that any consecutive nodes in the list form an SCC, and SCCs are separated by the "SEP" node.

The reason this works is that when a graph is transposed, nodes in an SCC will remain in the same SCC, while links between SCCs will be "broken".

Since each directed edge is an implication, we realise that if a literal n and its negation $\neg n$ are in the same SCC, there is a contradictory loop formed where both n and $\neg n$ both must be true. In this case, the problem has no solution. Thereby we can classify the problem as **Unsatisfiable**.

Section 3. Generating a Solution

If there are no SCCs that contain both a literal and its negation, this is in fact sufficient to say that the formula is **Satisfiable**. The proof for this is as follows:

First, we note that if there is an implication $a \Rightarrow b$, then there necessarily is also the implication that $\neg b \Rightarrow \neg a$.

Next, we sort the SCCs in topological order, that is $SCC[a] \geq SCC[b]$ if there exists a path from Node a to Node b. Since it is possible that a vertex a can reach $\neg a$ (as long as the opposite is not simultaneously true, otherwise they would be in the same SCC and we would already have determined the problem to be unsatisfiable), we realise that for there to **not** be a contradiction, a has to be assigned as false. Let us assume that this is the case.

Now, we need to prove that there does not exist a variable x, such that both x and $\neg x$ are reachable from $\neg a$ (since we assigned a as false). Using the first statement from this proof, if we were to assume that both x and $\neg x$ are reachable from $\neg a$ (i.e. $\neg a \Rightarrow x$ and $\neg a \Rightarrow \neg x$), this implies that $\neg x \Rightarrow a$ and $x \Rightarrow a$ are also true. This would lead to a contradiction, as it means $\neg a$ can reach a, which violates our second statement.

Thus, we have proven that if a literal a and its negation are not the in the same SCC, then for any Node a, if a can reach $\neg a$ then $\neg a$ cannot reach a. We can then "fix" $\neg a$ as part of the solution. We have then subsequently proven that no Node x exists such that x and $\neg x$ such that both can be reached by $\neg a$. Applied to every node, this means that as long as every SCC does not contain both a literal and its negation, we can assign every literal a value that does not cause a contradiction, proving the existence of a solution.

With the above proof, arriving at a solution is done by assigning the **negation** of the variables encountered for the first time in topological order.

Section 4. Pseudocode and Example

Overview of Kosaraju 2-SAT Solver:

1. Create an implication graph, each variable creates two vertices (The positive and its negation)
2. For each clause (a or b)
   Create edge $\neg a \rightarrow b$ and $\neg b \rightarrow a$
3. Run SCC algorithm (Algorithm to determine strongest connected components)
4. If no conflict, the 2-SAT instance is satisfiable, thus find a solution
5. If a variable and its negation are in the same SCC, the formula is unsatisfiable.

The step by step breakdown of our algorithm for a given test input is shown below.
Let us consider the following test input:

```
P cnf 3 4
1 2 0
-2 -3 0
-1 3 0
3 -2 0
```

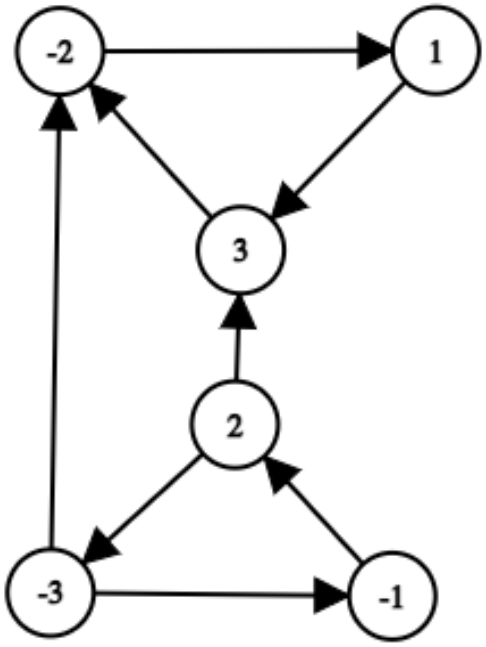
This is equivalent to the following equation in CNF form:

$$(1 \vee 2) \wedge (\neg 2 \vee \neg 3) \wedge (\neg 1 \vee 3) \wedge (3 \vee \neg 2)$$

This can then be written as a series of implications:

$\neg 1 \Rightarrow 2$ $\quad 2 \Rightarrow \neg 3$ $\quad 1 \Rightarrow 3$ $\quad \neg 3 \Rightarrow \neg 2$
$\neg 2 \Rightarrow 1$ $\quad 3 \Rightarrow \neg 2$ $\quad \neg 3 \Rightarrow \neg 1$ $\quad 2 \Rightarrow 3$

Step 1: We Generate an implication Graph as follows



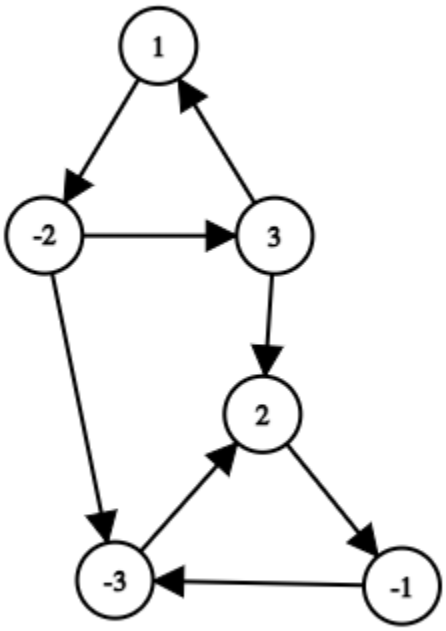
Step 2: Perform DFS traversal of the graph. Push node to stack before returning.

We can perform DFS by starting at the node with value 1. Once all the neigbours of a given node are visited, that particular node is pushed into the stack. Thereby the completed stack is as follows:

First in Last in

-2	**3**	**1**	**-3**	**2**	**-1**

Step 3: Transpose graph by reversing the edges.



Step 4: Pop nodes one by one from the stack, do DFS again on the modified graph.

From the stack, we can clearly see that -1 is popped out first and DFS is performed on -1. Thereby the steps are as follows:

1. We set the value of node -1.visited = True
2. We append node -1 to our SCC array (Array to store strongly connected components)
3. We visit the children of node -1 which is node -3. Now recursive DFS is invoked on node -3. The value of node -3.visited = True. Steps 2 & 3 are repeated for node -3, correspondingly node 2 is added into the SCC array. From node 2 it is clear node 2 has no other neighbours. As a result, “SEP” is inserted into our SCC array. SCC = [-1,-3,2,”SEP”]
4. We pop the next element in the stack which is 2 which is already visited as a result “SEP” is added onto the SCC array, correspondingly we also visited the next element in the stack which is -3. Hence the SCC array is as follows: SCC = [-1,-3,2,”SEP”,”SEP”,”SEP”,]
5. The next element to be popped is 1. Steps 1,2,3 are repeated and we continue the process until the stack is empty
6. Our Final SCC Array: SCC = [-1, -3, 2, 'SEP', 'SEP', 'SEP', 1, -2, 3, 'SEP', 'SEP', 'SEP'], which is then cleaned up to be SCC_ls = [[-1,-3,2],[1,-2,3]]

Step 5: Check if two complements exist in a single SCC. In this case, complements do not exist, the problem is satisfiable.

Step 6: Generate solution. From here we assign the negation of the values to the literals from the topmost SCC to get a solution: **[1: True, 2: False, 3: True]**.

3. Performance Analysis

Step 1: Generate implication graph and its transpose: ***O(n+m)***.
All nodes(n) and edges are added in the graph, and edge is corresponding to the number of clauses(m).

Steps 2 & 3: DFS traversal for 2 times: ***O(n+m)***.
Search through all nodes and their edges.

Step 4: Check complements in SCC: ***O(n)***.
We converted the separated SCCs to a hash table with 2n entries.

Step 5: Assign solution: ***O(n)***.
We loop through all nodes to assign false.

The overall time complexity will be $O(c1*n+c2*m+c3)$, where $c1$, $c2$, $c3$ are positive integers.

Thus, the time complexity of the whole algorithm is ***O(n+m)*** where n is the number of vertices and m is the number of edges in the implication graph. Thereby the algorithm has **polynomial time complexity**.

4. Equivalence of 2-SAT & 1-SAT Problem:

In the case of the 1-SAT problem, there would be one literal for every clause and for the function to be true, we'd require every literal value to be true. Now the 1-SAT problem would definitely not be satisfiable if a variable and its negation are part of the same CNF form. Hence 1-SAT problems are always solvable provided a variable and its negation are not part of the CNF form. Thereby the literals would form an SCC where we'd have to check whether its negation is part of the same input. This indicates that the algorithm for 2-SAT can easily be applied for a 1-SAT problem in polynomial time complexity.

5. Difference between 2-SAT and 3-SAT

In the case of a 3-SAT problem, where each clause has at most 3 literals, we realise that we cannot employ the same algorithm. This is because, for a 2-SAT problem, we only need to fix one literal to form an implication for the other literal, and this by extension works for that other literal in the clause as well. In other words, for every implication caused by negating the first literal, there will be another implication formed by negating the second literal (i.e. if $a \Rightarrow b$, then $\neg b \Rightarrow \neg a$). This is the first statement in our Proof of correctness in the "Generating a Solution" section above.

On the other hand, for a 3-SAT problem, two of the literals in a 3-literal clause must be fixed before any implications can be constructed. For example, considering the clause $(a \vee b \vee c)$, we can try to come up with the following implications: $\neg a \wedge \neg b \Rightarrow c$, $\neg a \wedge \neg c \Rightarrow b$, and $\neg b \wedge \neg c \Rightarrow a$. Alternatively, we can fix a single literal, but it will only imply that at least one of the two remaining literals must be true. In this case, we get the following: $\neg c \Rightarrow a \vee b$, $\neg b \Rightarrow a \vee c$, and $\neg a \Rightarrow b \vee c$.

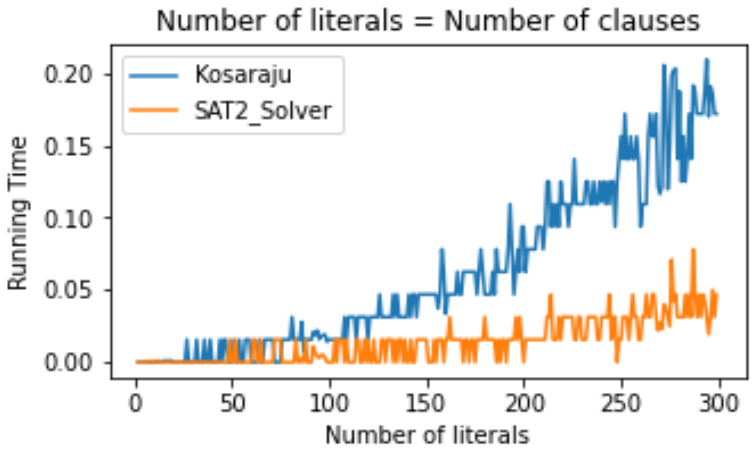
Thus, we see that in either case, we cannot put individual literals into separate vertices in a graph and draw any meaningful edges between them. If we tried to group the two-literal expressions together into a single node, and connect them to their single-literal implication, we will never form any SCCs. This is because single literals will only imply an "OR" relationship between the other two literals, whereas we needed an "AND" relationship to create an implication for a single literal.

6. Bonus

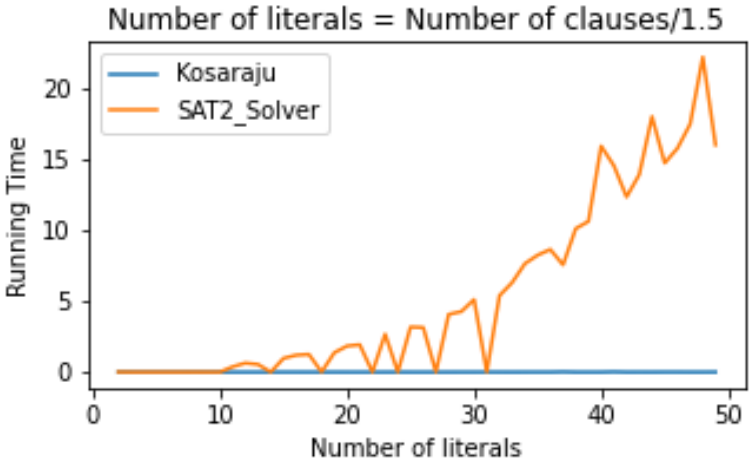
Another way to solve a 2-SAT problem is by "brute-force". That is to say we set all literals to be false initially, and randomly assign true literals to reduce the problem scale. If we take one change to the literal truth table as a 'step', since 2-SAT expression has $O(n^2)$ clauses, each step takes $O(n^2)$ time. Technically, this random algorithm will keep changing the truth table until it find a valid result. However, we set the maximum step to $100 * number\ of\ literals^2$. It returns "None" if the step exceed the maximum step as it is very likely that this 2-SAT has no solution.

To compare the performance of bonus SAT2_Solver and Kosaraju's algorithm, we randomly generate test cases with different number of literals and clauses.

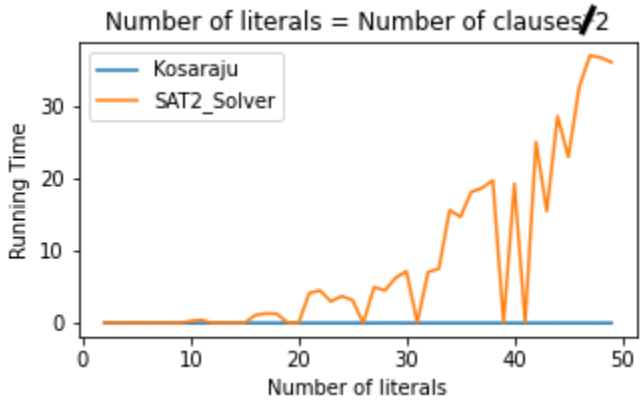
Experimentation data and results are shown below (Running time is measured in seconds,and the spikes or 0s are mainly caused by random problem complexity, which can be omitted as we are looking at the general trends.)



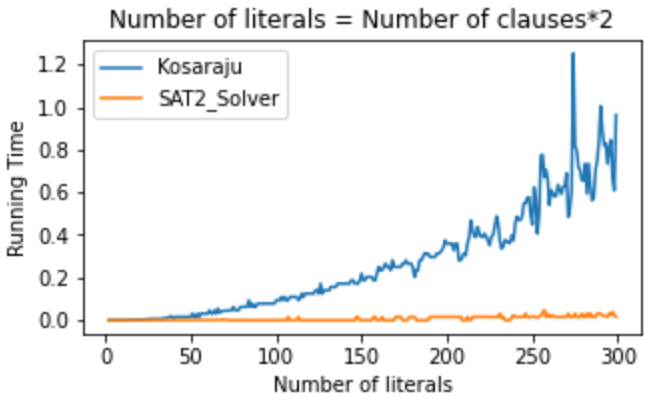
Case 1: number of literals equals to the number of clauses



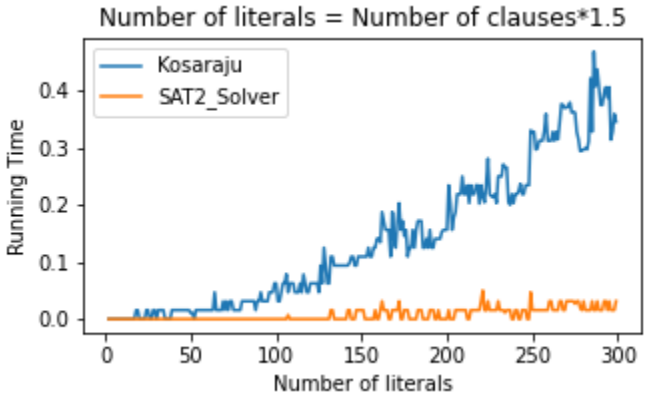
Case 2: number of literals equals to the number of clauses over 1.5



Case 3: number of literals equals to the number of clauses over 2



Case 4: number of literals equals to the number of clauses times 2



Case 5: number of literals equals to the number of clauses times 1.5

Although due to device restraints, we couldn't run large enough number of test cases, we can still conclude that when the number of literals >= the number of clauses, SAT2_Solver performs *slightly better* than Kosaraju's algorithm, but as the number of clauses increases, Kosaraju's algorithm performs *significantly better* than bonus SAT2_Solver. Above indicates that the amount of increase in the running time of the brute-force algorithm is significantly larger than the amount of increases in the ratio $\frac{number\ of\ clauses}{number\ of\ literals}$, since the program needs to loop through all unsatisfied clauses every time it randomly chooses a literal to be true.