50.004 - Introduction to Algorithms

2D Design Challenge – 2SAT Solver Report

**Introduction**

2-Satisfiability (2-SAT) is a subset of satisfiability problems where there are only 2 input variables per clause. Interestingly, unlike other n-SAT problems (where n>2), the 2-SAT problem can be solved in Polynomial Linear Time (P) whereas all the other cases can only be solved in Non-Polynomial Time (NP).

A 2 SAT problem with n clauses can be written in Conjunctive Normal Form (CNF) as follows:

“Wedge” – AND Operator

“Vee” – OR Operator

“Not” – INV Operator

**Method**

The 2-SAT Problem can be solved in Polynomial Linear Time using Kosaraju’s Algorithm, which is essentially a Depth First Search (DFS) algorithm. Kosaraju’s Algorithm aims to find strongly connected components (SCC) (Fig. 1) of a directed graph.

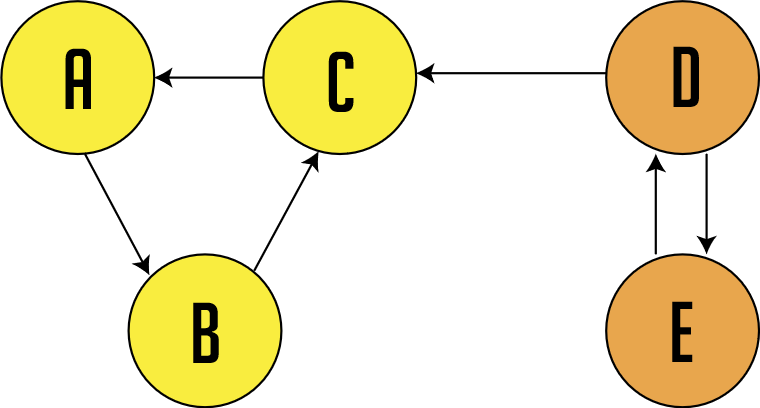


Figure Directed graph with 2 Strongly connected components (A,B,C) and (D,E)

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 2 SCCs in Fig. 1.

**Construction of Graph from CNF**

Create a graph G = (V, E) with 2n vertices. Intuitively, each vertex resembles a true or not true literal for each variable. For each clause (A V B), where ‘A’ and ‘B’ are literals, create a directed edge from ‘¬ B’ to ‘A’ and from ‘¬ A’ to ‘B‘

Example

Given,

Take the first clause , create 2 edges

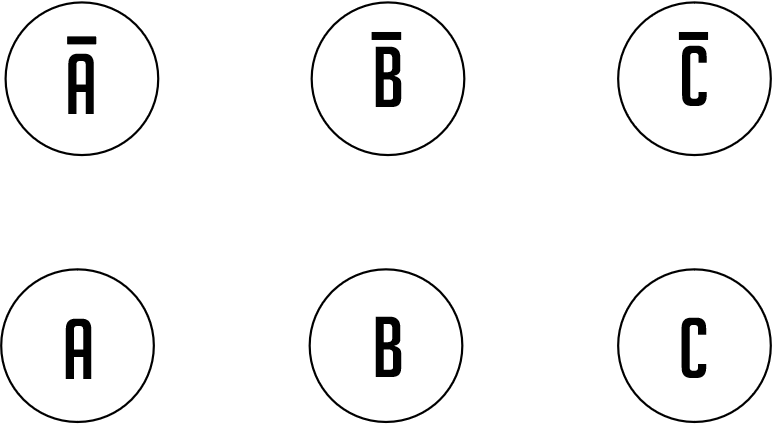


Figure Graph with 2n Vertices

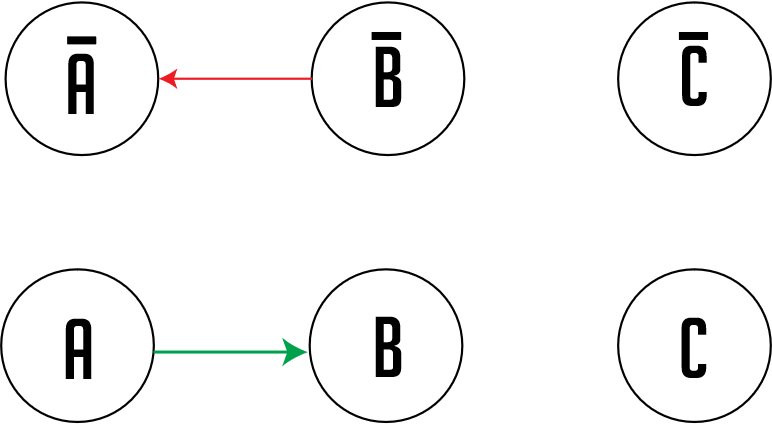
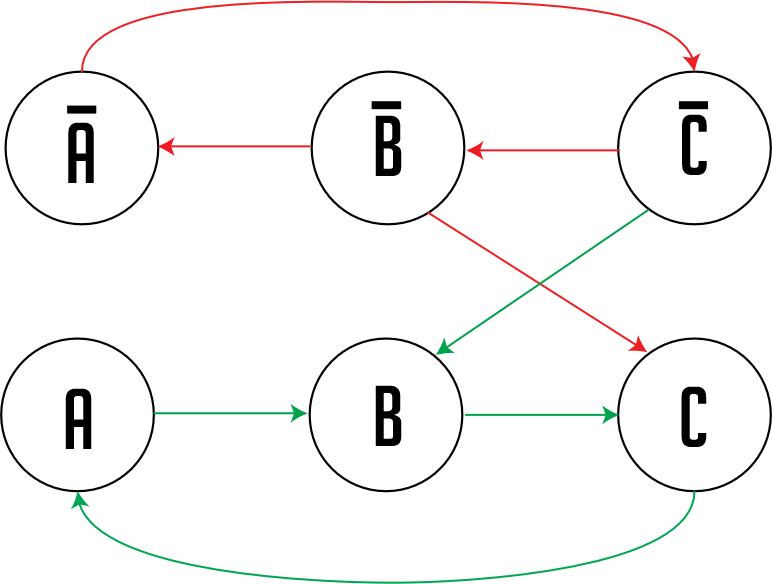


Figure Connecting Edges

and so on…

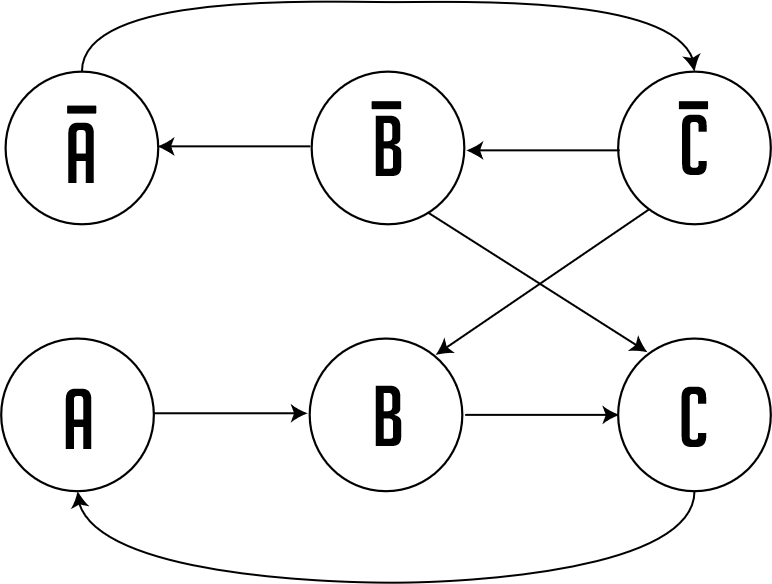


**Kosaraju’s Algorithm**

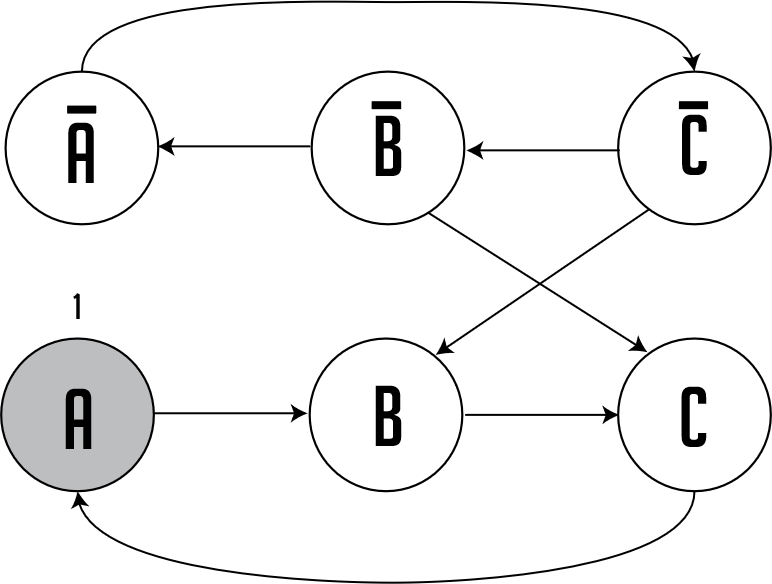
1. Create an empty stack S
2. Perform DFS on the graph. In DFS traversal, after calling the recursive DFS for adjacent vertices of a vertex, push the vertex to S.
3. Reverse directions of all arcs to obtain the transpose graph.
4. One by one pop a vertex from S while S is not empty. Let the popped vertex be V. Take V as the source and perform DFS starting from V. All vertices that are connected to V are in the same SCC.

Example

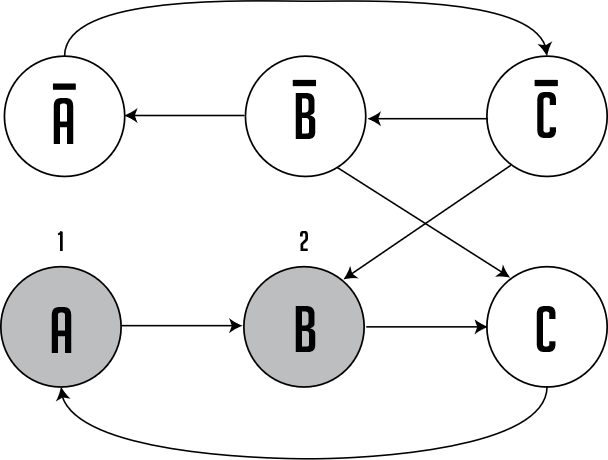
Given graph G,



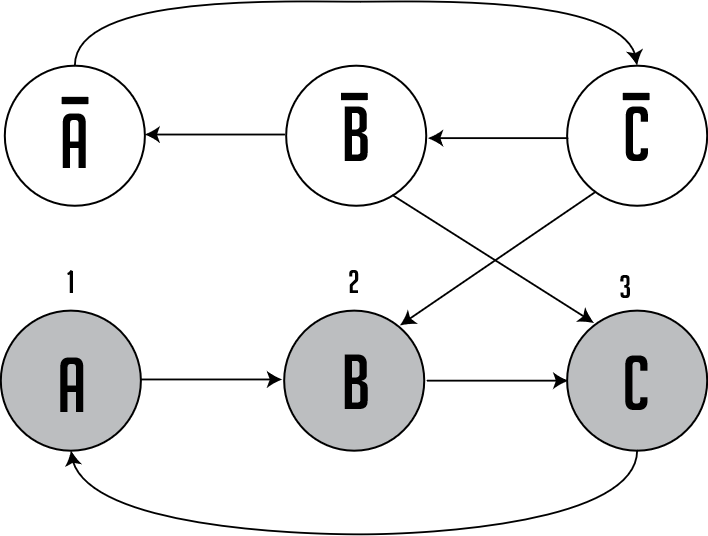
We declare “A” to be the source vertex and start the DFS from there. The “visited” vertices are shaded grey.



Stack: []

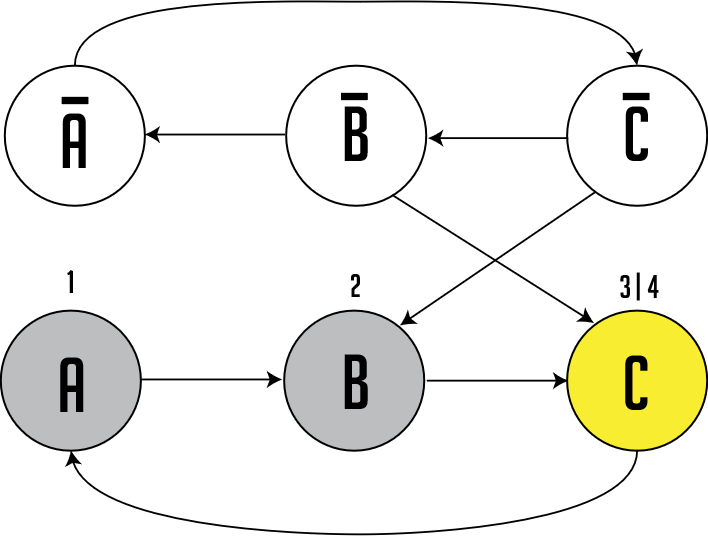


Stack: []

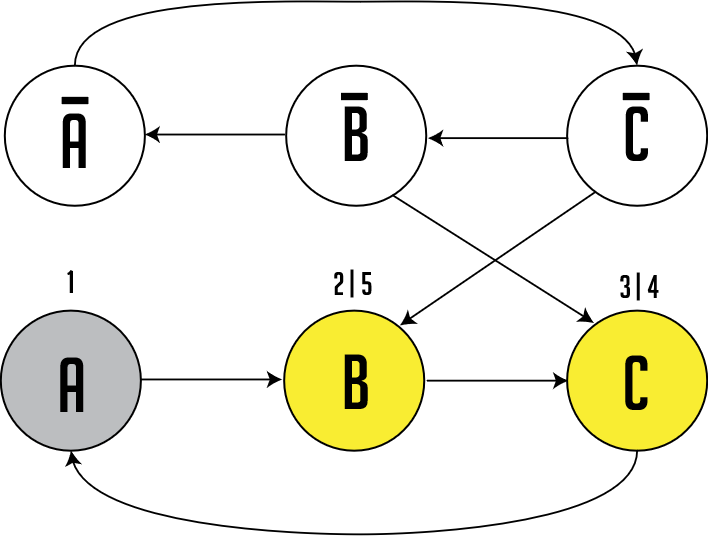


Stack: []

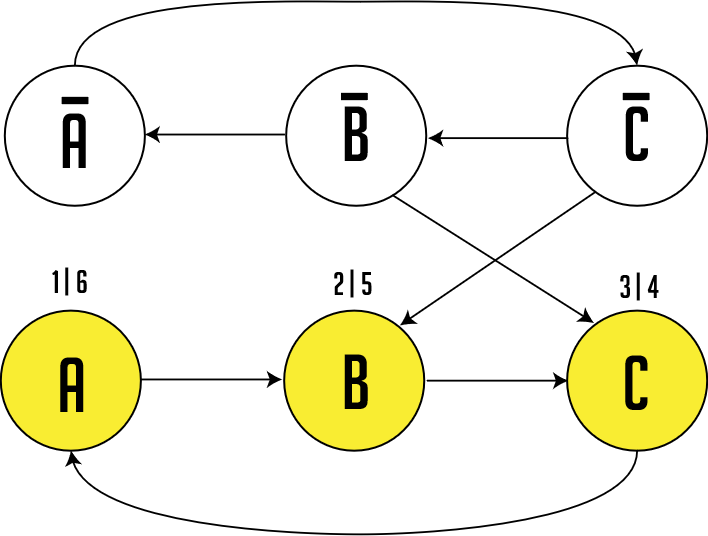
Backtracking:



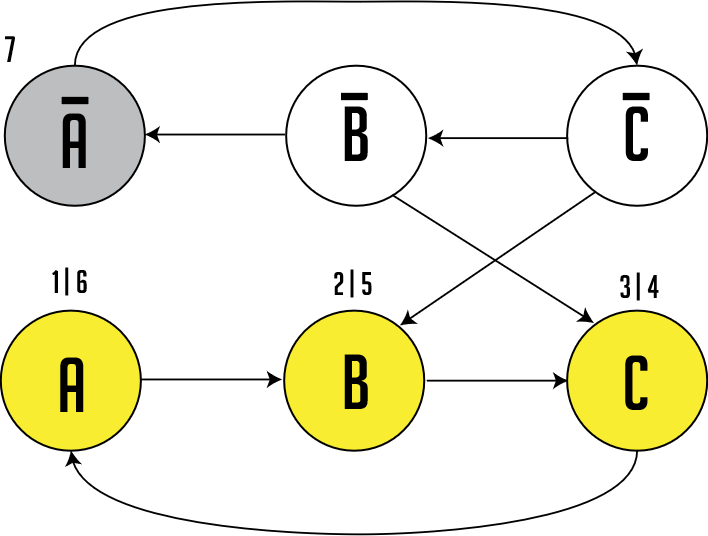
Stack: [C]



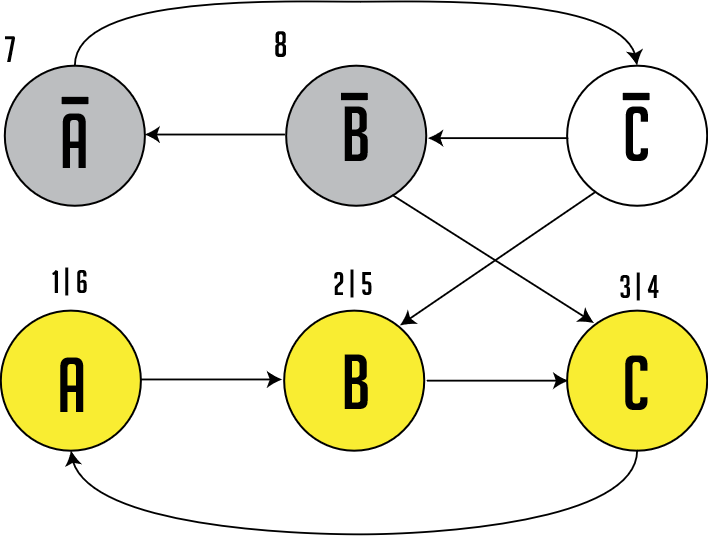
Stack: [C, B]



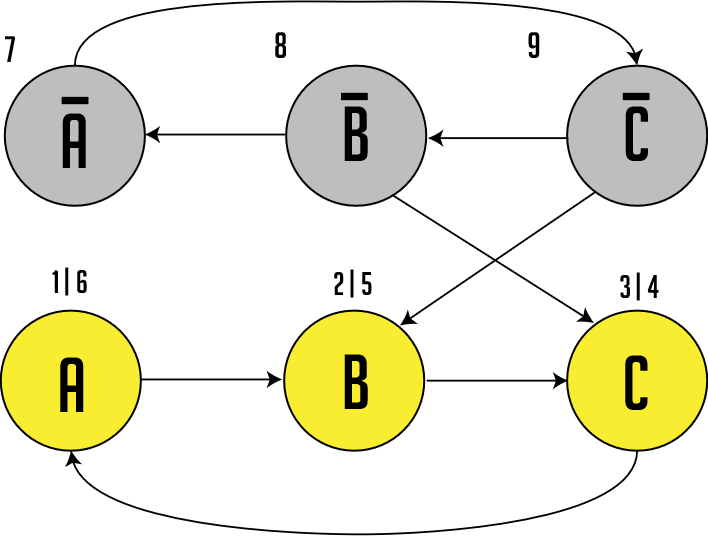
Stack: [C, B, A]



Stack: [C, B, A]

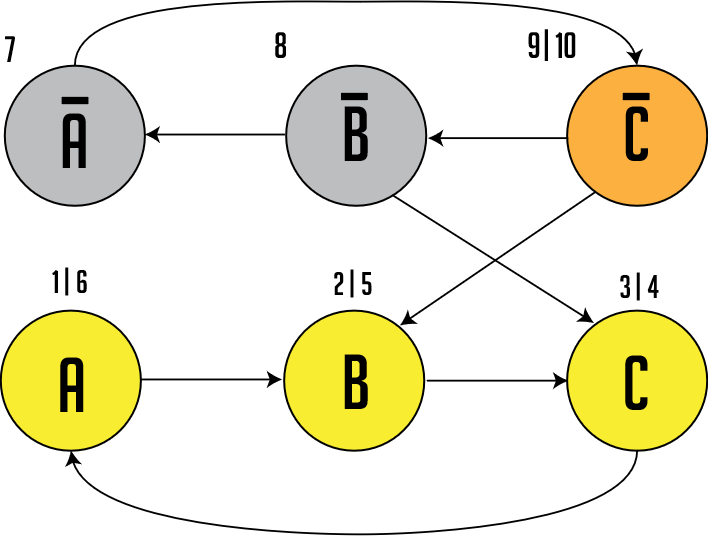


Stack: [C, B, A]

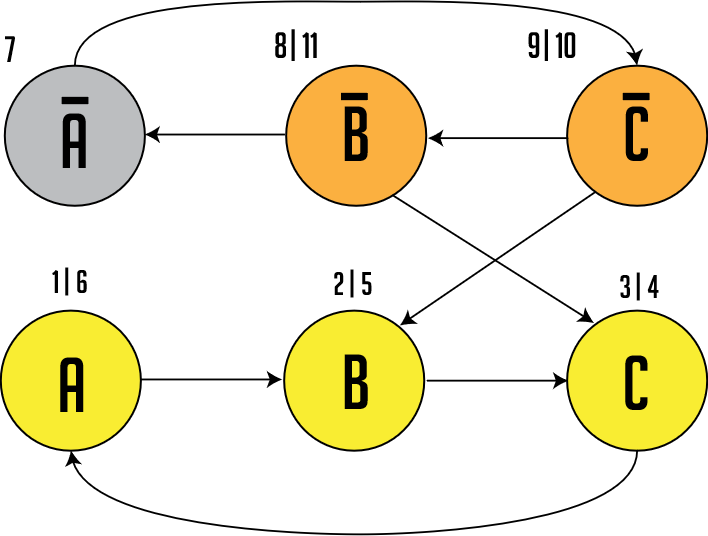


Stack: [C, B, A]

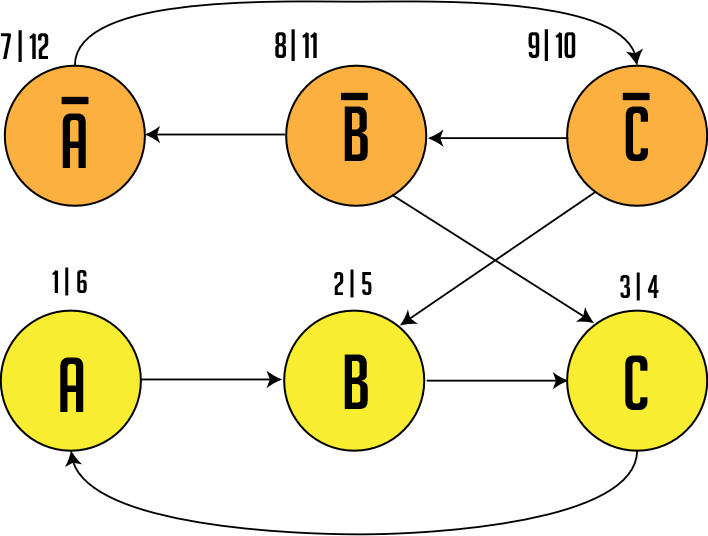
Backtracking:



Stack: [C, B, A, ¬C]

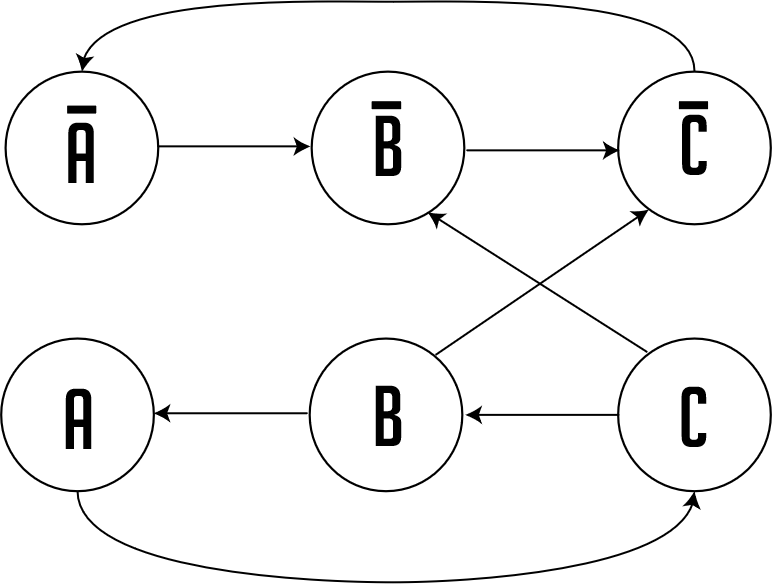


Stack: [C, B, A, ¬C, ¬B]

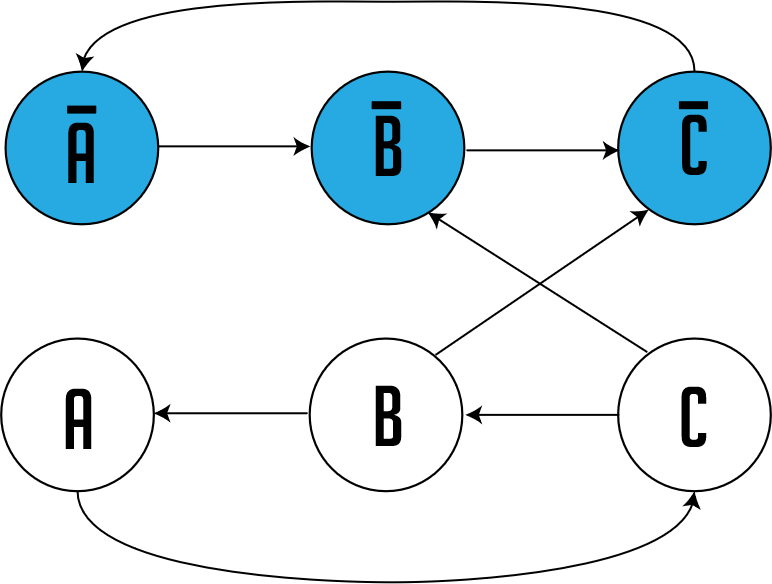


Stack: [C, B, A, ¬C, ¬B, ¬A]

Now we traverse the transposed graph (graph with all original **edges reversed**). We pop the top of the stack and use that value as the source vertex until the stack is empty.

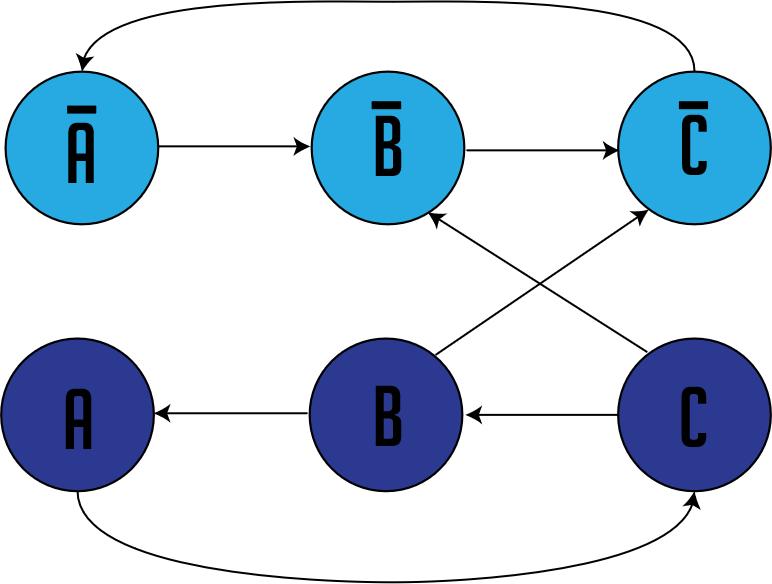


Starting with ,



Stack: [ C, ­B, A] (¬B ¬C are visited in this DFS, so we pop them)

SCC: [¬A ¬B ¬C]



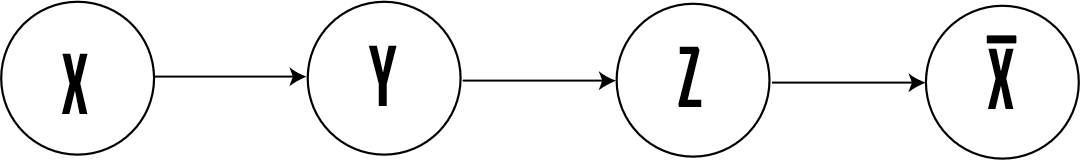
Stack: []

SCCs: [¬A ¬B ¬C], [A B C]

A 2 CNF Formula is unsatisfiable iff there exists a variable X, such that:

1. There is a path from X to ¬X or,
2. There is a path from ¬X to X.

(Proof by Contradiction) Assume that Suppose that the CNF is satisfiable. And there is a path from X to ¬X i.e.



From construction, the edge Y🡪Z indicates that there is a clause (¬Y Z). Recall that each edge means “implies” (i.e. Y 🡪 Z means Y implies Z, which means if Y is true, it implies Z must be true).

First, we let X be True. This means that all literals from the path X to Y must be True. Similarly, all literals from the path from Z to ¬X must be False. By definition, ¬X is false when X is True.

This results in an edge between Y and Z where Y is True and Z is false. Consequently, the clause (¬Y Z) becomes False, which contradicts with our initial assumption that the CNF is satisfiable.

**Therefore X and ¬X cannot be in the same SCC.**

**Performance Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CLAUSES | AVERAGE TIME (ns) | | | |
| 10 | 509038.6 | | | |
| 100 | 4480181.9 | | | |
| 1000 | 28849895.5 | | | |
| t1 (ns) | t2 (ns) | t3 (ns) | t4 (ns) | t5 (ns) |
| 881544 | 316298 | 1801523 | 230612 | 538800 |
| 2338912 | 3936270 | 5754366 | 9482943 | 1967958 |
| 16762026 | 38787574 | 5220855 | 82889446 | 15189000 |
| t6 (ns) | t7 (ns) | t8 (ns) | t9 (ns) | t10 (ns) |
| 175603 | 506359 | 138931 | 198171 | 302545 |
| 4603422 | 2063870 | 9233289 | 1924939 | 3495850 |
| 10184651 | 16613221 | 16069132 | 12137095 | 74645955 |

We tested the Kosaraju’s Algorithm on CNF files with 10, 100 and 1000 clauses, both satisfiable and unsatisfiable and plotted the average run time of 10 executions against the number of clauses.

Observe that there is a **linear relationship** between the number of clauses and the execution time.

**Why does the algorithm only work for 2-SAT and not 3-SAT?**

Recall that we can express every 2-SAT clause in the form of**.** Also, means that if we set X to be true, Y must also be true. If we set X to be false, Y must be false too. We can observe that this equation is very straightforward. However, with 3-SAT, we can observe case-multiplication. For examplecan be expressed as If we set A to be true, then either B or C must be true, we will not have a specific answer for which B or C is true in this step, hence the algorithm has to be further expanded and will not work in linear time anymore.

**Bonus: Random Walk Algorithm**

We can begin the Random Walk Algorithm by assigning a truth value to every literal in the CNF. Then, we find a clause that is not yet satisfied, choose a random literal in that clause and flip its value (from True to False/ False to True). Test the entire CNF again for satisfiability. If the CNF is not satisfiable, find another clause that is unsatisfied and flip another literal in it at random again.

**Bonus: Performance Analysis**

For satisfiable:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| CLAUSES | AVERAGE TIME (ns) | t1 | t2 | t3 | t4 | t5 |
| 10 | 0.591269 | 0.211218 | 3.385834 | 0.165378 | 0.182303 | 0.215449 |
| 100 | 0.128388 | 0.077223 | 0.07687 | 0.10649 | 0.07405 | 0.56313 |
| 1000 | 0.43037 | 0.251064 | 0.873081 | 0.295493 | 0.20875 | 0.283152 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| t6 | t7 | t8 | t9 | t10 |
| 0.194645 | 0.765532 | 0.139989 | 0.503891 | 0.148452 |
| 0.085686 | 0.064529 | 0.068761 | 0.084276 | 0.082865 |
| 1.401302 | 0.277157 | 0.260584 | 0.235901 | 0.217213 |

For unsatisfiable:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| CLAUSES | AVERAGE TIME (ns) | t1 | t2 | t3 | t4 | t5 |
| 10 | 2.961952 | 1.665059 | 2.634053 | 4.215895 | 2.94083 | 2.933778 |
| 100 | 151.85 | 116.2347 | 126.6052 | 262.0682 | 161.1353 | 100.5045 |
| 1000 | 28526.44 | 28949.84 | 29166.56 | 32761.29 | 25190.46 | 27956.92 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| t6 | t7 | t8 | t9 | t10 |
| 5.514585 | 1.257787 | 4.422176 | 2.717271 | 1.318084 |
| 107.0737 | 150.6713 | 132.4132 | 158.2636 | 203.5308 |
| 39616.84 | 28058.97 | 24684.12 | 24192.24 | 24687.17 |

From the results we can see that the random walk algorithm is good when there is a solution, but bad if there is there is no solution. We can limit the number of steps before declaring that the CNF is unsatisfiable, but doing so may eliminate the possibility of finding a solution when it exists.