2D Challenge Report

50.004 Introduction to Algorithms Sept-Dec 2019 Term

Team 3-9: Yeh Swee Khim, Lin Cheng, Jia Yunze, Xie Han Keong, Jisa Mariam Zachariah

---

Table of Contents

Github link: https://github.com/han-keong/solve-2sat/

Introduction

In this 2D project, our group was tasked to come up with a 2-SAT solver for the 50.004 component. A SAT solver is a program which can solve boolean satisfiability (SAT) problems. The objective of a SAT problem is to determine if a given boolean expression can be evaluated to true. Specifically, a boolean expression refers to a formula which consists of *clauses* that contain the *literals* of a set of *variables*. To illustrate, the following is an example of a boolean expression.

$$F(x, y) = (x \vee y) \wedge (\neg y)$$

This formula consists of the clauses $(x \vee y)$ and $(\neg y)$, which contain the literals x, y, and $\neg y$. Another way to write it is:

$$F(x, y) = (x\ OR\ y)\ \ AND\ (NOT\ y)$$

We say that y is the *positive literal* of the variable y, while $\neg y$ is the *negative* literal or the *negation* of the positive literal.

Having said that, the role of a SAT solver is to find at least one combination of assignments that will *satisfy* the boolean expression. So, it should find that the formula above has the following solution.

$$x = 1, y = 0$$

A 2-SAT problem is simply a special case of the SAT problem which has at most 2 literals in each clause. There may be many variables in a formula, but each clause (besides empty clauses) can only have either one or two literals. The example above is a 2-SAT problem.

The SAT problem is commonly known to be the very first problem that was proved to be NP-complete by Cook and Levin in 1971. This means that all SAT solvers will have exponential worst-case running time $O(2^n)$. Even state-of-the-art SAT solvers, mainly based on an algorithm known as the Davis–Putnam–Logemann–Loveland (DPLL) algorithm, incorporate many modern strategies like clause learning, non-chronological backtracking, branching heuristics, and so on. However, their worst-case running time is still exponential as there will always be a case when all possible 2^n assignments are tested.

Nonetheless, the simplified 2-SAT case is much easier to solve. In fact, there exists algorithms which are able to solve it in linear time. Several algorithms like Kosaraju's algorithm and Tarjan's algorithm have been developed which make use of a depth-first search, which has a running time of a much smaller scale of $O(V + E)$. These algorithms are based on the principle of finding strongly connected components in the implication graph that can be derived from 2-SAT boolean expressions. They exploit the fact that the number of implications scale linearly with the number of clauses and variables when each clause has only two literals. In our implementation, we used Tarjan's algorithm because it is an elegant solution that only uses one pass of a depth-first search, but first, we shall discuss about how to obtain the implication graph from a boolean expression in order to solve a 2-SAT problem using its strongly-connected components.

Obtaining the Implication Graph of a Boolean Formula

A 2-SAT formula consists of a conjunction (an AND operation) of multiple clauses, where each clause is a disjunction (an OR operation) of at most two literals. To construct the implication graph, we will have to consider the relationship between each clause and its logically equivalent implications.

P	$\neg P$	Q	$\neg Q$	$P \vee Q$	$\neg P \Rightarrow Q$	$\neg Q \Rightarrow P$
F	T	F	T	F	F	F
F	T	T	F	T	T	T
T	F	F	T	T	T	T
T	F	T	F	T	T	T

Figure 1: Truth table of implications and the disjunction between P and Q

From the truth table in Figure 1, we can identify from rows 1 to 4 that for all possibilities, the disjunction $(P \vee Q)$ is logically equivalent to the implication $\neg P \Rightarrow Q$. Likewise, it is logically equivalent to the opposite implication $\neg Q \Rightarrow P$. Following this, we can also derive that the disjunction $(\neg P \vee Q)$ is logically equivalent to the two implications $P \Rightarrow Q$ and $\neg Q \Rightarrow \neg P$. This allows us to convert any clause that we see in a boolean expression into its corresponding implications.

In order to illustrate this, we consider the following formula, F, in Figure 2 below. In order to solve this boolean expression using the strongly connected components of its implication graph, we convert each clause in F into its corresponding implications.

$F(x, y, z) = (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y) \wedge (x \vee \neg z)$	
Clauses	**Implications**
$(x \vee \neg y)$	$\neg x \Rightarrow \neg y$ $y \Rightarrow x$
$(\neg x \vee y)$	$x \Rightarrow y$ $\neg y \Rightarrow \neg x$
$(\neg x \vee \neg y)$	$x \Rightarrow \neg y$ $y \Rightarrow \neg x$
$(x \vee \neg z)$	$\neg x \Rightarrow \neg z$ $z \Rightarrow x$

Figure 2: The set of implications derived from F

Now that we have found the set of implications corresponding to the boolean expression for F, we can represent each implication using an edge in a graph. For example, we can represent $\neg x \Rightarrow \neg z$ with a directed edge from node $\neg x$ to node $\neg z$ as illustrated by the red arrow in the implication graph of F as shown in Figure 3.

Since there are two implications for every clause in a 2-SAT formula, this implies that a 2-SAT formula with n clauses will have $2n$ edges. Furthermore, in the implication graph (as can be seen in Figure 3), each variable is associated with two different nodes–one for the positive literal such as x, and one for the negative literal $\neg x$. This implies that a 2-SAT formula with n variables will have $2n$ nodes.

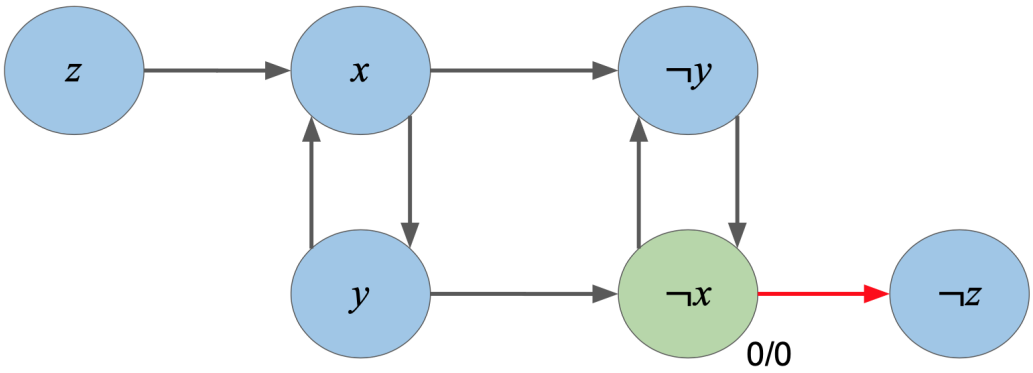


Figure 3: Implication graph derived from the set of implications of F

In our implementation, we use an adjacency list to represent the implication graph for the 2-SAT formula. An adjacency list stores the successors of each node in a doubly linked list and has a space complexity of $O(V + E)$. However, instead of a doubly linked list, we used a mapping (i.e. a Python `dict`) to store the nodes and their successors as key-value pairs. This allowed us to circumvent the problem of node indexing caused by the nodes having positive and negative values.

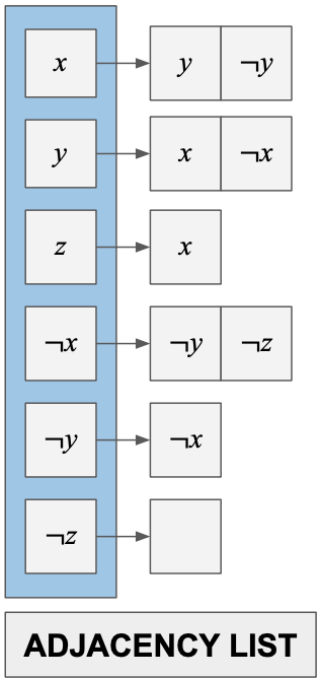


Figure 4: Adjacency list representing the implication graph of F

Strongly Connected Components of a Directed Graph

After obtaining the implication graph of a 2-SAT formula, we are able to analyze the graph and search for its strongly connected components. A strongly connected component (SCC) is a subgraph of a directed graph in which there is a cyclic path between all pairs of nodes. That is to say, it is possible to reach any other node from any particular node in the SCC. For example, consider the implication graph of F shown in Figure 5. Nodes x and y form an SCC because x can be reached from y while y can be reached from x. A single node can also be considered an SCC on its own–take for instance the nodes z and $\neg z$ in Figure 5.

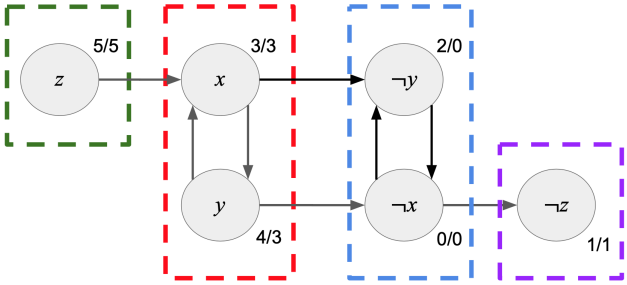


Figure 5: The implication graph of F subdivided according to its SCCs

Due to the fact that every edge in an implication graph represents an implication from one literal to another, when a group of nodes are connected together in an SCC, it implies that each one of them has the same truth value. For instance, in Figure 5, if x is false, then the edge connecting x to y (i.e. the implication $x \Rightarrow y$) implies that y must be false as well. Similarly, if x is true, then y must be true too. This property of SCCs can actually be exploited so that finding unsatisfiable 2-SAT formulas becomes trivial. Let us see why.

Consider the case when an SCC contains a literal and its negation, say for instance when both $\neg x$ and x are present in an SCC. Setting one to true would mean that the other also has to be true because of what we have discussed previously, but that would be impossible as both of them have to be the opposite of each other. As such, this constitutes a contradiction and therefore the 2-SAT formula must be unsatisfiable. Thus, we can see that it becomes very simple to check for unsatisfiable cases. In fact, it only has a time complexity of $O(E)$ as the number of SCCs that we need to check is bounded above by the number of implications which is two times the number of clauses, and therefore it has to be linear with respect to the number of clauses.

However, two questions remain: how do we find strongly connected components in the first place, and what should be done for the satisfiable case? We shall discuss the latter in a later section in this report and shift our attention to the former question for now. In order to obtain the strongly-connected components of our implication graph, we shall use an algorithm published by Robert Tarjan in 1972, which describes a way to use one pass of a depth-first search and a method to keep track of the lowest index of each node's successors in order to find the strongly connected components of a graph in $O(V + E)$ time. This will be discussed in greater detail in the next section.

Tarjan's Algorithm for Strongly Connected Components

It has been said that Tarjan's algorithm is elegant in the sense that it only requires one depth-first search to find all the strongly connected components in the graph. Using our formula $\boldsymbol{F}$ and its implication graph, we shall step through how Tarjan's algorithm determines that the graph contains four SCCs. We will use a blue node to represent an unvisited node, a green node to represent the current node that is being visited, an orange node to represent a node that has been visited and is on the stack, and a grey node to represent a node that has been visited and has also been popped off the stack (to be added into an SCC). A red arrow (→) symbolizes that a node is going to visit another node, and red numbers (e.g. 0/1) symbolize that they are currently being compared. Lastly, nodes that are enclosed with a dashed box indicate that the algorithm has identified them as belonging to an SCC. This color scheme will be very helpful for providing a more intuitive understanding of what is happening at a particular step of the algorithm.

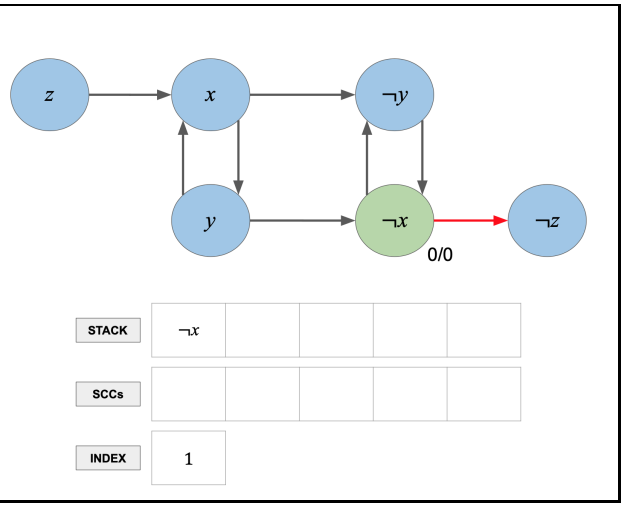
Now, before we begin, we instantiate an index counter to zero and create two mappings (e.g. with Python `dict`) to store the index and something we call the *lowlinks* of each node. The lowlink of a node is the lowest index of all its successors. It will be used later in the algorithm to determine if we have found a root node of an SCC. Note that the index and the lowlink of each node will be indicated respectively by two numbers separated by a slash (e.g. 0/1). Finally, we create a stack just like the classic implementation of a depth-first search, as well as another list to store the SCCs that we have found.

We start by visiting a node (any node) by invoking the recursive depth-first search method, which we shall call `DFS_visit`. Every time we visit an unvisited node, we assign the current index to its index and increment the index counter. We also initiate the node's lowlink value to its index and proceed to push the node onto the stack. Once that is done, we will visit each of the node's successors in the same way as a depth-first search.

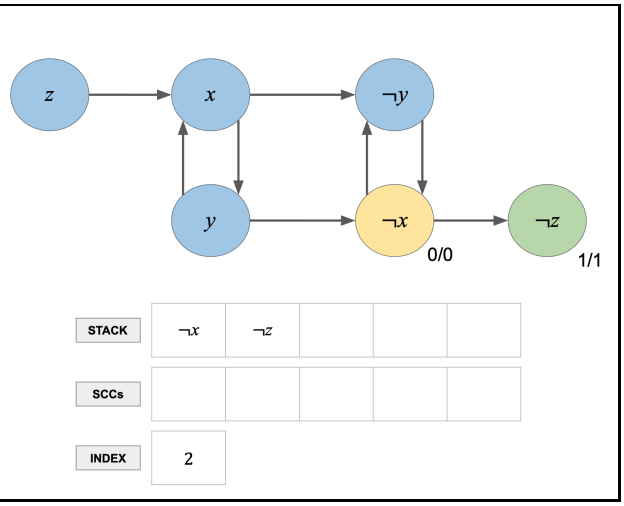
When visiting a successor, there are three possible cases that may happen. In the following few snippets, these three cases will be illustrated in a stepwise fashion using the implication graph in Figure 3.

Visiting a Tree Edge

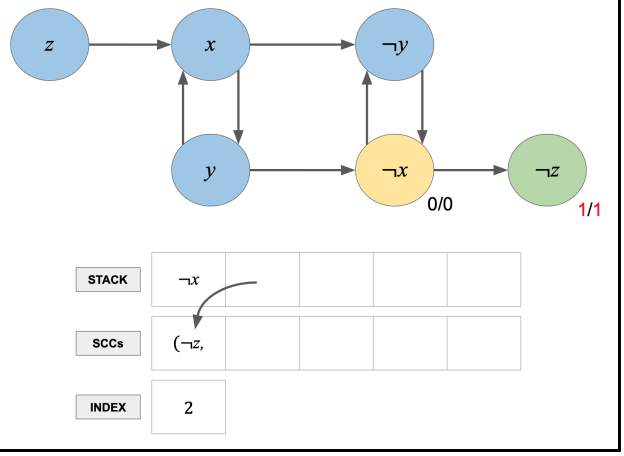
Here we have $\neg x$ going to visit the leaf node $\neg z$. This is a tree edge because the successor $\neg z$ is an unvisited node. Tree edges are the most common case as much as they are the most important because tree edges are responsible for advancing the depth-first search to the next level of recursion. This is because the unvisited successor will also visit its own successors recursively, until a leaf node (like $\neg z$) or a node with all back or cross edges is visited.



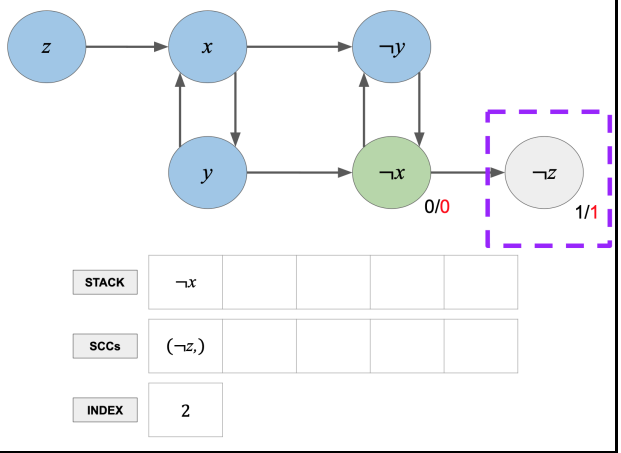
We step forward after invoking the recursive method `DFS_visit(¬z)`. In its scope, the current index was assigned to $\neg z$'s index and the index counter was incremented. The lowlink of $\neg z$ was initialized to its index, and $\neg z$ was pushed onto the stack, as per Tarjan's algorithm and classic depth-first search. Next, `DFS_visit` will try to visit the successors of $\neg z$, but there are none as $\neg z$ is a leaf node. Thus, $\neg z$ can be said to have completed its depth-first search.



Now that the successors of $\neg z$ have all been visited (there were none), Tarjan's algorithm states that if the index and the lowlink of the current node $\neg z$ are equal, then the current node is a root to an SCC, and thus nodes should be popped off the stack and stored in a new SCC. Here we observe that $\neg z$ has an index and lowlink of 1. Thus, a new SCC is created and nodes are popped off the stack and stored in it, which has just happened to $\neg z$ here.



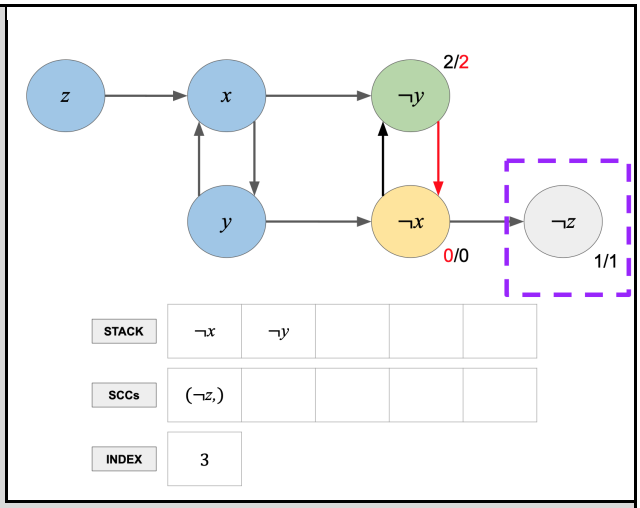
According to Tarjan's algorithm, the new SCC is complete when the current node is added to it, which means that the new SCC was finished after $\neg z$ was added. Thus, `DFS_visit(¬z)` returned execution back to the scope of `DFS_visit(¬x)`. At this point, Tarjan's algorithm states that the lowlink of the successor just visited and the current node should be compared, and the current node's lowlink should be set to the smaller of the two.



The reason for updating the node's lowlink is to check if a successor has found a lower index than the current node's lowlink through a back edge to an ancestor. If true, it implies that the node and its ancestor belong in the same SCC. In this case, the current node $\neg x$ has the smaller lowlink, so no changes will be made, and $\neg x$ will simply visit its next successor, $\neg y$.

Visiting a Back Edge

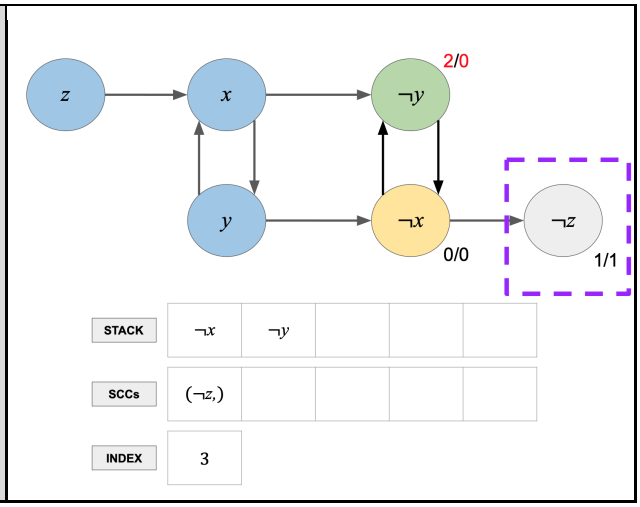
Here we have $\neg y$ going to visit $\neg x$, which has been visited but is still inside the stack. This is a back edge because it brings the depth-first search back to a visited node whose visit to its successors has not yet returned–that is really the node's ancestor. The back edge is the quintessential source of SCCs because when a node points to its own ancestor, it creates a directed cycle where it can reach itself again by following the path from its ancestor that led to it.



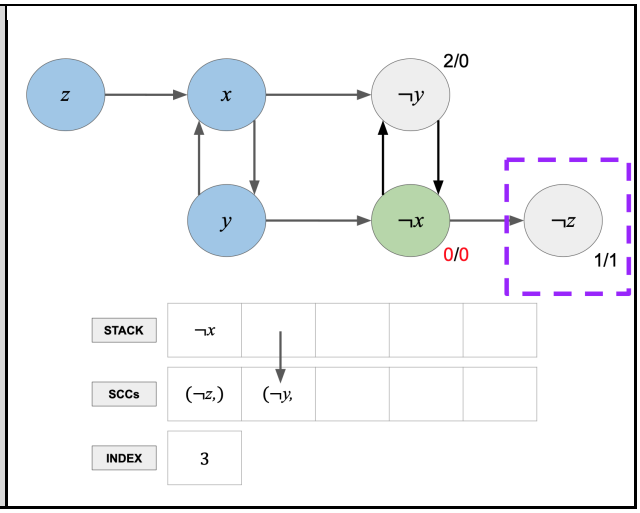
In this case, according to Tarjan's algorithm, the index of the successor (which is really the node's ancestor) and the lowlink of the current node should be compared, and the current node's lowlink should be set to the smaller of the two. We can see that the lowlink of $\neg y$ is larger than the index of $\neg x$. Thus, the lowlink of $\neg y$ should be set to the index of $\neg x$.

This snippet exemplifies the purpose of lowlinks–the lowest index of a node's successors can only be lower than the node's default lowlink (i.e. the node's index) when one of them is actually an ancestor of the node itself. If no such successor exists, then the node's lowlink will never change, because the lowlink of its successors will always be larger than the node's. This is because all successors must have been visited after the node, so their indices (and lowlinks, since no back edges are present) must all be greater. Conversely, when the lowlink of a node's successor is smaller than its own, it implies that a path to an ancestor was found, meaning that both the node and its successor are in the same SCC.

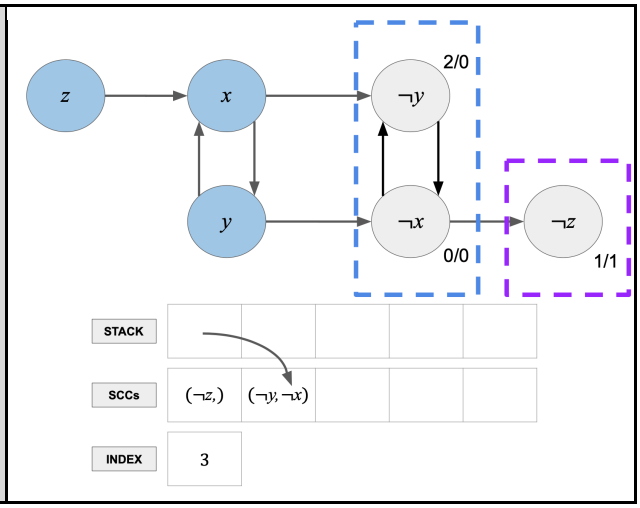
Having visited the back edge to $\neg x$, node $\neg y$ has no more successors to visit. By Tarjan's Algorithm, if the index and the lowlink of the current node $\neg y$ are equal, then the current node is a root to an SCC, and thus nodes should be popped off the stack and stored in a new SCC. However we can see that is not the case since $\neg y$'s lowlink was set to a lower value, and so `DFS_visit(`$\neg y$`)` will terminate and return to the node which visited it, node $\neg x$.



After `DFS_visit(¬y)` returned to `DFS_visit(¬x)`, the lowlinks of $\neg y$ and $\neg x$were compared to see if $\neg y$ found a smaller lowlink. We can see that both lowlinks are 0, thus the lowlink of $\neg x$ did not change. Now, node $\neg x$ has no more successors to visit, thus it has completed its depth-first search. Next, its index and lowlink are compared to see if it is a root to an SCC. Since they equate, $\neg x$ is indeed a root node, so nodes are popped off the stack and stored in a new SCC, beginning here with $\neg y$.

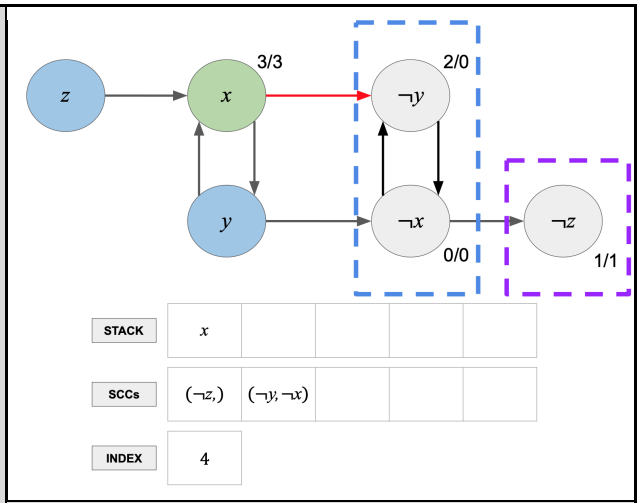


As stated before, the SCC is complete when the node itself is popped off the stack and added to the SCC. Here, $\neg x$ itself is popped off the stack and added to the new SCC, completing it, leading to the termination of `DFS_visit(¬x)`. In this particular scenario, the depth-first search has halted after searching through half the graph, creating one tree. This means that another depth-first search has to be initiated for the other half, which in turn will create a forest.



Visiting a Cross Edge

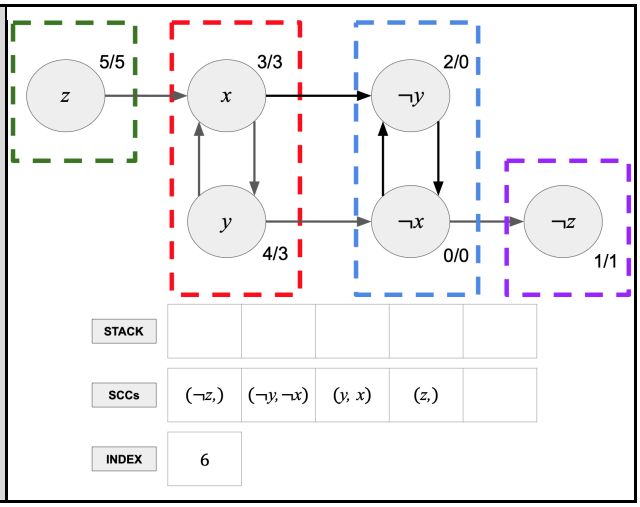
Here we have node x going to visit $\neg y$, which has not only been visited but has also been popped off the stack and added to an SCC of its own. This is a cross edge because it represents a path that crosses over into another tree whose depth-first search has already been resolved. Cross edges will interfere with a node's lowlink, since the successor has already been visited in an early tree, which implies that it will have a smaller index.



If cross edges are considered, this may potentially lead to false positives in the new tree, since setting the node's lowlink to a lower value normally implies that an ancestor was found. This will affect cases where the node's ancestors have no back edges. This is why in Tarjan's Algorithm, we ignore cross edges completely. Therefore, here node x will not visit $\neg y$ and will not update its lowlink–it will simply bypass $\neg y$ and visit its next successor, y.

Expected Result

We can see that Tarjan's Algorithm follows a classic depth-first search very closely. With the three cases of visiting successors described above, we can step through the entire graph in order to find out what the expected result of the algorithm is–the following four SCCs in this particular order. This is in fact a reverse topological order because it was constructed by popping nodes off the stack, which naturally builds up nodes in the correct topological order.



Pseudocode

Given a graph G represented by an adjacency list $G = (V, E)$

1. Initialize a stack H, a list L to store SCCs, and an index counter k
2. For each node v in V and its respective adjacency list e in E do the following
    3. If v not visited then do `DFS_visit(`v`)`, where `DFS_visit(`v`)` is the recursive function described by the following
        4. Assign current index k to v's index, and increment k
        5. Initialize lowlink of v to v's index
        6. Push v onto stack H
        7. For each successor w in e, do the following checks
            8a. If w not visited, do `DFS_visit(`w`)`
            8b. After `DFS_visit(`w`)` returns, if w has smaller lowlink, set lowlink of v to w's lowlink
            9. Otherwise (w visited), if w not in stack H, advance to next w
            10. Otherwise (w visited and in H), if w has smaller index, set lowlink of v to w's index
        11. After all w in e checked, if v's index equals its lowlink, do the following
            12. Create a new list C
            13a. Pop a node off stack H and add it into C
            13b. If node v not popped off, repeat 13a
            13c. Otherwise, convert C into `tuple(`C`)` and add it into L
14. After all v in V visited, return L

Solving the 2-SAT Problem

We have discussed how to obtain the implication graph of a 2-SAT formula (pg. 3-4) as well as the advantages (pg. 5) and a method (pg. 6-10) of finding the SCCs in the implication graph. The last step to solving a 2-SAT problem is to analyze the SCCs to find out if the formula is satisfiable or not.

As discussed earlier, we can exploit a property of SCCs so that finding the unsatisfiable case becomes trivial. This can be done by checking if any SCC contains a literal and its negation, as this constitutes a contradiction and therefore the 2-SAT formula must be unsatisfiable. If no such SCC can be found, then it is sufficient to say that the formula is satisfiable. However, how can we find a combination of assignments that will satisfy the boolean expression? Consider the result of applying Tarjan's Algorithm to our formula F shown in Figure 6.

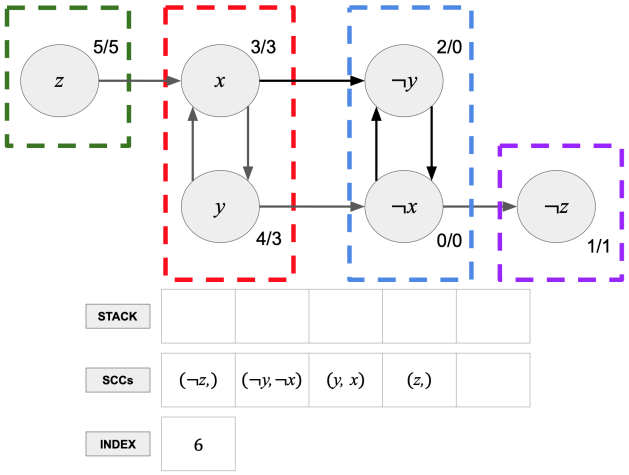


Figure 6: The result of Tarjan's Algorithm on the implication graph of F

Notice that each SCC can be considered as a single node in a graph of SCCs, and this graph is directed and acyclic. Just like the original implication graph of F, the edges of this graph of SCCs are made up of implications. Now, let us consider the truth table of an implication in Figure 7.

P	Q	$P \Rightarrow Q$
F	F	T
F	T	T
T	F	F
T	T	T

Figure 7: Truth table of an implication between P and Q

Notice here that if we were to set P to false, $P \Rightarrow Q$ still holds true. Likewise, if we were to set Q to true, $P \Rightarrow Q$ still holds true as well. This implies that we are able to assign false to nodes in topological order of the graph of SCCs (from P to Q), or true to nodes in reverse topological order (from Q to P)–provided that the nodes have not been assigned with truth values–and the edges of the graph will still hold true. We now have two ways to find a set of assignments to satisfy the formula.

We observe that conveniently, the SCCs that are returned from Tarjan's Algorithm are in reverse topological order (See Figure 6). Therefore, we shall carry out the reverse topological

order of assignments. That is to say we will assign true to each SCC in reverse topological order if a truth value has not been assigned to it. Now, we shall demonstrate how this can be done. From Figure 6, we have the following implications.

$$z \Rightarrow (y \Leftrightarrow x) \Rightarrow (\neg y \Leftrightarrow \neg x) \Rightarrow \neg z$$

We start by assigning $\neg z = 1$. This implies $z = 0$, so the leftmost SCC is constrained to be false. Moving on, we assign the next SCC in reverse topological order to true, that is to say $(\neg y \Leftrightarrow \neg x) = 1$. This implies that $x = 0$ and $y = 0$, which constrains the second SCC from the left to be false. In this manner, we have obtained a suitable combination of assignments $x = 0, y = 0, z = 0$ that satisfy F and the 2-SAT formula is solved.

Implementation of 2-SAT Solver

The implementation for our 2-SAT Solver in Python can be found below. We decided to implement it in Python because it is very convenient and user-friendly, and also allows for rapid and flexible development. `My2SATSolver` is a class that encapsulates several methods required for deriving the solution to a 2-SAT problem. The main method to call is `My2SATSolver.solve`, which takes in a cnf string or file object and returns the output "FORMULA UNSATISFIABLE" or "FORMULA SATISFIABLE" followed by a list of bindings in increasing order.

`My2SATSolver.solve`

`My2SATSolver.solve` will parse the cnf string or file object, generate the implication graph, obtain a list of strongly connected components, and carry out the truth assignments of literals in the order of that list, which is already in reverse topological order as a side effect of Tarjan's algorithm. The assignments are then sorted in increasing order and returned.

```
class My2SATSolver:
    """
    2-SAT solver based on strongly-connected components of implication graphs.

    Based on Tarjan's algorithm to find strongly-connected components.
    """

    @staticmethod
    def solve(cnf):
        """
        Solve a 2-SAT problem.

        Uses strongly-connected components to find satisfiability.

        Input: str (cnf text) OR file object (cnf file)
        """
        result = My2SATSolver.parse_cnf(cnf)
        if not result:  # failed to parse, return None
            return None
        _, clauses = result

        start_time = time.time()  # start timing here
        graph = My2SATSolver.create_graph(clauses)
        sccs = My2SATSolver.tarjan_scc(graph)

        assignments = {}
        for scc in sccs:
            for node in scc:
                if -node in scc:  # literal and its negation are present
                    print("FORMULA UNSATISFIABLE")
                    end_time = time.time()
                    time_taken = end_time-start_time
                    print("time taken: " + format_time(time_taken))
                    return None
                if node not in assignments:  # assign the literal to true
                    assignments[node] = 1
                    assignments[-node] = 0  # and its negation to false

        end_time = time.time()  # stop timing here
        time_taken = end_time-start_time

        # sort assignments by increasing order, getting rid of negative literals
        assignments = {k:assignments[k] for k in sorted(assignments) if k > 0}

        print("FORMULA SATISFIABLE")  # print outputs
        print_long(" ".join(map(str, assignments.values())))
        print("time taken: " + format_time(time_taken))

        return assignments
```

Figure 8: Code for `My2SATSolver.solve`

My2SATSolver.parse_cnf

We wrote a method called `My2SATSolver.parse_cnf` which takes in a cnf file and converts it into a set of clauses. It has very robust error checking, and it will also fail silently and inform the user when the incorrect number of variables and/or clauses was given in the cnf. The full code can be found in the Github link at the top of the report.

```
@staticmethod
def parse_cnf(cnf):
    """
    Takes in cnf file and outputs its set of clauses.

    Input:  str (cnf text) OR file object (cnf file)
    Output: list of 1-tuples or 2-tuples [(a,), (b,c), ...]
    """
    num_var = 0
    num_clause = 0
    var = []
    fmt = None
    clauses = []

    if isinstance(cnf, io.TextIOBase):
        lines = cnf

    elif isinstance(cnf, str):
        lines = cnf.splitlines()

    for i, line in enumerate(lines):
        if fmt != "cnf":
            if line.startswith("c"):  # preamble
                continue

            if not line.startswith("p"):
                print("invalid problem statement")
                return None
```

Figure 9: Code for `My2SATSolver.parse_clauses` (not the full code)

My2SATSolver.create_graph

We also wrote a method called `My2SATSolver.create_graph` which takes in the set of clauses generated from `My2SATSolver.parse_cnf` and constructs the corresponding implication graph according to the implication table in Figure 2. The implication graph is expressed in an adjacency list, and is stored in a Python `dict`.

```
@staticmethod
def create_graph(clauses):
    """
    Takes in a set of clauses and outputs the adjacency list
    of the implication graph which can be formed with the clauses.

    Input: list of 1-tuples or 2-tuples [(a,), (b,c), ...]
    Output: dict of int mapped to list of int {a:[b], c:[d,e], ...}
    """
    graph = {}

    for clause in clauses:
        if len(clause) == 2:
            a, b = clause
        else:  # length 1
            a = b = clause[0]

        if -a not in graph:  # initialize adjacency list of node
            graph[-a] = []
        if b not in graph[-a]:  # prevent duplicate successors
            graph[-a].append(b)

        if -b not in graph:
            graph[-b] = []
        if a not in graph[-b]:
            graph[-b].append(a)

    return graph
```

Figure 10: Code for `My2SATSolver.create_graph`

My2SATSolver.tarjan_scc

We implement Tarjan's algorithm in `My2SATSolver.tarjan_scc` to find all the strongly connected components in the implication graph that we generate. It contains a nested helper method `DFS_visit` which will be responsible for carrying out the recursive depth-first search on each node in the graph while finding the lowest index of their successors, and ultimately popping nodes off the stack when a root node to a strongly connected component is found. These nodes will be stored in a tuple which will be added to the list that will be returned, `sccs`.

```
@staticmethod
def tarjan_scc(graph):
    """
    Tarjan's Algorithm (named for its discoverer, Robert Tarjan) is a
    graph theory algorithm for finding the strongly connected components
    of a graph.

    Based on: http://www.logarithmic.net/pfh/blog/01208083168
    """
    # initialize variables
    index_ctr = [0]
    lowlinks = {}
    index = {}
    stack = []
    sccs = []

    def DFS_visit(node):
        index[node] = index_ctr[0]
        lowlinks[node] = index_ctr[0] # initialize lowlink to index
        index_ctr[0] += 1
        stack.append(node)

        try:
            successors = graph[node]
        except:
            successors = []

        for successor in successors:
            # tree edge: visit and update lowlinks
            if successor not in lowlinks:
                DFS_visit(successor)
                lowlinks[node] = min(lowlinks[node],lowlinks[successor])

             # back edge: don't visit, but compare its index with lowlink
            elif successor in stack:
                lowlinks[node] = min(lowlinks[node],index[successor])

            # cross edge: ignore. move on to next successor
            else:
                continue

        # root node to SCC found. pop SCC off stack.
        if lowlinks[node] == index[node]:
            scc = []

            while True:
                successor = stack.pop()
                scc.append(successor)
                if successor == node: break

            sccs.append(tuple(scc))

    # visit all unvisited nodes
    for node in graph:
        if node not in lowlinks:
            DFS_visit(node)

    return sccs
```

Figure 11: Code for `My2SATSolver.tarjan_scc`

Complexity Analysis

We exclude the time it takes to parse the cnf because that is not part of the problem. There are three phases in `My2SATSolver.solve` that constitute the time that recorded. The first is a call to `My2SATSolver.create_graph` to create the implication graph out of the clauses that have been generated from the cnf. The next is a call to `My2SATSolver.tarjan_scc` to find the SCCs of the implication graph. The last section is inside the scope of `My2SATSolver.solve` where it carries out the truth value assignments in order of the SCCs that have been returned from `My2SATSolver.tarjan_scc`.

In phase 1, `My2SATSolver.create_graph` does a simple `for` loop over the clauses which is in $O(E)$ time. Everything within the loop is in $O(1)$, thus the overall time complexity for phase 1 is $O(E) * O(1) = O(E)$. Since the graph returned is in the form of an adjacency list, we know that it will have a space complexity of $O(V + E)$.

Next, in phase 2, `My2SATSolver.tarjan_scc` carries out a depth-first search on the graph. We know that this will have $O(V + E)$ time complexity. In terms of space complexity, `My2SATSolver.tarjan_scc` uses two dictionaries to store the index and lowlinks of each node, which is $2 * O(V)$. The index counter is simply $O(1)$, while the maximum size of the stack is the total number of nodes in the graph which is $O(V)$. Lastly, it returns a list of SCCs, which is also $O(V)$ since every node will appear in exactly one SCC. Thus, the total space complexity is $2 * O(V) + O(1) + O(V) + O(V) = O(V)$.

Lastly, in phase 3, `My2SATSolver.solve` uses two `for` loops to check for each node in each SCC. This is essentially checking through each node once which is $O(V)$, since every node will appear in exactly one SCC. The assignments inside the loop are simply $O(1)$. Thus, the overall time complexity is $O(V) * O(1) = O(V)$. Note that we have excluded the time taken to sort the assignments as it happens after the 2-SAT problem is solved. In terms of space complexity, each node is being assigned a truth value, so this means that the space complexity will be $O(V)$.

Therefore, the overall time complexity is $O(E) + O(V + E) + O(V) = O(V + E)$, while the overall space complexity is $O(V + E) + O(V) + O(V) = O(V + E)$, as shown in Figure 12.

Phase	Method	Time Complexity	Space Complexity
1	`My2SATSolver.create_graph`	$O(E)$	$O(V + E)$
2	`My2SATSolver.tarjan_scc`	$O(V + E)$	$O(V)$
3	`My2SATSolver.solve` (truth value assignments)	$O(V)$	$O(V)$
1-3	`My2SATSolver.solve` (overall)	$O(V + E)$	$O(V + E)$

Figure 12: Overall time and space complexity of `My2SATSolver.solve`

Performance Analysis

A set of 19 test cases ranging from 2 variables and 4 clauses to 5273 variables and 6100 clauses were tested for a total of 5 repetitions and the average time taken for each case was tabulated in a spreadsheet and plotted against the total number of variables and clauses. We obtained a strongly linear relationship with an R-squared value of 97.29%, which implies that a linear model is a statistically good fit. Hence, we conclude that the experimental results agree with our analysis that the 2-SAT solver has time complexity $O(V + E)$.

```
In [87]: My2SATSolver.solve_all(*cnf_paths)
solving 2sat-2-4a.cnf
FORMULA UNSATISFIABLE
time taken: 0.1449585ms

solving 2sat-2-4b.cnf
FORMULA SATISFIABLE
0 1
time taken: 0.03600121ms

solving 2sat-4-5.cnf
FORMULA SATISFIABLE
0 1 1 0
time taken: 0.03218651ms

solving 2sat-8-12.cnf
FORMULA SATISFIABLE
0 1 0 1 1 1 1 0
time taken: 0.05507469ms
```

```
In [88]: My2SATSolver.print_cases()
2,4,UNSAT,0.00014495849609375
2,4,SAT,3.600120544433594e-05
4,5,SAT,3.218650817871094e-05
8,12,SAT,5.507469177246094e-05
8,13,SAT,5.2928924560546875e-05
8,14,UNSAT,6.699562072753906e-05
22,54,SAT,0.000141143798828125
59,99,SAT,0.00030422210693359375
429,500,SAT,0.0023097991943359375
427,505,UNSAT,0.0025959014892578125
871,1001,SAT,0.0046939849853515625
1750,2001,SAT,0.00954580307006836
2612,3001,SAT,0.013339757919311523
3472,4001,SAT,0.024562835693359375
4339,5001,SAT,0.04754900932312012
5999,6000,SAT,0.05155587196350098
5203,6000,SAT,0.0448000431060791
5192,6001,SAT,0.06324982643127441
5273,6100,SAT,0.0304110050201416
```

Figure 13: Screenshots of the outputs of our code

				Time taken (s)					
num_vars V	num_clauses E	$V+E$	RESULT	Run 1	Run 2	Run 3	Run 4	Run 5	Average
2	4	4	UNSAT	5.60E-05	5.70E-05	0.00013208	4.98E-05	0.0001018	7.9346E-05
2	4	6	SAT	2.93E-05	4.60E-05	2.60E-05	0.00022292	3.58E-05	7.2002E-05
4	5	9	SAT	3.10E-05	3.81E-05	3.10E-05	3.19E-05	3.62E-05	3.3665E-05
8	12	20	SAT	5.20E-05	5.82E-05	5.41E-05	5.51E-05	5.51E-05	5.4884E-05
8	13	21	SAT	5.10E-05	5.60E-05	5.10E-05	5.70E-05	6.20E-05	5.5408E-05
8	14	22	UNSAT	6.70E-05	7.20E-05	6.89E-05	9.30E-05	7.08E-05	7.4339E-05
22	54	76	SAT	0.00013614	0.00014281	0.000139	0.00013709	0.00013876	0.00013876
59	99	158	SAT	0.00045514	0.00030208	0.00030112	0.00029993	0.00030208	0.00033207
429	500	929	SAT	0.00205636	0.00276494	0.00232482	0.00239015	0.00219917	0.00234709
427	505	932	UNSAT	0.00450301	0.00286436	0.00213408	0.00230908	0.00220084	0.00280228
871	1001	1872	SAT	0.00416589	0.00413895	0.00461912	0.00467896	0.00467014	0.00445461
1750	2001	3751	SAT	0.00838494	0.01086211	0.00873804	0.04861784	0.00899458	0.0171195
2612	3001	5613	SAT	0.01269984	0.06412315	0.01830912	0.01444674	0.01681209	0.02527819
3472	4001	7473	SAT	0.02500916	0.02552223	0.02458477	0.02596378	0.0188489	0.02398577
4339	5001	9340	SAT	0.02143121	0.02683902	0.02249599	0.02069712	0.06354213	0.03100109
5999	6000	11999	SAT	0.06577706	0.03409195	0.06537795	0.03270316	0.0296278	0.04551558
5203	6000	11203	SAT	0.02524018	0.0259521	0.02574396	0.05743575	0.02916503	0.03270741
5192	6001	11193	SAT	0.02604485	0.06426692	0.03791118	0.02760696	0.02843523	0.03685303
5273	6100	11373	SAT	0.02631783	0.02609777	0.02830005	0.02833676	0.06681395	0.03517327

Figure 14: Spreadsheet of time data for each of our test cases

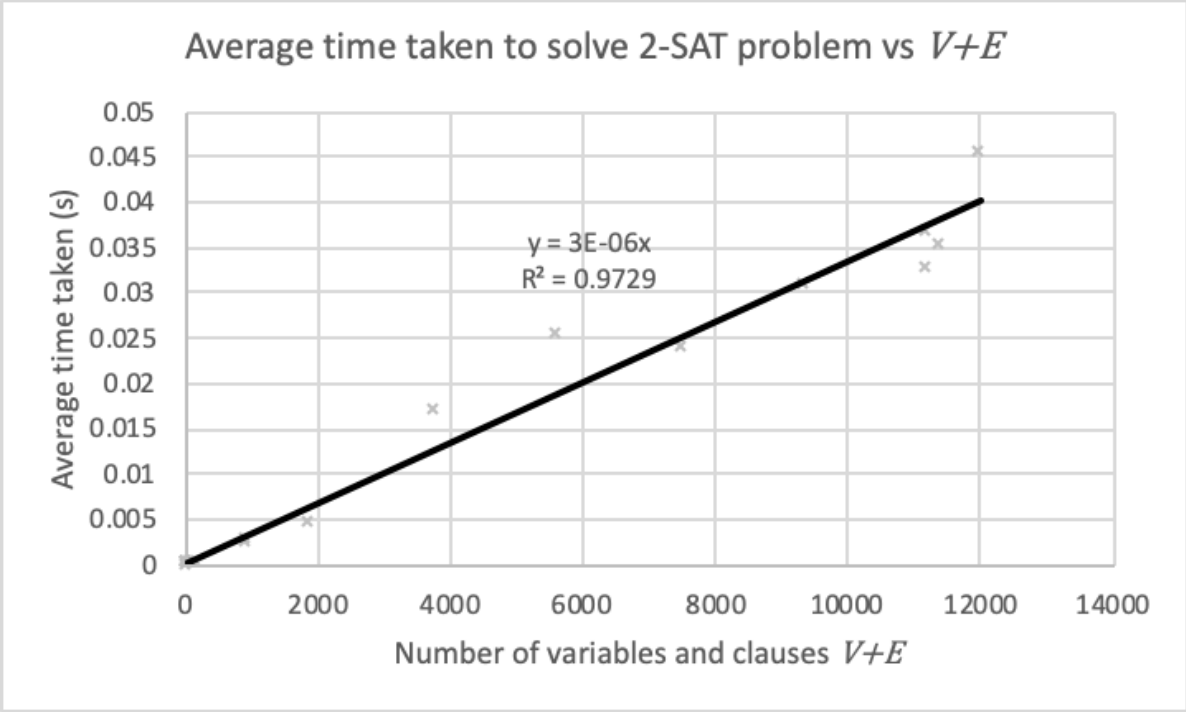


Figure 15: The graph of average time taken vs $V+E$ that we plotted from our data

Solving 3-SAT Problems?

We have found that it is possible to solve 2-SAT problems in linear time by using the strongly connected components of the implication graph. However, can the same approach be applied to solving 3-SAT problems? We know that SAT problems are NP-complete, which can never be solved in polynomial time. Furthermore, it is actually possible to reduce all SAT problems into 3-SAT problems using dummy variables and clause splitting. If we can somehow solve the 3-SAT problem in linear or polynomial time just like we can solve the 2-SAT, that would mean all SAT problems can be solved in linear time. However, that is not possible because representing 3-SAT problems using an implication graph is impossible. Take for example the simple 3-SAT formula with one clause $F(x, y, z) = (x \vee y \vee z)$. The number of implications that we can write from this is shown in Figure 16.

$F(x, y, z) = (x \vee y \vee z)$		
Clause	**Implication-Clause**	**Implications**
$(x \vee y \vee z)$	$\neg x \Rightarrow (y \vee z)$	$\neg x \Rightarrow (\neg y \Rightarrow z)$ $\neg x \Rightarrow (\neg z \Rightarrow y)$ $\neg(\neg y \Rightarrow z) \Rightarrow x$ $\neg(\neg z \Rightarrow y) \Rightarrow x$
	$\neg y \Rightarrow (x \vee z)$	$\neg y \Rightarrow (\neg x \Rightarrow z)$ $\neg y \Rightarrow (\neg z \Rightarrow x)$ $\neg(\neg x \Rightarrow z) \Rightarrow y$ $\neg(\neg z \Rightarrow x) \Rightarrow y$
	$\neg z \Rightarrow (x \vee y)$	$\neg z \Rightarrow (\neg x \Rightarrow y)$ $\neg z \Rightarrow (\neg y \Rightarrow x)$ $\neg(\neg x \Rightarrow z) \Rightarrow y$ $\neg(\neg z \Rightarrow x) \Rightarrow y$

Figure 16: Implications for a simple 3-SAT problem

When the implications for a 3-SAT formula are written out, we find that there are nested implications in each one of the implications we obtain. This means that it would be impossible to draw the graph of implications. Consider the first implication in the list $\neg x \Rightarrow (\neg y \Rightarrow z)$. It would mean that there is an edge that points from node $\neg x$ to another edge going from $\neg y$ to z, which is totally absurd and violates the fundamental definition of an edge in graph theory. In fact, it becomes exponentially difficult to verify these implications because of their nested implications. In order to verify if $\neg x \Rightarrow (\neg y \Rightarrow z)$ is true, we need to verify if $\neg y \Rightarrow z$ is true, but we also need to verify that $\neg y \Rightarrow (\neg x \Rightarrow z)$ and $\neg y \Rightarrow (\neg z \Rightarrow x)$ is true, and so on. The network of implications amongst just these 6 nodes gets very complicated. This only worsens when more clauses are added. Thus, strongly connected components cannot be used to solve 3-SAT problems since implication graphs for 3-SAT problems are not feasible. Furthermore, since all SAT can be expressed as 3-SAT, we can conclude that all k-SAT problems with $k > 2$ are not solvable with strongly connected components of an implication graph.

Bonus: Random Walk Solver

As part of the bonus component of this 2D challenge, we implemented a random walk algorithm in order to compare the performances of both algorithms. The idea of the random walk can be summarized by the following pseudocode:

Given a set of clauses C containing n variables
1. Create a mapping S and set each variable to 0
2. Repeat until MAX_TRIES or TIME_OUT
    3. Find a *bad clause* c in C whose truth value is 0 and do the following
        4. Choose a random literal $c[0]$ or $c[1]$
        5. Flip the assignment for its variable v i.e. $S[v] = 1 - S[v]$
    6. If no such c exists then print "FORMULA SATISFIABLE" and return S
7. If MAX_TRIES or TIME_OUT reached then print "FORMULA UNSATISFIABLE"

Figure 17: Pseudocode of random walk algorithm

We made a copy of `My2SATSolver` which implements `My2SATSolver.random_walk` as shown in Figure 18. We keep track of the number of steps taken for each solve.

```
@staticmethod
def random_walk(var, clauses, k=100, timeout=60):
    assignments = dict.fromkeys(var, 0)

    def calculate_truth(clause):
        for literal in clause:
            if literal > 0:
                if assignments[literal] == 1:
                    return 1
            elif assignments[-literal] == 0:
                return 1
        return 0

    def flip(literal):
        assignments[abs(literal)] = 1-assignments[abs(literal)]

    num_var = len(var)
    is_timeout = False
    start_time = time.time()
    for num_steps in range(k*num_var**2):
        if time.time()-start_time > timeout:  # check for timeout
            is_timeout = True
            break
        is_unsat = False  # reset flag
        for clause in clauses:  # find a bad clause
            if calculate_truth(clause) == 0:
                is_unsat = True
                bad_clause = clause
                break
        if is_unsat:  # choose a random literal
            if len(bad_clause) == 1:
                literal = bad_clause[0]
            elif random.random() > 0.5:
                literal = bad_clause[0]
            else:
                literal = bad_clause[1]
            flip(literal)  # and flip it
        else:
            break

    result = {"num_steps":num_steps+1}
    if not is_unsat:
        result["assignments"] = assignments
        result["result"] = "SAT"
    else:
        result["assignments"] = {}
        if is_timeout:
            result["result"] = "TIMEOUT"
        elif is_unsat:
            result["result"] = "UNSAT"
    return result
```

Figure 18: Code for `My2SATSolver.random_walk`

We ran the same set of 19 test cases for the random walk and like before, took the average time and steps taken of 5 runs. The results of the tests are as follows.

```
In [169]: My2SATSolver.solve_all(*cnf_paths)
solving 2sat-2-4a.cnf
FORMULA UNSATISFIABLE
time taken: 0.56385994ms

solving 2sat-2-4b.cnf
FORMULA SATISFIABLE
0 1
time taken: 0.01716614ms

solving 2sat-4-5.cnf
FORMULA SATISFIABLE
0 1 1 0
time taken: 0.04506111ms

solving 2sat-8-12.cnf
FORMULA SATISFIABLE
1 1 1 1 1 1 0 0
time taken: 0.08130074ms
```

```
In [170]: My2SATSolver.print_cases()
2,4,UNSAT,0.0005638599395751953,400
2,4,SAT,1.71661376953125e-05,4
4,5,SAT,4.506111145019531e-05,21
8,12,SAT,8.130073547363281e-05,33
8,13,SAT,4.291534423828125e-05,12
8,14,UNSAT,0.007756948471069336,6400
22,54,SAT,2.47955322265625e-05,1
59,99,SAT,0.0018107891082763672,173
429,500,SAT,0.03352713584899902,511
427,505,TIMEOUT,60.000054121017456,1175847
871,1001,SAT,0.44215893745422363,2887
1750,2001,SAT,0.5331230163574219,2017
2612,3001,SAT,2.6421799659729004,6221
3472,4001,SAT,1.7371840476989746,3044
4339,5001,SAT,3.994753122329712,5880
5999,6000,TIMEOUT,60.00053882598877,341283
5203,6000,SAT,7.032751083374023,8627
5192,6001,SAT,4.75616979598999,5653
5273,6100,SAT,7.140641689300537,7459
```

Figure 19: Screenshots of the outputs of the random walk

				Time taken (s)					
num_vars V	num_clauses E	$V+E$	RESULT	Run 1	Run 2	Run 3	Run 4	Run 5	Average
2	4	6	UNSAT	0.00057387	0.00057864	0.00058413	0.00056386	0.00060606	0.00058131
2	4	6	SAT	2.10E-05	1.22E-05	1.72E-05	1.72E-05	1.79E-05	1.7071E-05
4	5	9	SAT	1.62E-05	4.63E-05	1.50E-05	4.51E-05	5.58E-05	3.5667E-05
8	12	20	SAT	9.99E-05	4.60E-05	3.10E-05	8.13E-05	3.31E-05	5.827E-05
8	13	21	SAT	4.22E-05	5.63E-05	4.29E-05	4.29E-05	6.60E-05	5.0068E-05
8	14	22	UNSAT	0.00948024	0.00718808	0.00633192	0.00775695	0.00766397	0.00768423
22	54	76	SAT	2.48E-05	6.60E-05	2.31E-05	2.48E-05	2.50E-05	3.2759E-05
59	99	158	SAT	0.00093579	0.00059319	0.00212216	0.00181079	0.00099087	0.00129056
429	500	929	SAT	0.02827287	0.03828001	0.02319884	0.03352714	0.02194118	0.02904401
427	505	932	TIMEOUT	60.0001128	60.0001171	60.000041	60.0000541	60.000103	60.0000856
871	1001	1872	SAT	0.29412293	0.57056189	0.10399508	0.44215894	0.29013133	0.34019403
1750	2001	3751	SAT	0.63702512	0.62231517	0.49970579	0.53312302	0.53207088	0.56484799
2612	3001	5613	SAT	3.15672183	3.58564711	2.42671585	2.64217997	3.34557104	3.03136716
3472	4001	7473	SAT	2.04896617	1.81363821	2.39563632	1.73718405	1.85088372	1.96926169
4339	5001	9340	SAT	3.62763596	4.81640673	5.06936288	3.99475312	3.58748794	4.21912932
5999	6000	11999	TIMEOUT	60.0004909	60.0005171	60.0003028	60.0005388	60.000288	60.0004275
5203	6000	11203	SAT	6.926682	5.31705785	7.38890171	7.03275108	10.936341	7.52034674
5192	6001	11193	SAT	4.36299205	6.03079009	5.19365191	4.7561698	4.77531219	5.02378321
5273	6100	11373	SAT	5.12481213	4.75711489	4.92059398	7.14064169	5.59046412	5.50672536

Figure 20: Spreadsheet of time data for each of our test cases for the random walk

				num_steps					
num_vars V	num_clauses E	$V+E$	RESULT	Run 1	Run 2	Run 3	Run 4	Run 5	Average
2	4	6	UNSAT	400	400	400	400	400	400
2	4	6	SAT	8	2	2	4	4	4
4	5	9	SAT	5	23	3	21	29	16.2
8	12	20	SAT	41	17	9	33	11	22.2
8	13	21	SAT	19	23	23	12	22	19.8
8	14	22	UNSAT	6400	6400	6400	6400	6400	6400
22	54	76	SAT	1	1	1	1	1	1
59	99	158	SAT	77	48	140	173	81	103.8
429	500	929	SAT	425	500	351	511	323	422
427	505	932	TIMEOUT	1189661	1182751	1061405	1175847	1108107	1143554.2
871	1001	1872	SAT	2025	3853	750	2887	1677	2238.4
1750	2001	3751	SAT	2426	2323	1818	2017	1897	2096.2
2612	3001	5613	SAT	7847	8160	5666	6221	7705	7119.8
3472	4001	7473	SAT	3618	3294	3480	3044	3291	3345.4
4339	5001	9340	SAT	5410	7037	6652	5880	5195	6034.8
5999	6000	11999	TIMEOUT	376061	265000	678356	341283	672178	466575.6
5203	6000	11203	SAT	8027	6227	7423	8627	10691	8199
5192	6001	11193	SAT	5247	6995	5573	5653	5204	5734.4
5273	6100	11373	SAT	6139	5821	5474	7459	6512	6281

Figure 21: Spreadsheet of steps data for each of our test cases for the random walk

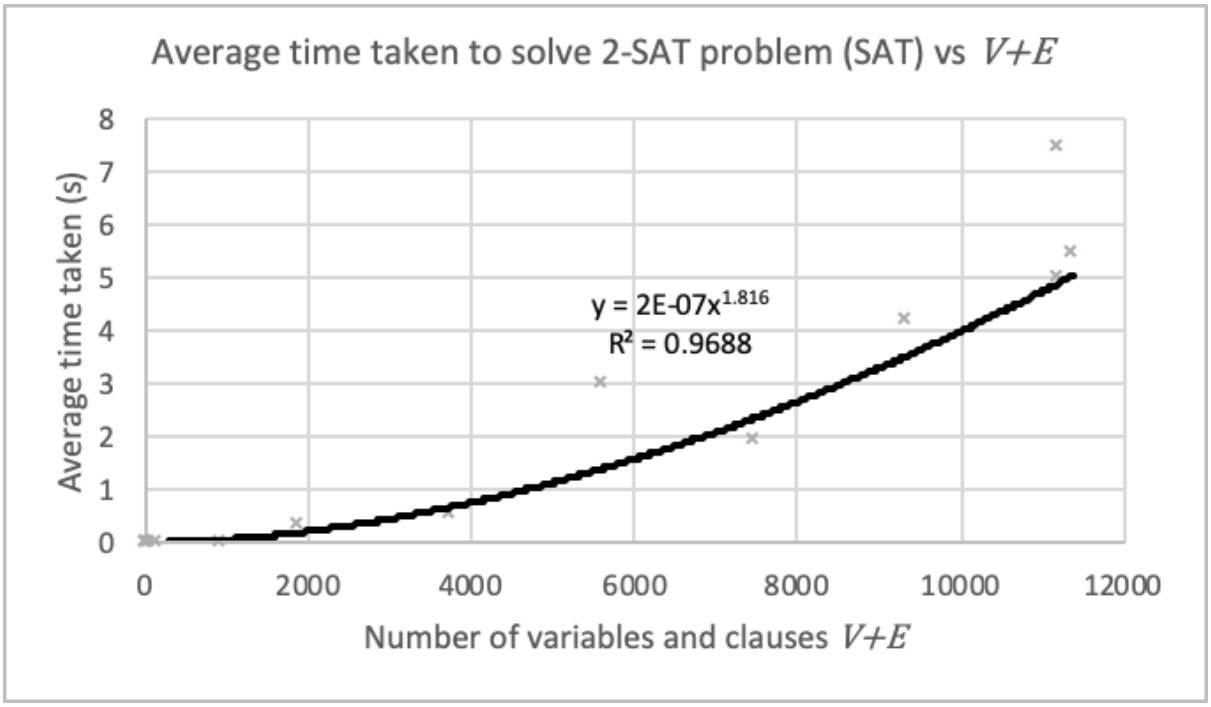


Figure 22: The graph of time taken vs $V+E$ that we plotted from the random walk data

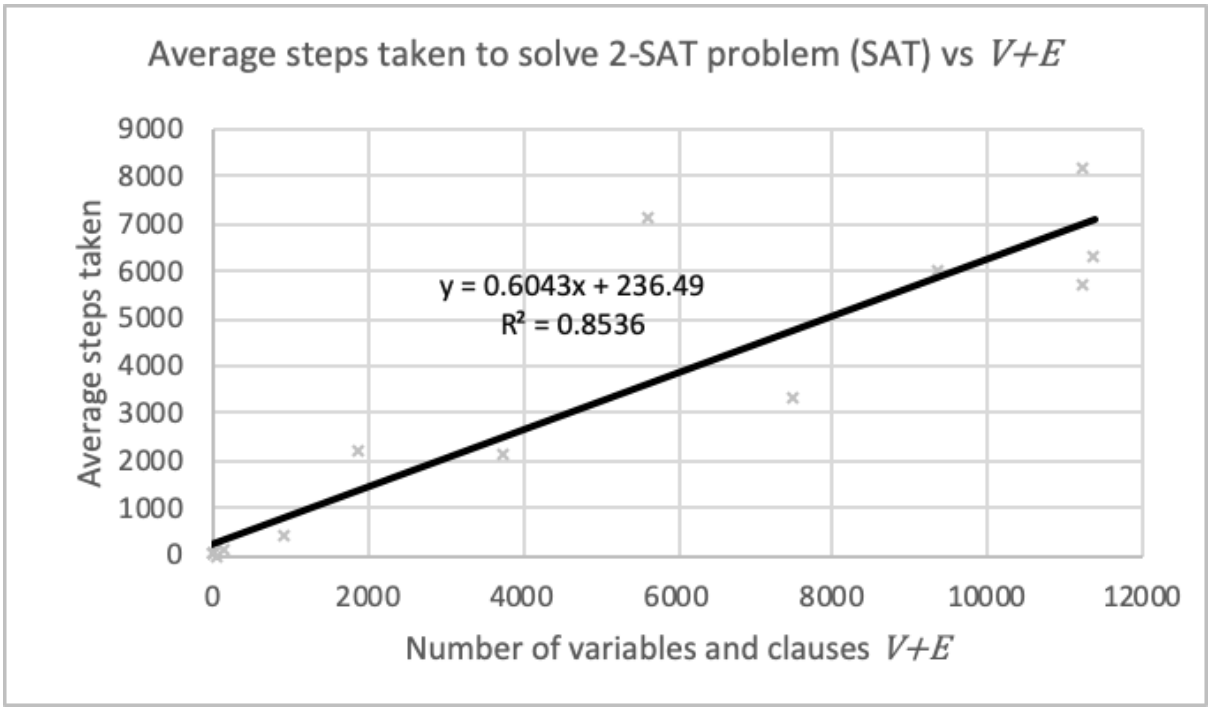


Figure 23: The graph of steps taken vs $V+E$ that we plotted from the random walk data

It is clear to see that the random walk does not do well compared to the SCC approach when it comes to 2-SAT problems. This is especially true for UNSAT cases, which is $O(E)$ for SCC but is always the worst case for the random walk, only returning after trying a maximum number of times, which by theory is set to $\Theta(V^2)$ to ensure accuracy. Even for SAT cases, we see that the random walk has polynomial time complexity, which fares worse than SCC which is linear.

We would like to discuss why the random walk is worse than the SCC approach for 2-SAT. The random walk algorithm is based on probabilities. The argument is that if there is a 50% chance while flipping a literal's assignment that we get one step closer to the correct answer, then by calculations with Markov chains the average number of steps required is $n^2 - i^2$, where n is the number of variables and i is the number of correct assignments at the start (we initialise all variables to 0, and i is the number of them which are correct). There is certainly an element of luck, and sometimes running time can vary a lot, as seen in the data collected (as much as 100%, see runs 2 and 3 for $V + E = 76$ in Figure 20). However, on average the expected number of steps is $O(n^2)$.

We find it interesting to note that the results (only for SAT cases) suggest that number of steps scale linearly. This is much better than what the theory states–it should be polynomial on average. We suspect this is due to the element of randomness and could just be an anomaly. Nonetheless, the time complexity is still polynomial. The number of steps is $O(V^2)$ according to theory while the complexity of each step is $O(E)$ as we search through all clauses to find a bad clause. Thus, the overall time complexity is $O(V^2E)$, which is polynomial. Our results seem to support this, suggesting that our basic random walk algorithm runs in $O(n^{1.8})$ time.

On an ending note, we note that this implementation of the random walk is very basic and naive, and many improvements can be made in order to optimize the running time. But arguably, it cannot compare the linear time taken to check UNSAT cases for SCC. However, the random walk algorithm can be extended to 3-SAT problems and beyond, where the SCC approach will not work anymore. In fact, random walk is one of the common strategies used in modern 3-SAT solvers such as Schöning's Algorithm, which can achieve time complexities of $\Theta\left(1.33^n\sqrt{n}log(n)\right)$.

References

"2-SAT" CP-Algorithms. Accessed Nov 9, 2019 https://cp-algorithms.com/graph/2SAT.html

"Tarjan's Algorithm to find Strongly Connected Components" OpenGenus Foundation. Accessed Nov 9, 2019. https://iq.opengenus.org/tarjans-algorithm

"Robust topological sorting and Tarjan's algorithm in Python" Jan 11,2012. Accessed Nov 9, 2019. http://www.logarithmic.net/pfh/blog/01208083168

Random Search on 3SAT. (2016, April 19). Retrieved November 10, 2019, from https://www.comp.nus.edu.sg/~rahul/allfiles/cs6234-16-random-3sat.pdf