2D Project — 50.004 Introduction to Algorithms

V S Ragul Balaji James Raphael Tiovalen Anirudh Shrinivason Jia Shuyi Gerald Hoo Shoham Chakraborty

1 Part A — Deterministic Graph-Based Algorithm

1.1 Overview of Algorithm

A solver for a 2-SAT problem can follow many methods. For a 2-SAT problem to be SATISFIABLE in Conjunctive Normal Form (CNF), every clause must be `true`. Since there are only 2 literals in each clause in a 2-SAT problem, within a single clause, we notice that we can form two implications, thus converting it into an Implicative Normal Form (INF). This sets one of the literals within the clause to be `false` and for the entire clause to be `true`, we need the other literal to be `true`. In other words, we are forbidding the four possible joint assignments of a pair of literals. This defines certain constraints between the variables, which can be propagated throughout the whole implication graph. Each pair of constraints can be considered to be an edge between the variables in an implication graph for the boolean satisfiability problem.

To create the implication graph, we first implement a directed graph using an adjacency list. In the `Graph` class defined in the `dfs/kosaraju.py` file, we create the adjacency list by defining the graph's vertices, as well as a function to add edges. Since each variable produces two literals (the variable itself and its negation), we create $2n$ vertices, where n is the number of variables. These vertices are added as the keys of the dictionary G defined in the `Graph` class. The values for each key would be a list containing consecutive outgoing edges from the vertex defined in the key. This keeps the connection between each individual vertex to the other vertices.

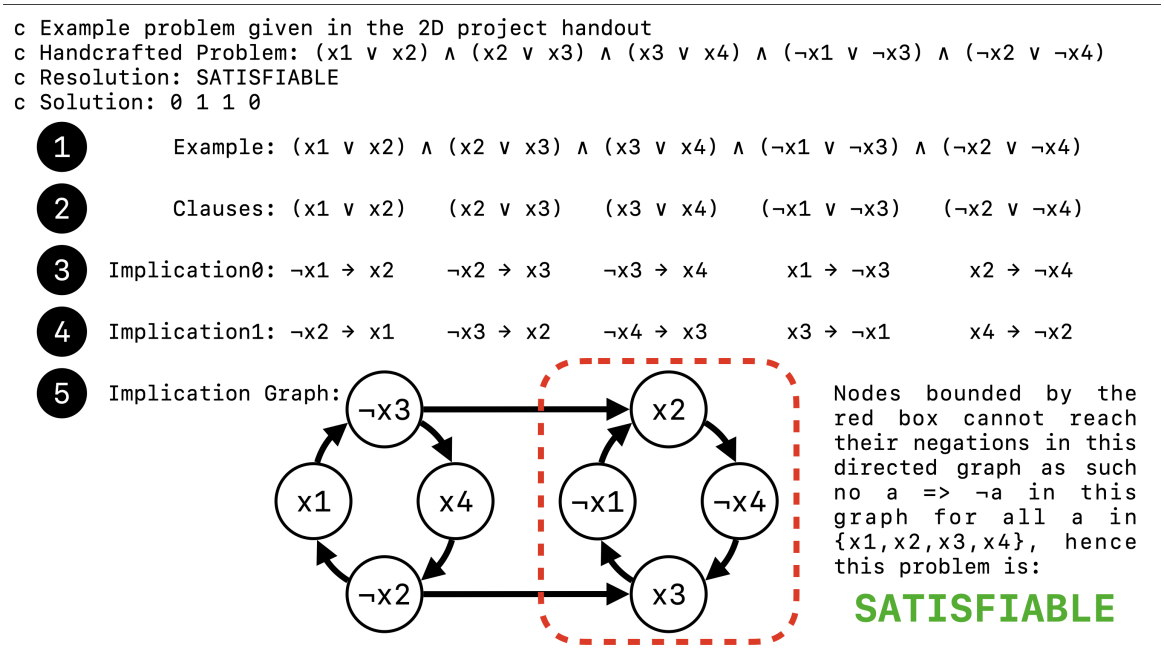

Figure 1: Guided example of solving 2SAT by hand in polynomial time

In Figure 1, we demonstrate this process using an example. This example is also defined in the `cnf/example.cnf` file. Steps 3 and 4 in Figure 1 are defined in lines 110-111 in the `kosaraju.py` file. Using the implication graph, we find the Strongly-Connected Components (SCCs) of the graph and group vertices that are along the same path using Kosaraju's Algorithm. We first create a stack and a depth-first search (DFS) is implemented to traverse through the graph. Adjacent vertices are pushed to the stack. The same procedure is executed on the inverse of the graph, where the direction of all of the edges is reversed. After that, while the stack is not empty, we pop each vertex from the stack. Using this algorithm, we can find all the SCCs of the graph.

Next, we can check for contradiction in each of the SCC. If a variable and its corresponding negation is in the same SCC, the whole 2-SAT formula is deemed to be UNSATISFIABLE, as it is impossible to assign two literals to the same variable. This is because within a single SCC, you can traverse from any vertex to any other vertex. Otherwise, the formula is deemed as SATISFIABLE.

If the formula is SATISFIABLE, we can output a possible solution for the formula. We can do this by grouping the SCCs together and connect them using a directed acyclic graph. Using the graph in Figure 1 as an example, going in topological order, assign 0 to the first group of SCC (which is the left SCC in this specific case) and assign 1 to the second group of SCC (the one highlighted by the red rectangle). We would then get the corresponding output of the variables by equating the literals in any SCC to the assigned value.

1.2 Time Complexity Analysis

In general, we know that 2-SAT is in P and is tractable due to the mechanism of forcing the other literal to be `true` in a single clause by assigning a literal within that same clause as `true`. When we pick an assignment, 3 cases could happen:

1. We reach a contradiction. In this case, it means that there can only be a satisfying assignment if we use the other truth value for that specific variable. Thus, we can simplify the formula using this new assigned value for that variable and repeat the process.
2. The "forcing" of the value assignment for a specific variable does not affect other variables and clauses. In this case, adopt these truth values, eliminate the clauses that they satisfy and continue the process.
3. We find a satisfying assignment.

In Cases 1 and 2, we have spent at most $O(n^2)$ time and have reduced the length of the formula by ≥ 1. Thus, in total, we have spent at most $O(n^3)$ time.

In fact, our specific implementation method of using a DFS traversal on an implication graph using Kosaraju's Algorithm would take even less time:

1. To create the implication graph, we set up the vertices and edges in $O(V + E)$ time, where V is the number of vertices and E is the number of edges in the graph.
2. We implement DFS to traverse through the implication graph using Kosaraju's Algorithm in $O(V + E)$ time.
3. We set up the inverse/transpose implication graph in $O(V + E)$ time.

4. We implement the DFS again through the inverse implication graph in $O(V + E)$ time.

Thus, this reduction of the 2-SAT problem to finding SCCs implemented in our deterministic algorithm would cause the algorithm to take only linear time.

Meanwhile, k-SAT problems for $k \geq 3$ would be NP-complete (as shown by the Cook-Levin Theorem) and thus, the time taken would have non-polynomial asymptotics. This is because the size of the CNF formula is exponential in the size of the original boolean formula for $k \geq 3$.

A possible improvement that could be made would be to implement Tarjan's Algorithm to conduct the search for the Strongly-Connected Components. While both Kosaraju's Algorithm and Tarjan's Algorithm would take $O(V + E)$ time, Tarjan's Algorithm has a lower constant factor to the runtime since it would need to go through the whole graph and execute DFS only once (instead of two times for Kosaraju's Algorithm, once for the normal graph and another instance of the DFS traversal for the inverse graph).

2 Part B — Randomised Algorithm

2.1 Overview of Algorithm

function RANDOM_WALK($\mathbb{F}, L$)

// $\mathbb{F}$: *a list of clauses*
// L : *a list of all variables used in* $\mathbb{F}$

```
1: Store all variables as keys in a dictionary 𝔻 with initial values false
2: for i ← 1 to 100 × (L.length)^2 do
3:     Using 𝔻, assign boolean values to 𝔽
4:     ℂ ← all invalid clauses in 𝔽
5:     if ℂ.length ≠ 0 then
6:         V ← a random variable from a random clause in ℂ
7:         V ← ¬V
8:         Update 𝔻 with V
9:     else
10:        return SATISFIABLE
11:    end if
12: end for
13: return UNSATISFIABLE
```

First, we arbitrarily assign Boolean value `false` to all n variables. In each of the $100n^2$ steps, we randomly choose a variable from a randomly selected invalid clause (that is, the clause evaluates to `false`) and negate its assignment. We then check if the resultant formula is satisfied or not. If no solution is found after $100n^2$ steps, we return UNSATISFIABLE. If at any point during the $100n^2$ steps the formula is satisfied, we return SATISFIABLE.

2.2 Time Complexity Analysis

Let X_i be the number of correct assignments at step i. Assuming worst-case initialization, all variables are assigned incorrectly, we have $X_0 = 0$. This forces $X_1 = 1$, since flipping any variable would give us a correct assignment.

For $1 \leq i \leq n-1$, the probability for X_i transiting to X_{i+1} is at least $\frac{1}{2}$ while that to X_{i-1} is at most $\frac{1}{2}$. This can be easily seen from the table below:

Wrong Clause $A + B$		A	B
Actual Value		T	T
Both Wrong Assignment		F	F
One Wrong Assignment		F	T
$P(X_i \text{ to } X_{i+1})$	Both Wrong	1	
	One Wrong	0.5	
$P(X_i \text{ to } X_{i-1})$	Both Wrong	0	
	One Wrong	0.5	

Let us suppose the worst case – that the probability X_i goes up is $\frac{1}{2}$, and down is $\frac{1}{2}$. This process is similar to a random walk.

Let h_i be the expected number of steps to reach n on our random walk when we start at step i. We have

$$h_i = \frac{h_{i-1}}{2} + \frac{h_{i+1}}{2} + 1 \quad \Rightarrow \quad h_i - h_{i+1} = h_{i-1} - h_i + 2. \tag{1}$$

Using the base case $h_0 = h_1 + 1$, the next 2 steps are

$$h_1 - h_2 = h_0 - h_1 + 2 = h_1 + 1 - h_1 + 2 = 3, \tag{2}$$
$$h_2 - h_3 = h_1 - h_2 + 2 = 3 + 2 = 5, \tag{3}$$

where Eqn. (2) is substituted into Eqn. (3). By careful observation, we formulate the following expression:

$$h_i - h_{i+1} = 2i + 1. \tag{4}$$

As shown above, this expression holds for $i = 1$. Assume the expression is `true` for $h_k - h_{k+1} = 2k+1$ for some positive integer k, we want to prove that the expression holds for $i = k + 1$.

$$\begin{aligned} h_{k+1} - h_{k+2} &= h_k - h_{k+1} + 2 \qquad \text{By Eqn. (1)} \\ &= 2k + 1 + 2 \\ &= 2(k + 1) + 1. \end{aligned}$$

Thus, by mathematical induction, Eqn. (4) is true for all $k \in \mathbb{Z}^+$. Using Eqn. (4), we sum all the steps from $i = 0$ to $i = n$:

$$\begin{aligned} h_0 &= h_n + \sum_{i=0}^{n-1} (h_i - h_{i+1}) \\ &= \sum_{i=0}^{n-1} (2i + 1) \\ &= n + 2\left(\frac{n^2 - n}{2}\right) \\ &= n^2, \end{aligned}$$

where $h_n = 0$.

Therefore, the average time complexity of the randomised algorithm is $O(n^2)$. In other words, we will find a solution in n^2 steps **on average**. If we decide to run the algorithm for $2n^2$ steps, the probability of not finding a solution is at most $\frac{1}{2}$. Thus, if we run the algorithm for $100n^2$ steps (as it is in the pseudo-code), the probability of not finding a solution is $\left(\frac{1}{2}\right)^{50} = 2^{-50}$.

3 Performance Comparison

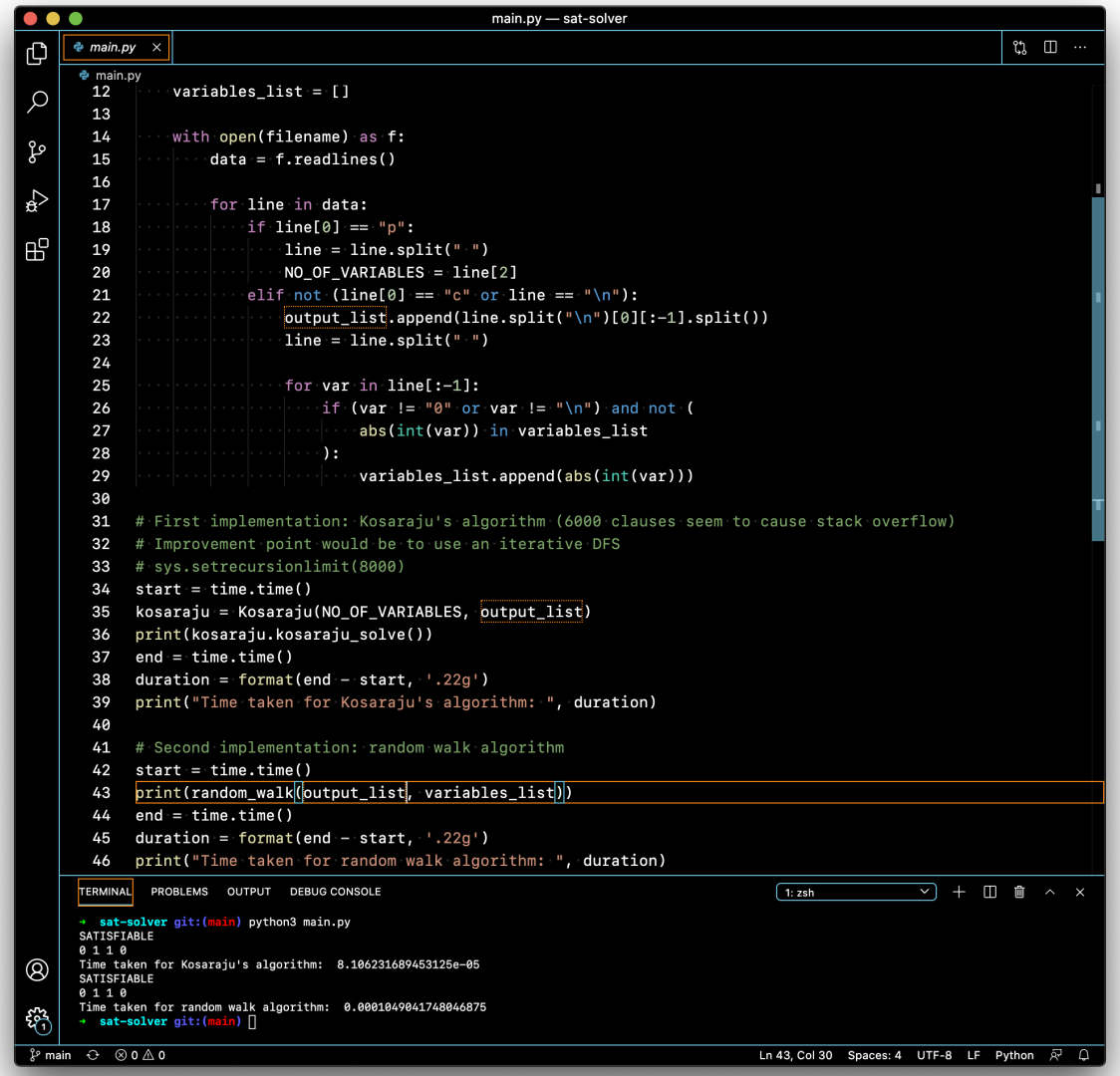

Figure 2: Benchmarking script used to compare the speed of the two implementation in Part A and Part B. Kosaraju's Algorithm runs faster than the randomised algorithm as predicted by the algorithmic analysis.

Even though the probability of not finding a solution is very small (2^{-50} for $100n^2$ steps), the randomised algorithm is not a practical substitute for the deterministic one since it takes longer time than the deterministic algorithm. There is also some chance that a solution is not found and hence an incorrect conclusion/statement of the problem's satisfiability could be made. The randomised algorithm is also dependent on the size of the variables.

However, we should not dismiss the usefulness of the idea of randomised local search entirely. We

should be aware that the strategy of using randomised local search is useful to improve over naive brute-force search for NP-complete k-SAT problems, such as the 3-SAT problem. In fact, for a 3-SAT problem, the naive brute-force method would take $O(2^n)$ time, while a version of the randomised local search with a clever twist (such as Schöning's stochastic local search algorithm) would take $O\left(\frac{4}{3}^n\right)$ time, which is significantly better than $O(2^n)$.