# 50.004 Introduction to Algorithms – 2D Report

## Glen Choo, Goh Jia Hao, Lau Siaw Young, Nikhil Sharma, Tan Hao Qin

## **Introduction**

In our 2D project, our group is tasked with implementing an algorithm that would allow us to be able to solve a 2-Satisfiability problem in polynomial time. A boolean satisfiability problem entails finding out if it is possible for us to assign values to boolean variables such that the given boolean formula can be fulfilled. In a 2-Satisfiability problem, the boolean formula given contains clauses that at most contains two different literals. For example, as the input to a 2SAT is a set of clauses, and each clause is made up of two literals. As the clause is the OR of the two literals, at least one of the literals would have to be true, in order for the clause to be satisfied, and as the clauses are in conjunction with each other, each of the clauses has to be satisfied in order for the entire expression to be satisfied. After much research, we were able to implement two separate algorithms that can solve the problem in linear time, the first of which makes use of the properties of strongly connected components within the problem’s implication graph, whereas the second is a randomizing algorithm that attempts to find a solution through a random walk.

## **Algorithm 1 – Strongly Connected Components**

The first algorithm is adapted from the linear time algorithm discovered by Aspvall, Plass and Tarjan in 1979. The algorithm consists of several components and makes extensive use of graphs. Our algorithm can be divided into several parts: reduction of the problem, forming an implication graph, finding strongly connected components and lastly assignment of the answer.

Our algorithm starts off by reducing the problem given. We do so by finding trivial literals as well as single literal clauses. Trivial literals are literals that do not have its negation found within the entire boolean formula. For example, the following formula contains a trivial literal as as its negation, , cannot be found within the entire formula.

Equation 1. A formula containing a trivial literal,

Since its negation does not appear in the formula, we can safely assign variable to be true as there will not be a conflict. After the assignment, the formula can be simplified by removing all clauses containing literal from the formula as these clauses will already be satisfied.

Equation 2. Reduced formula from Equation 1, after literal

Another way we can reduce to problem is by finding unit clauses, clauses that contains a single literal. For example, the following formula contains a unit clause which contains a unit literal .

Equation 3. A formula containing a unit clause with a single literal,

In single literal clauses, the only way to satisfy the clause is to set the lone variableto be true. Similarly, we can follow the above steps assign the literal given to be true, and remove all clauses containing that single literal. An addition step that needs to be done is that we would have to remove its negated literal from all clauses in the formula as its negated literal be false.

( )

Equation 4. The simplified formula after removing the unit clause from Equation 3

Upon the simplification of the problem, it is likely that we will be able to further simplify the problem. By reiterating the simplification algorithm until the formula stays constant, we would be able to reduce our problem to a much simpler form. Furthermore, given that our problem is a 2-SAT, our formula will contain only two-literal clauses and empty clauses, or even no clauses. In the cases where there are empty clauses, our algorithm stops as an empty clause is unsatisfiable, which implies that the formula would also be unsatisfiable. If the formula contains no clauses, it implies that the formula is already satisfied as we only remove satisfied clauses from the formula. We can obtain the satisfying solution by retrieving all previously assigned variables and arbitrarily assigned unassigned variables to be true. We can do these assignments as the values of these variable does not affect the satisfiability of our formula.

Finding trivial literals takes a complexity of , where refers to the number of variables, and refers to the number of clauses. This is due to the fact that we would have to iterate through clauses 2n times for every single variable we have. Finding unit clauses is more complicated as it takes a time complexity of as we would have to iterate through clauses to find unit clauses, and when we find a unit clause, we would have to iterate through clauses to remove our literals and clauses. However, we can reduce the complexity of the algorithm to by computing an occurrence map for every single literal that maps the clauses that each literal that appear in, as well as occurrence count of every literal in the formula. With an occurrence count, finding trivial literals would take a time complexity of ) as we only have to refer to the occurrence count to find all trivial literals. Finding unit clauses takes ) time. With the occurrence map, removing clauses and literals from takes time. As such, the entire process will take a time complexity of where refers to the number of instances where a unit clause and trivial literal is found.

After the simplification of the problem, we would proceed on with forming an implication graph that represents the formula. As our remaining clauses are all disjunctions of two literals. We are able to represent the disjunctions as two implications on an implication graph due to the following logic. In this example clause, , the clause is satisfiable when either or is true. As such, if is false, it implies that is true and vice versa.

As such, given the above clause, we would obtain the following implications, and . An implication graph is a graph where all possible literals in the problem are represented as vertices and all implications in the formula are represented as directed edges from one literal to another. To form the implication graph takes a time complexity of as each clause would result in two implications that we would have to add to our graph.

After forming the implication graph, our algorithm attempts to find strongly connected components in the implication graph. Strongly connected components are sets of vertices on the graph that is reachable from every other vertices in the set. As such, this implies that if any vertex which represents a literal in a strongly connected component is true, this would imply that all other literals in the set is also true. The algorithm we implemented is Tarjan’s strongly connect components algorithm which implements a depth first search within the graph which explores the edge at most twice. As such the algorithm runs in time.

After sorting the graph into its strongly connect components, our algorithm looks at each literal within each set of strongly connected components that we found. If there exists a set which contains a literal and its negation, the formula would be unsatisfiable as all components within the set must hold the same value, otherwise, the formula is satisfiable. This segment takes complexity as we look at all literals once. Given a satisfiable formula, our algorithm would then proceed on to condense the graph by condensing each set of strongly connected components into one vertex, and removing all edges that point towards itself. Our resultant graph will be a directed acyclic graph. Due to the Tarjan’s algorithm, the resultant sets would already be in reverse topological order. Starting from our last vertex, we would assign each component (literal) within the condensed vertices in topological order to be false if it is **not** already assigned. If the selected condensed vertex has any edges to other vertex, we would also recursively assigned its components to be false before we move on to the next vertex. Once the algorithm ends, if there are any unassigned variables, we would assign an arbitrary value to it and our final assignment will be a solution that satisfies the given formula. This takes a complexity of time as we assign each variable once. All in all, the entire algorithm would take a complexity of .

## **Algorithm 2 – Random Walk**

In order to compare and contrast between two algorithms to obtain a faster one, we went with one deterministic algorithm as outlined above, and one randomized algorithm. A deterministic algorithm is a type of algorithm that, given the same input, will always produce the same output. This helps ensure reliability, and hence can be used in the real world without a fear of having random issues affect the program.

On the other hand, randomized algorithms tend to have faster execution times, as well as smaller space requirements. They are also relatively easy to comprehend. Randomized algorithms are algorithms that rely on random variables in order to make random choices. This would mean that the output can vary if the algorithm is run multiple times using the same input. Randomized algorithms are useful because instead of making a guranteed good choice, we can make a random choice that is usually faster, and hope that is good. This is particularly useful if guaranteeing a good choice is difficult. Furthermore, random algorithms have no worst case input. However, there might be a case where the randomized algorithm might be slow, or give a wrong answer.

The random walk algorithm is an algorithm that was first proposed by Papadimitriou that runs in polynomial time, with a probability of at least ½. The algorithm starts off by assigning all the variables to a certain value, be it true or false, and then attempts to evaluate the expression to make sure that all the clauses are satisfied. Once the algorithm finds a failed clause, it randomly selects a literal within the failed algorithm, and attempts to flip the value, i.e. if the boolean value of the literal was false at first, it would be flipped to become true, and vice versa. This loop would run until all the clauses have been satisfied, or until a certain number of iterations have been reached. The second limit is set in place as the flipping of boolean variables can go on indefinitely. This algorithm makes use of the random walk theory, where small fluctuations, and the succession of these steps can lead to an eventual answer.

For the random walk, the hamming distance between the correct variable assignments and the current variable assignments are dependent on the probability that we are flipping the correct variable. This probability of decreasing the hamming distance by one is about ½, however the same probability applies to increasing the hamming distance. So we can see that the average number of steps needed to walk from position 0 to a position , is about . Hence the average amount of steps needed to walk from a position i to a position n is about . As mentioned before, the probability of the hamming distance increasing by 1 is about ½, and therefore the average steps taken to move from a certain position i to a position would be where . This can be resolved to .

𝐴ccording to Markov’s Inequality, there is an upper bound for the probability that a function of a random variable is greater than or equal to some positive constant. Therefore, the probability that after steps we have not reached n is about .

## **Conclusion**

Therefore, the deteministic algorithm runs in time while the randomized algorithm runs in at most time. Hence we can see that at small values, and by a good assignment of values, we are able to reach an answer quickly using the randomized algorithm, but the determinstic algorithm will perform better under larger numbers, or under cases where the expression is unsatisfiable. In order to better visualize, and see this values in effect, various test cases were run. They ranged all the way from small numbers of variables and clauses, to large cases that went up all the way to 6000 variables and clauses. From the table, we can see that the randomized algorithm performed better where there were not many clauses and variables, as compared to when there was a huge number of variables and clauses. This happens because the randomized algorithm has a worse case of but if by chance, the correct variables were to be flipped, the equation should be solved faster. However, in the case of the deterministic algorithm, the algorithm takes time, in any case.

|  |  |  |
| --- | --- | --- |
| Test Case (Variable, Clause) | Randomized Algorithm (ms) | Deterministic Algorithm (ms) |
| 4,5 | 1.06 | 3.12 |
| 22, 64 | 9.34 | 9.22 |
| 2, 4 (Unsat) | 2.45 | 2.93 |
| 2, 4 | 1.21 | 3.46 |
| 8, 12 | 1.67 | 4.14 |
| 460, 99 | 8.69 | 15.82 |
| 500, 500 | 67.23 | 51.23 |
| 500, 505 (unsat) | 32387.50 | 43.59 |
| 1000, 1001 | 167.78 | 78.14 |
| 2000, 2001 | 283.20 | 118.23 |
| 4000, 4001 | 682.67 | 269.69 |
| 5000, 5001 | 1197.20 | 334.12 |
| 6000, 6001 | 1520.12 | 458.67 |

Figure 1. Collection of data from running the algorithms

Thus from these test cases, we can see that the randomized algorithm works well when it comes to small test cases, but the deterministic algorithim is much faster than the randomized algorithm when there are many test cases, and especially so if they expression is unsatisfiable. Therefore, we can safely say that the deterministic algorithm is preferred to the randomized algorithm as it is only slightly slower than the randomized algorithm where there are a few clauses and variables, but it is much faster when it comes to having large test cases. It also helps that the deterministic algorithm always returns the correct answer, but the randomized algorithm has a chance of returning the wrong answer.