**2D Algorithm Report**

**Team Members:**

**Joel Huang, Lim Yu Fei, Valerene Goh,**

**Lim Yao Jie, Cheah Yee Kei, Chan Terng Tseng Nigel**

**Problem Definition**

A 2-SAT (2-satisfiability) problem consists of the issue of finding an assignment for a set of Boolean variables that are configured in CNF (conjunctive normal form) for clauses that have a maximum limit of two literals. A cluster in CNF is basically an AND of ORs, for example

In which, should every disjunction pair such as

return true, one or both of those variables should be true, and the problem is satisfied.

1. **Proposed Solution**

The 2-SAT problem is identified to be a NP (Nondeterministic Polynomial time) problem. This means that it can be solved in polynomial time. Instead of using the DPLL algorithm proposed, we decided to implement a graphing solution, which will enable us to solve the 2-SAT problem in linear time.

**Implication Graph**

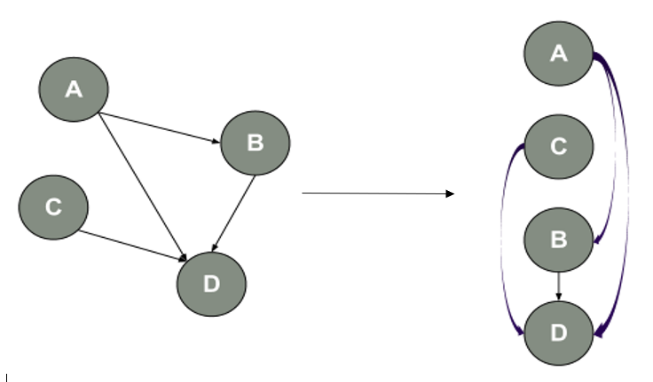
Given all and in the formula, we create a Hashmap of the vertices, with their edges as and , as both implications are equivalent to the OR operation

. A cyclic connection define strongly connected components (SCC) within the graph: e.g. . A 2-SAT problem is satisfiable if and only if there are no contradicting elements in **ANY** of its SCCs. Initialization of the graph will be in time.

**Topological Sort**

Topological sort is then implemented on the implication graph, which will order every directed edge (our implications), , such that will always appear before . Figure 1 illustrates the workings of a topological sort.

Depth-First Search (DFS) is then used for graph traversal to ensure that every vertex is only visited once, which will have a complexity of where is the number of vertices and being the number of edges in the problem.



**Figure 1: Topological sort**

**Pseudocode for DFS:**

1. For every vertex:
   1. check and verify if it has already been visited.
      1. If not, it will visit and set it to be visited.
   2. Recursively call the search function for every connected vertex
   3. Push the vertex into the stack.
2. After all vertices are visited, the stack is returned.

The resultant stack will be a chain of implications for all and . To satisfy the equation, we will have to ensure that the Boolean equation of the entire chain of implications results in TRUE. The truth table for an implication is as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P | Q | P → Q |  | P | Q | P → Q |
| T | T | T |  | F | T | T |
| T | F | F |  | F | F | T |
| F | T | T |  | T | T | T |
| F | F | T |  | T | F | F |

**Figure 2: Truth tables for implications**

As we can see from the tables, there will be two cases:

* If appears before (right table), it means , and hence must be TRUE for the expression to be TRUE.
* If appears before (left table), it means , and hence must be FALSE for the expression to be TRUE.

The processing of the resultant array to produce a solution for the 2-SAT problem is doable in as the algorithm needs to loop through the array of implications only once.

**Satisfiability**

To determine the satisfiability of the 2-SAT problem, we inject the path-based strong component algorithm to run parallel to the topological sorting to generate the SCC groups.

After grouping, we check whether a contradiction exists within each SCC group, which is by checking whether and exists in the same group. This means that

and , which is not the case. We will exit the program at this stage and return an error as the boolean formula is unsatisfiable.

The satisfiability algorithm takes time to compute.

**Performance Discussion and Optimization**

Half of our application’s running time is determining whether a vertex’s negation exist within the same SCC group (checking for contradiction). If we were to know whether the Boolean expression was satisfiable beforehand, the performance can be at least 50% faster by forgoing the satisfiability check. The resultant data is also unsorted as DFS visits each node randomly. If the result requires to be sorted an extra time complexity will have to be added.

1. **(Bonus) Randomizing algorithm**

We decided to implement Papadimitriou’s randomizing algorithm to solve the 2-sat problem. The algorithm makes use of a random walk method to flip a randomly-chosen variable of an unsatisfied clause for each iteration, until there are no more unsatisfied clauses left, in which case the algorithm will exit and return the updated Boolean values of the variables. By placing an a priori bound on the number of iterations, say 2n2logn iterations for the worst case, if the algorithm exits with remaining unsatisfied clauses, we can safely say that a satisfied assignment doesn’t exist and that the formula is unsatisfiable.

The following is a pseudocode of our implementation of Papadimitriou’s algorithm (for a k-sat problem, since our deterministic approaches are catered for k-sat as well).

**Pseudocode for Papadimitriou:**

1. Repeat log2n times:
   1. Randomly assign all variables to true/false.
   2. repeat 2n2 times where n is number of variables
      1. If current assignment satisfies all clauses, return “FORMULA SATISFIABLE”.
      2. If there exist unsatisfied clauses, do iteration of local search: pick an arbitrary unsatisfied clause, flip the value of one of the k local variables in this clause. (for 2-sat, k = 2)
2. After all iterations, if never return satisfiable, return “FORMULA SATISFIABLE”.

**Performance Discussion and Optimization**

From the pseudocode, it is observable that this particular algorithm runs in polynomial time as there are iterations for the worst possible case. On average, however, due to the iteration on local variable search, randomisation will unlikely lead to the worst-case scenario unless the solution is unsatisfiable. The average runtime is hence only . We can also improve the progress measure of the random walk by varying the probability of the randomly-chosen variable during local search after each iteration (see annex for theorem). In the case of the above, we set it to have equal probability for simplicity, i.e. the variable will be chosen uniformly at random.

1. **Comparison and Evaluation:**

|  |  |  |
| --- | --- | --- |
|  | Deterministic Algorithm (10 Clauses) | Randomizing Algorithm  (10 Clauses) |
| Time (ms) | 0.39 | 1.7 |
| Complexity |  |  |

The time is taken by running across 3 different files with 10 clauses 5 times, and taking the average.

Complexity of an algorithm implemented is a measure of the amount of time required by an algorithm to check satisfiability of the CNF file input. The randomizing algorithm will generally take a longer time as compared to the deterministic algorithm as complexity is for the randomizer and for the deterministic approach when computed in linear time (section above).

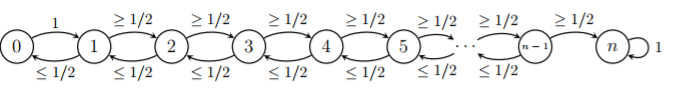
The deterministic algorithm is more efficient than the randomizing algorithm. Hence, the randomizing algorithm is **NOT** a practical substitute for the deterministic one since the time needed to check for satisfiability varies. However, the Randomizing Algorithm will work on SAT problems that are not 2-SAT, while our Deterministic Algorithm will only work for

2-SAT problems.

1. **Annex**

**Theorem**

By Papadimitriou’s algorithm, for a satisfiable 2-SAT instance with n variables, we can produce a satisfying assignment with probability . Figure 3 shows a representation of the probability for each iteration.



**Figure 3: State diagram of Papa’s random walk**

**Proof of theorem**

1. Let be an arbitrary satisfying assignment to the clause, i.e. == true (reference)
2. Let algorithm’s final assignment after iterations of the inner iteration have completed ( local flips), where
3. Progress measure: = number of agreeable variable assignments
   1. If AND , satisfiable assignment. Return “FORMULA SATISFIABLE”.
   2. If found satisfiable assignment before , Return “FORMULA SATISFIABLE”.
4. Suppose FALSE (not a satisfying assignment to the clause). Then pick one of the unsatisfying clause – let chosen clause be one with variables must give a different value to one or both of the 2 variables or then choice of flipping those who differ

|  |  |
| --- | --- |
| Case 1 (100% probability) | If and differ on both and , after flipping either,  (guaranteed one more agreeable) |
| Case 2 (50% probability) | If and differ on exactly one of or , if flip the correct one by luck, (one more agreeable) |
| Case 3 (50% probability) | If and differ on exactly one of or , if flip the wrong one (an agreeable variable), (one less agreeable) |

To summarize, the randomizing algorithm runs in polynomial time. It will always be correct (unsatisfiable) if there exist an unsatisfiable instance.