NOTE : Running the Java code

The zip file contains

- the Android Studio project folder *Project2D*
- files *sat1.cnf* and *unsat1.cnf* that we used as simple test cases.

The classes to be examined can be found at *code2d\src\main\java\sat*

- directedGraph
- randomWalk
- SATSolver (ignore this class since it implements DPLL algorithm for 50.001)
- SATSolverTest

The main class *SATSolverTest* takes in the CNF file of a 2-SAT problem and the name of the algorithm to be used. Please provide file path and algorithm name as program arguments. If only file path is provided, the DPLL algorithm will be used.

Instead of returning SATISFIABLE or UNSATISFIABLE, we have printed this onto the console and returned the solution (or null). Examples,

```
PROGRAM ARGUMENTS
“../sat1.cnf” directedGraph
CONSOLE
2-SAT Solver directedGraph
SATISFIABLE
5 = FALSE 4 = TRUE ~3 = FALSE ~1 = FALSE ~2 = TRUE 1 = TRUE 3 = TRUE 2 =
FALSE ~5 = TRUE ~4 = FALSE

PROGRAM ARGUMENTS
“../unsat1.cnf” randomWalk
CONSOLE
2-SAT Solver randomWalk
UNSATISFIABLE

PROGRAM ARGUMENTS
“../sat1.cnf” someAlgo
CONSOLE
Algorithm not defined
```

2-SAT Solver

USING STRONGLY CONNECTED COMPONENTS

Basis of algorithm

Every clause of a 2-SAT problem is of the form $p \vee q$, which can be expressed as $\neg p \Rightarrow q$ and as $\neg q \Rightarrow p$. We can build an implication graph from these statements with a node for every variable and its negation. Only the variable or its negation can be true at a time, hence a contradiction arises if the two nodes are connected in this graph and the problem can be deemed as unsatisfiable. Every edge in this graph represents an implication from one literal to another, thus when a group of nodes are connected together in a strongly connected component each of them has the same truth value. We can check for satisfiability by determining whether there is any literal which belongs to the same SCC as its negation. We chose to implement Kosaraju's algorithm which uses two passes of DFS to get the strongly connected components from the graph.

Further explanation

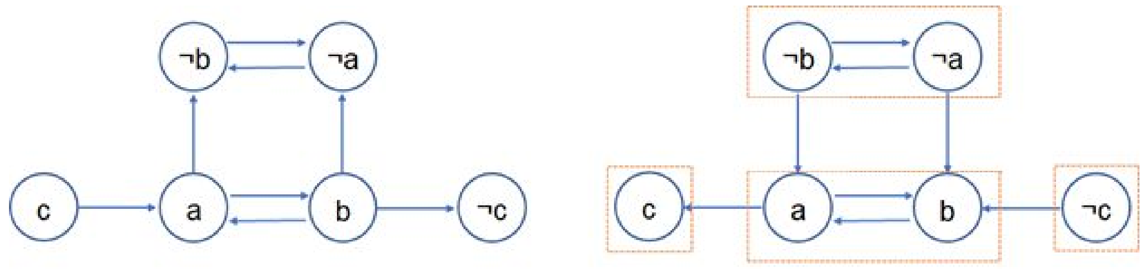
a	b	$\neg a$	$\neg b$	$a \vee b$	$\neg a \Rightarrow b$	$\neg b \Rightarrow a$
F	F	T	T	F	F	F
F	T	T	F	T	T	T
T	F	F	T	T	T	T
T	T	F	F	T	T	T

Truth table for disjunction and implications

From the above truth table, we can see that $a \vee b$ is equivalent to $\neg a \Rightarrow b$ and $\neg b \Rightarrow a$ (if one of the variables is false, then the other must be true). We can thus convert a formula in conjunctive form to the implicative form.

Consider a satisfiable 2-SAT problem as given below.

$$Conjunctive: \ (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$$
$$Implications: \ \neg a \Rightarrow \neg b, \ b \Rightarrow a, \ a \Rightarrow b, \ \neg b \Rightarrow \neg a, \ a \Rightarrow \neg b, \ b \Rightarrow \neg a, \ \neg a \Rightarrow \neg c, \ c \Rightarrow a$$



Implication graph (L) SCCs marked on transposed implication graph (R)

The implications stored as an adjacency list give the implication graph as in above figure. According to Kosaraju's algorithm, after the first DFS on this graph we perform a second DFS on the transposed graph and obtain the 4 SCCs as boxed in orange. For any variable we check that its negation does not belong to the same SCC, hence the given 2-SAT problem is satisfiable with a possible assignment of $a = FALSE,\ b = FALSE,\ c = FALSE$.

Time complexity

Consider that a 2-SAT problem is converted to an implication graph G.
Let $V = \#\ of\ vertices\ in\ G = \#\ of\ variables$
and $E = \#\ of\ edges\ in\ G = \#\ of\ implications = 2 \times \#\ of\ clauses$

Checking all the SCCs for a contradiction takes $O(E)$ since the number of SCCs is bounded above by the number of implications. Th runtime for Kosaraju's algorithm is $O(V+E)$. Hence the overall time complexity of our algorithm is $O(V+E)$.

Why does it not work for 3-SAT?

Using strongly connected components cannot solve 3-SAT problems since it is not feasible to build implication graphs for 3-SAT problems. In the 2-SAT problems, the conversion of conjunctive form to the implicative form results in at most two implications. Consider $p \vee q$, which results in two implications: $\neg p \Rightarrow q$ and as $\neg q \Rightarrow p$. However, in 3-SAT, the formula of $p \vee q \vee r$ will result in significantly more implications, such as $(\neg p \wedge \neg q) \Rightarrow r$ and $(\neg p \wedge \neg r) \Rightarrow q$ and $(\neg q \wedge \neg r) \Rightarrow p$. This will result in an implication graph that grows exponentially with more and more clauses in the formula, hence making solving only possible in exponential time. Moreover, the complexity of linearity graph makes checking for contradiction very difficult as to find whether there is a circular path that connects a literal from its negation takes significantly longer than linear time

USING RANDOM WALK[1]

Pseudocode

```
function solve(F):
      let A be a random assignment to the n variables of formula F
      repeat for MAX_FLIPS times:
            if F is satisfiable by A
                  return “SATISFIABLE”
            else
                  pick a clause c that is not satisfied by A
                  pick a variable v in c at random
                  flip the value of v in A
      end repeat
      return “UNSATISFIABLE”
```

Can it perform better than the deterministic approach?

The random walk algorithm is based on there being a probability of 0.5 of getting a step closer to (or further from) the solution when flipping a literal assignment. Following the resulting chain of equations, the average amount of time to walk from i to n is $n^2 - i^2$ where n is the number of variables and i is the number of variables whose assignments are correct.

For unsatisfiable problems, random walk is always worst-case as it returns only after flipping the maximum number of times which is set to $\Theta(V^2)$. With satisfiable problems, number of flips is set to $O(V^2)$ and each flip takes $O(E)$ since we run through all clauses in the process. We get $O(V^2E)$ overall which means it has polynomial time complexity. In either case, the random walk algorithm performs worse than the SCC algorithm with its linear time complexity.

[1] http://people.seas.harvard.edu/~cs125/fall14/lec19.pdf