50.004 Introduction to Algorithms

2D Project Report

Cohort CI03

Group 4

Gao Yunyi	1002871
Kang Tae Woong	1002854
Li Zihao	1002966
Lu Jiankun	1002959
Ong Sze Teng	1003130
Tan Yi Xuan	1002887
Zwe Wint Naing	1002791

1 Introduction and overall design

The 2D project involves building a solver for the 2-satisfiability problem which runs in polynomial time. We decided to go with the recommendation of using the strongly connected components (SCCs) approach, of building an implication graph from a set of 2-literal clauses in conjunctive normal form (CNF). We used Tarjan's strongly connected components algorithm ("Tarjan's algorithm") to produce a sequence of SCCs from a given graph, in reverse topological order. Once we have the SCCs, we can check for satisfiability of the implication graph, and if satisfiable, we then proceed to solve it, and finally print the solution to standard output. This overall solving method is programmed in our `TwoSAT.java` source file. All workings for the main part is inside the `TwoSAT` folder of the submission.

This method for solving a SAT problem only works for 2-SAT. This is because it relies on an implication graph. In 2-SAT, each clause has up to 2 literals. Being a disjunction of up to 2 literals, each clause can only be true if at least one literal is true. This means if one literal is false, the other has to be true, or in other words, a clause $(p \vee q)$ can be rewritten as a pair of implications: $(\neg p \rightarrow q) \Leftrightarrow (\neg q \rightarrow p)$. Hence, this method cannot work for 3-SAT or any n-SAT for $n > 2$, as one literal being false no longer requires another to be true, unless of course one reduces the problem into a 2-SAT problem.

2 Input parsing and graph building

As instructed, we have created a parser that takes in CNF files with the format described. However, as the focus is in solving, we have decided to follow only a stricter subset of the format: our program does not accept consecutive whitespaces in the clauses. This means no double spaces, no empty lines, and no spaces at the ends of lines. Also, all clauses must have exactly 2 literals. If there are 1-literal clauses, the clause can simply be expanded into the logical disjunction of the literal with itself, to obtain a 2-literal clause.

Input parsing

Our program first reads from an input `.cnf` file, and parses the file. As it parses the clauses, it produces ordered pairs of literals, which represents ordered pairs of nodes on the implication graph, which in turn represents edges of the graph. This part of the program also creates a list of literals from the number of variables declared in the input file. Once parsing is complete, a Graph object is created, with the list of literals and list of ordered pairs of literals, representing nodes and edges, passed to the constructor of the Graph object. This process is programmed in the `Loader.java` file.

Graph building

We have defined a `Graph` class in the `Graph.java` file that handles all the graph-related logic from problem representation through Tarjan's algorithm to actually solving the problem. The `Graph()` constructor takes in the list of literals and list of ordered pairs of literals, interprets them as nodes and edges, and produces a `Graph` object storing the adjacency lists of all nodes in the graph. This class also contains several private variables to facilitate Tarjan's algorithm.

Note: our `Graph` class does not represent a generic graph. It is an ad-hoc design without considering generality, so each instance is only intended to be used once, by calling the methods `tarjan()`, `satisfiable()` and `solve()`, in that order.

Time complexity analysis

1. Create a mapping `nodes` from literal name to `Node` object, and a mapping `adjacencyLists` from all nodes to their corresponding set of nodes that they point to.
   O(1)
2. Create each node, registering to it to `nodes`, and creating its initially empty entry in `adjacencyLists`
   O(|V|)
3. For each edge, add the child node to the parent node's `adjacencyLists` entry.
   O(|E|)

Hence overall, the time complexity of graph building is $O(|V| + |E|)$.

Note: this does not take into account the time complexity of parsing. The reason is that the parsing time complexity is linear with respect to the number of characters in the input file, but this is not expressible in terms of number of variables and number of clauses, e.g. there can be arbitrarily many comment lines, which arbitrarily increases the time needed beyond any complexity expressible in the number of clauses and variables.

3 Finding strongly connected components (SCCs)

We used Tarjan's algorithm to find the SCCs of the graph. This step is implemented in `Graph.java` as the `tarjan()` method of the `Graph` class.

To our understanding, Tarjan's algorithm can be explained as follows:

1. Keep an `index` variable as a counter for assigning indices to visited nodes in the graph. Initialize to 1. Nodes would have their indices initialized to 0 representing that a node has not yet been visited.

2. Keep a stack that serves as a temporary storage for nodes. We kept it as a private field of the `Graph` object.
3. Keep a list of lists of nodes `SCCs`, each sub-list being a list representation of an SCC.
4. For each node in the graph, visit it if it has not yet been visited.

In the context of Tarjan's algorithm, visiting a node consists of the following steps:

1. Assign the node two properties `node.index` and `node.lowlink`, both to the value of the `index` counter. The former assignment automatically marks the node as visited.
2. Increment the `index` counter by 1.
3. Push the node onto the stack.
4. For each child node the current node has, check if they are already visited.
   This step is O(number of outbound edges), not counting the time complexity of visiting the child nodes.
   1. If they are not, visit them. Once done visiting, update the `lowlink` of the current node to that of the child node if that of the child node is smaller.
   2. If they are already visited, check if they are currently in the stack. If they are, set the `lowlink` of the current node to that of the child node if that of the child node is smaller. We keep a `Node.inStack` field, to keep this step *O(1)*.
   3. If neither case fits, do nothing.
5. If the current node's `lowlink` is still equal to its `index`, pop all nodes on the stack up to and including the current node, and put them in a list which represents an SCC. Put this SCC into the `SCCs` list, and return to the parent node.
   This step is O(number of nodes to pop).

The mechanism that allows this algorithm to work is that `lowlink` represents the lowest-index node in the stack that the current node can reach through a directed path. In acyclic subgraphs, this is always the node itself, so the node forms its own SCC, with itself being the root node. In a cyclic SCC, this is always the node in the SCC with the lowest index, which we would also call the root node. Step 5 uses `lowlink` to checks whether a node is a root node. If a node is not a root node it remains on the stack before returning to its parent node. If it is a root node, this means all nodes on the stack up to the current node itself form an SCC, hence pop them and output them as a group. All these ensure that all nodes in an SCC are always popped together, thus achieving the intended effect of this algorithm.

Since nodes are popped before they return to an ancestor node, this creates a side effect that child nodes are popped before parent nodes, hence the natural order with which Tarjan's algorithm outputs SCCs is the reverse topological order.

Time complexity

The time complexity of visiting each node, minus that of visiting its children node, is:

O(number of outbound edges + number of nodes to pop)

Since each node is visited exactly once and popped exactly once, the total time complexity becomes:

O(total number of outbound edges of all the nodes + number of nodes in the whole graph)

which can be rephrased as:

O(number of edges in the whole graph + number of nodes in the whole graph)

or in symbols:

$O(|V| + |E|)$.

4 Satisfiability check

Since any node in an SCC has a directed path to any other node in that same SCC, this means if any node's literal is true, all literals in the SCC must be true, and thus also that any literal being false requires all literals in the SCC to be false; i.e. all literals in an SCC have the same truth value. An unsatisfiable graph is one that requires a variable and its inversion to have the same truth value. This translates to having a variable and its inversion in the same SCC. Hence to check for satisfiability, we can simply check that there is not any SCC in the graph that contains a variable as well as its inversion. This check is implemented as the `satisfiable()` method of the `Graph` class in `Graph.java`.

Time complexity analysis

1. For each SCC:
    1. Keep a `HashSet checklist` that keeps track of whether each variable has already been encountered. This enables $O(1)$ average case insert and membership checking, or $O(\log |V|)$ worst-case due to Java's (Java version 8+) tree-based collision handling.
    2. Look at each node. If its checklist entry is already `true`, return `false`. Otherwise, set its checklist entry to `true`.
2. The for loop did not find any problem, return `true`.

This looks at each node exactly once, so the time complexity is $O(|V|)$ average case or $O(|V| \log |V|)$ worst-case.

5 Solution assignment

If there is no SCC containing a variable as well as its inversion, one can always find a solution to the implication graph. An efficient way to do this is to assign `true` to all the variables in each SCC, in reverse topological order, or skip the SCC altogether if its variables have already been assigned values. This can be done efficiently as Tarjan's algorithm naturally outputs SCCs in reverse topological order.

The order is important as starting from a source node and assigning `true` would lead to all subsequent nodes being true. However, two complementary SCCs (SCCs which nodes are the inversions of each other) can occur in the same implication chain, and would thus simultaneously be assigned the same value, leading to a contradiction. If it is done in reverse topological order, there is no propagation of `true` values down the graph, so no assignment on an SCC would ever force the complementary SCC to be `true`.

This step is implemented as the `solve()` method of the `Graph` class in `Graph.java`.

Time complexity

In the worst case of an acyclic graph, each node forms its own SCC, so the program would have to check and skip or assign each literal exactly once. Hence, the time complexity is $O(|V|)$.

6 Overall time complexity

Adding up all the time complexity of each step, we get:

$$O(|V| + |E|) + O(|V| + |E|) + O(|V| \log |V|) + O(|V|) = O(|V| + |E|)$$

In terms of $v = ½|V|$ for the number of variables, and $c = ½|E|$ for the number of clauses, this gives:

$$O(|V| \log |V| + |E|) = O(½v \log (½v) + ½c) = O(v \log v + c)$$

If we use the average case time complexity instead, or if we can tweak the hash table implementation to eliminate collision for this particular use case, we can obtain:

$O(v + c)$

Hence, this algorithm is linear time average case and quasilinear time worst-case, with respect to the sum of the number of variables and the number of clauses. This fulfills the polynomial time requirement of the project.

7 Bonus part: randomizing 2-SAT solver

We implemented the randomizing algorithm as described. The code is in the `Rand2SAT` folder of the submission. We ran the two solvers with two sample input files. The small input file has 4 variables and 4 clauses; the large input file has 6100 variables and 6000 clauses. The time taken to run the solver is as described in the table below:

Solver	Small input file	Large input file
Deterministic solver	Avg. 42.6 ms over 10 runs	Avg. 328.5 ms over 10 runs
Randomizing solver	Avg. 43.3 ms over 10 runs	2 runs: 822.3 s, 745.5 s

Table of solver running time with given input file

Analysis

For small input files, the two solvers seem to have similar performance.

For large input files, the deterministic solver's time complexity shows its advantage over the randomizing solver's time complexity: the randomizing solver needed more than 2000 times longer to run. Therefore, for large files the randomizing solver is not a practical replacement.

This is expected:

- the time complexity of the deterministic solver is $O(v + c)$ average case, or $O(v \log v + c)$ worst-case
  (as derived in above sections)
- the time complexity of the randomizing solver is $O(cv^2)$
  (due to the v^2 average case upper bound, the $100v^2$ worst case iteration limit, and up to c clause evaluations per iteration)

Hence there should be no surprise that the running time difference is so big when input size becomes large.

Besides that, given that the randomizing solver is not guaranteed to produce the correct output, it should only be used if the application is not very sensitive to incorrect output.

Conclusion

Overall, this means that the randomizing solver is generally not a practical replacement for the deterministic solver. The only time when it is more practical is probably when one does not have enough time to implement the more complex deterministic solver.