CS 170 Project Write Up

By: Denny Hung, Kyung Geun Kim, Yoon Kim

A. Introduction

We have used 3 SAT solver to solve the Wizard Ordering problem. The 3 SAT problem is an important problem in computer science that has been studied by many computer scientists. Due to this, there exist many papers and algorithms that solves 3 SAT problem efficiently and reliably. This is the main reason why we chose to reduce Wizard Ordering problem into 3 SAT problem. We also considered other algorithms. The first algorithm we've considered was using greedy algorithm to find some approximate ordering of wizards and then using backtracking to meet all the constraints. The other algorithm we've considered was to tailor this problem into optimization problem to solve it. However, we thought this method would reveal too many local optima so the problem becomes impractically difficult. We have reduced the Wizard Ordering problem into 3 SAT and we used a solver found online to solve the problem.

B. Reduction

1. Main Idea
   Let one of our constraint be $A\ B\ C$ then, this implies $\{(A < C)\ \textit{and}\ (B < C)\}$ or $\{(A > C)\ \textit{and}\ (B > C)\}$. Define a boolean variable X_{AB} which implies $A < B$ for any A, B. Because we want to keep it as simple as possible, we will always have the subscript in alphabetical order meaning that will not define something like X_{BA}. Then, the part of constraint $\{(A < C)\ \textit{and}\ (B < C)\}$ becomes $X_{AC} \wedge X_{BC}$. However in order to express $\{(A > C)\ \textit{and}\ (B > C)\}$ part of the constraint, we need more condition than just having $\neg X_{AC} \wedge \neg X_{BC}$ due to how the variable is defined. According to our definition, $\neg X_{AB}$ will be equivalent to $A \leq B$ so the equality must be eliminated or it will create a cyclic ordering such as $A \leq B \leq C \leq A$ which is true when $A = B = C$. In order to resolve this problem, we enforced a constraint that will eliminate all cyclic ordering possible. For every permutations with 3 variables, say A, B, C we imposed the following:

$$\neg(X_{AB} \wedge X_{BC} \wedge \neg X_{AC}) \equiv (\neg X_{AB} \vee \neg X_{BC} \vee X_{AC}) \quad \ldots \quad (1)$$
$$\neg(\neg X_{AB} \wedge \neg X_{BC} \wedge X_{AC}) \equiv (X_{AB} \vee X_{BC} \vee \neg X_{AC}) \quad \ldots \quad (2)$$

The constraint (1) says $A < B < C$ *and* $A \geq C$ is not possible (2) says $A \geq B \geq C$ *and* $A < C$ is not possible. Therefore, when we enforce these two constraints to all possible permutations we can indeed eliminate all cyclic orderings. Now we only need to transform the constraint $(X_{AC} \wedge X_{AB}) \vee (\neg X_{AC} \wedge \neg X_{AB})$ into CNF format.

$$\begin{aligned}
&(X_{AC} \wedge X_{BC}) \vee (\neg X_{AC} \wedge \neg X_{BC}) \\
&\equiv \{X_{AC} \vee (\neg X_{AC} \wedge \neg X_{BC})\} \wedge \{X_{BC} \vee (\neg X_{AC} \wedge \neg X_{BC})\} \\
&\equiv (X_{AC} \vee \neg X_{BC}) \wedge (X_{BC} \vee \neg X_{AC}) \quad \ldots \quad (3)
\end{aligned}$$

Finally, we now are able to describe explicit conditions that are equivalent to wizard orderings in CNF form:

$$ABC \equiv (X_{AC} \vee \neg X_{BC}) \wedge (X_{BC} \vee \neg X_{AC}) \wedge (\neg X_{AB} \vee \neg X_{BC} \vee X_{AC}) \wedge (X_{AB} \vee X_{BC} \vee \neg X_{AC})$$

Therefore, the wizard ordering is correct if and only if we satisfy all the constraints we generated.

2. Proof

   Define functions f and h such that it reduces the Wizard Ordering problem into 3 SAT problem. The function f takes an instance of Wizard Ordering problem and transforms it into 3 SAT problem following the above scheme generating all required constraints. The function h will take the solution given by the 3 SAT solver and transform them into wizard orderings. The exact implementations will be described in the next section.

   (a) Proving f and h has Polynomial Runtime

   Let the number of wizard be n and number of constraints be m then the function f creates 2 implication clauses for all possible combination of 3 wizards and it creates 2 clauses for each ordering constraints given. Therefore, total clauses generated is $2\binom{n}{3} + 2m \in O(n^3 + m)$. The runtime for f is $O(n^3 + m)$ due to this reason. One thing to note is that in our problem instances, m is bounded by $10n$. When the 3 SAT solver returns solution for each variable, we construct a DAG using each wizard as a vertex and edge according to the ordering found. The construction of DAG will take at most $\binom{n}{2} \in O(n^2)$ time when every possible pair of wizard is considered. The function h topologically sorts the DAG found using DFS but this is still linear in the number of edges in our DAG which is $O(n^2)$. Therefore, h runs in $O(n^2)$ time. This proves f and h runs in polynomial time.

   (b) Proving f and h is Correct

   In the previous section we proved that the 3 SAT constraints generated and wizard ordering are completely equivalent conditions. Therefore, it is trivial that if the 3 SAT solver outputs a solution, there exist a corresponding solution to Wizard Ordering problem instance. Also, when there exist a solution for the Wizard Ordering problem instance, there must be a solution to corresponding 3 SAT problem because it's a contradiction if it did not. So taking the contraposition, if there exist no solution for the instance of 3 SAT problem, there exist no solution for Wizard Ordering problem also.

3. Details on Preprocessing and Postprocessing

   (a) Preprocessing Methods

      i. **generateVariables()**
      Reads the input file corresponding to the file name stored in the instance of the solver, adding each unique name found in the constraints to a set of wizard names. Then, for each of the wizard names, generates a bi-directional mapping between the name and an unique wizard ID. For each pair of wizard IDs,represented by integers x and y, where $x < y$, we generate a bi-directional mapping between a variable ID and a list of wizard IDs, $[x, y]$. This mapping is necessary because the SAT solver uses the DIMACS CNF format, which means each literal in the SAT formula's clauses must be an integer.

      ii. **getVariableIDbyWizard(int first, int second)**
      This helper method appropriately returns the variable ID corresponding to the wizard IDs specified by $first$ and *second*.

      iii. **generateConstraintClauses()**
      A clause in our program is specified by an array of signed integers. This method performs a second pass over the input file and reads every constraint. For each of the constraints, we add the clauses specified by (3) to the list of clauses. For each literal inside a clause, we add the positive value of a variable ID if we want to enforce the variable to be true, or its negative value otherwise.

      iv. **generateImplicationClauses()**
      For each triplet of distinct wizard IDs, this method generates the implication clauses specified by (1) and (2) and adds them to the list of clauses.

   (b) Black Box Methods

      i. **addAllClauses()**
      This method adds the clauses generated in the pre-processing methods to the SAT solver.

      ii. **run()** This method runs the SAT solver, and stores the variable's assignments in a list, which will be post-processed.

(c) Postprocessing Methods

i. **generateWizardGraph()**

Constructs a Directed Acyclic Graph given the wizard assignments from the SAT solver. The blackboxed SAT solver outputs an integer array of variable IDs. Each variable ID is either positive or negative, which indicates the direction of the edge between the two wizards. The direction of edges are as follows: if $age(A) < age(B)$, $A \Rightarrow B$, and if $age(A) > age(B)$, $B \Rightarrow A$.

ii. **generateOrderedWizardPair(int assignment)**

Given an assignment from the SAT solver, this method returns the correct ordering of the pair of wizards, i.e. if the assignment $= -1$ with pair of wizards $[A, B]$, then it'll return $[B, A]$ since wizards' $age(A) > age(B)$, and vice versa for positive 1.

iii. **generateToplogicalOrdering(Digraph dag)**

Given a Directed Acyclic Graph, this method finds and returns the topologically sorted ordering of the vertices, or wizards, ordered from the source vertex to the sink vertex.

iv. **generateSolution(Topological topo)**

Given a Topological ordering of wizard IDs, generates an ordering of wizard names from the bi-directional map of wizard IDs to names. Such ordering of wizard names is the solution to the given input file.

v. **outputSolutionToFile()**

Writes the solution to the given input file to the corresponding output file.

C. Steps to Replicate Our Algorithm

1. Libraries Used

(a) SAT4J

We decided to pick this specific SAT solver because it is conveniently built to support the programming language we were most comfortable using, Java. Another factor that led to our decision to use SAT4J was the ease of creating the input and reading their output, which is standard among common SAT solvers.

(b) Google Guava

Google's Guava library is used solely for their implementation of HashBiMap, which enabled us to easily map IDs and wizards/variables without the need to keep track of multiple mappings. Mainly used to write cleaner code.

(c) Princeton's Algorithms

Princeton's Algorithms library was used in our post-processing methods to build a directed graph, and to enable topological sorting. Also, their Stopwatch was used to measure our running time. Mainly used to reduce to amount of code we needed to write for post-processing the SAT solver's results.

2. How to run our code?

   (a) Obtain the source code either from Gradescope or by cloning from **this GitHub repository.**

   (b) Open the project using IntelliJ IDEA.

      i. Start IntelliJ IDEA.

      ii. Click "Open".

      iii. Select the root directory of this project. Follow directions in screen.

   (c) Ensure all dependencies are correctly set up.

      i. Go to File > Project Structure > Project Settings > Libraries.

      ii. Make sure "guava", "javalib", and "sat4j" have been added. If any library is missing, add them by clicking the $'+'$ sign, followed by $Java$, and select the corresponding folder inside the "./lib/*" sub-directory. There is no need to build the libraries used in this project. The corresponding .jar files are already included.

   (d) To run the program on all the files we've solved so far, simply run the main method located in "./src/main/java/com/wizardordering/WizardOrdering.java". The expected running time of the largest file we've solved, "staff_180.in", is around 100 minutes, thus expect the program to take up to two hours to complete all the files up to the aforementioned one. If you need to run a particular staff input, please follow comments in the main method.

   (e) The Phase II input files are located in "./src/resources/phase2_inputs/inputsXX/" where XX = 20, 35 or 50 depending on the number of wizards. Their corresponding output files are located in "./src/resources/phase2_outputs/".

   (f) The staff input files are located in "./src/resources/Staff_Inputs/". The corresponding outputs are located in "./src/resources/Staff_outputs/".

```java
package main.java.com.wizardordering;

import edu.princeton.cs.algs4.Stopwatch;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;


public class WizardOrdering {
    private static final String INPUT_FILES_PATH = "./src/main/resources/phase2_inputs/";
    private static final String STAFF_INPUT_PATH = "./src/main/resources/Staff_Inputs/";
    private static final String INPUT_FILE_EXTENSION = ".in";
    private static List<WizardOrderingSolver> solvers = new ArrayList<>();
    private static List<WizardOrderingSolver> staffInputSolvers =  new ArrayList<>();

    /**
     * Runs the WizardOrderingSolver instances for Phase II input files.
     */
    private static void solveAll() {
        Stopwatch watch = new Stopwatch();
        for (WizardOrderingSolver solver: solvers) {
            solver.preProcess();
            solver.run();
            solver.postProcess();
        }
        System.out.println("All files solved in: " + watch.elapsedTime() + " s\n");
    }

    /**
     * Runs the WizardOrderingSolver instances for staff input files.
     */
    private static void solveAllStaff() {
        for (WizardOrderingSolver solver: staffInputSolvers) {
            //solver.randomAssign();
            solver.preProcess();
            solver.run();
            solver.postProcess();
        }
    }


    /**
     * Only used to solve one of the staff files at a time.
     * @param filename
     */
    private static void solveStaff(String filename) {
        for (WizardOrderingSolver solver: staffInputSolvers) {
            if (solver.getFileName().equals(filename)) {
                solver.preProcess();
                solver.run();
                solver.postProcess();
            }
        }
    }
    /**
     * Initializes WizardOrderingSolver instances for each of the assigned Phase II input files.
     */
    private static void start() {
        File inputFilesFolder = new File(INPUT_FILES_PATH);
        File[] listOfSubFolders = inputFilesFolder.listFiles();
        for (File subfolder : listOfSubFolders) {
            File[] listOfInputFiles = subfolder.listFiles();
            Arrays.sort(listOfInputFiles);
            for (File inputFile : listOfInputFiles) {
                String fileName = inputFile.getName();
                if (inputFile.isFile() && fileName.endsWith(INPUT_FILE_EXTENSION)) {
                    solvers.add(new WizardOrderingSolver(inputFile));
                }
            }
        }
    }

    /**
     * Initializes WizardOrderingSolver instances for each of the staff input files.
     */
    private static void startStaffFiles() {
        File inputFilesFolder = new File(STAFF_INPUT_PATH);
```

```java
        File[] listOfFiles = inputFilesFolder.listFiles();
        for (File inputFile : listOfFiles) {
            String fileName = inputFile.getName();
            if (inputFile.isFile() && fileName.endsWith(INPUT_FILE_EXTENSION)
                    && filterStaffInput(inputFile)) {
                staffInputSolvers.add(new WizardOrderingSolver(inputFile));
            }
        }
    }

    /**
     * Determines if inputFile is one of the files that can be currently solved.
     * @param inputFile
     * @return true if the number of wizards is 180 or fewer and false otherwise.
     */
    private static boolean filterStaffInput(File inputFile) {
        if (inputFile.getName().length() == 11) return true; // staff_60.in / staff_80.in

        int firstNumber = Character.getNumericValue(inputFile.getName().charAt(6));
        return firstNumber < 2;
    }

    /**
     * Main driver. Just run this.
     * @param args
     */
    public static void main (String[] args) {
        start();
        solveAll();
        startStaffFiles();
        solveAllStaff(); // Please comment out this line and uncomment next line if want to run a single staff file.
        //solveStaff("staff_80.in");  //Replace XXX with the number in the staff file if only want to run a single staff file.

    }
}
```

```java
package main.java.com.wizardordering;

import com.google.common.collect.BiMap;
import com.google.common.collect.HashBiMap;
import edu.princeton.cs.algs4.Stopwatch;
import edu.princeton.cs.algs4.Digraph;
import edu.princeton.cs.algs4.Topological;
import org.junit.Assert;
import org.sat4j.core.VecInt;
import org.sat4j.minisat.SolverFactory;
import org.sat4j.specs.ContradictionException;
import org.sat4j.specs.IProblem;
import org.sat4j.specs.ISolver;
import org.sat4j.specs.TimeoutException;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;

public class WizardOrderingSolver {
    // Bi-directional mapping between a wizard ID (int) and a wizard name (String).
    private BiMap<Integer, String> wizIdToName;
    // Bi-directional mapping between a variable ID (int) and pair of Wizard IDs.
    private BiMap<Integer, List<Integer>> varIdToWizID;
    private ISolver solver;
    private File inputFile;
    private Set<String> wizardSet;
    private List<int[]> clausesByConstraints;
    private List<int[]> clausesByImplication;
    private int[] assignments;
    private String[] solution;

    // For statistics
    private int numConstraints;
    private double elapsedTime;

    /**
     * Constructor
     * @param inputFile, the file from where the constraints will be read.
     */
    public WizardOrderingSolver(File inputFile) {
        this.solver = SolverFactory.newDefault();
        this.inputFile = inputFile;
        this.wizardSet = new HashSet<>();
        this.clausesByConstraints = new ArrayList<>();
        this.clausesByImplication = new ArrayList<>();
        this.numConstraints = 0;
    }

    /**
     *  Returns file name.
     * @return filename
     */
    public String getFileName() {
        return this.inputFile.getName();
    }

    /**
     * preProcess
```

```java
     * Reduces the wizard ordering problem to an instance of 3-SAT.
     */
    public void preProcess() {
        this.generateVariables();
        this.generateConstraintClauses();
        this.generateImplicationClauses();
    }

    /**
     * Given two wizard IDs, return the corresponding variableID.
     * @param first, wizardID
     * @param second, wizardID
     * @return variableID(min(first, second), max(first,second)) * k
     *         where k = (first < second ? 1 : -1)
     */
    private int getVariableIDbyWizard(int first, int second) {
        int varID = 0;
        List<Integer> key = new ArrayList<>();

        if (first < second) {
            key.add(first);
            key.add(second);
            varID = varIdToWizID.inverse().get(key);
        } else {
            key.add(second);
            key.add(first);
            varID = varIdToWizID.inverse().get(key) * -1;
        }
        return varID;
    }

    /**
     * Creates the set of wizard names mapped to a wizardID
     * Generates variables for each pair of wizard IDs, A and B, where A < B.
     * The variableID corresponding to (B, A) is denoted as -1 * variableID
     * corresponding to (A, B).
     * Moreover, +(A, B) == -(B, A) <=> age(A) < age(B)
     *           -(A, B) == +(B, A) <=> age(A) > age(B)
     */
    private void generateVariables() {
        try {
            BufferedReader buf = new BufferedReader(new FileReader(this.inputFile));

            // Read number of wizards and constraints respectively.
            int numWizards = Integer.parseInt(buf.readLine().trim());
            this.numConstraints = Integer.parseInt(buf.readLine().trim());


            for (int i = 0; i < numConstraints; i++) {
                StringTokenizer st = new StringTokenizer(buf.readLine());
                while (st.hasMoreTokens()) {
                    wizardSet.add(st.nextToken());
                    if (wizardSet.size() == numWizards)
                        break;
                }
            }

            // Create mapping between wizard names and integer value.
            int wizId = 1;
            this.wizIdToName = HashBiMap.create(numWizards);
            for (String name : wizardSet) {
                this.wizIdToName.put(wizId++, name);
            }

```

```java
            // Create mapping between variables and integer value.
            int varId = 1;
            //this.varIdToVar = HashBiMap.create(this.numWizards * (this.numWizards - 1) / 2);
            this.varIdToWizID = HashBiMap.create(numWizards * (numWizards - 1) / 2);
            for (int i = 1; i <= numWizards; i++) {
                for (int j = i + 1; j <= numWizards; j++) {
                    List<Integer> lst = new ArrayList<>();
                    lst.add(i);
                    lst.add(j);
                    this.varIdToWizID.put(varId++, lst);
                }
            }

            buf.close();
        } catch (IOException e) {
            System.out.println("Error reading wizard names.");
            System.exit(-1);
        }
    }

    /**
     * generateConstraintClauses
     * For each constraint read from the input file:
     *      Let a, b, c be the wizardIDs corresponding to the three names read, respectively.
     */
    private void generateConstraintClauses() {
        try {
            BufferedReader buf = new BufferedReader(new FileReader(this.inputFile));

            // Skip to start of constraints.
            buf.readLine();
            buf.readLine();

            for (int i = 0; i < numConstraints; i++) {
                StringTokenizer st = new StringTokenizer(buf.readLine());

                int a = this.wizIdToName.inverse().get(st.nextToken());
                int b = this.wizIdToName.inverse().get(st.nextToken());
                int c = this.wizIdToName.inverse().get(st.nextToken());


                // Ensures all three wizard names are unique.
                Set<Integer> uniqueValues = new HashSet<>();
                uniqueValues.add(a);
                uniqueValues.add(b);
                uniqueValues.add(c);

                if (uniqueValues.size() != 3) {
                    continue;
                }

                int a_c = getVariableIDbyWizard(a, c);
                int b_c = getVariableIDbyWizard(b, c);
                this.clausesByConstraints.add(new int[]{a_c, -1 * b_c});
                this.clausesByConstraints.add(new int[]{-1 * a_c, b_c});
            }
            Assert.assertEquals(this.numConstraints * 2, this.clausesByConstraints.size());
        } catch (IOException e) {
            System.out.println("Error reading constraints.");
            System.exit(-1);
        }
    }
    /**
     * For each trio of wizard IDs (i, j, k), there exists 3! = 6 valid orderings between them.
```

```java
     * They are as follows:
     *      Variable(i, j) ^      Variable(j, k) =>      Variable(i, k)     Equation (1)
     * -1 * Variable(i, j) ^ -1 * Variable(j, k) => -1 * Variable(i, k)     Equation (2)
     *      Variable(i, j) ^ -1 * Variable(i, k) => -1 * Variable(j, k)     Equation (3)
     * -1 * Variable(i, j) ^      Variable(i, k) =>      Variable(j, k)     Equation (4)
     * -1 * Variable(i, k) ^      Variable(j, k) => -1 * Variable(i, j)     Equation (5)
     *      Variable(i, k) ^ -1 * Variable(j, k) =>      Variable(i, j)     Equation (6)
     * By boolean algebra, equations (1), (3), and (5) are equivalent to the CNF clause:
     * -1 * Variable(i, j) v -1 * Variable(j, k) v       Variable(i, k)     Equation (7)
     * Similarly, equations (2), (4), and (6) are equivalent to the CNF clause:
     *      Variable(i, j) v      Variable(j, k) v  -1 * Variable(i, k)     Equation (8)
     *
     * Total of (N choose 3) * 2 = N * (N - 1) * (N - 2) / 3 clauses added.
     */
    private void generateImplicationClauses() {
        int numWizards = wizardSet.size();
        for (int i = 1; i <= numWizards; i++) {
            for (int j = i + 1; j <= numWizards; j++) {
                for (int k = j + 1; k <= numWizards; k++) {
                    int i_j = getVariableIDbyWizard(i, j);
                    int j_k = getVariableIDbyWizard(j, k);
                    int i_k = getVariableIDbyWizard(i, k);

                    this.clausesByImplication.add(new int[]{-1 * i_j, -1 * j_k, i_k});
                    this.clausesByImplication.add(new int[]{i_j, j_k, -1 * i_k});
                }
            }
        }
    }

    /**
     * Adds all clauses to the solver.
     */
    private void addAllClauses() {
        try {
            for (int[] clause : this.clausesByConstraints)
                this.solver.addClause(new VecInt(clause));
            for (int[] clause : this.clausesByImplication)
                this.solver.addClause(new VecInt(clause));
        } catch (ContradictionException e) {
            System.out.println("ERROR: Clauses contain contradiction!");
            System.exit(-2);
        }
    }

    /**
     * Runs the solver.
     */
    public void run() {
        try {
            Stopwatch watch = new Stopwatch();
            System.out.println("Now running " + this.getFileName());
            addAllClauses();

            IProblem problem = solver;

            if (problem.isSatisfiable()) {
                this.elapsedTime = watch.elapsedTime();
                this.assignments = problem.model();
                System.out.println("File " + this.getFileName() + " has been solved!!");
            }
        } catch (TimeoutException e) {
            System.out.println("Error: Timeout exception.");
            System.exit(-3);
```

```java
        }
    }

    /**
     * PostProcessing method.
     * Builds a Directed Acyclic Graph and topologically sorts the wizards.
     */
    public void postProcess() {
        Digraph dag = this.generateWizardGraph();
        Topological topo = this.generateToplogicalOrdering(dag);
        this.generateSolution(topo);
        this.printStatistics();
        this.outputSolutionToFile();
    }

    public void printAssignments() {
        if (getFileName().equals("input5.in")) {
            for (int i = 0; i < this.assignments.length; i++) {
                System.out.print(this.assignments[i] + " ");
            }
            System.out.print("\n");
            System.out.println(Arrays.toString(this.solution));
        }
    }

    /**
     * Constructs a Directed Acyclic Graph given the wizard assignments.
     * Following variant holds: +(A, B) == -(B, A) <=> age(A) < age(B)
     *                          -(A, B) == +(B, A) <=> age(A) > age(B)
     * Direction of edges: if age(A) < age(B), A -> B and vice-versa.
     * @return Digraph
     */
    private Digraph generateWizardGraph() {
        Digraph dag = new Digraph(this.wizardSet.size());

        for (int assignment : this.assignments) {
            int[] orderedWizardPair = generateOrderedWizardPair(assignment);
            dag.addEdge(orderedWizardPair[0] - 1, orderedWizardPair[1] - 1);
        }

        return dag;
    }

    /**
     * Given an assignment, returns the correct ordering of wizards,
     * i.e. if assignment = -1 with pair [A, B], then return [B, A] since age(A) > age(B).
     * @param assignment varID
     * @return int[] ordered pairing of wizards
     */
    private int[] generateOrderedWizardPair(int assignment) {
        int[] result = new int[2];
        List<Integer> pair;
        int i = 0, j = 1;

        if (assignment > 0) {
            pair = this.varIdToWizID.get(assignment);
        } else {
            pair = this.varIdToWizID.get(assignment * -1);
            i = 1;
            j = 0;
        }

        result[i] = pair.get(0);
        result[j] = pair.get(1);
```

```java
        return result;
    }

    /**
     * Given a Directed Acyclic Graph, returns a Topological ordering.
     * @param dag DAG
     * @return Topological
     */
    private Topological generateToplogicalOrdering(Digraph dag) {
        return new Topological(dag);
    }

    /**
     * Given a Topological ordering, generates an ordering of wizard names.
     * @param topo topological ordering
     */
    private void generateSolution(Topological topo) {
        if (!topo.hasOrder()) return;

        this.solution = new String[this.wizardSet.size()];
        Iterator<Integer> iter = topo.order().iterator();
        int i = 0;

        while (iter.hasNext()) {
            String wizard = this.wizIdToName.get(iter.next() + 1);
            this.solution[i] = wizard;
            i++;
        }
    }

    /**
     * Return the assignment established by adding wizard names to a set.
     * Not used.
     */
    public void randomAssign() {
        try {
            BufferedReader buf = new BufferedReader(new FileReader(this.inputFile));

            // Read number of wizards and constraints respectively.
            int numWizards = Integer.parseInt(buf.readLine().trim());
            this.numConstraints = Integer.parseInt(buf.readLine().trim());


            for (int i = 0; i < numConstraints; i++) {
                StringTokenizer st = new StringTokenizer(buf.readLine());
                while (st.hasMoreTokens()) {
                    wizardSet.add(st.nextToken());
                    if (wizardSet.size() == numWizards)
                        break;
                }
            }
            this.solution = new String[wizardSet.size()];
            int i = 0;
            for (String name: wizardSet) {
                this.solution[i++] = name;
            }

            this.outputSolutionToFile();
        } catch (IOException e) {
            System.exit(-2);
        }
    }

    /**
```

```java
     * Prints useful statistics.
     */
    public void printStatistics() {
        System.out.println("******** STATISTICS **********");
        System.out.println("File name: " + this.inputFile.getName());
        System.out.println("Number of wizards: " + this.wizardSet.size());
        System.out.println("Number of variables: " + this.varIdToWizID.size());
        System.out.println("Number of clauses by constraints: " + this.clausesByConstraints.size());
        System.out.println("Number of clauses by implication: " + this.clausesByImplication.size());
        printAssignments();
        System.out.println("Running time: " + this.elapsedTime + "s\n");
    }

    /**
     * Outputs solution to file.
     */
    public void outputSolutionToFile() {
        FileWriter fileWriter = null;
        BufferedWriter bufferedWriter = null;
        StringBuilder sbFile = new StringBuilder("./src/main/resources/");

        if (this.inputFile.getName().substring(0, 5).equals("staff")) {
            sbFile.append("Staff_outputs/");
        } else {
            sbFile.append("phase2_outputs/");
        }

        sbFile.append(removeExtension(this.inputFile.getName()).replaceAll("in", "out"));
        sbFile.append(".out");

        try {
            fileWriter = new FileWriter(sbFile.toString());
            bufferedWriter = new BufferedWriter(fileWriter);

            StringBuilder sb = new StringBuilder();

            for (String wizard : this.solution) {
                sb.append(wizard);
                sb.append(" ");
            }

            bufferedWriter.write(sb.toString().trim());
        } catch (IOException e) {
            System.out.println("Error writing wizard names.");
            System.exit(-1);
        } finally {
            try {
                if (bufferedWriter != null) bufferedWriter.close();
                if (fileWriter != null) fileWriter.close();
            } catch (IOException ex) {
                System.out.println("Error closing writers.");
                System.exit(-1);
            }
        }
    }

    /**
     * Removes the extension from a given filename.
     * @param filename filename
     * @return String
     */
    private String removeExtension(String filename) {
        if (filename == null) return null;
        int dotIndex = filename.lastIndexOf(".");
```

```
        if (dotIndex == -1) return filename;
        return filename.substring(0, dotIndex);
    }
}
```