Software report

Buciu Ștefan, ICA 256

1. Sudoku

* Problem definition

Sudoku is a puzzle where a 9x9 grid is given and the goal is to fill it with values from 1 to 9 such that there are no repeated values on rows, columns, and inside the 3x3 squares.

* CSP representation

We can represent the Sudoku problem as follows:

* Variable: every cell will have a variable assigned, thus where
* Domain: every variable can have values between 1 and 9, thus
* Constraints: the constraints are imposed by the puzzle rules
  + No value can be repeated on a row
  + No value can be repeated on a column
  + No value can be repeated in a 3x3 square
* Solution
* Backtracking:

Given a board, we recursively try all possible values for each empty cell such that no constraint is violated.

This method is implemented in the class *SudokuBacktracking* in the Java source code provided.

* Problem Reduction (node consistency):

Given a board, precompute the domains for each variable and we consider values only from the computed domains. For example, if a board is given with value at position , we then remove the value from all domains of the variables on the same row, column, and in the same 3x3 square. With this approach, when a value is chosen for an empty cell, it will not waste resources on computing a solution using the value .

This method is implemented in the class *SudokuPRNodeConsistency* in the Java source code provided.

* Basic Search Strategy (forward check):

This approach follows a similar idea with Node Consistency, by updating the domains once a variable is committed to. Thus, whenever we start committing a new value, the domain will be up to date with all the previous choices made.

This method is implemented in the class *SudokuSSForwardCheck* in the Java source code provided.

* Analysis

We performed tests on the puzzles provided on [this website](https://sandiway.arizona.edu/sudoku/examples.html).

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Simple** | **Medium** | **Hard** |
| Backtracking | 9 ms | 250 ms | 80582 ms |
| Node consistency | 9 ms | 267 ms | 96480 ms |
| Forward check | 13 ms | 293 ms | 98248 ms |

We can observe that the running time is similar for the proposed variations. However, in the case of the forward check approach, it is usually slower than the other two. An explanation for this would be the way in which we represent the domains, needing a clone for each value committed. A more efficient representation, such as a bitmask could help speed up this approach.

1. Maximum Cut

* Problem definition

The Maximum Cut problem is NP-complete, and it aims to partition an undirected graph into two sets such that the number of edges that cross the cut is maximized (i.e., one end of the edge in each set). We will be trying to solve this problem version, however there is a more general version where the edges have a positive weight assigned.

* CSP representation

We can represent the MaxCut problem as follows:

* Variable: given a graph with a set of vertices , a variable will be assigned for each vertex.
* Domain: after a cut, a vertex can be part of a set , thus the domain for each variable is .
* Constraints: after a cut, a set of edges will be selected . Given a selected edge , then the vertices connected by it and must have variables assigned such that .
* Solution
* Backtracking:

We solve MaxCut with backtracking by iterating over the given edges and keeping track of the partitioning. If we select edge , then each vertex is added to a set and . However, since the graph is undirected, we check both configurations (i.e., adding to S and to T).

A solution is correct if the constraints are satisfied (i.e., ). If we have a solution, we then compute the size of the cut by counting how many edges have the first vertex in a set, and the second in the other set.

This method is implemented in the class *MaxCutBacktracking* in the Java source code provided.

* Problem Reduction:

1. Pruning

To help reduce the problem space, we can use a pruning approach where, given an intermediate cut such that , and an edge then .

This method is implemented in the class *MaxCutPRPruning* in the Java source code provided.

1. Arc consistency

Given a set of chosen edges, we build an agenda with both edge directions (i.e., ). Initially, all vertices can be part of either set. Until the agenda is empty, we select an edge (i.e., constraint) and check if the current domains of are violated. If the domain changes, we then add all arcs that have the vertex in question on the right side.

After that, we then check if the configuration is valid and try to update the max cut.

This method is implemented in the class *MaxCutPRArcConsistent* in the Java source code provided.

* Basic Search Strategy (forward check):

Initially, all edges are considered. During backtracking, when we commit to an edge, when then filter out all edges that would violate the committed edge. The resulting filtered collection is then passed to the next backtracking iteration.

This method is implemented in the class *MaxCutSSForwardCheck* in the Java source code provided.

* Analysis

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **G2 (8 nodes, 14 edges)** | **G3 (6 nodes, 9 edges)** |
| Backtracking | 26349 ms | 124 ms |
| Pruning | 5728 ms | 43 ms |
| Arc consistency | 219 ms | 18 ms |
| Forward check | 1016667 ms | 115 ms |

From the tests performed, the *arc consistency* variant is the fastest. However, as graphs grow larger, all algorithms have a long running time. This is due to the complexity of the problem and the backtracking approach. The *forward check* variant is the slowest and that’s because on every backtrack iteration, it must update the domain of possible edges.

**Source code**

1. Sudoku

package sbuciu.sudoku.model;  
  
import java.io.File;  
import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.\*;  
  
public class Board {  
 public static final short *EMPTY* = 0;  
 public static final short *N* = 9;  
 public static final short *M* = *N* / 3;  
  
 private final short[][] board = new short[*N*][*N*];  
  
 public Board(short[][] board) {  
 if (board.length != *N* || board[0].length != *N*) {  
 throw new IllegalArgumentException(String.*format*("The board does not represent a %dx%d sudoku board", *N*, *N*));  
 }  
  
 initBoard();  
  
 for (short r = 0; r < *N*; r += 1) {  
 for (short c = 0; c < *N*; c += 1) {  
 if (board[r][c] == *EMPTY*) {  
 continue;  
 }  
  
 // check value range  
 if (board[r][c] < 1 && board[r][c] > 9) {  
 throw new IllegalArgumentException("Illegal value found on board");  
 }  
  
 // check if we can commit the value  
 if (!commit(new Pos(r, c), board[r][c])) {  
 throw new IllegalArgumentException("Illegal value found on board");  
 }  
 }  
 }  
  
 }  
  
 public static Board read(String path) {  
 if (!new File(path).exists()) {  
 throw new IllegalArgumentException(String.*format*("Path %s does not exist.", path));  
 }  
  
 List<String> lines;  
 try {  
 lines = Files.*readAllLines*(Paths.*get*(path));  
 } catch (IOException e) {  
 throw new IllegalArgumentException(String.*format*("Could not read content at path %s.", path));  
 }  
  
 if (lines.size() != *N*) {  
 throw new IllegalArgumentException(String.*format*("Content at path %s needs to be %d rows long and represent a sudoku board.", path, *N*));  
 }  
  
 final short[][] board = new short[*N*][*N*];  
 for (short row = 0; row < *N*; row += 1) {  
 final String[] cells = lines.get(row).split(",");  
 if (cells.length != *N*) {  
 throw new IllegalArgumentException(String.*format*("Content at path %s needs to be %d columns long and represent a sudoku board.", path, *N*));  
 }  
  
 for (short col = 0; col < *N*; col += 1) {  
 if (cells[col].isEmpty()) {  
 board[row][col] = *EMPTY*;  
 continue;  
 }  
  
 board[row][col] = Short.*parseShort*(cells[col]);  
 }  
 }  
  
 return new Board(board);  
 }  
  
 public String encode() {  
 final StringBuilder bob = new StringBuilder();  
 for (int i = 0; i < *N*; i += 1) {  
 for (int j = 0; j < *N*; j += 1) {  
 bob.append(board[i][j]);  
 if (j < *N* - 1) {  
 bob.append(',');  
 }  
 }  
  
 bob.append(System.*lineSeparator*());  
 }  
  
 return bob.toString();  
 }  
  
 */\*\*  
 \* Fill board with static value EMPTY  
 \*/* private void initBoard() {  
 for (int i = 0; i < *N*; i += 1) {  
 Arrays.*fill*(board[i], *EMPTY*);  
 }  
 }  
  
 */\*\*  
 \* Validate pos and return the value at that cell. If the pos is invalid, Empty is returned.  
 \*  
 \** ***@param*** *pos - pos of the cell  
 \** ***@return*** *the value in the cell  
 \*/* public short at(final Pos pos) {  
 if (pos.noPos || pos.r < 0 || pos.r >= *N* || pos.c < 0 || pos.c >= *N*) {  
 return *EMPTY*;  
 }  
  
 return board[pos.r][pos.c];  
 }  
  
 */\*\*  
 \* Find the top-leftmost empty cell  
 \*  
 \** ***@return*** *the top-leftmost empty cell  
 \*/* public Pos findEmptyPos() {  
 for (short r = 0; r < *N*; r += 1) {  
 for (short c = 0; c < *N*; c += 1) {  
 if (board[r][c] == *EMPTY*) {  
 return new Pos(r, c);  
 }  
 }  
 }  
  
 return Pos.*noPos*();  
 }  
  
 */\*\*  
 \* Given a position and a value, check if any of the row-col-square constraints is violated.  
 \* If no constraint is violated, then board is updated.  
 \*  
 \** ***@param*** *pos - the position where we need to set the value  
 \** ***@param*** *value - the value to set  
 \** ***@return*** *true if no constraint is violated, false otherwise.  
 \*/* public boolean commit(final Pos pos, short value) {  
 // Check if position is valid  
 if (pos.noPos || at(pos) != *EMPTY*) {  
 return false;  
 }  
  
 // Check if valid row-wise and col-wise  
 for (int i = 0; i < *N*; i += 1) {  
 if (board[pos.r][i] == value || board[i][pos.c] == value) {  
 return false;  
 }  
 }  
  
 // Check if valid in the correct (NHxNH) square  
 final int sqr = pos.r / *M*, sqc = pos.c / *M*;  
 for (int r = sqr \* *M*; r < (sqr + 1) \* *M*; r += 1) {  
 for (int c = sqc \* *M*; c < (sqc + 1) \* *M*; c += 1) {  
 if (board[r][c] == value) {  
 return false;  
 }  
 }  
 }  
  
 // Commit  
 board[pos.r][pos.c] = value;  
 return true;  
 }  
  
 */\*\*  
 \* Clear the cell at the given position.  
 \*  
 \** ***@param*** *pos - position of the cell to clear  
 \** ***@return*** *false if the position is invalid, false otherwise.  
 \*/* public boolean clear(final Pos pos) {  
 if (pos.noPos) {  
 return false;  
 }  
  
 board[pos.r][pos.c] = *EMPTY*;  
 return true;  
 }  
  
 public short[][] getBoard() {  
 return board;  
 }  
}

package sbuciu.sudoku.model;  
  
import java.util.Objects;  
  
public class Pos {  
 public final short r;  
 public final short c;  
 public final boolean noPos;  
  
 public Pos(short r, short c) {  
 this.r = r;  
 this.c = c;  
 noPos = false;  
 }  
  
 private Pos() {  
 r = c = -1;  
 noPos = true;  
 }  
  
 public static Pos noPos() {  
 return new Pos();  
 }  
  
 @Override  
 public String toString() {  
 return String.*format*("P(%d,%d)", r, c);  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Pos pos = (Pos) o;  
 return r == pos.r && c == pos.c;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(r, c);  
 }  
}

package sbuciu.sudoku.model;  
  
public class SudokuSolution {  
 public final short[][] board;  
 public final boolean isSolved;  
  
 public SudokuSolution(short[][] board, boolean isSolved) {  
 this.board = board;  
 this.isSolved = isSolved;  
 }  
  
 public String encode() {  
 final StringBuilder bob = new StringBuilder();  
 for (short[] shorts : board) {  
 for (int j = 0; j < board[0].length; j += 1) {  
 bob.append(shorts[j]);  
 if (j < board[0].length - 1) {  
 bob.append(',');  
 }  
 }  
  
 bob.append(System.*lineSeparator*());  
 }  
  
 return bob.toString();  
 }  
}

package sbuciu.sudoku;  
  
import sbuciu.sudoku.model.Board;  
import sbuciu.sudoku.model.Pos;  
import sbuciu.sudoku.model.SudokuSolution;  
  
import java.util.Arrays;  
  
public abstract class Sudoku {  
 */\*\*  
 \* Represents the given Sudoku board that we need to solve.  
 \*/* protected final Board board;  
  
 private long startTime;  
  
 public Sudoku(Board board) {  
 this.board = board;  
 }  
  
 protected abstract boolean internalSolve();  
  
 private long elapsed() {  
 return (System.*nanoTime*() - startTime) / 1\_000\_000;  
 }  
  
 public SudokuSolution solve() {  
 startTime = System.*nanoTime*();  
 boolean isSolved = internalSolve();  
 System.*out*.printf("elapsed %d ms%n", elapsed());  
  
 return new SudokuSolution(board.getBoard(), isSolved);  
 }  
  
 protected void l(final String s) {  
 System.*out*.println("elapsed " + elapsed() + "ms: " + s);  
 }  
  
 protected void lbt(final int depth, final Pos pos, final short[] row) {  
 final StringBuilder bob = new StringBuilder();  
 for (int i = 0; i < depth; i += 1) {  
 bob.append(" ");  
 }  
  
 bob.append("depth ").append(depth + 1);  
 bob.append(" ").append(pos);  
 bob.append(" ").append(Arrays.*toString*(row));  
  
 l(bob.toString());  
 }  
}

package sbuciu.sudoku;  
  
import sbuciu.sudoku.model.Board;  
import sbuciu.sudoku.model.Pos;  
  
*/\*\*  
 \* Class that solves the Sudoku problem formulated as a CSP by using backtracking.  
 \*/*public class SudokuBacktrack extends Sudoku {  
 public SudokuBacktrack(Board board) {  
 super(board);  
 }  
  
 private boolean backtrack(final int depth) {  
 final Pos pos = board.findEmptyPos();  
 if (pos.noPos) {  
 // All cells are filled, a solution is found  
 return true;  
 }  
  
 // We iterate all 9 values  
 for (short value = 1; value <= Board.*N*; value += 1) {  
 // Check if the value can be used for that cell  
 if (board.commit(pos, value)) {  
 lbt(depth, pos, board.getBoard()[pos.r]);  
 if (backtrack(depth + 1)) {  
 return true;  
 }  
  
 // No solution found for the selected value, we need to reset the cell.  
 board.clear(pos);  
 }  
 }  
  
 return false;  
 }  
  
 @Override  
 protected boolean internalSolve() {  
 return backtrack(0);  
 }  
}

package sbuciu.sudoku;  
  
import sbuciu.sudoku.model.Board;  
import sbuciu.sudoku.model.Pos;  
  
import java.util.ArrayList;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Set;  
  
import static sbuciu.sudoku.model.Board.\*;  
  
*/\*\*  
 \* Class that solves the Sudoku problem formulated as a CSP by using node consistency.  
 \*/*public class SudokuPRNodeConsistency extends Sudoku {  
  
 */\*\*  
 \* Matrix used to represent the available domains for each variable.  
 \*/* private final List<List<Set<Short>>> constraints = new ArrayList<>();  
  
 public SudokuPRNodeConsistency(Board board) {  
 super(board);  
 initConstraints();  
 }  
  
 */\*\*  
 \* Compute the domains based on the initial board values.  
 \*/* private void initConstraints() {  
 final Set<Short> values = new HashSet<>();  
 for (short val = 1; val <= *N*; val += 1) {  
 values.add(val);  
 }  
  
 // Add full domain 1-9 for each variable  
 for (int r = 0; r < *N*; r += 1) {  
 final List<Set<Short>> row = new ArrayList<>(*N*);  
 for (int c = 0; c < *N*; c += 1) {  
 row.add(new HashSet<>(values));  
 }  
 constraints.add(row);  
 }  
  
 // Remove values based on the row-col-square constraints  
 for (int r = 0; r < *N*; r += 1) {  
 for (int c = 0; c < *N*; c += 1) {  
 final short value = board.getBoard()[r][c];  
 if (value == *EMPTY*) {  
 continue;  
 }  
  
 // Remove values row and column wise.  
 for (int i = 0; i < *N*; i += 1) {  
 constraints.get(r).get(i).remove(value);  
 constraints.get(i).get(c).remove(value);  
 }  
  
 // Remove values in the square  
 final int sqr = r / *M*, sqc = c / *M*;  
 for (int i = sqr \* *M*; i < (sqr + 1) \* *M*; i += 1) {  
 for (int j = sqc \* *M*; j < (sqc + 1) \* *M*; j += 1) {  
 constraints.get(i).get(j).remove(value);  
 }  
 }  
 }  
 }  
 }  
  
 private boolean backtrack(final int depth) {  
 final Pos pos = board.findEmptyPos();  
 if (pos.noPos) {  
 // All cells are filled, a solution is found  
 return true;  
 }  
  
 // We iterate only available domain  
 for (final short value : constraints.get(pos.r).get(pos.c)) {  
 // Check if the value can be used for that cell  
 if (board.commit(pos, value)) {  
 lbt(depth, pos, board.getBoard()[pos.r]);  
 if (backtrack(depth + 1)) {  
 return true;  
 }  
  
 // No solution found for the selected value, we need to reset the cell.  
 board.clear(pos);  
 }  
 }  
  
 return false;  
 }  
  
 @Override  
 protected boolean internalSolve() {  
 return backtrack(0);  
 }  
}

package sbuciu.sudoku;  
  
import sbuciu.sudoku.model.Board;  
import sbuciu.sudoku.model.Pos;  
  
import java.util.ArrayList;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Set;  
  
import static sbuciu.sudoku.model.Board.\*;  
import static sbuciu.sudoku.model.Board.*M*;  
  
*/\*\*  
 \* Class that solves the Sudoku problem formulated as a CSP by using forward checking.  
 \*/*public class SudokuSSForwardCheck extends Sudoku {  
  
 public SudokuSSForwardCheck(Board board) {  
 super(board);  
 initConstraints();  
 }  
  
 */\*\*  
 \* After a commit, remove the value based on the row-column-square constraints from the available domains.  
 \** ***@param*** *constraints - matrix of domains  
 \** ***@param*** *r - the row of the cell committed  
 \** ***@param*** *c - the row of the cell committed  
 \** ***@param*** *value - the value of the cell committed  
 \*/* private void commit(final List<List<Set<Short>>> constraints, final int r, final int c, final short value) {  
 // Remove values row and column wise.  
 for (int i = 0; i < *N*; i += 1) {  
 constraints.get(r).get(i).remove(value);  
 constraints.get(i).get(c).remove(value);  
 }  
  
 // Remove values in the square  
 final int sqr = r / *M*, sqc = c / *M*;  
 for (int i = sqr \* *M*; i < (sqr + 1) \* *M*; i += 1) {  
 for (int j = sqc \* *M*; j < (sqc + 1) \* *M*; j += 1) {  
 constraints.get(i).get(j).remove(value);  
 }  
 }  
 }  
  
  
 */\*\*  
 \* Compute the domains based on the initial board values.  
 \*/* private List<List<Set<Short>>> initConstraints() {  
 final List<List<Set<Short>>> constraints = new ArrayList<>(*N*);  
  
 final Set<Short> values = new HashSet<>();  
 for (short val = 1; val <= *N*; val += 1) {  
 values.add(val);  
 }  
  
 for (int r = 0; r < *N*; r += 1) {  
 final List<Set<Short>> row = new ArrayList<>(*N*);  
 for (int c = 0; c < *N*; c += 1) {  
 row.add(new HashSet<>(values));  
 }  
 constraints.add(row);  
 }  
  
 for (int r = 0; r < *N*; r += 1) {  
 for (int c = 0; c < *N*; c += 1) {  
 final short value = board.getBoard()[r][c];  
 if (value == *EMPTY*) {  
 continue;  
 }  
  
 commit(constraints, r, c, value);  
 }  
 }  
  
 return constraints;  
 }  
  
 */\*\*  
 \* Create a matrix clone of the available domains  
 \** ***@param*** *constraints - matrix of domains  
 \** ***@return*** *clone of constraints  
 \*/* private List<List<Set<Short>>> clone(final List<List<Set<Short>>> constraints) {  
 final List<List<Set<Short>>> cc = new ArrayList<>(*N*);  
 for (final List<Set<Short>> row : constraints) {  
 final List<Set<Short>> rowClone = new ArrayList<>(*N*);  
 for (final Set<Short> s : row) {  
 rowClone.add(new HashSet<>(s));  
 }  
  
 cc.add(rowClone);  
 }  
  
 return cc;  
 }  
  
 private boolean backtrack(final int depth, final List<List<Set<Short>>> constraints) {  
 final Pos pos = board.findEmptyPos();  
 if (pos.noPos) {  
 // All cells are filled, a solution is found  
 return true;  
 }  
  
 // We iterate only available domain  
 for (final short value : constraints.get(pos.r).get(pos.c)) {  
 // Check if the value can be used for that cell  
 if (board.commit(pos, value)) {  
 lbt(depth, pos, board.getBoard()[pos.r]);  
  
 // Create a clone and update the domain  
 final List<List<Set<Short>>> cConstraints = clone(constraints);  
 commit(cConstraints, pos.r, pos.c, value);  
  
 // Recursive call using the updated domain  
 if (backtrack(depth + 1, cConstraints)) {  
 return true;  
 }  
  
 // No solution found for the selected value, we need to reset the cell.  
 board.clear(pos);  
 }  
 }  
  
 return false;  
 }  
  
 @Override  
 protected boolean internalSolve() {  
 return backtrack(0, initConstraints());  
 }  
}

1. Maximum cut

package sbuciu.maxcut.model;  
  
import java.util.Objects;  
  
public class Edge {  
 private final int u;  
 private final int v;  
 private final int w;  
  
 public Edge(int u, int v, int w) {  
 this.u = u;  
 this.v = v;  
 this.w = w;  
 }  
  
 public Edge(int u, int v) {  
 this.u = u;  
 this.v = v;  
 this.w = 1;  
 }  
  
 public int u() {  
 return u;  
 }  
  
 public int v() {  
 return v;  
 }  
  
 public int w() {  
 return w;  
 }  
  
 public Edge reversed() {  
 return new Edge(v, u, w);  
 }  
  
 @Override  
 public String toString() {  
 return "Edge{" +  
 "u=" + u +  
 ", v=" + v +  
 ", w=" + w +  
 '}';  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Edge edge = (Edge) o;  
 return u == edge.u && v == edge.v && w == edge.w;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(u, v, w);  
 }  
}

package sbuciu.maxcut.model;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Objects;  
  
public class Graph {  
 private final int n;  
 private final boolean[][] adjacencyMatrix;  
 private final List<Edge> edges = new ArrayList<>();  
  
 public Graph(final int n) {  
 this.n = n;  
 adjacencyMatrix = new boolean[n][n];  
 }  
  
 public int getN() {  
 return n;  
 }  
  
 public void addEdge(final int u, final int v, final int w) {  
 edges.add(new Edge(u, v, w));  
 }  
  
 public void addEdge(final int u, final int v) {  
 edges.add(new Edge(u, v));  
 adjacencyMatrix[u][v] = adjacencyMatrix[v][u] = true;  
 }  
  
 public void addEdge(Edge edge) {  
 edges.add(edge);  
 }  
  
 public List<Edge> getEdges() {  
 return edges;  
 }  
  
 public boolean areConnected(final int u, final int v) {  
 return adjacencyMatrix[u][v] || adjacencyMatrix[v][u];  
 }  
  
 @Override  
 public String toString() {  
 final StringBuilder bob = new StringBuilder();  
 bob.append("Graph{n=").append(n).append(", edges=[").append(System.*lineSeparator*());  
  
 for (final Edge edge : edges) {  
 bob.append(edge.toString()).append(System.*lineSeparator*());  
 }  
  
 return bob.append("]}").toString();  
 }  
  
 @Override  
 public boolean equals(final Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Graph graph = (Graph) o;  
 return n == graph.n && edges.equals(graph.edges);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(n, edges);  
 }  
}

package sbuciu.maxcut.model;  
  
public class MaxCutSolution {  
 private final Graph graph;  
 private final int cost;  
 private final int[] partitions;  
  
 public MaxCutSolution(Graph graph, int cost, int[] partitions) {  
 this.graph = graph;  
 this.cost = cost;  
 this.partitions = partitions;  
 }  
  
 public Graph getGraph() {  
 return graph;  
 }  
  
 public int getCost() {  
 return cost;  
 }  
  
 public int[] getPartitions() {  
 return partitions;  
 }  
}

package sbuciu.maxcut.io;  
  
import sbuciu.maxcut.model.Graph;  
  
import java.io.File;  
import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.List;  
  
public class GraphIO {  
 public static Graph read(String path) throws IOException {  
 final File file = new File(path);  
 if (!file.exists()) {  
 throw new IllegalArgumentException("path does not exist.");  
 }  
  
 final List<String> lines = Files.*readAllLines*(Paths.*get*(path));  
 final int n = Integer.*parseInt*(lines.get(0));  
  
 final Graph graph = new Graph(n);  
 for (int i = 1; i < lines.size(); i += 1) {  
 final String[] edge = lines.get(i).split(" ");  
 final int u = Integer.*parseInt*(edge[0]) - 1,  
 v = Integer.*parseInt*(edge[1]) - 1;  
  
 graph.addEdge(u, v);  
 }  
  
 return graph;  
 }  
}

package sbuciu.maxcut;  
  
import sbuciu.maxcut.model.Edge;  
import sbuciu.maxcut.model.Graph;  
import sbuciu.maxcut.model.MaxCutSolution;  
  
import java.util.Arrays;  
import java.util.Collection;  
import java.util.HashSet;  
import java.util.Set;  
  
public abstract class MaxCut {  
 */\*\*  
 \* Represents the given graph for which we need to find the max cut.  
 \*/* protected final Graph graph;  
  
 */\*\*  
 \* Represents a solution to the max cut problem represented as a variable assignment array where a value of:  
 \* - -1 means that the vertex at that index is part of set S  
 \* - 1 means that the vertex at that index is part of set T  
 \* There can be a multiple cuts, however this variable will keep the assignment of a maximum cut.  
 \*/* protected int[] maxCutAssignment = new int[0];  
  
 */\*\*  
 \* Keeps track of the count of edges that are in the maximum cut.  
 \*/* protected int maxCut = 0;  
  
 private long startTime;  
  
 public MaxCut(Graph graph) {  
 this.graph = graph;  
 }  
  
 */\*\*  
 \* Checks if a value assignment is valid by verifying that no vertex is present in both sets.  
 \*  
 \** ***@return*** *true if intersection is an empty set, false otherwise.  
 \*/* protected boolean isValid(final Set<Integer> S, final Set<Integer> T) {  
 if (S.size() + T.size() != graph.getN()) {  
 return false;  
 }  
  
 final Set<Integer> intersection = new HashSet<>(S);  
 intersection.retainAll(T);  
  
 return intersection.isEmpty();  
 }  
  
 */\*\*  
 \* When we have a solution, we check if the cut is better than the maximum cut we keep track of. If it is, then we  
 \* update the cut.  
 \*/* protected void updateCut(final Set<Integer> S, final Set<Integer> T) {  
 int cut = 0;  
 for (final int u : S) {  
 for (final int v : T) {  
 // Count the number of edges that are cut.  
 if (graph.areConnected(u, v)) {  
 cut += 1;  
 }  
 }  
 }  
  
 // If this is a better cut, we update the global parameters.  
 if (cut > maxCut) {  
 maxCut = cut;  
 maxCutAssignment = new int[graph.getN()];  
 for (final int var : S) {  
 maxCutAssignment[var] = -1;  
 }  
  
 for (final int var : T) {  
 maxCutAssignment[var] = 1;  
 }  
  
 l(String.*format*("MaxCut cost=%s partition=%s", maxCut, Arrays.*toString*(maxCutAssignment)));  
 }  
 }  
  
 protected abstract void internalSolve();  
  
 */\*\*  
 \* Logs the current backtracking depth and the current solution built.  
 \*  
 \** ***@param*** *depth - the backtracking depth  
 \** ***@param*** *S - one set of the maxcut partition  
 \** ***@param*** *T - second set of the maxcut partition  
 \*/* protected void lbt(final int depth, final Set<Integer> S, final Set<Integer> T) {  
 final StringBuilder bob = new StringBuilder();  
 for (int i = 0; i < depth; i += 1) {  
 bob.append(" ");  
 }  
  
 bob.append(depth + 1);  
 bob.append(" S={");  
 int i = 0;  
 for (final int v : S) {  
 bob.append(v);  
 if (i < S.size() - 1) {  
 bob.append(",");  
 }  
  
 i += 1;  
 }  
  
 bob.append("} T={");  
 i = 0;  
 for (final int v : T) {  
 bob.append(v);  
 if (i < T.size() - 1) {  
 bob.append(",");  
 }  
  
 i += 1;  
 }  
  
 l(bob.append("}").toString());  
 }  
  
 */\*\*  
 \* Logs the number of constraints currently considered.  
 \*  
 \** ***@param*** *constraints - list of edges  
 \*/* protected void lcg(final Collection<Edge> constraints) {  
 l("constraints count " + constraints.size());  
 }  
  
 */\*\*  
 \* Logs a message along with the elapsed time since the algorithm started.  
 \*  
 \** ***@param*** *s - the message to be logged  
 \*/* protected void l(final String s) {  
 final long elapsed = (System.*nanoTime*() - startTime) / 1\_000\_000;  
 System.*out*.println("elapsed " + elapsed + "ms: " + s);  
 }  
  
 public MaxCutSolution solve() {  
 startTime = System.*nanoTime*();  
 internalSolve();  
 final long elapsed = (System.*nanoTime*() - startTime) / 1\_000\_000;  
 System.*out*.printf("elapsed %d ms%n", elapsed);  
  
 return new MaxCutSolution(graph, maxCut, maxCutAssignment);  
 }  
}

package sbuciu.maxcut;  
  
import sbuciu.maxcut.model.Edge;  
import sbuciu.maxcut.model.Graph;  
  
import java.util.HashSet;  
import java.util.Set;  
  
  
*/\*\*  
 \* Class that solves the MaxCut problem formulated as a CSP by using backtracking.  
 \*/*public class MaxCutBacktrack extends MaxCut {  
  
 */\*\*  
 \* Represents one set of a partition  
 \*/* private final Set<Integer> S = new HashSet<>();  
 */\*\*  
 \* Represents the other set of a partition  
 \*/* private final Set<Integer> T = new HashSet<>();  
 */\*\*  
 \* Set that keeps track of already chosen edges  
 \*/* private final Set<Edge> E = new HashSet<>();  
  
 public MaxCutBacktrack(Graph graph) {  
 super(graph);  
 }  
  
 private void backtrack(final int depth, final int index) {  
 // We reached the end of the edges.  
 if (index >= graph.getEdges().size()) {  
 return;  
 }  
  
 lbt(depth, S, T);  
 // We have a solution if all vertices are in a set and if the configuration is valid  
 if (isValid(S, T)) {  
 updateCut(S, T);  
 }  
  
 // We iterate all edges starting from the given index  
 for (int i = index; i < graph.getEdges().size(); i += 1) {  
 final Edge edge = graph.getEdges().get(i);  
  
 if (E.contains(edge)) {  
 // No reason to consider this edge again.  
 continue;  
 }  
  
 E.add(edge);  
 lcg(E);  
  
 // Update the sets and make a recursive call.  
 S.add(edge.u());  
 T.add(edge.v());  
 backtrack(depth + 1, i);  
 T.remove(edge.v());  
 S.remove(edge.u());  
  
 // Due to the fact that the graph is undirected, we can build a solution using the other direction.  
 S.add(edge.v());  
 T.add(edge.u());  
 backtrack(depth + 1, i);  
 T.remove(edge.u());  
 S.remove(edge.v());  
  
 E.remove(edge);  
 }  
 }  
  
 @Override  
 protected void internalSolve() {  
 backtrack(0, 0);  
 }  
}

package sbuciu.maxcut;  
  
import sbuciu.maxcut.model.Edge;  
import sbuciu.maxcut.model.Graph;  
  
import java.util.HashSet;  
import java.util.Set;  
  
*/\*\*  
 \* MaxCut Problem Reduction  
 \*/*public class MaxCutPRPruning extends MaxCut {  
  
 */\*\*  
 \* Represents one set of a partition  
 \*/* private final Set<Integer> S = new HashSet<>();  
 */\*\*  
 \* Represents the other set of a partition  
 \*/* private final Set<Integer> T = new HashSet<>();  
 */\*\*  
 \* Set that keeps track of already chosen edges  
 \*/* private final Set<Edge> E = new HashSet<>();  
  
 public MaxCutPRPruning(Graph graph) {  
 super(graph);  
 }  
  
 */\*\*  
 \* Checks if we can consider the edge for the solution  
 \*  
 \** ***@param*** *edge - the edge considered for pruning  
 \** ***@return*** *- true if the edge violates the existing constraints, false otherwise  
 \*/* private boolean canPrune(final Edge edge) {  
 return S.contains(edge.v()) || T.contains(edge.u());  
 }  
  
 private void backtrack(final int depth, final int index) {  
 // We reached the end of the edges.  
 if (index >= graph.getEdges().size()) {  
 return;  
 }  
  
 // We have a solution if all vertices are in a set and if the configuration is valid  
 if (isValid(S, T)) {  
 updateCut(S, T);  
 }  
  
 lbt(depth, S, T);  
 // We iterate all edges starting from the given index  
 for (int i = index; i < graph.getEdges().size(); i += 1) {  
 final Edge edge = graph.getEdges().get(i);  
  
 if (E.contains(edge)) {  
 // No reason to consider this edge again.  
 continue;  
 }  
  
 E.add(edge);  
 lcg(E);  
  
 if (!canPrune(edge)) {  
 // Update the sets and make a recursive call.  
 S.add(edge.u());  
 T.add(edge.v());  
 backtrack(depth + 1, i);  
 T.remove(edge.v());  
 S.remove(edge.u());  
 }  
  
 if (!canPrune(edge.reversed())) {  
 // Due to the fact that the graph is undirected, we can build a solution using the other direction.  
 S.add(edge.v());  
 T.add(edge.u());  
 backtrack(depth + 1, i);  
 T.remove(edge.u());  
 S.remove(edge.v());  
 }  
  
 E.remove(edge);  
 }  
 }  
  
 @Override  
 protected void internalSolve() {  
 backtrack(0, 0);  
 }  
}

package sbuciu.maxcut;  
  
import sbuciu.maxcut.model.Edge;  
import sbuciu.maxcut.model.Graph;  
  
import java.util.HashSet;  
import java.util.LinkedList;  
import java.util.Queue;  
import java.util.Set;  
  
*/\*\*  
 \* MaxCut Problem Reduction  
 \*/*public class MaxCutPRArcConsistent extends MaxCut {  
  
 */\*\*  
 \* Set that keeps track of already chosen edges  
 \*/* private final Set<Edge> E = new HashSet<>();  
  
 public MaxCutPRArcConsistent(Graph graph) {  
 super(graph);  
 }  
  
 static class QueueItem {  
 public final Edge edge;  
 public final boolean isReversed;  
  
 QueueItem(Edge edge, boolean isReversed) {  
 this.edge = edge;  
 this.isReversed = isReversed;  
 }  
 }  
  
 private void arcConsistency(final int depth) {  
 // E contains the edges that we are currently consider for the solution. It represents the arcs.  
 final Queue<QueueItem> agenda = new LinkedList<>();  
 for (final Edge e : E) {  
 agenda.add(new QueueItem(e, false));  
 agenda.add(new QueueItem(e.reversed(), true)); // Add also the reversed arc  
 }  
  
 final Set<Integer> S = new HashSet<>(), T = new HashSet<>();  
 for (int i = 0; i < this.graph.getN(); i += 1) {  
 // Vertex i can be in either set  
 S.add(i);  
 T.add(i);  
 }  
  
 while (!agenda.isEmpty()) {  
 final QueueItem item = agenda.remove();  
 if (item.isReversed) {  
 if (T.contains(item.edge.u())) {  
 S.remove(item.edge.u());  
  
 for (int i = 0; i < this.graph.getN(); i += 1) {  
 if (this.graph.areConnected(i, item.edge.u())) {  
 // The domain changed, so we add all arcs where u is on the right side.  
 agenda.add(new QueueItem(new Edge(i, item.edge.u()), false));  
 }  
 }  
 }  
  
 continue;  
 }  
  
 if (S.contains(item.edge.v())) {  
 T.remove(item.edge.v());  
  
 for (int i = 0; i < this.graph.getN(); i += 1) {  
 if (this.graph.areConnected(i, item.edge.v())) {  
 // The domain changed, so we add all arcs where v is on the right side.  
 agenda.add(new QueueItem(new Edge(i, item.edge.v()), true));  
 }  
 }  
 }  
 }  
  
 if (isValid(S, T)) {  
 updateCut(S, T);  
 }  
  
 lbt(depth, S, T);  
 }  
  
 private void backtrack(final int depth, final int index) {  
 // We reached the end of the edges.  
 if (index >= graph.getEdges().size()) {  
 return;  
 }  
  
 // We iterate all edges starting from the given index  
 for (int i = index; i < graph.getEdges().size(); i += 1) {  
 final Edge edge = graph.getEdges().get(i);  
  
 if (E.contains(edge)) {  
 // No reason to consider this edge again.  
 continue;  
 }  
  
 E.add(edge);  
 lcg(E);  
  
 arcConsistency(depth);  
 backtrack(depth + 1, i);  
 E.remove(edge);  
 }  
 }  
  
 @Override  
 protected void internalSolve() {  
 backtrack(0, 0);  
 }  
}

package sbuciu.maxcut;  
  
import sbuciu.maxcut.model.Edge;  
import sbuciu.maxcut.model.Graph;  
  
import java.util.ArrayList;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Set;  
  
*/\*\*  
 \* MaxCut Simple Search  
 \*/*public class MaxCutSSForwardCheck extends MaxCut {  
  
 */\*\*  
 \* Represents one set of a partition  
 \*/* private final Set<Integer> S = new HashSet<>();  
 */\*\*  
 \* Represents the other set of a partition  
 \*/* private final Set<Integer> T = new HashSet<>();  
  
 public MaxCutSSForwardCheck(Graph graph) {  
 super(graph);  
 }  
  
 */\*\*  
 \* This function filters out any edge that would not produce a valid solution given the selected edges.  
 \*  
 \** ***@param*** *edges - list of edges to filter  
 \** ***@param*** *e - the latest selected edge  
 \** ***@return*** *a list with the filtered edges  
 \*/* private List<Edge> commit(final List<Edge> edges, final Edge e) {  
 final List<Edge> validEdges = new ArrayList<>();  
 for (final Edge edge : edges) {  
 if (edge.equals(e)) {  
 continue;  
 }  
  
 if (!S.contains(edge.v()) && !T.contains(edge.u())) {  
 validEdges.add(edge);  
 }  
 }  
  
 return validEdges;  
 }  
  
 private void backtrack(final int depth, List<Edge> edges) {  
 if (isValid(S, T)) {  
 updateCut(S, T);  
 }  
  
 lbt(depth, S, T);  
 lcg(edges);  
 for (final Edge edge : edges) {  
 S.add(edge.u());  
 T.add(edge.v());  
 backtrack(depth + 1, commit(edges, edge));  
 T.remove(edge.v());  
 S.remove(edge.u());  
  
 final Edge reversed = edge.reversed();  
 S.add(reversed.u());  
 T.add(reversed.v());  
 backtrack(depth + 1, commit(edges, edge));  
 T.remove(reversed.v());  
 S.remove(reversed.u());  
 }  
 }  
  
 @Override  
 protected void internalSolve() {  
 backtrack(0, graph.getEdges());  
 }  
}