Lisa Ye
05/13/2020
CS 4613
Futoshiki

Instructions to Run
*Along with the requested source code in plain text file, I have also attached the original .cpp file
Programming Language: C++
IDE used: Visual Studio C++ 2017

To Compile the Code:
<u>If you are using an IDE:</u>
1. Include the source code (probably need the .cpp file) in the project/workspace/solution
2. Hit whatever button that is equivalent to run on the IDE.

<u>If you are using command line:</u>
1. Make sure you are inside the correct directory
2. Type in the following to compile:
    gcc Source.cpp -o futoshiki
3. Afterwards, type in the following to run the program:
    futoshiki

To Run the Program:
- The program will prompt you to enter the input file name you want to test and the output file name you want to generate.
- Make sure you have the input files in the same folder as the source code. After you run the program, the output will be generated inside the same folder as the source code and will be named after the name you have inputted.
- On Windows, you will need to include the ".txt" extension while entering the file name. Not sure on other operating systems, but include it just to be safe.

2 1 5 4 3
1 3 4 2 5
4 5 1 3 2
5 2 3 1 4
3 4 2 5 1

3 4 2 5 1
1 5 4 2 3
2 3 5 1 4
5 1 3 4 2
4 2 1 3 5

3 1 5 2 4
5 2 3 4 1
1 3 4 5 2
4 5 2 1 3
2 4 1 3 5

```
/*
	Lisa Ye
	CS 4613 Artificial Intelligence
	Project 2 - Futoshiki

*/


#include<string>
#include<iostream>
#include<fstream>
#include<vector>
#include<algorithm>
#include<queue>
#include<cmath>
#include <sstream>
using namespace std;

bool DEBUG = false; // for everything before backtracking
bool DEBUG2 = true;
char GREATER_THAN = '>';
char LESS_THAN = '<';
char VERTICAL_GREATER_THAN = 'v';
char VERTICAL_LESS_THAN = '^';

// represents a position on the board
struct Position {
	int row;
	int column;
	Position(int row, int column) : row(row), column(column) {}
};

// Represents a cell in the board
// Represent a variable in CSP
class Cell {
	friend ostream& operator<<(ostream& os, const Cell& rhs);
public:

	Cell(int value, const vector<int>& domain) : value(value), domain(domain) {}
	// getters
	int getVal() const { return value; }
	int getDomainSize() const { return domain.size(); }
	vector<int> getDomain() const { return domain; }
	vector<Position> getLessThanMe() const { return lessThanMe; }
	vector<Position> getGreaterThanMe() const { return greaterThanMe; }
	// setters
	void setVal(int val) { value = val; }
	// removes @val from the domain
	bool removeValFromDomain(int val) {
		auto result = find(domain.begin(), domain.end(), val);
		if (result != domain.end()) {
			domain.erase(result);
```

```
                return true;
            }
            else {
                // cerr << "value does not exist in the domain, failed the remove" << endl;
                return false;
            }
        }
        // add constraints to current cell
        void setConstraints(char symbol, int row, int column) {
            Position rhs(row, column);
            if (symbol == GREATER_THAN || symbol == VERTICAL_GREATER_THAN) {
                lessThanMe.push_back(rhs);
            }
            else if (symbol == LESS_THAN || symbol == VERTICAL_LESS_THAN) {
                greaterThanMe.push_back(rhs);
            }
            else {
                cerr << "failed to set constraints: invalid symbol" << endl;
            }
        }

private:
        /*
            Possible value for the cell(1,2,3,4,5)
            If a cell has an initial value, domain should only have that value
        */
        vector<int> domain;

        /*
            holds the positions in which the value at those positions
            must be less than ourself
        */
        vector<Position> lessThanMe;
        /*
            holds the positions in which the value at those positions
            must be greater than ourself
        */
        vector<Position> greaterThanMe;

        // holds the current value of this cell
        int value;
};


// reads in input file and translate text to initial board
void readInput(ifstream& ifs, vector<vector<int>>& initial);
void fillInput(const vector<vector<int>>& data, vector<vector<Cell*>>& board);
// reads in input file and translate inequality constraints
void fillConstraints(ifstream& ifs, const vector<vector<Cell*>>&);
// returns a deep copy of the board
vector<vector<Cell*>> boardDeepCopy(const vector<vector<Cell*>>& board);
// output operator overloading
```

```
ostream& operator<<(ostream& os, const vector<vector<int>>& rhs);
ostream& operator<<(ostream& os, const vector<vector<Cell*>>& rhs);
ostream& operator<<(ostream& os, const Position& rhs);
//forward checking algorithm
bool forwardCheck(vector<vector<Cell*>>& board);
// forward check, returns false when the check does not modify any variable's domain
bool check(vector<vector<Cell*>>& board, size_t row, size_t column);
// check if board is valid, i.e. does not violate any constraints
// after assigning @value at board[row][column]
// return false if board is not valid, return true if pass all constaints(board is valid)
bool checkConstraint(const vector<vector<Cell*>>& board, int row, int column, int value);
// selects next variable to be assigned, returns position of that variable/cell
// use most constrained heuristic: variables with smallest domain
// then use most constraining heuristic: variables with most constraints tied to it
Position selectVariable(const vector<vector<Cell*>>& board);
// checks if a board is complete: if all variable/Cell has a value
bool isComplete(const vector<vector<Cell*>>& board);
// given assignment, assigns the assignment to the @board
void assign(vector<pair<Position, int>> assignments, vector<vector<Cell*>>& board);
// removes an assignment from the list of assignments
void removedAssignment(vector<pair<Position, int>>& assignments, const pair<Position, int >& toRemove);
// driver function for recursion
bool backtrackSearch(vector<vector<Cell*>>& board);
// recursive backtracking function
vector<pair<Position, int>> backtrack(vector<pair<Position, int>>& assignment, vector<vector<Cell*>>&
board);


int main() {

	// vector to hold data from initial board
	vector<vector<int>> initial(5, vector<int>(5, 0));
	vector<vector<Cell*>> initialBoard(5);
	// opens file
	ifstream input;
	cout << "Enter the input file name(include the .txt):" << endl;
	string filename;
	cin >> filename;
	string outputFilename;
	cout << "Enter the output file name(include the .txt)" << endl;
	cin >> outputFilename;
	input.open(filename);
	// check if file is valid
	if (!input) {
		cout << "failed to open file" << endl;
		exit(1);
	}
	// fill vector from input
	readInput(input, initial);
	fillInput(initial, initialBoard);
	// read rest of input file
	// fill in the constraints
```

```
fillConstraints(input, initialBoard);
// testing reading of input
if (DEBUG) {
	cout << initialBoard << endl;
	cout << *initialBoard[2][4] << endl;
	cout << *initialBoard[3][4] << endl;
}

// First and foremost: forward check to reduce domain
if (!forwardCheck(initialBoard)) {
	cout << "Problem cannot be solved" << endl;
}

if (DEBUG) {
	cout << *initialBoard[4][0] << endl;
	cout << *initialBoard[3][0] << endl;
	cout << "testing checkConstraints:" << endl;
	cout << "Position [4,0]" << endl << *initialBoard[4][0] << endl;
	cout << "should be 0: " << checkConstraint(initialBoard, 3, 0, 2) << endl;
	cout << "should be 0: " << checkConstraint(initialBoard, 4, 1, 3) << endl;
	cout << "should be 0: " << checkConstraint(initialBoard, 2, 0, 3) << endl;
	cout << "should be 1: " << checkConstraint(initialBoard, 3, 0, 5) << endl;
}
if (DEBUG) {
	cout << "testing deep copy of board:" << endl;
	cout << *initialBoard[3][0] << endl;
	vector<vector<Cell*>> copy = boardDeepCopy(initialBoard);
	cout << *copy[3][0] << endl;

}


// backtracking search
bool success = backtrackSearch(initialBoard);

// after algorithm, board should display solution
if (DEBUG2) {
	cout << initialBoard;
}

// write to output file
ofstream output(outputFilename);
if (success) {
	for (size_t i = 0; i < initialBoard.size(); i++) {
		for (size_t j = 0; j < initialBoard[i].size(); j++) {
			output << initialBoard[i][j]->getVal() << " ";
		}
		output << endl;
	}
}
else {
	output << "This puzzle cannot be solved.";
```

```
	}


	// free board off heap
	for (const vector<Cell*>& row : initialBoard) {
		for (Cell* cell : row) {
			delete cell;
		}
	}
	// close filestream when finished
	input.close();

}

void readInput(ifstream& ifs, vector<vector<int>>& initial) {
	string row;
	int num;
	// fills up initial board
	for (int i = 0; i < 5; i++) {
		getline(ifs, row);
		istringstream ss(row);
		size_t j = 0;
		while (ss >> num) {
			initial[i][j] = num;
			j++;
		}
	}
}

ostream& operator<<(ostream& os, const vector<vector<int>>& rhs) {
	for (size_t i = 0; i < rhs.size(); i++) {
		for (size_t j = 0; j < rhs[i].size(); j++) {
			os << rhs[i][j] << " ";
		}
		os << endl;
	}
	return os;
}

ostream& operator<<(ostream& os, const vector<vector<Cell*>>& rhs) {
	for (size_t i = 0; i < rhs.size(); i++) {
		for (size_t j = 0; j < rhs[i].size(); j++) {
			os << rhs[i][j]->getVal() << " ";
		}
		os << endl;
	}
	return os;
}

ostream& operator<<(ostream& os, const Position& rhs) {
	os << "[" << rhs.row << "," << rhs.column << "]" << endl;
	return os;
```

```
}

ostream& operator<<(ostream& os, const Cell& rhs) {
	os << "value: " << rhs.value << endl;
	os << "domain: { ";
	for (int val : rhs.domain) {
		os << val << " ";
	}
	os << "}" << endl;
	os << "These positions must be less than me:" << endl;
	for (const Position& pos : rhs.lessThanMe) {
		os << pos;
	}
	os << endl;
	os << "These positions must be greater than me:" << endl;
	for (const Position& pos : rhs.greaterThanMe) {
		os << pos;
	}
	os << endl;
	return os;
}

void fillInput(const vector<vector<int>>& data, vector<vector<Cell*>>& board) {
	vector<int> fullDomain = { 1,2,3,4,5 };
	for (size_t i = 0; i < data.size(); i++) {
		for (size_t j = 0; j < data[i].size(); j++) {
			// if cell has initial value,
			if (data[i][j] != 0) {
				// assign the value, and the domain with the value
				board[i].push_back(new Cell(data[i][j], { data[i][j] }));
			}
			// if cell has no value
			else {
				// assign 0 and the full domain
				board[i].push_back(new Cell(0, fullDomain));
			}
		}
	}
}

void fillConstraints(ifstream& ifs, const vector<vector<Cell*>>& board) {
	string row;
	char symbol;
	// for empty line
	getline(ifs, row);
	// for left and right inequality
	for (int i = 0; i < 5; i++) {
		getline(ifs, row);
		istringstream ss(row);
		size_t j = 0;
		while (ss >> symbol) {
			// same i, j > j+1
```

```
                        if (symbol == GREATER_THAN) {
                                //lhs
                                board[i][j]->setConstraints(GREATER_THAN, i, j+1);
                                //rhs(note that symbol is flipped)
                                board[i][j + 1]->setConstraints(LESS_THAN, i, j);
                        }
                        // same i, j > j+1
                        else if (symbol == LESS_THAN) {
                                //lhs
                                board[i][j]->setConstraints(LESS_THAN, i, j + 1);
                                //rhs(note that symbol is flipped)
                                board[i][j + 1]->setConstraints(GREATER_THAN, i, j);
                        }
                        j++;
                }
        }
        // for empty line
        getline(ifs, row);
        // for top and down inequality
        for (int i = 0; i < 4; i++) {
                getline(ifs, row);
                istringstream ss(row);
                size_t j = 0;
                while (ss >> symbol) {
                        // same j, i > i+1
                        if (symbol == VERTICAL_GREATER_THAN) {
                                //lhs
                                board[i][j]->setConstraints(VERTICAL_GREATER_THAN, i+1, j);
                                //rhs(note that symbol is flipped)
                                board[i+1][j]->setConstraints(VERTICAL_LESS_THAN, i, j);
                        }
                        // same j, i > i+1
                        else if (symbol == VERTICAL_LESS_THAN) {
                                //lhs
                                board[i][j]->setConstraints(VERTICAL_LESS_THAN, i+1, j);
                                //rhs(note that symbol is flipped)
                                board[i+1][j]->setConstraints(VERTICAL_GREATER_THAN, i, j);
                        }
                        j++;
                }
        }


}

vector<vector<Cell*>> boardDeepCopy(const vector<vector<Cell*>>& board) {
        vector<vector<Cell*>> result(5);
        for (size_t i = 0; i < board.size(); i++) {
                for (size_t j = 0; j < board[i].size(); j++) {
                        result[i].push_back(new Cell(*board[i][j]));
                }
        }
```

```
	return result;
}

bool forwardCheck(vector<vector<Cell*>>& board) {
	// continuously do forward checking until domains of all variable is unchanged
	// or domain of any variable is empty --> failure
	bool done = false;
	while (!done) {
		// go through each variable, look for the one that already has a value
		// their neighbors: every Cell in their row or column or inequality
		for (size_t i = 0; i < board.size(); i++) {
			for (size_t j = 0; j < board[i].size(); j++) {
				// if it has a value
				if (board[i][j]->getVal() != 0) {
					// then do forward check on this Cell's neighbors
					done = !check(board, i, j);
				}
			}
		}
	}
	// check if any domain is empty
	for (size_t i = 0; i < board.size(); i++) {
		for (size_t j = 0; j < board[i].size(); j++) {
			if (board[i][j]->getDomainSize() == 0) {
				cout << Position(i, j) << " is empty" << endl;
				return false;
			}
		}
	}
	return true;
}

bool check(vector<vector<Cell*>>& board, size_t row, size_t column) {
	bool changed = false;
	// reduce domain via row's alldiff constraint
	int val = board[row][column]->getVal();
	for (size_t j = 0; j < board[row].size(); j++) {
		if (j != column) { // leave the curr one out
			// take the val out of each Cell from the row
			// if failed to remove, that means already removed
			if (board[row][j]->removeValFromDomain(val)) { changed = true; }
		}
	}
	// reduce domain via column's alldiff constraint
	for (size_t i = 0; i < board.size(); i++) {
		if (i != row) { // leave the curr one out
			// take the val out of each Cell from the row
			// if failed to remove, that means already removed
			if (board[i][column]->removeValFromDomain(val)) { changed = true; }
		}
	}
	// reduce domain via inequality constraints
```

```
	vector<Position> lessThan = board[row][column]->getLessThanMe();
	vector<Position> greaterThan = board[row][column]->getGreaterThanMe();
	// loop through all positions that should be less than curr position
	for (const Position& pos : lessThan) {
		// remove all values greater than val from these positions
		for (int i = 5; i > val; i--) {
			if (board[pos.row][pos.column]->removeValFromDomain(i)) { changed = true; }
		}
	}
	// loop through all positions that should be greater than curr position
	for (const Position& pos : greaterThan) {
		// remove all values less than val from these positions
		for (int i = 1; i < val; i++) {
			if (board[pos.row][pos.column]->removeValFromDomain(i)) { changed = true; }
		}
	}
	return changed;
}

bool checkConstraint(const vector<vector<Cell*>>& board, int row, int column, int value) {

	// check alldiff row constraints
	for (size_t j = 0; j < board[row].size(); j++) {
		if (j != column) {
			if (board[row][j]->getVal() == value) { return false; }
		}
	}
	// check alldiff column constraints
	for (size_t i = 0; i < board.size(); i++) {
		if (i != row) {
			if (board[i][column]->getVal() == value) { return false; }
		}
	}
	// check inequality constraints
	vector<Position> lessThan = board[row][column]->getLessThanMe();
	vector<Position> greaterThan = board[row][column]->getGreaterThanMe();
	// loop through all positions that should be less than curr position
	for (const Position& pos : lessThan) {
		if (board[pos.row][pos.column]->getVal() != 0) {
			if (board[pos.row][pos.column]->getVal() > value) { return false; }
		}
	}
	// loop through all positions that should be greater than curr position
	for (const Position& pos : greaterThan) {
		if (board[pos.row][pos.column]->getVal() != 0) {
			if (board[pos.row][pos.column]->getVal() < value) { return false; }
		}
	}

	return true;
}
```

```
Position selectVariable(const vector<vector<Cell*>>& board) {
	// stores the candidates, as there can be multiple
	vector<Position> candidates;
	int leastDomainSize = 5;
	// most constrained variable:
	// loop through once to find least domain size
	for (size_t i = 0; i < board.size(); i++) {
		for (size_t j = 0; j < board[i].size(); j++) {
			if (board[i][j]->getDomainSize() < leastDomainSize && board[i][j]->getVal() == 0) {
				leastDomainSize = board[i][j]->getDomainSize();
			}
		}
	}
	// loop through again to find the variables with the least domain size
	for (size_t i = 0; i < board.size(); i++) {
		for (size_t j = 0; j < board[i].size(); j++) {
			// if variable has least domain size AND does not already have a value assgined
			if (board[i][j]->getDomainSize() == leastDomainSize && board[i][j]->getVal() == 0) {
				candidates.push_back(Position(i, j));
			}
		}
	}
	// most constaining variable:
	// # of constraints = degree
	Position result = candidates[0];
	int maxDegree = 0;
	for (const Position& pos : candidates) {
		int degree = 0;
		degree += board[pos.row][pos.column]->getGreaterThanMe().size();
		degree += board[pos.row][pos.column]->getLessThanMe().size();
		if (degree > maxDegree) {
			maxDegree = degree;
			result = pos;
		}
	}
	return result;
}

bool isComplete(const vector<vector<Cell*>>& board) {
	for (size_t i = 0; i < board.size(); i++) {
		for (size_t j = 0; j < board[i].size(); j++) {
			if (board[i][j]->getVal() == 0) { return false; }
		}
	}
	return true;
}

void assign(vector<pair<Position, int>> assignments, vector<vector<Cell*>>& board) {
	for (pair<Position, int> assignment : assignments) {
		Position pos = assignment.first;
		board[pos.row][pos.column]->setVal(assignment.second);
	}
```

```
}

void removedAssignment(vector<pair<Position, int>>& assignments, const pair<Position, int >& toRemove) {
	for (size_t i = 0; i < assignments.size(); i++) {
		pair<Position, int> assignment = assignments[i];
		if (assignment.first.row == toRemove.first.row &&
			assignment.first.column == toRemove.first.column &&
			assignment.second == toRemove.second) {
			assignments[i] = assignments[assignments.size() - 1];
			assignments.pop_back();
			return;
		}
	}
}

bool backtrackSearch(vector<vector<Cell*>>& board) {
	// a list of pair<Position, int> representing assignments
	// pair.first is of type Position, indicates the variable
	// pair.second is of type int, indicates the value assigned to the variable
	vector<pair<Position, int>> assignments;
	vector<pair<Position, int>> result;
	// get result
	result = backtrack(assignments, board);
	// if there isn't a solution, return false
	if (result.size() == 0) {
		if (DEBUG2) { cout << "no solution" << endl; }
		return false;
	}
	// otherwise, fill in the solution
	if (DEBUG2) { cout << "YAY, solution found" << endl; }
	assign(result, board);
	return true;

}

// empty return vector indicates failure
vector<pair<Position, int>> backtrack(vector<pair<Position, int>>& assignment, vector<vector<Cell*>>&
board) {

	// save old state of board in case need to revert
	vector<vector<Cell*>> old = boardDeepCopy(board);
	// carry out the assignments
	assign(assignment, board);
	// check if assignment is complete
	if (isComplete(board)) { return assignment; } // if yes, return assignment: found solution
	// if not, select next variable to assign
	Position nextVar = selectVariable(board);
	// loop through the domain of this variable
	for (int val : board[nextVar.row][nextVar.column]->getDomain()) {
		// check if assignment is consistent/valid
		if (checkConstraint(board, nextVar.row, nextVar.column, val)) {
			// if yes, add value to assignment
```

```
                        pair<Position, int> newAssignment(nextVar, val);
                        assignment.push_back(newAssignment);
                        // call recursive function
                        vector<pair<Position, int>> result = backtrack(assignment, board);
                        // if success[assignment vector is not empty]
                        if (result.size() != 0) { return result; }
                        // remove assignment
                        // can't just pop back because we don't know if in our recursive calls
                        // whether we kept some assignemnts or not, can't guranteed it's the last one
                        removedAssignment(assignment, newAssignment);
                }
                // if not, don't worry about it, continue onto the next domain value
        }
        // free board off heap before reverting back
        vector<vector<Cell*>> toDelete = board;
        for (const vector<Cell*>& row : toDelete) {
                for (Cell* cell : row) {
                        delete cell;
                }
        }
        board = old; // revert the board back to the old state
        // return failure, which is represented by an empty vector
        return {};
}
```