CS 890BR Constraint Programming

Final Project
on

Implementation of Crossword Puzzle & n-Queens Problem

---

by

Tejaswini Balguri (200439882)
tbm067@uregina.ca

Submitted in partial fulfilment of the degree of
Master's in computer science: Prof. Sultan Ahmed

Department of Computer Science
University of Regina
August 9, 2022

Table of Contents

1. ABSTRACT

Crossword puzzle and n-Queens can be described as a classic constraint satisfaction problem. The Crossword Puzzle falls under the category of symbolic constraint satisfaction problem. By this, it means a CSP in which variables range over non-numeric domains. The n-Queens Problem falls under the category of constraint satisfaction problem on integers. This is probably the most known CSP. The crossword puzzle consists of a grid with fillable and nonbillable cells. Predefined set of words can be filled into these fillable cells either vertically or horizontally. Our goal is to fill each fillable cell with a letter that results in forming a meaningful word. For n-Queens problem, the goal is to place N queens on a N*N chessboard (where N>=3) so that no two queens attack each other.

In CS 890BR Constraint Programming class, we have learnt the theoretical implementation of the crossword puzzle and n-queen's problems. We were able to successfully represent crossword and n-queens problem as CSP by determining domains, variables and constraints. This theoretical representation is feasible in case of smaller number of words for crossword puzzle and if the value of 'n' is either 4 or 8. But, if the number of words or the queens are considerably large, then the theoretical implementation becomes difficult. So, through this project, I chose to implement them programmatically in order to solve the given crossword or n-queens problem irrespective of the grid size or the 'n' value for queen. Motivated by this, in this project, I am implementing N-Queens using backtrack approach which is a powerful brute-force approach. Whereas crossword puzzle is being implemented using forward checking and arc consistency. After applying these techniques I'm presenting the results with different values of queens for n-Queens problem and different set of words for crossword puzzle.

2. INTRODUCTION

n-Queens and Crossword puzzles are frequently used as examples of constraint satisfaction problems (CSPs) and can be used to great effect in crossword-puzzle creation and for finding optimal solution for placing n-Queens on a chess board.

2.1 Constraint Satisfaction Problem

Constraints are set of rules limiting the values assigned to a variable or group of variables. A constraint satisfaction problem consists of a finite set of constraints, each on a subsequence of a given sequence of variables. To solve any problem in the form of CP, we need to formulate it as CSP. To accomplish this, first introduce some variables ranging over specific domains and constraints over these variables. Then choose some language in which constraints are expressed usually a small subset of 1st order logic.

Given:
Variables $x_1 , \ldots , x_n$,
Domains $D_1 , \ldots , D_n$,
Constraint Satisfaction Problem (CSP)

$$\{C ; x_1 \in D_1 , \ldots , x_n \in Dn\}$$

C – constraints, each on a subsequence of $x_1 , \ldots , x_n$. $(d_1 , \ldots , dn) \in D_1 \times \ldots \times Dn$ is a solution to

$$\{C ; x_1 \in D_1 , \ldots , x_n \in D_n\}$$

if for every constraint $C \in C$ on $x_{i1} , \ldots , x_{im}$ $(d_{i1} , \ldots , d_{im}) \in C$

2.2 Problem Definition

2.2.1 Crossword Puzzle

The crossword puzzle consists of a grid with fillable and nonbillable cells. Predefined set of words can be filled into these fillable cells either vertically or horizontally. Our goal is to fill each fillable cell with a letter that results in forming a meaningful word from the set of predefined words list both vertically and horizontally. To sum up the problem, a predefined set of words list will be given, we need to fill letters in the blank cells that satisfies each of the constraint that we define like length of the word and letter should match the word at intersection point both vertically and horizontally.

We need to formulate this as a CSP. Then we will define variables, domains and constraints.
Variables: Unblocked cells
Domains: The domain of each variable is the set of letters of the alphabet.
Constraints: For each vertical or horizontal contiguous segment of unblocked cells, we add a constraint between the cells in that segment, constraining the letters assigned to the cells in that segment to form a word that appears in the dictionary
An example of crossword puzzle is given below

A	D	I	M	■	R	I	P	S	■	F	A	T
C	U	T	E	■	E	T	A	T	■	A	L	E
D	E	S	D	E	M	O	N	A	■	I	A	N
C	L	A	U	D	I	O	■	T	U	R	N	S
■	■	■	S	E	T	■	T	E	R	M	■	■
A	S	W	A	N	■	P	E	N	N	A	M	E
D	I	I	■	■	H	A	L	■	■	I	M	S
O	R	L	A	N	D	O	■	E	D	D	I	E
■	■	D	I	A	S	■	A	S	U	■	■	■
S	T	O	M	P	■	P	R	A	N	C	E	D
E	R	A	■	P	E	T	R	U	C	H	I	O
L	I	T	■	E	L	S	A	■	A	I	N	T
L	O	S	■	R	I	D	S	■	N	A	S	H

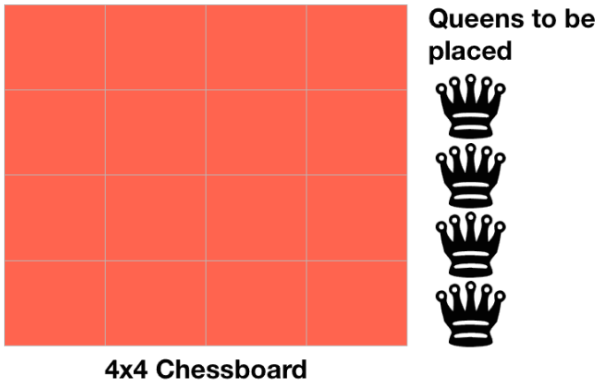
Fig:1 Example - Crossword Puzzle

2.2.2 n-Queens Problem

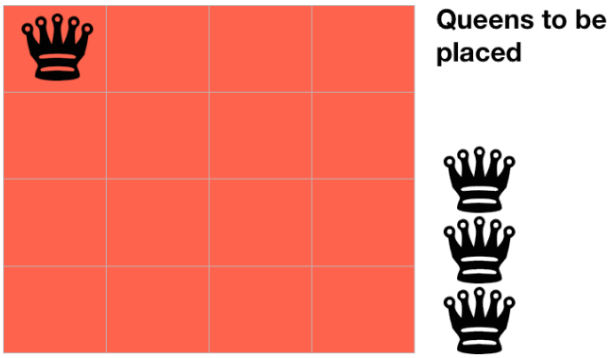
The n-Queens Problem falls under the category of constraint satisfaction problem on integers. This is probably the most known CSP. The goal is to place N queens on a N x N chessboard (where N>=3) so that no two queens attack each other. Also, a queen can move in a row, diagonally or in its line. One possible representation of this problem as a CSP uses n variables, $x_1, \ldots, x_n$, each with the domain [1...n]. Here, xi represents the position of the queen placed in the ith column of the chess board. The constraints for this problem are as follows for $i \in [1...n-1]$ and $j \in [i+1...n]$:

- $x_i \neq x_j$ (no two queens in the same row)
- $x_i - x_j \neq i - j$ (no two queens in each South-West – North-East diagonal)
- $x_i - x_j \neq j - I$ (no two queens in each North-West – South-East diagonal)

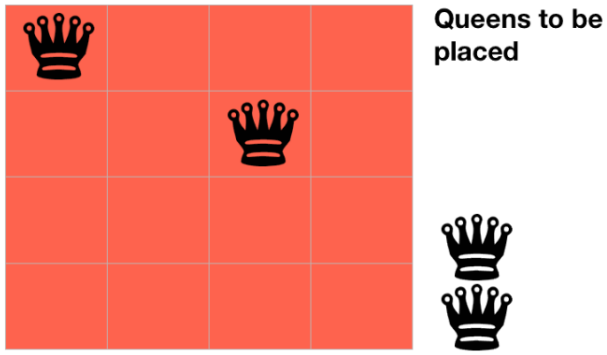
Let us consider an example of 4x4 chessboard and number of queens are 4



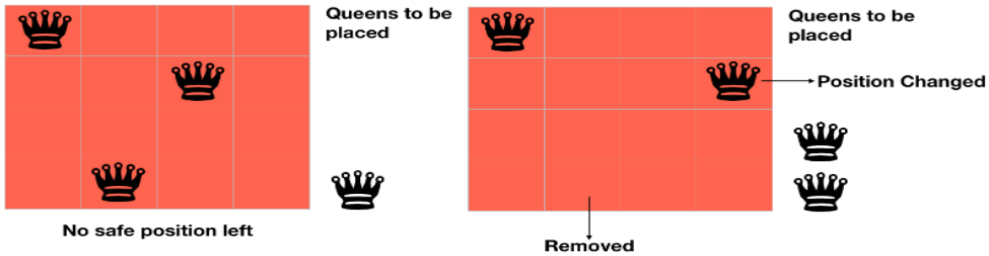
Initially we are placing the first queen in first row



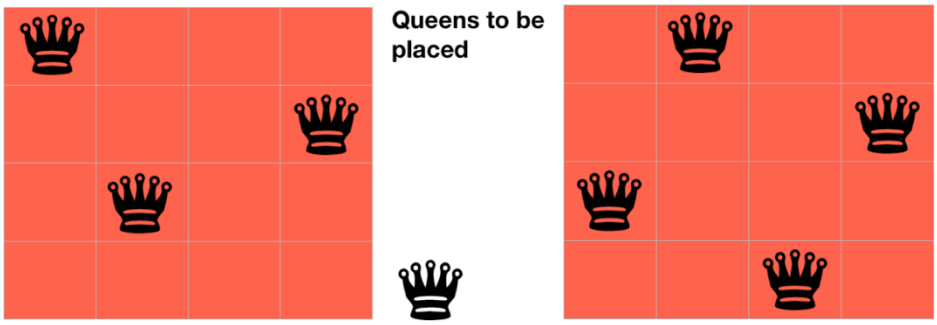
Next step is to place the queen in safe position so that first queen will not attack second one.



Now we place third queen but there is no safe place for 4th queen. So, we remove it from the position



Now we place the third queen and finally place the 4th queen so that no two queens attack each other and obtain the solution.



In this project, I am trying to achieve the solutions for both n-Queens and crossword puzzle by following the below approach:
For ***Crossword Puzzle***, I will represent it as CSP by determining variables, domains and constraints. I will initially pass a grid of blocked and unblocked cells, which will be filled by letters in the unblocked cells. Also, I will pass a list of words that will be filled in unblocked cells. Then after successful CSP representation of the problem, I will apply arc consistency and forward checking to evaluate the solution and match the constraints. In-case the solution is not feasible I will backtrack the solution and try with new words.
For ***n-Queens Problem,*** I will simulate the results on a chess board. I will use backtrack algorithm for implementing this.
Languages: JAVA (Crossword Puzzle), JavaScript & HTML (n-queens)

3. BACKGROUND KNOWLEDGE

For solving crossword puzzle, I am using the below techniques

- Forward Checking
- Arc Consistency

Whereas, for n-Queens problem, I am using below technique to find the solution

- Backtracking

3.1 Forward Checking

Future conflict avoidance is made simple by using forward checking. It applies a restricted form of arc consistency to the variables that have not yet been instantiated rather than performing arc consistency to the instantiated variables. Because forward checking only examines the constraints between the present variable and the future variables, we refer to restricted arc consistency. Any value in the domain of a "future" variable that conflicts with an assignment made to the current variable is (temporarily) removed from the domain. The benefit of approach is that it is instantly apparent that the partial solution is inconsistent if the domain of a future variable becomes empty. As a result, branches of the search tree that will end in failure can be trimmed earlier using forward checking than with conventional backtracking. It should be noted that anytime a new variable is considered, all of its remaining values are assured to be consistent with the previous variables, therefore it is no longer essential to validate an assignment against previous assignments.
Forward Checking steps can be simply described as
Suppose when a variable ‘x’ is assigned

- Examine each unassigned variable Y connected to X by a constraint
- Delete from Y’s domain any value inconsistent with the value chosen for X
- If assignment becomes impossible (anywhere), backtrack.

The branches of the search tree that will result in failure can be pruned sooner with forward checking because it finds the discrepancy earlier than simple backtracking. This lessens the size of the search tree and, ideally, the overall amount of labour. But it should be remembered that when each assignment is added to the present incomplete solution, forward checking performs extra work.

3.2 Arc Consistency

Consistency is an inference mechanism that strengthens the search by ruling out variable assignments. The most basic consistency check compares the current assignment to the list of restrictions. One of the most effective propagation methods for binary constraints is arc consistency. We look for a supporting value to assign to the other variable for each value of a variable in the constraint. If there is none, it is safe to omit the value. The limitation is arc consistent in all other cases.
(Arc Consistency) The pair (X, Y) of constraint variables is arc consistent if for each value there exists a value such that the assignments X = x and Y = y satisfies all binary constraints between X and Y. A CSP is arc consistent if all variable pairs are arc consistent. Consider a straightforward CSP with the binary constraint A B, the variables A and B being bounded by their respective domains and. Based on the constraint and the restriction we have on A, we can conclude that value 1 can be securely deleted from the database.
For example, a constraint C on the variables x, y with the domains X and Y (so $C \subseteq X \times Y$) is arc consistent if

- $\forall a \in X \; \exists \; b \in Y \; (a, b) \in C$,
- $\forall b \in Y \; \exists \; a \in X \; (a, b) \in C$.

A CSP is arc consistent if all its binary constraints are.

3.3 Backtracking

Backtracking is a common algorithmic strategy that involves looking through all potential combinations to find a solution to a computing problem; once the solution is discovered, the algorithm is stopped. With a top-down, left-to-right search path, this algorithm conducts a deep search in a tree. Simple backtracking (BT), which can be thought of as a combination of pure generate & test and a fraction of arc consistency, executes some sort of consistency mechanism. The BT algorithm verifies the validity of constraints considering partial instantiation by checking the arc coherence among already instantiated variables. It is possible to only verify the constraints and arcs that contain the last instantiated variable because the domains of instantiated variables only contain one value. The algorithm goes back to a fresh instantiation if any domain is lowered because the accompanying constraint is therefore inconsistent.
The backtracking method kicks in when this search is unsuccessful, or a tree terminal node is located. In order to uncover alternative answers, this process forces the system to go back along the same path. For each error that is discovered, the algorithm goes back one node of the tree.

Advantages

- Backtracking can almost solve any problems, due to its brute-force nature.
- Can be used to find all the existing solutions if there exists for any problem.
- It is a step-by-step representation of a solution to a given problem, which is very easy to understand.
- Very easy to write the code, and to debug.

Disadvantages

- It is very slow compared to other solutions.
- Depending on the data that you have, there is the possibility to perform a very large search with Backtracking and at the end do not find any match to your search params.

4. ACCOMPLISHMENT

4.1 Crossword Puzzle

Initially I am representing the CP as a CSP by defining variables, domains, constraints
Variables: Unblocked cells
Domains: The domain of each variable is the set of letters of the alphabet.
Constraints: For each vertical or horizontal contiguous segment of unblocked cells, we add a constraint between the cells in that segment, constraining the letters assigned to the cells in that segment to form a word that appears in the dictionary

4.1.1 Flowchart

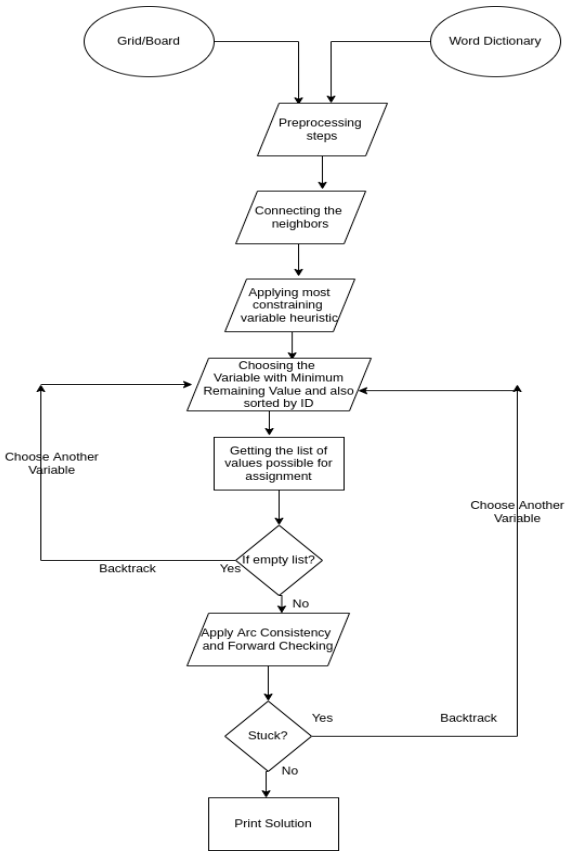


Fig:2 Flowchart for crossword puzzle problem

4.1.2 Task Description

Pre-Processing Steps:

From the word dictionary, 26 x (possible word length) combinations are made and saved so that every alphabet is at every possible index of the word length for every conceivable word length. It has been determined that every empty space, both horizontally and vertically, will be treated as a variable (in CSP term).

Connecting the neighbours:

All horizontal and vertical variables have been examined, and those that are connected are saved as an adjacency list.

Applying Most Constraining Variable Heuristics:
As a heuristic, the Most Constraining Variable is selected. The horizontal or vertical variable with the most significant constraints (connections) is typically picked first in constraining variable heuristics.

Choosing the Variable with Minimum Remaining Value:
The variable with the least feasible value is picked and assigned first among all those variables with the highest constraint. Once more, if any variables meet both requirements, this list is sorted using the ID that was given to them. If there are no more viable values for the variable, we must go back and select different variables.

Applying Arc Consistency and Forward Checking:
Forward Checking:
We will loop over all the neighbours and apply the constraints set by assigning the value after each successful assignment. Any neighbour whose values have decreased to zero will serve as a cue to change the value. The second value will be the next one that is appropriate and meets the criteria for choosing a value. If there is no other value, there will be a backtracking.

Arc Consistency:
The arc consistency stage will start once some of the neighbours' values have been removed during forward checking. As we traverse each neighbour in arc consistency, we take the intersection cell between that neighbour and each of their neighbours.
In order to fix the alphabets at the junction cell and prune the domains of the neighbours and the neighbours' neighbours, we will add additional constraint through arc consistency.
If there are no words that can be assigned in this way, we must go back and pick another word from the list while keeping the limitations in mind.

The solution is printed if, after applying forward checking and arc consistency, we discover a solution that satisfies all the criteria. Additionally, if after testing all possible values we are unable to identify any feasible assignment, we print that the given grid and given word dictionary have no potential solutions.

4.2 n-Queens Problem

As mentioned earlier, I used a backtracking algorithm to solve the n-Queens problem.

4.2.1 Flowchart

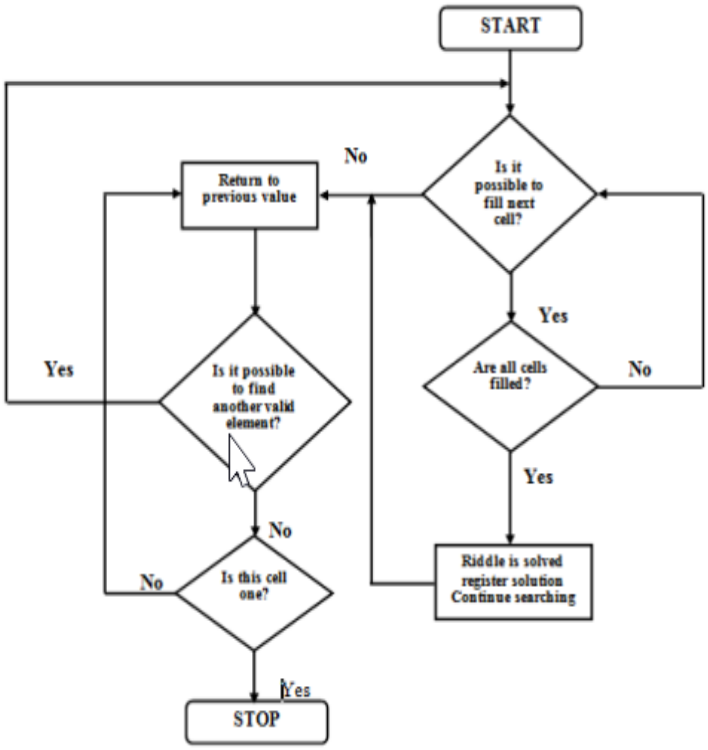


Fig:3 Flowchart for n-Queens problem

4.2.2 Task Description

To solve the n-Queens problem using the backtracking algorithm, we start by placing a queen on any row in the first column. We then place another queen in the second column. Now, we check if the position of queens on the chessboard is valid. If the positions are valid, we continue by placing the queen in the next column whereas if the position is invalid, we backtrack the solution, which means that we remove the queen from the current column and then we find if there is any other row in the column where the queen can be placed in such a way that the positions of queens in all the columns are valid.

4.3 Code

4.3.1 Crossword Puzzle Code

In crossword puzzle, I have pre-defined four different grids and four different word lists (one for each grid). The following files are created for the crossword puzzle.

i. Placement.java

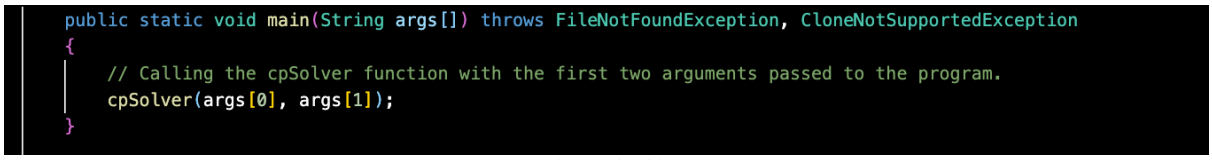
This file can be considered as a helper file as it contains all the classes and methods that are required to solve the crossword puzzle. Connection, Intersection, BestCombination, Combination, BoardSize are few classes that are in this file.

```
// Connection class
class Connection implements Cloneable {

    char character;
    int position;

    Connection(char character, int position){
        this.character = character;
        this.position = position;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException{
        return super.clone();
    }
}

// Intersection class
class Intersection {

    int id;
    int sIndex;
    int dPosition;

    Intersection(int id, int sIndex, int dPosition){
        this.id = id;
        this.sIndex = sIndex;
        this.dPosition = dPosition;
    }

    public String toString() {
        return "[Intersection Id : " + id + " Source index: " + sIndex + " Destination position: " + dPosition+"]";
    }
}
```

1. Connection and Intersection classes in Placement.java

ii. cpcsp.java

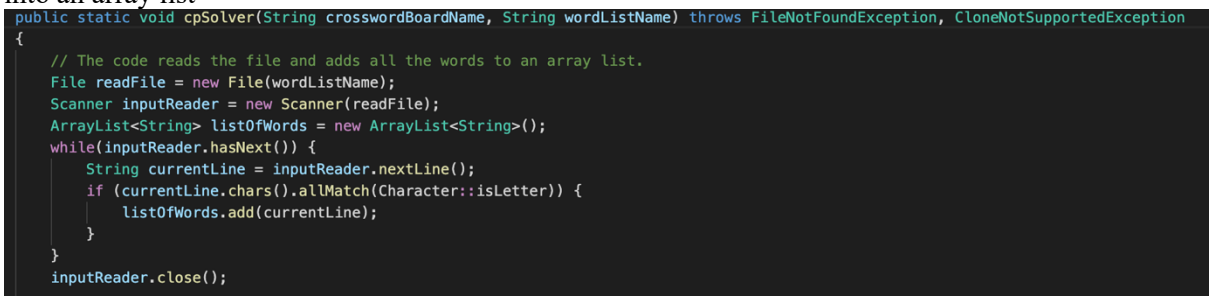
This file contains the main method which is the starting point of execution. This method calls **cpSolver** method that takes the grid file name and word list name as arguments.



```
public static void main(String args[]) throws FileNotFoundException, CloneNotSupportedException
{
    // Calling the cpSolver function with the first two arguments passed to the program.
    cpSolver(args[0], args[1]);
}
```

2. main method in cpcsp.java

This code snippet **cpsolver()** is responsible for reading the crossword file and words and adding them into an array list



```
public static void cpSolver(String crosswordBoardName, String wordListName) throws FileNotFoundException, CloneNotSupportedException
{
    // The code reads the file and adds all the words to an array list.
    File readFile = new File(wordListName);
    Scanner inputReader = new Scanner(readFile);
    ArrayList<String> listOfWords = new ArrayList<String>();
    while(inputReader.hasNext()) {
        String currentLine = inputReader.nextLine();
        if (currentLine.chars().allMatch(Character::isLetter)) {
            listOfWords.add(currentLine);
        }
    }
    inputReader.close();
```

3. Reading the input files

Below code is responsible for determining the horizontal and vertical placements of the words and find the exact intersection between these horizontal and vertical placements and vice versa.

```
// Getting the horizontal and vertical placements of the words.
List<Placement> horizontalPlacements = new ArrayList<Placement>();
List<Placement> verticalPlacements = new ArrayList<Placement>();
getHorizontalPlacements(horizontalPlacements, crosswordBoard, combinationBySize);
getVerticalPlacements(verticalPlacements, crosswordBoard, combinationBySize);

// Finding the intersections between the horizontal and vertical placements.
for (int i = 0; i < horizontalPlacements.size(); i++)
{
    for (int j = 0; j < verticalPlacements.size(); j++)
    {
        int[] charPosition = horizontalPlacements.get(i).intersects(verticalPlacements.get(j));
        if (charPosition != null)
        {
            horizontalPlacements.get(i).crossConnection.add(new Intersection(verticalPlacements.get(j).id, charPosition[0], charPosition[1]));
        }
    }
    horizontalPlacements.get(i).mostConstrainingPlacementHeuristic = horizontalPlacements.get(i).crossConnection.size();
}

// Finding the intersections between the vertical and horizontal placements.
for (int i = 0; i < verticalPlacements.size(); i++)
{
    for (int j = 0; j < horizontalPlacements.size(); j++)
    {
        int[] charPosition = verticalPlacements.get(i).intersects(horizontalPlacements.get(j));
        if (charPosition != null)
        {
            verticalPlacements.get(i).crossConnection.add(new Intersection(horizontalPlacements.get(j).id, charPosition[0], charPosition[1]));
        }
    }
    verticalPlacements.get(i).mostConstrainingPlacementHeuristic = verticalPlacements.get(i).crossConnection.size();
}
```

4. Find horizontal and vertical placements

Below code creates a stack of boards and pushes the initial state onto the stack. It also creates a variable to track number of backtracks and start time.

```
Stack<BoardState> currentboardState = new Stack<BoardState>();
currentboardState.push(initialS);
int backTrack = 0;
char[][] currentCrossWordBoard;
long startTime = System.currentTimeMillis();

while (!currentboardState.isEmpty()) {

    // The code is creating a temporary array list of placements.
    ArrayList<Placement> tempPlacements = new ArrayList<Placement>();
    currentCrossWordBoard = displayPresentState(currentboardState.peek(), crosswordBoard);

    // The code is checking if the current board is solved or not. If it is solved, it prints the
    // solution and breaks out of the loop.
    if (isBoardSolved(currentCrossWordBoard))
    {
        System.out.println("Solution Found:\n");
        // Function to print the crossword board.
        displayBoard(currentCrossWordBoard);
        System.out.println();
        ArrayList<String> hors = new ArrayList<String>();
        ArrayList<String> vers = new ArrayList<String>();
        for (int i = 0; i < currentboardState.peek().placements.size(); i++)
        {
            if (currentboardState.peek().placements.get(i).startXCoord == currentboardState.peek().placements.get(i).endXCoord)
            {
                hors.add(currentboardState.peek().placements.get(i).currentPlacement);
            }
            else
                vers.add(currentboardState.peek().placements.get(i).currentPlacement);
        }
        break;
    }
```

5. Creating stack of boards and backtrack variables

Below snippet performs assignment of words to the cell with least possible words. Also, this code checks if a word can be assigned on the board or not and create a new board in case of no feasibility.

```
// The code is trying to assign a word to the placement with the least number of possible
// words.
for (int i = 0; i < values.size(); i++)
{

    ArrayList<Placement> t1 = new ArrayList<Placement>();

    for (Placement placement : currentboardState.peek().placements)
    {
        t1.add((Placement) placement.clone());
    }
    Placement selected = t1.get(mrvIndex);
    ArrayList<Placement> t2 = new ArrayList<Placement>(t1);
    Collections.sort(t2, (a,b) -> a.id - b.id);
    boolean wordAlignmentPossible = selected.addNeighborConnection(values.get(i), t2, usedWordList);

    // The code is checking if the word can be placed on the board. If it can, it will place the
    // word on the board and add it to the list of words. It will then sort the list of words by the
    // most constraining placement heuristic. It will then create a new board state with the new list
    // of words and push it onto the stack.
    if (wordAlignmentPossible)
    {
        selected.currentPlacement = values.get(i);
        beginStateCopy.get(selected.id).words.add(values.get(i));
        Collections.sort(t1, (a,b) -> b.mostConstrainingPlacementHeuristic - a.mostConstrainingPlacementHeuristic);
        BoardState nextState = new BoardState(t1);
        nextState.selectedWord = selected.currentPlacement;
        currentboardState.push(nextState);
        isStuck = true;
        break;
    }
}
```

6. Checking if a word can be places on the board and creating a new board if no

Below snippet checks the placement status. If it's stuck then it clears all the words from the placement

```
if (!isStuck)
{
    beginStateCopy.get(selectedPlacement.id).words.clear();
    currentboardState.pop();
    backTrack++;
}
```

7. Clearing the words if placement is stuck

This code checks the board state, and prints the solution if it's non-empty

```
if (currentboardState.isEmpty())
{
    System.out.println("Solution doesn't exists");
    System.out.println("Total no. of backTracks: "+backTrack);
    System.out.println("Total Time Taken: " + (endTime - startTime) + "ms");
}
else
{
    System.out.println("Total no. of backTracks: "+backTrack);
    System.out.println("Total Time Taken: " + (endTime - startTime) + "ms");
}
```

8. Print the solution

4.3.2 n-Queens Problem Code

I created the following files to solve the n-Queens problem.

i. index.js

The index.js file is the starting point of execution. When the user executes the code through the command line, index.js takes the arguments (board size) and invokes the **Queens** method to compute the solution and the **printSolution** method to print the solution.

```
JS index.js > ...
1   const LOG = require('./src/logsFormat.js')
2   const Queens = require('./src/queens.js')
3   const printSolution = require('./src/solution.js');
4
5   var myArgs = process.argv.slice(2);
6
7   myArgs.map( e => {
8
9       console.log(`\n${LOG.fg.cyan}**********N-Queens**********${LOG.reset}`);
10      console.log(`Size of Board: ${LOG.fg.white}${e}${LOG.reset}`);
11
12      /* This is a way to measure the time it takes to run the algorithm. */
13      const startTime = Date.now();
14      let queens = Queens(e)
15      const secondsElapsed = (Date.now() - startTime)/1000;
16
17      if(typeof queens !== 'string'){
18          console.log(`Seconds Elapsed: ${LOG.fg.green}${secondsElapsed}${LOG.reset}`);
19
20          let solution = printSolution(queens);
21
22          console.log("\nQueens positions by column:");
23          console.log(queens);
24          console.log(solution);
25      }
26  })
```

1. index.js

ii. solution.js

The solution.js file contains the **printSolution** method. The printSolution method takes the solution (the position of queens) as input and prints the solution on the console. Based on the value in a cell on the chessboard, the appropriate background color is chosen.

```
function printSolution(queens){

    /* Creating an array of the alphabet, and then mapping it to the ASCII code. */
    const alphabetASCII = Array.from(Array(queens.size)).map((e, i) => i + 65);
    const alphabet = alphabetASCII.map((x) => ` ${String.fromCharCode(x)}`);

    /* Creating a variable called isCellColored and setting it to true. Creating an empty array
    called solution. */
    let isCellColored = true;
    let solution = []

    /* Printing to the console. */
    console.log(`${LOG.fg.magenta}---- This is your solution ----${LOG.reset}`);
    console.log(`  `, ...alphabet);

    /* Printing the solution to the console. */
    for(let i=0; i<queens.size; i++)
    {
        /* Creating an iterator for the queens set, creating a new array for the solution, and then
        checking if the size of the queens set is even. If it is, then it is setting isCellColored to
        false. */
        let iterator = queens.values();
        solution[i] = new Array();
        queens.size % 2 == 0 ? isCellColored = !isCellColored : {}

        for(let j = 0; j<queens.size; j++)
        {
            /* Inverting the value of isCellColored. */
            isCellColored = !isCellColored

            /* Checking if the value of the iterator is equal to the value of i. If it is, then it is
            pushing the value of 1 to the solution array. */
            if( iterator.next().value == i )
            {
                solution[i].push(`${LOG.bg.red}${isCellColored ? LOG.fg.black : LOG.fg.white} 1`)
                continue;
            }

            /* Pushing a string to the solution array. */
            solution[i].push(`${isCellColored? LOG.bg.white : LOG.bg.black}${isCellColored? LOG.fg.black : LOG.fg.white} 0`)
        }
        console.log(...solution[i], `${LOG.reset}`);
    }

    /* Returning the solution array. */
    return solution
}
```

2. printSolution method

iii. queens.js

This is an important file in our project as it contains the backtracking code. The following methods are available in the queens.js file.

a. fillableRows

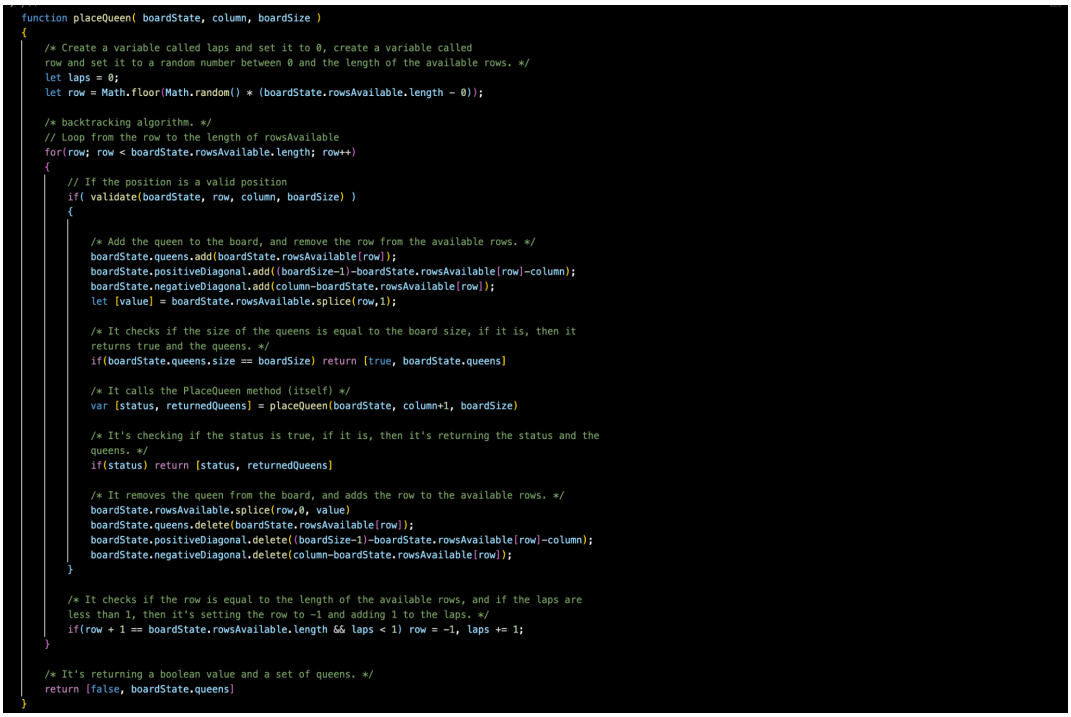
This method takes the board size as input and initializes an array that keeps track of which rows are filled.

```
function fillableRows(boardSize)
{
    let rowsAvailable = new Array();
    for(let index = 0; index < boardSize; index++)
    {
        rowsAvailable.push(index);
    }
    return rowsAvailable
}
```

3. fillableRows method

b. PlaceQueen

This method takes the board state, column, and board size as input. The backtracking algorithm is implemented in this method. The method starts by choosing a random valid row and then the rows are looped from the current row to the last row. For each row, the method checks if the position is valid, if so, adds the row to the solution, and if not, backtracks the solution by removing it from the solution and finds an alternate row.



```
function placeQueen( boardState, column, boardSize )
{
    /* Create a variable called laps and set it to 0, create a variable called
    row and set it to a random number between 0 and the length of the available rows. */
    let laps = 0;
    let row = Math.floor(Math.random() * (boardState.rowsAvailable.length - 0));

    /* backtracking algorithm. */
    // Loop from the row to the length of rowsAvailable
    for(row; row < boardState.rowsAvailable.length; row++)
    {
        // If the position is a valid position
        if( validate(boardState, row, column, boardSize) )
        {

            /* Add the queen to the board, and remove the row from the available rows. */
            boardState.queens.add(boardState.rowsAvailable[row]);
            boardState.positiveDiagonal.add((boardSize-1)-boardState.rowsAvailable[row]-column);
            boardState.negativeDiagonal.add(column-boardState.rowsAvailable[row]);
            let [value] = boardState.rowsAvailable.splice(row,1);

            /* It checks if the size of the queens is equal to the board size, if it is, then it
            returns true and the queens. */
            if(boardState.queens.size == boardSize) return [true, boardState.queens]

            /* It calls the PlaceQueen method (itself) */
            var [status, returnedQueens] = placeQueen(boardState, column+1, boardSize)

            /* It's checking if the status is true, if it is, then it's returning the status and the
            queens. */
            if(status) return [status, returnedQueens]

            /* It removes the queen from the board, and adds the row to the available rows. */
            boardState.rowsAvailable.splice(row,0, value)
            boardState.queens.delete(boardState.rowsAvailable[row]);
            boardState.positiveDiagonal.delete((boardSize-1)-boardState.rowsAvailable[row]-column);
            boardState.negativeDiagonal.delete(column-boardState.rowsAvailable[row]);
        }

        /* It checks if the row is equal to the length of the available rows, and if the laps are
        less than 1, then it's setting the row to -1 and adding 1 to the laps. */
        if(row + 1 == boardState.rowsAvailable.length && laps < 1) row = -1, laps += 1;
    }

    /* It's returning a boolean value and a set of queens. */
    return [false, boardState.queens]
}
```

4. placeQueen method

c. Queens

This method takes the board size as input and checks if the input is valid. In this project, I considered to have a minimum board size of 4. This method calls the placeQueen method which uses a backtracking algorithm and returns the solution.

```
function Queens(boardSize)
{

    /* It checks if the board size is less than 4 and it's different of 1, if it is, then it's
    returning a string. */
    if(boardSize<4 && boardSize != 1) return `Not posible! ${boardSize} < 4 and it's different of 1`

    /* It creates the initial state of the board. */
    let rowsAvailable = fillableRows(boardSize);
    let column = 0
    var negativeDiagonal = new Set(); // [ -|n-1|, n-1 ]
    var positiveDiagonal = new Set(); // [ -|n-1|, n-1 ]
    var queens = new Set(); // Max length = n

    const state = {
        rowsAvailable: rowsAvailable,
        negativeDiagonal,
        positiveDiagonal,
        queens
    }

    /* It calls the placeQueen method. */
    let [status, returnedQueens] = placeQueen(state, column, boardSize)

    /* It checks if the status is false, if it is, then it's returning a string, if it's not, then
    it's returning the queens. */
    return !status ? 'It was not posible' : returnedQueens;
}
```

5. Queens method

iv. logsFormat.js

This file contains a constant with a dictionary of foreground and background colors. These are used in displaying the solution on the console.

```
src > JS logsFormat.js > ...
// Constant for Log Style
const LOG = {
    reset: "\x1b[0m",
    bright: "\x1b[1m",
    dim: "\x1b[2m",
    underscore: "\x1b[4m",
    blink: "\x1b[5m",
    reverse: "\x1b[7m",
    hidden: "\x1b[8m",
    // Foreground (text) colors
    fg: {
        black: "\x1b[30m",
        red: "\x1b[31m",
        green: "\x1b[32m",
        yellow: "\x1b[33m",
        blue: "\x1b[34m",
        magenta: "\x1b[35m",
        cyan: "\x1b[36m",
        white: "\x1b[37m",
        crimson: "\x1b[38m"
    },
    // Background colors
    bg: {
        black: "\x1b[40m",
        red: "\x1b[41m",
        green: "\x1b[42m",
        yellow: "\x1b[43m",
        blue: "\x1b[44m",
        magenta: "\x1b[45m",
        cyan: "\x1b[46m",
        white: "\x1b[47m",
        crimson: "\x1b[48m"
    }
};

module.exports = LOG
```

6. logsFormat.js

5. RESULTS

This section comprises the results for crossword puzzle and n-Queens problem.

5.1 Execution Steps

Unzip the main CS_890BR Project.zip file for running both crossword and n-Queens code.

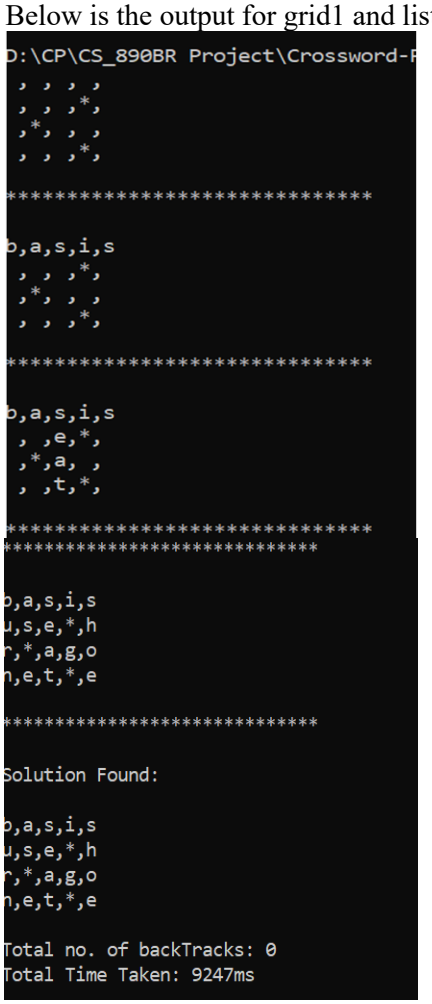
5.1.1 Crossword Puzzle Results

In the terminal, go to the CS_890BR Project\Crossword-Puzzle\src location and execute below commands

- javac *.java
- java cpcsp ../data/grid1.txt ../data/wordList1.txt

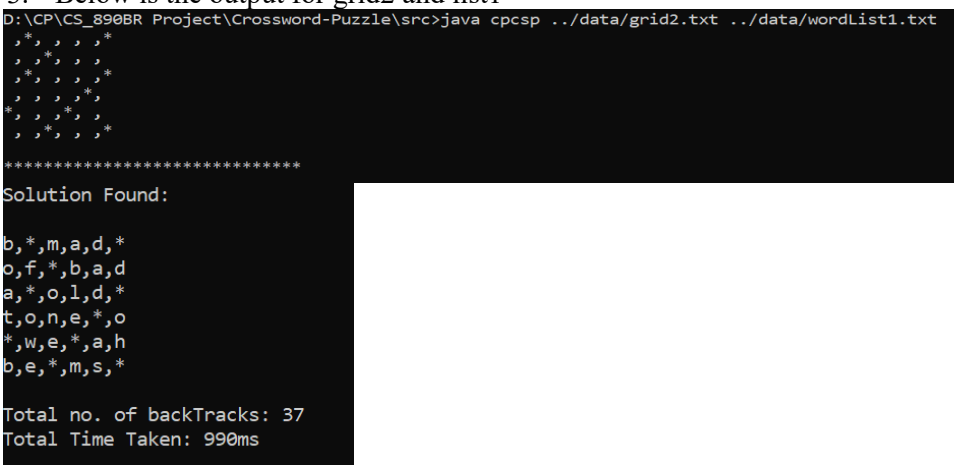
Here we are providing grid and wordslist as input

1. Below is the output for grid1 and list1



2. Below is the output for grid1 and list2

3. Below is the output for grid2 and list1



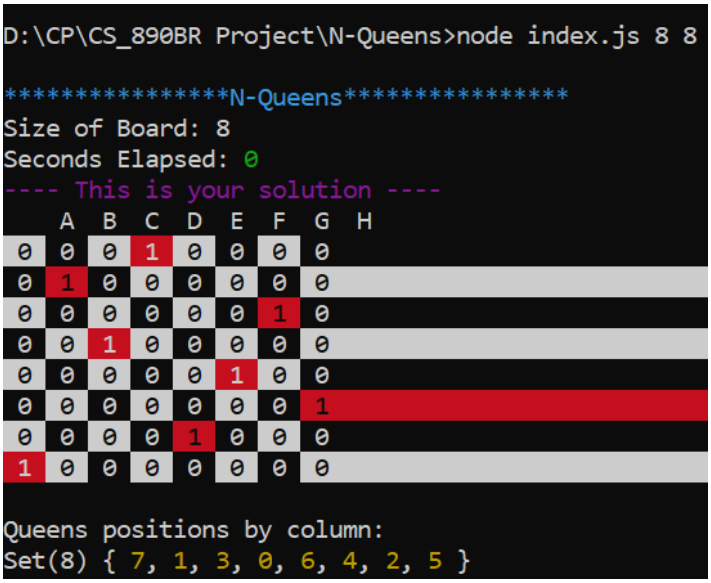
5.1.2 n-Queens Problem Results

In the terminal, go to the CS_890BR Project\N-Queens location and execute below command

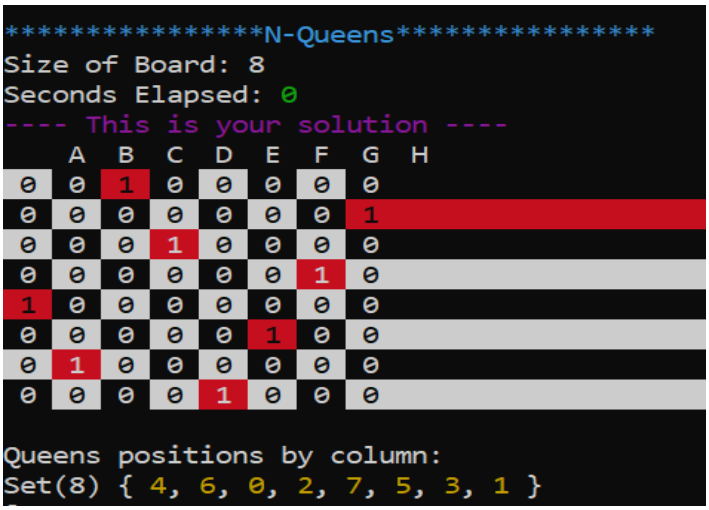
- node index.js 8 x 8

where 8 x 8 is the chessboard size and this can be 4 x 4, 16 x 16, 32 x 32……

1 When the size of the chessboard is 8 x 8, the two solutions are {7, 1, 3, 0 ,6, 4, 2, 5}, {4, 6, 0, 2, 7, 5, 3, 1}



8 x 8 Solution 1



8 x 8 Solution 2

2. When the size of the chessboard is 4 x 4, the two solutions are {2, 0, 3, 1}, {1, 3, 0, 2}

```
D:\CP\CS_890BR Project\N-Queens>node index.js 4 4

***************N-Queens***************
Size of Board: 4
Seconds Elapsed: 0.001
---- This is your solution ----
    A  B  C  D
 0  1  0  0
 0  0  0  1
 1  0  0  0
 0  0  1  0

Queens positions by column:
Set(4) { 2, 0, 3, 1 }
```

4 x 4 Solution 1

```
***************N-Queens***************
Size of Board: 4
Seconds Elapsed: 0.001
---- This is your solution ----
    A  B  C  D
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0

Queens positions by column:
Set(4) { 1, 3, 0, 2 }
```

4 x 4 Solution 2

6. CONCLUSION

The main goal of this project is to solve the crossword puzzle and n-queens problem given different input values. Through this project, I was able to implement crossword puzzle with different combinations of grid size and various words list. The code successfully takes grid and wordlist as input and displays the possible solution with the number of backtracks and time. I have used arc consistency and forward checking techniques in order to achieve the result for crossword puzzle.
For n-Queens, I am using a powerful brute force approach – backtracking in order to find the solution. Based on given board size, this n-Queens program correctly displays the solutions i.e., placing the queens on the board such that no two queens attack each other.

7. REFERENCES

1 Apt, K. (2010). Principles of constraint programming (3rd ed.). Cambridge: Cambridge University Press.