Algorithmics	Student information	Date	Number of session
	UO: 295180	01/05/24	7
	Surname: Orviz Viesca		
	Name: Mario		

Escuela de Ingeniería Informática
Universidad de Oviedo

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Branch and bound algorithms

The branch and bound algorithms consist of a breath-first search in a tree of possible solutions and states of a problem, pruning the branches of the tree as soon as possible when we know for sure that they would not give us the correct solution for the problem.

This problem is not solved using recursion as the backtracking one. Instead, the main program that will perform the execution of the algorithm (NumericSquareBaB.java) has a priority queue (a heap) where the different states of the execution will be stored.

This priority queue of states will be sorted depending on the heuristic of each state, putting the ones with the best heuristic at the beginning of the queue and the ones with the worst at the end of the queue.

As we need to determine a valid heuristic and, we cannot predict if a value is more valid than other one as we have divisions and multiplications (for example, if we only had additions, we can predict that a row value will probably give their corresponding value considering the current value and the empty spaces). Therefore, the heuristic used is that the states with the fewer empty spaces will have better heuristic value than the ones with more empty spaces.

To solve this problem applying a branch and bound algorithm, first a criteria for pruning the branches needs to be established. In this specific case, we cannot apply an heuristic formula each time we have a new node (maybe the only way to prune an uncompleted row is looking if there're any divisions, the only operation that would decide if a branch is valid or not before its completion as we don't allow unprecise divisions like 9/4), so the criteria to prune a branch is applied each time a row or a column is completed.

Approach:

Coords:

Algorithmics	Student information	Date	Number of session
	UO: 295180	01/05/24	7
	Surname: Orviz Viesca		
	Name: Mario		

To keep track of the execution point where the algorithm is, a new java class named Pair was created, and each node will have an instance of this class storing the coordinates of the row and column. The implementation is:

```
public class Pair<T1, T2> {
    public final T1 x;
      public final T2 y;
      public Pair(T1 x, T2 y) {
        this.x = x;
        this.y = y;
      }
}
```

Each time that a recursive call is done by the method that performs the branch and bound, a new object Pair<Integer, Integer> is created and the method *translateCoords(index)* will return the new instance with the corresponding x and y coordinates.

```
public Pair<Integer, Integer> translateCoords(int value){
        int rowCord = value/size;
        int colCord = value%size;
        return new Pair<Integer, Integer>(rowCord,colCord);
}
```

Data organization

All the numbers are feed into an integer matrix called variable data with the values -1 for the mutable values and the corresponding values for the immutable ones, and the row and col operators are stored in two different string matrices called rowOperators and colOperators.

Then, a boolean matrix with the same dimensions as the integer matrix will contain the indexes where an immutable value is stored.

All these data is obtained from a string matrix that is the original data for the problem.

Then, the class State is created to store states of the problem in the queue. It works like a "wrapper" for each instant of the execution of the algorithm. Therefore, for each valid state of the board a new State equivalent to a node in the theory of the branch and bound algorithm is crated and pushed into the queue.

Algorithmics	Student information	Date	Number of session
	UO: 295180	01/05/24	7
	Surname: Orviz Viesca		
	Name: Mario		

Thusi class State.java has as attributes and integer matrix containing the values of the list at a given instant of time and the index of the following execution. Note that the queue will never store a matrix with all the values covered as it's redundant. We can prune those values before inserting them into the queue, as they're whether a solution or an invalid state.

The State class also provides a getHeuristic() method returning an int indicating the amount of unfilled values in the matrix (those set to -1)

```
public int getHeuristic() {
    int heuristic = 0;
    for(int i = 0; i < matrixStatus.length; i++) {
        for(int j = 0;  j < matrixStatus.length; j++) {
            if(matrixStatus[i][j] == -1) heuristic++;
        }
    }
    return heuristic;
}
```

Branch and bound solution

The Branch and bound algorithm is performed by the method branchAndBound, that has a public safe interface for the user and an internal version, that is managed internally to prevent errors.

The public safe branch and bound method is the one in charge of generating the first 10 sates for the first mutable place in the matrix. For this reason, the first thing to be done by this algorithm is to search for the first mutable value in this matrix. When it finds it, using a for loop it generates the first 10 states of that position and push them into the queue.

Then, the program enters a while loop that will iterate until a valid solution is found or the queue is empty (no solution exists for the original board). Inside of this wile method, the state at the head of the queue is obtained and then a call to the private version of branch and bound is performed.

Branch and bound private method

First of all, this method calls the function translateCoords, with parameter the index that it receives. When it obtains its corresponding Pair object, it does a safe copy of the number matrix. The boolean variable isImmutable is used to determine if the value at the position corresponding with that state is mutable or not. The treatment of mutable or immutable values would be more or less the same, but the mutable values will repeat the execution 10 times while the immutable ones will only execute twice.

The method enters a loop from 0 to 9, and, if it's immutable, it breaks the loop automatically. If it's mutable, it continues, updating the value of the current position by the i value of the loop.

If the x component of the Pair object corresponds to the last row of the board, it means that the column is filled by numbers and we can apply one pruning condition, that is checking if the corresponding mathematical operation of the column gives the right result. This checking is performed by the method checkCol, that gets the current state of the matrix and the corresponding column index. If the column is alright, two situations can be met: The y component of the Pair object corresponds to the index of the last column of the graph (we're at the bottom right corner and therefore the whole matrix is filled with number) or not.

In the first case scenario, we need to perform the final checking of the last row with the method checkRow, that, as the checkCol method, validates that the row is correct. If this row is correct, the algorithm has found a valid solution and will store the solution to the solution matrix. Then, the global variable found solution is set to true and therefore, the while on the previous method will break.

In the second case scenario, a new state object is created with the copy of the board and the index of the **next position to be evaluated**, and it's added to the queue.

Outside of this conditional statement, we can find other two scenarios: The component y of the Pair object corresponds to the last column index (the algorithm reached the last element of a row) or not (the algorithm is in a position that is not the end of a row or the end of a column)

Algorithmics	Student information	Date	Number of session
	UO: 295180	01/05/24	7
	Surname: Orviz Viesca		
	Name: Mario		

In the first case scenario, we can apply the pruning condition of checking if the row gives us the expected output or not. If it does not give us the correct solution for that row, we can prune that branch avoiding useless iterations and create the new one incrementing again the value at that position. If the row is evaluated to true, a new state object is created with the copy of the board and the index of the **next position to be evaluated**, and it's added to the queue**.**

In the second case scenario, the algorithm is in a location from where we cannot check or prune anything, so just again, a new State object is created and added to the queue.

In the case of an immutable number, the algorithm behaves the same way but outside the loop. That means, it performs the same checking (in case we have an immutable value at the end of a row or column) and just continue with the execution of the algorithm.

```
private void branchAndBound(int[][] values, int index) {
    Pair<Integer, Integer> coords = translateCoords(index);
    int[][] safeCopy = copyOf(values);
    boolean isInmutable = false;
    for(int i = 0; i < 10; i++) {
        if(inmutables[coords.x][coords.y]) {
            isInmutable=true;
            break;
        }
        safeCopy[coords.x][coords.y] = i;
        if(coords.x==size-1) {
            if(checkCol(safeCopy, coords.y)) {
                if(coords.y==size-1) {
                    if(checkRow(safeCopy, coords.x)) {
                        copySolution(safeCopy);
                        foundSolution = true;
                    }
                } else {
                    statesHeap.add(new State(safeCopy, index+1));
                }

            }
        }
        else if(coords.y==size-1) {
            if(checkRow(safeCopy, coords.x)){
                statesHeap.add(new State(safeCopy, index+1));
            }
        }
        else {
            statesHeap.add(new State(safeCopy, index+1));
        }
    }
```

Algorithmics	Student information	Date	Number of session
	UO: 295180	01/05/24	7
	Surname: Orviz Viesca		
	Name: Mario		

```
    if(isInmutable&&coords.x!=size-1&&coords.y!=size-1) {
        statesHeap.add(new State(safeCopy, index+1));
    }else if(isInmutable&&coords.x==size-1) {
        if(checkCol(safeCopy, coords.y)) {
            if(coords.y==size-1) {
                if(checkRow(safeCopy, coords.x)) {
                    copySolution(safeCopy);
                    foundSolution = true;
                }
            }
            else {
                statesHeap.add(new State(safeCopy, index+1));
            }
        }
    }else if(isInmutable&&coords.y==size-1) {
        if(checkRow(safeCopy, coords.x)){
            statesHeap.add(new State(safeCopy, index+1));
        }
    }
}
```

Heuristic and pruning conditions used.

In this type of problem, I think that the only way to check the quality of a node is to give more priority to those nodes that has fewer unknown numbers, so the nodes with the greatest number of numbers established that have passed the previous prune are the ones more favourable to give us a valid solution.

In my implementation of the problem, a node is only stored in the queue if it has a chance to solve the problem. No node with an invalid row or column will be stored as this will be checked always before pushing a new node into the queue.

Algorithmics	Student information	Date	Number of session
	UO: 295180	01/05/24	7
	Surname: Orviz Viesca		
	Name: Mario		

Time measurements

Test case	*Time for first solution (backtracking) [ms]*	*Number of developed nodes (backtracking)*	*Time for first solution (branch and bound) [ms]*	*Number of developed nodes (branch and bound)*
Test00	LoR	16	LoR	20
Test01	LoR	3479	26.2	2507
Test02	LoR	1265	LoR	287
Test03	12.6	672658	68	57148
Test04	14.9	884701	83	85007
Test05	16.8	1143116	1076	1275054
Test06	LoR	17167	139	173574
Test07	50.2	2673799	629	587829

The algorithm that worked better is the backtracking one. I think that this is due to the heuristic used in the branch and bound algorithm. I think that this type of problem does not allow us to use an heuristic that makes the algorithm faster than the backtracking one, just counting the unknown spaces is not enough.

But, as it can be seen, in general the branch and bound algorithm tends to generate less nodes (in some cases), but maybe the time to add them to que queue and the non-recursive approach makes the execution slower.