AI Methods N-Queens BFS Coursework

# 1.) a.)

## Completeness

A search algorithm is “complete” if it is guaranteed to find the “goal state” whenever the input has a “goal state” at some point within its branches.

## Time Complexity

The maximum time taken/operations ran, before a search technique gets a solution to a problem. This is calculated in big O notation due to it being different based on the number of inputs, the hardware it is ran on, etc. This allows us to get a comparison between search techniques without having to physically test every one.

## Space Complexity

The amount of states that have to be stored in order for the goal state to be found or the amount of states become exhausted. We express this as the worst possible case in big O notation for the same reasons as in the Time Complexity explanation.

## Optimality

A search algorithm is “optimal” if it will always find the “goal state” that has the shortest (lowest cost) path from the root node out of all the possible solutions. This generally means that the goal state has the least child states before it gets to the root state.

# 1.) b.)

Key  
**b = branching factor m = maximum depth d = depth of goal state**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Completeness | Time Complexity | Space Complexity | Optimality |
| a. BFS | True | O(bd) | O(bd) | True |
| b. DFS | False | O(bm) | O(bm) | False |
| c. A\* Search | True | O(bd) | O(bd) | True |

## Completeness

BFS and A\* search will always find a “goal state” if one exists, and so they are “complete” algorithms, this is because they search systematically through every “state” on that level, before moving to the next level down in every branch. DFS however, is not a “complete” algorithm as in the case that the depth of one of the branches is infinitely long, and there is no “goal state” in that branch, it will never get out of that branch in order to find the “goal state”, and will therefore never complete.

## Time Complexity

BFS and A\* search both have the same time complexity, in fact, everything about them in the table above is the same. How they work is different, however, as BFS will simply search through every “state”, checking whether it is a suitable “goal state”. It will do this until it finds a goal state, at which point it will return that state and stop searching. If there was only one state that matched the goal in the tree, then both BFS and A\* search would have to go through every possible state to find it, hence why it has an O(bd) time complexity. DFS has a different time complexity of O(bm) as it must go to the deepest part of the branch before it can try another branch. This means that it will take a much longer time than BFS or A\* search in a lot of cases, especially when the branch is extremely deep.

## Space Complexity

BFS and A\* search both have the same space complexity due to the fact that you need to store every possible solution until you find a state that matches the “goal state”. You have to do this due to the fact that once you finish going across a all the nodes on that level, you have to move down and continue every branch from where you were. This means that you are storing a lot of solutions in memory. The advantage with DFS over BFS and A\* for space complexity is that it completes an entire branch before moving on and you can therefore discard those states as you will never need to continue on from them. This makes it very useful in a case where you have limited amounts of memory available to you.

## Optimality

Both the BFS and A\* searches are “optimal”. BFS is optimal because it will search all possible states on each level of the tree before moving down to the next level. This means that when it finds a “goal state”, this must be an “optimal” state as there were no goal states on the previous levels and so the only other possible goal states would be on the same level or a deeper level of the tree. A\* search does the same as BFS, apart from it estimates the “heuristic cost” of each move to a new state before making that move and decides what the best route to take is based on that. This could cause issues, but due to the fact that A\* search does not overestimate the heuristic cost and can therefore only underestimate or get it correct, it will still evaluate all feasible solutions and so will therefore still find the most optimal solution still. DFS is not “optimal” as it will go to the bottom of every branch before moving to the next branch and once it finds a solution it will stop there. This means that the provided solution could even be the worst possible solution and DFS would not care, therefore this is not an optimal search.

# 2.)

## Algorithm

Below I have included the source code for the BFS algorithm that I implemented. This is directly copied from my source code.

**public** State bfs(State rootNode) {

*//Stores all of the current states that need to be processed and have their children extracted.*

Queue<State> queue = **new** LinkedList<>();

*//Add the root node to the queue.*

queue.add(rootNode);

*//Continually loop until the queue is empty.*

**while**(!queue.isEmpty()) {

*//Get the next state in the queue, store it in a variable and remove that state from the queue.*

State state = queue.remove();

*//Get all of the child states the current state has and loop over them.*

**for** (State child : state.getChildSolutions()) {

*//Check if the child state is the goal state.*

**if**(child.isGoal()) {

*//If the child state is the goal state, end the BFS there and return that state.*

**return** child;

}

*//Checks if the child state is acceptable.*

**if**(child.isAcceptable()) {

*//If the child state is an acceptable state, then add that state to the queue.*

*//This ensures that we aren't continually wasting time going*

*//over states that will never get us to the goal state.*

queue.add(child);

}

}

}

**return** **null**;

}

## N-Queens n=4 Trace Output

The trace below is every state that was an acceptable state, one that had no conflicts.

[0, -1, -1, -1]  
[1, -1, -1, -1]  
[2, -1, -1, -1]  
[3, -1, -1, -1]  
[0, 2, -1, -1]  
[0, 3, -1, -1]  
[1, 3, -1, -1]  
[2, 0, -1, -1]  
[3, 0, -1, -1]  
[3, 1, -1, -1]  
[0, 3, 1, -1]  
[1, 3, 0, -1]  
[2, 0, 3, -1]  
[3, 0, 2, -1]

## Code

Below is the entirety of my source code for the project.

### Main.java

**package** pro.zackpollard.nqueensbfs;

**public** **class** Main {

*/\*\**

*\* This is just a standard way of getting out of the static way of things,*

*\* as it is not OOP and just causes more issues than it is worth.*

*\**

*\* @param args The arguments provided when the startup command was ran (unused by this program).*

*\*/*

**public** **static** **void** main(String[] args) {

*//Create a new NQueens object to run the program.*

**new** NQueens();

}

}

### NQueens.java

**package** pro.zackpollard.nqueensbfs;

**import** **java.util.Scanner**;

*/\*\**

*\* @author Zack Pollard*

*\*/*

**public** **class** NQueens {

*/\*\**

*\* This constructor simply allows the Main class and main method run the rest of the program.*

*\*/*

**public** NQueens() {

*//Creates a new Scanner which is used for getting input from the console.*

Scanner scanner = **new** Scanner(System.in);

*//Variable for use later.*

String line;

*//Simple print to tell the user what we want them to input.*

System.out.print("Please enter the number of queens you would like: ");

*//This loop will run until the input is "quit" or "exit". Each time the loop runs it grabs the next line*

*//of input from the console and runs the code inside the loop against that.*

**while**((line = scanner.nextLine()) != **null** && line != "quit" && line != "exit") {

*//Used to store the n value entered by the user.*

**int** amount;

**try** {

*//Parse the string value to an integer.*

amount = Integer.parseInt(line);

} **catch**(NumberFormatException e) {

*//Error if the string entered was not a number and ask for input again.*

System.err.println("The provided number was not an integer.");

System.out.print("Please enter the number of queens you would like: ");

**continue**;

}

*//Checks that the value of n provided is within the bounds that were specified by the specification.*

**if**(amount < 11 && amount > 3) {

*//Creates a new ProblemCalculator object, passing the value of n that was*

*//provided and then telling the calculator to run.*

**new** ProblemCalculator(amount).run();

}

*//Print out an empty line to keep the console a little cleaner and then ask for another input.*

System.out.println();

System.out.print("Please enter the number of queens you would like: ");

}

}

}

### ProblemCalculator.java

**package** pro.zackpollard.nqueensbfs;

**import** **java.util.LinkedList**;

**import** **java.util.Queue**;

*/\*\**

*\* @author Zack Pollard*

*\*/*

**public** **class** ProblemCalculator {

**private** **final** **int** amount;

**private** **final** Queue<State> queue;

*/\*\**

*\* This constructor is used by the NQueens class in order to create a*

*\* ProblemCalculator based on the n that was entered by the user in the console.*

*\**

*\* @param amount*

*\*/*

**public** ProblemCalculator(**int** amount) {

*//Sets the objects amount variable to the provided n.*

**this**.amount = amount;

*//Initialises the queue variable by creating a new LinkedList object (an ordered list)*

**this**.queue = **new** LinkedList<>();

}

*/\*\**

*\* This method is called in order to start the problem calculator running.*

*\*/*

**public** **void** run() {

*//Creates a new root node with the provided n.*

State rootNode = **new** State(amount);

*//Runs the bfs method and stores its solution in a variable.*

State state = **this**.bfs(rootNode);

*//Prints out the solution and all of its parent states.*

state.printAllStates();

}

*/\*\**

*\* This will run the breadth first search algorithm on the provided solution.*

*\* The solution provides the goal state, so this solution should work generically*

*\* with any problem, not just N-Queens.*

*\**

*\* @param rootNode*

*\* @return*

*\*/*

**public** State bfs(State rootNode) {

*//Stores all of the current states that need to be processed and have their children extracted.*

Queue<State> queue = **new** LinkedList<>();

*//Add the root node to the queue.*

queue.add(rootNode);

*//Print the root node array.*

rootNode.printArray();

*//Continually loop until the queue is empty.*

**while**(!queue.isEmpty()) {

*//Get the next state in the queue, store it in a variable and remove that state from the queue.*

State state = queue.remove();

*//Get all of the child states the current state has and loop over them.*

**for** (State child : state.getChildSolutions()) {

*//Check if the child state is the goal state.*

**if**(child.isGoal()) {

*//Print the goal state array.*

child.printArray();

*//If the child state is the goal state, end the BFS there and return that state.*

**return** child;

}

*//Checks if the child state is acceptable.*

**if**(child.isAcceptable()) {

*//If the child state is an acceptable state, then add that state to the queue.*

*//This ensures that we aren't continually wasting time going*

*//over states that will never get us to the goal state.*

queue.add(child);

*//Print the state array.*

child.printArray();

}

}

}

**return** **null**;

}

}

### State.java

**package** pro.zackpollard.nqueensbfs;

**import** **java.util.\***;

*/\*\**

*\* @author Zack Pollard*

*\*/*

**public** **class** State {

*//This is an array of all the positions, the array position is the X co-ordinate on the chess board*

*//and the value in the array at that position is the Y co-ordinate on the chess board.*

**private** **int**[] positions;

*//The size of the chess board.*

**private** **final** **int** n;

*/\*\**

*\* This constructor is used to create a new root node to start the search with.*

*\**

*\* @param n The size of the chess board.*

*\*/*

**public** State(**int** n) {

*//Sets the objects n variable to the provided n.*

**this**.n = n;

*//Initiates the int array to the correct length.*

positions = **new** **int**[n];

*//Loops through the entire array and sets all values to -1 rather than 0.*

**for** (**int** i = 0; i < positions.length; i++) positions[i] = -1;

}

*/\*\**

*\* This is used for creating the child states, which is why it is a private constructor.*

*\* It is used by the #getChildSolutions() method within this file.*

*\**

*\* @param n The size of the chess board.*

*\* @param positions The positions of the queens on the new state.*

*\*/*

**private** State(**int** n, **int**[] positions) {

*//Sets the objects n variable to the provided n.*

**this**.n = n;

*//Sets the objects positions variable to the provided positions.*

**this**.positions = positions;

}

*/\*\**

*\* This method is used to print out this state and all of its parent states.*

*\* It prints a series of chessboards with the positions of the queens to the console.*

*\*/*

**public** **void** printAllStates() {

*//Increments an int 'max' on each iteration in order to limit the print from the root object to n.*

**for**(**int** max = 0; max < positions.length; ++max) {

*//Prints a blank line for separation.*

System.out.println();

*//Increments an int 'x' on each iteration in order to print all rows.*

**for**(**int** x = 0; x < positions.length; ++x) {

*//Gets the position of the queen in the current row.*

**int** position = positions[x];

*//Increments an int 'y' on each iteration in order to print all columns.*

**for**(**int** y = 0; y < n; ++y) {

*//Checks if the position of the queen equals current 'y' and the current row is less than 'max'*

**if**(position == y && x <= max) {

*//Print "Q " to signify the presence of a queen on the board.*

System.out.print("Q ");

*//If the space is not the location of a queen.*

}**else**{

*//Print "- " to signify there is no queen in that position.*

System.out.print("- ");

}

}

*//Prints a blank line for separation.*

System.out.println();

}

}

}

*/\*\**

*\* This method prints a chessboard to console representing this state.*

*\*/*

**public** **void** printPositions() {

*//Prints a blank line for separation.*

System.out.println();

*//Increments an int 'x' on each iteration in order to print all rows.*

**for**(**int** x = 0; x < positions.length; ++x) {

*//Gets the position of the queen in the current row.*

**int** position = positions[x];

*//Increments an int 'y' on each iteration in order to print all columns.*

**for** (**int** y = 0; y < n; ++y) {

*//Checks if the position of the queen equals current 'y'*

**if**(position == y) {

*//Print "Q " to signify the presence of a queen on the board.*

System.out.print("Q ");

*//If the space is not the location of a queen.*

}**else**{

*//Print "- " to signify there is no queen in that position.*

System.out.print("- ");

}

}

*//Prints a blank line for separation.*

System.out.println();

}

}

*/\*\**

*\* Prints the positions array to the console.*

*\*/*

**public** **void** printArray() {

*//Converts the positions array to a string, and prints it to the console.*

System.out.println(Arrays.toString(positions));

}

*/\*\**

*\* This method is used to check whether the state is acceptable up to this point. This means that*

*\* the searching algorithm can remove any states that are not acceptable immediately rather than waiting until*

*\* the very end of the search to check whether they meet the goal state.*

*\**

*\* @return True if the state is acceptable, false otherwise.*

*\*/*

**public** **boolean** isAcceptable() {

*//Variable to store the current row.*

**int** current = 0;

*//Loops through the positions to find the first -1 or hit 'n', whichever comes first.*

**while**(current < n && positions[current] != -1) {

*//Adds one to the 'current' variable.*

++current;

}

*//Subtracts one from current to go from an index of 1 to an index of 0 to match how arrays work.*

--current;

*//Increment through all of the rows from 0 to 'n'.*

**for**(**int** i = 0; i < n; ++i) {

*//Check that the selected row isn't the same as the current row.*

*//Also check that a queen has been added to the selected row.*

**if**(i != current && positions[i] != -1) {

*//Check if the position in the column of the currently selected row*

*//is the same as the position of the current row in the column.*

**if**(positions[i] == positions[current]) {

*//Return false as this state isn't acceptable.*

**return** **false**;

*//Check to see if the selected queen is on a diagonal with the current queen.*

} **else** **if**(Math.abs((current) - i) == Math.abs(positions[current] - positions[i])) {

*//Return false as this state isn't acceptable.*

**return** **false**;

}

}

}

*//Return true as all checks have passed successfully.*

**return** **true**;

}

*/\*\**

*\* This method is used to check whether the state is a valid "goal state". This simply checks whether*

*\* the positions array has a position for the last queen, and if it does, checks that the solution #isAcceptable()*

*\* and then return accordingly.*

*\**

*\* @return True if the state is a "goal state", false otherwise.*

*\*/*

**public** **boolean** isGoal() {

*//Check if the last row has a queen in it and if it does check if the state is acceptable.*

**if**(positions[n - 1] != -1 && isAcceptable()) {

*//Return true as this is a goal state.*

**return** **true**;

}

*//Return false as this is not a goal state.*

**return** **false**;

}

*/\*\**

*\* This method is used to get all possible children of the current state. This will return a Set of the*

*\* child states.*

*\**

*\* @return A Set of the possible child states or null if no child states were found.*

*\*/*

**public** Set<State> getChildSolutions() {

*//Create a variable of type Set which contains State objects.*

*//Initiate that Set as a LinkedHashSet (an set which maintains the order elements are entered in).*

Set<State> children = **new** LinkedHashSet<>();

*//Create a variable for the current row.*

**int** row = 0;

*//Loops through the positions to find the first -1 or hit 'n', whichever comes first.*

**while**(row < n && positions[row] != -1) {

*//Adds one to the 'row' variable.*

++row;

}

*//Check whether the row is the final row and if it is whether that row already has a queen in it.*

**if**(row - 1 == n && positions[row] != -1) {

*//Return null if this is the case as this state will have no children.*

**return** **null**;

}

*//Increment through all of the rows from 0 to 'n'.*

**for**(**int** i = 0; i < n; ++i) {

*//Make a copy of the array so that we can edit it without editing the original.*

**int**[] childPositions = positions.clone();

*//Set the next free row to the currently set column ('i').*

childPositions[row] = i;

*//Add this child to the set of child states.*

children.add(**new** State(n, childPositions));

}

*//Return the set of child states.*

**return** children;

}

}