Problem set 1

Breadth-First Search

|  |  |
| --- | --- |
| GOAL Reached first | G2 |
| Expanded states | {S,A, B, F, C, D, E, G2} |
| OPEN List | {G1} |
| CLOSE List | {S, A, B, F, C, D, E} |

Depth-First Search

|  |  |
| --- | --- |
| GOAL Reached first | G1 |
| Expanded states | {S, A, C, D, G1} |
| OPEN List | {F, B} |
| CLOSE List | {S, A, C, D} |

Best-First Search

|  |  |
| --- | --- |
| GOAL Reached first | G1 |
| Expanded states | {S, A, D, G1} |
| OPEN List | {B, C, E} |
| CLOSE List | {S, A, D} |

A\*

|  |  |
| --- | --- |
| GOAL Reached first | G1 |
| Expanded states | {S, A, C, D, G1} |
| OPEN List | {F, B} |
| CLOSE List | {S, A, C, D} |

SMA\*

|  |  |
| --- | --- |
| GOAL Reached first | G1 |
| Expanded states | {S, A, C, D, G1} |
| OPEN List | {F, B} |
| CLOSE List | {S, A, C, D} |

“Three” or “3”: 3-liter jug initially empty

“Four” or “4”: 4-liter jug initially empty

Tap: T to pull water from

Drain: D to dump water into

A)

State: Details on the amount of water in each jug.

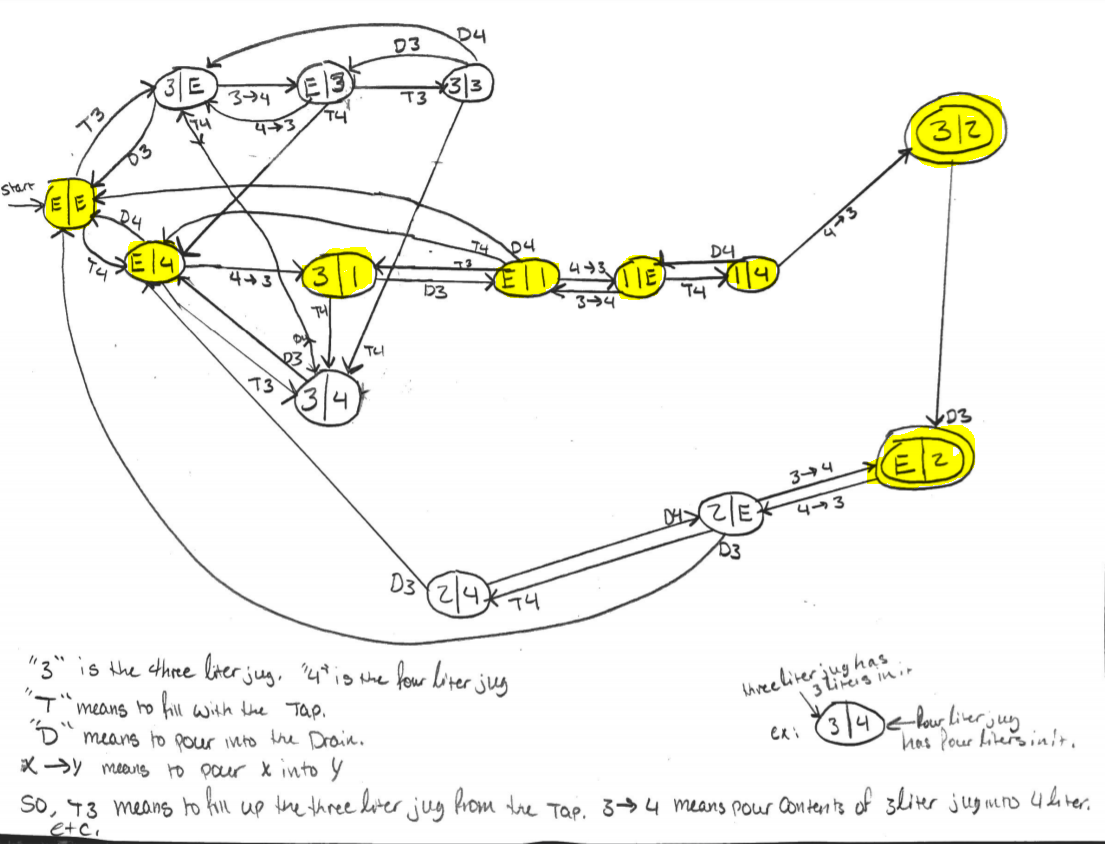
Start State: Both jugs “three” (“3”) and “Four” (“4”) are empty.

Goal condition: Have precisely **two (2)** liters of water in “Four” (“4”).

Operators/Schema:

* No more than three (3) liters of water can be added to “three” (“3”)
* No more than four (4) liters of water can be added to “four” (“4”)
* We fill and empty the jugs into the drain or into the other jug and from the tap or from the other jug perfectly, no water is spilled.
* No measuring tools are available
* If “Four” (“4”) has **two (2)** liters of water in it, we’ve reached our goal state and are done

B) State Space:



The path to the goal state is yellow or gray depending on if the color printing is working.

* “3” is the three-liter jug
* “4” is the four-liter jug
* “T” means to fill with the tap
* “D” means to pour into the drain
* X->Y means to pour X into Y

So, T3 means to fill up the three-liter jug (“3”) from the tap. 3->4 means to pour the contents of “3” into “4”

Example:

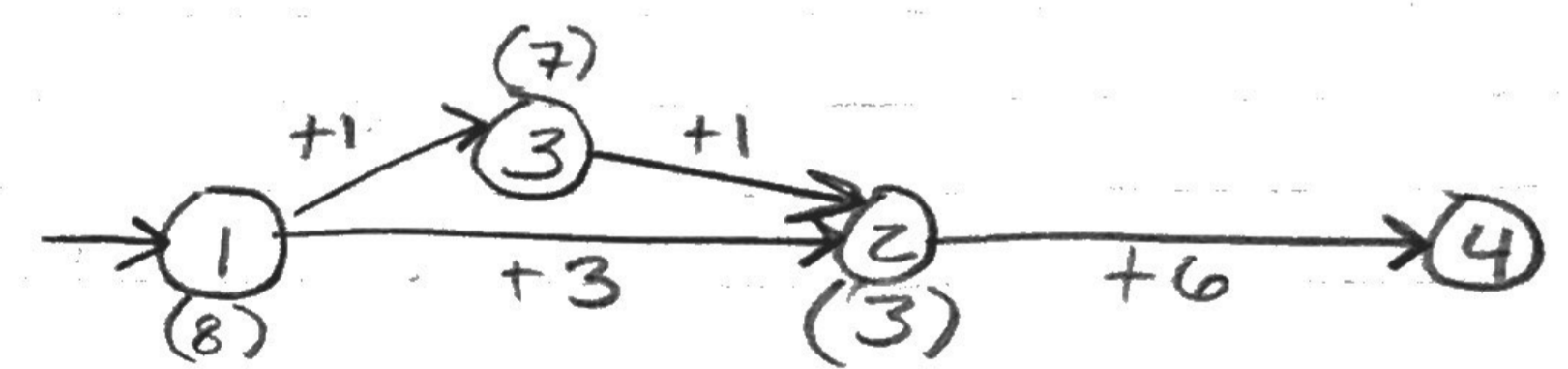
(3|4) <- the 4-liter jug, “4” is on the right and is full because 4/4.

^the 3-liter jug, “3”, is on the left and is full because 3/3

(3|4) –D3🡪(0|4)

In the transition above, we start with both jugs being full. We pour “3” into the drain and are then left with (0|4) i.e. 3 is empty and four is full.

1. For an evaluation function h for A\* to be good, it must be admissible. For it to be admissible, the estimated cost must be less than or equal to the actual cost of getting to the goal state. It must be less than or equal to the actual cost of getting to the goal because otherwise the algorithm could overlook the optimal solution because of overestimation.
2. It will not always find the optimal solution because the heuristic can overestimate, thus it is no longer admissible.



Here, 1->2 would be chosen over 1->3 because when adding the h costs, 1 -> 2’s 6 cost is less than 1->3’s 7 cost. We can then reach the goal by 2->4 for a cost of 6 and overall cost of 9. However, if we choose to go 1->3->2->4 we get a cost of 8, 1+1+6. Thus we see 1->3 is being overestimated in its actual cost.

1. If h(1) and h(2) are both admissible heuristics for A\* then h(x) must also be admissible. h(x) will be a better choice than h(1) or h(2) individually because it chooses the closer estimate to the true cost between the two of them, as both are admissible and thus cannot overestimate.

4. I ran out of time to complete number 4. However, with more time I would have first solved the minimization problem for the minimum of 

Pseudocode:

Int x, y;

Function FindMin(int x1, y1, x2, y2, sp, z)

Case = (current case 1 or 2)

For(int I = 0; I < 33; i++){

Switch statement

Case 1:

Run FindMin with the following parameters:

x1 = 2.9

Y1 = -2.5

X2 = 4.2

Y2 = 0

p =30

z = 0.03

case 2:

Run FindMin with the following parameters:

x1 = 3.2

y1 = 3.2

x2 = -2

y2 = 0

p = 120

z = 0.1

Then record the results

Then once more for the final run

Record those results

5.

* Brief description of the strategy you used to solve the CSP

I used Graph Coloring[[1]](#footnote-1) and backtracking to solve the CSP problem. I pulled the information found in the CSV files into lists for organization and eventually into arrays to be used in functions.

* Pseudo code of your CSP solver

Pull the CSV files into a usable form with lists and arrays.

Boolean colorCheck(State, color, AdjacencyList, colorAnswers){

For (every cell in the adjacency list at the x index the state we are looking at is in){

If(AdjacencyList[StateIndex][i] equals “1” and color == colorAnswers[i]{

Return false;

//i.e. if any of the adjacent states has the same color as the state we’re //currently trying to set, return false

}

}

else we return true;

}

Void ColorState(State, color, AdjacencyList, colorAnswers){

For(every color available){

If(colorCheck returns true){

colorAnswers[CurrentState] = current color;

if(there are still states left){

ColorStates(State + 1, color, AdjacencyList, colorAnswers);

}

}

}

}

* Explain the Pseudo code in a paragraph

My first function, colorCheck, for each state, loops through all of the other states searching for a state that is adjacent (in this case marked by a “1”). colorCheck returns false if the color cannot be set i.e. the found, adjacent states hold the same color. Else, it returns true. By returning true, colorCheck returns to ColorStates, which calls colorCheck. ColorStates, for each color, loops through each of the states by calling colorCheck for each state and every color combination. When colorCheck returns true, the color is set and the next state is pulled up to be checked until no states are left and the results are stored in an array to be printed.

* Source code for the implementation and instructions on how to run your code

*/\*  
\* Used the Graph coloring wikipedia article:  
\* “Graph Coloring.” Wikipedia, Wikimedia Foundation, 7 Feb. 2020, en.wikipedia.org/wiki/Graph\_coloring.  
\* \*/***import** java.io.\*;  
**import** java.util.ArrayList;  
  
**public class** MapColor {  
  
 **public static void** main(String[]args) {  
  
 */\*  
 \* The csv files MUST be in the program folder with the .idea, out and src files.  
 \* To change the file being used:  
 \* change the command line argument  
 \* Map of Australia = Aus.csv  
 \* Map of India = India.csv  
 \* Map of the US = US.csv  
 \*  
 \* for my runs, but the files can be named whatever as long as they're in the arg[0] slot.  
 \* \*/  
  
 //String csvFile = "C:\\Users\\colin\\IdeaProjects\\Data Structures\\MapColorProblem\\US.csv";//don't know why it got thrown into my data structures folder* String csvFile = args[0];  
 BufferedReader buffread = **null**;  
 String row = **""**;  
 String splitter = **","**;  
  
 ArrayList<String> Names = **new** ArrayList<String>();  
 ArrayList<String> OneDim = **new** ArrayList<String>();  
  
 **int** count = 0; *//Count to use later when creating array sizes* **try** {  
 buffread = **new** BufferedReader(**new** FileReader(csvFile));  
 **while** ((row = buffread.readLine()) != **null**) {  
 **int** j = 0;  
 String[] states = row.split(splitter);  
  
 **for** (**int** i = 0; i < states.**length**; i++) {  
 *//System.out.println("State: " + count + ": " + country[i]);* OneDim.add(states[i]);*//Dump everything into a list to sort later* }  
 count++;  
 }  
 }**catch**(FileNotFoundException e){  
 e.printStackTrace();  
 }**catch**(IOException e){  
 e.printStackTrace();  
 }**finally**{  
 **if**(buffread!=**null**){  
 **try**{  
 buffread.close();  
 }**catch**(IOException e){  
 e.printStackTrace();  
 }  
 }  
 }*//File error handling  
 //System.out.println("--///////-");* String[][] Grid = **new** String[count-1][count-1];*//Grid to hold the adjacency list* String[] arrNames = **new** String[count - 1];*//To hold the names for documentation later* **int**[] colorInfo = **new int**[count - 1];  
 colorInfo[0] = 0;  
  
 */\*  
 \* Color info:  
 \* 4 Colors:  
 \* Red, Green, Blue, Yellow, None  
 \* 1 , 2 , 3 , 4 , 5  
 \* Respectively.  
 \*  
 \* The 5th color, none, is required to find solutions for the maps of India and the US  
 \* \*/* ArrayList<String> Temp = **new** ArrayList<String>();  
  
 **for**(**int** i = 0; i < OneDim.size(); i++){  
 **if**(OneDim.get(i).equals(**"0"**) || OneDim.get(i).equals(**"1"**) || OneDim.get(i).equals(**","**)){  
 *//System.out.print(OneDim.get(i) + " ");  
 //System.out.print("i: "+i+" ");  
 //System.out.print("-");* Temp.add(OneDim.get(i));*//Sorting only the adjacency list into this list to be put into a 2d array* }  
 **else** {  
 *//System.out.println(OneDim.get(i));  
 //System.out.println(OneDim.get(i) + ", ");* Names.add(OneDim.get(i));*//Throwing the names into the Names list to be put into a 1d name array  
 //System.out.println();* }  
 *//System.out.println();* }  
 *//System.out.println("\n---");* **for**(**int** i = 1; i < count; i++){  
 *//System.out.println(Names.get(i));* arrNames[i-1] = Names.get(i);*//Names starts at 1 and arrNames starts at 0 to avoid the blank space at the beginning of names at 0,0.  
 //System.out.println(arrNames[i-1]);* }  
  
 *//System.out.println(Temp);* **int** counter = 0;  
 *//System.out.println("--");* **for**(**int** x = 0; x < Grid.**length**; x++){  
 **for**(**int** y = 0; y < Grid.**length**; y++){  
 *//System.out.print(Temp.get(counter) + " ");  
 //System.out.println(counter);* Grid[x][y] = Temp.get(counter);*//Placing all of the Adjacency list info into an array for easier use  
 //System.out.println("Grid["+x+"]["+y+"]: " + Grid[x][y]);* counter++;  
 }  
 }  
  
 */\*  
 for(int i = 0; i < Grid.length; i++){  
 System.out.print(arrNames[i] + " is adjacent to: ");  
 for(int j =0; j < Grid.length; j++){  
 if(Grid[i][j].equals("1")){  
 //System.out.print("Grid: "+Grid[i][j]);  
 System.out.print(arrNames[j]+", ");  
 }  
 }  
 System.out.println();  
 }\*/ //checking adjacency  
  
 /\*  
 for(int i = 0; i < 4; i++){  
 System.out.println("i: " + i);  
 }//Just error checking to look through the grid and array to check for adjacency when printing later  
 \*/  
  
 ColorStates*(0, 0, Grid, colorInfo);  
  
 **for**(**int** i = 0; i < colorInfo.**length**; i++){  
 *//System.out.println("Colors " + i +": " + colorInfo[i]);// printing the answer AFTER the function has run* }*//error checking to see the answer* **try** {  
  
 File output = **new** File(**"output.txt"**);  
  
 FileWriter write = **new** FileWriter(output);  
 BufferedWriter buff = **new** BufferedWriter(write);  
  
 **for** (**int** i = 0; i < Grid.**length**; i++) {  
 buff.write(arrNames[i] + **":"** + colorInfo[i] + **"->"**);  
 *//System.out.print(arrNames[i] + ":" + colorInfo[i] + "->");* **for** (**int** j = 0; j < Grid.**length**; j++) {  
 **if** (Grid[i][j].equals(**"1"**)) {  
 *//System.out.print("Grid: "+Grid[i][j]);  
 //System.out.print(arrNames[j]+", ");  
 //System.out.print(arrNames[j] + ":" + colorInfo[j] + ", ");* buff.write(arrNames[j] + **":"** + colorInfo[j] + **", "**);  
 }  
 }  
 *//System.out.println();* buff.newLine();  
 }*//checking adjacency* buff.close();  
 }  
 **catch**(IOException e){  
 e.printStackTrace();  
 }  
  
  
 }*//end of main* **public static boolean** colorCheck(**int** stateNum, **int** color, String[][]Grid, **int** []colorInfo){  
 */\*  
 \* This function is used inside of ColorStates to check and see if the states adjacent to the current state being checked are of the same color  
 \* If they are not the same color then we return true, if they are the same color, we return false.  
 \* \*/* **for**(**int** i = 0; i < Grid.**length**; i++){  
 **if**(Grid[stateNum][i].equals(**"1"**) && color == colorInfo[i]){*//if the state is adjacent and is the same color* **return false**;  
 }  
 }  
 **return true**;*//we're good and can set the color.* }  
  
 **public static void** ColorStates(**int** stateNum, **int** color, String[][]Grid, **int**[] colorInfo){  
  
 **for**(color = 1; color < 5; color++){*//starts at 1 because I'm considering the first color to be 1, not 0.* **if**(*colorCheck*(stateNum, color, Grid, colorInfo)){*//If true, keep going, else exit  
 //System.out.println("It's good");//error checking* colorInfo[stateNum] = color;*//setting the colors* **if**(stateNum + 1 < Grid.**length**){*//If there are still states left, keep going, else stop and print the answer  
 ColorStates*(stateNum + 1, color, Grid, colorInfo);*//recurse* }  
 }  
 }  
 }  
}

The above code, and the project files as a whole are included in the .zip file holding everything.

To run the program, the .csv files (in my case were changed to Australia.csv, US.csv and India.csv) MUST be located in the program folder like how they are organized in the program folder in the .zip folder. The .csv files must either be passed by direct path (shown by a commented out line) or by command line argument. The resulting output.txt holds the answer formatted as shown in the question.

All coding and runs were done on IntelliJ IDEA

I will gladly give an in person demo of my problem 5 or supply any additional information needed to ensure my program runs, as requested.

Ex:

Western Australia:3->Northern Territory:4, South Australia:2,

1. “Graph Coloring.” *Wikipedia*, Wikimedia Foundation, 7 Feb. 2020, en.wikipedia.org/wiki/Graph\_coloring. [↑](#footnote-ref-1)