ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

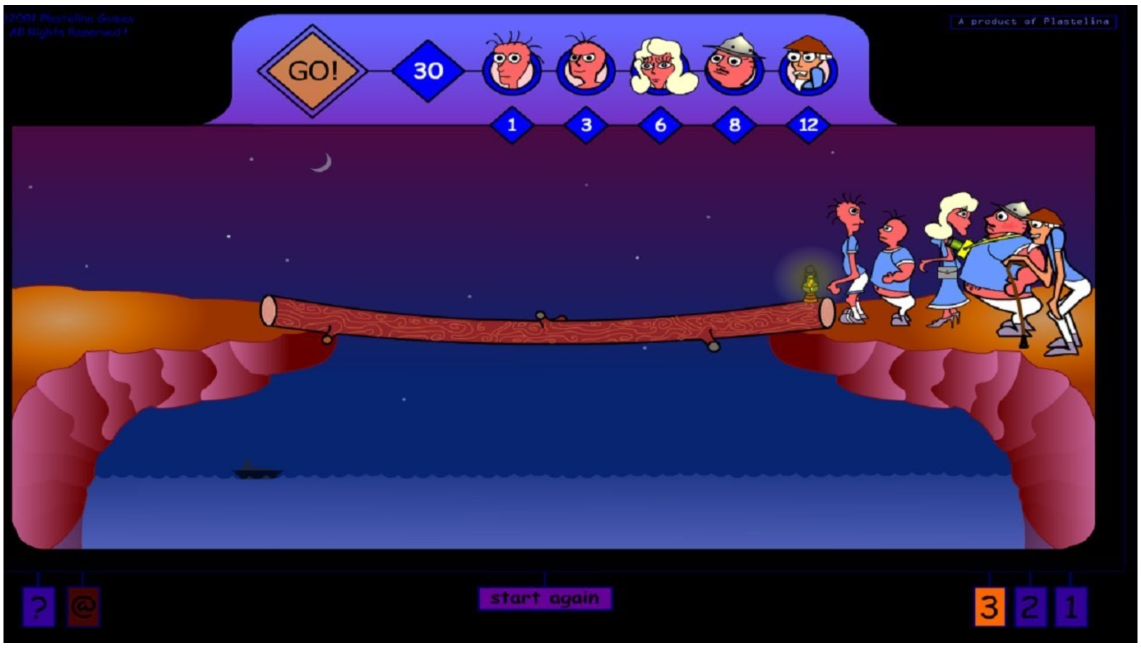
Artificial Intelligence
1st Project

Panagiotis Katsos
Epaminondas Ioannou
Petros Tsotsi

2020-2021

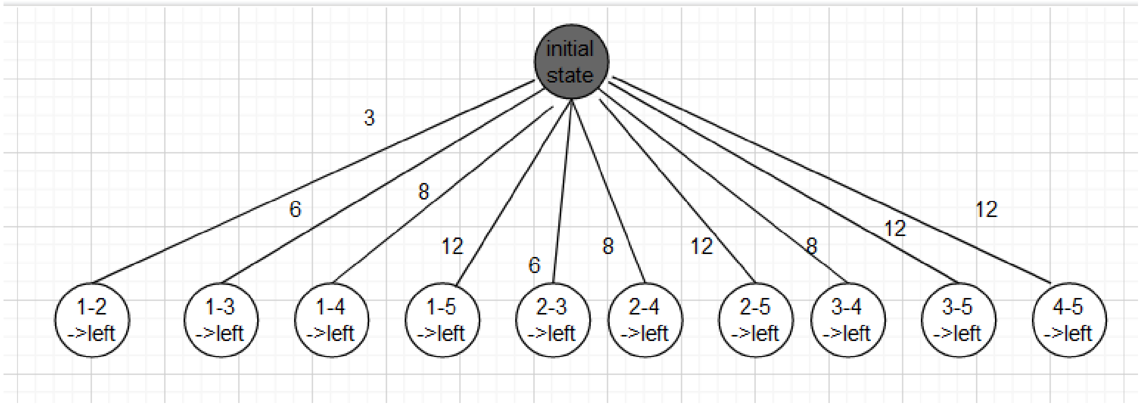
Introduction

In the context of the first programming project, our team has chosen to solve the problem of Bridge and Torch. The solution of this problem was implemented in Java. The following is an analysis of the algorithm that was used in order to solve the problem, as well as an explanation of the code.



1 Basic Concept

After experimenting with different algorithms, like BFS, DFS etc, we concluded that the A star algorithm was best suited for this specific problem. We will use the example of the above figure to show how A star works. The family members are 5, with times 1,3,6,8,12 and ids 1,2,3,4,5 respectively. According to the rules of the game, the desired result of the algorithm is for all members to cross to the other side in less than 30 seconds. The problem begins with all members being on the right side. This state is called the initial state (tree root). Family members can either move to the right or to the left, depending on their current position. After each move the state changes. It is obvious that in the first step of the algorithm one or two members will move to the left, because all members are on the right side. With this way more states are produced which are the initial state's children. At this point it is worth noting that we have found that the best solution must include 2 people at a time while moving to the left and only 1 person on the trunk at a time while moving to the right. Based on the previous view, we decided for time-efficiency reasons not to include all potential children of a state, but only those which comply with the above restrictions. For example, if our initial state is the one pictured above, after the first step of the algorithm, the children-states that are going to be produced are all the combinations of 2 people: 1-2, 1-3, 1-4, 1-5, 2-3, 2-4, 2-5, 3-4, 3-5, 4-5, total of 10 children-states.



The heuretic we invented, removed some restrictions, in order to be acceptable. The cost of each move is greater than zero, so the heuretic will also be consistent. Therefore, by having a consistent and acceptable heuretic it is known that the A star with closed set algorithm will lead to an optimal solution, in this case the minimum amount of time for all members to cross the bridge. The restriction we removed is the amount of people that can cross the bridge. According to the rules, the bridge can hold up to 2 people, but by removing this restriction there can be as many as possible. The heuretic calculates the score of each child-state as follows:

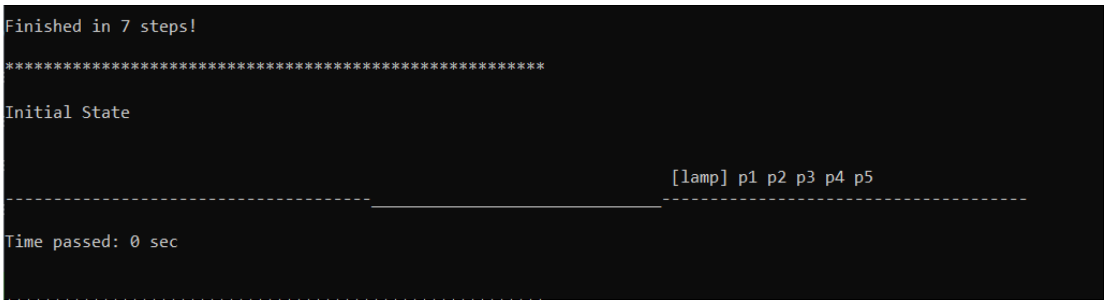
- If in the state that is evaluated, the lamp is on the right side, then its score is equal to the maximum time of the people that are on the right side. This is reasonable, as we have removed the restriction of only 2 people on the bridge, so for all members to cross, the amount of time required is equal to the maximum.

- If in the state that is evaluated, the lamp is on the left side, then its score is equal to the summation of the minimum time of people that are on the left side and the maximum time of people that are on the right side. Again, since there is no restriction on the number of individuals, the best solution for the specific state is that the fastest person returns back on the right side (holding the lamp) and then all family members cross the bridge together to the right side.

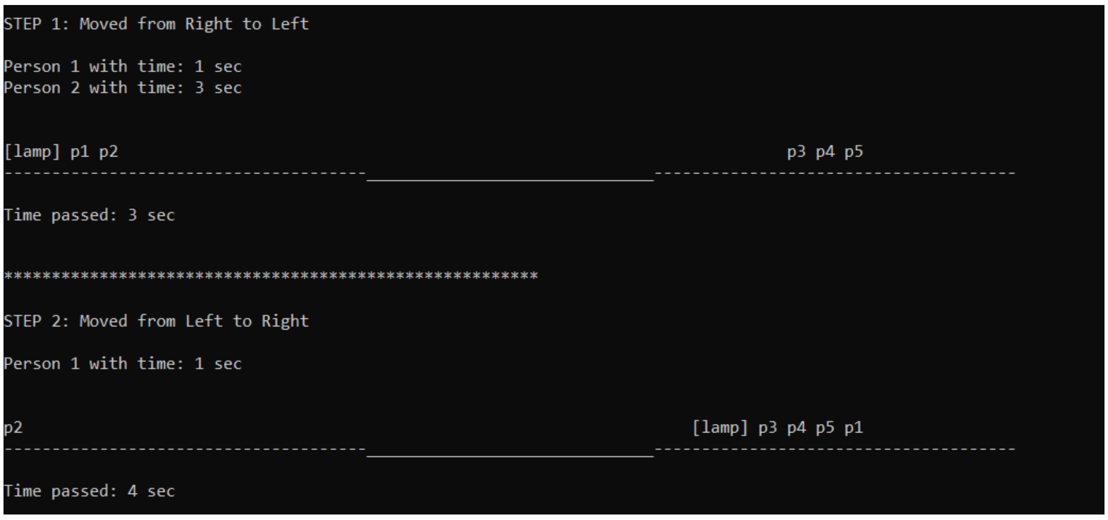
This heuretic function calculates an integer number that represents the score of each state. This score is equal to the time that is required from the specific state to reach a terminal state. **Then through the evaluate function, which is analyzed below, this cost is added to the time that has passed for the specific state to reach at that point. This is how the final score for each state is calculated. So in this way only the children of the state which has the best score, are produced (in our case minimum score) each time. As long as the heuretic function is consistent and acceptable, then the solution is surely optimal.**

Now we will show how the program we implemented, displays the optimal path for this specific example.

Initial Step

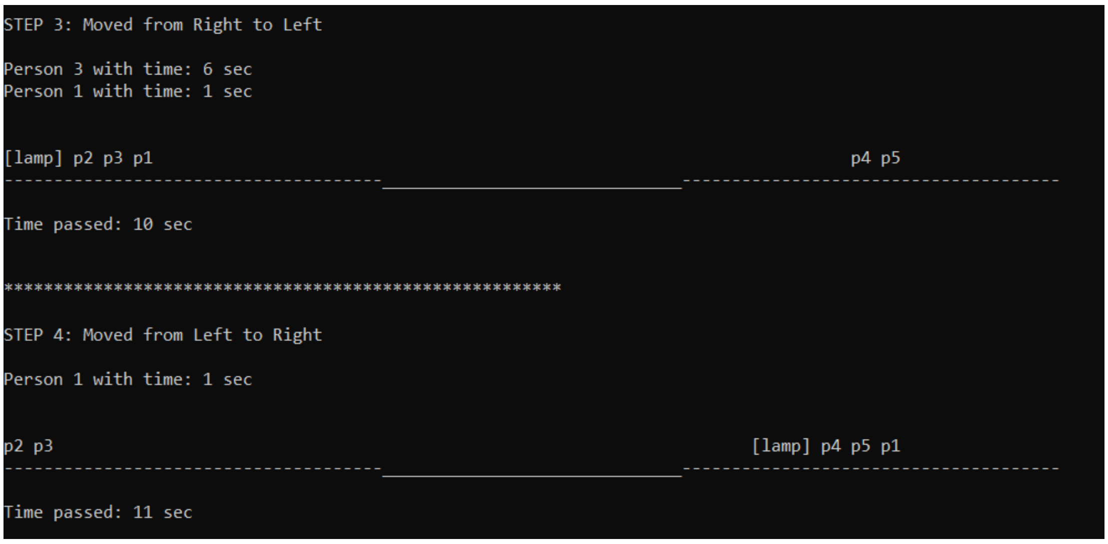

```
Finished in 7 steps!

*******************************************************

Initial State


                                                                    [lamp] p1 p2 p3 p4 p5
--------------------------------------______________________________-------------------------------------

Time passed: 0 sec


-------------------------------------------------------
```

1st and 2nd step



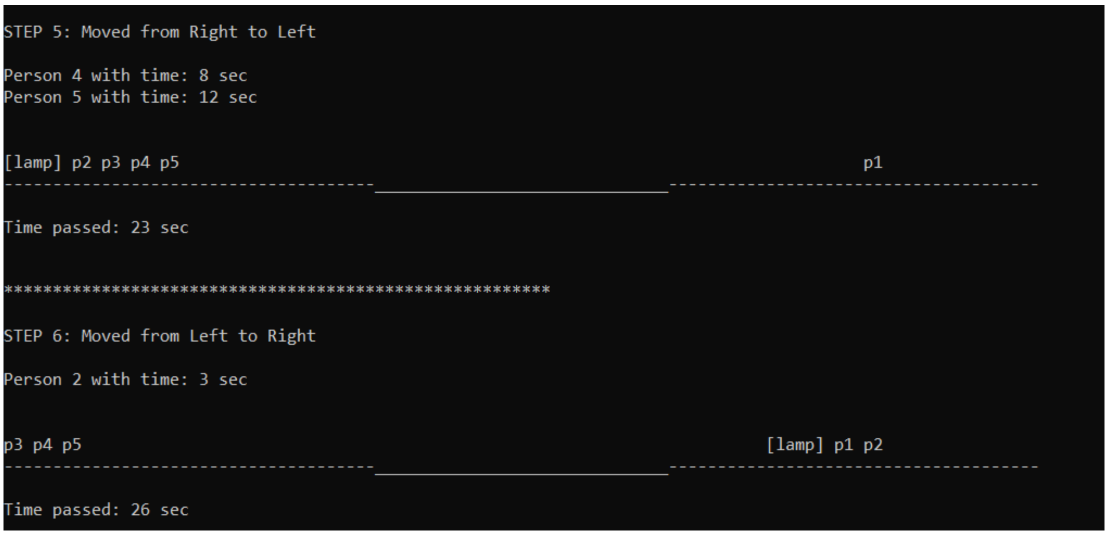
```
STEP 1: Moved from Right to Left

Person 1 with time: 1 sec
Person 2 with time: 3 sec


[lamp] p1 p2                                                                 p3 p4 p5
-------------------------------------______________________________-------------------------------------

Time passed: 3 sec


********************************************************

STEP 2: Moved from Left to Right

Person 1 with time: 1 sec


p2                                                                  [lamp] p3 p4 p5 p1
-------------------------------------______________________________-------------------------------------

Time passed: 4 sec
```

3rd and 4th step



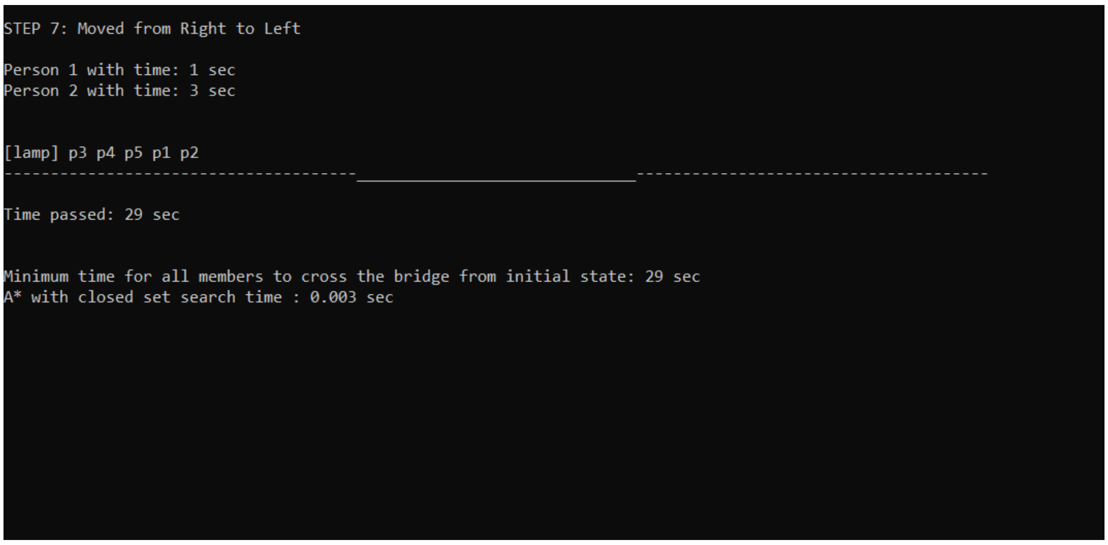
```
STEP 3: Moved from Right to Left

Person 3 with time: 6 sec
Person 1 with time: 1 sec


[lamp] p2 p3 p1                                                                   p4 p5
-------------------------------------______________________________-------------------------------------

Time passed: 10 sec


********************************************************

STEP 4: Moved from Left to Right

Person 1 with time: 1 sec


p2 p3                                                              [lamp] p4 p5 p1
-------------------------------------______________________________-------------------------------------

Time passed: 11 sec
```

5th and 6th step



```
STEP 5: Moved from Right to Left

Person 4 with time: 8 sec
Person 5 with time: 12 sec


[lamp] p2 p3 p4 p5                                                                p1
-------------------------------------______________________________-------------------------------------

Time passed: 23 sec


********************************************************

STEP 6: Moved from Left to Right

Person 2 with time: 3 sec


p3 p4 p5                                                                [lamp] p1 p2
-------------------------------------______________________________-------------------------------------

Time passed: 26 sec
```

7th step and Terminal State



```
STEP 7: Moved from Right to Left

Person 1 with time: 1 sec
Person 2 with time: 3 sec


[lamp] p3 p4 p5 p1 p2
-------------------------------------_____________________________-------------------------------------

Time passed: 29 sec


Minimum time for all members to cross the bridge from initial state: 29 sec
A* with closed set search time : 0.003 sec
```

We can see that our algorithm does indeed find a solution with a total amount of time less than 30 seconds (29).

2 Code Analysis

To solve this problem 4 classes were constructed that are called respectively SpaceSearcher, State, Person and Main. **SpaceSearcher class implements A star with closed set algorithm the same way that it is implemented in the lab code, so no further explanation is needed**. The remaining classes are analyzed below.

- **Class Person:** This class represents a family member. Every object contains its id, time variables which correspond to the id number this particular member has and the time that is required so they can cross the bridge. There are two constructors. The first one takes as arguments the two variables that were just mentioned and proceeds to make the appropriate initializations. The second constructor is empty and is used just for object creation, with no initialization. Only the first constructor is used in our code. We have also included some getters and setters methods for the id and time variables, because they are set to private.

- **Class State:** In this class we implement Comparable <State> so that we can override the compareTo method, to compare the scores of two different states. For each state there is a right side variable that contains all family members that are currently on that side and its type is ArrayList<Person>. Same logic with the left side variable. The remaining variables that are used for each State object are analyzed below:

  1. ligthSide variable of String type that holds a string value (e.g "left" or "right") so for every state we know where the lamp is and which are the next possible moves, states-children.
  2. score variable of int type that will represent the score of every state, that will be calculated by evaluate method (method analysis in next page).
  3. father variable of State type which will hold the State object which represents the father of this specific state. This is mainly used in order to display the path that was followed from initial to terminal state, when a solution is found.
  4. cost variable of int type that represents the amount of time that has passed to reach

this specific state. This variable is of vital importance, because in contrast to bestFS, in A* algorithm we are also interested in the cost of reaching a state starting from the root (besides the cost of the heuretic function).

There are 3 constructors. The first constructor is used for the initialization of the root state, so it takes only 1 argument, which is a variable of ArrayList<Person< type. This value is then assigned to the rightSide variable, since all family members are on the right side, in the initial state. Also with the appropriate set method, which will be mentioned below, we set the lightSide variable to "right". The second constructor takes the arguments left, right, lamp, cost in order to assign their values to the corresponding variables mentioned above. We access the father and score variables via setter methods only, to ensure encapsulation. We also have a third constructor, which is empty, in case we just want to create a new State object without initializing any attribute. At this point we will explain in detail every method of the class other than setters and getters.

– **boolean is Terminal():** This method simply checks if the state we reached is terminal. The only command that is used is a return statement which is return rightSide.isEmpty(). If this is true then that means the right side is empty and we reached a terminal state.
– **int heuristic():** This method returns an integer which represents an evaluation score of a specific state. In the body of this method an if condition initially checks if the lamp is on the right side, i.e if the lightSide variable is equal to "right". If this is the case, we initialize an int variable and make it equal to 0, in order to find the maximum amount of time of the family members located on the right side. Since we have removed the restriction of only 2 members, the cost to reach a terminal state is equal to this maximum amount of time. This is very easily achieved with a for loop on the rightside which is an ArrayList<Person>. Using an if statement we check if there is a longer amount of time than the present maximum amount and if there is this becomes our new max. Once we exit the for loop this maximum amount of time is returned. If the lamp is on the left side (i.e the variable lightSide is equal to left) the cost to reach a terminal state is equal to the minimum amount of time from members on the left (to return back with the lamp) plus the maximum amount of time from the people on the right (including the person who returned from the left side). In a similar way, as the one that was explained above, we find the min and max from each side and finally return their sum.
– **void evaluate()**: The evaluate method according to the definition of A* sets the total score of the specific state equal to the sum of the time (cost) that has passed until then plus the result that the heuristic returns.
– **ArrayList<State>getChildren():** This method generates the children of a specific state. We emphasize here that the restriction we have, is not to make movements to the left side, which contain only 1 person and not to make movements to the right side which contain 2 people, as this obviously will not produce the correct result, so there is no reason to consider chlidren that make these moves as possible optimal solutions. The following variables are initialized:
  1. **ArrayList<State> children = new ArrayList<State>()**: This variable is an arraylist of states which represents the children of a particular state that we are currenly exploring.
  2. **ArrayList<Person> temp = new ArrayList<Person>()**: This variable is temporary, as it represents the 2 people who will from left to righ each time. All possible combinations are considered.

3. **State child**: This variable will represent the child we produce each time for a specific state.
4. **int maxim = 0**: This variable will be used to move from right to left, as with any combination, both family members will move with the slowest's pace, so we must find the larger amount of time of the two and assign it to this variable.
5. **int time1=0**: This variable will contain the amount of time which is required for the first member of the two to cross the bridge (when moving from right to left).
6. **int time2=0**: This variable will contain the amount of time which is required for the second member of the two to cross the bridge (when moving from right to left).

To produce the children of a state each time, we initially check if the lightSide is equal to ”right” or ”left”, since depending where the lamp is we know which are the next possible moves. We can do this very easily with the if conditions- if (lightSide == ”right”) -else if (lightSide == ”left”). If the first condition is true, that means that the lamp is on the right when examining this specific state. We first check a very special case in which only 1 member needs to cross the bridge. This will be true when no one is on the left side and when the size of the right side is equal to 1- if (this.leftSide.isEmpty () && (this. rightSide.size () == 1)). We then construct another state object which is initialized with it's father's elements with the command child = new State (this.leftSide, this.rightSide, 'right', this.cost). Then for this specific state we assign the address of its father to the variable father, using the command child.setFather(this). This is an important point as if it is a terminal state we need the path starting from the root. Then we add to the cost of the child the amount of time the specific person makes to pass across (to the right side), with the command child.setCost(this.rightSide.get(0).time). Since we have ensured that we have only one person on the right side, with the command that was shown above, we add his time to the total cost. We add this person to the temp array with the command temp.add(this.rightSide.get(0)) and call the moveLeft method with temp as an argument, which as we have already stated, will always contain family members (in this case only one) that move to the left. This method is analyzed below. It is worth noting that since only 1 family member needs to cross the bridge from the start of the game and only 1 child will be produced from this initial state, there is no need to evaluate. Finally we add the specific child to the children ArrayList with the command children.add(child). If more than one family members are in the initial state of the game then for a specific state the pattern we follow to produce children, is to have 2 for loops so that we can have all possible combinations with 2 people from the right side that will move to the left (meaning one child-state for each possible combination). Then we follow this exact pattern as we saw for only 1 person, with the additional changes that we must find the amount of times for 2 people with the commands time1=this.rightSide.get(i).getTime() and time2=this.rightSide.get(j).getTime() and then we find the maximum time of the two with the command maxim=Math.max(time1, time2) since we wil add the maximum time to the cost of the specific child with the command child.setCost(maxim). We also add to the temp ArrayList the 2 people that are selected, with the command temp.add(this.rightSide.get(i)) and temp.add(this.rightSide.get(j)). We call moveleft method with the command child.moveLeft(temp), evaluate the child with the command child.evaluate and finally add the child to the children ArrayList. The program works in a very similar way when the lamp is on the left side, only that in this case we will not have 2 for loops, since the restriction we have made is that only one person will move to the right each time.

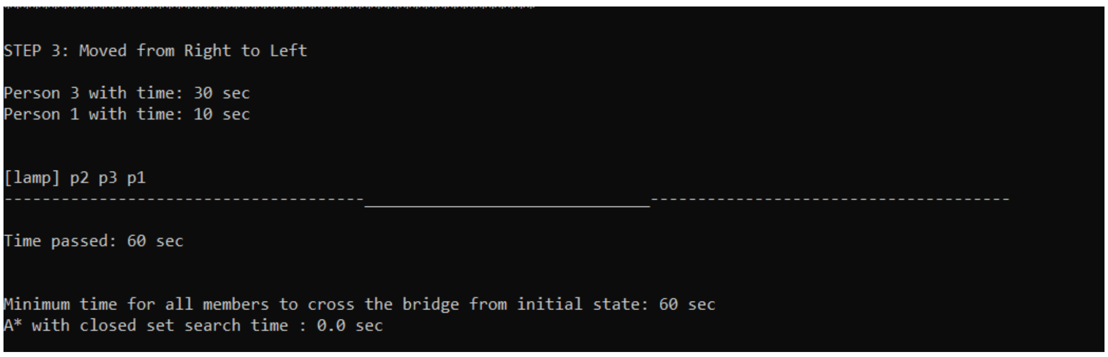
  - **void moveLeft(ArrayList<Person> people,int max):** This method, as one can tell from its name, performs a movement to the left. An arraylist with Person objects must be passed as an argument. We remove these people from the rightSide for the specific state, and we add them to the left side with the add command. Finally we make the variable lightSide equal to 'left' since after the movement to the left the lamp is now on the left side.
  - **void moveRight(Person person):** In this method since we follow the initial constraint that it makes sense to move only 1 person back to the right side, we have one Person object as an argument and not an Arraylist. We remove this person from the leftSide and add it to the rightSide.
  - **void print(String side):** A simple method that prints the people on each side and the bridge for a specific state.

- **Class Main:** This is the runner program. Initially we create a Scanner object so that we can give as inputs the number of family members, their time and the time limit that exists for all members to pass across. Then we print the message: Give the number of family members, so that the user gives the appropriate input. We achieve this with the command int N=in.nextInt(). Then we create an ArrayList of Person objects called family, which contains all family members. Then we enter a for loop and ask the user to give as input each member's time. At the end of each loop we create a Person object by giving its id and time as arguments to the constructor and we add it to the ArrayList family with the command family.add(temp). Once we exit the for loop we create a State object called init_state and pass the family as an argument to the constructor to make the appropriate initialization in the state, i.e all Person objects must be contained in the rightSide Arraylist. We create a SpaceSearcher object with the command SpaceSearcher space _searcher = new SpaceSearcher () so that we can execute A* with closed set algorithm. Then we initialize the State terminal variable to null. We also create a variable of long type called start with the command long start = System.currentTimeMillis(). We will use it to get the present time, in order to find the total amount of time our algorithm requires to complete. Next we call the SpaceSearcher method which accurately implements A* with closed set with the command: terminal = space_searcher.A_StarClosedSet(init_state). This method will return the terminal state if it found one or null otherwise. Then we assign the present time (after the algorithm has been executed) to the end variable with the command: long end = System.currentTimeMillis(). Then we check whether terminal variable is Null. If this is true, no solution was found and an appropriate message is printed. If it is not null then a solution has been found. We first find the exact path until we reach a terminal state, by adding the father of each state in an ArrayList, starting from the terminal state. Then we simply print each state in an understandable way, i.e the initial state and each next move (see pages 3-5).

3 Experimental Data

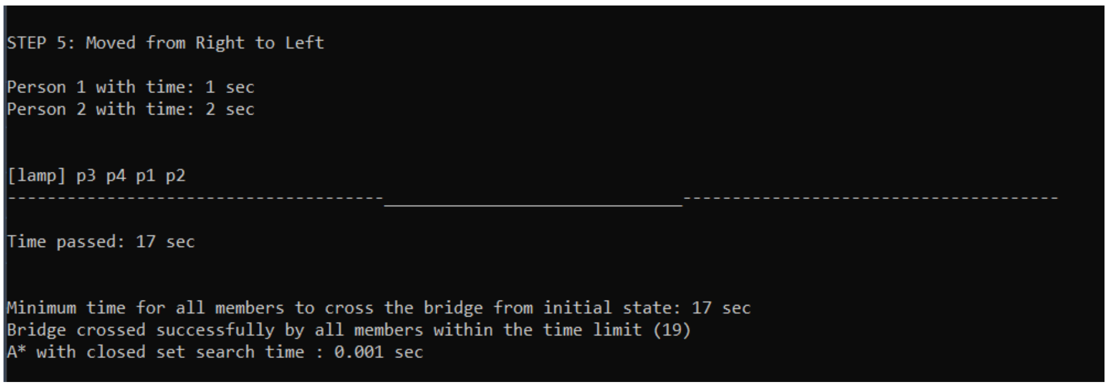
Here we can see the results of our program with some experimental data.

N=3 and time required for each person 10,20,30



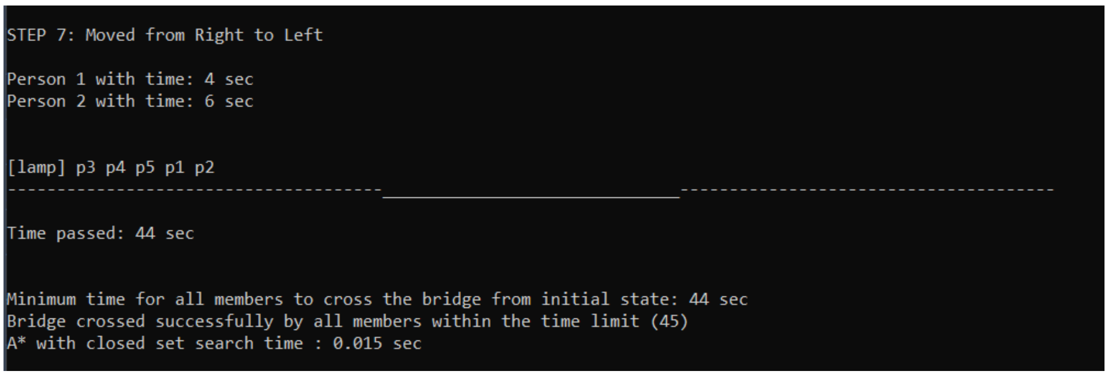
```
STEP 3: Moved from Right to Left

Person 3 with time: 30 sec
Person 1 with time: 10 sec


[lamp] p2 p3 p1
------------------------------------____________________________------------------------------------

Time passed: 60 sec


Minimum time for all members to cross the bridge from initial state: 60 sec
A* with closed set search time : 0.0 sec
```

N=4 and time required for each person 1,2,5,10



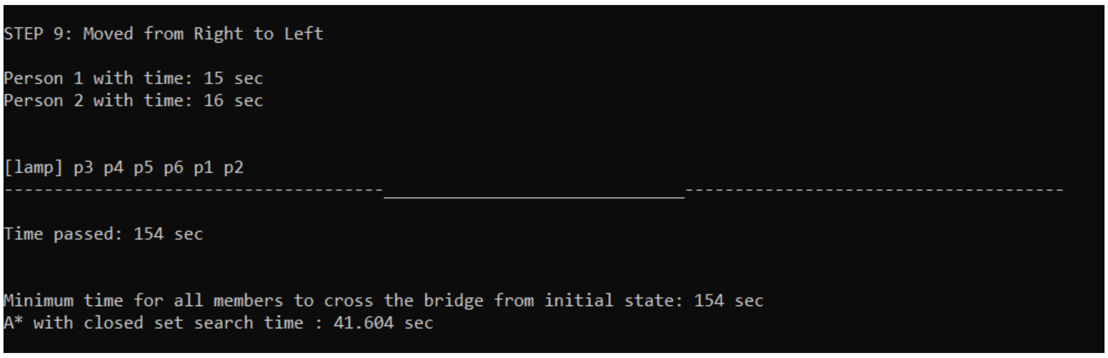
```
STEP 5: Moved from Right to Left

Person 1 with time: 1 sec
Person 2 with time: 2 sec


[lamp] p3 p4 p1 p2
------------------------------------____________________________------------------------------------

Time passed: 17 sec


Minimum time for all members to cross the bridge from initial state: 17 sec
Bridge crossed successfully by all members within the time limit (19)
A* with closed set search time : 0.001 sec
```

N=5 and time required for each person 4,6,7,9,11



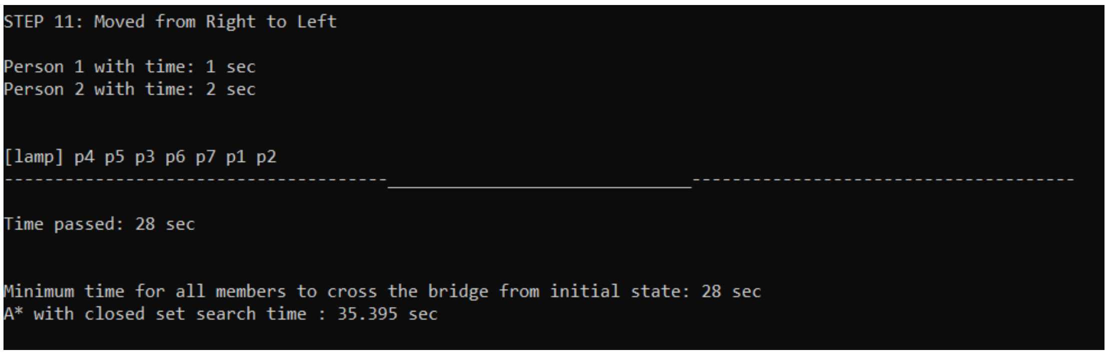
```
STEP 7: Moved from Right to Left

Person 1 with time: 4 sec
Person 2 with time: 6 sec


[lamp] p3 p4 p5 p1 p2
------------------------------------____________________________------------------------------------

Time passed: 44 sec


Minimum time for all members to cross the bridge from initial state: 44 sec
Bridge crossed successfully by all members within the time limit (45)
A* with closed set search time : 0.015 sec
```

N=6 and time required for each person 15,16,18,19,24,25



```
STEP 9: Moved from Right to Left

Person 1 with time: 15 sec
Person 2 with time: 16 sec


[lamp] p3 p4 p5 p6 p1 p2
-------------------------------------______________________________-------------------------------------

Time passed: 154 sec


Minimum time for all members to cross the bridge from initial state: 154 sec
A* with closed set search time : 41.604 sec
```

N=7 and time required for each person 1,2,3,4,5,6,7



```
STEP 11: Moved from Right to Left

Person 1 with time: 1 sec
Person 2 with time: 2 sec


[lamp] p4 p5 p3 p6 p7 p1 p2
-------------------------------------______________________________-------------------------------------

Time passed: 28 sec


Minimum time for all members to cross the bridge from initial state: 28 sec
A* with closed set search time : 35.395 sec
```