Report for Lab 2 Assignment CSP

Introduction

A school wants to create a schedule for the lectures. For that, they want to assign to each day a number of subjects, and for each subject a teacher that lectures that subject. Attending to the constraints given in the problem, that restricts the number of subjects that can be lectured a day, which subjects can be lectured by which teachers or whether a subject has to be lectured after, before, on the same day, on a different day… with respect to other subject.

Model and Implementation

We decided to choose the following variables: on one hand we have variables that represent a slot of time for a day of the week (from monday to thursday). The domain of this variables are the different subjects that can be lectured during the week. On the other hand we have variables that represent the different subjects and their domain is formed by the teachers than are going to lecture the subjects. For the rest of the document, the first ones will be called time variables and the second ones will be called teacher variables.

When defining the constraints we can distinguish three types: constraints that affect the time variables, constraints that affect the teacher variables and constraints that affect both (there's only one defined in the statement). Now we need to consider how are the variables affecting others as they start to choose values of the domain. In this case, values of time variables will be affected by other time variables them and values of teacher variables will also be affected by other teacher variables values. Remarking that values of teacher variables will be affected/will depend on which values time variables are choosing.
That is the reason why our model will first choose values for the time variables and for the teacher variables in second place.

Constraints affecting the time variables are:

1. Each subject should be lectured two hours except for Physical Education that should be lectured one hour. We check all of the slots of the week counting how many times each subject has appeared. If they all have 2 except Physical Education, we know we have found a good combination
2. Social and Human Science should be lectured in two consecutive hours. For each day we will look at their slots. If in one slot we found that it is social and Human science we will check the next one. If there isn't or it is another subject we know that combination is not possible
3. Mathematics won't be lectured the same day that English is. For each day we will look at all of their slots. If in the same day both the counter for Maths and english is different than one, we know that that combination is false

4. Mathematics won't be lectured the same day that Natural Science is. Works as previous constraint, buth checking for math and natural science
5. Mathematics should be lectured in first hour of the day. This is achieved using the NostInset constraint over the time slots were math is not possible
6. Natural Science should be lectured in last hour of the day.This is achieved using the NostInset constraint over the time slots were Natural Science is not possible

Constraints affecting the teacher variables are:

7. A teacher should lecture two different subjects. We check all the subjects and see how many times each teacher has appeared. If all of them have 2, we know is a good combination
8. Two teachers cannot lecture the same subject. If previous constraint is met, this will also be met. Also, it comes as compulsory from the way we have design our variables
9. If Andrea lectures Physical Education, then Lucía will lecture Social and Human Science. If the teacher assigned for PE is Andrea, we look for the one assigned to Social and Human Science. If its Lucía, the combination is okay

Constraints affecting both time variables and teacher variables:

10. If Natural Science or Social and Human Science are placed in the first hour of Monday or Thursday, then Juan won't lecture any of them. We check the undesired slots and if they have Natural science or Social and Human Science and we see that Juan also is assigned to them, we know is a bad combination

As a result of applying this constraints, the total number of solutions is 324.

Adding Teachers

When adding a new teacher, the number of solutions should increase a bit, as we are making the domain of the teachers variables grow. In this case , it grows to 576. We needed to change Constraint 7 and assign one subject to the new teacher and reducing to one the subjects of another teacher, so that the total number of lectured subjects in the week was still six.

Removing Teachers

When removing teachers, the number of solutions should decrease as we are reducing the number of possible values in the domain of the teachers variables.
For this specific case, when removing Lucia from the domain, the number of possible solution decreases to 60. We need to consider that for this case we are not only removing a teacher, but also removing Constraint 9, as Lucia no longer belongs to the domain and therefore, that constraint has no sense.

Adding Subjects

A new subject needs to be added to the schedule. This subject is going to be ICT (Information and Communication Technology). When adding this subject, we need to redefine some of the constraints. Constraint 1 now will need to check that ICT is also in the schedule. We have decided that the number of lectures for this subject is going to be one. Constraint 7 needs to be modified as well, because the number of subjects is incremented by one, so Juan will lecture three subjects. As we are keeping the number of hours the

same, one of the existing subjects needs to reduce their number of lectures to 1. Choosing Social and Human Sciences makes Constraint 2 useless, so it will be removed.
As expected, by adding another subject and removing a constraint the number of solutions has grown up to 13200 solutions.

Removing Subjects

For this test, we decided to remove the Spanish subject. By doing this, we have to take into account that some of the previously defined constraints are no longer valid. For instance, Constraint 1 has to be redefined in order to avoid that some hours in the schedule have no lectures. We have decided to let any subject have more than two hours, no less than two hours and Physical Education will still be lectured only one hour.
Constraint 7 has to be changed as now there are only five subjects instead of six, so Juan will lecture only one subject.
As a result, the total number of solutions is 42

Adding Constraints

The first constraint forces Juan to lecture English and forces Andrea to lecture Natural Science. By itself, it generates a total of 54 solutions. The second constraint doesn't allow English to be lectured before all the Spanish lectures have taken place. By itself, it generates a total of 54 solutions. The third constraint avoids that Lucia lectures more than one class on Mondays. By itself, it generates a total of 114 solutions.
By adding this three constraints the number of solutions were reduced to 3.

Removing Constraints

Removing Constraint 9 raises the number of solutions to 360. This increase is so minimum because the constraint is not affecting a large number of variables. However, if Constraint 2 is removed, then the number of solutions should be much larger, as it affects all the time variables plus a teacher variable. In this case, the number of solutions grows to 6480, confirming our previous assumptions, and explains why there was such an increase in the test of adding a subject.

Full Timetable

We decided to increase the length of the problem to see how the answers increased. Therefore, we add 3 extra hours (belonging to Friday) and 2 extra subjects (Philosophy and Computers) to be able to fulfilled that extra time. The number of solutions grew in such a way that after more than an hour, we decided to stop the experiment. Adding a few extra variables increased the complexity of the problem in an exponential way

Heuristic Search

Introduction

For this problem, a bus has to travel a map to pick students from bus stops and deliver them at their corresponding schools. This map is represented with a graph. Each node of the matrix is a bus stop where we can find a school, a student or students from the same or different schools or nothing. There is a cost when travelling from one node to another and the cost of going from node A to node be does not necessarily has to be as the cost of going from node B to node A. In other words, the the cost matrix that is going to represent the travelling cost in our model doesn't have to be symmetric.
There is a maximum capacity for the bus and once students get into the bus, they don't leave until the reach to their respective schools.

Model

This problem can be seen as a variation of the pick and deliver problem.
In order to solve this problem we have to implement a graph (which will be represented as a matrix) and define the states and actions allowed in the problem.
A state will be composed by the bus position, the students that need to be picked up, and the students that have to be delivered. The initial state will be the bus stop were the bus is placed before starting to search, all the students that need to be picked up and no students delivered. The goal state will be the initial position of the bus, no students to pick up and no students to deliver.
There are three possible actions in this model:

- Move:

    - Preconditions: There is an immediate way to go from the actual stop to a next one
    - Cost: The cost of that way
    - Result: The bus moves from one position to another
- Pick a Student:
    - Precondition: Bus should be in a stop where there are students waiting to be pick and the amount of students in the bus is smaller or equal that it's total size
    - Cost: Picking a student has no cost
    - Result: One student of that stop is picked and added to the amount of students in the bus(we only pick students one by one to allow the possibility of going through a stop and picking students even if we can't pick all of them
- Deliver a Student:
    - Precondition: Bus is in a stop where there is a school and in the bus there is a student that wants to go to that school
    - Cost: Delivering a student has no cost
    - Result: One student leaves the bus and gets to its destination (thus disappearing from the perspective of the problem

3 different heuristics were designed:

- Pick Distance: Calculates the distance between the position of the bus to the students. An optimistic criteria is used so we only add the minimum distance between the bus and each of the stops were students can be picked, assuming that if we go to a stop we will pick all the possible students. If we add the distance of each of the students instead of the stops we would count multiple times the same distance making the heuristic non-admissible.
- Bus Deliver: Calculates the addition of the smallest distance between the students that are in the bus and their respectives schools. Same as before, we join the students that go to the same school to avoid overcounting
- Lower Boundaries: After designing the 2 previous heuristics we find out that each of them tackled different stages of the problem, so we decided to join them, thus creating a lower boundary to the total real cost (the addition of the distances calculated before will give us the smallest distances needed to pick a student and deliver it to it's desired school. We know, addition of 2 admissible heuristics doesn't have to give an admissible one but that will be test later

Structure of the code

We can differentiate four different parts inside the code: the parser of the input file that contains the problem to be solved, a Node class containing all the information relevant for a state, a Solver class containing information for the search algorithm and the algorithm itself, and the main that invokes the constructor and functions required to start the search algorithm as well as creating the output files for the solution and the statistics of the program. On the following sections, each part will be described in depth.

Initialization Phase

It is made up of four different functions parse functions. All of them make use of two important functions which are strip and split. This functions allow us to remove blank spaces or escape characters and store just the information we need.

- Matrix parser: is in charge of parsing the cost matrix of the graph. The double hyphens "--" are changed for 100000000000000000000, which is our definition of infinity. To know the dimensions of the matrix, the first line is read and the last word if it is kept. This word is formed by a "P" and followed by an index number. As the function is keeping the last word, it will know the number of nodes that the graph has and therefore, the dimensions of the matrix that has to be created. This matrix is assigned to a variable called MAP, and it will be a tuple to avoid any undesired modification.
- Schools parser: after n iterations reading a line of the input text containing the problem, the next line to read will contain the bus stops where the schools are located at. This information is stored inside a dictionary called SCHOOLS.
- Students parser: the next line to read will contain the bus stops were the students are waiting and to which schools they have to be delivered. This information will be stored inside a variable called STP (Students To Pick), which will be used later as parameter for the constructor of the node containing the initial state.
- Bus parser: the next and last line of the input file, will contain the information regarding bus, being this information its capacity and the bus stop were it will start travelling from.

Following this parse functions we have the initialization of the Floyd-Warshall matrix. It will first make a copy of the values of the MAP that we have already parsed. To find the number of vertices we simply invoke the len function over the MAP which will give us the number of rows. Then we will execute Floyd-Warshall Algorithm to obtain a new matrix that show as the smallest cost to go from one vertice to another. Here we find the reason of changing in the matrix parser -- to our infinite. Floyd-Warshall will never choose this way. At last we go again through the map making our diagonal 0.

Node class

Attributes

This class is used for creating the nodes of the search tree with the search function. The nodes contain the following attributes.

This attributes conform a state in our model:

- pos: stores the position of the bus in the graph.
- stp: students that have to be picked from a bus stop. It is a dictionary containing every stop containing students, how many of them and to which schools they belong.
- std: students inside the bus that have to be delivered. It is a dictionary that contains the number of students and to which school they belong.

This attributes contain important information for the Solver to perform the search:

- g: cost attribute that stores the accumulated cost after performing the action
- f: the evaluation function that indicates how good or bad is the state inside a node of the search tree to be expanded.
- children: a list with all the children nodes that were generated from the current state.

This attributes contain information related to the node that will be used by the Solver to save the solution and what happened in the search process.

- parent: contains the parent node that generated a node. For the initial node (the root) the value is None. It is used to print the path designed as solution by the search algorithm.
- action: contains the action that generated that node, to provide later a better explanation in the output file of what happened during the search.

Functions

The functions inside this class are the following:

- constructor: receives the position of the bus (pos), the students to pick (stp), the students to deliver that are inside the bus (std), the parent node and the action that generated that child.
- pickUp Student: checks if there is any student in the current bus stop. If there is a student and if there is enough space to pick them. In case this conditions are satisfied, a new node is created with the same information of the current node except that the student is removed from the attribute *stp* and added to the attribute *std,* the attribute actions saves a string with the action that was performed (picking), the attribute parent contains the current node. We also change the new node giving as accumulated cost the one from his parent and as total cost, the sum of this accumulated cost with the addition of the heuristic cost (which we obtain from the heuristic method) Finally, the node is inserted into the child list of the current state.
- deliverStudent: checks if there is a school in the current position. If so, and there's a student in the bus that has to be delivered in that school, a new node is created with the same information of the current node except that the student is removed from the attribute std, the parent will be the current node, action is a string with the action performed (delivering) and accumulated cost the one from his parent and as total cost, the sum of this accumulated cost with the addition of the heuristic cost. Finally, the node is inserted into the child list of the current state.
- move: the bus moves to any possible node in the graph that is connected to the current node. For each of this possibilities a new node is created, the parent will be the current node, action is a string with the action performed (moving).This time the new accumulated cost will need the addition of the cost of moving (obtained in the loop) while the total cost will be calculated the same way as before. Finally, the node is inserted into the child list of the current state.
- heuristic: Sees what has been the decided heuristic thanks to value of the global variable and executes it. For the correct use of the heuristics, init_Floyd_matrix must have been done:
    - Pick_Distance: Using the floyd matrix, for every stop in StudentsToPick we look for the shortest distance to the position of the bus simply looking in the matrix. We store them in an auxiliary array and pick the minimum
    - Bus_Deliver: For every entry of the dictionary studentsToDeliver we will look in the floyd matrix to it's shortest distance to the desired school. We will only add entries if the bus is not over the school stop we are looking to. The resulting heuristic is the addition of each of this entries.

    - Lower_Boundaries: Combines the 2 heuristics explained before, first looking at StudentsToPick and then at studentsToDeliver and adding all the distances and returning this addition
- elements in STD: auxiliary functions that helps to know how many students are in the bus at the moment by executing a counting loop over the std dictionary. It helps pickUp Student function to know if there is free capacity on the bus
- toString and printPath: both functions used to generate the output files (output and statistic). printPath is a recursive function that calls for toString and then looks for the parent of the node until we reach the root node (until a node doesn't have a parent). toString simply stores information of the state (bus position and the action it was executed) in an external output array

Solver class

Object that has the functions designed for obtaining a solution to the problem proposed

Functions

- Constructor: Creates an instance of the class and receives an initial starting Node (state)
- Solver: In charge of the resolution of the problem once an initial state has been given. First of all, it will calculate the heuristic value of our initial state and change the pertinent attribute in the node to store this information, initialize an openSet array that will be storing our different opened nodes and appends the initial node to it, initialize a closeSet array that will be storing our closed nodes and will create a success variable initialized to False. Next we have a loop that will be running the different operations needed until we run out of open nodes or we find our final goal. This loop can be divided in 3 stages:
    - Choosing best node to expand: We go through the nodes of the open list and pick the one with smallest total cost (f) and, in case 2 nodes have the same total cost, we choose the one with smallest heuristic cost. If this node is the final node (we don't have students to pick or deliver and we are in our initial position) the loop is broken and we begin to print our path
    - If the node is not our goal, we generate its children
    - After generating all of them we need to know if any of this generated children is useful.For this we first look through through the open list and if we find a node that has the same state as our actual node and a smallest total cost, we take this node out of our list of new Children. If our actual node has smallest total cost we will remove the node in the open list. This will also be done in the close list but without comparison of total cost. If it's on the closed list we already know that is our best possible option with that attributes. If after this 2 loops the node is still valid, we append it to the open list and continue to check for next node in the new children generated list

Final steps

From the solver we will have a boolean value that will tell us if it has been possible to find a solution. If we couldn't find it we simply print an alert in the screen telling the user. But if we have found it we will have to write the output files.
For the statistic files we will use a bunch of global variables:

- statistics_endtime: When we find our goal in the solver, this variable will store the time. In combination with the starting time, we will find the total time spent in the problem
- statistics_cost: When a solution is found, this variable will store the accumulated cost of the final node
- statistics_stops: Acts as a counter. Each time the toString() method is called we add 1 to the counter. With this we will know how many states have been needed to solve the problem
- statistic_expansions: Each time we go through the solver loop, we add one to this counter. This will show how many nodes were expanded in the end

For the output file we will use the output array we have been creating while invoking the printPath method. For having the correct order we invert this array and start writing in the file that the variable input has shown us. Due to stylistic needs we decided to pop the initial position of the array before reversing and print all the path putting the unicode arrow symbol after each of the writings. At last we write the initial position (that is also the last one) that we have pop and we have finished.

Bash Script

The script uses getopts to parse the flags that will be used to run the script.
The flag -f is used to indicate that the argument is the file containing the problem to be solved. It needs to be used with an argument.
The flag -u is used to indicate the heuristic that the algorithm will take into account to solve the problem. It needs to be used with an argument, which will be the desired heuristic, designed with the names pickD, busD and lowB, corresponding with Pick_Distance, Bus_Deliver anf Lower_Boundaries respectively.
The flag -h displays a couple of lines with help to use the script. No argument is needed.
If the script is executed just with the -f flag, no heuristic will be taken into account to solve the problem.

Test cases and Comparative analysis[1]

For this section we will run a series of tests cases using all the possible heuristics as well as without them. This will help found if our heuristics are founding the optimal path and which is the best.
First we will do an input test by introducing in our command line simply: ./BusRouting.sh
In this case the Bash script will jump and tell our user that there is no input given.

[1] Tests were run in a Lenovo ideapad 100-15IBD with 2.2 GHz of processor

If we continue in this line and try to execute a random generated heuristic name:
./BusRouting.sh -f ejemplos/problem2.prob -u 3erh
Bash again will jump and tell us that that heuristic does not exist

We will now began with using the files we have provide in the “ejemplos” folder:

- Case-1.prob is the example we were given but it's good to check that the results are the same :

No Heuristic	Pick Student	Deliver Bus	Low Boundaries
Time: 14.27034 Overall cost: 102 # Stops: 27 # Expansions: 8430	Time: 4.18962 Overall cost: 102 # Stops: 27 # Expansions: 3727	Time: 6.83934 Overall cost: 102 # Stops: 27 # Expansions: 4882	Time: 0.20943 Overall cost: 102 # Stops: 26 # Expansions: 633

- Case-2.prob is a smaller version of the previous test case and will be good for seeing how much the solutions degrade when making more complex problems:

No Heuristic	Pick Student	Deliver Bus	Low Boundaries
Time: 0.01469 Overall cost: 58 # Stops: 19 # Expansions: 270	Time: 0.01285 Overall cost: 58 # Stops: 19 # Expansions: 208	Time: 0.01045 Overall cost: 58 # Stops: 19 # Expansions: 192	Time: 0.00589 Overall cost: 58 # Stops: 19 # Expansions: 119

- Case-3.prob we decided to see what happened when we reduced the bus size significantly (only 2 people in the bus)

No Heuristic	Pick Student	Deliver Bus	Low Boundaries
Overall time: 4.236087799072266 Overall cost: 114 # Stops: 28 # Expansions: 4868	Overall time: 1.0603370666503906 Overall cost: 114 # Stops: 28 # Expansions: 1911	Overall time: 3.1075971126556396 Overall cost: 114 # Stops: 28 # Expansions: 3614	Overall time: 1.7453055381774902 Overall cost: 134 # Stops: 30 # Expansions: 2222

- Case-4.prob: We added a school and some children

No Heuristic	Pick Student	Deliver Bus	Low Boundaries
Overall time: 19.29238986968994	Overall time: 19.05065107345581	Overall time: 21.284444570541382	Overall time: 17.70395588874817

Overall cost: 274 # Stops: 49 # Expansions: 10134	Overall cost: 274 # Stops: 49 # Expansions: 9214	Overall cost: 274 # Stops: 49 # Expansions: 9945	Overall cost: 274 # Stops: 49 # Expansions: 8684

After looking at the statistics found in the running of the test cases we can have some things in clear:

- Most important thing we have found is that the Low Boundaries heuristic is non admissible. In case we change the size of the bus we don't find the optimal solution. We have decided to keep the heuristic as a showing of how the addition of 2 admissible heuristics doesn't have to create a better admisible one. It may be faster for every problem but some possibilities will make that heuristic useless
- The less the size of the bus, the lower time the problem will take because the possible combinations of students inside the bus are much reduced
- But if more schools are added, the complexity of the operations makes the use of heuristics slower, one even spending more time that without heuristcs, even if they generate less nodes. The Deliver Bus heuristic is the main suffer of this quality.
- The bigger the complexity of the problem, the more the heuristics will help (which makes sense because when we are not using heuristics we expando lots more nodes)

Once we have look at the conclusion and at the test we can say that pick Student is the best heuristics from all the different that we have thought of.

Personal Thoughts

The assignment has been quite challenging in the coding aspect of it. Most of the ideas presented and models were designed quite easily but transforming them in code was a huge gap. Nevertheless, it provides good tools that has help us to understand in a little bit of depth both CSP and heuristic Search.

Another one of the main difficulties was finding possible heuristics. First of all, we designed one that depended on the number of students without taking into account any of the cost. This has help us realised that those kind of estimations are not heuristics, which at first seemed like something weird.

One thing that still doesn't make sense for us, is the necessity of including a bash script for executing the code when the python file can be directly executed without much trouble. It's good knowledge to have but it has wasted time that could be focused in making a cleaner code.

All in all, it was a fine exercise that in the end took much more time that we had initially thought of, due to misconceptions and overestimation of our coding abilities.