Second Lab Assignment

Constraint Satisfaction and Heuristic Search Tasks

Group 89
Ignacio Talavera Cepeda - 100383487
Luis Rodríguez Rubio - 100383365

Index

Introduction

The goal of this second lab assignment is to get familiar with the modelling and implementation of real-world constraint satisfaction problem (CSP) using python-constraint model, and to implement a functional heuristic search with the A* algorithm and different heuristics in order to solve a version of the pickup and delivery traveling salesman problem which involves a scholar bus system.

Our objective with this document is to make a clear and complete documentation of our projects and to explain every decision and step taken. This has a big importance in this project, as it is composed by several parts which have different files and implementations. May this report serve as a correct map to traverse through our work.

We will divide the report in several parts, following the schema of description of the models, analysis of the results and conclusion. Beneath this, and as the problem has two very differentiated parts, we will divide the whole document into the first and the second half, and each one will have the previous parts. This structure can be clearly seen in the index.

This report will not include in any case source code. The explanations will be made focusing on the main ideas supporting the code and in the design decision taken, not on the real implementation. The programming language of choice is, in both cases, Python 3. The version used for the compilation and run of the programs is Python 3.7.3 64-bit.

Part 1: CSP

Description of the model

In this first part, we face a time-tabling problem. We have to imagine the scenario of a school, where teachers must assign to each class hour a subject, and they must decide which one teaches which subject. It is a problem that teachers face in the real world every year of their careers. The school has six different subjects (Natural Sciences, Social and Human Sciences, Spanish Language and Literature, Mathematics, English and Physical Education) and three teachers (Lucía, Andrea and Juan). There are three hours of lecture from Monday to Wednesday, and two on Thursday. Friday is free (this is the one part of the problem that does not happen in real life).

There are eight different constraints we must satisfy:

1. Each lecture last one hour, and only one subject can be lectured: this constraint is satisfied without further implementation in our model of the problem, which we will discuss later.
2. All subjects should be lectured twice per week but Physical Education (PE from now on), which will be lectured once: we resolved this constraint introducing the subjects of the week in a list and limiting the number of instances of each one on 2 (except PE, which is on one).
3. Socials and Human Sciences (Socials from now on) should be lectured in consecutive hours: our approach is similar than the second constraint. We made a list of all lecture hours. We restricted that the first instance of Socials appears on the second, fifth or eighth position of the list (which will mean than the first hour of Socials happens on the last lecture hour of a day, and thus it cannot have consecutive hours), and then we checked that, once we found the first instance of Socials, the second must be in the next lecture hour.
4. Mathematics should not be lectured the same day than Natural Sciences (Naturals from now on) or English: this is a straightforward checking. We find in which day of the

week there are instances of Mathematics, and we make impossible for Naturals or English to be there.

5. Mathematics should be the first in the morning, and Natural Sciences should be last: it is also a very straight-forward comprobation, forcing Maths to appear on the indexes 0, 3, 6 or 9 of the list made with all the lecture hours (first hours in the morning) and forcing Naturals to appear on the indexes 2, 5, 8 and 10. We have placed this constraint before of the fourth, as it will save as computations (Naturals and Maths will be constricted, less checking will be required)
6. Each teacher should lecture two different subjects, that should be different than those of her/his colleagues: we will count the number of subjects that has each teacher, and then we will check that this value is 2.
7. Lucía will lecture Socials provided that Andrea takes care of PE: or, in other words, if Andrea teaches PE; Lucia will not teach Socials. It is a direct checking.
8. Juan wants to lecture neither Naturals nor Socials, if any of these are allocated first in the morning either on Monday of Thursday. This is the most important constraint of the problem, as it relates the two subproblems (Subjects to teachers and Subjects to Hours).

We started modelling making a clear separation of the two subproblem. Until we saw the last constraint, it made sense that this conflict could be resolved in these two steps, that could be independent from each other:

a) Assign two subjects to each teacher
b) Assign a subject to each lecture hour

But, as we read carefully the constraints, we discovered the need of modelling it in just one problem. There, we only had one conflict: how to make the program assign two different subjects to each teacher while keeping the overall stable other part of the problem working. We maintained the idea of the independent problem but introducing them in the same problem. So, we have as variable a set with the lectures (which has as domain the different subjects) and a set with the subjects (which has as domain the different teachers).

Our constraint network P is defined as the tuple $P = (X, D, R)$, where the different elements of the tuple are:

- *X* is the set of variables; $X = \{'M1', 'M2', 'M3', 'T1', 'T2', 'T3', 'W1', 'W2', 'W3', 'Th1', 'Th2',$ 'PE', 'Naturals', 'Socials', 'Maths', 'English', 'PE'$\}$
- *D* stands for the domain, the possible value for each variable, and is a set of sets, as each variable has a different domain. We can define the starting domain for all variables, prior to arc and path consistency, (where the domain of each variable will be reduced according to the constraints) as $D_i = 'PE', 'Naturals', 'Socials', 'Maths', 'Spanish', 'English\ \forall_i \in \{x_0, \ldots, x_{10}\}$; $'Lucia', 'Andrea', 'Juan' \forall_i \in \{x_{11}, \ldots, x_{16}\}$
- *R* is the set of constraints, defined as a binary relationship between values of different values. Our constraints could be denoted as $R = \{R_1, \ldots, R_8\}$.

Analysis of results

After coding *P* into Python and using *python-constraint*, we get the following solution first solution:

Monday 1	English
Monday 2	Spanish
Monday 3	Naturals
Tuesday 1	Maths
Tuesday 2	Spanish
Tuesday 3	PE
Wednesday 1	Maths
Wednesday 2	Socials
Wednesday 3	Socials
Thursday 1	English
Thursday 2	Naturals

Naturals	Lucia
PE	Juan
English	Andrea
Spanish	Lucia
Maths	Andrea
Socials	Juan

This global instation of our model, where all variables have been assigned a value from its domains, holds into our set of constraints *R*. Our computer spent 20,10 seconds to get it.

Test Cases

In order to have a deeper understanding of the problem, we have tried different variations to the given problem. First of all, we can make the problem a little bit more relaxed by adding a new lecture hour, the third on Thursday. A quick update and some of the constraints must be done, and after execution we can see that the time spent to get one solution has growed exponentially. The same happens when a teacher or a subject is added. This happens due to the way that *python-constraint* works; it checks all the possible value assignment for each variable. Therefore, adding a constraint to the first subproblem will lead the program to check 6^{12} instead of 6^{11}.

Some modifications to the constraints could also be made. We could relax a little bit the problem, for example, by erasing the fourth and fifth constraint, which may seem like restrictions without sense in the statement (in a real world problem they will be justified for sure, but as we manage a subproblem where we fit the schedule with the necessities and preferences of every teacher, they seem unused). Without these constraints, as the program has to made two less comprobations at each possible instation, the time to executed is reduced linearly.

Conclusion

After making these test cases, we can see how the problem and the *python-constraint* module works together. Our modelling is divided in two subproblem, and as python-constraint checks each possible value of the variables, it will check $6^{11} \cdot 3^6$ possible solutions. The constraints take linear time to execute, as they have to be checked in every candidate solution. Therefore, the thing that influences the most to the time at which the program executes is the number of variables and their possible domains (this is exponential).

Nevertheless, this process has a lot of potential, as CSP processes can aim to solve problems that in other way could be much more complex. It is a very, very powerful weapon.

Part 2: Heuristic Search

Description of the model

In order to give a proper description of the model developed to solve this problem, we must define the problem space (explaining the states, the operators set, the initial state and the goal state), the way that the problem are represented, an explanation of the A* and how it solves the problem and the different heuristics implemented.

This problem is a variation of the so-called Pickup and Delivery Traveling Salesman Problem (PDTSP), which tries to get the optimal path for an agent to move through a different graph, picking up elements for some places and delivering them to them destination. In our specific case a school bus must travel through a graph, picking children up from the nodes and delivering them to their respective schools.

We have designed the problem state as a *Python* class, with three types of elements: unmutable parameters, mutable parameters and class functions. As the unmutabble parameters, we will store all the information that remains the same in each possible state of the problem. The information that will change, depending on the given state, will be stored as mutable parameter, and it will allow us to distinguish between states. Some useful actions will be performed via class functions.

- Unmutable parameters:
    - Grid: information about the graph that the algorithm is traversing.
    - Max Capacity: maximum number of children that can be in the bus at the same time.
    - School Positions
    - Initial Position
    - Floyd: Floyd-Warshall matrix, were the minimum cost to go in between each pair of nodes will be stored. This element will be useful for the implementation of our heuristic.

        - Heuristic: number representing the heuristic chosen from all that we implemented for the problem.
    - Mutable parameters:
        - Current Position (of the bus).
        - Current Capacity: number of children already in the bus.
        - Pending Children: the children waiting to be delivered and the school to where they should be delivered.
        - Onboard Children: children inside the bus and their destination.
        - GScore: cost from the initial state to the current state.
        - FScore: sum of the GScore and the heuristic cost (calculated when the constructor is invoked).
        - Parent Node: node from which our current node was expanded
    - Class functions:
        - Equal override: override of the equal operator in order to compare states just by current position, pending children and onboard children.
        - Hash override: override of the hash operator in order to hash the states considering the current position, pending children and onboard children.
        - Is Goal: Boolean function which returns true if the node is the goal, and false otherwise.

As operators, we have the following:

Operator	Precondition	Effect
Movement	The current state has a neighbor to which it can travel	The next state will change its current position to the position to which the movement has been applied
Pickup	In the position of the current state there are children waiting to be picked and the bus is not full	The children disappear from the position where they were waiting the bust and are added to the current capacity and the on-board children of the bus
Delivery	In the position of the current state there is a school and on the bus there is one or more children that has as destination that school	The children disappear from the current capacity, the on-board children and the pending children

Our graph is represented as $G = (V, E)$, where $V = \{1, \ldots, n\}$ is the vertex set and $E = \{(i, j): i \neq j,\ i, j \in V\}$ is the edge set. With E is associated a cost matrix $C = (C_{ij})$ which represents the cost of moving through nodes. We have designed a parser that handles the input file and extracts and stores the information about edges, travel costs and the positions of children and schools.

As search algorithm, we are implementing A*. It is an informed search algorithm created by Peter Hart, Nils Nilsson and Bertram Raphael and first published in 1968. From a starting node of a graph, the algorithm is aimed to find a path to the goal state with the smallest cost. It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each iteration , A* needs to determine which of its paths to extend. It does so computing the cost of the path and an estimate of the cost required to extend the path all the way to the goal, following the formula $f(n) = g(n) + h(\mathrm{n})$, where $g(n)$ is the cost from the initial node to the current node and $h(\mathrm{n})$ is the heuristic estimation of the cost from the current node to the goal state. It is a complete and admissible algorithm, which means that it will always find the least-cost solution if and only if the heuristic implemented is admissible (if it never overestimates the real cost). The time complexity of this algorithm is $\theta(b^d)$ and the space complexity is $\theta(b^d)$.

Using A*, it is very important to use admissible heuristics functions. By doing so, we will ensure that we maintain the property of the A* algorithm which always returns the least-cost path from the start to the goal. We have proposed two different heuristics, both based on the Floyd-Warshall algorithm. This algorithm, proposed in 1962 by Robert Floyd and Stephen Warshall (almost simultaneously, but working independently from each other), is an algorithm for finding shortest paths in weighted graphs. It has a time complexity of $\theta(|V^3|)$ and a space complexity of $\theta(|V^2|)$

The first heuristic is called Minimum Pick-up. Using the matrix provided by Floyd-Warshall algorithm (which gives us the minimum travel cost in between each pair of nodes), this function evaluates node as the sum of the minimum distance from the current position to each pickup position and from the current position to the initial position. This heuristic relaxes the problem,

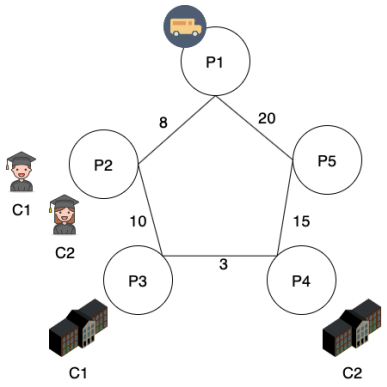
just seeking for children to pick-up and ignores the rest of the partial objectives of the problem. It is admissible, as it will never overestimate the cost: it is only evaluating the action of picking up and going back to the start node. When traversing through the graph, the real cost will be higher, as it will include the distance to move between pickups and deliveries. When there are not more pending children, the heuristic will return the perfect estimation, the real cost to the initial position. It is also consistent, as the Floyd-Warshall algorithm follows the principle of Optimality, which states that an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. As all optimal paths are formed by optimal subpaths, these calculations never overestimate distance from any neighbor vertex to the goal, plus the cost of reaching that neighbor.

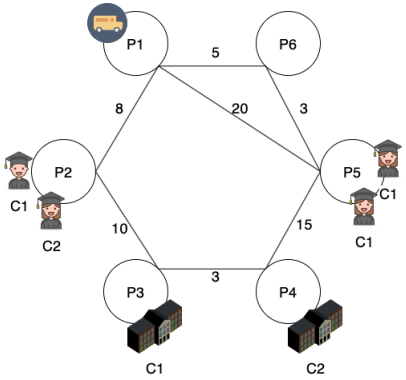
Our second implemented heuristic is called Minimum Cost to Access Children. Using the matrix provided by the Floyd-Warshall algorithm in a very similar way than in the previous heuristic, it will compute the ratio between the number of children of each stop and the maximum capacity of the bus. This ratio is truncated and added 1 and represents the number of times the bus will need to get to that stop in order to pick up every child waiting. This number obtained is multiplied by the edge with minimal cost connected to this stop as at very least the bus will spend that arriving that node N times, being N, the ratio explained before. This heuristic is admissible and consistent even though we take into account number of pending children (when picking and dropping cost is zero) because we don't just stick to this value, we calculate number of times we need to visit a compulsory node and multiply this by the minimum 1-step cost to get to that node so we are taking as reference value the best option the search algorithm could take avoiding overestimate total cost.

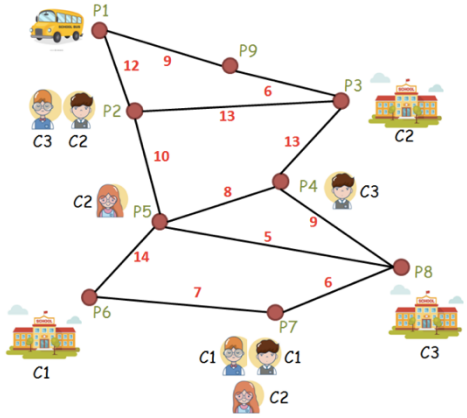
Analysis of results and Test Cases

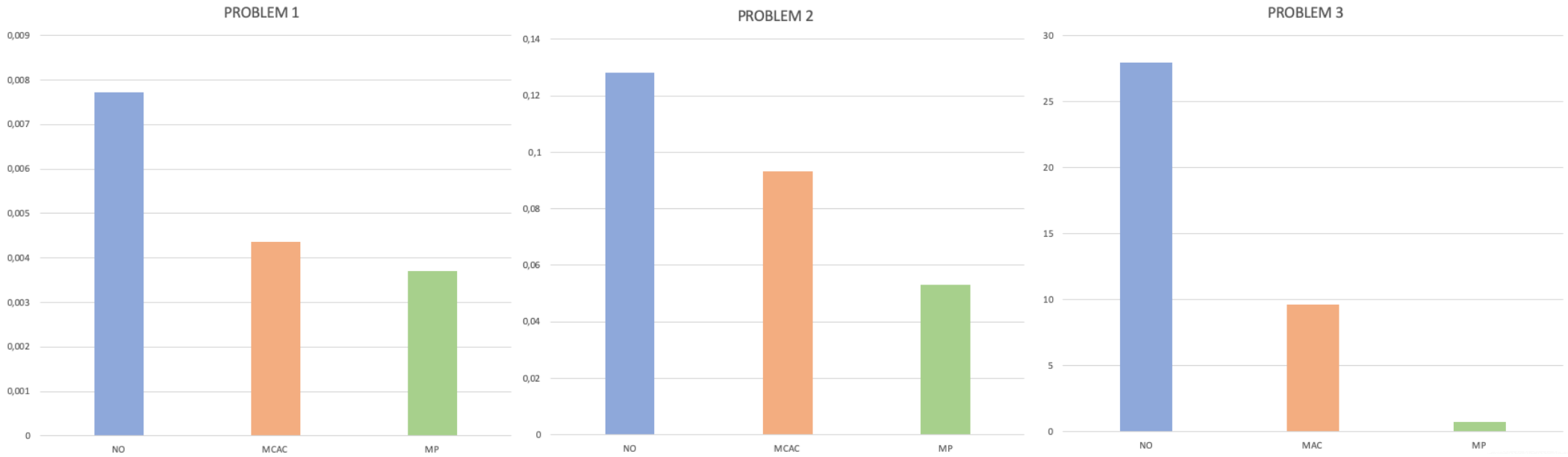
The proposal of different test cases and performance measure using the different heuristics is key to properly understanding the problem and the efficiency of different heuristics. We will perform our test cases using the three example problems, and we will compare the running time of 10 tries and compute the average.

Time in seconds\Case Problem	Case 1 (optimal cost = 42, number of stops = 7)	Case 2 (optimal cost = 58, number of stops = 11)	Case 3 (optimal cost = 102, number of stops=12)
No heuristic (Dijkstra's)	Time = 0,0077257s Expanded Nodes = 32	Time = 0,1282412 Expanded Nodes = 214	Time = 27,9300741 Expanded Nodes = 6624
MP Heuristic	Time = 0,00371561 Expanded Nodes = 14	Time = 0,05309609 Expanded Nodes = 85	Time = 0,75830289 Expanded Nodes = 384
MCAC Heuristic	Time = 0,00435993 Expanded Nodes = 17	Time = 0,09323949 Expanded Nodes = 130	Time = 9,653278 Expanded Nodes = 1554









In these graphs, we can see in a visual way the time performance (in seconds, in the y axis) in between all three heuristics (NO stands for no heuristic, when the Dijkstra's algorithm is applied). Both heuristics work well, but the performance obtained with the MP heuristic (specifically on the bigger, third problem) is surprisingly good. The MP has an overall speedup of 38.83 over the no heuristic function. Differences between our heuristics become more significant as the problem gets bigger (number of children and schools increases) expanding 94,2% less nodes in our best test case (Dijkstra vs MP in problem 3).

Conclusions

This second part of the lab assignment has taught us so much about the behavior of search algorithms and, in specific, of the procedure of guided search algorithms.

Implementing the algorithm was a delicate part, more that we could imagine when we studied the algorithm in class, without getting into code. Even though we had the overall idea and the behavior that it should have, it was thought to keep the performance and the complexity in its place while getting the code to work. The election of *Python* as language helped us a lot. It may not be the most efficient programming language, but its grammatic and built-in functionalities made our work easier.

We have seen the importance and the potential of heuristic functions. The process of designing them was very challenging at first, and we had to get through a lot of documentation. We read about constraint relaxing and PDTSP. The experience of the last year's *Artificial Intelligence* project also guided us onto the good direction, but the heuristics implemented then could not be used: we had a grid instead of a graph (with terrain costs) and derivations of Manhattan and Euclidean distances makes no sense in this problem.

The knowledge of Dynamic Programming algorithms was the key to develop powerful heuristics to these problems as we had two different tools to make efficient calculations of least-cost paths: Bellman-Ford-Moore and Floyd-Warshall algorithms. We finally decided to stick to the second one because of its capability to compute this path between all pair of vertices at once (Bellman-Ford-Moore algorithm may have a smaller complexity, but it would have been necessary to use it several times instead of one). We have achieved a significant reduction of the expanded node and the performance time, and we are very happy and satisfied with the results obtained in the test cases.

Overall Conclusions

The work done about CSP and guided search algorithm has been very satisfying, and it is very easy to see how powerful and useful these procedures are to solve real world problems. The uses that both can have are countless.

The first part turned out to be very quick, the only problem was to get familiar with the tool. But the modelling was very interesting, and we added a very useful tool to our ability pool. In the second part resided the difficulty of the lab assignment, and we started to get results very slowly. But we also gained confidence, and we experienced the implementation of the A* algorithm in practice and not only in theory. Getting through the creative process of the heuristic function was frustrating at first, but we started getting confident very quickly and we enjoyed the process of make assumptions, trying to turn them into real functions and testing them. We failed many, many times, but we also learnt a lot (and have so much fun).

This lab assignment has been really enriching. CSP was a totally new concept for us and it is always exciting to discover new fields that can have so much potential solving real-life problems; and the A* star implementation was the perfect way to learn, by designing and coding, such an important algorithm. We are looking forward to your feedback.