Lab assignment 2: *Constraint Satisfaction and Heuristic Search*

Heuristics & Optimization 2019-2020 (Group 89)

Marcos Gonzalez Carrasco (100383355@alumnos.uc3m.es)
Raul Olmedo Checa (100346073@alumnos.uc3m.es)

December 15, 2019

Contents

1 Introduction **2**

1.1 About this lab assignment . . . 2

1.2 Document contents . . . 2

2 Problem Modelling **3**

2.1 Part 1: Time-tabling . . . 3

2.1.1 Variables . . . 3

2.1.2 Domain . . . 4

2.1.3 Restrictions . . . 4

2.2 Part 2: Heuristic Search . . . 5

2.2.1 Design Of The States . . . 5

2.2.2 Operators . . . 6

2.2.3 Heuristic Functions . . . 6

2.2.4 Initial State . . . 6

2.2.5 Final State . . . 6

2.2.6 A* Algorithm . . . 7

3 Analysis of results **8**

3.1 Part 1: Time-tabling . . . 8

3.2 Part 2: Heuristic Search . . . 8

4 Conclusions **10**

Chapter 1

Introduction

1.1 About this lab assignment

This report contains all the information related to the second lab assignment of the heuristics and optimization course. In particular, the tasks required for this assignment were to model and solve a time-table assignment problem as a constraint satisfaction problem (CSP) and to route a school bus through the different stops and schools using heuristic search algorithms. The assignment and therefore this document is organized into two sections:

Part 1: model and solve using Python in conjunction with the library Python Constraint the time-tabling problem presented by the school as a CSP, obtaining one solution that fulfills all constraints presented in the assignment.

Part 2: model as an heuristic search task the stops that a school bus must make to be routed in the most optimal way through the different bus stops (where it must pickup students) and schools (where it must drop them).

1.2 Document contents

This document has been divided into four chapters trying to follow the structure proposed in the assignment's manual. Below there is a description of the content for each one of them:

Chapter 1 (the current one): provides an overall description of the work carried out in the laboratory and the structure of the document.

Chapter 2: the modeling of both parts is presented to the reader alongside with the constraints of each part of the assignment. All design decisions carried out during the practice are stated in this section.

Chapter 3: analyzes the results obtained during the testing phase of the assignment (for both parts) and answers the questions proposed in the assignment's guide.

Chapter 4 explains the difficulties we have faced while completing this lab assignment, both from a technical and a conceptual standpoint.

Chapter 2

Problem Modelling

2.1 Part 1: Time-tabling

The school has contacted us to schedule the classes for a certain course, the information they provide consists on the available time slots, subjects and teachers, also indicating the different constraints that must hold, complicating the manual assignment.

- The subjects are distributed among the week from $Monday$ to $Thursday$ and from **9-10** to **11-12** (3 hours each day) except $Thursdays$, where the classes end at **11**.
  In order to denote this slots, we used a subset of $\mathbb{N}$ in the following way:

	Mon	**Tue**	**Wed**	**Thu**
9:00 - 10:00	0	4	8	12
10:00 - 11:00	1	5	9	13
11:00 - 12:00	2	6	10	

Table 2.1: Schedule

  The gap between days is intended to mark the change between days (simplifying later the checking of consecutive classes).

- The variables are denoted by *NSC*, *HSC*, *SP*, *MAT*, *EN*, *PE*, which correspond to the six subjects that the school teaches in that course. To allow us to specify the number of hours per week that each subject has, a number is added at the end of the names above stated (e.g. *HSC* is decomposed into *HSC1* and *HSC2*) except *PE*, that is the only subject taught once during the week.

- Finally, the three teachers are denoted as AND, LUC and JUA, since each one of them must teach two subjects, we append a number after these names (e.g. LUC will be decomposed into *LUC1* and *LUC2*).

  The domain of the variables regarding teachers is denoted with numbers that are related to the subjects in the school, therefore:

$$NSC = 0,\ HSC = 1,\ SP = 2,\ MAT = 3,\ EN = 4,\ PE = 5. \tag{2.1}$$

2.1.1 Variables

Below there is the definition in mathematical notation of the variable set for the constraint network:

$$X = \{NSC1, NSC2, HSC1, HSC2, SP1, SP2, MAT1, MAT2, EN1, EN2, PE,\\ , LUC1, LUC2, AND1, AND2, JUA1, JUA2\} \tag{2.2}$$

2.1.2 Domain

Initially, the domain is the following:

$$D_i = \{0, 1, 2, 4, 5, 6, 8, 9, 10, 12, 13\}, \quad \forall i \in X \setminus \{AND1, AND2, LUC1, LUC2, JUA1, JUA2\} \tag{2.3}$$
$$D_i = \{0, 1, 2, 3, 4, 5\}, \quad \forall i \in \{AND1, AND2, LUC1, LUC2, JUA1, JUA2\} \tag{2.4}$$

After reading the constraints of the problem, we can delete all values of the domain in which the *NSC* subject is not on the first hour of the day, resulting in: $D_{NSC1} = D_{NSC2} = \{2, 6, 10, 13\}$. The same happens with the domain regarding *MAT*, $D_{MAT1} = D_{MAT2} = \{0, 4, 8, 12\}$, since the problem forces it to be at first hour. Therefore, our final domain for this variables is:

$$D_{NSC1} = D_{NSC2} = \{2, 6, 10, 13\} \tag{2.5}$$
$$D_{MAT1} = D_{MAT2} = \{0, 4, 8, 12\} \tag{2.6}$$

2.1.3 Restrictions

The following restrictions are presented in the document:

- Each lecture lasts one hour, and only one subject can be lectured.
  The first part of this constraint is fulfilled during the domain selection phase, we denoted each one of the possible slots with a natural number, enforcing in this way a 1-hour class in each slot.
  For the second part:

$$i \neq j, \quad \forall i \in X, \ j \in X \setminus \{i\} \tag{2.7}$$

- All subjects should be lectured two hours every week, but Physical Education, that should be assigned only one hour.
  This constraint is modeled during the variable selection phase, we chose two different values for each subject, denoting the number of hours per week that they must be taught.

- The two hours devoted to each subject can be lectured consecutively or not (or even on different days), but Social and Human Sciences that should be lectured in two consecutive hours.

$$NSC1 + 1 = \ NSC2 \ \lor \ NSC2 + 1 = NSC1 \tag{2.8}$$

- Mathematics should not be lectured the same day than Natural Sciences or English.

$$MAT1 \lor MAT2 \in i \implies NSC1, NSC2, EN1, EN2 \notin i, \quad \forall i \in days \tag{2.9}$$

  where

  days: set denoting the domain values grouped by day as denoted in 2.1

- In addition, Mathematics should be lectured first in the morning, and Natural Sciences should be last (note that we took the second hour on Thursday as last for that day).

$$MAT1, MAT2 = \{0, 4, 8, 12\} \tag{2.10}$$
$$NSC1, NSC2 = \{2, 6, 10, 13\} \tag{2.11}$$

- Each teacher should lecture two different subjects, that should be different than those of her/his colleagues.

$$AND1 \neq AND2 \neq LUC1 \neq LUC2 \neq JUA1 \neq JUA2 \tag{2.12}$$

- Lucia will lecture Social and Human Sciences provided that Andrea takes care of Physical Education.

$$AND1 \vee AND2 = 5 \implies LUC1 \vee LUC2 = 1 \tag{2.13}$$

- Juan wants to lecture neither Natural Sciences nor Social and Human Sciences, if any of these are allocated first in the morning either on Monday or Thursday.
  Since in 2.5 we reduced the domain for NSC, it is not required to include *NSC1* nor *NSC2* in the constraint below, since *JUA* will always be able to teach it.

$$HSC1 \vee HSC2 = \{0, 12\} \implies JUAN1, JUAN2 \neq 0 \tag{2.14}$$

- Although this was not requested, we chose to add another constraint to avoid the duplicated solutions that are included in the model chosen. All tuples of variables (e.g. *MAT1* and *MAT2*) containing the slot assignment are forced to be instantiated in ascending order (*MAT1* is always before *MAT2* and therefore its value will be always lower, according to our design).

$$i > j, \quad \forall (i, j) \in S \tag{2.15}$$

  where

  S: set with the tuples containing all variables related to a subject.

- Analogous to what we did on the previous constraint, we force teachers to get the subject assignment in ascending order to avoid the repetition of solutions.

$$i > j, \quad \forall (i, j) \in T \tag{2.16}$$

  where

  T: set with the tuples containing the subject assignment for each teacher.

2.2 Part 2: Heuristic Search

2.2.1 Design Of The States

The states in this problem must represent the position of the bus and the children, while also containing as information the layout of the graph. For the second part, an auxiliary object has been created called node. Nodes contain the following information:

type: This stores if the node is a school or not

name: The name of the node, such as P1 in the example

schoolName: The name of the school, it is left empty for stops

adjacent: This stores the index of the nodes adjacent to this one

cost: This stores the cost of moving to the adjacent nodes

States themselves follow this structure:

buspos: Index of the node the bus is currently at

capacity: The number of students the bus can hold

g: The cumulative cost up to this state

initPos: The index of the initial position of the bus

nodeList: The list of nodes of the graph, containing node objects (a pointer to the list for programming purposes) *studentsInNodes*: The representation of where each student is within the graph. Nodes were decided to be immutable, so it has been moved to the state. It follows this organization: (((1,C1),(2,C2)), (), ((2,C3))), where empty slots are left for all nodes to simplify access and storage. CX is the school the student belongs to and the number is how many students from that school are within the node

students: The list of students currently within the bus

parent: A pointer to the parent state for retracing the solution

2.2.2 Operators

The operators take a state and return all possible children it can have. In this problem, the only possible operations are to move to an adjacent node, to pick up a student or to drop off a student. Only operating on one student at a time was considered, since otherwise optimality may not be guaranteed.

Moving students simply looks at the nodes adjacent to the buspos and creates new states for each one of them.

Picking up a student looks at the students within the current node in studentsInNodes only if the node is a stop or a school with a non-matching student, removes one of them and adds them to the students of the state. If the value in studentsInNodes reaches 0, the value-student pair is removed.

Dropping off a student checks if the node is a school, looks at the students within "students" in the state, picks one of them, checks if the student and school match and if so, removes them from students and adds them to studentsInNodes.

2.2.3 Heuristic Functions

We didn't manage to implement an admissible heuristic function for this problem, nonetheless this are the approaches we took:

- We tried to force the bus to pick up students by adding the minimum cost between the students and its corresponding school, for that task we used the Floyd-Warshall algorithm to obtain the costs between two nodes, with the final value of the heuristic function being the sum of the costs between each student and the school, however this is inadmissible because the bus can pick up more than one student at the same time.
  We also tried to count students with the same destination as one group, but even doing so it remained inadmissible, since the path that the bus takes is shorter than going directly between student and school for every student. Presumably this heuristic would be admissible with the bus capacity equaling to one.
- Another approach we tried was to relax the adjacency of the nodes, we assumed that students were able to reach its corresponding school with a unitary cost, but this provided almost no information and it was not strictly admissible (e.g. five students within one unit of cost from their school), when we tested it the result obtained was different from the one in the uninformed search and the cost was higher.

2.2.4 Initial State

This state is provided within the input file, parsed and transformed into the initial state that is inserted into open.

2.2.5 Final State

Checking if a state is the goal in this problem is fairly simple, all that must be checked is that buspos is initPos, that students is empty, that all stops in studentsInNodes are empty and that all students at a school match with said school.

2.2.6 A* Algorithm

This is pseudocode for the A* implementation used in this lab. It's a fairly standard implementation which checks if the successor already exists within open or closed with a lower f=h+g and ignores it if so. It is important for open to be a sorted list based on f.

```
Add initial state to open
While open is not ready
  Pop the first state from open
  If it is a goal, halt and obtain solution path
  Otherwise, generate its successors
  Add the current state at the end of closed
  For each successor
    If it exists in open with lower f, continue
    If it exists in closed with lower f, continue
    Otherwise, insert it into open in its correct position based on f
    Also, set its parent to the current state
If a solution was not found, report on it
```

Looking through open and closed to find repeat states and inserting successors in order is what takes most time, as it requires iterating through lists that rapidly expand as the problem becomes harder. Attempts were made to optimize this but since open must be ordered based on f and closed can't be ordered or else the pointers break, no search other than iterating through them was possible.

Chapter 3

Analysis of results

3.1 Part 1: Time-tabling

We performed several tests to see how the program performs by varying domain, variables and constraints and an additional one to check for incorrect assignments and as a baseline for comparing the number of solutions and execution time:

schedule.py: main program containing just one solution for the problem modeled.

schedule-test-1.py: taking into account the base program, we added one more lecture hour to the timetable and one more weekly hour to the subject Physical Education to test how many possibilities are added to the solution.

schedule-test-2.py: we added to the previous program another constraint regarding the two variables (PE1 and PE2) that forces those two hours to be on different days.

schedule-test-3.py: similar to what we did with schedule-1, we added one more hour and another English class to the timetable.

schedule-test-4.py: following the previous approach, we added one more constraint regarding English classes to avoid having them on the same day.

schedule-base.py: this program computes all solutions for the timetabling problem with the constraints required in the assignment, it also checks for any inconsistency in those solutions and measures the execution time.

Regarding tests number 1 (35208 *solutions*) and 3 (10368 *solutions*), the same number of variables with the same domain were added, although the number of solutions varies significantly due to the constraints affecting those variables:
In the first case we duplicated the variable storing Physical Education classes, which is only affected by the constraints 2.7 and 2.15, whereas the second case adds another English class, which is restricted by the same constraints plus 2.9, since the former is less restricted, the number of solutions is bigger than the latter, as can be seen at the beginning of this paragraph.

In tests number 2 and 4 we checked the same principle as above, we added the same new constraint in both problems, forcing Physical Education in *schedule-test-1* and English classes in *schedule-test-3* to be on different days. As we expected, the variable restricted by the most constraints obtained the less solutions (5184) compared with the one regarding PE classes (14832), even though initially both had the same domain.

3.2 Part 2: Heuristic Search

We performed several test to check the validity of our implementation, we included four of them into the solution of this problem.

no-solution-test.prob: we designed the problem to have no solutions in order to check that our program was able to explore all possible states and stop based on the fact that no solution was reached but all nodes were visited.

test-1.prob: this was the example provided in the problem statement and we used while we were developing the program.

test-2.prob: reduced problem with huge differences in the costs to ease the manual computation of our solution.

test-3.prob: small variation from the previous test by deleting just one node and changing the capacity of the bus to make the solution path longer and check its correctness.

test-4.prob: we used ten nodes and twelve students from three schools distributed among the stops, this test was intended to be a benchmark to get the point in which the program becomes slow by the number of expansions computed, however the solution was found in 3s.

test-5.prob: this test was exactly the same as the previous one, but we added two new students placed in the stop number nine, as result instead of spending three seconds obtaining the solution, it took more than three minutes to get the solution path.

Chapter 4

Conclusions

For the first part, although it was quite straightforward and took us almost no time to get our first solutions, we struggle to get the whole idea of the lab assignment, at first we though that it was focused on obtaining the minimum set of solutions with no repetition whatsoever, with that goal in mind we tried to reduce the time used to compute the solutions and the number of different combinations for the assignment, that is why we included some new constraints in our modelling and implementation.
We ended up refreshing our python knowledge and how to use the python-constraint library to perform this type of CSP assignment problems with no hassle.

With the second part we went through more problems than desired, particularly in the model implementation using C++ language.
At the beginning our program was poorly optimized and took a long time to compute the solution even by using small problems, so we had to change how the program worked to get better results. After some time we realized that the problem was related with the compiler embedded in VSCode, we switched to g++ with the optimization flags enabled and the problem ran with no problems.
We had some troubles too with the scope of pointers, we did not realize that some references were being destroyed, creating some nasty memory problems and cyclic references that took some time to debug and fix.
When the time to implement an heuristic came, we had no idea how to focus the development, it would have been nice to have this explained more thoroughly during classes, in order to reach this part of the assignment with more theory and examples than what we had at that point. Our main problem was the admissibility of the heuristic, since we always ended up overestimating the cost of the optimal solution (and the popular distance-based heuristics did not work well under this problem characterisitics).