uc3m | Universidad Carlos III de Madrid

Bachelor in Applied Mathematics and Computing
2022-2023

Heuristics and Optimization

Project 2: Constraint Satisfaction and Heuristic Search

December, 2022

Group 121

Github repository

Ignacio Gómez Fernández - 100429871@alumnos.uc3m.es

Gonzalo Prats Juliani - 100429904@alumnos.uc3m.es

CONTENTS

1 Introduction

This report contains the step-by-step analysis of the two proposed problems, constraint satisfaction and heuristic search.

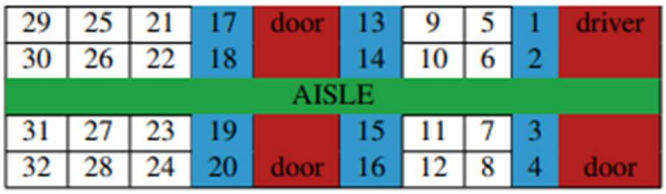
For the first one, we will provide a thorough explanation of our thought process. We started by defining the domains for each of the variables and, with the constraints set to meet the statement of the problem we arrived at the possible set of solutions, if any. Some of the test cases will return one, multiple or no solutions, depending on the restrictions on each domain and constraints

The second part was centered around a custom-made A* algorithm to make the queue for the bus the most efficient possible. This is what is called a heuristic search problem and has its foundations on the idea of defining a function (heuristic function) that will never over-estimate the cost of the optimal solution so that we can explore possibilities (via node trees) following both the cost and the heuristic functions

2 Part 1

2.1 Description of the problem

We are given the task of organizing several students into a bus. For this purpose, we will assign a specific seat number to every student and, of course, every seat will only fit one of them. However, there are some restrictions to the problem as not every student is the same, there are some of them that need to be taken care of in a different way. A reduced mobility student (R) will have a designated area in which they can seat as well as special rules for who can seat near them. Also, troublesome students (C) cannot seat wherever as they cannot seat near another troublesome student. With all these and the possible combinations for the seating arrangements, we must place each student from our list in a different seat, considering that this is the layout of the bus (blue seats are meant for reduced mobility students).



2.2 Problem modeling

As in every Constraint Satisfaction Problem, we must define first a constraint network $R = (X, D, C)$, where X represents a finite set of variables, D is the set of domains for each of the variables and C consists of the restrictions for the values those domains can take. In our case, we decided to model the problem as follows.

For the variables, we decided to take the students as they were the elements we would have to map to a certain seat around the bus while looking for the optimal solution. The other option

was to take the seats in the bus as the variables and assign them to the students but we didn't take that way as the constraint satisfaction would have complicated things.

$$X = \{students\} = \{s_{ID}\}_{ID=1}^{N} = \{s_1,\ s_2,\ s_3, \dots, s_N\}$$

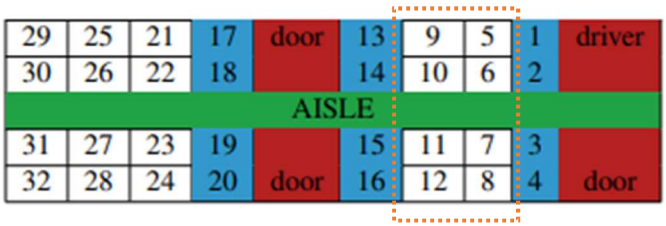
Then, for the domains, we decided to create several of them so that we could apply them to the specific variable we wanted. In a way, we were restricting certain general arrangements from the beginning by personalizing the legal positioning in the bus.

First, we defined the full domain

$$fulldomain = \{seats\}_{seats=1}^{32}$$

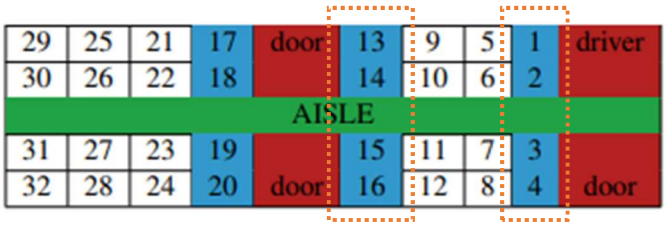
Then, we created the domain for first year students without a mobility restriction. There is a possibility that, if there are no reduced mobility students, every seat is available for the rest of students, but we will go into it later.

$$domain1yr = \{seats\}_{seats=5}^{12}$$



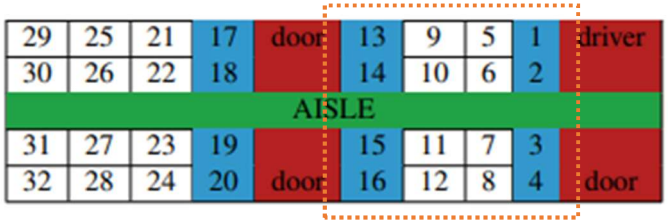
Then, complementary to this last one we had to define the domain for first-year students with a mobility restriction

$$domain1yrR = \{seats\}_{seats=1}^{4} + \{seats\}_{seats=13}^{16}$$



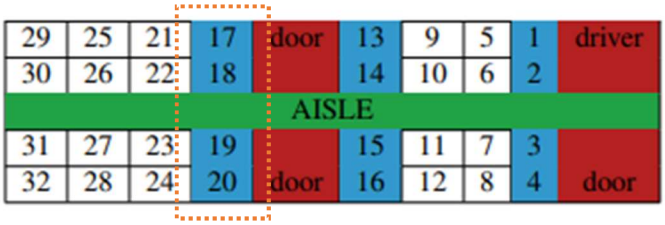
In fourth place, we created a domain for every first-year student, with and without a mobility restriction

$$domain1yrwithR = \{seats\}_{seats=1}^{16}$$



Moving on into the second-year students, we first defined the reduced mobility domain

$$domain2yrR = \{seats\}_{seats=17}^{20}$$

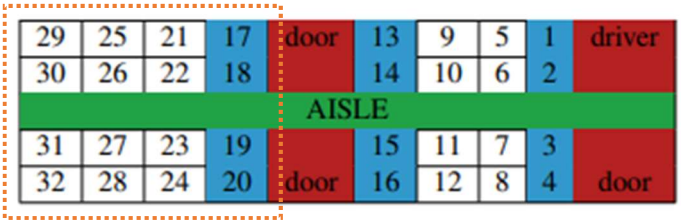


Second year students without a mobility restriction had to have a specific domain too

$$domain2yr = \{seats\}_{seats=21}^{32}$$

Finally, like the first-year case, we defined the union of those last two domains

$$domain2yrwithR = \{seats\}_{seats=17}^{32}$$



Once we had our variables and domains defined, we could have already started creating arrangements because we could map each student into a seat, but we were missing the last important part, the restrictions. One could see these restrictions as the function that creates the mapping for the variables and the domains, allowing only a certain number of values into the image of the function and discarding others that do not fit the desired criteria. With that in mind, we defined the restrictions.

- C_1: every seat must have a different student assigned to it. That is, no seat should have two different IDs
- C_2: no student can seat next to a reduced mobility student (the aisle counts as separation).
- C_3: siblings must be seated together. If they are in different years, the older one will seat in the first-year part of the bus. Also, the older sibling would have to seat in the aisle row. However, if either one of them is handicapped, then restriction C_2 applies

- C_4: a troublesome student cannot have another troublesome student or with reduced mobility near them. That is, any seat in contact with the one in question (the aisle does not count as separation in this case). For better understanding, here is a visual representation, where the yellow seat is the one with the troublesome student and the red ones the invalidated ones for either troublesome or reduced mobility students.

We can then model the restrictions as follows:

$$R_{bus} = \{(a_1, b_j), \dots, (a_{32}, b_n), j \neq n \; \forall j, n \in student_{id}\}$$

$$R_{redMob} = \{(a_{2n}, b_j), (a_k, b_n) | (a_{2n-1}, b_j), (a_m, b_n) \; where \; n = 1, 2, 7, 8, 9, 10 \mid b_j = student_{redMob} \mid a_k \neq 2n - 1 \mid a_m \neq 2n\}$$

$$R_{siblings}: \{(a_n, b_i), (a_m, b_j), b_i \; younger \; than \; b_j \rightarrow n, m \leq 16 \; and \; n \; mod \; 4 = 0 \rightarrow m = n - 1 \; and \; n \; mod \; 4 = 1 \rightarrow m = n + 1 \mid b_i, b_j \; same \; age \rightarrow n \; mod \; 4 = 1, 3 \rightarrow m = n + 1, n \; mod \; 4 = 0, 2 \rightarrow m = n - 1\}$$

$$R_{troublesome}: \{(a_n, b_i), (a_m, b_j), b_i, b_j \; troublesome, a_m \in neigbours(a_n)^c\}$$

With all these, we now have defined the constraint network of our problem and are ready to implement it into Python for its resolution.

2.3 Model implementation

The implementation of the proposed model above was done in Python and its resolution was done thanks to the library *constraint*, designed for problems like this one. It takes a set of variables, domains, and restrictions and applies its default Solver to print the solutions, if any.

So, going by the order in which we developed the code, we first had to get the students' information, since their characteristics were going to dictate the solutions and way to proceed. For that we used several I/O built-in python functions and organized the students with their attributes into an array. A dictionary with the students was also created for later use. The structure of each student in the list we were given was the following

$$student = ID, year, troublesome, mobility, IDsibling$$

where,

- ID: unique number that defines the student
- year: first (1) or second (2) year
- troublesome: student can be troublesome (C) or not troublesome (X)

- mobility: reduced mobility (R) or not reduced mobility (X)
- IDsibling: identifier of the student's sibling if they have one (0 otherwise)

Even though we had our list of students, we found that, since they all had different attributes, it would be easier for us if we divided them into their different categories, so we created arrays of students organized by their specific characteristics. First, we divided the list into four: regular students, reduced mobility students, troublesome students and finally, reduced mobility that were also troublesome. Second, we made the same differentiation but taking only the first-year students. And, finally, we did the same for the second-year ones.

We would also like to point out that we created a dictionary to keep track of the siblings. In it, the key would be the ID of the younger sibling and the value would be the older's. If they belonged to the same year, key and value were chosen according to lowest to highest ID ordering.

Apart from that, we also created an array for the bus seats, which would serve its purpose to map each student to one entry of the array.

We then had to add the variables to the problem and the way this is done in the constraint library is like defining a function, with the values and their domains. We could have just defined one *variable* object with all the students and the whole domain, but to simplify the restriction area, we created 4 *variable* objects, so that each segment of students was defined in their corresponding domain. These were the ones we defined and their domains.

$$S_{1XX} + S_{1C} \rightarrow domain1yrwithR$$

$$S_{2XX} + S_{2CX} \rightarrow domain2yrR + domain2yr$$

$$S_{1XR} + S_{1CR} \rightarrow domain1yrR$$

$$S_{2XR} + S_{2CR} \rightarrow domain2yrR$$

where S_{nij} indicates the student of n-year, i troublesome (C) or not (X), and j reduced mobility (R) or not (X). Notice how there is no overlap between the variables in terms of students, this is, no student appears more than once when adding all those variables up. Moreover, the set of students is fully covered with that definition of the variables, the only thing we did is divide their domains to facilitate the resolution later.

Then, each constraint was defined for the affected group, meaning that, if a restriction only applied to troublesome students, for example, it would only be defined by iterating over the set of troublesome students. To translate the described constraints into code, we also had to define several functions that are outlined and commented in the code, such as obtaining the neighboring seats of a specific one, checking whether two students are siblings or getting the translation from seat number to entry in the bus array.

One important part of the constraint definitions was the use of *lambda* functions. These are anonymous functions that are very helpful if the complexity of the function we want to define is low and we don't want to waste multiple lines of code by defining a regular function. We sometimes used them to set the restrictions as the *addConstraint* method takes two arguments, a function, and a set of values. Once again, please refer to the code in *CSPBusSeats.py* for the implementation and further explanation.

2.4 Analysis of the results and test cases

The results we obtained varied depending on the output, of course, but we added a line at the beginning of our output file that indicated the number of solutions instead of listing all of them. However, we did print the first 10 solutions so that the user can check that they are correct. We will now analyze some of the test cases we developed, the ones we consider to be the most representative, but for further cases, please refer to the folder *./CSP-tests*

2.4.1 Regular case (*students2.txt*)

This first case consisted of a regular input, meaning just random students with their attributes, to check that everything was working fine. We inputted 8 different students, from which 3 were first-year and 5 were second-year, 3 troublesome students, 2 with reduced mobility and among which there was a pair of siblings. With these parameters, we obtained 576576 possible configurations for the seating.

2.4.2 Everyone has their sibling on the bus (*students_test.txt*)

This time, the input was very similar to the previous case, with the difference that every student had their sibling among the other students. We also took this opportunity to test the different possibilities outlined for siblings (both first-year, both second-year, different years and troublesome or reduced mobility combinations). With all this, the 8 students had 353280 different combinations of seats arrangements, which is almost 40% less combinations than in the previous case. It makes sense since we are restricting the available seats.

2.4.3 High IDs, low number of students (*studentsHighId.txt*)

With this test, we wanted to accomplish two things. First, check that having a high ID would not affect the model (since IDs do not imply an order of any kind), and second, we also wanted to check that the model was working properly with a manageable number of solutions. So, we only had 3 students as input and none of them were troublesome nor had any mobility restriction: the first two (IDs 3 and 30 respectively) were first-years and the last one (ID 32) was a second-year and sibling with ID 30. With all these, we knew that we would only seat the students in the front part of the bus and that the siblings had only one configuration, the older one would always be seated in the aisle side. Due to its simplicity, we could calculate the number of possibilities:

- Number of possibilities for the siblings: 8
- Number of possibilities for ID 3: 14 (since two seats were already used by the siblings).

Notice that we did not consider the reduced mobility seats as there is no student with a mobility restriction

Finally, we could conclude that the total number of legal arrangements was $n_{solutions} = 8 \times 14 = 112$. And when we ran our model, we got the exact same number, which implies a correct functioning of the model.

2.4.4 Tests with troublesome students (*students_ntroubledSameYear.txt*)

The next critical part of the tests were the troublesome students, we had to check that the model would not seat any of them near each other. These are two tests in which we inputted *n* troublesome students.

For the first one, we used 4 and got 1896 solutions which made sense because none of them had a mobility restriction and thus, they could take up the whole *domain1yrwithR*.

On the other hand, when we used 5 troublesome students, the number of solutions dropped to 0. This was our expected result because, due to the number of seats in the bus, there is no physical way of seating them all without taking any of the second-year space (in our scenario they were all first-years but we would have gotten the same solution otherwise).

2.4.5 Tests with reduced mobility students (*students_ndisabled.txt*)

Finally, the last part was about reduced mobility students. In their case, our options were more limited since there were only a certain number of seats for them and they had a restriction that cut in half the available seats (each time a reduced mobility student was placed, the neighboring seat was not free anymore). As in the previous section, we tested these with *n* reduced mobility students

The first case, with 6 reduced mobility (4 first-year and 2 second-year), we obtained 3072 solutions. In them, all reduced mobility seats were used and, thanks to the different possibilities in both the chosen seat (could be either the aisle or the window seat) and the different combinations for the students, the number of solutions was as high as it was.

When we tested the case in which we had 7 reduced mobility students, we obtained 0 solutions, which made sense since there is no way to arrange them without making two of the students be next to each other (constraint C_2)

3 Part 2

3.1 Description of the problem

In this case, we are given a list with all the students that are to board the bus. Our objective is to organize them in a way that minimizes the time needed to enter the bus, considering certain differences in the time each students takes depending on the type they are. We must model the creation of the queue (out of all the possible combinations) so that the final cost of the whole process is the minimum.

3.2 Problem modeling

As described above, we have here a search problem and so, we must model it like one. The main characteristic of these kind of problems is that we can represented in a graph, where each configuration of the queue is represented by a node, and solving it means searching for the goal state (the queue filled with all the students) from a number of different paths. The key elements of this graph are the following.

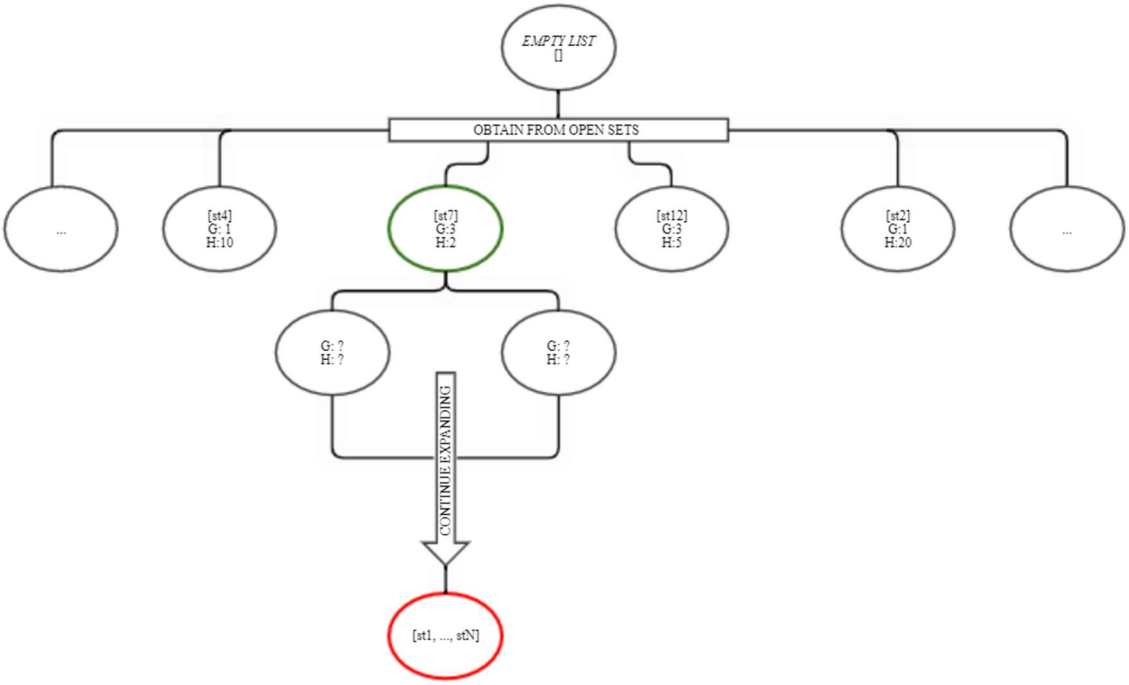
- *Set of states*: it is the current configuration of the queue, whenever we arrive at the meta state, we would have the complete list of students, ordered so that their cost is minimum
- *Operators*: these are the set of the additions to the queue and their corresponding cost. They are not constant as the cost depends on the type of student that is to be added. The restrictions given in the problem generate the set of states
    - *Regular operator*: whenever a non-troublesome and non-reduced mobility student is added to the queue, the total cost of the process increases by 1 unit
    - *Reduced mobility operator*: reduced mobility students take 3 units of cost to board. They also need a student to help them, in which case both students will take 3 units of time to board the bus (cost to reach the next state is 3). However, if the student helping is a troublesome one, then the overall time for both goes up to 6 units.

- *Troublesome operator*: a troublesome student will double the boarding time for the student in front and behind them in the queue. Also, every student boarding after them that has a higher seat number will also take double the time.

- *Initial State*: we will start with an empty list as no student has been assigned to a position in the queue.
- *Goal State*: all students have now been included into the queue. Once we reach this state, it means that all students have their position in the queue and our ordering is finished. Because of the restrictions of the problem, we cannot reach this state by adding a reduced mobility student last (last node), so we will take that into account when implementing the model.

We would also like to make two appreciations about the model. First, we could see the search in the graph as a tree since we get different possibilities from each node in terms of adding a new student. This way, when all nodes are expanded, we would get an inverse triangular shape for the search tree (the nodes available at each step decreases as fewer students are to be assigned). And second, the depth of the search tree is going to be, at most, the number of students that form the line because each operator adds one and only one student to the queue.

Then, to give a more visual representation of the model and how it would work, we will provide you with this diagram. In it, you will see that we will start from an empty list (initial state / node), and we expand it according to what's in the open set. For example, the first time we expand, we will get N nodes (being N the number of students), each with only one student in its list and a different one each time, since they are all available at the beginning. We will then choose the node with the lowest G (cost) + H (heuristic), which is marked in green as an example and continue expanding this way until we populate the list with all the students in it (red node in the end) and in the correct order



That last node represents the goal state and contains the cheapest configuration for the queue of students considering all the restrictions.

2.3 Model implementation

The complete implementation was made in Python and the only external library used for its development was *ast*, which we used to traverse the data we extracted from the original list of students (input) and pass it to the function as a dictionary.

Again, we will go through the decisions we took in the order in which they are written in code.

We first gave shape to our model with two necessary data structures. The first one is a *Node* class, which would store the information of the search state up until that point. For that, we defined several attributes for it:

- *state*: a list that stores the students that have been already assigned a place in the queue. This list is already ordered by the minimum cost to enter the bus and the first element is the student that would go in first
- *remaining*: list of the remaining students to place in the queue, they are yet to be assigned
- *cost*: the current cost of the queue, that is, the cost of the queue formed by the *state* list, we use it to keep track of the state we're in and the final cost of the queue
- *heuristic*: our heuristic function. We define it as an attribute so that we can use the *remaining* list to calculate the estimated cost. As usual, this function will never overestimate the cost to reach the goal state
- *totalcost*: it's *cost* + *heuristic* and it's used to select the next node to expand in the A* algorithm
- *parent*: it's a reference to the current node's predecessor and it's very useful to keep track of the order in which the decisions were made when constructing the list

The second one was the Student class and was used to define more clearly the attributes of each student. For it we defined the following:

- *id*: unique number assigned to each specific student
- *seat*: the seat they were assigned in the bus. This would be useful later when applying the restrictions
- *tr*: boolean value indicating if they are troublesome students or not
- *rm*: boolean value indicating if they are students with reduced mobility or not

With these two structures, we can now define a set of stated formed by lists of students, with all the information needed to apply the algorithm. However, we decided to create some auxiliary functions to help the readability and complexity of the code.

First, the function getStudents was used at the beginning of execution and was very useful to organize the information of all the students given as input. The function would take the initial dictionary of students, extract their individual data, and introduce them as Student structures into a list, so that they became more manageable.

Then, the function children was a key part in our algorithm; given a certain node, it would generate all its possible successors, considering those students that had not yet been assigned to the queue, this way we would avoid creating nodes with repeated students in their lists. Here, we also had to take care of an exception: if the last student in the queue was a reduced mobility one, the next one added could never be another student with reduced mobility, so we handled that by not creating such conflicting nodes (all this is commented in the code).

Finally, we get to the actual algorithm, which is defined in the function astar. This is an iterative function that selects, among the nodes in an open set, the one with the lowest totalcost (this is what the A* algorithm does, using the heuristic cost as one of the parameters for the final calculation). It will expand it (generate its children) and add all these new nodes to the open list. This node is then removed from the open set and included in the closed one as it has already been expanded. In each iteration, it is checked whether we had reached a final state, meaning that the state of the last node is full, and the remaining list is empty. While developing the algorithm we discovered a problem when the algorithm would finish execution and returned a queue with a reduced mobility at the end of it, but we solved it checking that no solution was possible if the last element of state was a reduced mobility student.

2.3.1 Heuristics used

As described, one of the fundamental pieces for our model to work are the heuristic functions we use to estimate the remaining cost to the goal state. So, to test the model and try to optimize it, we created the following

1. *complete relaxation of restrictions*: for this one we treated the remaining students as equals, so that they would all take the same time to board the bus. This is an admissible heuristic as it will never overestimate the cost of boarding everyone in the queue
2. *Dijkstra*: this is a very famous heuristic in which all heuristic costs are 0, this way we assure that it will be admissible. However, it requires a higher computational cost, which is observed in the results for each of the test cases

2.4 Analysis of the results and test cases

One thing to consider when analyzing the results is that it is possible that we get more than one solution with the same optimal cost. This is because some of the students might be interchangeable within the structure of the solution, such as students that are not troublesome nor have reduced mobility, it's change would never impact the total cost of the queue. We experienced this when running the model several times with the same data. Nevertheless, here are the test cases we tried our model against.

2.4.1 Simple case (*easytest.prob*)

For this first case, we left out troublesome students and focused on the model correctly placing the reduced mobility students. We only had four students, with two of them being reduced mobility ones. Our model correctly interpreted the situation and returned the solution in which they would be intercalated (reduced mobility, normal, reduced mobility, normal) and no reduced mobility student would be left out without another student to help them

2.4.2 Full case (*students.prob*)

This time, we included all kinds of combinations for the types and seats of the students, so that we could put our model to test with all possibilities. Out of the 8 students, we included 2 troublesome, 1 reduced mobility and 1 both troublesome and reduced mobility. When analyzing the results, we can clearly see the benefits of using the model in contrast with a brute force approach: 8050 nodes were expanded, which is around 80% less nodes expanded than the 40320

that the brute force algorithm would have taken. On the other hand, the distribution of the queue had the following properties:

- Troublesome students were sent to the back of the bus so that they would not disturb other students
- Reduced mobility students were placed strategically so that no troublesome student was behind them (doubling their cost to get into the bus)
- The student that had reduced mobility and was a troublesome one, was sent the furthest back possible, without having another troublesome student as their helper. This way, it would minimize the impact of both being a troublesome student and having reduced mobility

4 Conclusions

Constraint satisfaction and heuristic search problems are part of our everyday life and, even if we don't notice it, we are constantly solving them. For example, when we're going to meet some friends, we need to set a time and location for the meetup, which are both subject to some restrictions like, it cannot be between 1am and 8am (for example) or the place should be within 10km from our house. This is a clear CS problem. On the other hand, once we have set the time and place, we must get there, for which we will need to check what the fastest route by bus is (heuristic search problem). Lots of examples arise and this confirms the fact that knowing how to model and solve these kinds of problems will eventually lead to good results in both our personal and professional life.

Modeling the proposed problems was not as easy as it might seem from the problem statement because, although this was just an example with a few entries (32 is a low number compared to real life problems), we had to model it in a way we were able to scale it, if needed. In the process we ran into some problems such as difficulties to implement the restrictions without making the problem impossible to solve and a lot of try and error when developing the node structure for the queue (we first started with the students as nodes but then realized that the problem would have been much more complex). Nevertheless, we were able solve all them and thanks to that, we now have a much better understanding of the thought process behind a lot of complex decisions that involve some restrictions or cost. Overall, we found this project very interesting and an enriching experience that will be very valuable in our professional future