Heuristics and Optimization

Course 2022/2023

Constraint Satisfaction and Heuristic Search

PROFESSOR'S NAME: MAURICIO SALERNO

GITHUB LINK:
https://github.com/Rodriksen/HeuristicLab2

APPLIED MATHEMATICS AND COMPUTING
GROUP 121
SARA LOZANO SERROUKH 100452223
RODRIGO DE LERA DE LAS HERAS 100452323

INDEX

1. Introduction

In this document, it will be explained the implementation of a Constraint Satisfaction Problem and a Heuristic search.

For the CSP (part 1), we had to develop a program that solved the assignment problem of sitting students inside a bus.

For the Heuristic search (part 2), we had to develop a program that optimized the time spent by students to enter the bus.

After solving these problems, some tests were considered to check the correct functioning of the developed algorithms. Afterwards, an analysis of the obtained results was carried out.

Finally, some conclusions were derived regarding the written code and the obtained results.

2. Problem modeling

a. Part 1

For this part, we had to solve a Constraint Satisfaction problem. Let's first recall what we need to solve a problem of this type.

- Set of variables.
- Set of domains.
- Set of constraints.

To solve the problem, we used the package "constraint" of the python library.

The proposed problem was assigning bus seats to students with different characteristics. The program will obtain a text file with a specific format where the students and their characteristics will be specified.

The first thing we had to do was opening the given file and saving the information in a proper way. The procedure we followed was writing in a vector elements of a previously defined class. So we have a vector with all the students. The elements of this vector are objects of the class "Student", an object of this class will have information about each student, the stored information is:

- *id*: Student 's ID.
- *year:* Student 's year.
- *trouble:* C if the student is troublesome, X otherwise.
- *mobility:* R for reduced mobility students, X otherwise.
- *sibling:* Student's sibling ID or 0 if the student does not have a sibling.
- *values*: Student 's domain (which will be computed later).

- *label*: Student 's label (which will be used as a tag for the variable).

Then, the elements in the vector will help us when setting the variables of the problem, the variables are the labels in the class “Student”. The format of this label will be a string of the ID of the student, C/X to indicate if the student is troublesome and R/X to indicate if it is a student with reduced mobility. For instance one variable could be: ‘3XR’.

Once we have the vector of students, we have to specify the domain of each student. That is, which seats can each student occupy. The general domain of this problem is {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 25, 26, 27, 28, 29, 30, 31, 32}.

To set the domain for each student, we created the function setDomain() which will establish the domain of each student in the corresponding variable of the Student class. In this function we have two cases, the student has a sibling or not.

- For the case where there are no siblings, we developed the function setNoSibling(), in this function the domains were set in the following way:
  Students without reduced mobility:
  First year student = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
  Second year student = {17, 18, 19, 20, 21, 22, 23, 24, 25, 25, 26, 27, 28, 29, 30, 31, 32}
  Students with reduced mobility:
  First year student = {1, 2, 3, 4, 13, 14, 15, 16}
  Second year student = {17, 18, 19, 20}

- For the case with siblings we developed the function setSibling(), which will be used to restrict the domain of siblings of different years to the front of the bus in the following way: older brother has to be in the seat next to the aisle and the younger in the seat next to the window. In the case where the siblings are from the same year, we set their domain with the setNoSibling() function and they will be placed together with a constraint defined later.
  Siblings of different years without reduced mobility
  First year brother = {1, 4, 5, 8, 9, 12, 13, 16}
  Second year brother = {2, 3, 6, 7, 10, 11, 14, 15}
  Siblings of different years with reduced mobility
  First year brother = {1, 4,12, 13, 16}
  Second year brother = {2, 3, 14, 15}

With the end of the setDomain() function we acquire the domain of the variables in a vector in the values variable of the class Student.

Once we have all this information, we can start to create the problem with python tools. First, the variables need to be added to the problem. To do so, we iterate through our students vector and add the label and the vector values (both of the class Student) of the corresponding student with the function *addVariable(st.label, st.values)*. When iterating, we also created two vectors, one for reduced mobility students and one for troublesome students. This was done to facilitate the implementation of constraints, as it will be seen later. Secondly, the specified constraints in the statement were constructed.

- **One student per seat**, this was done with the *AllDifferentConstraint()* function from python-constraints.
- **Seat next to a reduced mobility student empty**. For this, we iterate through the reduced mobility vector and the students vector and implement the constraint *movSeat(reduced_student.label, student.label)*. We design this function to ensure that no student will be seated next to a reduced mobility student.
- **Troublesome students cannot be seated near a troublesome student nor a student with reduced mobility but if two students are troublesome and siblings they have to be seated together** (this will be controlled with a conditional). To compute this we iterated through the troublesome vector and inside this for loop we iterated through the troublesome vector and the reduced vector (there are two for loops inside the loop) and inside both loops we implemented the constraint *trouble(trouble_student.label, trouble_student2.label)* or *trouble(trouble_student.label, reduced_student.label)*, respectively.
- **Siblings have to be seated together**. For that, we iterate through the vector of students to find those that are siblings and to those variables we apply the constraint *sib_together(sib1.label, sib2.label)*.
- The remaining constraints are satisfied by the previous definition of the domains. Those constraints are that a **seat designated to a reduced mobility student can be occupied by any student** if it is free and **older students need to be at the back but if they have a sibling, they have to be seated together**.

Finally, with all the variables, domains and constraints added to the problem we can find the solutions with the *getSolutions()* function from the python package python-constraint.

To finalize the code, we wrote in a file 5 random solutions of the problem and the total number of possible solutions, if solutions can be found. On the other hand, if there are no solutions for the problem, in the output file it is written that 0 solutions are found.

Let's now formally define the model as a CSP, which is defined as a tuple R = (X, D, C).

- **Variables**: X_i = flag of the student, for example, '3XR'
- **Domain**: D_i = {*possible assigned seats*}, for example, {1, 2, 3, 4, 13, 14, 15, 16}
    - The general domain we are working on is D = {1:32}; however, not all variables have the same domain since it shows the seats at which a specific student is able to be assigned
- **Constraints**: C
    - One student per seat (C_1):

      $C_1 = (s_i \vee s_j) \ \forall$ i, j $\in$ D such that i $\neq$ j; "s" represents the seat of the bus.
    - Seat next to a reduced mobility student empty (C_2 and C_3):

      $C_2 = (r_i \vee s_j) \ \forall$ i, j $\in$ D such that if i is odd, j = i + 1 and if i is even, j = i - 1; "r" represents the seat occupied by a student with reduced mobility and "s" represents the seat that cannot be occupied.
    - Seats close to troublesome students not occupied neither by troublesome nor student with reduced mobility (C_3):

      $C_3 = (t_i \vee t_j) \ \forall$ i, j $\in$ D such that if i $\in$ {1, 5, 9, 13, 17, 21, 25, 29} (this domain corresponds to the seats in the left window) then j $\in$ [i-4, i-3, i+1, i+4, i+5]; if i $\in$ {4, 8, 12, 16, 20, 24, 28, 32} (this domain corresponds to the seats in the right window) then j $\in$ [i-5, i-4, i-1, i+3, i+4]; if i has any other value (this domain corresponds to the seats next to the aisle) then j $\in$ [i-5, i-4, i-3, i-1, i+1, i+3, i+4, i+5]; where t_i are troublesome students and t_j are either troublesome or students with reduced mobility.
    - Siblings seated together (C_4):

      $C_4 = (s_i \vee s_j) \ \forall$ i, j $\in$ D such that if i is an even number, then j = i-1 and if i is an odd number, then j = j+1; s_i corresponds to one sibling and s_j corresponds to another sibling.

b. <u>Part 2</u>

In this part, we had to solve a heuristic search problem. This problem consisted in deciding the order at which the students have to enter the bus to spend the minimum amount of time.

To simplify the algorithm, we defined two classes, the class "Student" and the class "State", this last class will be explained when defining the problem state as it corresponds to the state set. Therefore, let's check what the class "Student" is.

The “Student” class has the following elements:

- *label:* same characters of the input file (for example, ‘3XR’), these characters correspond to the characteristics of each student. The first character corresponds to the ID, the second to know if the student is troublesome (C/X) and the last character to acknowledge if the student has reduced mobility (R/X).
- *seat:* the place in which the student is located in the bus.
- *trouble:* character that shows if the student is troublesome or not. This was included to facilitate later how to establish the costs.
- *reduced:* character that shows if the student has reduced mobility. This was also included to show how to establish the corresponding cost and requirements specified in the statement.
- *flag:* string containing the label and the seat. This will be the parameter that will be added to the bus vector in the class “State”.

For the project, we needed to implement the A* algorithm. For this family of algorithms we need to define the problem space:

- **State set:** each state is an object of the class “State”. In this class, beside having the state defined, we have included some information about the state which will be needed when applying the operators. Each state is composed of two elements:

  bus: vector of the students inside the bus ordered by the sequence they followed to enter the bus (this is useful so that we do not have to compute later the path obtained with A* to reach the optimal solution). Inside this vector we are storing the label and seat of the corresponding student, this was done using the element *flag* of the “Student” class.

  outside: vector of students that have not entered the bus yet. This vector does not have any specific order.

  Then, the extra information of the state is:

  carry: this will be used to know if the last person that entered the bus was a troublesome student.

  heuristic: this parameter specifies the heuristic that is going to be used, this parameter is established as an input by the user. In our program we have defined two different heuristics; therefore, the possible values for this variable are 1, 2 or 3.

  multipliers: list containing the seats of the troublesome students that are in the bus vector. This allows us to check which students are placed in seats with a higher value so that we can duplicate the corresponding cost.

  state_cost: cost of moving from the previous state to the current one.

  g: total cost of the path, including the current state cost.

h: value of the heuristic at the corresponding state

f: sum of the cost function (f) and the heuristic cost function (h).

- **Initial state:** our initial state is an element of the "State" class, where "bus" is an empty list and "outside" is the vector of "Student" objects containing the information about each student on the input file.
- **Goal state:** we do not have an explicit description of the final state, our process of finding the goal is reaching a state that has an empty "outside" list, meaning that every student has entered the bus. Consequently, to know whether the reached state is final or not, we have defined the function *isFinal()* in the class "State" that checks if the *outside* vector is empty.
- **Operators:** we define two operators that are functions of the "State" class, which are "moveDisabled" and "moveNormal". These operators are better explained later with the formal definition of the model.

Now, we can proceed on explaining the code. The first thing we needed to do was to read the input file and store the information of each student in a vector of "Student" objects, that is later used as the "outside" parameter of the initial state. Then we create "open_list", which is a priority queue, append the initial state to "open_list" and enter the main loop of the algorithm (before entering the loop, we initialized the variable "expanded_counter" to 0 to keep track of the number of nodes expanded).

Inside the while loop; first, we select the element of "open_list" with the smallest "f" value to expand it, using the capabilities of priority queues. For expanding the current state we check if it is final with the function explained before. If we reach a final state, then we compute the needed time for reaching the solution and save all the desired results in the corresponding text files. If the reached state is not final, then we have to find the descendants/children of the current state. To find the children, we have to perform to the students in the "outside" list the corresponding operator. The "moveNormal" operator moves one normal or troublesome student from the "outside" list to the "bus" one, and calculates the associated cost of this transition taking into account if the previous student was troublesome (using the "carry" marker) and to double the cost if a troublesome student has entered before and has a lower seat (we make use of the "multipliers" vector and a loop to check if the student's seat is higher than any seat on the "multiply" list). The "moveDisabled" function moves two students at once, one student with reduced mobility and a normal or troublesome student, and has a similar way of working as "moveNormal". Finally, we append every child state obtained in this stage to the "open_list", and keep iterating through inside this while loop until a final solution is reached or until there are no more nodes to expand.

Finally, if the solution is found, two file have to be created, in the file with extension .output we will save the initial and final states and in the file with extension .stat we store the information regarding the total cost, the execution time, the number of nodes expanded and the length of the solution. If there is no solution to the problem, an output file will be generated to notify that no solution was found.

Due to the nature of our states, we could perform some simplifications to the original A* algorithm. These simplifications are based on the fact that for this problem, there is only one way to reach each state, which is appending the last student of the current state to the previous state. Therefore, we did not use a "closed_list" because we do not need to check if we have expanded a state with different costs, since there is only one way of reaching the state. Apart from that, when expanding a node, we do not need to check if it is already in the "open_list" for the same reason.

Then, the formal definition of this problem would be:

- **Set of states**: each state is a tuple $S_i = (B_i , O_i , C_i , M_i , SC_i)$
    - B_i : this variable is a list consisting of the students that are in the bus
    - O_i : it is a list of student objects that are not still in the bus
    - $C_i \in D = \{0,1\}$: the carry is a variable that shows if the last student was troublesome
    - M_i : vector containing seats of previous troublesome students.
    - SC_i : cost from the previous state to the current state
    - **Initial state:** $S_0 = (\{\} , \{every\ student\ in\ the\ input\} , 0 , \{\} , 0)$
    - **Goal state:** $S_0 = (\{optimal\ order\ of\ students\} , \{\} , C_f , M_f , SC_f)$
- **Operators:**
    - *moveDisabled*(x,y)
        - Preconditions:
            - $(x.reduced = \text{“R”}) \wedge (y.reduced \neq \text{“ R”})$
            - $x, y \in O_i$
        - Effects:
            - $O_{i+1} = O_i \setminus \{x, y\}$
            - $B_{i+1} = \{B_i , x.flag, y.flag\}$
            - $SC_{i+1} = k , k \equiv cost\ of\ moving\ to\ the\ new\ state$
            - $y.troublesome = \text{“C”} \Rightarrow (C_{i+1} = 1) \wedge (M_{i+1} = \{M_i , y.seat\})$
            - $x.troublesome = \text{“C”} \Rightarrow (M_{i+1} = \{M_i , x.seat\}$
            - $y.troublesome = \text{“X”} \Rightarrow C_{i+1} = 0$

- *moveNormal*(x)
    - Preconditions:
        - $(x.reduced \neq \text{“R”})$
        - $x \in O_i$
    - Effects:
        - $O_{i+1} = O_i \setminus \{x\}$
        - $B_{i+1} = \{B_i, x.flag\}$
        - $SC_{i+1} = k$, $k \equiv$ *cost of moving to the new state*
        - $x.troublesome = \text{“C”} \Rightarrow (C_{i+1} = 1) \wedge (M_{i+1} = \{M_i , x.seat\})$
        - $x.troublesome = \text{“X”} \Rightarrow C_{i+1} = 0$

3. Analysis of results

a. Part 1

To test the functioning of the developed program, we wrote some tests. For these tests, the number of variables is not considerably big to ensure a solution with a reasonable execution time of the program.

With **students1.txt**, we printed some of the obtained solutions and checked manually that the students were placed in the correct spot.

With **students2.txt**, we check that the seat next to a reduced mobility student is left empty. To do so, we introduced 6 students with reduced mobility and some other students. Checking the output of the problem, we saw that the seat next to the student with reduced mobility was not occupied.

With **student3.txt**, we check that sibling of the same year are seated together in the corresponding section, students of different years (one of them having reduced mobility) are seated in the same section but not together, siblings from different years being both of them troublesome are seated together in the from section of the bus and siblings from the same year, both with reduced mobility have to be seated in the same section but not together.

With **students4.txt**, we check that if the input file has more than 2 students with reduced mobility in year 2, the problem does not have a solution.

With **students5.txt**, we check that if the input file has more than 4 students with reduced mobility in year 1, the problem does not have a solution.

With these two last tests, we can clearly check in a reasonable amount of time that if the number of students bypass the number of seats, then there is no solution for the problem. This can be extended to different situations:

- When the number of students in the first year is greater than 16, the solution of the problem is 0.

- When the number of students in the second year is greater than 16, there is no solution to the problem.
- When the number of students is greater than the capacity of the bus, the problem does not have a solution.

These conclusions were not reached with specific tests due to the large number of nodes to be expanded if we tried to compute the test. However, it was tried and verified that it took a large amount of time.

With all these tests, it is also proved that each student has only one seat assigned and that any student can be placed in a seat designated to a reduced mobility student if they are not occupied by a reduced mobility student.

With this analysis, we were able to check that the time and space complexity of this type of algorithm is incredibly high. When more students were included in the input file, the algorithm performed significantly worse in terms of execution time. Another result that can be derived from the analysis is that with the increasing number of students, more solutions were found.

b. Part 2

To test the functioning of the developed program, we wrote some tests. For these tests, we wanted to check the efficiency of each heuristic regarding the time spent running the program and the number of nodes expanded. To perform these analyses different cases were considered. The nomenclature that is going to be used to explain each test is: stY-X.txt, where Y is the test performed and X corresponds to the heuristic (each test was tried with each of the defined heuristics).

With **st1-X.prob**, the first test performed was in a case where there were no troublesome students nor students with reduced mobility. The goal of this test is to obtain the solutions for the cheaper case. These results will be compared with the other results to check the increase of the cost.

With **st2-X.prob**, we introduced some troublesome students to check the increase of the cost.

With **st3-X.prob**, we considered some students without reduced mobility and others with reduced mobility to check the increase of the cost, as it was done in the previous test.

With **st4-X.prob**, we considered a situation where we have students, troublesome students and students with reduced mobility.

For these four tests, the same number of students was considered to compare the execution of the different heuristics with different situations.

With **st5-X.prob**, we consider the situation with a different number of students and a variate group of students, that is considering also troublesome students and students with reduced mobility. The aim of this test is to check the increase/decrease of the cost with the changing number of students. As less students were considered, we appreciate a lower cost.

With all these tests, we can easily check that no student with reduced mobility will be the last in the final state, as it was restricted. To check this, look in the .output file of the corresponding test, the last position of the final state.
With **st6-X.prob**, we finally checked that if no solution for the problem is found, instead of creating two files, we just generate an output file notifying that there is no solution for such a problem.
With **st7-X.prob,** it was checked the growth in complexity regarding time and space. As we can see, with 10 students, the program with heuristic 1 took around 1 minute to find the optimal solution and expanded around 268000 nodes.

Before obtaining any conclusions, let's compare the heuristics used. Recall that the heuristic function is used to estimate the cost, therefore to find an admissible heuristic, the estimated cost cannot be greater than the real cost.
Heuristic 1: h1 = number of students outside the bus. With this heuristic, we are relaxing the cost of the troublesome students and the students with reduced mobility, we are assuming all students take 1 unit of time to enter the bus.
Heuristic 2: h2 = number of students with reduced mobility outside the bus. With this heuristic, we are relaxing the time spent by students that do not have reduced mobility to 0, so only students with reduced mobility spend one unit of time entering the bus. In this case, the three established costs in the statement are relaxed, simply by setting them to 1 or 0.
Heuristic 3: h3 = 0. In this heuristic, we considered the most relaxed case, where the students enter the bus without spending any time. This is considered an uninformed search.

We ensured that these heuristics are admissible. The heuristics were designed simply by relaxing some of the costs associated with the time spent entering the bus as it was just explained.

As we can see $h3 < h2 < h1$, which implies that h1 is our most informed heuristic and therefore the best of our options. h3 is clearly the most uninformed heuristic as it is as computing the Dijkstra algorithm and h1 is clearly more informed than h2 because, in no case with a solution, there will be more students with reduced mobility outside the bus than students in general.

Performing the first 5 tests with the different heuristics, the following results were derived:

		nodes expanded	total time (s)	total cost
h1	st1-1	46	0,0082	8
	st2-1	7946	0,7212	12
	st3-1	21529	1,4282	10
	st4-1	6919	0,7643	17
	st5-1	23	0,0017	6
h2	st1-2	69281	3,9255	8
	st2-2	15939	1,4792	12
	st3-2	41029	2,3767	10
	st4-2	8069	0,9477	17
	st5-2	27	0,0016	6
h3	st1-3	69281	3,844	8
	st2-3	15939	1,5357	12
	st3-3	41029	2,4163	10
	st4-3	9018	0,9508	17
	st5-3	27	0,0017	6

When computing the table above, the sixth test was not considered as it is the test where no solution can be found. What is more, the seventh test was not considered as it was included in the test battery just to show the exponential growth in memory and time consumption.

As we can see in the table, for each test stY-X, the cost is the same independently of the heuristic use, which is reasonable. What is more, with heuristic 1, we obtained the best results regarding execution time and nodes expanded, as was expected because it is the most informed search. In addition, if we compare the first four tests using the same heuristic, we can check that the cases where there were no students with reduced mobility nor troublesome, the best results were obtained, which shows the correctness of the implementation. We can also compare the results of the tests 4 and 5 with the same heuristics and check that when we have heterogeneous groups of different sizes, the smaller group has the smaller values.

We can also see that in the first test, heuristics 2 and 3 offered the same result. This is caused because the third one is h=0, and since there are no reduced mobility students in the input, our second heuristic is also always equal to 0.

4. Conclusions

For this last section in the report, some conclusions from the work done will be summed up.

Firstly, it is noticeable that when having a great amount of students, the algorithm developed to assign the bus seats is not appropriate (at least when computing it with an ordinary laptop) due to the space and time complexity. However, we can see the usefulness of the program when being used in a real case, as the algorithm provides multiple solutions, if they exist, for a specific problem.

Secondly, the algorithm developed for deciding the order at which students enter the bus has a considerable impact in real life. At the end of the day, it is of best interest to spend less time at a bus stop picking up children. This way, the school bus can get to the school earlier, for instance. What is more, we used the A* algorithm to obtain the solution; therefore, thanks to the properties of this algorithm we are ensuring:

- A solution is found if there is any.
- With an admissible heuristic and a finite number of descendants , we ensure that the solution found is optimal.

In addition, contrary to the first lab assignment, the algorithms developed for the specified problems are not general. Thus, if we tried to reuse these algorithms with different data, multiple changes in the codes will need to be done.