# 3 Heuristic Search

In this section we will present our solution to an heuristic search problem.
It consists on determining the optimum sequence of operations to traverse a bus with a maximum passenger capacity over a weighted graph, and transport passengers from their origins to their destinations.

## 3.1 Data structures

The problem parameters are delivered on a text file, which contains the following items.

- Weighted graph, assumed to be directed.
- A mapping of school IDs to station IDs.
- A mapping of waiting passengers, grouped by common destination, to stations.
- A definition of the bus characteristics, its initial location, and its maximum passenger capacity.

A single state on the state space of this problem is composed by a set of data structures derived from the parsing of the previous items.

- A bus, with its initial and current stations as attributes and a list of embarked passengers.
- A constant reference to the transition graph, to query the available neighbours of the current station, cost of transition and lowest cost value of the optimal path between any pair of stations. This is an immutable structure during the program execution.
- A mutable vector of stations, each defined by an station ID and a vector of passengers waiting to embark.

To identify each state within the search task we use a hash function developed by Paul Hsieh known as the Super Fast Hash.

This algorithm uses the 16 bits shifters present in x86 architectures to achieve a high speed hash, used on a buffer assembled with the bus location, its passengers, all passengers waiting on the stations and the parent ID of the state.

Other structures related to the state are required for the search task, like the buffer storing the cumulative transition cost of the bus, and its expanded form, which is to be stored into the closed list upon state expansion.

The expansion form constructs the minimum information to reconstruct the problem solution.

It is composed by the current node ID, the parent ID, the current station id where the bus is located and two mutually exclusive flags, indicating if this state embarks or disembarks a passenger. It also contains a school destination of an embarking passenger to satisfy requirements on the problem solution presentation.

## 3.2 Operations

On the search task, new states are yielded via a generic function called 'get_successors', which verifies the preconditions of the following operations, executes them to create new successors and finally returns them.

#### 3.2.1 Transit to adjacent station

Preconditions

- Receives a station ID as an argument, which must be adjacent to the current station ID of the bus.

Postconditions

- The cost of the new state has the cost of the parent state incremented by the transition cost to the station ID given as parameter.
- The current station ID for the bus is the station ID given as parameter.

This operation will generate as many successors as adjacent stations to the current one in the transition graph.

#### 3.2.2 Embark passenger

Preconditions

- Number of passengers in the passenger vector of the bus does not exceed its maximum passenger capacity.
- Selected passenger is located on the passenger list of the current station.

Postconditions

- The passenger is removed of the station list and inserted into passenger vector of the bus.
- The cost of the state generated is that of the cumulative cost of the parent incremented by one to avoid null transitions.

This operation can generate as many successors as passengers on the current station.

#### 3.2.3 Disembark passenger

Preconditions

- The passenger given as parameter, or an equivalent one, that is, with same origin and destination, is already embarked.
- The current station ID of the bus corresponds with the destination of the passenger to disembark.

Postconditions

- The passenger is removed from the passenger vector on the bus. This passenger is now out the system.
- The cost of the state generated is that of the cumulative cost of the parent incremented by one to avoid null transitions.

This operation can generate as many successors as passengers on the bus whose destination station is the current one.

### 3.3 Heuristics

The purpose of this section is to present two admissible heuristics to guide the search towards paths that lead to a solution.

An admissible heuristic is one such that its application yields a total cost that will not exceed the optimal value of reaching a final state.

To obtain an optimal result, we need to use admissible heuristics.

For both of the following heuristics we make use of the minimum distance cost among any pair of stations. To obtain this value, we have implemented the Bellman Ford algorithm seen in class into the graph module.

### 3.3.1 Maximum cost to reach a station with passengers

This heuristic will return the maximum distance to recover to a station with passengers.
It is an admissible heuristic because even if the bus needs to recover all the passengers at some point at their stations, it also needs to deliver them and return back the the station of origin.
If there are no passengers in any station, its value will be zero.

$$h_1(n) = \begin{cases} \max cost(current, station) \forall station \text{ if passengers to recover} \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

To execute the implementation with this heuristic, write 'max_distance_passenger' as second argument.

```
./bus-routing <problem> max_distance_station
```

### 3.3.2 Maximum distance to deliver a passenger

This heuristic will return the maximum distance to deliver any passenger of the bus from its current location.
It is an admissible heuristic because even if the bus delivers all the passengers at some point to their destinations, it will also need to deliver them and return back the the station of origin.

$$h_2(n) = \begin{cases} \max cost(current, passenger.destination) \forall passenger \in \text{ bus} \\ 0 \text{ if no passengers on the bus} \end{cases} \tag{2}$$

To execute the implementation with this heuristic, write 'max_distance_station' as second argument:

```
./bus-routing <problem> max_distance_passenger
```

### 3.3.3 Heuristic combination

The previous heuristics can both be applied at the same time using:

```
./bus-routing <problem> all
```

On this mode, the maximum value of both heuristics at each state will dominate, which implies that the combination of both heuristics will still be admissible.

## 3.4 Analysis of results

The implementation is a C++ program built using CMake. In the delivery there will be a README file with instructions for compilation and execution.
For the analysis of results we assembled different cases with varying levels of complexity.
We executed the tests on a headless Linux machine with a Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz and 32 GB of DDR3 RAM. Once the problem execution exceeded the available memory, the process was killed.

### 3.4.1 One origin and one destination

On this test case there is only one station with students. The bus has to recover the students, take them to the school and come back to the initial station.

```
   P1 P2 P3 P4 P5 P6 P7 P8 P9
P1 -- 12 -- -- -- -- -- -- 9
P2 12 -- 13 -- 10 -- -- -- --
P3 -- 13 -- -- -- -- -- -- 6
P4 -- -- 13 -- 8  -- -- 8  --
P5 -- 10 -- 8  -- 14 -- 5  --
P6 -- -- -- -- 14 -- 7  -- --
P7 -- -- -- -- -- 7  -- 6  --
P8 -- -- -- 9  5  -- 6  -- --
P9 9  -- 6  -- -- -- -- -- --
C1: P6;
P2: 1 C1, 1 C1;
B: P1 5
```

- No heuristics.

  ```
  Overall time: 0.028362 seconds
  Overall cost: 76
  # Stops: 7
  # Expansions: 9733
  ```

- Using maximum distance to deliver a passenger.

  ```
  max_distance_passenger.statistics
  Overall time: 0.009303 seconds
  Overall cost: 76
  # Stops: 7
  # Expansions: 1964
  ```

- Using maximum cost to reach a station with passengers.

  ```
  max_distance_station.statistics
  Overall time: 0.018603 seconds
  Overall cost: 76
  # Stops: 7
  # Expansions: 5394
  ```

- Using all heuristics.

  ```
  Overall time: 0.009913 seconds
  Overall cost: 76
  # Stops: 7
  # Expansions: 1964
  ```

Solution

```
P1 ->P2 (S: 1 C1, 1 C1) ->P5 ->P6 (B: 1 C1, 1 C1) ->P5 ->P2 ->P1
```

### 3.4.2 Two destinations and four origins.

On this test case we have two schools and several stations with students to embark and transport to those schools.

```
   P1 P2 P3 P4 P5 P6 P7 P8 P9
P1 -- 12 -- -- -- -- -- -- 9
P2 12 -- 13 -- 10 -- -- -- --
P3 -- 13 -- -- -- -- -- -- 6
P4 -- -- 13 -- 8  -- -- 8  --
P5 -- 10 -- 8  -- 14 -- 5  --
P6 -- -- -- -- 14 -- 7  -- --
P7 -- -- -- -- -- 7  -- 6  --
P8 -- -- -- 9  5  -- 6  -- --
P9 9  -- 6  -- -- -- -- -- --
C1: P6; C2: P3;
P2: 2 C2; P4: 1 C2; P5: 1 C2; P7: 3 C2
B: P1 5
```

- No heuristics.

  ```
  Not enough memory.
  ```

- Using maximum distance to deliver a passenger.

  ```
  Overall time: 28.679071 seconds
  Overall cost: 104
  # Stops: 11
  # Expansions: 4357332
  ```

- Using maximum cost to reach a station with passengers.

  ```
  Overall time: 16.711261 seconds
  Overall cost: 104
  # Stops: 11
  # Expansions: 1986691
  ```

- Using all heuristics.

  ```
  Overall time: 4.589207 seconds
  Overall cost: 104
  # Stops: 11
  # Expansions: 432056
  ```

  Solution

```
P1 ->P2 (S: 1 C2, 1 C2) ->P5 ->P8 ->P7 (S: 1 C1, 1 C1, 1 C2) ->P6
(B: 1 C1, 1 C1)->P5 (S: 1 C2) ->P4 (S: 1 C2) ->P3
(B: 1 C2, 1 C2, 1 C2, 1 C2, 1 C2) ->P9 ->P1
```

### 3.4.3 Two origins and one destination

On this test case there are two stations with passengers waiting, and one destination or school.

```
   P1 P2 P3 P4 P5 P6 P7 P8 P9
P1 -- 12 -- -- -- -- -- -- 9
P2 12 -- 13 -- 10 -- -- -- --
```

```
P3 -- 13 -- -- -- -- -- -- 6
P4 -- -- 13 -- 8  -- -- 8  --
P5 -- 10 -- 8  -- 14 -- 5  --
P6 -- -- -- -- 14 -- 7  -- --
P7 -- -- -- -- -- 7  -- 6  --
P8 -- -- -- 9  5  -- 6  -- --
P9 9  -- 6  -- -- -- -- -- --
C1: P6;
P2: 2 C1; P8: 3 C1;
B: P1 5
```

- No heuristics.

  ```
  Overall time: 2.481303 seconds
  Overall cost: 86
  # Stops: 9
  # Expansions: 535446
  ```

- Using maximum distance to deliver a passenger.

  ```
  Overall time: 1.373996 seconds
  Overall cost: 86
  # Stops: 9
  # Expansions: 230389
  ```

- Using maximum cost to reach a station with passengers.

  ```
  Overall time: 1.137985 seconds
  Overall cost: 86
  # Stops: 9
  # Expansions: 191669
  ```

- Using all heuristics.

  ```
  Overall time: 0.586383 seconds
  Overall cost: 86
  # Stops: 9
  # Expansions: 93610
  ```

  Solution

```
P1 ->P2 (S: 1 C1, 1 C1) ->P5 ->P8 (S: 1 C1, 1 C1, 1 C1) ->P7
->P6 (B: 1 C1, 1 C1, 1 C1, 1 C1, 1 C1) ->P5 ->P2 ->P1
```

### 3.4.4 One destination, three origins on multiple trips

On this test case there is only one destination but three stations with more passengers than the bus can handle on a single trip, so several trips are required to complete this problem.

```
   P1 P2 P3 P4 P5
P1 -- 12 -- 10 --
P2 12 -- 13 -- 10
P3 -- 13 -- 23 --
```

```
P4 5  -- 13 -- 8
P5 -- 10 -- 8  --
C1: P4
P2: 11 C1; P5: 1 C1; P3: 3 C1
B: P1 5
```

- No heuristics.

  ```
  Not enough memory
  ```

- Using maximum distance to deliver a passenger.

  ```
  Not enough memory
  ```

- Using maximum cost to reach a station with passengers.

  ```
  Not enough memory
  ```

- Using all heuristics.

  ```
  Not enough memory
  ```

For this case, the number of expansions required to solve this problem is so large that it fills all the 32 GB of RAM memory and the swap memory of the system, so it is killed by the operating system.
A possible explanation for this is that the operations embark/disembark dominate the generation of new states, generating states very similar in cost and degenerating the best-first search into breath-first search. Possible solutions for future work would be a different state representation or generating another heuristic specific for this case. However, given how peckish for memory A* is, a better solution might be to implement a deepening algorithm like IDA* if we cannot spare resources into more RAM memory.

#### 3.4.5 3 destinations and 4 origins

On this test case there are several schools or destinations and multiple stations with different students.

```
   P1 P2 P3 P4 P5 P6 P7 P8 P9
P1 -- 12 -- -- -- -- -- -- 9
P2 12 -- 13 -- 10 -- -- -- --
P3 -- 13 -- -- -- -- -- -- 6
P4 -- -- 13 -- 8  -- -- 8  --
P5 -- 10 -- 8  -- 14 -- 5  --
P6 -- -- -- -- 14 -- 7  -- --
P7 -- -- -- -- -- 7  -- 6  --
P8 -- -- -- 9  5  -- 6  -- --
P9 9  -- 6  -- -- -- -- -- --
C1: P6; C2: P3; C3: P8
P2: 1 C2, 1 C3; P4: 1 C3; P5: 1 C2; P7: 2 C1, 1 C2
B: P1 5
```

- No heuristics.

  ```
  Not enough memory
  ```

- Using maximum distance to deliver a passenger.

  ```
  Not enough memory
  ```

- Using maximum cost to reach a station with passengers.

  ```
  Not enough memory
  ```

- Using all heuristics.

  ```
  Overall time: 206.095004 seconds
  Overall cost: 115
  # Stops: 12
  # Expansions: 17505637
  ```

For this case, included in the statement for this problem, we only had enough memory to solve it using all the available heuristics.
Solution

```
P1 ->P2 (S: 1 C2, 1 C3) ->P5 ->P4 (S: 1 C3) ->P8 (B: 1 C3, 1 C3)
->P7 (S: 1 C1, 1 C1, 1 C2) ->P6 (B: 1 C1, 1 C1) ->P5 (S: 1 C2) ->P4
->P3 (B: 1 C2, 1 C2, 1 C2) ->P9 ->P1
```

## 4 Concluding remarks

To sum up, the difficult in the constraint programming problem was to create the CSP problem, with the corresponding variables, domain and constraints.
Once you have them create the program in python and develop it was easier. But the key was create a good set of variables with their corresponding domain. On the other hand, coding it was easier.
We needed to be careful about how to develop the constraints, because we can be really helpful by the pre-computed constraints, as AllDifferentConsraint().
To finish with the first part, is amazing how many solutions can be generated from a single problem, we had problem about be out of memory at first!, and the computation of all possible solutions take us more or less, thirty minutes. It is incredible how a ”simple” problem can be so exponentially complex...
We can't imagine how real problems with so much variables, constraints and more factors to take into account can affect the complexity, maybe in the real world we have to settle by finding a solution in a reasonable time.
We have similar thoughts for the second part, whose development was a truly humbling experience due to the magnitude of the problem and its apparent simplicity.
We could not provide enough memory to handle all the cases, which might be due to the fact that we used A* instead of an iterative deepening approach (IDA*). A good personal project we might do in the future is to rewrite this project in that way to satisfy our curiosity.
The world really needs computer scientists with the skill to handle these kind of problems. We will keep working to get better at these tasks.