Artificial Intelligence and Decision Systems (IASD)

In orbit assembly of large structures

Cristina Melício (78947) Diogo Carvaho (78456)
Instituto Superior Técnico

I. Introduction

In this project we need to schedule the launches of components for the assembly in orbit of a large structure. Given a set of components to be launched, a launch time line and costs, and a construction plan the goal is to determine the assignment of components to launches, such as the total cost is minimized. It is known that the payload of a single launch vehicle is limited, and no components can stay unconnected in space (unless it is the first launch and only one component is launched).

II. Implementation

To solve the problem we define first classes only to store the information from the *.txt files:

- class `Launch` which stores for each launch (l_i), the date of launch, maximum payload (p_i), fixed (f_i) and variable cost (u_i)
- class `Component` which stores the name of the component (v_i) to be sent, weight (w_i) and a list of adjacent components

Now we need to define the **state space**. For us the **state** of a given node is the list of components already in orbit and the number of launches used. An **action** is defined as sending in launch l_i a possible set of components S_i, taking into account all the constraints (maximum payload, components need to be connected, etc...). This choice implies that the depth of the node is equal to the number of launch decisions already made (which includes empty launches). When nothing is sent in a given launch even tough the number of components in orbit remains the same, using this definition for state, the node's and parent's state are not equal because depth has changed. Since we can obtain the same list of components in orbit, using the same amount of launches (including empty ones) but with different actions, our space state is a **graph** and not a tree. The maximum length of any path in our state space (m) will be equal to the number of launches available (n_l). The initial state of our problem is an empty list of components in orbit and 0 launches used. The path cost is the sum of the costs of all actions made to go from the root node until node n. Each decision D_j has a cost c_j given by:

$$c_j = \begin{cases} 0 & \text{if } S_i = \{\} \\ f_i + u_i \sum_{v_k \in S_i} w_i & \text{otherwise} \end{cases} \quad (1)$$

The goal state is any state with all components already in orbit. The optimal solution will be the one that minimizes the path cost $g(n)$ from the initial state to a goal state. With this state space description we can finally implement the following classes:

- class `Node` which stores the state, depth, parent node, payload sent, path cost and the heuristic associated (if informed)
- class `Problem` which has access to 2 dictionaries. One containis objects of type `Launch` (ordered by date, the key is the position in this new order) and the other contains objects type `Component` (key is the component ID). We also define the `initial_state`, the `Successor(n)` and `GoalTest(n)` functions, and calculate the `path_cost`.

Final remarks regarding the successor function. Before expanding any node we verify if it is still possible to send in the remaining launches all the weight missing, considering we can divide components in pieces. If this condition is not fulfilled we do not generate the successors being it uninformed or informed search. This helps improving the trimming of the tree. This could also have been implemented in the form of an heuristic that would assign the value to infinity (or to the maximum possible number allowed).

III. Search Algorithm

Two search algorithms were implemented, one informed and another uninformed. Their implementation is completely separated from the domain dependent part covered in the last section. For the uninformed search we use the **Uniformed Cost Search** algorithm because only this one guarantees that we find the optimal solution. The other methods like Bread-First, Depth-First or variations don't assure optimality because they don't take into account that actions might have different costs and the path cost is not a non-decreasing function of depth.
In the informed search we choose the **A* Search** because it is optimal and complete if the heuristic function is consistent in the graph search case. Greedy Best-First is not optimal because it only considers the heuristic cost so it was easy to discard. We could have implemented Recursive Best-First Search or Simplified Memory Bounded A*, but the first suffers from excessive node regeneration and lack of memory space didn't seemed so important.

Both of them make use of data structures to save explored and frontier nodes. The main difference is the way this methods sort the frontier. The UCS sorts via the lowest path cost, g(n). The A* method tries to minimize the path cost plus the heuristic , h(n), for each node. The frontier is implemented as a **Priority Queue** in class `HeapQueue`. This class was implemented to allow the discard of redundant nodes keeping the one with smaller evaluation function. Note that two nodes are redundant if they have the same components on space and same number of launches, in this case same depth. As we have a graph search we also implement a close list which stores all explored nodes. All newly generated repeated nodes that match previously generated nodes can be discarded instead of being added to the frontier

IV. Heuristics

We developed several heuristics to compare by determining the real cost functions of relaxed versions of the problem.

Heuristic 1 - We choose the cheapest values of f_i and u_i (not yet used) and consider a launch with this prices and no weight limit. The value h_1 is then:

$$h_1(n) = \min(f_i) + \min(u_i) \cdot W \quad , \quad i = d, ..., n_l \tag{2}$$

To prove consistency the following condition needs to be full filled:

$$\begin{aligned} h_1(n) &\leq c(n,n') + h_1(n') \\ \min(f_j) + \min(u_j) \cdot W &\leq f_{n,n'} + u_{n,n'} \cdot W_{n,n'} + \min(f_k) \\ &\quad + \min(u_k) \cdot (W - W_{n,n'}) \end{aligned}$$

And this condition is always true.

Heuristic 2 - First we define the cost per kg ρ_i of each launch i as :

$$\rho_i = \frac{f_i + u_i \times p_i}{p_i} \qquad i = 1, ..., n_l \tag{3}$$

It is clear because of (3) that ρ_i never overestimates the cost of the launch i for a payload smaller than its maximum payload. Also more generally:

$$\rho_i \cdot W \leq f_j + u_j \cdot W \text{ , if } \rho_i \leq \rho_j \wedge W \leq p_i \tag{4}$$

We then define h_2 as the cost of filling all the launches in increasing order of ρ_i according to the algorithm below.

```
weight_missing = sum(weight_comp_not_sent)
loop on launches ordered by rho_i:
|    if payload_i < weight_missing:
|    |    h2 += rho_i*payload_i
|    |    weight_missing _= payload_i
|    else:
|    |    h2 += rho_i*payload_missing
|    |    weight_missing = 0
|    |    break
```

Once again one must prove the heuristic is consistent. This time it is more difficult to make the calculations. First we need to understand that $h_2(n)$ and $h_2(n')$ are piece wise functions made of lines with growing slope ρ_i. We can also define a lower boundary for the cost to move from n to n' based in the parameter ρ

$$\begin{aligned} c(n,n') = &\ f_{n,n'} + u_{n,n'} \cdot W_{n,n'} \\ &\leq \rho_{n,n'} \cdot W_{n,n'} \end{aligned}$$

For any conditions of figure 1 we can then visualize that the value of $h_2(n)$ is always smaller than the sum of this lower limit of $c(n,n')$ plus $h_2(n,n')$, for a given set of ρ_i. We couldn't find a better way to prove consistency.

One must notice that none of the heuristics above dominates the other. Imagine the simplest case where only one component is not yet in space and one launch is left. In this situation $h_1(n) \geq h_2(n)$ because of (4). On the other hand imagine a case where two components (v_1,v_2) with the same weight W are left, two launches (1,2) with the same maximum payload $p = W$ and same fixed cost (f) are available. In this case we obtain :

$$\begin{aligned} h_1(n) &= f + \min(u_i) \cdot 2W \quad , \quad i = 1, 2 \\ h_2(n) &= 2 \cdot f + (u_1 + u_2) \cdot W \end{aligned}$$

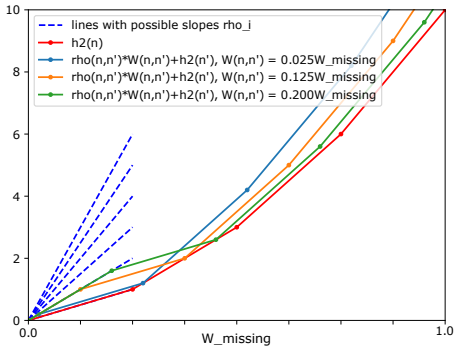

Figure 1: Comparsion between $h_2(n)$ for a given set of possible ρ_i and sum of cost function from node n to n' and $h_2(n')$ in function of the weight missing from orbit ($W_{missing}$) and the weight sent in this launch considering it uses the second lowest value of ρ_i

Then there exist cases where $h_2(n) > h_1(n)$. Since both functions are consistent we can use as heuristic the maximum of $h_1(n)$ and $h_2(n)$, which will also be consistent . This should reduce the effective branching factor while still achieving the optimal solution.

V. Results and Conclusions

In table I we present the results for the files trivial2.txt, simple1.txt and mir.txt and for the methods implemented explained before.

As can be seen in table I the space complexity, ie. the number of nodes generated, decreased when we started using the informed search. Of course for easier problems this doesn't improve a lot the performance, since the maximum depth is small. As we go to more difficult problems we see that h_2 performs better than h_1. This happens because h_1 only takes into account one of the fixed costs of the launch, and for a lot of components normally more launches are needed. We can also notice that the effective branching factor decreases with the use of the informed search. Finally the function max(h_1,h_2) performances better than h_1 and h_2 because it is the dominant one, ie. it is always greater or equal than the others. And as h_1 and h_2 are consistent heuristics than the maximum of them is also consistent.

Problem	**Method**	**Space Comp.** (# Nodes)	**Time/s**	**b***
trivial2	UCS	30	0.007	3.141
	h_1	14	0.006	2.466
	h_2	14	0.006	2.466
	max(h_1,h_2)	14	0.006	2.466
simple1	UCS	652	0.058	3.655
	h_1	524	0.043	3.499
	h_2	244	0.021	3.005
	max(h_1,h_2)	238	0.023	2.990
mir	UCS	25693	7.733	2.760
	h_1	13554	3.774	2.589
	h_2	5561	2.160	2.368
	max(h_1,h_2)	4053	1.317	2.291

Table I: Search methods' performances for different problems, using a quad-core 1.30 GHz base frequency processor

We found that there are some randomization in the results obtain when there are more then one optimal state. This fact is associated with the use of dictionaries in the search process, because when they are created they don't have any order.

REFERENCES

[1] Russel S. and Norvig Peter *Artificial Intelligence A Modern Approach*, 3rd edition.