Christopher Edgley Nº 75258 Inês Lourenço Nº 75637

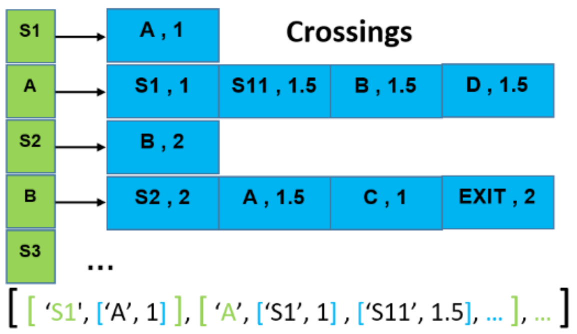
Project Report

The program code is divided in 4 files: A domain-dependent file, called **domain.py**, which handles all problem specific data structures and functions; Two files with implementations of search algorithms adapted from the General Search Algorithm, **solver_UniformCost.py** and **solver_AStar.py** (A*), both generic for any node-tree search problem; And a fourth file called **utilities.py**, containing useful functions which isolating the domain-dependent layer from the domain-independent layer.

(a) Describe how you represent the problem state, the operator(s), the initial state, the goal state, and the path cost

To solve the problem at hand a node-tree is generated, where each node contains the necessary information about the problem at a given moment, i.e., the state space. Nodes are implemented in the form of a class structure, with the following attributes: **stSpace,** the state space represented by the node, **parent** is the index in **nodeInsts** of the parent node, **opText** stores the operation text in the format prescribed by the assignment for output, including the associated cost (in the fourth position of this list), **totCost** represents the total accumulated cost, i.e. the cost function g(n) for a given node, and **heuCost** stores the heuristic function h(n) for the given state space.

The first element of the **state space** of a Node consists of the state of the CTS, with CTS[0] representing which cask is loaded, if any (‘Empty’ if unloaded or [*cask_name, cask_length, cask_weight*] if loaded with cask), and CTS[1] represents the CTS's location in the HCB. The remaining elements of the state space represent the state of each stack in the HCB, that is, whether it is empty or has casks, with the latter including the corresponding lengths and weights of each cask, in LIFO order, as described in the assignment. This is shown in the *Crossings* figure.

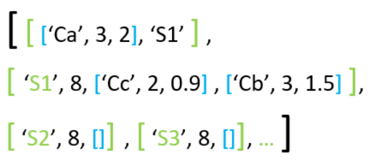


[[‘S1', [‘A’, 1]], [‘A’, [‘S1’, 1] , [‘S11’, 1.5], ...], ...]

The other information that is needed to build the node-tree is the allowed operations in each state. These operations can either be Load/Unload, or Move. The loading/unloading operations are realised by checking the state of the CTS and the state of each stack (function **loadUnload**), and the move operation by analysing the locations adjacent to the CTS's present location, these can be stacks, crossings or the exit, (function **moveCTS**).

In order to implement this, these functions require not only the state space list as represented in figure *State_Space,* but also a list of *Crossings* (and stacks and exit), where each place contains a list of all the places adjacent to it, with the respective path length, i.e. cost.

State_Space



[[[‘Ca’, 3, 2], ‘S1’] ,
[‘S1’, 8, [‘Cc’, 2, 0.9] , [‘Cb’, 3, 1.5]],
[‘S2’, 8, []] , [‘S3’, 8, []], ...]

Following this reasoning the initial and goal states are defined by this format, in which the first position of the state space is defined as [‘Empty’, ‘EXIT’] for the initial state, and the goal is reached when the first position of the state space of a node is [['*goalCask*', *length, weight*], '*EXIT*'].

(b) Identify and justify the search algorithm implemented

The search algorithm implemented for the uninformed search was the **Uniform Cost Search**. The first algorithm implemented was the BFS (Breadth-First Search), but it was abandoned due to its non-optimality. Besides being less memory and time consuming, the new algorithm is optimal and complete because it always chooses the child node with the smallest total cost (f(n) = g(n)) where all the steps in the algorithm have a positive cost. When a certain node is expanded, all his children are analysed and if they have already been explored (found in the closed list), then they are ignored as the cost is necessarily greater. If the node had already been found but not expanded (therefore in the open list) the algorithm keeps the node with smallest total cost.

The goal of the informed search is to help the program to more likely choose an appropriate path in the tree, in order to more easily reach the optimal goal. Therefore the **A*** (A star) method was chosen, which works in exactly the same way as the uniform cost, with the addition of a heuristic function. The evaluation function becomes f(n) = g(n) + h(n). This means that instead of only considering the costs of each operation, "rewards" and "punishments" are given to the method for some of the actions of the CTS, according to what is predicted to be a better or worst action to perform in a certain situation, with the goal in mind. The algorithm is then identical to the uniform cost search, except in regards to the the evaluation function f(n). And in the same circumstances (strictly positive action costs), choosing h as an admissible function, A* is also optimal and complete.

(c) Describe and justify the heuristic functions developed

In order to fulfill the goal of the informed search algorithm, we want to find the best possible heuristic function, already knowing that it always has to be admissible and consistent. Having a good heuristics means that we can "influence" the program to make decisions that we believe are good. If the heuristic function is 0, then the algorithm becomes the uniform cost. On the other hand, a perfect heuristic is the one that always considers the shortest path (in terms of cost) and never explores other non-optimal goals. The first requirement is that h(n) has to be admissible, that is, it can never overestimate the true cost to goal. Bearing this in mind, it was clear that the heuristic had to be able to calculate at least the shortest past between the exit and the stack where the goal casks is, and back. So this was the first heuristic that was tried (let's call it *h1*), and for that the Dijkstra algorithm was used. This algorithm is called for every new node of the graph, but if the node already existed then the D ijkstra has already been calculated. As the goal was to find the best heuristic possible, it was found that by adding some more costs to this function it gets "stronger" and therefore has to explore less nodes. The final heuristic (*h2*) was therefore to add the costs of loading and unloading the necessary casks. In this case, these casks are the ones that are in the *goalStack* in front of the *goalCask*, that always need to be loaded and unloaded to some other stack, as well as the cost of loading the *goalCask*.

Finally, the best implemented heuristic function was: The sum of the shortest path between the current node and the goal stack (only if it isn't loaded with the *goalCask* yet), plus the shortest path between the goal stack and 'EXIT' – multiplied by (*goalCaskWeight+1)*, plus the cost of loading and unloading all the casks in the *goalStack* ahead of the *goalCask*, plus the weight of loading the *goalCask.*

When implementing this, caution is taken in order to assure that it is still admissible. It is since the costs calculated by the heuristic are ones that will always be in the path, that is, it is impossible to find a solution that don't include at least these costs.

The function is not perfect though, as it doesn't take in consideration, for example, the calculation of the shortest path from one stack to the others, in the situation where the CTS loaded a cask in front of the *goalCask* and needed to unload it in some other stack.

(d) Compare quantitatively and comment the performance of at least two different heuristic functions implemented

Using the approach described in the previous question it was possible to notice clear improvements between these two heuristic functions. The first one is *h1*, that only calculated the shortest path between the initial place and the *goalStack*, and then from the *goalStack* to the goal state. The implemented heuristic, as well as considering the cost of moving, also multiplied the *goalCaskWeight* in the return path towards the 'EXIT' and additionally the cost of loading and unloading casks which "buried" the goal cask. This last function proved to be very efficient for solving the problem, as can be seen in Table 1, which shows the comparison between the Uniform Cost algorithm and the A* with these 2 different heuristic functions. The greatest difference between the last two can be justified by the explanation in the answer to the previous question, regarding the "strength" of each one of them, that is, how far away they are from h(n)=0, without overestimation.

g-s10d4-L2.dat	**Time spent to find each of the Casks, in seconds**		
	Uniform Cost	**H1: Dijkstra**	**H2: Dijkstra + caskWeights**
C3	0.2	0.49	0.2
C2	13	5	0.35
C1	15	99	3
C0	N/A	2723	51
(Expanded Nodes)	N/A	(29920)	(4083)

Table 1