Mini-Project I

Artificial Intelligence and Decision Systems (IASD)
Prof. Rodrigo Ventura

ITER Logistics

Goncalo Afonso 75924, Rui Silva 75855
Date: 05/11/16

Problem Statement

In order to store the information cointaned in .dat files, we created classes, each one respecting to stacks, crossings and casks. The stacks' class contained their length, weight, an ordered list with the casks stored on it and a function to calculate, based on the casks stored, the remaining space available inside of it. Using the information stored in the crossings' class, a dictionary was created, corresponding each location of the map to its adjacent and the correponding distances.

The problem state was given to each node with the form of a list with 3 main elements: the current location; the ID of the cask that is currently loaded (0, if none); and the current disposition of casks inside the stacks.

We also defined a problem's class where we defined the actions/operations the agent can perform. They are 'Move', 'Load' and 'Unload'. 'Move' allows the agent to step foot in adjacent locations, changing the location in the state of a child node. 'Load' and 'Unload' occour when the agent is at a stack and wants to perform that action. This changes the ID if the cask in the state of a child node.

The cost of these operations was given as in the assignemnt and the path cost defined as the cumulative sum of the cost of each operation until the desired node.

So, knowing that the agent always departs from 'EXIT' unloaded and returns having the pretended cask loaded, the initial state took the form: [$'EXIT'$,0,$casks'disposition$] (the last parameter depends on the data file) and the goal state, as we don't care what will be the cask's disposition inside the stacks, is [$'EXIT'$,$pretendedcask$].

Both final and goal states are included in the Problem Class, which the objective is to represent the stuff concerned specificly about this storing problem.

Search Strategies

As asked, we implemented one uninformed and one informed search strategy to achieve the proposed goal, but still general to solve this kind of problems.

The uninformed search strategy choosen was the Uniform Cost Search (UCS). The reason of our choice was that to solve our problem we needed to pay special atention to path cost, otherwise we wouldn't get to the desired optimal solution. The other methods like Bread-First, Depth-Fisrt or simillar methods didn't assured optimality to this problem because they didn't take into account the cost per operation and the path cost is not a non-decreasing function of depth.

The informed search strategy was A*. Greedy Best-First is not optimal 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.

This two search algorithms that we implemented have the same root of implementation(Best First Search). Both of them make use of data structures to save explored and frontier nodes. The main diference 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 for each node, $h(h)$.

Heuristics

We implemented two heuristic functions. They are explained in the subsections above. In the next section we also compare them and infer some conclusions about which one is better for different situations.

Heuristics 1

The first one is based on dictionary previously calculated using the Graph class. This dictionary contains the minimum distances between all the locations to the stack where our goal cask is. Using the UCS method, the function *dist_to_pre_stack* sees if the dictionary already has an entry for the distance between each location and the stack where our goal cask is. If it doesn't have, runs uniform cost search with that location being on the initial state. Now, giving the optimal path between the two, we store all the distances between the locations in optimal path and the pretended stack in the same dictionary.

It's also important to say that in the Problem class we evaluated the minimum edge of the map and the minimum wiegth of a cask, storing them in the variables *min_edge* and *min_wei* respectively.

The algorithm evaluates three situations and is described below. In order to guarantee consistency, the operations described in the scheeme "go to other stack" and "get to the stack where our goal is" are calculated by just considering two times the minimum egde - $2 \cdot minimumedge$ - a value that will never exceed the distance between two stacks, since we know that a stack only connects to a crossing. The operation of load/unload a blocking cask will be given by $1+min\_wei$. When our pretended cask is loaded, we know that the cost to get to the goal state is ideally the distance from current location to 'EXIT' (value that we can obtain through operations from the above dictionary) times ($1+goal\_cask\_weight$).
Giving this, we never overestimate the cost between a node and another, guaranteeing consistency, thus admissibility of the heuristic. So the A* search will always provide us an optimal solution.

ALGORITHM:
getbackfromstack ← distance
from current location to EXIT loaded with goal cask
if *some cask is loaded* **then**
  if *cask loaded is our goal cask* **then**
    h ← getbackfromstack
  else
    h ← unload the current cask + go to the stack where our goal is + (load the blocking cask + go to other stack + unload it + go to the stack where our goal is)* number of blocking casks + load goal cask + getbackfromstack
  end
else
  h ← go to our goal's stack + (load the blocking cask + go to other stack + unload it + go to the stack where our goal is)* number of blocking casks + load goal cask + getbackfromstack
end

Heuristics 2

We thought about improving heuristic 1 by giving a better guidance when getting rid of the blocking casks. We noticed that for many maps, the best way to do this was guiding them to the nearest stack available (space non ocupated > cask length). This method is based on first fit algorithm. This algorithm doesn't always give an optimal solution, making this heuristic non-admissible. However, most of the time it will improve our results, saving explored nodes, leading to less time spent doing the calculations.

		s1ex - Cb	s2 - Cc	s3 - C1	L1 - C6	L2 - C0
UCS	Explored	81	175	166	1342	80362
	time of search	0,0065	0,0462	0,0676	1,9408	3395
Heuristic 1	Explored	29	38	34	57	4444
	time in pre calculations	0,0045	0,0055	2,0205	0,3674	0,2842
	time in heuristics	0,0050	0,0080	0,0150	0,0190	14,5954
	time of search	0,0100	0,0145	2,0360	0,3868	14,8800
Heuristic 2	Explored	25	NOT - OPTIMAL SOLUTION	25	57	78
	time in pre calculations	0,0215		19,2306	7,2393	2,8355
	time in heuristics	0,0020		0,0105	0,0185	0,0365
	time of search	0,0235		19,2411	7,2578	2,8720

Figure 1: search methods

In order to do this, we did the same dictionary calculation as explained above, but this time to every stack instead of just to the pretended stack.

In this case instead of *min_edge* we use the real distance from the stack where our goal cask is to the nearest stack where the blocking cask can fit. We also use the real value of weight for each cask instead of an aproximation by default.

As we said earlier, this introduces some problems in some maps although it guides better in other maps. This problems appear when we have to make space to put any of the blocking casks because they don't fit in any available stack. There may be other cases that simply is better to change first some cask in order to put the blocking casks, even if there is space available.

ALGORITHM:
getbackfromstack ← distance
from current location to EXIT loaded with goal cask
if *some cask is loaded* **then**
  if *cask loaded is our goal cask* **then**
    h ← getbackfromstack
  else
    for *each blocking cask* **do**
      get distance to closer
      stack available h1+=load the blocking cask
      + go to the nearest stack available + unload
      it + go to the stack where our goal is
    end
    h ← go to nearest stack available + unload the
    current cask + go to the stack where our goal
    is + h1 + load goal cask + getbackfromstack
  end
else
  for *each blocking cask* **do**
    get distance to closer stack available h1+=load the
    blocking cask + go to the nearest stack available
    + unload it + go to the stack where our goal is
  end
  h ← go to the stack where our
  goal is + h1 + load goal cask + getbackfromstack
end

Comparison between Heuristics and Conclusions

We gathered some relevant informtion to compare the performance of each heuristic (and UCS) in table 1.

In the cases we have blocking casks, heuristics 2 performs better. That stems from the fact that this heuristic provides a better guidance for the casks blocking the pretended cask that we have to move, as we can see in s1ex (Cb), s3 (C1) and L2(C0). This only occours for the blocked casks that need to be placed in the nearest stock available. In the case of s2 (Cc), heuristic 2 fails to deliver optimal results, because in order to unblock the pretended cask it has to change other cask in other stock and our heuristic function doesn't provides a good aproximation to that case. In L1 (C6), the number of nodes explored are the same. This happens because the two heuristic functions procced the same way for the pretended cask in the first place to get out. This case is also good to notice that when we have the same number of nodes explored, the difference in computation time of these heuristics is in the pre computation. While heuristic 1 just needs to compute the distances of every edge to the pretended stack, heuristic 2 has to perform the distances to every stack.

In L2 (C0), C0 has three blocking casks in front of it, so heuristic 2 make a huge difference. It reduces the number of explored nodes in 98.2%. Although it wastes about more than two sencods in pre calculations (Dictionary), it saves more than 14 seconds in heuristic computation. However in cases like s3 (C1), given the complexity of the map, the pre calculation time is huge for heuristic 2.

With that in mind, we can conclude that the first heuristic behaves better when the cask to retrive has no blocking casks and the map is huge. The second should be used in cases that we have blocking casks and that casks should be placed in the nearest stack available. This pre analysis can save us a lot of expanded nodes and thus time.

Another remark, if we always work with the same map the pre calculation time is only wasted once and from there we can compute as many times as we want the search algorithm with that heuristic.

Finally, the heuristic we attached to this report is heuristic 1. Our choice was based on the fact that this heuristic is consistent and thus it can find the optimal solution for every problem of this kind and the fact that takes less time in the pre computation of the dictionary.

References

[1] Norvig, aimacode *https://github.com/aimacode*. Modified at 26/05/2016.

[2] Russel, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.