David Pérez Carrasco | 25984360
Lucas Amar | 37082823
10/07/2022

CPSC 322 Assignment 1

Question 1 (27 points): Comparing Search Algorithms

For the algorithms in questions 1.1-1.4, answer the following questions:

1.1. [5 points] Depth-first search

(a) Expanded nodes = A, C, D, F, G, Z

(b) Path = [A,C,D,F,G,Z]

(c) Cost = 32

1.2. [5 points] Breadth-first search

(a) Expanded nodes = A, B, C, C, C, D, E, F, F, F, G, H, Z

(b) Path = [C,F,Z]

(c) Cost = 30

1.3. [5 points] A*

(a) Expanded nodes = C, A, B, D, E, F, C, C, F, F, G, H, Z

(b) Path = [C,F,H,Z]

(c) Cost = 25

1.4. [5 points] Branch-and-bound

(a) Expanded nodes = C, D, F, G, Z, F, H, Z, F, Z, C, F, Z

(b) Path = [C,F,H,Z]

(c) Cost = 25

1.5. [7 points]

(a) [1 point] Did BFS and B&B find the optimal solution for this graph?
Yes, BFS and B&B found the optimal solution for this graph.

(b) [3 points] Are BFS and B&B optimal in general? Explain your answer.
BFS is optimal in general if all costs are equal and B&B is optimal in general if h(n) is non-negative and admissible, although this last is not optimally efficient, since its first approach may not be the optimal solution, because it finds first the easiest solution to find treating the frontier as a stack.

(c) [3 points] Did B&B expand fewer nodes than A*? Explain if your answer is true in general for these two algorithms and why.

Yes, indeed B&B expanded fewer nodes than A*, and this is usual since B&B treats the frontier as a stack, avoiding expanding redundant nodes and getting to the goal node quicker. This is, indeed, the strongest point of B&B, that it is supposed to search the same optimal solution as A* in fewer steps, although it may be not complete if there are loops, as happens with DFS.

Question 2 (27 points): Uninformed Search: Peg Solitaire

2.1. [12 points] Suppose you were to write a program to solve peg solitaire using search. (You may use the labels provided in Figure 3 for referring to positions on the board, though this is not mandatory).

(a) [4 points] How would you represent a node/state in your code?

We are going to represent the board as a single array, consisting of 49 integers. We would consider a 1D array instead of a 2D matrix because of the space complexity, since in the case we were going to actually program this code, we should consider a lot of moves at each state, so a 2D matrix would be more expensive. The positions 1,2,6,7,8,9,13,14,36,37,41,42,43,44,48,49 will be equal to null, so we will not consider these positions, since they are representing the corners of the square board, which does not exists in a Peg Solitaire board. Then, for every value i in 1:49, except for those already commented, board[i]=1 if there is a peg in that position, board[i], otherwise (and board[i]=null in the commented positions). Therefore the start node is that in which all non-null positions of the board are equal to 1 except for the 25th one, which corresponds to the centre and then is 0. In the Figure 3 labels, 23, 24 and 25 positions will be p, x and A respectively.

(b) [2 points] In your representation, what is the goal node?

The goal node is that in which all the non-null positions of the board array are equal to 0 except for the 25th position, corresponding to the centre, which is equal to 1, the solitaire peg.

(c) [3 points] How would you represent the arcs? Note: you do not need to consider differences between valid and invalid moves; you may assume that some other part of your code would handle that.

The arcs represent a peg move. Since we do not have to consider differences between valid and invalid moves we will not consider too many constraints, but we basically will represent an arc as a change of position of a 1, followed by a conversion from 1 to 0 of the adjacent position. For instance if we move the peg in the 23th position to the 25th position in the first move, then the arc will transport the 1 in board[23] to board[25], and set 0 to board[24], as well as for board[23], which has just left its position. Hence the array will be filled with 0s until there will be one just 1 remaining, which should be in 25th position.

If we move one peg two positions down, it should be moved to the position whose number is the actual one plus 14, and the position between both (actual+7) would become a 0, and in the case moving up it should be resting 14 positions in the array, and 7 for the new empty position, and we assume that another part of the code will consider if this new position is valid and within the board limits or not, so the arcs will be represented just as this 1 move and a new 0 in the corresponding positions of the array.

(d) [3 points] How many possible board states are there? Note: this is not the same as the number of "valid" or "reachable" game states, which is a much more challenging problem.

If we do not consider the valid game states, then the all possible board states consists in all the possible combinations between the positions, if there is a pig or not in each one of them, then, since the board consists in 33 positions, we could have 2^33 possible states (each position could be empty or not), if we consider the board with 33 pigs, even one in the centre, as a valid state.

2.2. [9 points] The search tree:

(a) [6 points] Write out the first three levels (counting the root as level 1) of the search tree. (Only label the arcs; labeling the nodes would be too much work).

Considering the root level as:

root level: board = [null, null, 1, 1, 1, null, null, null, null, 1, 1, 1, null, null, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, null, null, 1, 1, 1, null, null, null, null, 1, 1, 1, null, null]

The first three levels will be, explaining just the arcs with the labels of Figure 3 and then with the labels in our code:

1. Root level
    - 1.1. Level 2 first node: Move pig in position o to x, free p. board[23]=0, board[24]=0, board[25]=1
        - 1.1.1. Level 3 first node: Move pig in position A to p, free x. board[26]=0, board[25]=0, board[24]=1
        - 1.1.2. Level 3 second node: Move pig in position K to p, free F. board[38]=0, board[31]=0, board[24]=1
        - 1.1.3. Level 3 third node: Move pig in position d to p, free i. board[10]=0, board[17]=0, board[24]=1
    - 1.2. Level 2 second node: Move pig in position B to x, free A. board[27]=0, board[26]=0, board[25]=1
        - 1.2.1. Level 3 fourth node: Move pig in position p to A, free x. board[24]=0, board[25]=0, board[26]=1
        - 1.2.2. Level 3 fifth node: Move pig in position M to A, free H. board[40]=0, board[33]=0, board[26]=1
        - 1.2.3. Level 3 sixth node: Move pig in position f to A, free k. board[12]=0, board[19]=0, board[26]=1

1.3. Level 2 third node: Move pig in position e to x, free j. board[11]=0, board[18]=0, board[25]=1
   1.3.1. Level 3 seventh node: Move pig in position G to j, free x. board[32]=0, board[25]=0, board[18]=1
   1.3.2. Level 3 eighth node: Move pig in position h to j, free i. board[16]=0, board[17]=0, board[18]=1
   1.3.3. Level 3 ninth node: Move pig in position l to j, free k. board[20]=0, board[19]=0, board[18]=1
1.4. Level 2 fourth node: Move pig in position L to x, free G. board[39]=0, board[32]=0, board[25]=1
   1.4.1. Level 3 first node: Move pig in position j to G, free x. board[18]=0, board[25]=0, board[32]=1
   1.4.2. Level 3 second node: Move pig in position E to G, free F. board[30]=0, board[31]=0, board[32]=1
   1.4.3. Level 3 third node: Move pig in position I to G, free H. board[34]=0, board[33]=0, board[32]=1

We are assuming that a part of our code will discard non-valid moves, so for instance the moves that implies not jumping any other pig are not considered in the tree. Otherwise it could create loops, for example if one son of the first node of level 2 is returning back the pig in x to o position, jumping the free position p.

(b) [3 points] What can you say about the length of the solution(s)?

First, we have to say that if we also had considered the non-valid moves, then the number of nodes of the third level would be also 4 per father, so there will be 16 in the third level instead of 12. But starting from the fact that we only considered valid moves, the length shows us that the second move has less chances than the first one, but it is just because the first moves have to be done in the laterals, but depending on the moves the fourth and other moves could have more than 4 chances, although it is a fact that as the game progresses there will be less leafs per level since there will be less pigs to move. The length also tells us the great amount of how many possible states could be in a game, since in just 2 moves we have created 12 different states, and if we also considered non-valid moves it could be a ridiculous number of possible states, the same as commented before. The length of the solution also tell us that there are multiple ways of reaching the solution, since no two nodes from the second level coincide.

2.3. [6 points] The search algorithm:

(a) [3 points] What kind of search algorithm would you use for this problem? Justify your answer.

We would use a heuristic search algorithm as A* pr B&B, since there are almost countless ways of end states, but just one goal state, which can just be reached with a few approaches, so a heuristic search could tell us how far are we from the goal node in each state of the board, which could help us to determine which pig to move and where in order to reach the goal.

(b) [3 points] Would you use cycle-checking and/or multiple-path-pruning? Justify your answer.

We would use both of them in order to optimise our solution and avoid long cycles. Without them we would not be able to determine cycles in our path, which could turn a linear approach of getting the goal node into an exponential one. Moreover, with multiple-path-pruning we will not only be able to find cycles but to find less costly paths to the same state in case there are different paths which can get to the same node, which will lead us to the most optimal solution.

Question 3 (24 Points) FreeCell

3.1. [15 points] Suppose you were trying to write a program to solve FreeCell using search.

(a) [7 points] How would you represent a node/state in your code?

To represent a node in the FreeCell problem, I would use 8 arrays for the columns, and 2 arrays of length 4, one for the foundations and one for the placeholders.

I would treat the 8 columns as stacks as they work as LIFO. The foundations have a length of 4, one per suit, and in each spot would be the highest card added to it.

The placeholders would also be a length of 4, one per placeholder, and in each spot would be the card being held there. If a suit's foundation has not been started or the placeholder is not in use, the value stored would be *null*.

Each card would be represented in the arrays as having its value and suit i.e., 3H = 3 of hearts.

(b) [3 points] In your representation, what is a goal node?

The goal node would have all 8 column arrays and the placeholder array empty and the king of each suit in the foundations' array.

(c) [5 points] How would you represent the arcs? Note: you do not need to consider differences between valid and invalid moves; you may assume that some other part of your code would handle that.

Each arc represents a card being moved. The code would therefore change 2 of the 10 total arrays.

- ☐ If a card is added to a column to another, it would be pushed into the stack representing that column.
- ☐ If a card is moved from a column it would be popped from the stack representing that column.

- ☐ If a card is moved to a placeholder, *null* would be replaced for that card in the first position available in the placeholders' array.
- ☐ If a card is removed from a placeholder, it would be replaced for a null in the array and then the array would be reorganised to make sure the array is filled from the left to the right.
- ☐ If a card is moved to a foundation, the corresponding value representing the foundation would be replaced for its succeeding card that is being added.

3.2. [9 points] Give an admissible heuristic for this problem; explain why your heuristic is admissible. More points will be given for tighter lower bounds; for example, "h(n)=0 for all n" is a trivial (and useless) heuristic, and thus it will award you no points.

We would use A* search saying that a move has a cost of 1. The heuristic cost would be subtracting the number of cards in the foundations from 52. Therefore, you know how many cards you still need to add to those piles. This will never overestimate given that you will have at least 1 move per card remaining at any point. So the heuristic will be h(n) = 52- f(n), where f(n) is the number of cards in the foundations in node n.

To calculate the number of cards in the foundations would be to do the sum of the values of the cards in the foundations with A->1, J -> 11, Q->12, K->13 and the card's number for the rest of the cards.

However, this could be excessively optimistic in some cases, since it would not consider that some cards are blocked from the foundation. In order to take this into consideration, we could add m(n), which could be a heuristic optimistic estimate of the number of moves needed to be done to unblock some blocked cards. The expression, hence, will be something like this:
h(n) = 52 - f(n) + m(n)

And taking into consideration this new value will help to unlock blocked situations eliminating cycles.

Question 4 (15 points) Modified Heuristics

See how A* works with this setting and report the number of nodes expanded as well as the optimal path found [1 point]. This is your baseline.

Number of expanded nodes: 9

Optimal path found: UBC→JB→KB→DT→SP

```
CURRENT PATH:
  UBC --> JB --> KB --> DT --> SP (Goal)
Path to last Goal Node: UBC --> JB --> KB --> DT --> SP (Goal) Cost: 34.8
Nodes expanded: 9

NEW FRONTIER:
  Node: KB    Path Cost: 37,0    Path: UBC --> KD --> MP --> KB
  Node: DT    Path Cost: 38,9    Path: UBC --> KD --> JB --> KB --> DT
  Node: BBY   Path Cost: 34,5    Path: UBC --> JB --> KB --> BBY
  Node: BBY   Path Cost: 35,4    Path: UBC --> KD --> MP --> BBY
  Node: RM    Path Cost: 34,7    Path: UBC --> KD --> MP --> RM
  Node: BBY   Path Cost: 46,1    Path: UBC --> KD --> JB --> KB --> BBY
```

Output from the A* algorithm with the original heuristic

[10 points] Report your findings. Give brief (but informative) descriptions of what you did and the results (particularly in terms of the numbers of nodes expanded and the solution paths found). You may include relevant screenshots to illustrate your report.

IMPORTANT: Note the point value of this question compared to the other questions in this assignment. I'm not looking for a massive lab report or academic paper; you should be aiming for a small set of scenarios that sufficiently illustrate how the performance of A* can change with changes to heuristic accuracy and admissibility.

First, we notice that if we increase all the values in the same way the returned path is the same but the number of expanded nodes changes. For instance, if we add 2 to all the nodes, the number of expanded nodes is 7, and if we add 10, then there are just 7 expanded nodes, and the same with all the greater values. We can see in the screenshots that when increasing the general heuristic the algorithm considers less possible paths. However, if we decrease the value it still expands 9 nodes. When we add the values in different amounts it has a similar impact. For example if we multiply all the values by 2, then the algorithm expands 7 nodes, but if we multiply them by 5, it expands just 5. Again if we subtract different amounts, for instance dividing the values by 2, it still expands 9 nodes.

```
CURRENT PATH:
  UBC --> JB --> KB --> DT --> SP (Goal)
Path to last Goal Node: UBC --> JB --> KB --> DT --> SP (Goal) Cost: 34.8
Nodes expanded: 8

NEW FRONTIER:
  Node: KB    Path Cost: 30,0    Path: UBC --> KD --> JB --> KB
  Node: KB    Path Cost: 37,0    Path: UBC --> KD --> MP --> KB
  Node: BBY   Path Cost: 34,5    Path: UBC --> JB --> KB --> BBY
  Node: BBY   Path Cost: 35,4    Path: UBC --> KD --> MP --> BBY
  Node: RM    Path Cost: 34,7    Path: UBC --> KD --> MP --> RM
```

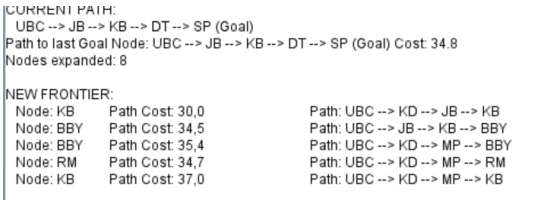
Returned path with h(n)=h(n)+2 for all nodes

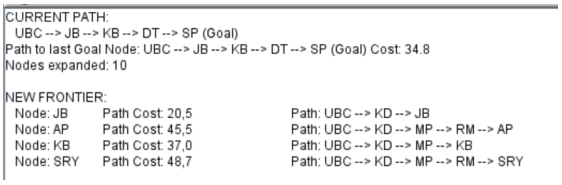
```
Path to last Goal Node: UBC --> JB --> KB --> DT --> SP (Goal) Cost: 34.8
Nodes expanded: 7

NEW FRONTIER:
  Node: MP    Path Cost: 24,5    Path: UBC --> KD --> MP
  Node: KB    Path Cost: 30,0    Path: UBC --> KD --> JB --> KB
  Node: BBY   Path Cost: 34,5    Path: UBC --> JB --> KB --> BBY
```

Returned path with h(n)=h(n)+10 for all nodes

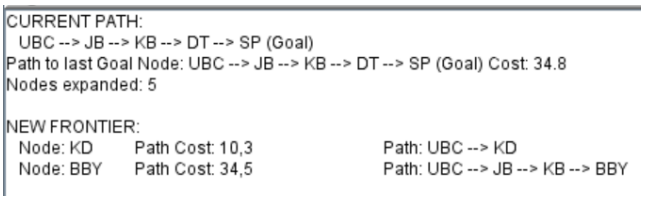
When it comes to the optimal path heuristic, if we increase it, surprisingly the number of expanded nodes changes, and it actually depends on the amount. If we decrease their heuristic it does not change, which makes sense since it is already the optimal path and the values can be decreased just a bit, but if we add 2 to all the heuristic values, then there are just 8 expanded nodes, but if we add 20 or more, the number of expanded nodes starts increasing, and now it is at least 10, as we see in the following screenshots.





Output with h(n)+=2 in the optimal path Output with h(n)+=20 in the optimal path

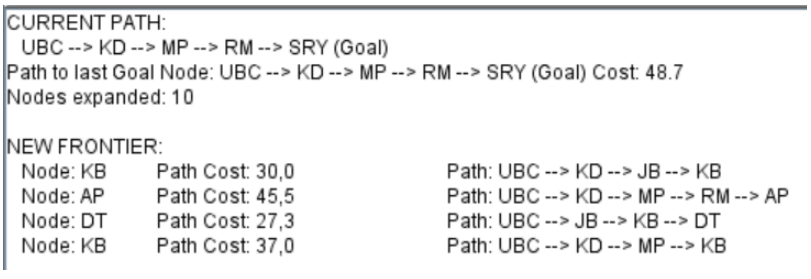
On the other hand, if we change the values of all the nodes except for the optimal path, the heuristic also changes, but in a more linear way. If we decrease the values of all these nodes the number of expanded nodes does not change, it does not matter how much we do decrease them, which makes sense since these nodes now have less heuristic but there were already 9 expanded nodes so that number will not change. However, if we increase their heuristic, the number of expanded nodes decreases accordingly. If we increase all the values by 10, then now there are 8 expanded nodes, but if we multiply all the values by 10, then there are just 5, which makes all the sense because with higher values they have less chances to be considered since the algorithm discards their high-costly paths.



Output with h(n)*=10 in all the nodes except for the optimal path

If we set all the values to be the exact minimum cost from each node to the goal the output is the same as the original one, which make sense because making all the goals having the same minimal value makes the algorithm works as if there were no heuristic values, just considering the arc costs, which will lead to the same output as the original approach.

Finally, if we set another node to be goal node we will not find its path with the first approach, since it will find less costly the path to SP, but if we arbitrarily change the heuristic values, making the path to SP seem more costly than the path to SRY, then we will get a path to Sry as we see in the following screenshot.



Output with SRY as a goal node and h(n) changed arbitrarily

[4 points] Summarize your findings (in a single sentence, if possible): what makes a heuristic "better"?

Setting lower heuristic values for the optimal paths and higher for the other paths makes a heuristic better.