Assignment One: Search

Question One

(a) Node: A state represents which yachts are in which berths.

In one representation, this can be done with a 10-element vector $b$ where the $i^{th}$ place in $b$ represents the $i^{th}$ berth. So if $b[i] = j$ means yacht $j$ is in berth $i$. If the $i^{th}$ place in the vector is set to 0, i.e., $b[i] = 0$, then the $i^{th}$ berth is vacant. It could also include a variable whose value is the empty berth (so it doesn’t need to search for it).

Another representation is to for the positions to correspond to the yachts, so that, $yp[i]$ specifies the berth of yacht $i$. You should also record where the empty berth is, which could be $yp[0]$.

number of states: There are $10! = 3628800$ states. This is the number of permutations of the 9 boats with the empty berth.

Goal node: The goal node is the state representing the original assignment $[0,1,2,3,...9]$ or $[0,1,2,3,...9,0]$ (depending on which representation you use and what to do with position 0, and whether arrays are assumed to start counting at 0 or 1).

Arcs: In the first representation, an arc can be represented by the berth of the yacht being moved to the empty position. Thus, if the action is $i$, the yacht at berth $i$ swaps with the empty position.

In the other representation, an action could specify the yacht which moves into the empty position. Then the action does a swap of $yp[i]$ and $yp[0]$.

The important thing is that the action provides enough information to determine the resulting state given and initial state.

(b) There are only two legal actions on the start state shown: moving yacht 2 from berth 1 to 9 and moving yacht from berth 3 to berth 9. The subtree “under” the second action is the smallest of the two (as there are fewer multiples of 3 than of 1), so students will probably expand that one....

(c) the choice is among BFS DFS and IDS. IDS is the best because it is complete, admissible (for arcs of unit cost) and has linear space complexity. Given not much memory would prevent BFS.

(d) The branching factor is 9 because there are at most 9 boats that could be moved from any state.

(e) time complexity of IDS: $O(9^m)$, space complexity $O(9 \times m)$

Question Two

(a) The state space can be seen as a triple $\langle x, y, c \rangle$ where $x$ is the horizontal position, $y$ is the vertical position and $c$ is a Boolean variable that specifies whether the robot has coffee. There are 87 states
(there are 45 squares the robot could be at with coffee and 42 squares the robot could be without coffee).

(b) A graph that represents the search space:

On the left is the states without coffee, and on the right are the states with coffee. There is a state for each white square that is not labeled with a $C_i$ on the left graph, and a state for each white square on the right graph. The neighbors of a node are the white squares adjacent to it, except for the nodes that would lead to one of the $C_i$ squares in the left graph—these lead to the corresponding $C_i$ node on the right graph.

(c) An admissible heuristic. For a node $\langle x, y, true \rangle$ (i.e., a state where the robot has coffee), $h(\langle x, y, true \rangle)$ is the Manhattan distance from $\langle x, y \rangle$ to the location $G$. For a node $\langle x, y, false \rangle$ (i.e., a state where the robot doesn’t have coffee) $h(\langle x, y, false \rangle)$ is the minimum (over $i \in \{1, 2, 3\}$) value of the Manhattan distance from $\langle x, y \rangle$ to $C_i$ plus the Manhattan distance from $C_i$ to $G$. This would be an exact distance, if not for the walls.

(d) Here are the states with the nodes in the order they are expanded in an $A^*$ search with multiple path pruning. The number in the center is the order the node is expanded. The number at the bottom is the $f$-value of the node. This assumes that for the nodes with the same $f$-value, the node added last is expanded first. There are multiple possible solutions, depending on the order the neighbors are considered when the $f$-values are the same.
Note that the path found goes right from S, then turns left to C3 then up and right to G.

(c) How will your heuristic and $A^*$ change if we assume that there is a hill going down towards the right so that steps to the right cost 0.5 and steps to the left cost 1.5, and steps up and down cost 1 each.

You can modify the Manhattan distance by seeing whether you go right or left and use 1.5 for the cost when you have to go left and 0.5 when it is right. $A^*$ search does not change as every node has the same $f$-value (the costs and the $h$-values are different, but the differences cancel out when summing).

**Question Three**

Code that implements (a) and (b) is in `searchSolution.py` in the code distributed for Assignment 2.

(a) We need to change just the definition of search to include a test as to wether the node is visited, and to add nodes to visited when expanded:

```python
def search(self):
    """returns next path from an element of problem’s start nodes
to a goal node.
Returns None if no path exists.
    """
```
while not self.frontier.empty():
    path = self.frontier.pop()
    if not self.multi_path_pruning or path.end() not in self.visited:
        self.display(2, "selecting: ", path)
        if self.multi_path_pruning: self.visited.add(path.end())
        if self.problem.is_goal(path.end()):
            self.display(1, "There are", len(self.frontier.frontierpq),
                          "nodes in the frontier and",
                          self.frontier.frontier_index - len(self.frontier.frontierpq),
                          "nodes have been expanded."
                          )
            return path
        else:
            neighs = self.problem.neighbors(path.end())
            for arc in neighs:
                self.add_to_frontier(Path(path, arc))
            self.display(3, "Frontier:", self.frontier)
            self.display(1, "Total of", self.frontier.frontier_index, "nodes expanded.")

(b) This is almost identical, but checks whether the node is in the path to that node, rather than in
self.visited. See code for assignment 2.

(c) In add_to_frontier use one of path.cost or self.problem.heuristic(path.end())
instead of the sum to get the methods. You need to compare the 9 combinations of (A*, best-first, least-
cost) × (no-pruning, loop-checking, MPP). Note that best-first with no pruning probably did not halt.
A* multiple path pruning should be the best. You should get the same answer as the AIspace search
algorithm gives.

Question Four

In all of these graphs, we assume that we are on a plane, with Euclidean distance (straight-line distance)
as the arc cost and as the heuristic function. We also assume that the neighbors are ordered from left to right.
The start node is s and the goal node is g.

(a) Give a graph where depth-first search is much more efficient (expands fewer nodes) than breadth-first
search.

Here breadth-first search expands every node, whereas depth-first search expands five nodes:

```
    s
   / \    
  a   b
   \ /     
  c   d     
     \    
    h  i  j  k
     \   / 
      g  l  m  n
```

(b) Give a graph where breadth-first search is much better than depth-first search.

Here depth-first search expands every node, whereas breadth-first search expands three nodes:

```
    s
   /  
  a   
    / 
   c   e
    /  
   b   f
    / 
   d   
    / 
   h   
    / 
   j   
    / 
   i   
    / 
   g   
```
(c) Give a graph where A* search is more efficient than either depth-first search or breadth-first search. Here depth-first search and breadth-first search expand every node, whereas A* search expands 4 nodes.

(d) Give a graph where depth-first search and breadth-first search are both more efficient than A* search. Here depth-first search expands three nodes, breadth-first search expands 4, yet A* search expands every node.
(e) In all of these graphs $A^*$ in the backward direction is more efficient than in the forward direction — there is only one path from $g$ to $s$ in the inverse graph. It doesn’t depend on the arc costs or the heuristic function.

Question Five

(a) This is the same as question 3(b). It is worth more because question 5(c) is more difficult.

(b) It acts like a breadth-first search when the values are equal. It can act like a depth-first search by adding $-$ before `self.frontier_index` in the argument to `heapq.heappush` in `Frontier.add`.

(c) There are lots of things you could say. For example:
   
   When the graph is a tree, and $A^*$ acts like depth-first search and the bound is slightly above the cost of the optimal path, $A^*$ and branch-and-bound expand the same nodes, and have the same count.
   
   The number of nodes expanded after branch-and-bound finished is a lower bound to the number that $A^*$ expands.
   
   I thought the following would be true (but it isn’t according to my experiments with the cyclic delivery problem, and I’m not sure why): branch and bound always explain less than or equal to the number of nodes that $A^*$ expands plus the number of element on the priority queue with $f$-value equal to the cost of the optimal path. This corresponds to the worst-case of branch-and-bound where it finds the goal last.

Question Six

It should not have taken more than a few hours. Most of this should have been in understanding the material, not in doing busy work. I hope it was reasonable, and you learned something.