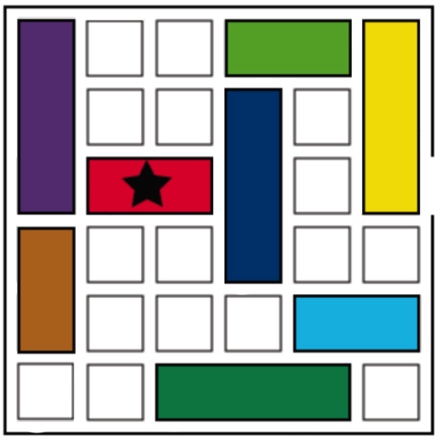
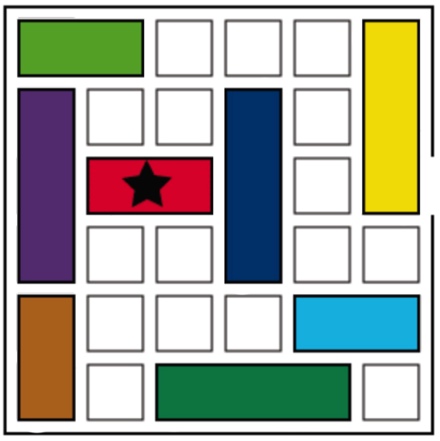
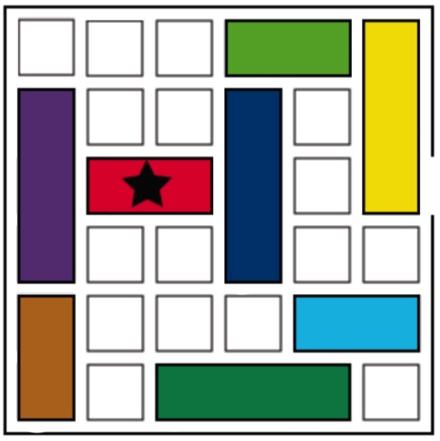
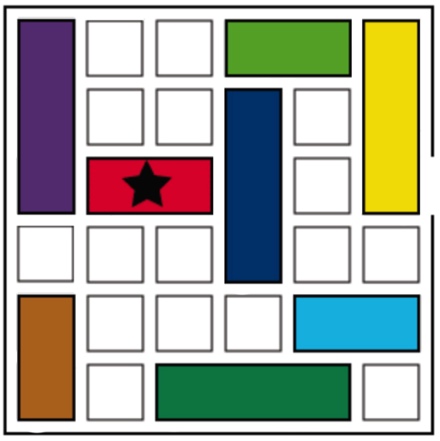
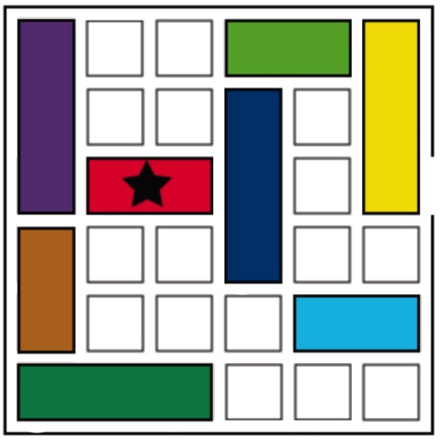
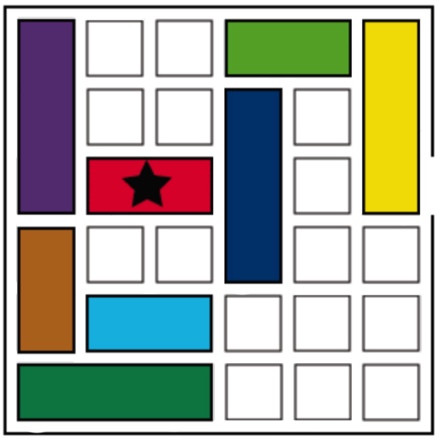
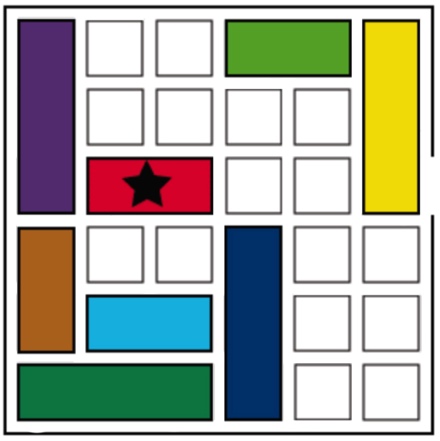
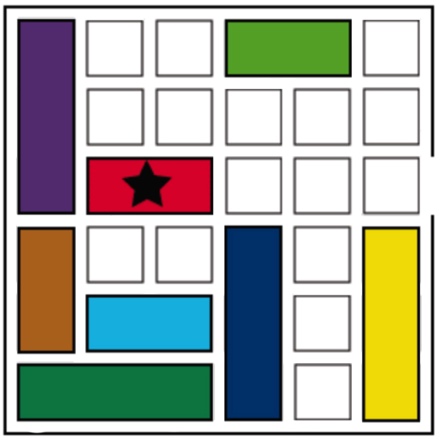
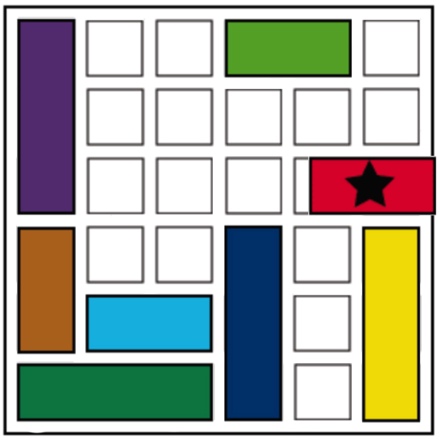
Hunter Leise

CSCI 3202

Problem Set 1

* 1.  0 1 2

 3 4 5

 6 7 8

* 1. To estimate the problem space for this puzzle, I multiplied the number of possible locations for each piece, while subtracting a tally of locations that conflict with one another:
* Light Green Car: 5 possible locations, 4 possible conflict states
* Purple Car: 2 possible locations, 2 possible conflict states
* Brown Car: 2 possible locations, 2 possible conflict states
* Red Car: 5 possible locations (goal state included), 4 possible conflict states
* Light Blue Car: 4 possible locations, 3 possible conflict states
* Dark Green Car: 4 possible locations, 5 possible conflict states
* Dark Blue Car: 4 possible locations, 7 possible conflict states
* Yellow Car: 4 possible locations, 7 possible conflicts states

Begin by adding up the possible conflict states and dividing them by two since all of the conflict states are being counted twice:

4 + 2 + 2 + 4 + 3 + 5 + 7 + 7 = 34 / 2 = 17 Conflict states

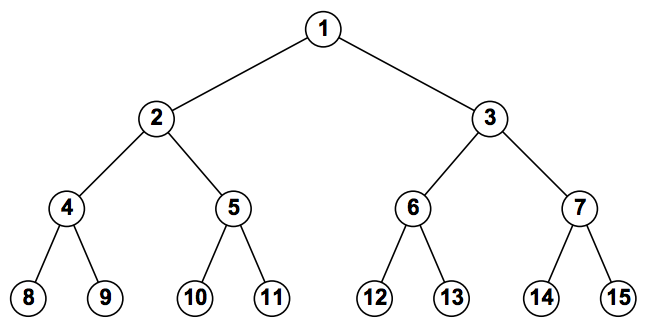
Then multiple all of the possible locations of each car and subtract the conflict states:

5 \* 2 \* 2 \* 5 \* 4 \* 4 \* 4 \* 4 = 25,600 – 17 = 25,583 states in the problem space

* 1. A plausible “h” function for A\* search would be the number of cars blocking the red car from the goal state. This would be easily computable and will always underestimate the number of moves remaining to solve the puzzle. For example, in the starting state of the rush hour puzzle from question 1.1, the “h” function would return 2 (since the dark blue and yellow cars are blocking the exit).
  2. My solution in 1.1 definitely has features in common with depth-first search because I committed to my early moves and continued until I found that my earlier moves didn’t result in a helpful state. Only then would I go back and try a different move. I don’t think my solution had many features in common with breadth-first search because I didn’t start off by trying every single possible first move, and then trying every possible second move for those, etc. My solution did have some features in common with means-ends analysis because I did analyze the difference between my current state and the goal state, and I did use that information to help dictate what my next move would be.

Bidirectional search would be a great idea for this puzzle because it’s generally easy to see what the goal state would need to look like (such that every car blocking the red car from the exit would be moved so it’s no longer blocking the exit). In addition, every move in rush hour is reversible, which is an important characteristic if one is to do search in the opposite direction from the goal state to the starting state.

2a)



2b) BFS: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

Depth-Limited (with limit 3): 1, 2, 4, 8, 9, 5, 10, 11

Iterative Deepening:

Depth limit 1: 1

Depth limit 2: 1, 2, 3

Depth limit 3: 1, 2, 4, 5, 3, 6, 7

Depth limit 4: 1, 2, 4, 8, 9, 5, 10, 11

2c) Bidirectional search would work very well in this problem since every move is reversible ( (n/2) or (n/2) – 1). The branching factor would be 2 moving forward (two children) and 1 moving backward (one parent).

2d) Yes, all you would need to do is start from the goal state and then make your way up the tree. Since each child only has one parent, it requires no search for you to work from the goal state up to the starting state.

2e) Yes, I can find an algorithm that outputs the solution to this problem without any search. I realize that all odd numbers would be “Right” actions and all even numbers would be “Left” actions. By working off of my solution from problem 2d, I would simply check if a number is even or odd, append the corresponding operation onto the front of a list, and recurse upwards by doing the same with its parent. For example, if the goal state was 11, you would place a “Right” operations since it’s odd. Its parent is a 5, which is also odd, so the operation array is [Right, Right]. Its parent is 2, which is even, so the operation array would be [Left, Right, Right]. No search required.

3) In response to the Searle paper, I find the combination reply particularly provocative. I think this links back to the classic zombie problem in philosophy which states, “how do you know any human other than yourself is conscious? What if they’re just zombies programmed to act like normal human beings without the conscious experience or sentience?” The combination reply seems like a reiteration of this problem, but with a being made of different materials than humans. The primary argument against the zombie problem is that we ourselves experience consciousness, therefore we have a strong reason to believe that others with minds and bodies similar to our own also experience said consciousness. In the case of an artificially intelligent robot, we can’t make this same connection. This extends to other animals as well. How do we know dogs or cats are conscious? I think the likely answer for now is that we simply can’t know. Searle is right in that complex processing doesn’t inherently entail understanding, but I don’t think it necessarily discredits understanding and consciousness just because the being doing that complex processing isn’t human.

4.1) I edited my program from question 5.2 by eliminating the goal state test such that it would run through every single possible state in the problem space. Once this is finished, I could just test the length of the visited states array. By doing so, I found that there were 63 possible states in the problem space.

4.2) My Code (Python 3)

*# Class to represent the problem space  
# The starting state and capacity lists should be the same length*class ProblemSpace:  
 def \_\_init\_\_(self, \_startingState, \_capacity):  
 self.start = \_startingState  
 self.capacity = \_capacity  
  
 *# DFS for the starting state and capacity of the problem space  
 # Goal state is when the last two cups both contain 2 quarts* def milkDFS(self):  
 stack = [(self.start, [self.start])] *# Add the starting state to the stack* visited = []  
  
 while stack:  
 (state, path) = stack.pop()  
 visited.append(state)  
 children = self.getChildren(state)  
 for child in children:  
 *# Check if the version of the state with the first two cups  
 # swapped is not in visited. Since both states are equivalent.* child2 = child.copy()  
 child2[0], child2[1] = child2[1], child2[0]  
  
 if (child not in visited) and (child2 not in visited):  
 *# Slight optimization such that if a child already exists  
 # in the stack, that must mean there's a faster  
 # way to get there, so skip it.* if (not any(child in childPath for childPath in stack)) and \  
 (not any(child2 in child2Path for child2Path in stack)):  
 if child[2] == 2 and child[3] == 2: *# Check if goal state* return path + [child]  
 else:  
 stack.append((child, path + [child])) *# DFS  
 # stack.insert(0, (child, path + [child])) # BFS* return [] *# Returns an empty list if there is no solution  
  
 # Returns a list of child states for the given state* def getChildren(self, state):  
 children = []  
  
 *# Loop through all possible 'to' and 'from' cups* for cupFrom in range(0, len(state)):  
 for cupTo in range(0, len(state)):  
 if (state[cupFrom] > 0) and \  
 (not self.isFull(state, cupTo)) and \  
 (cupFrom != cupTo):  
 childState = state.copy()  
 if childState[cupFrom] > self.getOpenSpace(childState, cupTo):  
 childState[cupFrom] -= self.getOpenSpace(childState, cupTo)  
 childState[cupTo] = self.capacity[cupTo]  
 else:  
 childState[cupTo] += childState[cupFrom]  
 childState[cupFrom] = 0  
 children.append(childState)  
  
 return children  
  
 *# Returns whether a cup is full, given a state and the cup's index* def isFull(self, state, index):  
 return True if state[index] == self.capacity[index] else False  
  
 *# Returns how much open space a cup has, given a state and the cup's index* def getOpenSpace(self, state, index):  
 return self.capacity[index] - state[index]  
  
  
problemSpace = ProblemSpace([40, 40, 0, 0], [40, 40, 5, 4])  
print(problemSpace.milkDFS())

Output

[[40, 40, 0, 0], [36, 40, 0, 4], [36, 40, 4, 0], [36, 36, 4, 4], [36, 36, 5, 3], [40, 36, 1, 3], [40, 39, 1, 0], [40, 39, 0, 1], [40, 34, 5, 1], [40, 34, 2, 4], [40, 38, 2, 0], [40, 38, 0, 2], [40, 33, 5, 2], [40, 33, 3, 4], [40, 37, 3, 0], [36, 37, 3, 4], [36, 37, 5, 2], [36, 40, 2, 2]]

This is not the optimal solution, which can be expected since DFS does not guarantee the optimal solution. When I run my code with BFS, however, I get the optimal solution of:

[[40, 40, 0, 0], [35, 40, 5, 0], [35, 40, 1, 4], [39, 40, 1, 0], [39, 40, 0, 1], [34, 40, 5, 1], [34, 40, 2, 4], [38, 40, 2, 0], [38, 36, 2, 4], [40, 36, 2, 2]]