Bin Dong

CS 2710 HW 1 Report:

1. The project needs to be run in Python 3.6 and above (the version my machine uses is python 3.6.4). My machine is an Intel Core i5 MacBook Pro with 2.7 GHz processor speed (MacOS High Sierra v.10.13.6).

2. I have briefly discussed the project with three people: Professor Hwa, Jianfeng He, and Zhuang Zinan. My discussion with Professor Hwa was similar to my discussion with a classmate, Jianfeng He. We talked about the different ways you could obtain the possible states for the water jug problem. They both directed me to drawing it out on a piece of paper and then deciding what are all the possible outcome given a specific state. From there, I was able to come up with my own code function that generates all the possible child state given a specific state. My discussion with a friend Zhuang Zinan, who is not in the class but had taken a similar course, was more about coming up with a heuristic function for the burnt pancake flipping problem. He directed me to look at only the goal state. He says to pay particularly close attention to the pattern between one pancake to another (specifically since we know that they are ordered, how are they ordered and what's the pattern between one pancake to the next). From there, I was able to come up with my own heuristic function for the pancake flipping problem by comparing any given state to the goal state and determining what properties must the state have in order to be the goal state, other than the two states being equal. Outside of discussion, I have used other source to get to know the algorithm better, specifically videos on how the algorithm works. The links to these videos are: BFS and DFS, Uniform Cost, Astar, and idastar. Please do note that these videos are examples of how the algorithm works but not how to code them (the videos only show demonstration on a high level and no code). The DFS video was a little bit different than what the class implementation is but it is from that video that I was able to refresh my mind on how a DFS algorithm looks. I also have used Wikipedia for more information about heuristics requirements to be admissible and consistent.

3. There are currently no known bugs in the program that I am aware of.

4. The action expansion order are as follows:

    a. Path Finding Puzzle:

        i. The order of expansion is in the same as the order in which the vertices are given to us in the config file. For example, let's say in the config file you have ("City_A", "City_B", 10) followed by ("City_A", "City_C" 10) in the following line. Then that means A's first child is B (since A->B comes before A->C in the config file). This also means that B's first child is A (since the graph has undirected edges). A's second child will be C (because again, C comes after B in the config file). But if you have

("City_A", "City_C" 10) followed by ("City_A", "City_B", 10) in the config file, then that means A's first child will be C and A's second child will be B (C comes before B in terms of line order).

ii. The heuristic function used for this puzzle is called "euclidean" (without the quotations and case sensitive). Euclidean distance makes the most sense to be an heuristic function. What is the shortest distance from point A to point B if there are no obstacles in the way? That would be a straight line. Euclidean Distance gives us that straight line. Since we are given the coordinate location of each vertices and we know the end goal, we can calculate the straight line distance by applying the formula $A^2 + B^2 = C^2$ where $A = (x_2 – x_1)$ and $B = (y_2 – y_1)$. Solving for C yields the straight line which is the shortest distance between the two path.

iii. The transcript of the run can be found in the same repository titled all_test_city_transcript.txt. The transcript will detail out bfs, dfs, iddfs, unicost, greedy, astar, and idastar for cities.config and test_cities.config.

iv. The transcript that only runs what is required (unicost, greedy, astar) as stated in the project writeup on can be found for test_cities_transcript.txt

b. Water Jug Puzzle:

   i. The order of expansion is: Filling, Emptying, and Transferring.
      1. The Filling is from Filling Jug 1 to Filling Jug 2. If there is a Jug 3, we Fill Jug 3 next.
      2. The Emptying is from Emptying Jug 1, to Emptying Jug 2. If there Is a jug 3, we empty Jug 3 next.
      3. The Transferring is a little bit more messier. The order of expansion for Transferring is:
         a. Transferring Jug 1 to Jug 2.
         b. If there is a jug 3, We transfer jug 1 to Jug 3.
         c. Transfer Jug 2 to Jug 1.
         d. If there is a jug 3, we transfer Jug 2 to 3.
         e. If there is a jug 3, we transfer jug 3 to jug 1.
         f. If there is a jug 3, we transfer Jug 3 to jug 2.

   ii. One example of this would be:
      1. Given state [1,0] and the max capacity is [3,3]
      2. Filling Jug 1 = [3,0]
      3. Filling Jug 2 = [1,3]
      4. Since there is no jug 3, we do nothing.
      5. Emptying Jug 1 = [0,0]
      6. Emptying Jug 2 = [1,0]
      7. Since there is no jug 3, we do nothing.
      8. Transferring Jug 1 to 2 = [0,1]
      9. Transferring Jug 2 to 1 = [1,0] (Since [1,0] already exist in step 6, we don't care about this state)

10. The possible actions will be in this order: [3, 0], [1, 3], [0, 0], [1, 0], [0, 1].

iii. The heuristic function used for this puzzle is called "volume" (without the quotations and case sensitive). The way this heuristic function works is that it first checks couple of conditions. If any of the conditions is true, we would return 1. If anything else, we would return a number 1 to n where n is the number of the jug (in this problem, we would return a number 1 to 3). Before I explain what the conditions are, I want to explain my code a tiny bit. Since we know for sure that the maximum jug that will be tested is three, I have engineered my code such that upon taking an input that only has 2 jug, I would assume that there is 3 jugs (I don't use the 3rd jug at all… its hidden away). The list that contains two jugs typically should look like this [jug1, jug2]. But I re-engineered it to look like this: [jug1, jug2, 0] (length of 2 vs length of 3). This is so that I can reuse the code no matter if there is 2 jugs or 3 jugs. As you can see from my actions above, it checks if there is a 3rd jug (I would calculate the number of jugs there are before re-engineering the list to size 3). With that said, I viewed each index of the jug as either X,Y,Z. The first index of the list will correspond with X. The second index of the list will correspond with Y. The third index of the list will correspond with Z, i.e., if jugs = [1,2,0], then X = jugs[0], Y = jugs[1], and Z = jugs[2]. The conditions are:
    1. If X+Y == Goal State's X (which is goal[0]) or X + Y = Goal State's Y (which is goal[1] or X + Y = Goal State's Z (which is goal[2]… again, we reengineered the starting state and the goal state to be of length 3).
    2. If its false, then we check if X + Z == Goal_X or X + Z == Goal_Y or X + Z = Goal_Z.
    3. If that fails, then we check if Y+Z == Goal_X or Y+Z == Goal Y or Y+Z == Goal_Z.
    4. If any of the conditions in step 1-3 returns true, we would return 1.
    5. If it all fails, we would count how many jugs are not matched. For example, let's if the goal state is [1,2,3] and our current state is [3,3,3], then we wound return 3 (since none of the jugs matched). If our current state is [1,3,4] then we would return 1 since only the first jug (index 0) matched. If the current state is [2,2,3], then we would return 2 since both the second jug and the 3rd jug matched (index 1 and index 2).
    6. The reason why I designed it this way is because for the water jug problem, it is really about how many jugs matched up and trying to get all the jugs to be just like the goal state. Since the heuristic function never returns an integer above the size of the jug, we only need to consider cases where it is only 1 or 2 steps away

from the actual goal state (because for this specific project, 3 jugs is the max, then that means the max estimation is of value 3. For an heuristic function to be admissible, it needs to be optimistic and predict a value less than actual cost. That means we don't need to consider any cases where we know for sure that it is more than 3 steps away because anything above 3 steps away, our function will be optimistic if not exact). After a long series of testing, I see that the solutions that takes 2 steps are typically 2 jugs that's not being matched up (either dump two jugs out, transfer 2 jugs, or fill, or any combination of these actions). I see that my heuristic function can estimate <u>**MOST**</u> of the states accurately but not for all. Since the heuristic can handle most of the cases and it was extremely challenging to come up with a better heuristic, I decided to go with this one.

iv. The transcript can be found in the same repository titled all_test_water_jugs_transcript.txt. The transcript will detail out bfs, dfs, iddfs, unicost, greedy, astar, and idastar for jugs.config and test_jugs.config.

v. The transcript that only runs what is required (bfs, dfs, greedy) as stated in the project writeup can be found in test_jugs_transcript.txt

c. Burnt Pancake Flipping Puzzle

i. The order of expansion is from the bottom to top. What this means is that the spatula will be placed at the bottom of the pancake stack first and slowly move its way up. For example, if the state is [-4, -3, -2, -1], then the first children will be placing the spatula on the bottom (after -1) and flip, yielding [1,2,3,4]. The second children will be placing after -2, yielding [2,3,4,-1]. The third children will be placing after -3 yielding [3,4, -2,-1]. The fourth child will be placing after -4 yielding [4, -3, -2, -1]. The last child will be placing before -4 (flipping nothing) which yields [-4,-3,-2,-1].

ii. The heuristic function used for this puzzle is called "FlipPoints" (without the quotation and case sensitive). The heuristic work as follows:

1. The pancake must be in increasing order. And they must be increasing by 1. If the pancake isn't in increasing order of 1, then for each pancake that breaks the rule, 1 cost is added to the prediction cost. For example, if there are 3 pancakes that are not in increasing order, then cost 3 will be returned.

    a. Consider the following state: [-5,-4, 2,1,-3]

        i. -5 to -4 is in increasing order of 1.

        ii. -4 to 2 is not in increasing order of 1.

        iii. 2 to 1 is in increasing order of 1

        iv. 1 to -3 is not in increasing order of 1.

        v. Return 2 (2 pancakes breaks the rule)

b. If they are in increasing order of 1, and the sum of the pancakes are negative, return 1. Otherwise return 0 (because if they are in increasing order of 1 and they're not negative, then the goal state has been found)

iii. The reason for this heuristic design is because after looking at the goal state closely, I have noticed that each pancake is in increasing order. For every pancake that is not in increasing order, we need to fix it. And that takes at LEAST 1 cost for every pancake that breaks the rule. Since it takes at least 1 cost, that gives us an under estimation compared to the actual cost. But we need to consider what happens if they are in increasing order of 1 but negative. Since they are increasing order of 1 but negative, it only requires 1 flip to get to the goal state. Thus if they are in increasing order of 1 and they sum up to a negative number, we can return 1. If they are in increasing order of 1 but sums up to a positive number, then a goal state has been found and we can return 0 (because h(goal) = 0).

iv. The transcript can be found in the same repository titled test_pancakes_transcript.txt. Please note that because of how long it takes to find a solution, I have decided to only run the 3 algorithms (iddfs, astar, and idastar) required by the project writeup rather than running every single algorithm for the two input configurations (test_pancakes1.config and test_pancakes2.config). If the program does not find a solution within 30 minutes, the program will automatically terminate.

5. Best Search:
   a. Path Finding Problem
      i. For the Path Finding puzzle and since we are trying to get the minimum distance, the best algorithm to use will probably be unicost. The reason for choosing unicost is simple. It considers all possible combinations of the path from the start state to the end state. Because it considers all possible solution, it is optimal (because exhaustive search is optimal). Since it is a path finding problem, it is safe to assume that the path has different cost and thus uniform cost (which is dijkstra's algorithm) can get you the shortest path cost.

      ii. This is a different story if all cost 1. If all the path has a cost of 1, then BFS would be best (it allows you to get to the goal state with the least amount of vertices). Since the path cost is 1, then the least amount of vertices is the optimal solution.

      iii. Using unicost, because we know that it is exhaustively search, it is obviously optimal. It is always complete in this example because our state space is finite (we are also keeping an explored list so no duplicates).

There is a hit on space complexity because it must keep an explored list but the time complexity is lowered because of the explored list (again, no duplicate states so if there is a duplicate state, we don't even look at it).

iv. Again, since the nature of the problem is to find the MINIMUM cost path, using uniform cost will always get you the optimal solution, which is why I think this is the best search strategy for this specific problem.

b. Water Jug Problem
   i. For the Water Jug Problem, I think BFS is the best searching algorithm. It gives you the least amount of steps to reach the goal but it keeps an explored list (a hit on space complexity) and it also takes a while (a hit on time complexity since it generates a lot of node). Because of the nature of the problem in this specific project (graph search), we have a finite state space. This means that we are always complete. Since each water jug problem is of cost 1, the least number of steps to get to the goal cost is the most optimal solution. One could argue that unicost is better than BFS. I cant disagree with that because they both keeps an explored list and they both give you the least number of steps to get to the solution. However, unicost is exhaustively searching for all possible path while BFS is just simply traversing down the graph. Both are complete and optimal though.

c. Burnt Pancakes Problem
   i. The burnt pancake problem is an easy one. It is obvious that greedy is the best algorithm if you have a decent heuristic function. The reason for this is that its quick! It's always trying to pick the shortest path based on the heuristic function to reach the end. The state space is simply too big for either BFS and DFS to handle (takes forever to find a solution). The same could be said for IDDFS which takes even longer because it resets in order to increase the depth. Uniform cost will look for all possible path and determine which one is the best (alike BFS in this case since all states have cost 1, but a lot slower because its exhaustively looking). Astar has the same problem with uniform cost. It exhaustively searches for all possible path. While it does consider the future cost, it still considers the past cost as well. Idastar has the same problem as iddfs. The fact that it resets makes it extremely slow to find a solution. That is why I think greedy is the best algorithm. It is not an algorithm that gets you the OPTIMAL solution but it does get you a solution relatively quick. The time complexity is small (because it only take what it considers as the best path at that instant in time). It does take a hit on the space complexity by keeping an explored list but at least it generates a solution at a really fast time.

d. My heuristic for the water jug problem is not consistent nor admissible. In order for it to be admissible, the predicted cost shoud always be less than the actual cost. Consider this state with 3 jugs: [1,2,4]. The maximum capacity is [2,2,4] and the goal state is [2,1,4]. Based on my heuristics, this will return a cost of 2. However, it is obvious that this only needs 1 step to reach the goal state (by dumping the 2nd jug into the first jug). My heuristics predicted a higher cost than the actual cost (and thus it's being pessimistic). Because the heuristic is not admissible, it cannot be consistent (according to the textbook Aritificial Intelligence A modern Approach Third Edition by Russel and Norvig on page 95).

e. My heuristic for the burnt pancake problem is admissible. It will never overestimate the cost mainly because it checks the ordering of the pancake and compare the current state's property to the goal state (which is that it must be in increasing order of 1 and the sum must be positive). If there is a specific pancake that doesn't break the rule, it will take at LEAST 1 cost to fix that. If there are n pancakes that does not follow the rule, it will take at LEAST n cost to fix that. My heuristics will always either under estimate or be exact on the prediction cost. However, I am unsure if it is consistent. In order to show that it is consistent, I must be able to prove that h(n) <= c(n, a, n') + h(n'). Since I am unable to provide a formal proof, I am unable to say for certain that it is consistent. I would argue that it is mainly because for every flip you do, it will cost 1. Since you are flipping in order to fix it, every flip, you either fix it or you don't fix it. If you don't fix it, h(n') will be the same as h(n). c(n,a,n') will be 1 (because you flipped it), thus c(n, a, n') + h(n') > h(n). However, let's say you fixed it. The cost predicted is equal to the total number of pancakes that has broken the rule. If you fix provide a fix to one pancake, then h(n') will reduce by 1 as well. Since you flipped the pancakes, c(n, a, n') will be 1. As a result, h(n) = c(n, a, n') + h(n'). However, this is not a formal proof and as a result, I cannot say with certain that it is also consistent. All I can say is that it is admissible.

f. The one thing I find interesting is that greedy was able to find a solution MUCH faster than astar. I always thought that greedy will beat out any other algorithms in terms of finding a solution but I was not expecting the difference to be that great. It also becomes increasingly more noticeable as the state space gets bigger and bigger. I have provided a transcript (greedy_on_pancakes_transcript.txt) for the pancake problem using greedy on test_pancakes1 and test_pancakes2's configuration. As you can see, it was able to spit out a solution extremely quickly (less than a second on test_pancakes2.config) while astar search took longer than 30 minutes. Granted, it is not optimal but at least there is a solution. In greedy_vs_other_algorithms.txt, you can see how greedy algorithm competes with other algorithm using only 5 pancakes (running test_pancakes3.config). As you can see, greedy beats out all other algorithms. Unicost took over 3 minutes and I had to interrupt it. It was really surprising to see just how much faster greedy is compared to other algorithms even on a smaller input size.