Samuel Singh – 1024640
MAI 5100

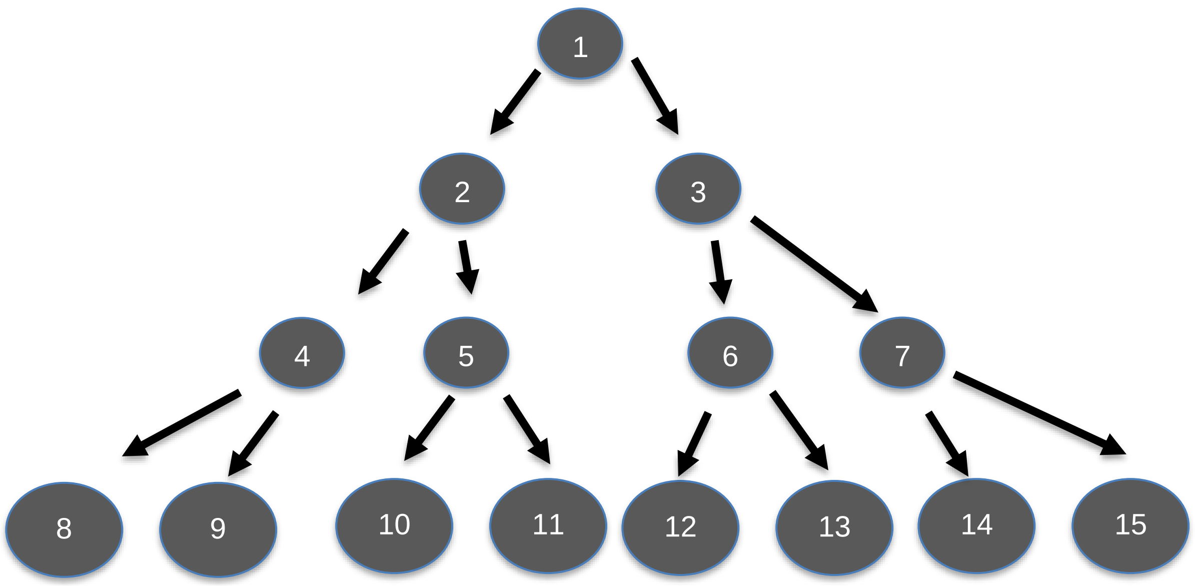
Homework 0
2025-03-25

Question 1

Consider a state space where the start state is number 1 and each state k has two successors: numbers 2k and 2k+1.

i. Draw the portion of the state space for states 1 to 15.
ii. Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3, and iterative deepening search.
iii. How well would bidirectional search work on this problem? What is the branching factor in each direction of the bidirectional search?
iv. Does the answer to (3) suggest a reformulation of the problem that would allow you to solve the problem of getting from state 1 to a given goal state with almost no search?
v. Call the action going from k to 2k **Left**, and the action going to 2k+1 **Right**. Can you find an algorithm that outputs the solution to this problem without any search at all?

i) State Space for states 1 to 15:



ii) Order of nodes visited:

a. Breadth First Search: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.
b. DLS with limit 3: 1, 2, 4, 8, 9, 5, 10, 11.
c. IDS combines depth-first search with increasing depth limits.
   i. For depth limit 0: node visited is 1.
   ii. Depth limit 1: Nodes visited are: 1, 2, 3.
   iii. Depth limit 2: Nodes visited are: 1, 2, 4, 5, 3, 6, 7
   iv. Depth limit 3: Nodes visited are: 1, 2, 4, 8, 9, 5, 10, 11
   v. The solution path then is: 1, 1, 2, 3, 1, 2, 4, 5, 3, 6, 7, 1, 2, 4, 8, 9, 5, 10, 11.

iii) Bidirectional search involves both movements from 'parent node' to 'child node' and also moving from the 'child node' back to the 'parent node.' Since the backwards movement is a movement from one node to one node the branching factor is 1 as opposed to the initial movement which involves moving from one node to two nodes; branching factor of 2. The backwards search therefore would be very efficient as each node only has one predecessor ie: k//2.

iv) Yes, the answer given above suggests that the problem could be reformulated in a manner that requires no search. If we model the problem as moving from the goal state(node 11) back to the start state(node 1) then by simply repeatedly dividing the current node state by 2 and flooring the value then we can traverse back to node one without any searching. Eg. At goal state, node 11:

a. 11//2 = 5

b. 5//2 = 2

c. 2//2 = 1 == goal state.

v) The state space is a simple binary tree structure where a node k has successors in the form 2k (or left) and 2k+1 (right). One possible algorithm to solve for the path from node 1 to node 11 is as follows:

a. The path will be solved for by moving backwards from the goal state to the start state and then reversing the order of movements:

```
i. Start at node 11

ii. While the node k != state node:

    1. If k == even:

        a. Parent is k/2 and Movement is to the left.

    2. If k == odd:

        a. Parent is (k-1)/2 and Movement is to the right
```

iii. This will yield the following:

1. k will be 11 at the initial state. Since k is odd this is a right movement..The parent is (11-1)/2 = 5.

2. k is now 5 which is odd so the movement is to the right. The parent is (5-1)/2 = 2.

3. k is now 2 which is even so the movement is to the left. Also the parent is 2/2 = 1.This is the start state so terminate the loop.

4. Reversing the order of moevements yields left, right, right.

Question 2

n vehicles occupy squares (1,1) through (n,1) (i.e., the bottom row) of an $n \times n$ grid. The vehicles must be moved to the top row but in reverse order; so the vehicle i that starts in (i,1) must end up in (n−i+1,n). On each time step, every one of the n vehicles can move one square up, down, left, or right, or stay put; but if a vehicle stays put, one other adjacent vehicle (but not more than one) can hop over it. Two vehicles cannot occupy the same square.

i. Calculate the size of the state space as a function of n.
ii. Calculate the branching factor as a function of n.
iii. Suppose that vehicle i is at (x,y). Write a nontrivial admissible heuristic hi for the number of moves it will require to get to its goal location (n−i+1,n), assuming no other vehicles are on the grid.
iv. Which of the following heuristics are admissible for the problem of moving all n vehicles to their destinations? Explain.
   a. $\sum h_i$
   b. $\max(h_1,\ldots,h_n)$
   c. $\min(h_1,\ldots,h_n)$

i) The size of the state space can be defined by examining a simplified model eg a 3x3 grid as shown below:

x	y	z

When no car is on the gird then then there is a total of n x n options ie. 3x3 = 9 options for placement. Once one car (x) is placed on the grid, the total available options for a subsequent car is reduced by one ie car y now has (nxn)-1 options or 9-1 = 8 options. Similarly, once cars x and y are placed on the grid, the total placement options for the third car is one less than before ie 7. So, the total number of combination possible here would be 9x8x7. To model this in a generalized manner, it can be rewritten in terms of n where n in this case is 3 so then the total number of combinations (the state space) would then be n^2! which would result in 9x8x7x6x5x4x3x2x1 but all values from 6 to 1 can be removed by simply dividing by 6! And 6! is the same as (n^2-n)! in this case. The final solution for computing this state space then would be (n^2!) / (n^2-n)!

ii) Each vehicle can move either up, down, left, right or they can make no movement(stay) so the total options for movements is 5 so technically the branching factor would be O(5^n) however there are possible conditions where the actual branching factor would be less eg. vehicles cannot occupy the same space so this would reduce the number of movement options for certain vehicles.

iii) A simple admissible heuristic for this problem would be the Manhattan distance. Given that the position of each vehicle at the initial state is in the form (x,y) where y = 1, and the goal position is defined as (n-i + 1,n) then the heuristic h_i can be defined as:

$$h_i = |(n-i+1)| + |n-y|$$

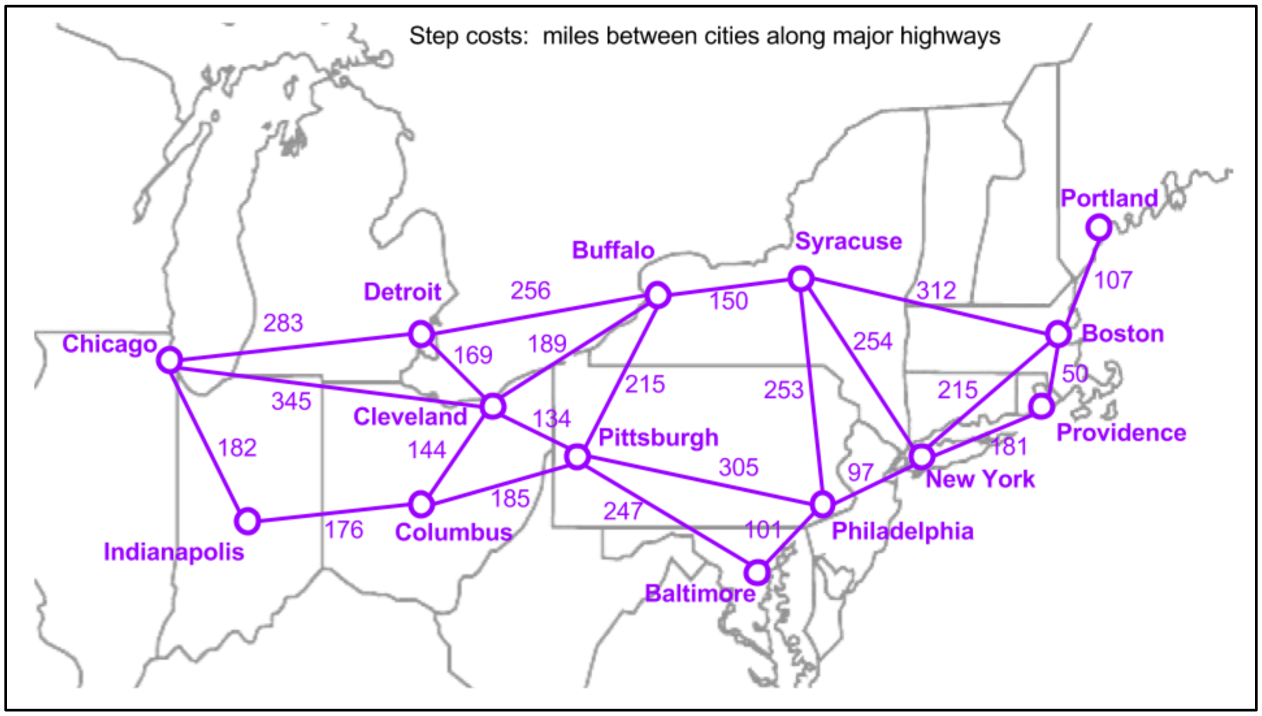
iv) Admissible or not:

a. ∑hi involves summing the heuristics of each car. Therefore, if any one car has a heurisitic that is not admissible, then the total heuristic might become inadmissible. ∑hi is admissible if each individual heuristic is admissible.
b. max(h1,…,hn) takes the largest heuristic from the set of individual heuristics. Since the true largest cost to get to the goal would be the cost to move the car that is furthest from the goal, then this statement would also be admissible.
c. min(h1,…,hn) is the inverse of the above in that it takes the smallest heuristic from a set of individual heuristics. This assumes that the cost to move a vehicle that is further from the goal is smaller than or equal to the cost to move a vehicle that is closer to the goal which is not true in reality and thus this heuristic is inadmissible.

Question 3

Consider the U.S. map shown below. The purple lines indicate major highways with the labeled mileages between the connected cities:

Cities: Chicago, Detroit, Buffalo, Syracuse, Boston, Providence, New York, Philadelphia, Baltimore, Pittsburgh, Cleveland, Columbus, Indianapolis, and Portland (ME).
Step Costs: Miles between cities along the purple highway segments (e.g., Chicago → Cleveland is 345 miles, Cleveland → Pittsburgh is 134 miles, etc.).



1. Goal: Plan a route from Chicago to Boston using each of the following search methods:

- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Uniform-Cost Search (UCS)
- A* Search

2. Procedure:

- Manually generate the search tree for each strategy.
- Label nodes in the order they are expanded (node 1 = root, node 2 = the second expanded, etc.).
- Include path costs (mileage so far) on each branch in your tree.
- Mark or shade any “failure” or pruned nodes as you discover they cannot lead to a better solution (for UCS or A*, for example).
- For A*, you must provide the heuristic values h(n) you used as you expand each node. A suitable heuristic would be the straight-line distance or a rough “driving distance” estimate from each city to Boston.

3. Solution Reporting:

- Solution Path: Indicate which path your search finds from Chicago to Boston.
- Total Route Cost: Provide the final total mileage.
- Optimality: State whether or not the solution found is optimal for each search method (hint: BFS and UCS typically guarantee optimality under standard assumptions; DFS does not; A* is optimal if your h(n) is admissible and consistent).

Depth First Search

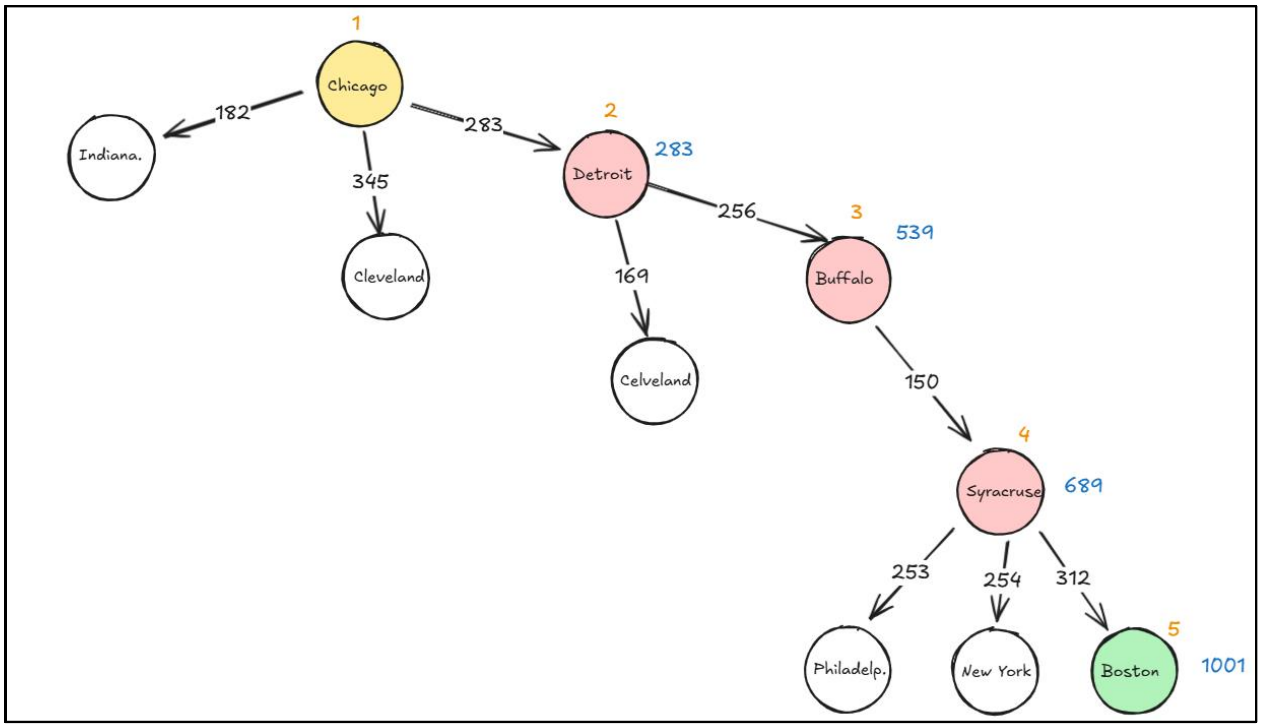


Figure 1 – DFS

The figure 1 shows the DFS search tree. The nodes were expanded in the order:

1. Chicago
2. Detroit
3. Buffalo
4. Syracruse
5. Boston

The final solution is Chicago, Detroit, Buffalo, Syracruse, Boston.
DFS search finds the solution with a total cost of 1001. While this solution is guaranteed to find a path to Boston, it doesn't guarantee the optimal path. It should be noted that the solution would change based on which node is explored first, eg in this case Detroit is explored first and subsequently Buffalo, etc. Alternatively, Indianapolis could have been explored first resulting in a different path to Boston.

Breadth First Search

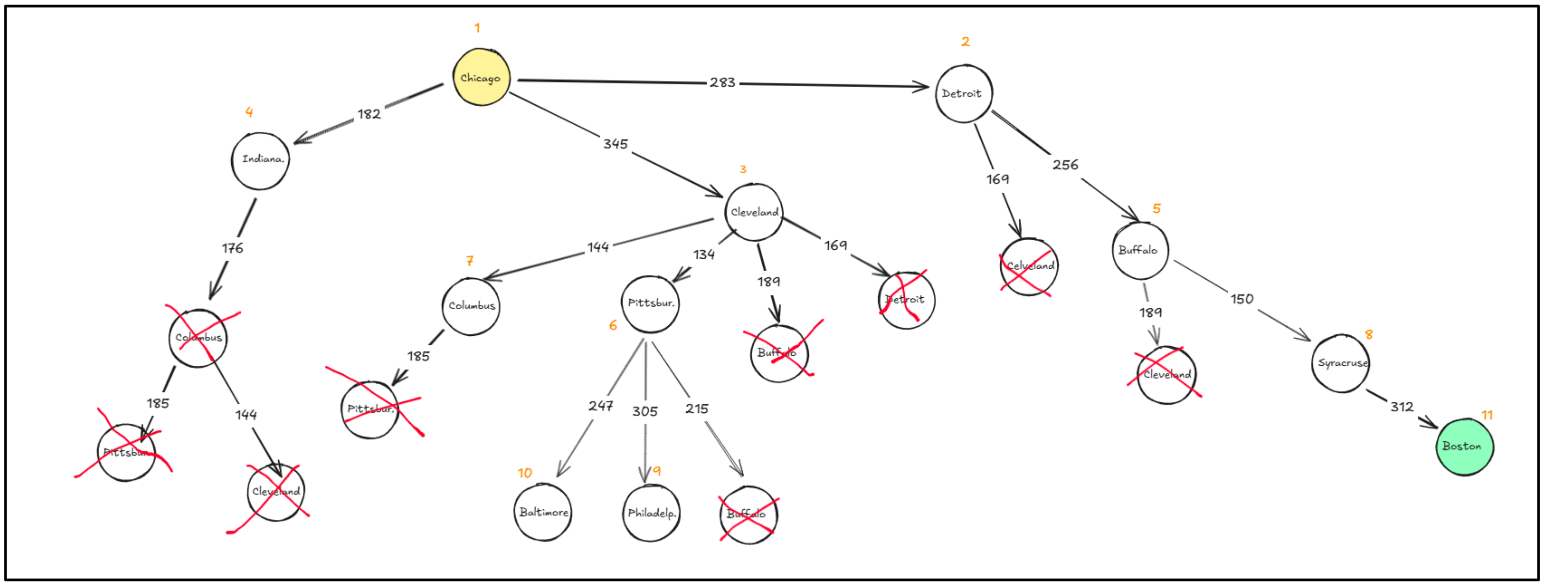


Figure 2 - BFS

The figure 2 shows the BFS search tree to get to Boston from Chicago. The red crosses indicate all locations that were previously visited. Based on this the nodes were expanded as follows:

1. Chicago
2. Detroit
3. Cleveland
4. Indianapolis
5. Buffalo
6. Pittsburg
7. Columbus
8. Syracruse
9. Philadelphia
10. Baltimore
11. Boston

The final solution to get to Boston then is in the order: Chicago, Detroit, Buffalo, Syracruse, Boston and costed a total of: 1001. As with DFS the solution would be different based on which node was explored initially, eg instead of exploring Detroit first, Indianapolis could be explored.
BFS will find a solution to Boston but it does not guarantee Optimality.

Uniform Cost Search

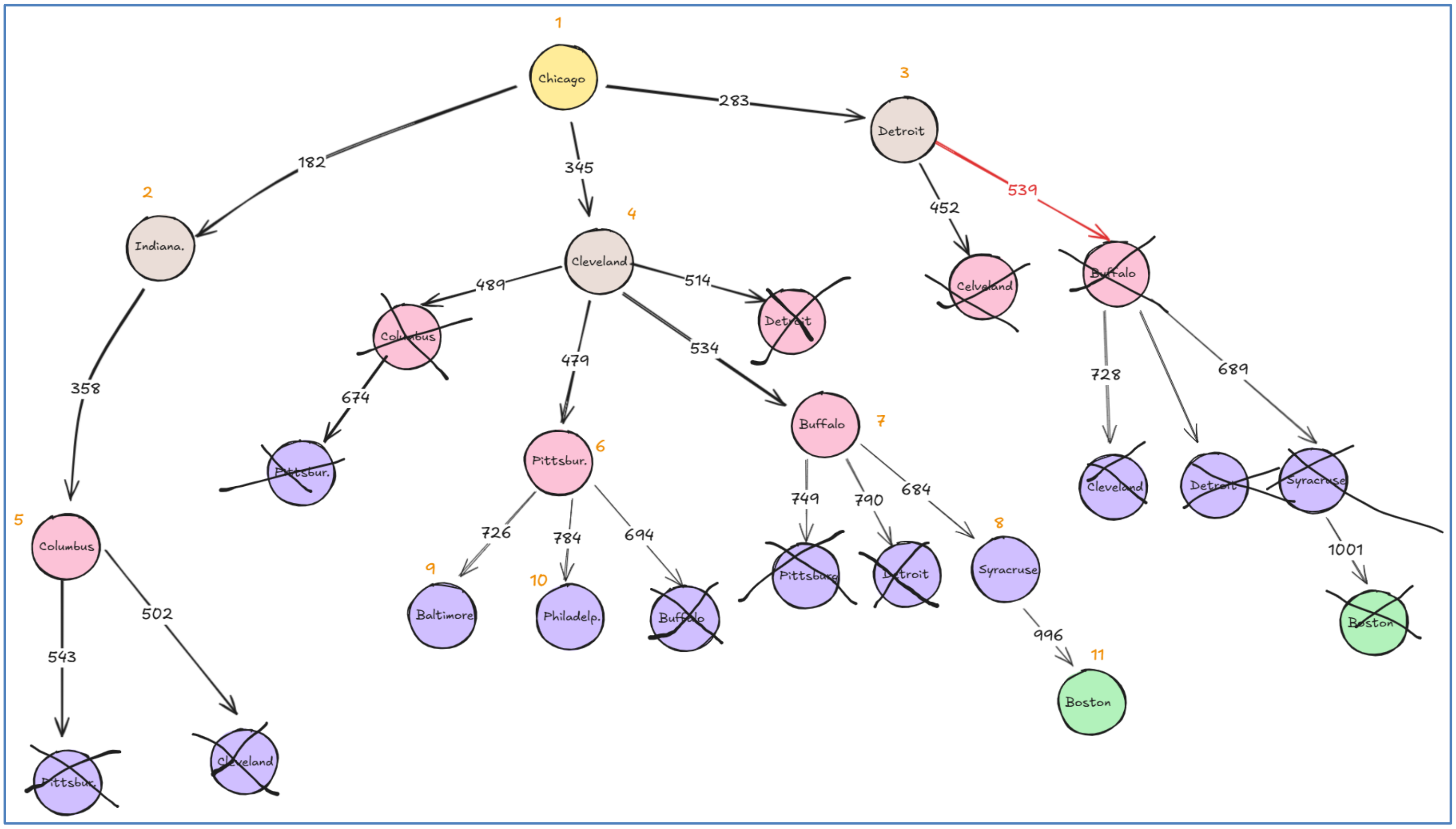


Figure 3 - UCS

The UCS search tree is shown in the figure above. The expanded states are as follows:

1. Chicago
2. Indianapolis
3. Detroit
4. Cleveland
5. Columbus
6. Pittsburg
7. Buffalo
8. Syracuse
9. Baltimore
10. Philadelphia
11. Boston

The solution path is Chicago, Cleveland, Buffalo, Syracuse, Boston with a total cost of 996. Since UCS considers the cost per path it is an optimal solution.

A* Search Tree

The following straight-line values are used for the A* heuristic:

- Chicago: 850 miles
- Detroit: 700 miles
- Indianapolis: 900 miles
- Cleveland: 600 miles
- Columbus: 700 miles
- Buffalo: 400 miles
- Syracuse: 300 miles
- Pittsburgh: 550 miles
- New York: 200 miles
- Philadelphia: 300 miles
- Baltimore: 400 miles
- Providence: 50 miles
- Portland: 100 miles
- Boston: 0 miles

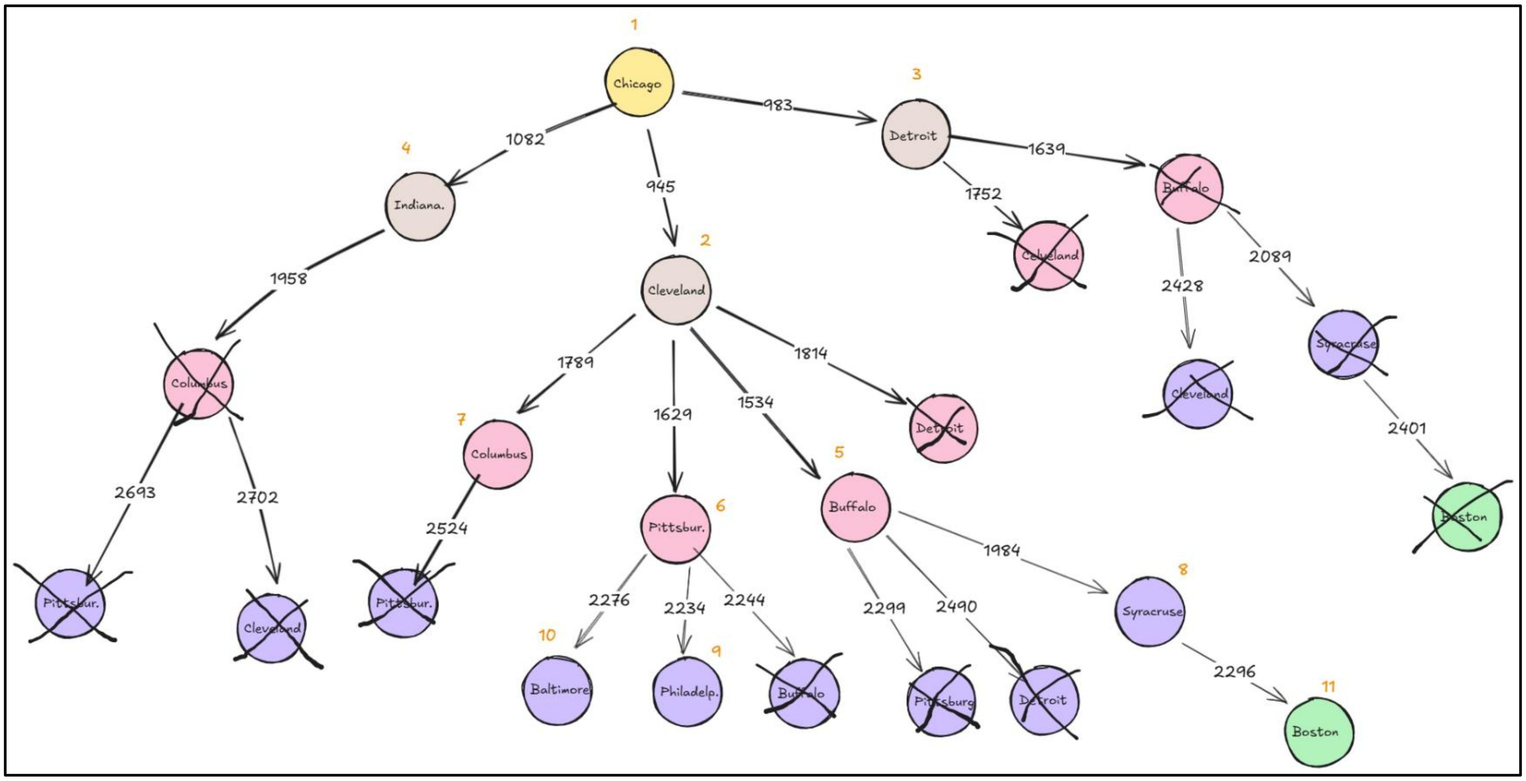


Figure 4 - A* Search Tree

The A star search applies the cost to get to each node plus the heuristic (straight line distance). Based on this, the expanded nodes are as follows:

1. Chicago
2. Cleveland
3. Detroit
4. Indianapolis
5. Buffalo
6. Pittsburg
7. Columbus
8. Syracuse
9. Philadelphia
10. Baltimore
11. Boston

The solution path then is Chicago, Cleveland, Buffalo, Syracuse, Boston with a total cost of 2296. Since the A* method applies both the path cost and a heuristic that is admissible the solution that is derived can be considered to be optimal.

Challenges

There were multiple areas that proved to be a bit annoying and tedious. During the initial stages of coding while trying to implement the DFS and BFS it was found that while the code I implement would sometimes arrive at the correct solution, the nodes that were expanded were not in the correct order. In other cases, it was found that some nodes were expanded more than once ie. they were revisited when they should not have. This was due to the way I was handling the visited states. Another issue that popped up was with handling one-way states. Prior to these I had expected the successor function to return at least two options to move on where at least one of those options would be an already visited state and therefore the next move forward would be to simply move to the next state that had not previously been visited. This would work for most cases assuming you had multiple options to work with. One simple but effective method that was used throughout the coding process to troubleshoot these and similar issues was to implement print statements at specific integral points of the code to see exactly what the code was doing under the various conditions. Another method that I adopted was to return a deliberately incorrect direction value to the called functions eg for the DFS and BFS I would return an array of one element ['WEST']. The code would of course fail the required tests but the autograder.py would highlight where the code went wrong and the requirements of the code for the various scenarios. This proved to be integral for the successful implementation of the various algorithms and saved a ton of time. It was found that prior to this, my agent might successfully complete the simpler tasks such as the tinymaze but would fail at either the medium or large maze.

Another challenge was the way I implemented the UCS algorithm. I started by using the heapq library to help me sort the priority queue. This at face value seemed like a good time saving idea and for the most part it worked. The library allowed me to very quickly and efficiently sort the queue needed to keep track of the lowest cost node and the path it took to get there. This method completely failed when I needed to complete the question 6 of the autograder. The heapq library was incapable of making comparisons with grid objects like the ones present in that question. This required a complete refactoring of the code which cost valuable time. This was a situation where a shortcut wound up costing more time. Lesson learnt.

An additional hurdle arose with the completion of the questions 5 through 8. This was primarily due to the time constraints and lack of knowledge on the required procedures to get the questions done. The initial phases of completing these set of questions were purely research based. It was after some time that the realization came that certain files such as the included util.py file had functions that would be useful for the completion of these questions. However, much time was consumed and thus attention was re-focused to the completion of the writeup aspect of the assignment.

Despite all of this the assignment was a very interesting and fun experience. It was incredibly exciting to see bugs in the code finally get resolved and witness the pacman agent navigate the mazes in real time. The assignment definitely set a strong basis for the completion of the more complex problems that are to come by giving me a hands on (yet sufficiently challenging) experience with problem solving via AI search agents. One concept that was reinforced was the way inadmissible heuristics could influence the overall response of the AI and the importance of determining the correct one. While all questions were not answered, the assignment also helped to reiterate the importance of adequate time management, especially in the event of repeated failures.