CIS 471: Homework 1

Zach Domke

1/29/19

Q1:

1. Variables: 1, 2, 3 … n (where n is the length of the words)

Domains: D = {a, b, c, d, e, … w, x, y, z}

Constraints: 1k, 2k, 3k, … nk is a word in the dictionary for each k

Solution: word1 == word2



1. Prefer Iterative deepening.

We do not want DFS because we could get caught in an infinite loop (bee -> bed -> bee …). We also do not want BFS because with 26 possibilities it can take quite a lot of memory to save all that data. Therefore Iterative Deepening is preferred because we can visit every option without getting in a loop, using too much memory, or going beyond n levels deep (where n is the length of the words.

Q2:

1. BFS: 781 expanded nodes, 3906 generated nodes, 3125 nodes in memory

Every node until the 4th layer is expanded and every node until the 5th layer is generated. There are so many nodes stored in memory because that is every node in the 5th layer (once BFS is done analyzing the 4th layer).

DFS: 0 expanded nodes, 8 generated nodes, 28 nodes in memory

There are no nodes expanded because DFS only ever generates the leftmost node available. Only 8 nodes are generated because at depth of 7 DFS analyzes and finds the goal, otherwise it would keep searching the leftmost node infinitely. Only 28 nodes in memory because we only need to store 4 nodes per layer (4 \* 7).

Iterative: 975 expanded nodes, 4881 generated nodes, 20 nodes in memory

We have so many expanded and generated nodes here because Iterative Deepening search regenerates the search tree each time it increases the depth limit. Only 20 nodes stored in memory at one time as we only need to store 4 per layer (4 \* 5).

1. DFS is best for the problem as presented in part a, but it does not find the optimal solution. Iterative Deepening would be best without the leftmost solution because it would actually find the answer (unlike DFS) and have less space requirement than BFS.

Q3:

2. We know that both h1(n) and h2(n) are admissible because they are both less than or equal to g(n). (h1(n) ≤ g(n) && h2(n) ≤ g(n))

|  |  |  |  |
| --- | --- | --- | --- |
| Node | h1(n) | h2(n) | g(n) |
| A | 10 | 10 | 13 |
| B | 9.5 | 10.5 | 12 |
| C | 8 | 10 | 11 |
| D | 5 | 5 | 7 |
| E | 1.5 | 2 | 2 |
| F | 3 | 2.5 | 4 |
| G | 0 | 0 | 0 |

1. We know that h1(n) is consistent if every edge is estimated (through the heuristic) at less than or equal to the actual cost of the edge.

|  |  |  |
| --- | --- | --- |
| Edge | h1(n – m) | Actual Cost |
| A -> B | .5 | 1 |
| A -> C | 2 | 4 |
| A -> D | 5 | 7 |
| B -> D | 4.5 | 5 |
| C -> D | 3 | 4 |
| D -> E | 3.5 | 5 |
| D -> F | 2 | 3 |
| E -> G | 1.5 | 2 |
| F -> E | 1.5 | 2 |
| F -> G | 3 | 5 |

1. We know that h2(n) is not consistent if not every edge is estimated at less than or equal to the actual cost of the edge. (in red)

|  |  |  |
| --- | --- | --- |
| Edge | h2(n – m) | Actual Cost |
| A -> B | -.5 | 1 |
| A -> C | 0 | 4 |
| A -> D | 5 | 7 |
| B -> D | 5.5 | 5 |
| C -> D | 5 | 4 |
| D -> E | 3 | 5 |
| D -> F | 2.5 | 3 |
| E -> G | 2 | 2 |
| F -> E | .5 | 2 |
| F -> G | 2.5 | 5 |



|  |  |  |  |
| --- | --- | --- | --- |
| Search Algorithm | A-D-E-G | A-D-F-G | A-B-D-F-E-G |
| Depth First |  |  | Returns this |
| Breadth First | Returns this |  |  |
| Uniform Cost |  |  | Returns this |
| A\* w/ h1 |  |  | Returns this |
| A\* w/h2 |  |  | Returns this |

2. E = [0, 2]
3. E = [1.5, 2]
4. E = [1.5, 3]

Q4:

* 1. One 8-puzzle heuristic is to sum up the total number of tiles not in the correct column and the total number of tiles not in the correct row. We know that this heuristic is admissible because if a tile is in the incorrect row, it requires at least one move. This applies to a tile in the wrong column as well. We know that if a tile needs to be moved at least once and it is only being counted in the heuristic once per incorrect column/row, then h(n) ≤ g(n).
  2. Another 8-puzzle heuristic would be similar to the previous one, but to also count the number of rows/columns the tile is off by. (e.g. if a tile is in the left row, but needs to be in the right row, then we add 2) This could be visualized as h(n) = |goalX(k) – currentX(k)| + |goalY(k) – currentY(k)| for all k tiles in n

This would be admissible as it would require that you only need to consider how many rows/columns a tile is off by, and we could only move it at most one row/column closer to our goal.