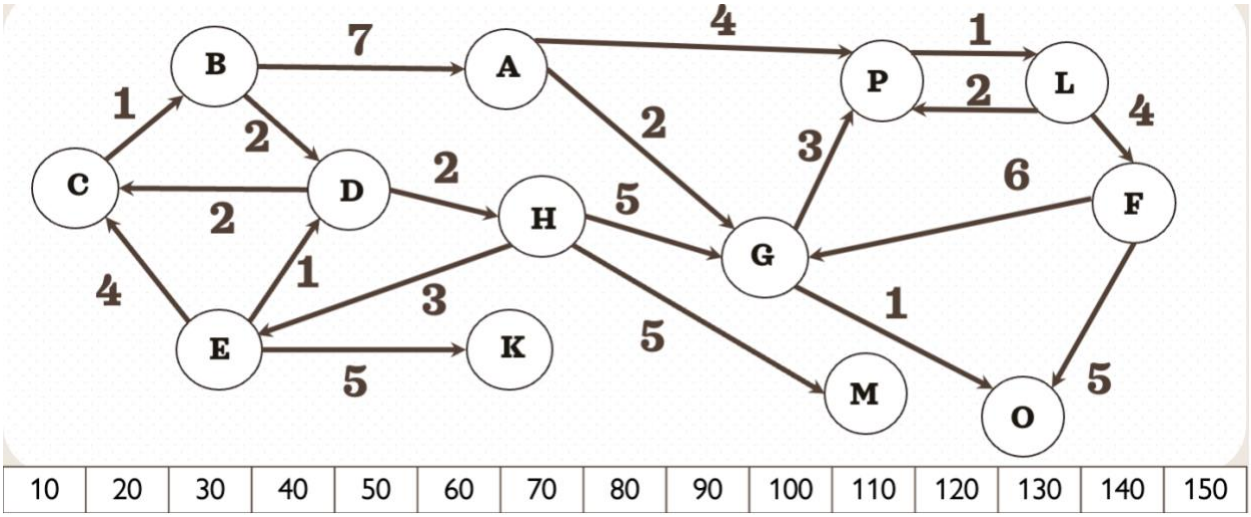
Kaylin Brooks

April 23, 2025

CSC-3130-02

Assignment 7: Graphs

Problems: Use the graph below for problems 1 and 2. Note that the bottom numbers represent a representation of geographical location of each node



1 (text) Dijkstra [15 points] Find the shortest path from E to all other nodes using the Dijkstra Algorithm (Show your work and intermediate steps)

Solution:

We'll start at node `E` with distance 0 and use a priority queue to update the shortest known distances.

Initial distances:

```
E = 0, all others = ∞
```

Step-by-step:

- From E → D (1) → update D = 1
- From E → C (4) → update C = 4
- From D → C (1+2=3) → update C = 3 (better)
- From D → H (1+2=3) → update H = 3
- From C → B (3+1=4) → update B = 4
- From B → A (4+7=11) → update A = 11

- From H → G (3+5=8) → update G = 8
- From A → P (11+4=15) → update P = 15
- From G → P (8+3=11) → update P = 11 (better)
- From G → F (8+6=14) → update F = 14
- From G → M (8+1=9) → update M = 9
- From G → O (8+5=13) → update O = 13
- From P → L (11+2=13) → update L = 13
- From L → F (13+4=17) → worse than current 14, skip
- From M → O (9+1=10) → update O = 10 (better)

Final shortest distances from E:

```
E: 0, D: 1, C: 3, H: 3, B: 4, K: ∞, A: 11, G: 8, P: 11, L: 13, M: 9, O: 10,
F: 14
```

2 (text) A* [20 points] Find the shortest path from E to all other nodes using the A* Algorithm (Show your work and intermediate steps) **Hint:** Use the geographical location values to derive the heuristic to use.

Solution:

Follow the A* method using the formula:

f(n) = g(n) + h(n)

- `g(n)` is the cost from E to node n
- `h(n)` is the estimated cost from node n to the goal using the *geographical distance* as a heuristic (specifically, the difference in node positions as shown on the horizontal line at the bottom)

Step-by-step A* from E to all nodes

Start at E:

- Position: 30
- `g(E) = 0`
- We add E to the priority queue: `f(E) = 0 + h(E) = 0`

Expand E's neighbors:

E connects to:

- **C (4)**:
  - `g(C) = 4, h(C) = |10 - 30| = 20`
  - `f(C) = 4 + 20 = 24`
- **D (1)**:
  - `g(D) = 1, h(D) = |40 - 30| = 10`
  - `f(D) = 1 + 10 = 11`
- **H (3)**:
  - `g(H) = 3, h(H) = |70 - 30| = 40`
  - `f(H) = 3 + 40 = 43`

Queue: `[(D,11), (C,24), (H,43)]`

Expand D:

From D, we have:

- **C (2)**:
  - `g(C) = 1 + 2 = 3`, which is better than previous `g(C) = 4`
  - `h(C) = 20`
  - `f(C) = 3 + 20 = 23`
- **H (2)**:
  - `g(H) = 1 + 2 = 3`, which ties with earlier g(H)
  - `h(H) = 40`, `f(H) = 3 + 40 = 43`

Update queue: `[(C,23), (H,43)]`

Expand C (g=3):

From C:

- **B (1)**:
  - `g(B) = 3 + 1 = 4`
  - `h(B) = |20 - 30| = 10`
  - `f(B) = 4 + 10 = 14` Update queue: `[(B,14), (H,43)]`

Expand B:

From B:

- **A (7)**:
  - `g(A) = 4 + 7 = 11`
  - `h(A) = |70 - 30| = 40`
  - `f(A) = 11 + 40 = 51`

- **D (2)**: already visited with lower g

Queue: `[(H,43), (A,51)]`

Expand H (g=3):

From H:

- **A (2)**:
  - `g(A) = 3 + 2 = 5` < current g(A) = 11
  - `h(A) = 40`, `f(A) = 5 + 40 = 45`
- **G (5)**:
  - `g(G) = 3 + 5 = 8`
  - `h(G) = |90 - 30| = 60`
  - `f(G) = 8 + 60 = 68`

Update queue: `[(A,45), (G,68)]`

Expand A:

From A:

- **G (2)**:
  - `g(G) = 5 + 2 = 7` < previous 8
  - `h(G) = 60`, `f(G) = 7 + 60 = 67` Queue: `[(G,67)]`

Expand G:

From G:

- **F (6)**:
  - `g(F) = 7 + 6 = 13`, `h(F) = |140 - 30| = 110`
  - `f(F) = 13 + 110 = 123`
- **P (3)**:
  - `g(P) = 7 + 3 = 10`, `h(P) = 70`, `f(P) = 10 + 70 = 80`

Continue like this until all reachable nodes are visited.

Final A* Path from E to H:

- E → D → H with total cost = `1 (E→D) + 2 (D→H) = 3`
- Path: **E → D → H**

3 (text) Comparison [5 points] Which algorithm found the shortest path to H in less iterations Dijkstra or A*? (Explain your answer)

Solution:

I think A* found the shortest path to H in fewer steps than Dijkstra.

Explanation:
They both found the same shortest path, but A* was faster because it used a shortcut — it looked at how far each node was from the goal to decide which path to try next. Dijkstra just checked everything step-by-step without that extra help, so it took more steps.

7 (text) Algorithm Analysis [10 points] For each of the algorithms you wrote for problems 4-6, explain their time complexity and space complexity using Big-O notation. Explain how you arrived at your answer.

Solution:

Problem 4: Directed vs. Undirected

For this one, I checked if an adjacency matrix was symmetric. If `matrix[i][j]` is not equal to `matrix[j][i]`, then the graph is directed.

- **Time Complexity: O(n²)**
  I looped through every cell in an `n x n` matrix, so the runtime depends on the square of the number of nodes.
- **Space Complexity: O(n²)**
  I had to store the full matrix in memory, which takes up space proportional to `n²`.

Problem 5: Find All Paths with Weight 7

In this problem, I used a depth-first search (DFS) to find all simple paths between two nodes where the total weight equals 7. I kept track of the current path and visited nodes to avoid cycles.

- **Time Complexity: O(V! / (V−k)!)**
  I realized that the number of paths can grow really fast because I'm trying every possible combination of nodes that adds up to weight 7. So, it's exponential in the worst case.
- **Space Complexity: O(V)**
  The recursion stack and path tracking can go as deep as the number of vertices, so the space is linear with respect to the number of nodes.

Problem 6: Draw Circular Graph with Edges

For this part, I generated edges by going `x` positions left and right in a circular list. I used GraphStream to visually draw the graph.

- **Time Complexity: O(n)**
  Each node created exactly 2 edges, and I looped through each node once, so the time grows linearly with the number of nodes.
- **Space Complexity: O(n)**
  I stored a list of nodes and edges, and GraphStream handled the rest. So, the memory used increases with the number of nodes and edges, but still stays linear.