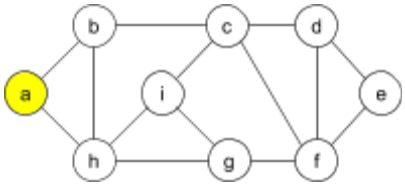
CS 1332 - Week 13

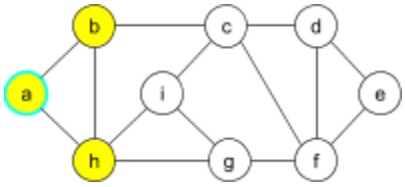
Monday

Topics

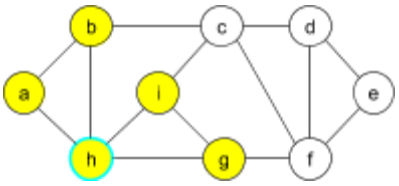
I. Breadth-First Search (BFS)
   A. Premise - search through the graph by first visiting everything one edge away from the start vertex, then everything 2 edges away, then 3 edges away, etc.
      1. eg. the level-order BST traversal is breadth-first
   B. Requirements:
      1. a starting vertex
      2. a visited set (eg. a hashset of vertices)
      3. a queue
      4. *a visited/return list (retains the order of the VS to return to user)
   C. Algorithm:
      1. Initialize visited set (VS)
      2. Initialize queue (Q)
      3. Mark the start vertex as visited (add it to VS)
      4. Q.enqueue(start)
      5. While the Q is not empty (and the VS does not contain all the vertices yet)
         a) v ← Q.dequeue()
         b) for all w adjacent to v
            (1) if w is not visited
               (a) mark w visited & add to return list
               (b) Q.enqueue(w)
   D. Example - visited, current vertex



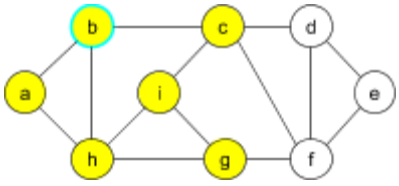
VS: A
Q: A



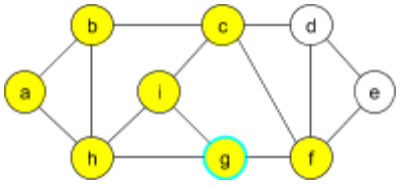
VS: A, H, B
Q: H, B



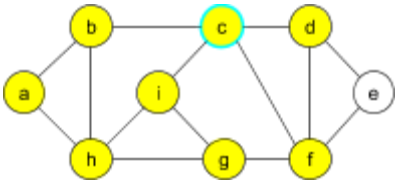
VS: A, H, B, I, G
Q: B, I, G



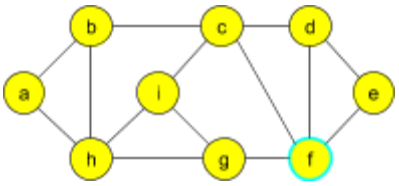
VS: A, H, B, I, G, C
Q: I, G, C



VS: A, H, B, I, G, C, F
Q: C, F



VS: A, H, B, I, G, C, F, D
Q: F



VS: A, H, B, I, G, C, F, D (all are visited, we can stop)
Q: E

II. Depth-First Search
  A. Premise - follow one path as far as you can before going back to try another
    1. eg. the pre-, post-, and inorder BST traversals are depth-first
  B. Requirements:
    1. a starting vertex
    2. a visited set (eg. a hashset of vertices)
    3. a **stack** (a stack data structure or the recursive stack)
    4. *a visited/return list
  C. Iterative Algorithm (with the stack data structure):
    1. Initialize visited set (VS)
    2. Initialize **stack** (S)
    3. **S.push(start)**
    4. While S is not empty (and the VS does not contain all the vertices yet)
      a) v ← **S.pop()**
      b) if v is not visited
        (1) mark v visited & add to return list
        (2) for all w adjacent to v
          (a) if w is not visited → **S.push(w)**
  D. Recursive Algorithm (with the recursive stack):
    1. Public wrapper
      a) Initialize VS and return list
      b) Call helper method on the start vertex
    2. Private helper (takes current vertex, graph, VS, return list)
      a) if the current vertex, v, is not visited
        (1) mark v visited & add to return list
        (2) for all w adjacent to v
          (a) if w is not visited → call helper on w

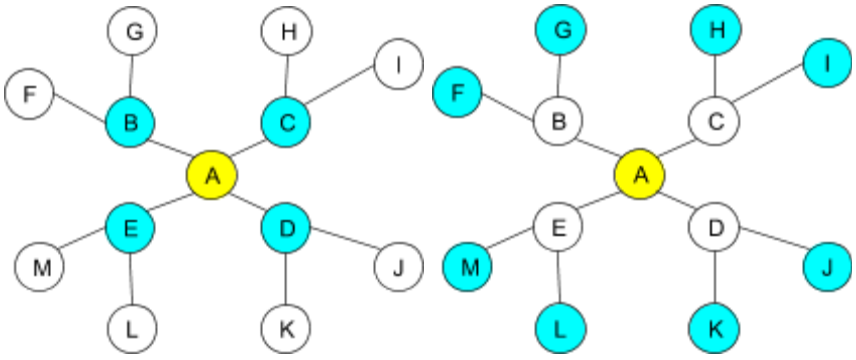
No Content - Wednesday

Friday

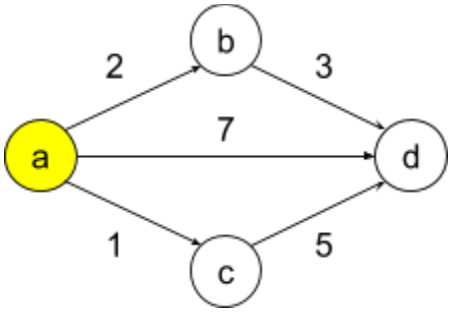
<u>Topics</u>

I. Shortest Path Algorithms
  A. BFS - Can be used to find shortest paths in UNweighted graphs

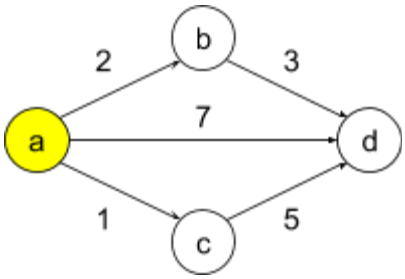
1. BFS visits everything one edge away, then two edges, etc., so the first time you encounter a vertex, the fewest number of edges possible have been traversed to reach it



2. BFS cannot find shortest paths in weighted graphs because the sum of the weights of several edges could be less than the weight of one edge
    a) Eg.

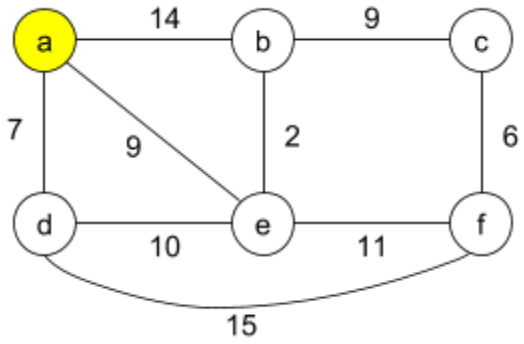


B. Dijkstra's Shortest Path Algorithm
    1. Premise - keep track of what we think is the cheapest total cost to get to each vertex from the start vertex
        a) for each iteration, finalize the smallest cost to get to one vertex
            (1) of the unfinalized vertices, we pick the one with the smallest total distance from the start
            (2) whenever we finalize a vertex path, we look at its neighbors to see if we've found a cheaper way to get to them
    2. Example (finding smallest by looking through map) - finalized



	A	B	C	D
Initial - A smallest	0	∞	∞	∞
Add paths from A: C smallest		2	1	7
Add paths from C: B smallest		2		6
Add paths from B: D smallest				5

3. Example (finding smallest by using a priority queue)



Priority Queue	Current Vertex	A	B	C	D	E	F
~~(a,0)~~	A	0	∞	∞	∞	∞	∞
(b,14), ~~*(d,7)*~~*, (e,9)*	D		14	∞	7	9	
(b,14), ~~(e,9)~~, *(f, 22)*	E		14	∞		9	22
(b,14), (f, 22), ~~*(b, 11)*~~*, (f, 20)*	B		11	∞			20
~~(b,14)~~, (f, 22), (f, 20), *(c, 20)*	B*			20			20
(f, 22), ~~(f, 20)~~, (c, 20)	F			20			20
(f, 22), ~~(c, 20)~~	C			20			

* if the priority queue gives a vertex you've already visited, you ignore it and move on

4. Stopping conditions
    a) We've finalized (visited) every vertex - stop even if the priority queue isn't empty
    b) The priority queue is empty - stop even if some vertices aren't finalized (usually means you have disconnected vertices)
5. Efficiency
    a) O((V+E) log (V))
        (1) log (V) comes from using a priority queue
        (2) V+E comes from looking at all the vertices and edges roughly once (BFS and DFS have O(V+E) run times)