2019-11-14 Finishing up DFS trees and building MSTs

Wednesday, November 13, 2019 6:06 PM

Announcement

- CS Club Hackathon fundraiser at Applebees 11/17 from 5-9PM!

Articulation Point Algorithm Efficiency

$O(V)$ - relative to # vertices

$O(E)$ - relative to # edges

$O(V)$ $|E| \le |V|^2$

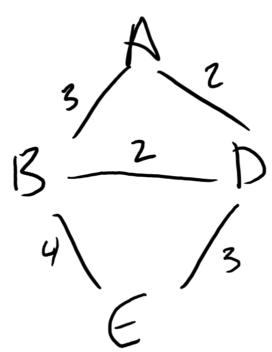
1. Having constructed a DFS tree w/ back edges, give each node in the tree an "ID" based on the order in which it was visited (root -> #1) $O(V)$
2. Next, find the lowest ID of the node that can be reached in the tree by taking zero or more forward (solid) edges, and **<u>up to one</u>** back edge (dotted line) $O(V^2)$
3. Express this as a fraction with step #1 numerator, step #2 denominator
   a. I.e. ID / LOW ID VALUE
4. A node is an articulation point
   a. if and only if it is a root and has more than two children OR
   b. When the node's direct child(ren) have a LOW VALUE >= its ID value

- If we invert our thinking and start with the bottom of the tree and work our way up, we only have to inspect each node once for step #2 of the above algorithm, thereby reducing the entire algorithm to O(V)

Minimum Spanning Trees

- An algorithm that determines the minimum cost edges required to maintain connectivity in a graph

Example Graph with Edge Weights



Depending on application domain, weights might represent

- Cost ($)
- Distance
- Time
- Throughput

MST finds the "cheapest" way to connect the entire graph,
but some routes may become slower (e.g. A->B in our example graph)

How might we represent edge weights programmatically?

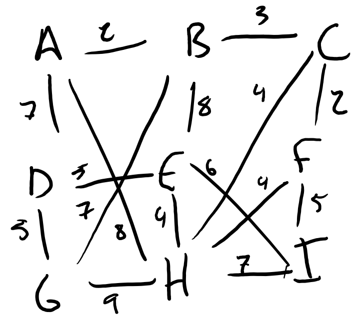
- Adjacency Matrix: Store edge weight in cell rather than always 1 or 0
- Edge list: store both edge weight and cost (good use of HT)
  - Key: Vertex*, Value: Edge Weight

Kruskal's Minimum Spanning Tree Algorithm

1. Put all edges into a min priority queue

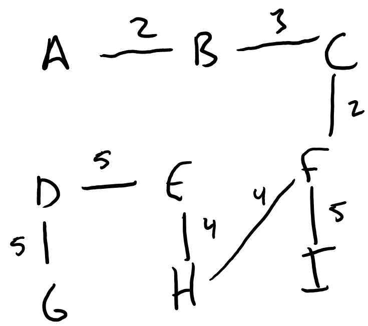
2. Pop off edge. See if edge connect a new node to the graph. If so, "accept" the edge Otherwise, reject.

Example



PQ pop order

1. ✓ AB (2)
2. ✓ CF (2)
3. ✓ BC (3)
4. ✓ HE (4)
5. ✓ HF (4)
6. ✗ HC (4)
7. ✓ DE (5)
8. ✓ DG (5)
9. ✓ FI (5) done
10. EI (6)
11. AD (7)
12. HI (7)
13. GB (7)
14. AH (8)
15. EB (8)
16. GH (9)



Considerations

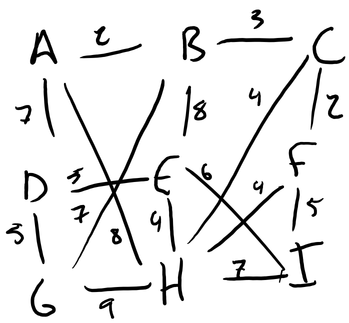
- How do we determine done-ness?
  - Once you have accepted V-1 edges, you're done
- How do you determine if two vertices are already connected?
  - Set operations are required
    - Basic set class can union and intersection is linear, However…
    - If all you're doing is union on disjoint sets, there exists a data structure called Disjoint Set that can perform unions in Log*N time (almost O(1)).
      - Log*(1) = Log(1)
      - Log*(2) = Log(Log(2))
      - Log*(3) = Log(Log(Log(3)))
- Analysis of Kruskal's Algorithm
  - Create PQ — $O(E)$
  - While not fully connected: $O(V)$
    - Pop off item from PQ add to graph if connects two disjoint items $O(\log E)$

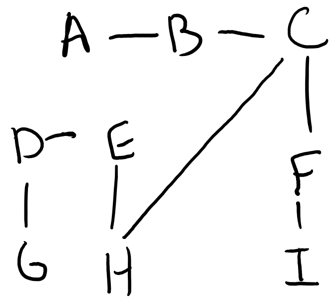
$O(V \log E)$

Prim's MST Algorithm

1. Pick some arbitrary starting vertex. Add all outgoing edges into a PQ
2. While graph is not fully connected:
   a. Pop off top edge. If vertex not seen before, accept vertex. Push all new edges from this accepted vertex into the PQ.

PQ /





PQ

- AB (2)
- AH (8)
- AD (7)
- BC (3)
- BE (8)
- BG (7)
- CF (2)
- CH (4)
- FI (5)
- FH (4)
- EH (4)
- HI (7)
- HG (9)
- DE (5)
- EI (6)