ECE 590.01
C++ Programming, Data structures, and Algorithms
Graphs III

Admin
• Reading
  • Chapter 9
  • Skipping 8 (optional reading if you want)
• Homework 4 out now
  • Due April 5
• Project
  • Should be making some solid progress here
  • Don’t wait until the last minute!

What have we been talking about?
• What did we talk about last time?

What have we been talking about?
• What did we talk about last time?
  • Graphs:
    • DFS
    • BFS
    • Dijkstra’s Shortest Path

Now: Minimal Spanning Tree
• Problem: Have many buildings which need power connected
  • Potential paths indicated with cost as edge weights
• Goal:
  • Connected tree
  • Minimal cost

MST: Algorithm 1—Prim’s
• Prim’s algorithm:
  • Pick a node
  • Build a tree out from it...
  • Add minimal cost edges out from current set of nodes
  • As long as they don’t form a cycle
MST: Algorithm 1—Prim’s
• Track a Priority Queue of edges for what to consider next

• Start with any node you want
• Add its edges to the PQ

• Repeat until all nodes are in the MST
  • Dequeue an edge from the PQ
    • If it makes a cycle, discard it
    • If not, add the edge + node to the MST
      • Then add the new node's addes to the PQ
MST: Algorithm 1—Prim’s

• Repeat until all nodes are in the MST
  • Dequeue an edge from the PQ
  • If it makes a cycle, discard it
  • If not, add the edge + node to the MST
  • Then add the new node’s address to the PQ
MST: Algorithm 1—Prim’s

• Repeat until all nodes are in the MST
  • Dequeue an edge from the PQ
    • If it makes a cycle, discard it
    • If not, add the edge + node to the MST
      • Then add the new node’s adds to the PQ

What is the running time?

Prim’s Running time

• It depends... on what?
Prim's Running time

- It depends… on what?
  - Representation of Graph
  - Adjacency Matrix
  - Adjacency List—let's assume this one.
  - PQ implementation
  - Binary Heap? Efficient—let's assume this one.
  - But fancy heaps can do better...

Prim's: Test for Cycle

- In this particular case, we can exploit the fact that we are building a tree to test for a cycle
  - Keep the Set of vertices already in our MST.
  - An edge creates a cycle iff the Set contains both ends of the edge already
  - We can implement the Set with a HT and this test (and update) in O(1) time

Prim's: slightly different formulation

- Slightly different approach: keep Vertices in the PQ instead of Edges
  - PQ ordered by shortest edge length from "tree so far" to that vertex
    - Add a vertex to the tree -> decrease key
    - "bubble up": O(lg N) time operation

Prim's running time

- O(V) "getAdjacencies" operations
  - We can implement this in O(1) time/operation
- O(E) "PQ.enqueue" operations
  - Each of these takes O(lg E) time in a binary heap
- O(E) "PQ.dequeue" operations
  - Also O(lg E) time
- O(E) "test For cycle" operations
  - ??
  - We didn't really pin down how to do this did we?
  - Add the edge in, do a DFS, see if we find a loop?
    - Seems slow… Can we do better?

Prim's running time

- O(V) "getAdjacencies" operations
  - We can implement this in O(1) time/operation
- O(E) "PQ.enqueue" operations
  - Each of these takes O(lg E) time in a binary heap
- O(E) "PQ.dequeue" operations
  - Also O(lg E) time
- O(E) "test For cycle" operations
  - O(1)

So that is O(V + E * lg E + E * lg E + E)
  - Equals: O(V + E * lg E)
  - Equals: O(E * lg E) / (Assuming E >= V)

MST: Algorithm 1.1—Prim's v2.0

- Start with all vertices in PQ
  - Distance to all but one is "infinity"
  - Distance to starting choice is 0
MST: Algorithm 1.1—Prim’s v2.0

- Start with all vertices in PQ
  - Distance to all but one is “infinity”
  - Distance to starting choice is 0
- Dequeue vertex
  - Update PQ edge weights
  - No need to test for cycle
MST: Algorithm 1.1—Prim’s v2.0

- Start with all vertices in PQ
- Distance to all but one is “infinity”
- Distance to starting choice is 0
- Dequeue vertex
- Update PQ edge weights
- No need to test for cycle

Prim’s running time

- O(V) “getAdjacencies” operations
  - We can implement this in O(1) time/operation
- O(V) “PQ.enqueue” operations
  - Each of these takes O(lg V) time in a binary heap
- O(V) “PQ.dequeue” operations
  - Also O(lg V) time
- O(E) “PQ.decreaseKey” operations
  - Each of these takes O(lg V) time

So now:

O(V + V * lg V + V * lg V + E * lg V)

Equals: O(V + E * lg V)  //Assuming E >= V

Difference in the two

- Difference between the two:
  - First: O(E * lg E)
  - Second: O(E * lg V)

  E >= V, so second is better right?
  - E ~= O(V^2)

Interestingly, this is practically true, but not theoretically:

O(E * lg E) = O(E * lg V) = O(E * 2 lg V) = O(E * lg V)

MST: Algorithm 2—Kruskal’s

- Kruskal’s algorithm:
  - Put all edges in PQ
  - Repeatedly dequeue an edge
    - If it makes a cycle, discard it
    - If not, add it to the tree

Kruskal’s algorithm:

- Put all edges in PQ
  - Repeatedly dequeue an edge
    - If it makes a cycle, discard it
    - If not, add it to the tree
Kruskal's algorithm:
- Put all edges in a PQ
- Repeatedly dequeue an edge
- If it makes a cycle, discard it
- If not, add it to the tree
MST: Algorithm 2—Kruskal’s

- Kruskal’s algorithm:
  - Put all edges in a PQ
  - Repeatedly dequeue an edge
    - If it makes a cycle, discard it
    - If not, add it to the tree

MST: Algorithm 2—Kruskal’s

- Kruskal’s algorithm:
  - Put all edges in a PQ
  - Repeatedly dequeue an edge
    - If it makes a cycle, discard it
    - If not, add it to the tree

As with Prim’s we could stop once we have all nodes
- Or just keep discarding edges until our PQ is empty

Note: may be different possible MSTs
- E.g., equal weights

TSP approximation with MST

- Recall: Traveling Salesperson Problem = NP complete
  - Can do a decent approximation of best solution with MST
    - If our graph obeys triangle inequality
      - Bounded to at most 2x optimal

Our graph does not
- Need: EB + BD >= ED  [and other problems]

Let’s see what happens if we change the edge weights
- Make an example that obeys the triangle inequality
TSP approximation with MST

• Step 1: Compute MST

Step 2: Do pre-order traversal
• G F C A B H E D

Step 3: Use that path
• Didn’t draw all of the edges here, so need to add in DG (29+10) to finish

Strongly Connected Components

• Strongly Connected Components
  • In a directed graph...
  • An SCC is comprised of nodes that can all reach each other

• H can get to everywhere...
  • But nowhere can get to H, so it’s in an SCC by itself
  • E B D can all get to each other
  • ACFG can’t get to E
  • ACFG can all get to each other
Before how, why?

- Why would we want to do this?
  - (Any application of graphs is fair game)

Graph as network/transport
- Can we move data/goods/etc from any point to any other?
- If not, where are the disconnects?

Compilers:
- Call graph:
  - Nodes = functions
  - Edges = “calls”
  - SCC is a mutually recursive group of functions...
  - Collapse into one node to make CG a DAG, do inter-procedural opt
  - Analyze recursive group as one entity

Strongly Connected Components

Now, let’s think about how we are going to do this

SCC: Inefficient approach

- Inefficient approach:
  - For each node N
    - Do a DFS from N
      - Build up the set of nodes you can reach from N
      - Put that set into a Map (Nodes -> Set of reachable nodes)
  - For each node N
    - Ans = {} 
    - For each X in Map[N]
      - If Map[X] contains S
        - Ans = Ans + X

What is the running time?

Running time

- V DFSes
  - Each O(V + E)
  - And doing Set insert O(1)
  - And Map insert: O(1)
- V more
  - Map lookup: O(1)
  - Iteration across a set of nodes: O(V)
  - Check set containment: O(1)
  - Add to set: O(1)
- O(V * (V+E) + V * V) = O(V * (V+E))

SCC: Efficient approach

- Keep a map from Nodes to
  - Index (increase for every node we visit)
  - Lowest index adjacent to it
  - Actually lowest “lowest index” adjacent to it
  - Note: will be lowest index it can reach
SCC: Efficient approach

• Keep a map from Nodes to
  • Index (increase for every node we visit)
  • Lowest index adjacent to it
    • Actually lowest "lowest index" adjacent to it
      • Note: will be lowest index it can reach
• Do a DFS
  • Maintain a separate Stack which we will pop from when we find an SCC
  • We “find” an SCC when our DFS gets back to a node where its index = its lowest index
SCC: Efficient approach

- Recursion Stack:
  - B E D A G
- Index:
  - 6

SCC: Efficient approach

- Recursion Stack:
  - B E D A G
- Index:
  - 5

SCC: Efficient approach

- Recursion Stack:
  - B E D A G C
- Index:
  - 6

SCC: Efficient approach

- Recursion Stack:
  - B E D A G C //already visited
- Index:
  - 6
SCC: Efficient approach

Recursion Stack:
1. B E D A G F
2. Index:
3. 7

ECE 590.01 (Hilton): Graphs III
SCC: Efficient approach

• Recursion Stack:
  • B E D A
• Index:
  • 7

When we get back to "A" we notice that $\text{Idx} = \text{Low}$
  • This means A was the first node we saw in some SCC
  • Everything with $\text{low} = 4$ is in the SCC
  • Pop Stack until we get to A, put each popped item into the SCC

At B, we find another SCC
  • $\text{Low} = \text{Idx} = 1$
  • A's low is higher (4 vs 1) -> no update

Our DFS ended...
  • But we have not assigned all nodes to SCCs
  • So start another DFS at a node with no SCC

Note that at D
  • $\text{Low} = 1$
  • A's low is higher (4 vs 1) -> no update

New DFS starts at H
SCC: Efficient approach

- Recursion Stack:
  - H B //already visited and assigned SCC
- Index:
  - 8

B is both visited and assigned an SCC
Not in the stack
So we don’t update our Low from it

After exploring H’s adjacencies, we find its in an SCC by itself

What is the runtime?

DFS done
All nodes in SCCs
Done!

Runtime for SCC

- Runtime is basically that of DFS O(V+E)
  - Map + index operations are O(1)
  - Need to search stack: could be O(V) for each node
  - …or could just keep Set of nodes in Stack for O(1) check

That’s all for graphs

- We’ve done a bunch of graphs
  - Many things in lecture
  - Plus max flow in recitation
- That is all we are going to do in this class
  - Learn more in algorithms classes
  - Or other classes that use such things
    - E.g., graph coloring register allocator in compilers