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
• 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
MST: Algorithm 1—Prim’s

- Track a Priority Queue of edges for what to consider next
- Start with any node you want
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
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 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 adders 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 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 adds 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 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 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 adds 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 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 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 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 added to the PQ
MST: Algorithm 1—Prim’s

- All nodes are in MST, so can stop
  - Could also continue to dequeue and discard edges until PQ is empty
MST: Algorithm 1—Prim’s

- All nodes are in MST, so can stop
  - Could also continue to dequeue and discard edges until PQ is empty

What is the running time?
MST: Algorithm 1—Prim’s

- All nodes are in MST, so can stop
  - Could also continue to dequeue and discard edges until PQ is empty

What is the running time?
It depends…
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 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: 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 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
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
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

**Prio Queue**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Priority</th>
<th>Adjacent Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>7</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>14</td>
<td>A</td>
</tr>
<tr>
<td>H</td>
<td>22</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>34</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
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

Prio Queue

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Priority</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>34</td>
<td>A</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
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
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
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

What is the running time now?
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:
- \(O(V + V \times lg V + V \times lg V + E \times lg V)\)
- Equals: \(O((V + E) \times lg (V))\)
- Equals: \(O(E \times lg (V))\) \(\text{ //Assuming } E \geq V\)
Difference in the two

- Difference between the two:
  - First: $O(E \times \lg E)$
  - Second: $O(E \times \lg V)$

- $E \geq V$, so second is better right?
  - $E \sim O(V^2)$

- Interestingly, this is practically true, but not theoretically:
  - $O(E \times \lg E) = O(E \times \lg V^2) = O(E \times 2 \lg V) = O(E \times \lg V)$
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
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

![Graph with labels and prioritized queue]

<table>
<thead>
<tr>
<th>Prio Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
</tr>
<tr>
<td>BH</td>
</tr>
<tr>
<td>EH</td>
</tr>
<tr>
<td>AD</td>
</tr>
<tr>
<td>DC</td>
</tr>
<tr>
<td>CF</td>
</tr>
<tr>
<td>AH</td>
</tr>
<tr>
<td>ED</td>
</tr>
<tr>
<td>CG</td>
</tr>
<tr>
<td>DF</td>
</tr>
<tr>
<td>AG</td>
</tr>
</tbody>
</table>
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
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
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

- 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
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: $EB + BD \geq ED$ [and other problems]
TSP approximation with MST

- 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
TSP approximation with MST

- Step 1: Compute MST
- Step 2: Do pre-order traversal
  - G F C A B H E D
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
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
Strongly Connected Components

- 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)
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
- Index:
  - 1
SCC: Efficient approach

- Recursion Stack:
  - B E
- Index:
  - 2

<table>
<thead>
<tr>
<th>Stk</th>
<th>Idx</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SCC: Efficient approach

- Recursion Stack:
  - B E D
- Index:
  - 3
SCC: Efficient approach

- Recursion Stack:
  - B E D B //visited already
- Index:
  - 3
SCC: Efficient approach

- Recursion Stack:
  - B E D
- Index:
  - 3
SCC: Efficient approach

- Recursion Stack:
  - B E D A
- Index:
  - 4
SCC: Efficient approach

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

<table>
<thead>
<tr>
<th>Stk</th>
<th>Idx</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>H</td>
<td>G</td>
<td>5</td>
</tr>
</tbody>
</table>
SCC: Efficient approach

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

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

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

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

<table>
<thead>
<tr>
<th>Stk</th>
<th>Idx</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>E</td>
<td>5</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SCC: Efficient approach

- Recursion Stack:
  - B E D A G F
- Index:
  - 7
SCC: Efficient approach

- Recursion Stack:
  - B E D A G F C //already visited
- Index:
  - 7
SCC: Efficient approach

- Recursion Stack:
  - B E D A G F
- Index:
  - 7
SCC: Efficient approach

- Recursion Stack:
  - B E D A G F G //already visited
- Index:
  - 7

<table>
<thead>
<tr>
<th>Stk</th>
<th>Idx</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>7</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>5</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SCC: Efficient approach

- Recursion Stack:
  - B E D A G F
- Index:
  - 7
SCC: Efficient approach

- Recursion Stack:
  - B E D A G
- Index:
  - 7
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 low = 4 is in the SCC
    - Pop Stack until we get to A, put each popped item into the SCC
SCC: Efficient approach

- Recursion Stack:
  - B E D
- Index:
  - 7

- Note that at D
  - Low is 1
  - A’s low is higher (4 vs 1) -> no update
SCC: Efficient approach

- Recursion Stack:
  - B E
- Index:
  - 7
SCC: Efficient approach

- Recursion Stack:
  - B
- Index:
  - 7

- At B, we find another SCC
  - Low = Idx = 1
  - Pop stack until we find B
SCC: Efficient approach

- Recursion Stack:
  - 
- Index:
  - 7

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

- Recursion Stack:
  - H
- Index:
  - 8
- New DFS starts at H

<table>
<thead>
<tr>
<th>Stk</th>
<th>Idx</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
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
SCC: Efficient approach

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

Same for E
SCC: Efficient approach

- Recursion Stack:
  - H
- Index:
  - 8

After exploring H’s adjacencies, we find it is in an SCC by itself
SCC: Efficient approach

- Recursion Stack:

- Index:
  - 8

DFS done
All nodes in SCCs
Done!

What is the runtime?
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