Course Notes for

CS 1501
Algorithm Implementation

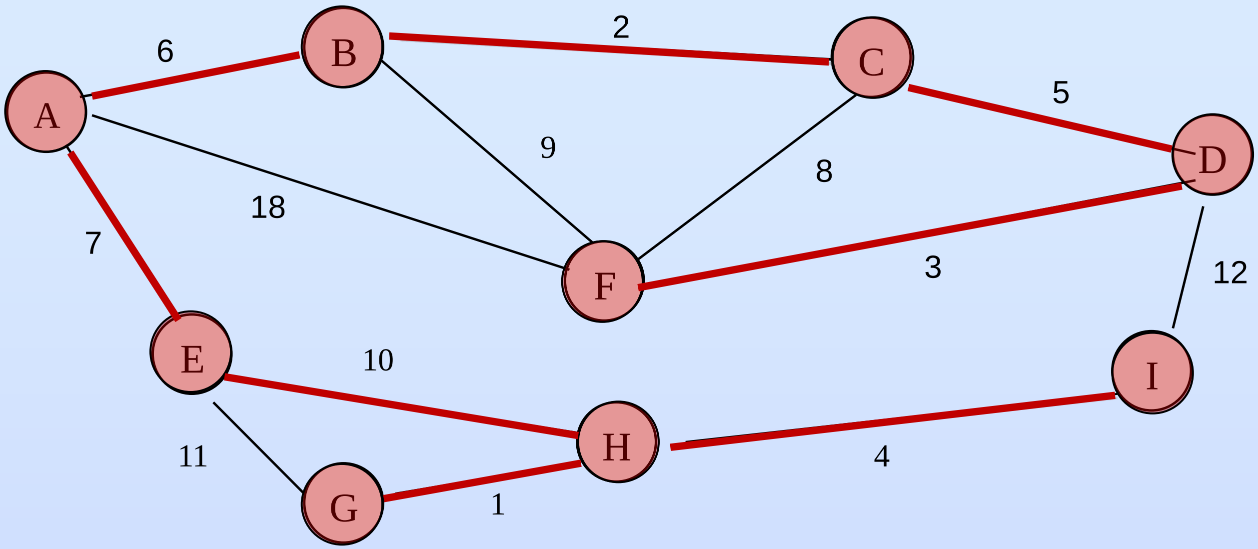
By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Prim's Algorithm
  - Idea of Prim's is very simple:
    - Let **T** be the current tree, and **T'** be all vertices and edges not in the current tree
    - Initialize **T** to the starting vertex (**T'** is everything else)
    - while (#vertices in **T** < v)
      - Find the smallest edge connecting a vertex in **T** to a vertex in **T'**
      - Add the edge and vertex to **T** (remove them from **T'**)
  - As is often the case, implementation is not as simple as the idea

Original graph showing edge weights

Prim's Algorithm



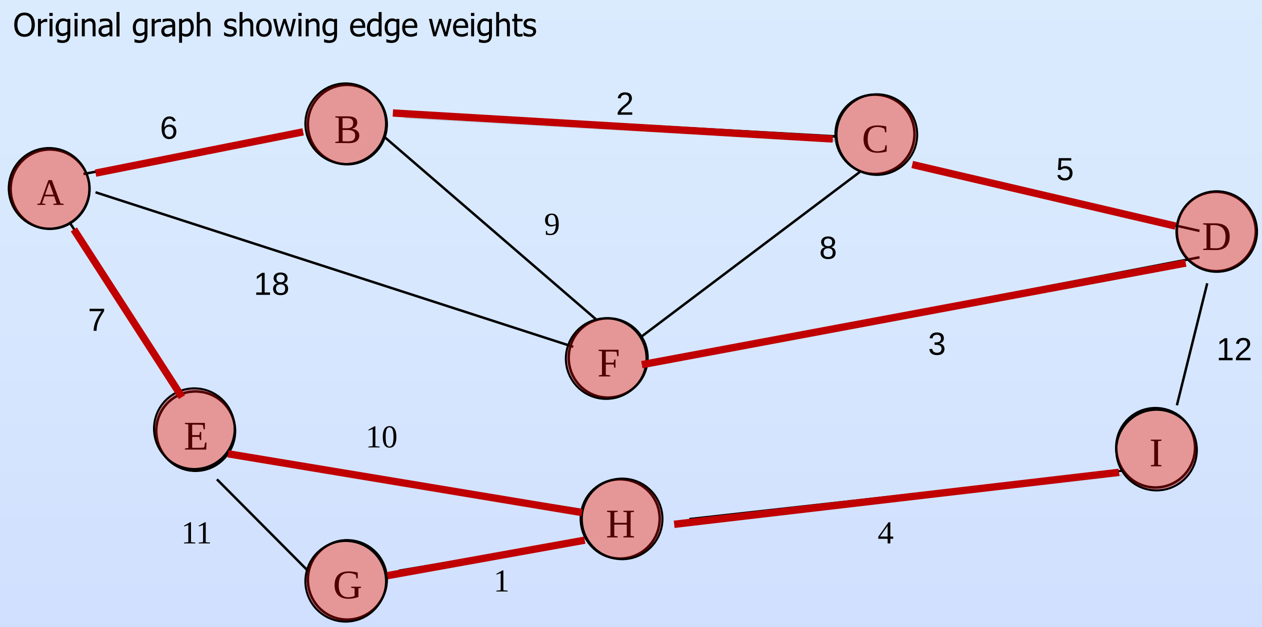
Tree shown in RED

- Initially tree consists of starting vertex only
- Each iteration we add one vertex and one edge to the tree
- Note that the smallest overall edge may not be added in a given iteration
  - Only edges that connect a tree vertex to a non-tree vertex can be added

- Naïve implementation:
  - At each step look at all possible edges that could be added at that step, choosing smallest
  - Let's look at the worst case for this impl:
    - Complete graph (all edges are present)
      - Step 1: (v-1) possible edges (from start vertex)
      - Step 2: 2(v-2) possible edges (first two vertices each have (v-1) edges, but shared edge between them is not considered)
      - Step 3: 3(v-3) possible edges (same idea as above)
      - …
    - Total: Sum (i = 1 to v-1) of i(v-i)
      - This evaluates to Theta(v^3)

- Better implementation:
  - Can we better organize the edges to reduce the run-time?
    - Yes – use a Heap Priority Queue (PQ)
    - As we visit a vertex, put its neighbor edges into a PQ
      - But only if one vertex of the edge is in the tree and the other is not
    - Then remove the edge with min value from the PQ and repeat
    - This way the PQ delivers the best edge "candidate" each time in a more efficient way
    - See wgraph.pdf and LazyPrimMSTTrace.java

Prim's Algorithm



Tree shown in RED

Priority Queue

AB	AE	AF	BC	BF	CD	CF	DF	DI	EG	EH	HG	HI
6	7	18	2	9	5	8	3	12	11	10	1	4

- Note that we only put edges into the PQ which would connect the tree to the "non-tree"
- However, as the tree forms, some edges become "internal" and would no longer be part of the tree (ex: CF).

- Run-time?
  - In the worst case each edge in the graph is added to the PQ at some point
  - In the worst case all e edges must be removed
    - Could stop earlier if we know tree has already been completed
  - We know a Heap PQ gives time
    - Theta(lgN)

    For both add() and deleteMin() for a PQ of size N
  - Since we have E edges, in the worst case **Lazy Prim** gives us
    - **Theta(ElgE)**
  - Definitely an improvement over v^3

Eager Prim MST

- Can we do even better?
  - Lazy Prim is putting up to e edges into the PQ. Is this necessary?
  - No. Instead, let's just keep track of the current "best" edge for each vertex in T'
    - i.e. the minimum edge that would connect that vertex to T
  - Then at each step we do the following:
    - Look at all of the "best" edges for the vertices in T'
    - Add the overall best edge and vertex to T
    - Update the "best" edges for the vertices remaining in T', considering now edges from latest added vertex
  - This is the idea of Eager Prim MST

Eager Prim MST

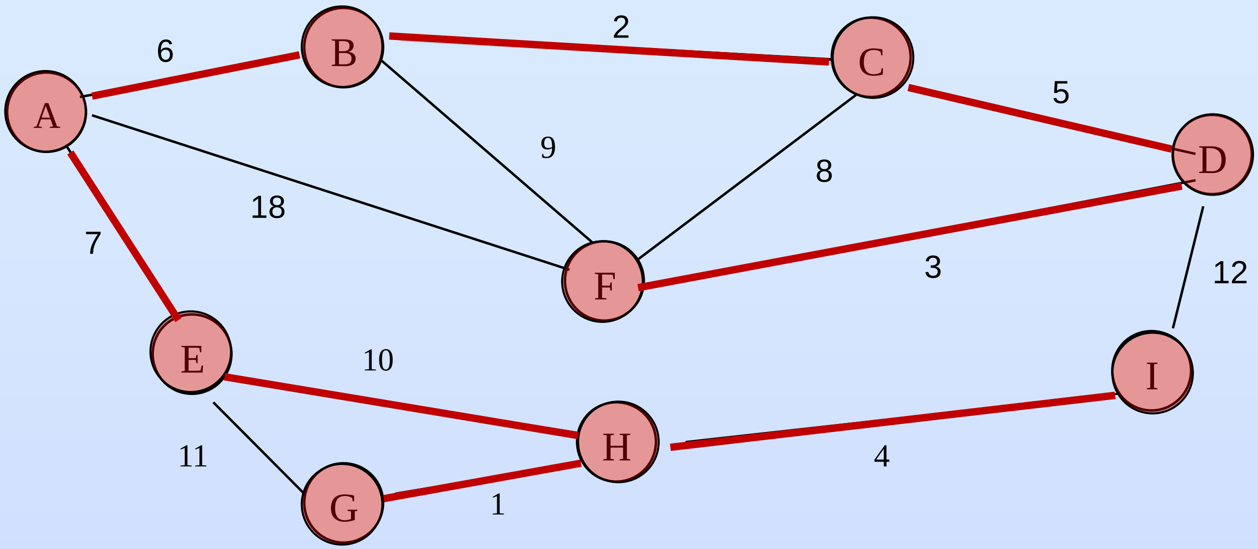
- Now (Vertex, Weight) pairs are stored in the PQ rather than just edges
  - Weight represents the "best" edge candidate to put that vertex into the tree
- At each step the overall best vertex (i.e. the one with the smallest edge) is removed from the PQ
  - Then its adjacency list is traversed, and the remaining vertices in the PQ are updated if necessary
- Algorithm continues until the PQ is empty
- We still consider all edges, but now the PQ entries are vertices rather than edges
  - Thus the run-time for Eager Prim is **Theta(ElgV)**

See next slide for trace

- Also see PrimMSTTrace.java

Prim's Algorithm

Original graph showing edge weights



Tree shown in RED

Indexable Priority Queue

B	E	F	C	F	D	F	F	I	G	H	G	I
AB	AE	AF	BC	BF	CD	CF	DF	DI	EG	EH	HG	HI
6	7	18	2	9	5	8	3	12	11	10	1	4

- Now we never have more than v entries in our PQ – one entry per vertex
- When a "better" edges is found for a vertex, we replace it in the PQ

- Lazy Prim:
  - Time: ElgE
  - Additional space: E (for PQ)
  - PQ requirements: MinPQ (Heap)
- Eager Prim:
  - Time: ElgV
  - Additional space: V (for PQ)
  - PQ requirements: Indexable MinPQ
    - We must be able to update the values in the PQ
    - But to update a value we must first find it
      - Discuss

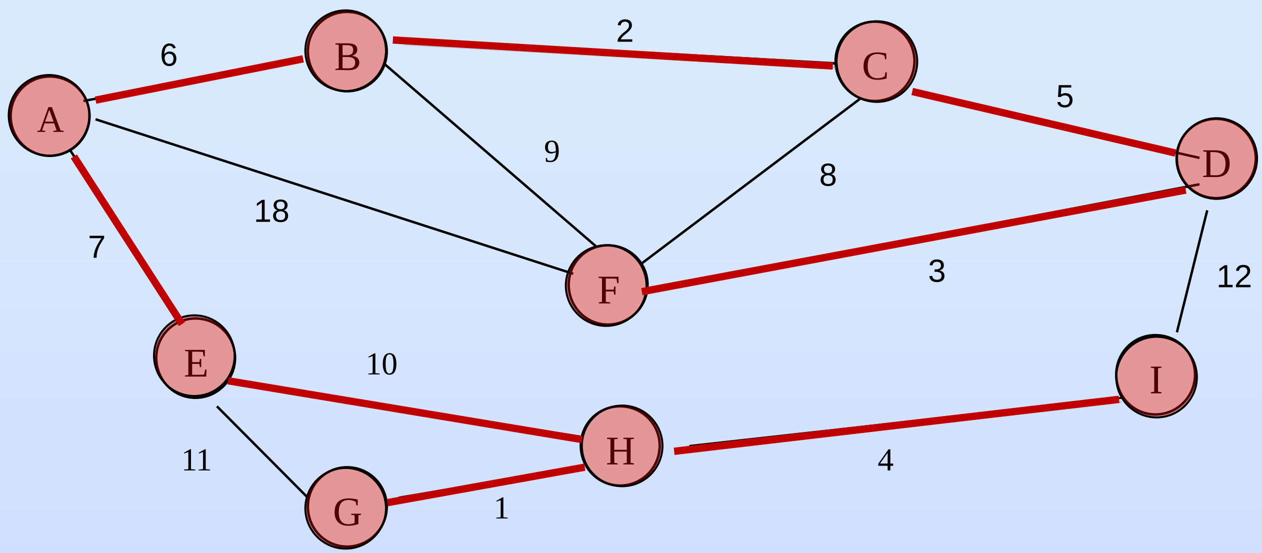
- What about an adjacency matrix?
  - We don't need a heap for the PQ in this case
  - Consider process of looking at the neighbors of a given vertex, v_i
    - It takes Theta(v) to traverse the row of the matrix to find the neighbors of v_i
    - In the same loop, we can easily add code to find the next best vertex to add to the tree
      - > Keep two arrays: best_edge, parent
      - > As we visit neighbors of v_i update the best_edge and parent arrays, keeping track of the overall minimum edge
      - > This adds a few extra statements within the loop but no extra asymptotic time
  - Thus, the overall run-time is the same as for BFS – Theta(v^2)

- How does this compare to Eager Prim on adjacency list?
  - elgv adj. list vs v^2 adj. matrix
  - Depending upon how e relates to v, one may be preferable

	e = v	e = vlgv	e ~= v^2
Adj. List	vlgv	vlg^2v	v^2lgv
Adj. Matrix	v^2	v^2	v^2

- Briefly, one other famous MST algorithm: Kruskal
  - Idea:
    - Insert all edges into a PQ
    - Remove next edge and add to MST if it does not create a cycle
  - Rather than building MST from a single source (as Prim does), Kruskal adds edges potentially throughout the graph
    - Eventually they connect into a single MST (assuming the graph is connected)

Kruskal's MST Algorithm



Priority Queue

AB 6	AE 7	AF 18	BC 2	BF 9	CD 5	CF 8	DF 3	DI 12	EG 11	EH 10	HG 1	HI 4

- Run-time:
  - PQ of edges: ElgE
  - Testing for cycles:
    - Union/Find data structure (if interested, see Section 1.5 of Sedgewick)
    - Find operation tests for cycles
      - May have to do up to E of these
    - Union operation adds the new edge
      - Will have to do V-1 of these
    - Each takes lgE (weighted quick union) or better (see Section 1.5)
      - So overall this does not increase the asymptotic run-time of the algorithm
  - Total **Theta(ElgE)**