CS/COE 1501

www.cs.pitt.edu/~lipschultz/cs1501/

Weighted Graphs

Last time, we said spatial layouts of graphs were irrelevant

- We define graphs as sets of nodes and edges
- However, we'll certainly want to be able to reason about bandwidth, distance, capacity, etc. of the real world things our graph represents
    - Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network
    - Having a road between two cities that is a 1 lane country road is very different from having a 4 lane highway
    - If two airports are 2000 miles apart, the number of flights going in and out will be drastically different from airports 100 miles apart

We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix:
    - Instead of 1, store the edge weight for all edges that exist
  - Adjacency list:
    - Add a field to list nodes to store the weight
- How do weights affect finding spanning trees/shortest paths?
  - The weighted variants of these problems are called finding the *minimum spanning tree* and the *weighted shortest path*

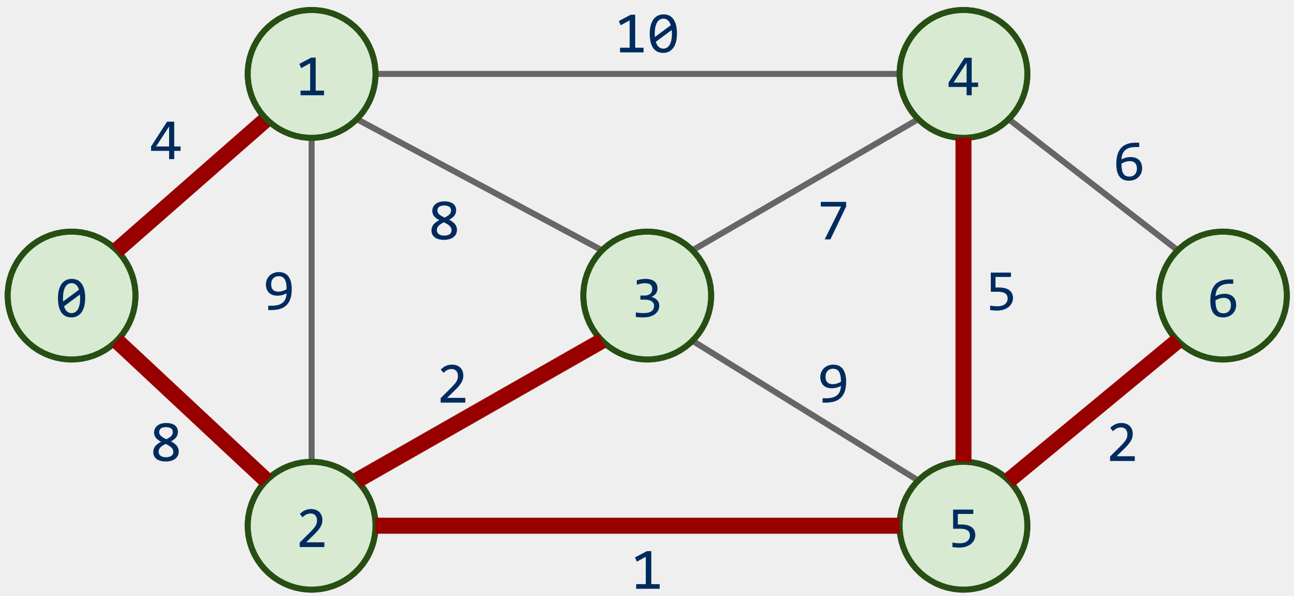
Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees
- MST is the spanning tree that has the minimum sum of the weights of its edges

Prim's algorithm

- Initialize T to the starting vertex
- Let T' be all vertices and edges not in T
  - So the entire graph minus the starting vertex
- while there are vertices not in T:
  - Find minimum edge in T' that connects to a vertex in T
  - Add the edge with its vertex in T' to T
    - Also remove them from T'

Prim’s algorithm



Implementing Prim's

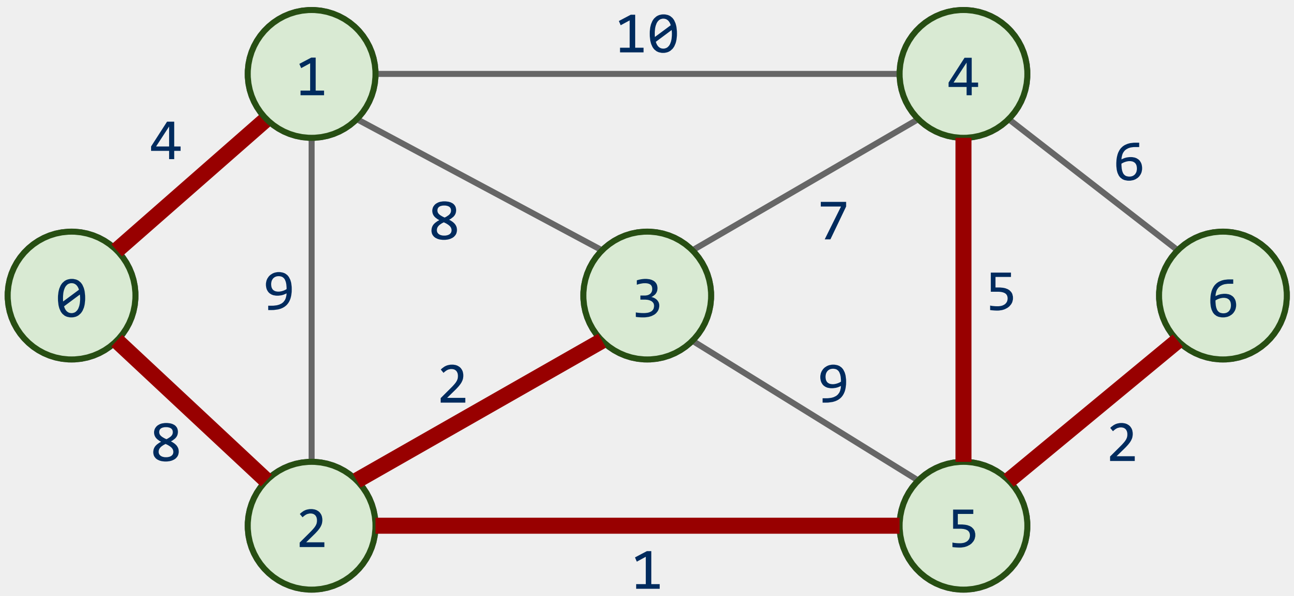
- BRUTE FORCE:
  - At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - v - 1 possible edges
  - Next iteration:
    - 2(v - 2) possibilities
      - Each node in T shared v-1 edges with other nodes, but the edge they shared with each other is in T, not T'
  - Next:
    - 3(v - 3) possibilities
  - …
- Runtime:
  - $\Sigma_{i = 1 \text{ to } v} (i * (v - i))$
    - Evaluates to …?

Utilizing our representations of graphs

- Let's assume we use an adjacency matrix:
  - Takes Θ(v) to check the neighbors of a given vertex
    - For every node we add to T, we’ll need to check all of its neighbors to check for edges to add to the MST next
      - During each neighbor check, maintain a parent and best_edge list

```
while num_vertices(T) < v:
    new = find_min(T, best_edge)
    T.append(new)
    for j = 0 to v:
        if M[new, j] && j ∉ T && M[new][j] < best_edge[j]:
            parent[j] = new
            best_edge[j] = M[new][j]
```

Prim's algorithm



	0	1	2	3	4	5	6
Parent:	--	0	0	2	5	2	5
Best Edge:	0	4	8	2	5	1	2

Runtime of this implementation of Prim's

- v vertices will be added to the MST
- So we do the following v times:
  - Search through the best_edge array to find the next addition to the MST
    - $\Theta(v)$
  - Search through the neighbors of the next vertex to adjust the parent and best edge arrays as needed
    - $\Theta(v)$
- So we do v * 2 * $\Theta(v)$ work
  - $\Theta(v^2)$

Can we improve on this?

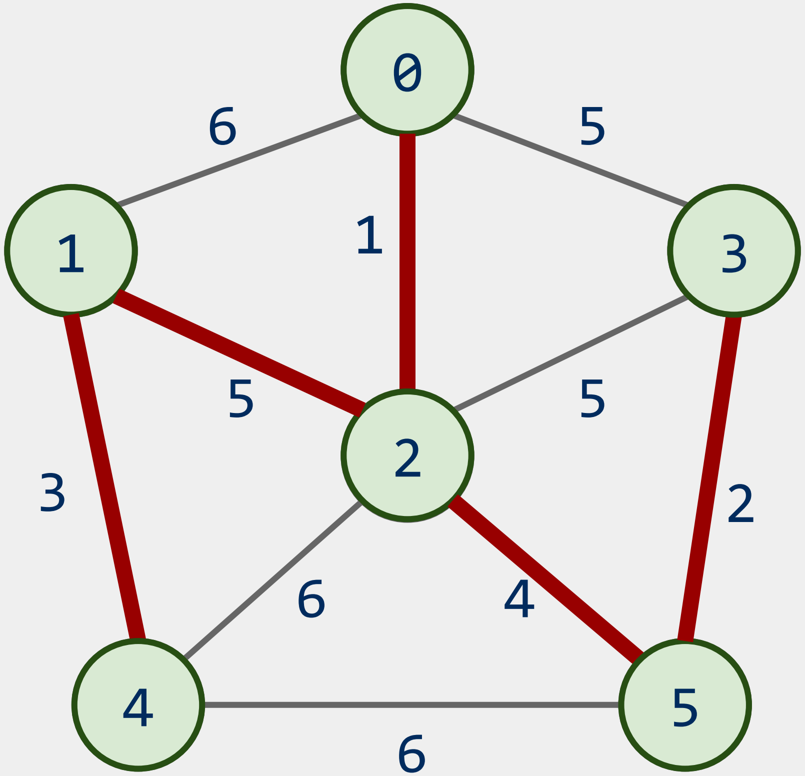
- Would using an adjacency list be any better?
    - How would we change the pseudocode?

```
while num_vertices(T) < v:

    new = find_min(T, best_edge)

    T.append(new)

    for j = 0 to v:

        if M[new, j] && j ∉ T && M[new][j] < best_edge[j]:

            parent[j] = new

            best_edge[j] = M[new][j]
```

What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in $\Theta(\lg n)$
    - Also $\Theta(\lg n)$ to add to the priority queue
- What does our algorithm look like now?
  - Visit a vertex
  - Add edges coming out of it to a PQ
  - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

Runtime using a priority queue

- Have to insert all e edges into the priority queue
  - In the worst case, we'll also have to remove all e edges
- So we have:
  - $e * \Theta(\lg e) + e * \Theta(\lg e)$
  - $= \Theta(2 * e \lg e)$
  - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Do we really need to maintain e items in the PQ?

- I suppose we could not be so lazy
- Just like with the adjacency matrix implementation, we only need the best edge for each vertex
  - PQ will need to be indexable
- This is the idea of *eager Prim's*
  - Runtime is $\Theta(e \lg v)$

Comparison of Prim's implementations

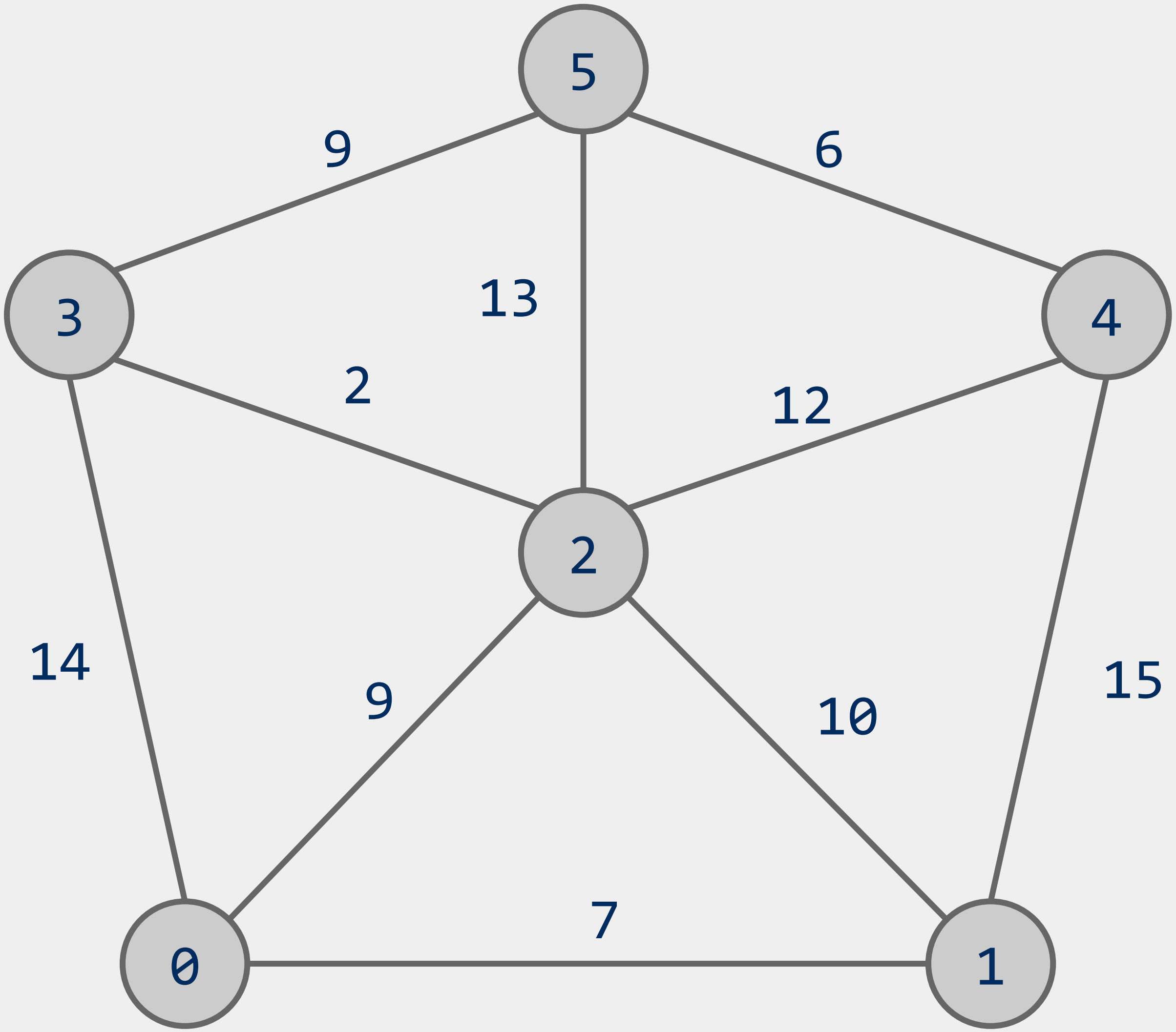
- Adjacency matrix Prim's
  - Runtime: $\Theta(v^2)$
  - Space: $\Theta(v)$
- Lazy Prim's
  - Runtime: $\Theta(e \lg e)$
  - Space: $\Theta(e)$
  - Requires a PQ
- Eager Prim's
  - Runtime: $\Theta(e \lg v)$
  - Space: $\Theta(v)$
  - Requires an indexable PQ

How do these compare?

Weighted shortest path

- Dijkstra's algorithm:
  - Set a distance value of MAX_INT for all nodes but start
  - Set cur = start
  - While destination is not visited:
    - For each unvisited neighbor of cur:
      - Compute tentative distance from start to the unvisited neighbor through cur
      - Update any vertices for which a lesser distance is computed
    - Mark cur as visited
    - Let cur be the unvisited node with the smallest tentative distance from start

Dijkstra's example



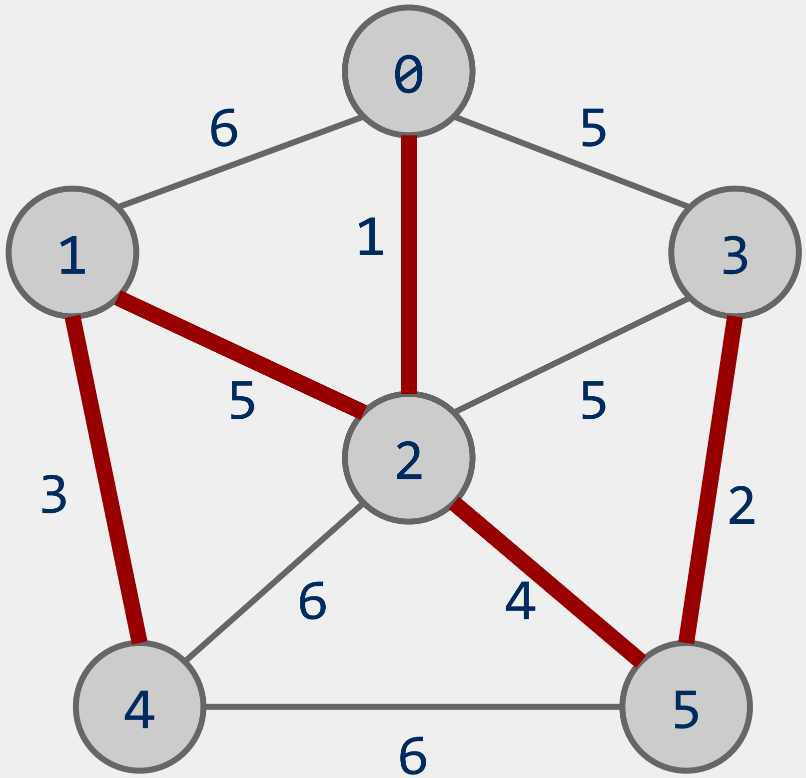
Analysis of Dijkstra's algorithm

- How to implement?
  - Best path/parent array?
    - Runtime?
  - PQ?
    - Turns out to be very similar to Eager Prims
      - Storing paths instead of edges
    - Runtime?

Back to MSTs: Another MST algorithm

- Kruskal's MST:
  - Insert all edges into a PQ
  - Grab the min edge from the PQ that does not create a cycle in the MST
  - Remove it from the PQ and add it to the MST

Kruskal's example



PQ:

1: (0, 2)

2: (3, 5)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (1, 2)

6: (0, 1)

6: (2, 4)

6: (4, 5)

Kruskal’s runtime

- Instead of building up the MST starting from a single node, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?