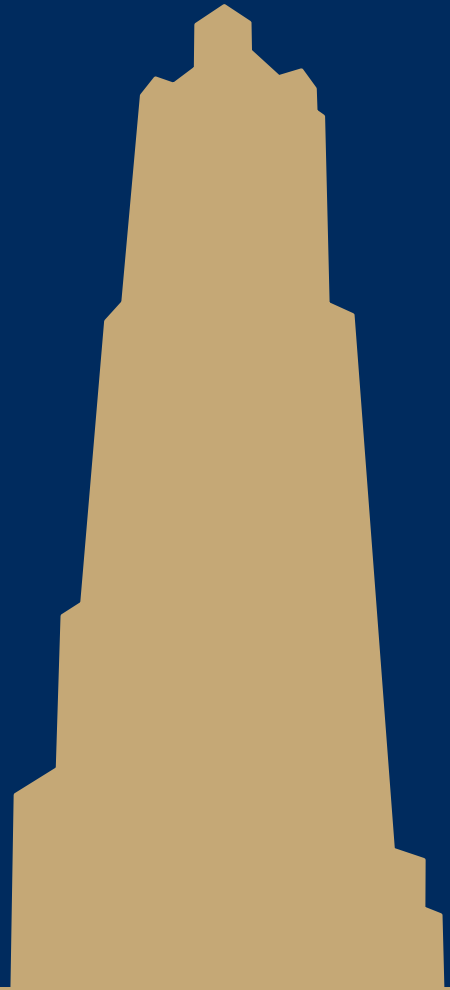


# CS/COE 1501

[www.cs.pitt.edu/~nlf4/cs1501/](http://www.cs.pitt.edu/~nlf4/cs1501/)

## Weighted Graphs



## Last time, we said spatial layouts of graphs were irrelevant

- We define graphs as sets of vertices 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 between them will be drastically different from airports 200 miles apart

# We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix?
  - Adjacency list?
  - Do we need a whole new graph representation?
- 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*

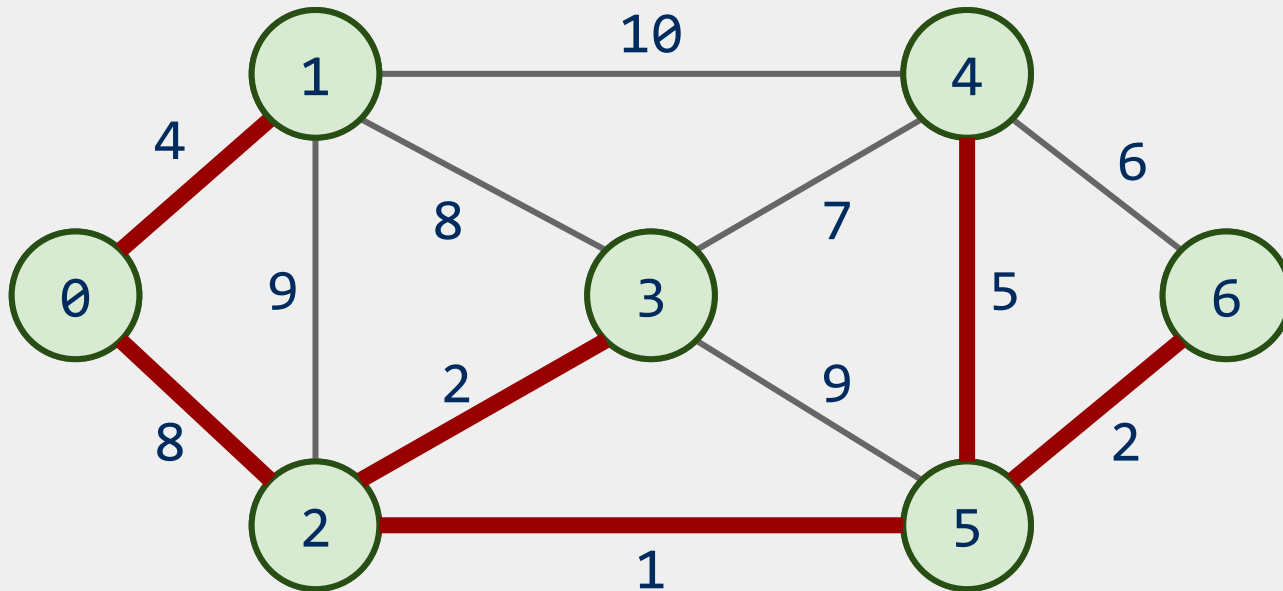
# Minimum spanning trees (MST)

- 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 contain the starting vertex
  - $T$  will eventually become the MST
- While there are vertices not in  $T$ :
  - Find minimum edge weight edge that connects a vertex in  $T$  to a vertex not yet in  $T$
  - Add the edge with its vertex to  $T$

# Prim's algorithm



# Runtime of Prim's

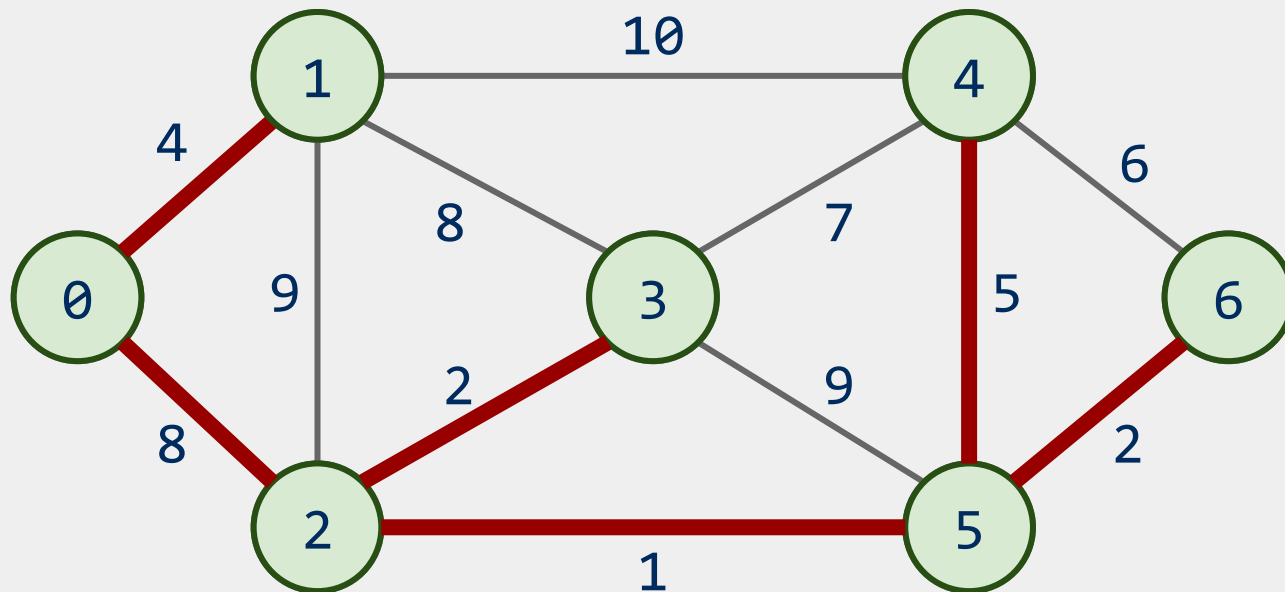
- At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - $v - 1$  possible edges
  - Next iteration:
    - $2(v - 2)$  possibilities
      - Each vertex in  $T$  shared  $v-1$  edges with other vertices, but the edges they shared with each other already in  $T$
  - Next:
    - $3(v - 3)$  possibilities
  - ...
- Runtime:
  - $\sum_{i=1 \text{ to } v} (i * (v - i))$ 
    - Evaluates to  $\Theta(v^3)$

# Do we need to look through all remaining edges?

- No! We only need to consider the *best* edge for possible for each vertex!



# 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

# OK, so what's our runtime?

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

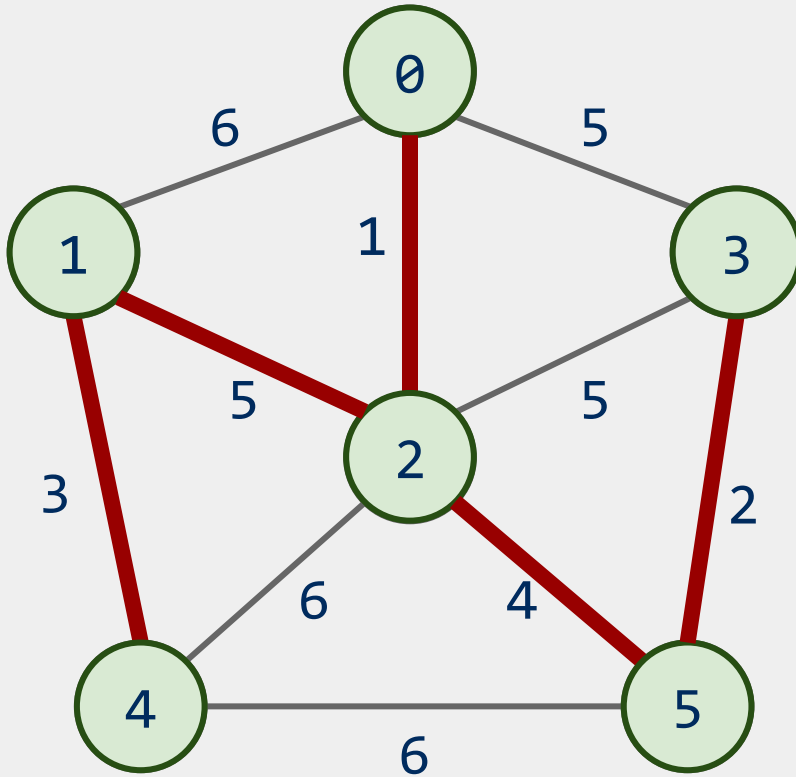


```
while num_vertices(T) < v:
    new = find_min(best_edge, T)
    T.add_vertex(new)
    for j = 0 to v:
        if M[new, j] && j ∉ T && M[new][j] < best_edge[j]:
            parent[j] = new
            best_edge[j] = M[new][j]
```

# 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:

1: (0, 2)

2: (5, 3)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (2, 1)

6: (0, 1)

6: (2, 4)

6: (5, 4)

# 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?

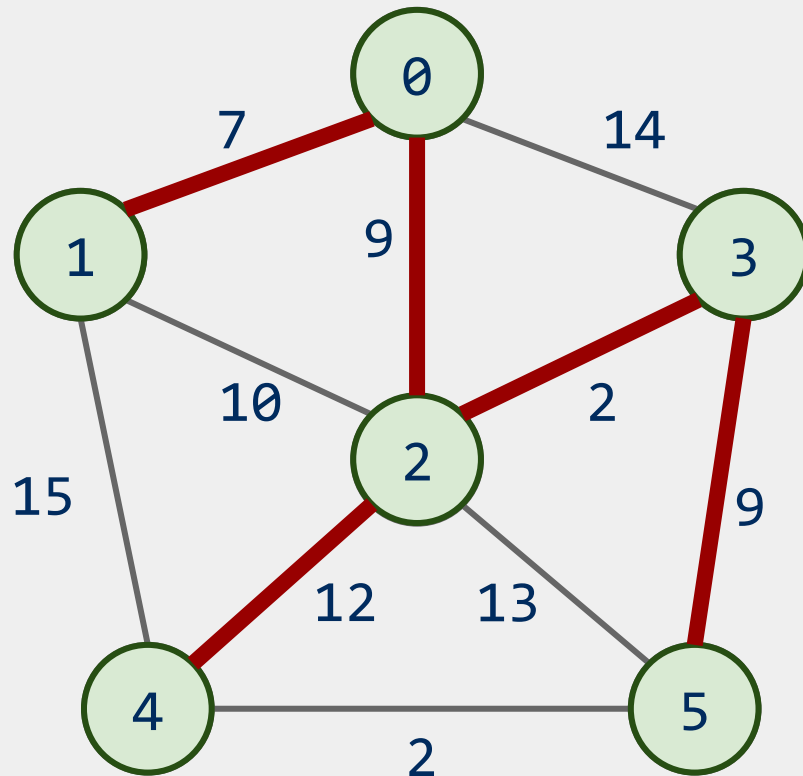


# Weighted shortest path

- Dijkstra's algorithm:
  - Set a distance value of MAX\_INT for all vertices 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 vertex with the smallest tentative distance from start





# Dijkstra's example



	Distance	Via
0	0	--
1	7	0
2	9	0
3	11	2
4	21	2
5	20	3

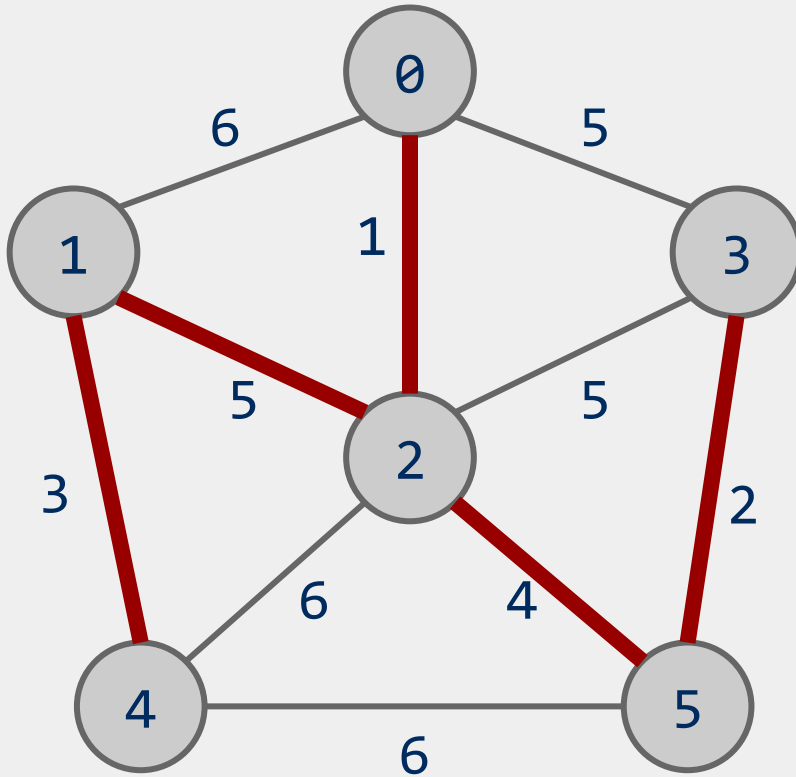
# Analysis of Dijkstra's algorithm

- 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 vertex, we build it up using edges all over the graph 
- How do we efficiently implement cycle detection?