CSE 417 Algorithms

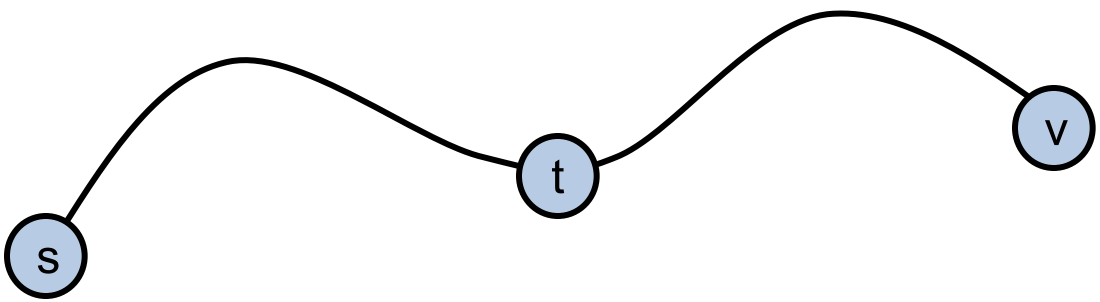
Winter 2020
Lecture 11
Dijkstra's algorithm

Dijkstra's algorithm

"In 1956 I did two important things, I got my degree and we had the festive opening of the ARMAC. **We had to have a demonstration**... For a demonstration for noncomputing people you have to have a problem statement that non-mathematicians can understand; they even have to understand the answer. So I designed a program that would find the shortest route between two cities in the Netherlands"

Image: http://cs-exhibitions.uni-klu.ac.at/index.php?id=29
Quote: https://dl.acm.org/doi/pdf/10.1145/1787234.1787249

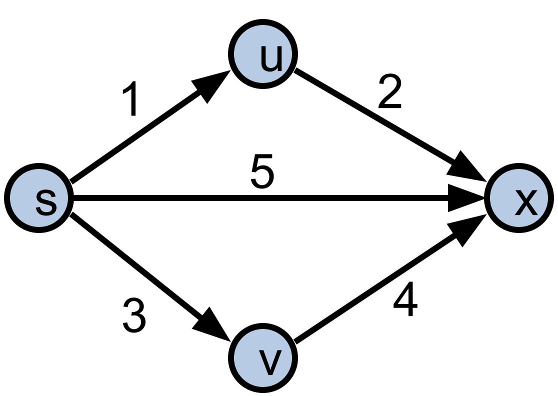
- If P is a shortest path from s to v, and if t is on the path P, the segment from s to t is a shortest path between s and t

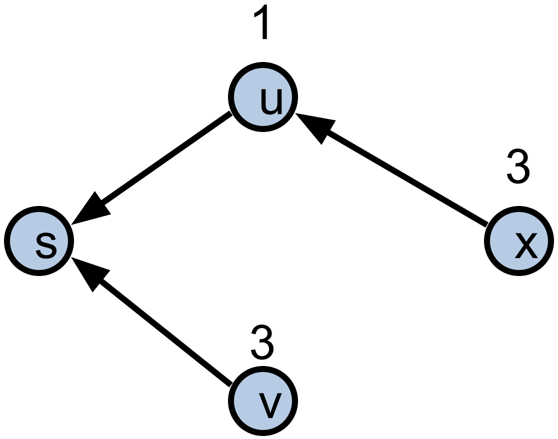


- WHY?

Single Source Shortest Path Problem

- Given a graph and a start vertex s
  - Determine distance of every vertex from s
  - Identify shortest paths to each vertex
    - Express concisely as a "shortest paths tree"
    - Each vertex has a pointer to a predecessor on shortest path





Assume all edges have non-negative cost

Dijkstra's Algorithm

S = { }; d[s] = 0; d[v] = infinity for v != s

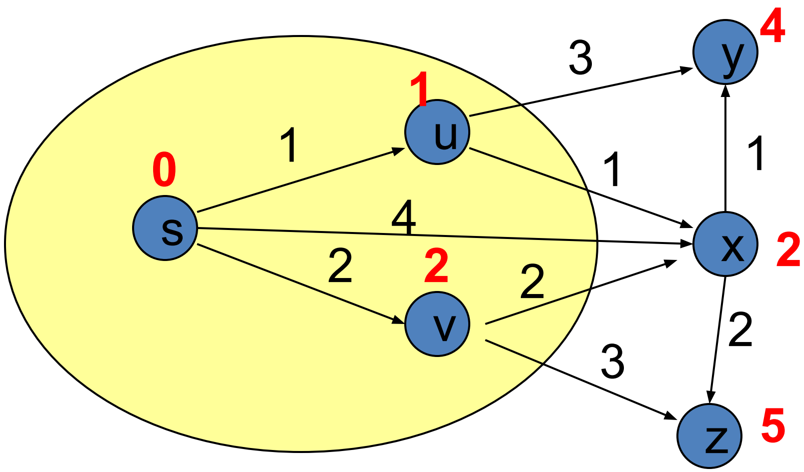
While S != V

Choose v in V-S with minimum d[v]

Add v to S

For each w in the neighborhood of v

d[w] = min(d[w], d[v] + c(v, w))



Assume all edges have non-negative cost

Dijkstra's Algorithm

S = { }; d[s] = 0; d[v] = infinity for v != s

While S != V

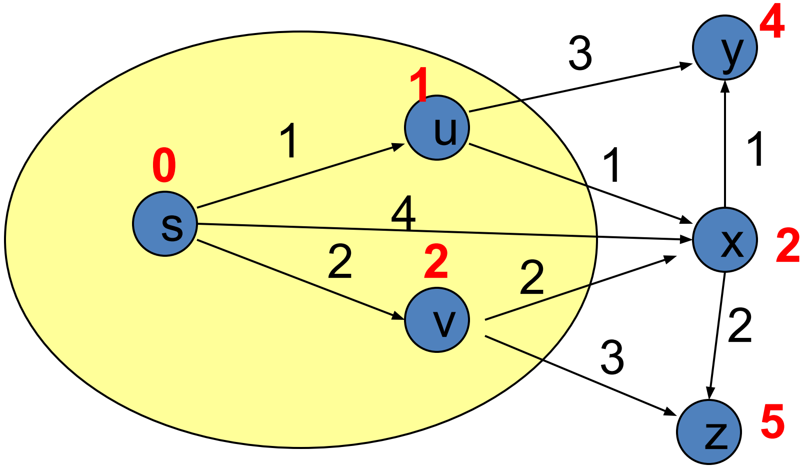
    Choose v in V-S with minimum d[v]

    Add v to S

    For each w in the neighborhood of v

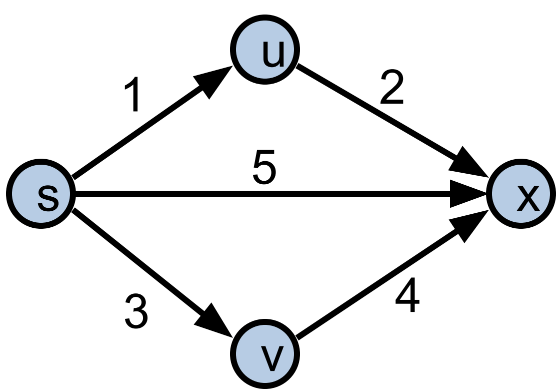
        d[w] = min(d[w], d[v] + c(v, w))

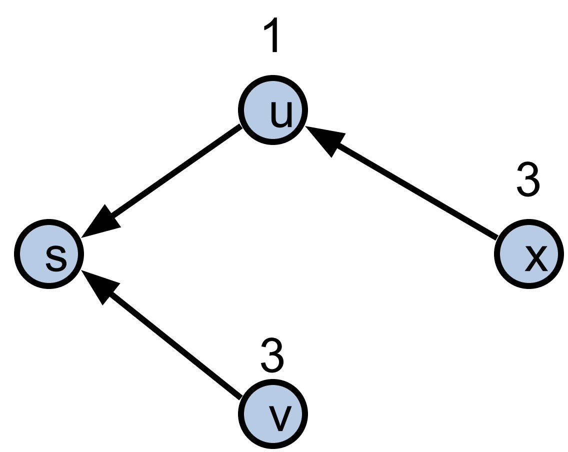
Something is missing from this pseudo-code!



Single Source Shortest Path Problem

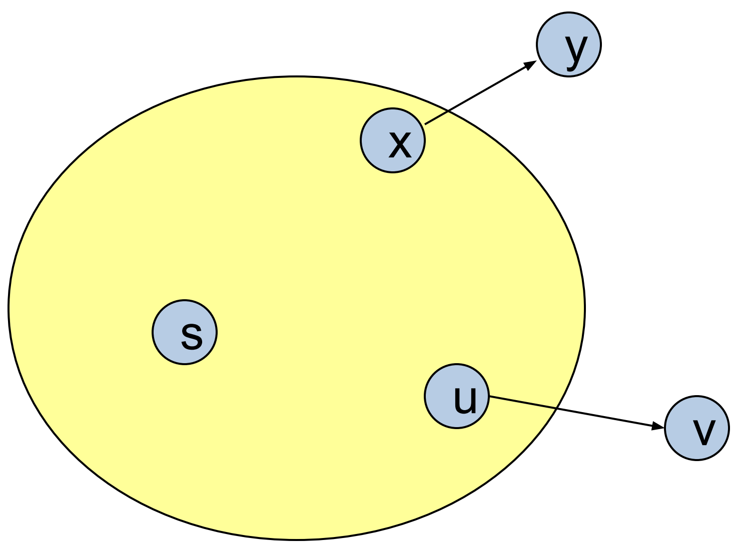
- Given a graph and a start vertex s
  - Determine distance of every vertex from s
  - Identify shortest paths to each vertex
    - Express concisely as a “shortest paths tree”
    - Each vertex has a pointer to a predecessor on shortest path





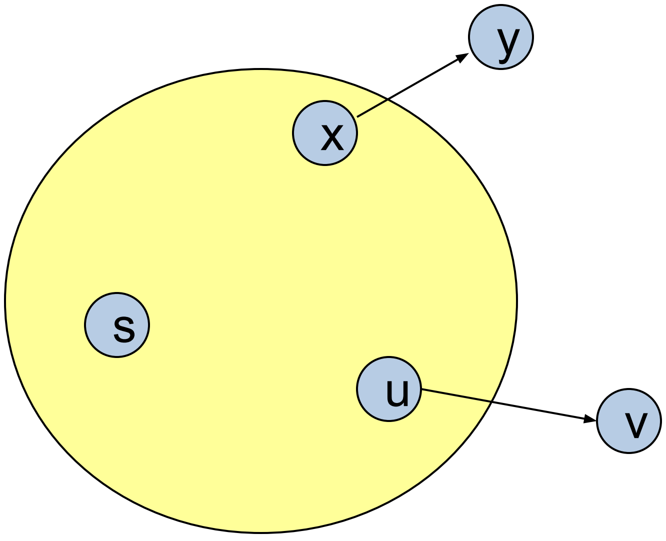
Correctness Proof

- Elements in S have the correct label
- Key to proof: when v is added to S, it has the correct distance label.



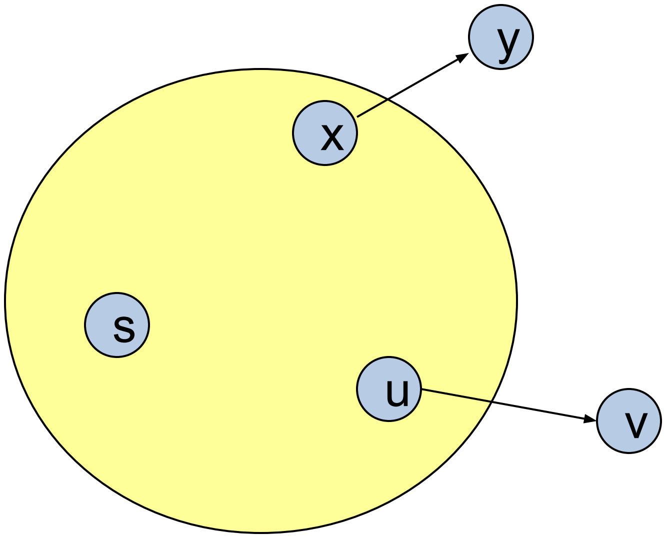
Proof

- Let v be a vertex in V-S with minimum d[v]
- Let P_v be a path of length d[v], with an edge (u,v)
- Let P be some other path to v. Suppose P first leaves S on the edge (x, y)
  - $P = P_{sx} + c(x,y) + P_{yv}$
  - $Len(P_{sx}) + c(x,y) >= d[y]$
  - $Len(P_{yv}) >= 0$
  - **Len(P) >=** d[y] + 0 **>= d[v]**



Proof

- Let v be a vertex in V-S with minimum d[v]
- Let P_v be a path of length d[v], with an edge (u,v)
- Let P be some other path to v. Suppose P first leaves S on the edge (x, y)
  - $P = P_{sx} + c(x,y) + P_{yv}$
  - $Len(P_{sx}) + c(x,y) >= d[y]$
  - $Len(P_{yv}) >= 0$
  - $Len(P) >= d[y] + 0 >= d[v]$



Notice: this is another exchange argument

Edge costs are assumed to be non-negative

Dijkstra's Algorithm Implementation and Runtime

S = { }; d[s] = 0; d[v] = infinity for v != s

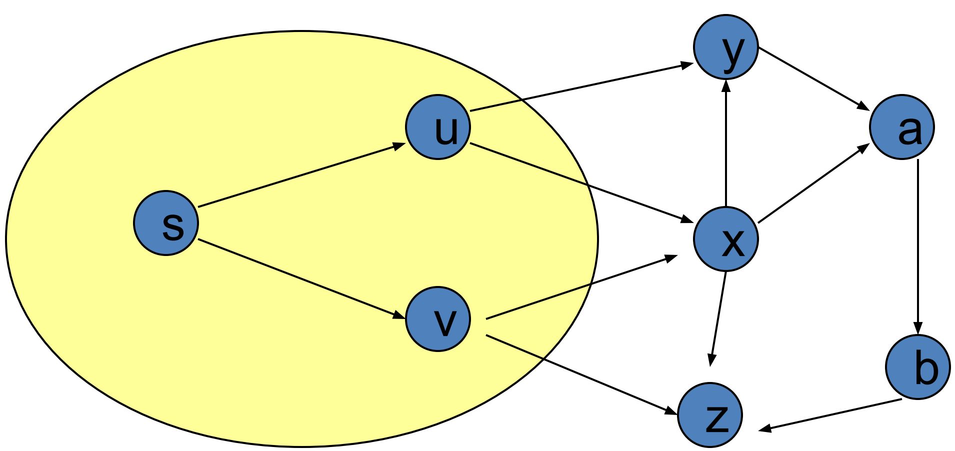
While S != V

Choose v in V-S with minimum d[v]

Add v to S

For each w in the neighborhood of v

d[w] = min(d[w], d[v] + c(v, w))



HEAP OPERATIONS

n Extract Mins

m Heap Updates

Run Time

- Basic Heap Implementation
  - O(log n) extract min and update key
  - O((m + n) log n) run time
- Fancy data structures: Fibonacci Heaps
  - O(m + n log n)
- Dense graphs
  - $O(n^2)$

What about Noam's solution

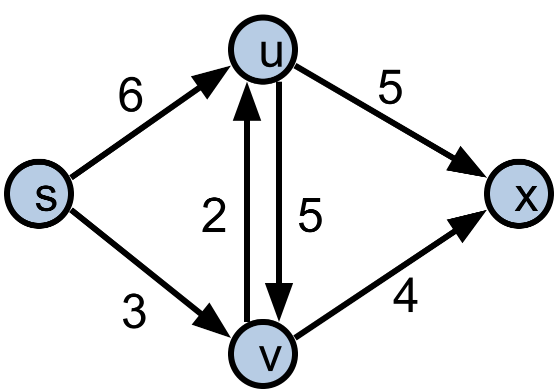
- Runtime of BFS is O(m+n)
- So if the sum of the edge weights is W, the runtime of the "dummy node" algorithm is

  O(W+n)

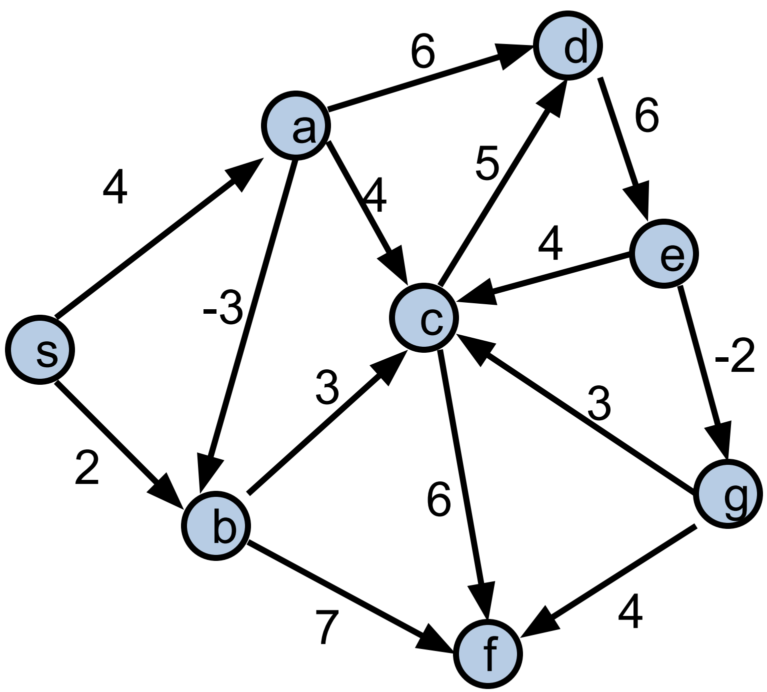
This assumes the graph has integer weights

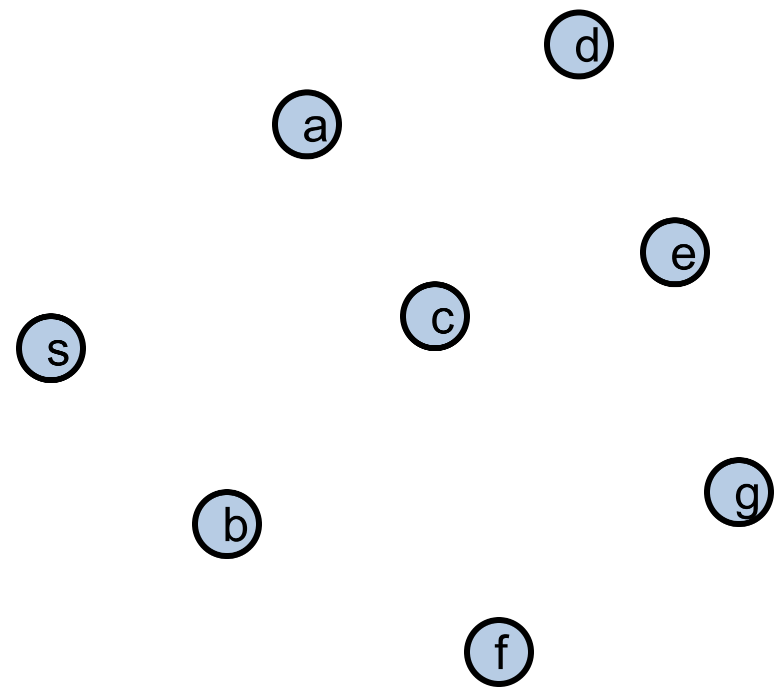
Bottleneck Shortest Path

- Define the bottleneck distance for a path to be the maximum cost edge along the path



Compute the bottleneck shortest paths





How do you adapt Dijkstra's algorithm to handle bottleneck distances

- Does the correctness proof still apply?

Dijkstra's Algorithm for Bottleneck Shortest Paths

S = { }; d[s] = 0; d[v] = infinity for v != s

While S != V

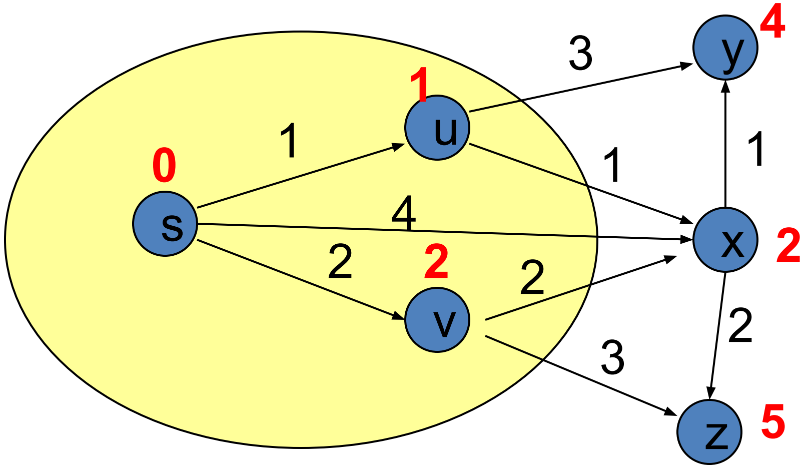
Choose v in V-S with minimum d[v]

Add v to S

For each w in the neighborhood of v

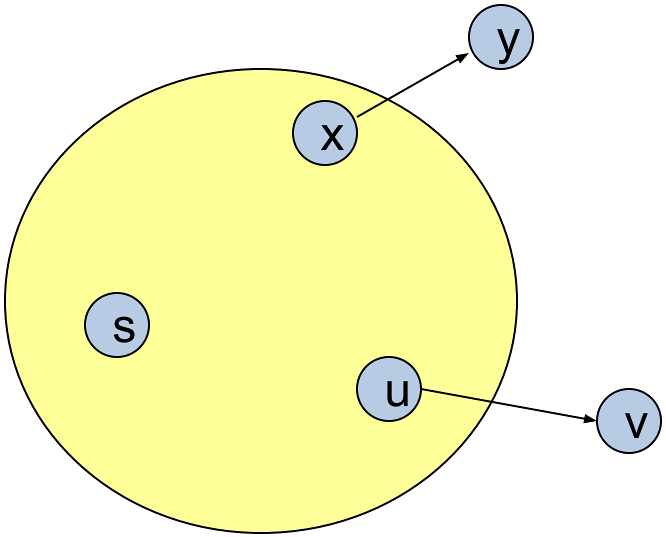
~~d[w] = min(d[w], **d[v] + c(v, w)**)~~

d[w] = min(d[w], **max(d[v], c(v, w))**)



Proof

- Let v be a vertex in V-S with minimum d[v]
- Let P_v be a path of length d[v], with an edge (u,v)
- Let P be some other path to v. Suppose P first leaves S on the edge (x, y)
  - $P = P_{sx} + c(x,y) + P_{yv}$
  - $Len(P_{sx}) + c(x,y) >= d[y]$
  - $Len(P_{yv}) >= 0$
  - **Len(P) >=** d[y] + 0 **>= d[v]**

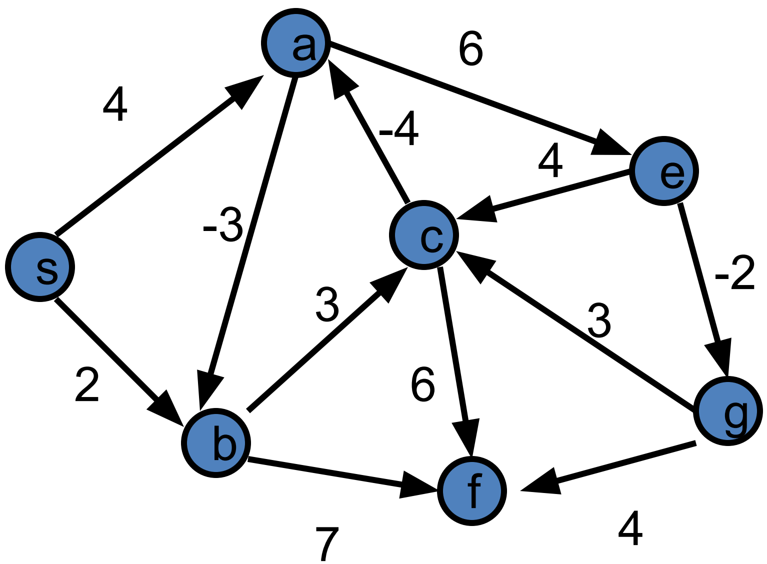


Negative Cost Edges

- Draw a small example with a negative cost edge and show that Dijkstra’s algorithm fails on this example

Shortest Paths

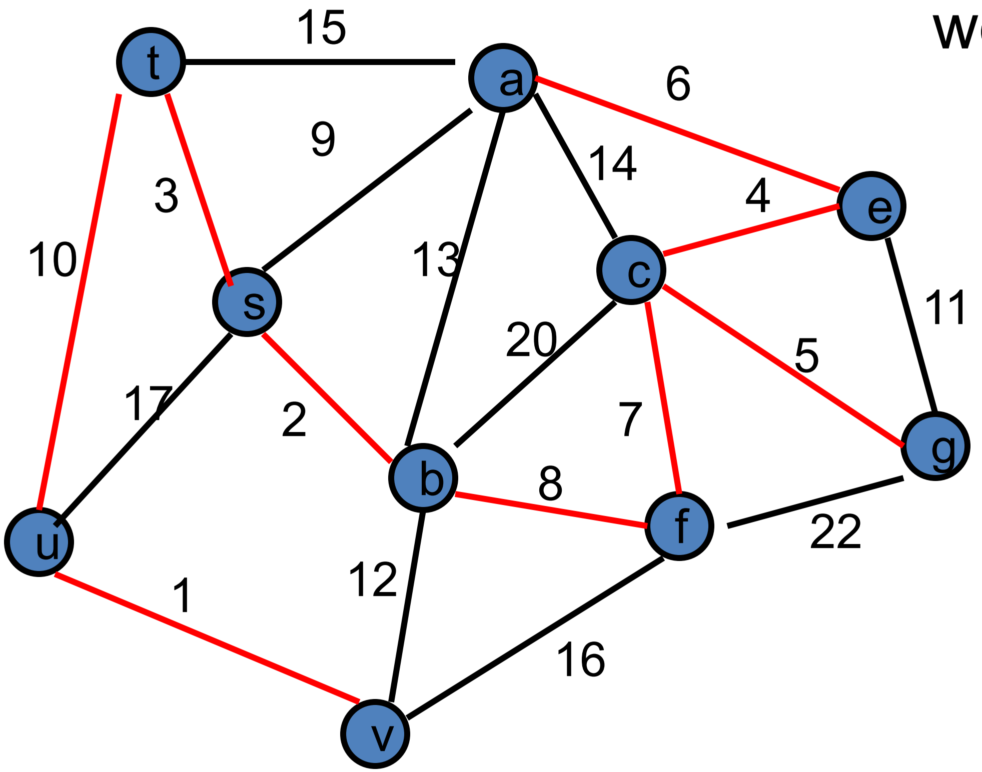
- Negative Cost Edges
  - Dijkstra's algorithm assumes positive cost edges
  - For some applications, negative cost edges make sense
  - Shortest path not well defined if a graph has a negative cost cycle



Minimum Spanning Tree

- Introduce Problem
- Demonstrate three different greedy algorithms
- Provide proofs that the algorithms work

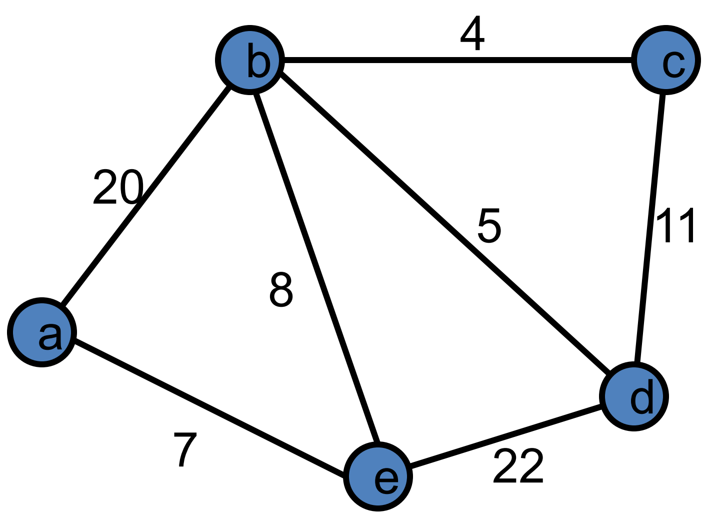
Minimum Spanning Tree



Undirected Graph G=(V,E) with edge weights

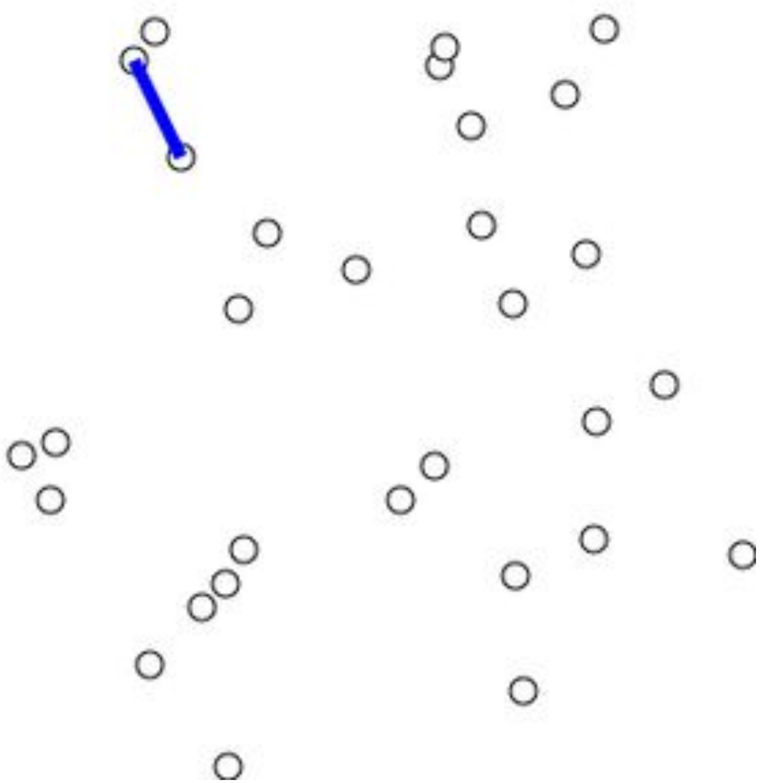
Greedy Algorithms for Minimum Spanning Tree

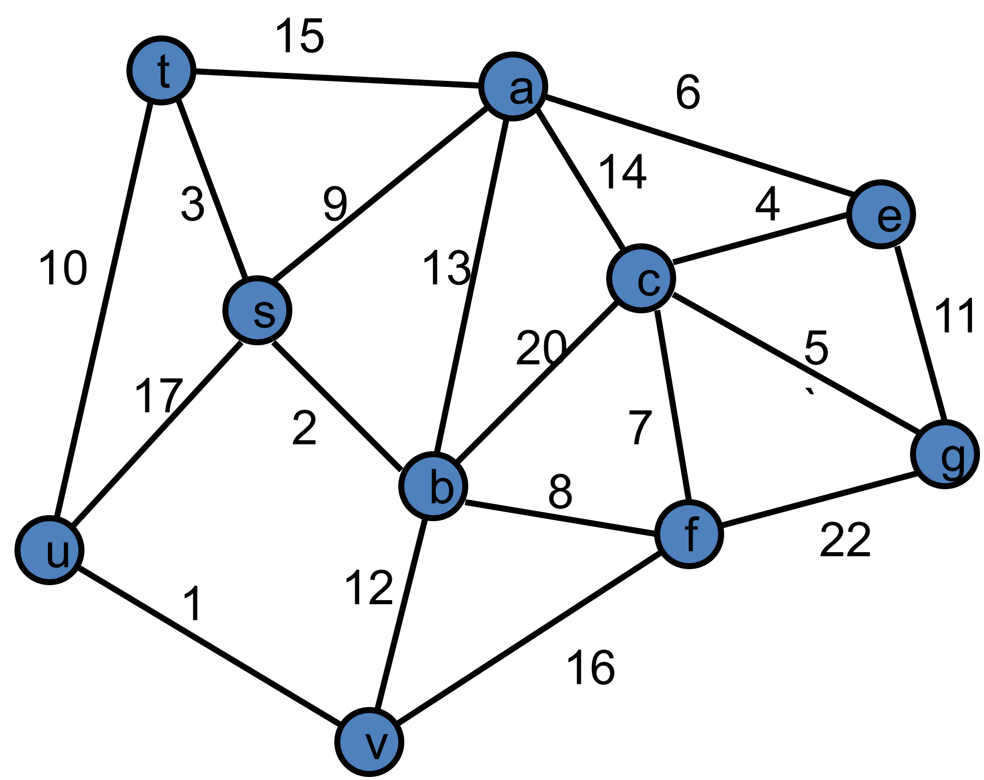
- [Jarnik/Prim/Dijkstra] Extend a tree by including the cheapest out going edge
- [Kruskal] Add the cheapest edge that joins disjoint components
- [ReverseDelete] Delete the most expensive edge that does not disconnect the graph



Greedy Algorithm 1
Jarnik's/Prim's/Djikstra's Algorithm

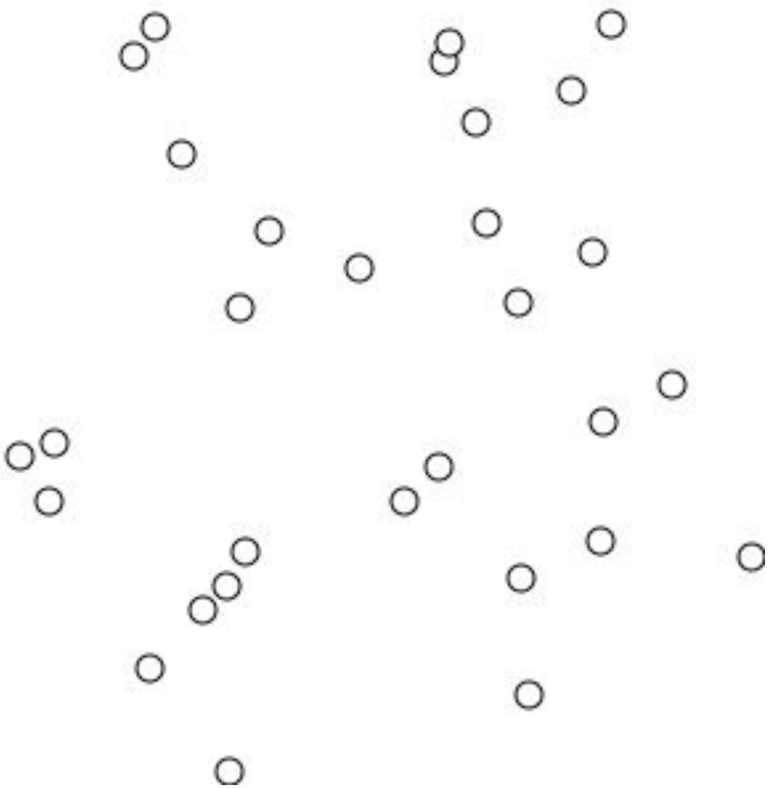
- Extend a tree by including the cheapest out going edge

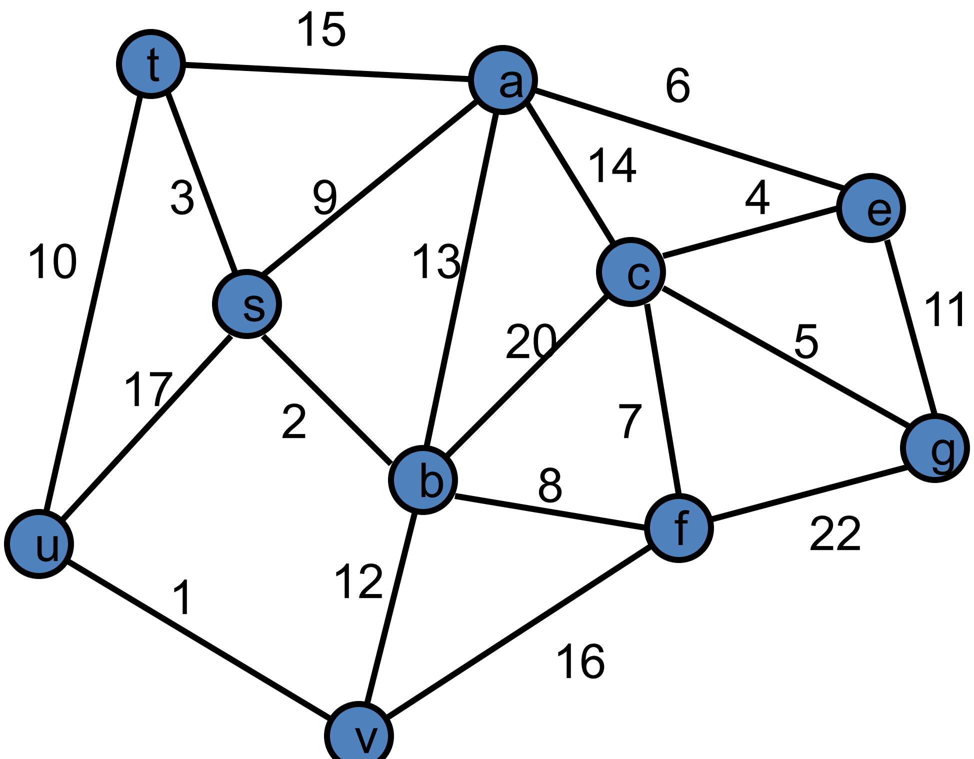




Greedy Algorithm 2
Kruskal's Algorithm

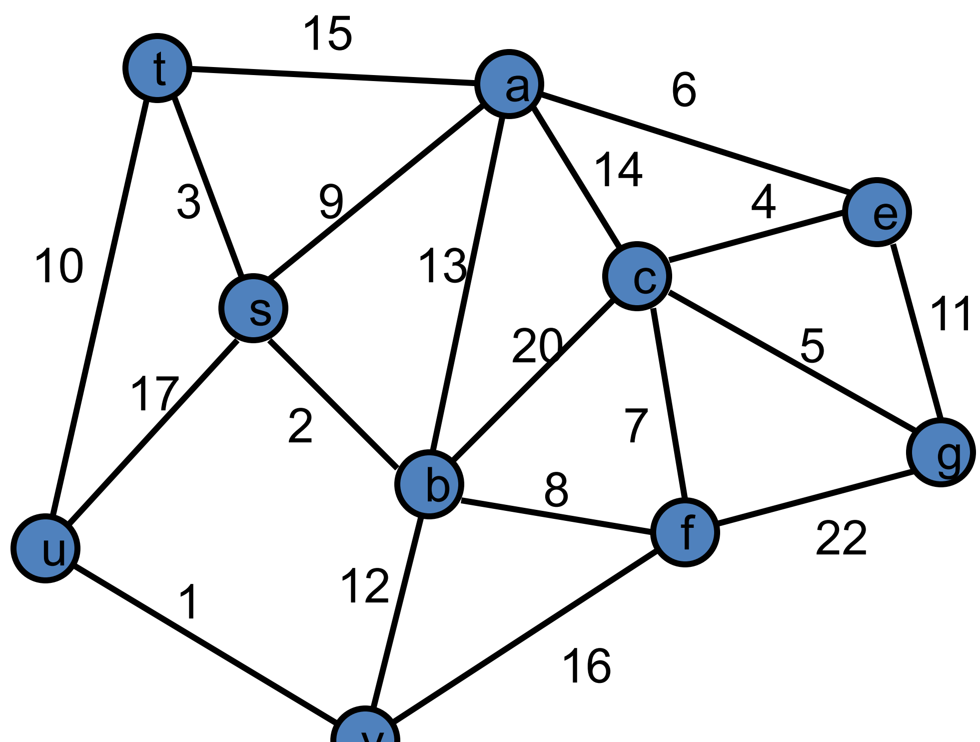
- Add the cheapest edge that joins disjoint components





Greedy Algorithm 3
Reverse-Delete Algorithm

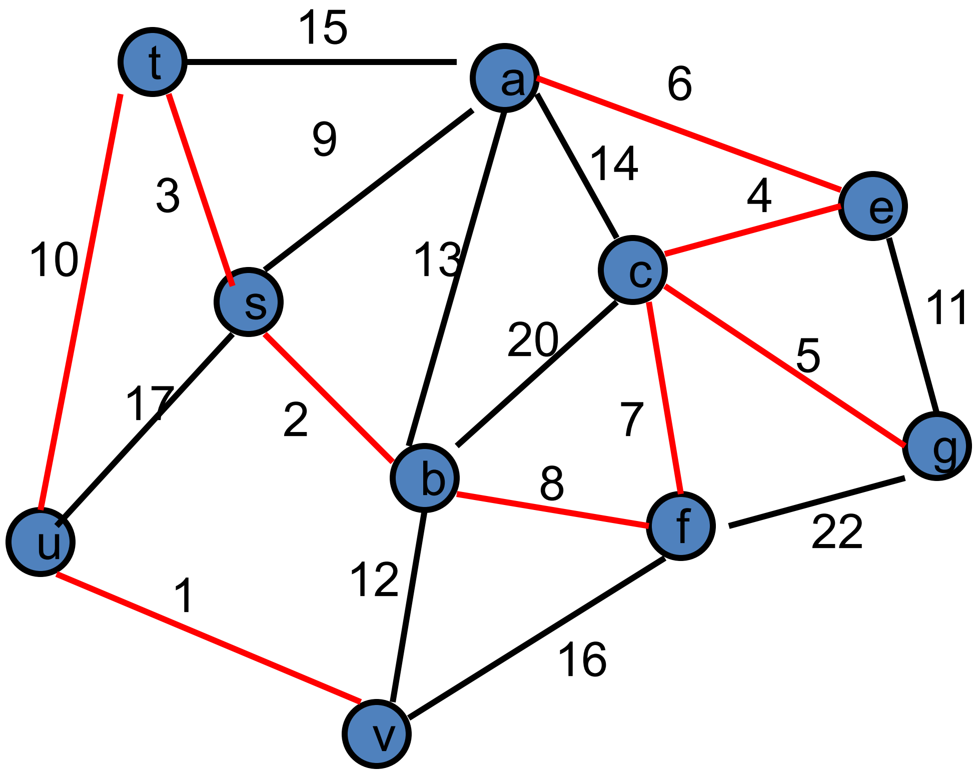
- Delete the most expensive edge that does not disconnect the graph
- Also by Kruskal



Construct the MST with the reverse-delete algorithm

Label the edges in order of removal

Minimum Spanning Tree

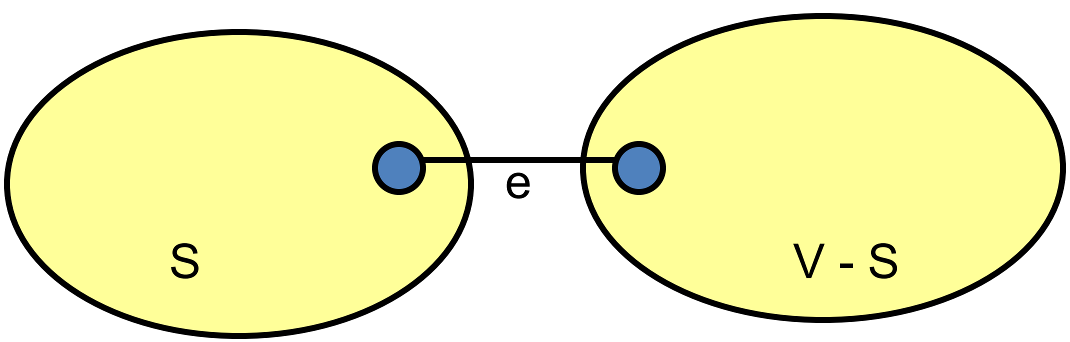


Why do the greedy algorithms work?

- For simplicity, assume all edge costs are distinct

Edge inclusion lemma

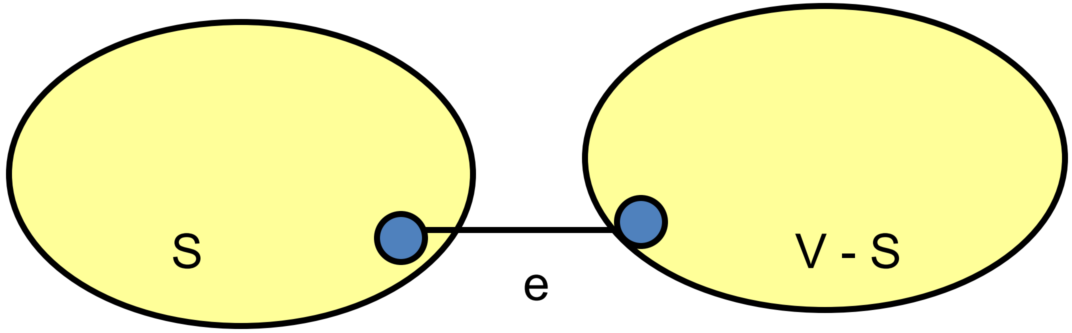
- Let S be a subset of V, and suppose e = (u, v) is the minimum cost edge of E, with u in S and v in V-S
- e is in every minimum spanning tree of G
  - Or equivalently, if e is not in T, then T is not a minimum spanning tree



e is the minimum cost edge between S and V-S

Proof

- Suppose T is a spanning tree that does not contain e
- Add e to T, this creates a cycle
- The cycle must have some edge $e_1 = (u_1, v_1)$ with u_1 in S and v_1 in V-S

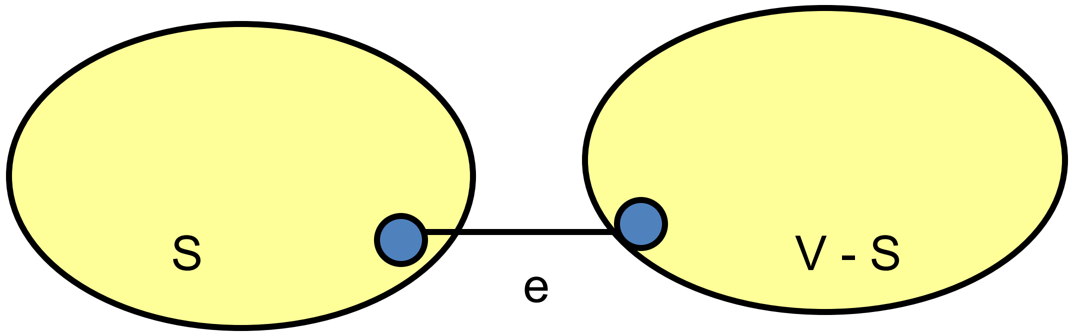


- $T_1 = T - \{e_1\} + \{e\}$ is a spanning tree with lower cost
- Hence, T is not a minimum spanning tree

e is the minimum cost edge between S and V-S

Proof

- Suppose T is a spanning tree that does not contain e
- Add e to T, this creates a cycle
- The cycle must have some edge $e_1 = (u_1, v_1)$ with u_1 in S and v_1 in V-S



- $T_1 = T - \{e_1\} + \{e\}$ is a spanning tree with lower cost
- Hence, T is not a minimum spanning tree

This is an exchange argument

Negative edge weights

- Dijkstra's algorithm (for shortest paths) fails
- Financial arbitrage corresponds to negative weight cycles
- Minimum spanning tree algorithms don't care
- Can you fix Dijkstra's to work with negative weights?

Errata

- Last week, we suggested that you could make the dummy-node algorithm for shortest paths (replace edges with weight *n* by *n* unweighted edges) work for non-integer weights (e.g. $\sqrt{2}$) by applying a function (e.g. squaring the weights)
    - This doesn’t work!