CSE 417
Algorithms and Complexity

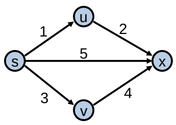
Autumn 2020
Lecture 12
Shortest Paths Algorithm and Minimum Spanning Trees

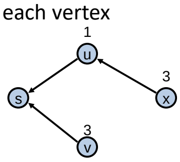
Announcements

- Reading
  - 4.4, 4.5, 4.7, 4.8
- Homework
  - Assignment will include a sample midterm

Single Source Shortest Path Problem

- Given a directed graph and a start vertex s
  - Determine distance of every vertex from s
  - Identify shortest paths to each vertex

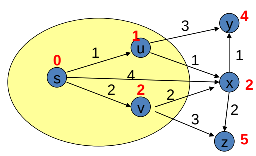




Assume all edges have non-negative cost

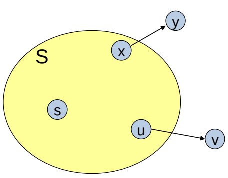
Dijkstra's Algorithm

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))



Correctness Proof

- Elements in S have the correct label
- Induction: when v is added to S, it has the correct distance label
  - Dist(s, v) = d[v] when v added to S



Dijkstra Implementation

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))

- Basic implementation requires Heap for tracking the distance values
- Run time O(m log n)

O(n^2) Implementation for Dense Graphs

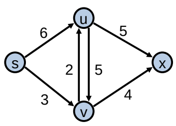
```
FOR i := 1 TO n
        d[i] := Infinity;  visited[i] := FALSE;
d[s] := 0;

FOR i := 1 TO n
        v := -1;  dMin := Infinity;
        FOR j := 1 TO n
                IF visited[j] = FALSE AND d[j] < dMin
                        v := j; dMin := d[j];
        IF v = -1
                RETURN;
        visited[v] := TRUE;

        FOR j := 1 TO n
                IF d[v] + len[v, j] < d[j]
                        d[j] := d[v] + len[v, j];
                        prev[j] := v;
```

Future stuff for shortest paths

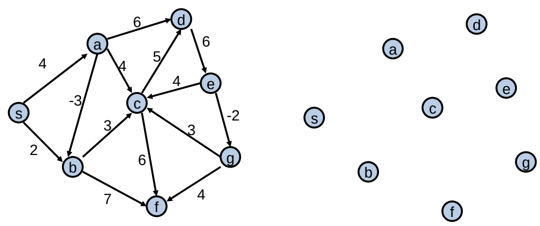
- Bellman-Ford Algorithm
  - O(nm) time
  - Handles negative cost edges
    - Identifies negative cost cycle if present
  - Dynamic programming algorithm
  - Very easy to implement

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] = negative infinity; d[v] = infinity for v != s

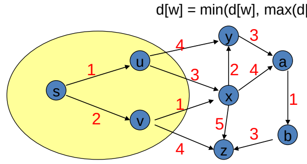
While S != V

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], max(d[v], c(v, w)))



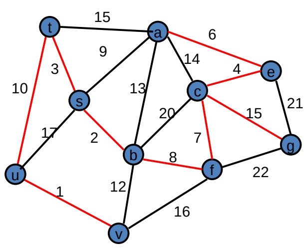
Minimum Spanning Tree

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

Minimum Spanning Tree Definitions

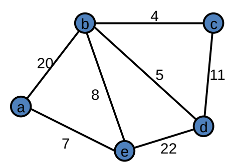
- G=(V,E) is an UNDIRECTED graph
- Weights associated with the edges
- Find a spanning tree of minimum weight
  - If not connected, complain

Minimum Spanning Tree



Greedy Algorithms for Minimum Spanning Tree

- Extend a tree by including the cheapest out going edge
- Add the cheapest edge that joins disjoint components
- Delete the most expensive edge that does not disconnect the graph

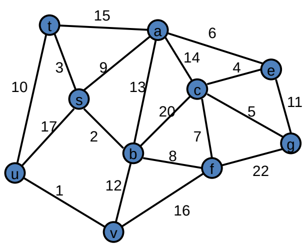


Greedy Algorithm 1
Prim's Algorithm

- Extend a tree by including the cheapest out going edge

Construct the MST with Prim's algorithm starting from vertex a

Label the edges in order of insertion

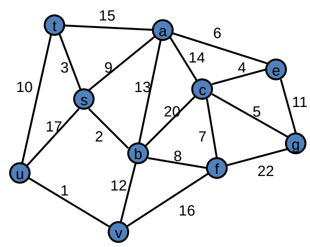


Greedy Algorithm 2
Kruskal's Algorithm

- Add the cheapest edge that joins disjoint components

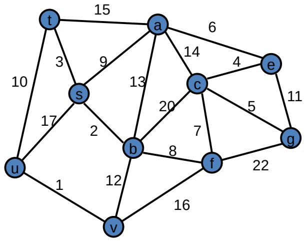
Construct the MST with Kruskal's algorithm

Label the edges in order of insertion



Greedy Algorithm 3
Reverse-Delete Algorithm

- Delete the most expensive edge that does not disconnect the graph



Construct the MST with the reverse-delete algorithm

Label the edges in order of removal

Dijkstra's Algorithm
for Minimum Spanning Trees

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

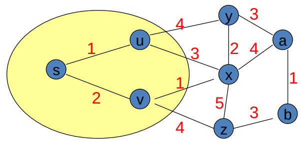
While S != V

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

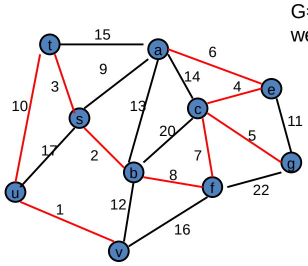
Add v to S

For each w in the neighborhood of v

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



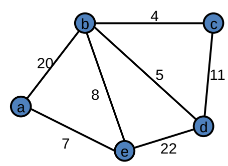
Minimum Spanning Tree



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

Greedy Algorithms for Minimum Spanning Tree

- [Prim] 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

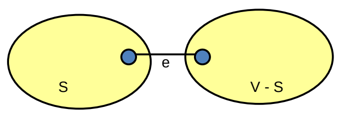


Why do the greedy algorithms work?

- For simplicity, assume all edge costs are distinct

Edge inclusion lemma

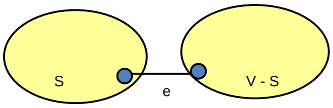
- Let S be a subset of V, and suppose e = (u, v) is the minimum cost edge of E, with u in S and v in V-S
- e is in every minimum spanning tree of G
  - Or equivalently, if e is not in T, then T is not a minimum spanning tree



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