Announcements

- Reading
  - 4.4, 4.5, 4.7
- Midterm
  - Wednesday, February 8
  - In class, closed book
  - Material through 4.7
  - Old midterm questions available
    - Note – some listed questions are out of scope
- No homework due on February 10

Dijkstra’s Algorithm

\[ S = \{ \} ; \; d[s] = 0 ; \; d[v] = \text{infinity} \text{ for } v \neq s \]

\[ \text{While } S \neq V \]
\[ \quad \text{Choose } v \text{ in } V - S \text{ with minimum } d[v] \]
\[ \quad \text{Add } v \text{ to } S \]
\[ \quad \text{For each } w \text{ in the neighborhood of } v \]
\[ \quad \quad d[w] = \min(d[w], d[v] + c(v, w)) \]

Assume all edges have non-negative cost

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

\[ S = \{ \} ; \; d[s] = 0 ; \; d[v] = \text{infinity} \text{ for } v \neq s \]

\[ \text{While } S \neq V \]
\[ \quad \text{Choose } v \text{ in } V - S \text{ with minimum } d[v] \]
\[ \quad \text{Add } v \text{ to } S \]
\[ \quad \text{For each } w \text{ in the neighborhood of } v \]
\[ \quad \quad d[w] = \min(d[w], d[v] + c(v, w)) \]

- 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 \( n \)
\[ d[i] := \text{Infinity} ; \; \text{visited}[i] := \text{FALSE} ; \; d[s] := 0 ; \]

FOR \( i := 1 \) TO \( n \)
\[ v := -1 ; \; dMin := \text{Infinity} ; \]
\[ \text{FOR } j := 1 \text{ TO } n \]
\[ \quad \text{IF } \text{visited}[j] = \text{FALSE} \text{ AND } d[j] < dMin \]
\[ \quad \quad v := j ; \; dMin := d[j] ; \]
\[ \text{IF } v = -1 \]
\[ \quad \text{RETURN} ; \]
\[ \quad \text{visited}[v] := \text{TRUE} ; \]

FOR \( j := 1 \) TO \( n \)
\[ \quad \text{IF } d[v] + \text{len}[v, j] < d[j] \]
\[ \quad \quad d[j] := d[v] + \text{len}[v, j] ; \]
\[ \quad \quad \text{prev}[j] := v ; \]
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

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

Greedy Algorithms for Minimum Spanning Tree

- Extend a tree by including the cheapest outgoing 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 outgoing 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
Dijkstra's Algorithm for Minimum Spanning Trees

\[ S = \{\} \text{, } d[s] = 0 \text{, } d[v] = \text{infinity for } v \neq s \]

While \( S \neq V \)

Choose \( v \) in \( V \) with minimum \( d[v] \)
Add \( v \) to \( S \)
For each \( w \) in the neighborhood of \( v \)

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

Greedy Algorithms for Minimum Spanning Tree

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

Minimum Spanning Tree

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

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

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 \( e_t = (u_t, v_t) \) with \( u_t \) in \( S \) and \( v_t \) in \( V - S \)

- \( T_t = T \setminus \{e_t\} + \{e\} \) is a spanning tree with lower cost
- Hence, \( T \) is not a minimum spanning tree