Shortest Paths

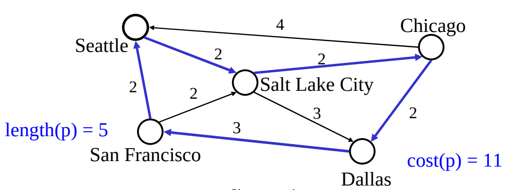
CSE 373
Data Structures
Winter 2007

Readings

- Reading Chapter 9
  - › Section 9.3

Recall Path cost ,Path length

- Path cost: the sum of the costs of each edge
- Path length: the number of edges in the path
  - › Path length is the unweighted path cost



Shortest Path Problems

- Given a graph G = (*V*, *E*) and a "source" vertex *s* in *V*, find the minimum cost paths from *s* to every vertex in *V*
- Many variations:
  - › unweighted vs. weighted
  - › cyclic vs. acyclic
  - › pos. weights only vs. pos. and neg. weights
  - › etc

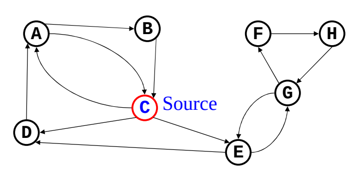
Why study shortest path problems?

- Traveling on a budget: What is the cheapest airline schedule from Seattle to city X?
- Optimizing routing of packets on the internet:
  - › Vertices are routers and edges are network links with different delays. What is the routing path with smallest total delay?
- Shipping: Find which highways and roads to take to minimize total delay due to traffic
- etc.

Unweighted Shortest Path

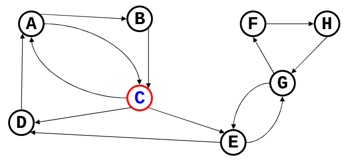
Problem: Given a "source" vertex *s* in an unweighted directed graph
G = (*V*,*E*), find the shortest path from *s* to all vertices in G

Only interested in path lengths



Breadth-First Search Solution

- Basic Idea: Starting at node s, find vertices that can be reached using 0, 1, 2, 3, …, N-1 edges (works even for cyclic graphs!)



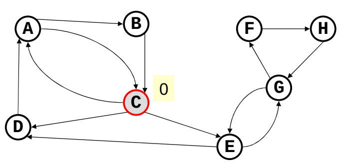
Breadth-First Search Alg.

- Uses a queue to track vertices that are “nearby”
- source vertex is **s**

```
Distance[s] := 0
Enqueue(Q,s); Mark(s)//After a vertex is marked once
                      // it won't be enqueued again
while queue is not empty do
    X := Dequeue(Q);
    for each vertex Y adjacent to X do
       if Y is unmarked then
          Distance[Y] := Distance[X] + 1;
          Previous[Y] := X;//if we want to record paths
          Enqueue(Q,Y); Mark(Y);
```

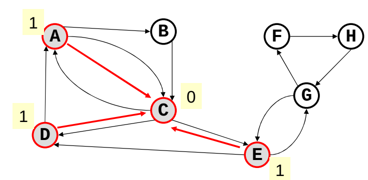
- Running time = O(|*V*| + |*E*|)

Example: Shortest Path length



Queue Q = C

Example (ct’d)

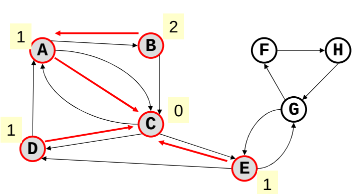


Queue Q = A D E

Indicates the vertex is marked

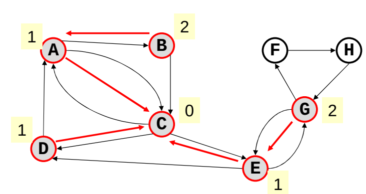
Previous pointer

Example (ct’d)



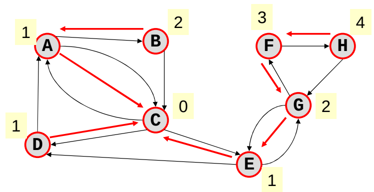
Q = D E B

Example (ct’d)



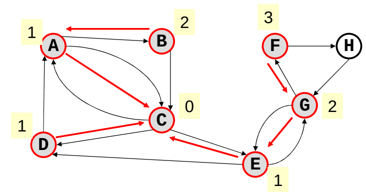
Q = B G

Example (ct'd)



Q = F

Example (ct'd)



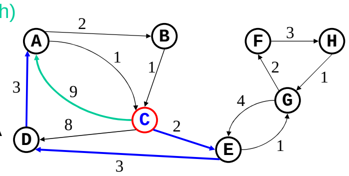
Q = H

What if edges have weights?

- Breadth First Search does not work anymore
  - minimum *cost* path may have more edges than minimum *length* path

Shortest path (length) from C to A:
C→A (cost = 9)

Minimum Cost Path = C→E→D→A (cost = 8)



Dijkstra's Algorithm for Weighted Shortest Path

- Classic algorithm for solving shortest path in weighted graphs (without negative weights)
- A greedy algorithm (irrevocably makes decisions without considering future consequences)
- Each vertex has a cost for path from initial vertex

Dijkstra's Algorithm

- Edsger Dijkstra (1930-2002)

Basic Idea of Dijkstra's Algorithm (1959)

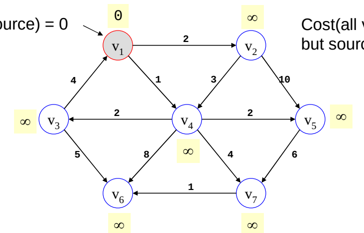
- Find the vertex with smallest cost that has not been "marked" yet.
- Mark it and compute the cost of its neighbors.
- Do this until all vertices are marked.
- Note that each step of the algorithm we are marking one vertex and we won't change our decision: hence the term "greedy" algorithm
- Works for directed and undirected graphs

Dijkstra's Shortest Path Algorithm

- Initialize the cost of s to 0, and all the rest of the nodes to ∞
- Initialize set S to be ∅
  - › S is the set of nodes to which we have a shortest path
- While S is not all vertices
  - › Select the node A with the lowest cost that is not in S and identify the node as now being in S
  - › for each node B adjacent to A
    - if cost(A)+cost(A,B) < B's currently known cost
      - – set cost(B) = cost(A)+cost(A,B)
      - – set previous(B) = A so that we can remember the path

Example: Initialization

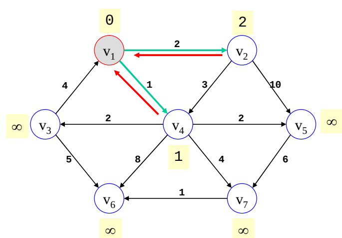
Cost(source) = 0



Cost(all vertices but source) = ∞

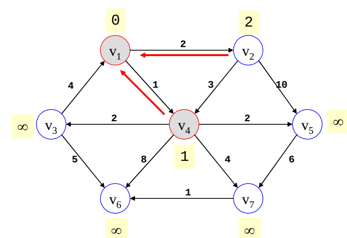
Pick vertex not in S with lowest cost.

Example: Update Cost neighbors



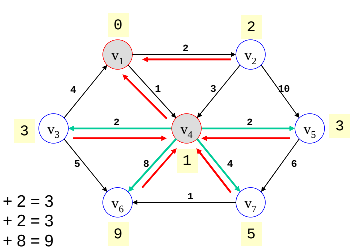
Cost(v_2) = 2
Cost(v_4) = 1

Example: pick vertex with lowest cost and add it to S



Pick vertex not in S with lowest cost, i.e., v_4

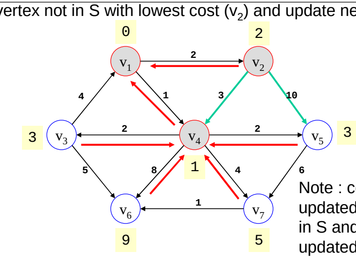
Example: update neighbors



Cost(v_3) = 1 + 2 = 3
Cost(v_5) = 1 + 2 = 3
Cost(v_6) = 1 + 8 = 9
Cost(v_7) = 1 + 4 = 5

Example (Ct'd)

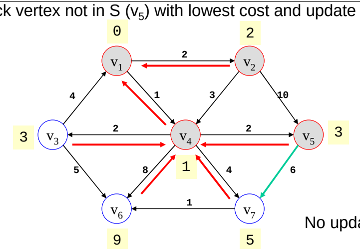
Pick vertex not in S with lowest cost (v_2) and update neighbors



Note : cost(v_4) not updated since already in S and cost(v_5) not updated since it is larger than previously computed

Example: (ct'd)

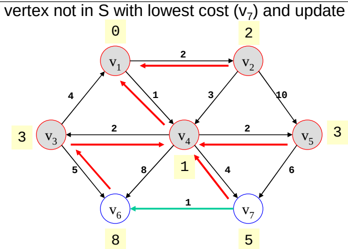
Pick vertex not in S (v_5) with lowest cost and update neighbors



No updating

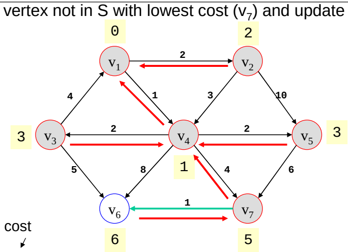
Example: (ct'd)

Pick vertex not in S with lowest cost (v_7) and update neighbors



Example: (ct'd)

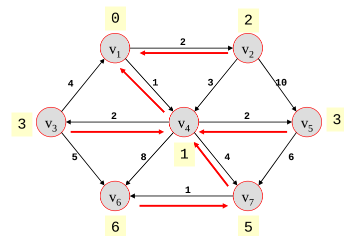
Pick vertex not in S with lowest cost (v_7) and update neighbors



Previous cost

Cost(v_6) = min (8, 5+1) = 6

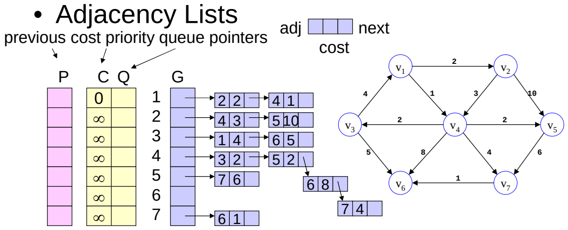
Example (end)



Pick vertex not in S with lowest cost (v_6) and update neighbors

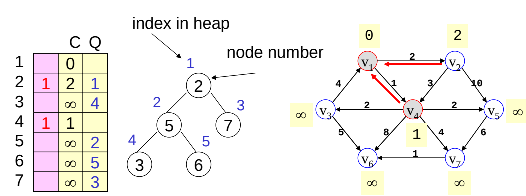
Data Structures

- Adjacency Lists



Priority queue for finding and deleting lowest cost vertex and for decreasing costs (Binary Heap works)

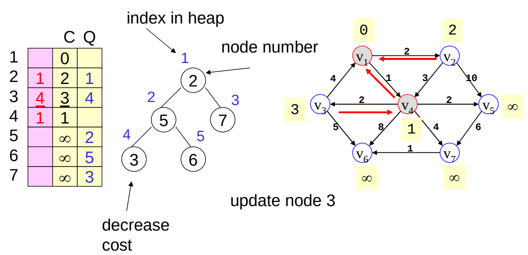
Priority Queue



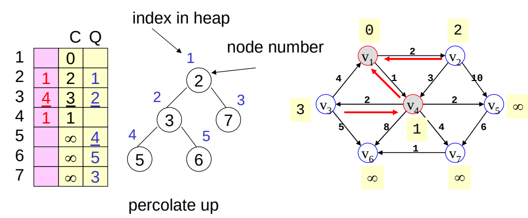
This is somewhat arbitrary and depends when the heap was first built

Before the update, but after find min.,i.e., v1 and v4 have been "deletemin"

Priority Queue



Priority Queue



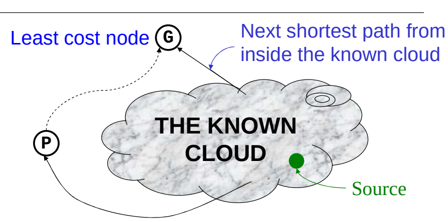
Time Complexity

- n vertices and m edges
- Initialize data structures O(n+m)
- Find min cost vertices O(n log n)
  - n delete mins
- Update costs O(m log n)
  - Potentially m updates
- Update previous pointers O(m)
  - Potentially m updates
- Total time O((n + m) log n) - very fast.

Correctness

- Dijkstra's algorithm is an example of a greedy algorithm
- Greedy algorithms always make choices that currently seem the best
  - Short-sighted – no consideration of long-term or global issues
  - Locally optimal does not always mean globally optimal
- In Dijkstra's case – choose the least cost node, but what if there is another path through other vertices that is cheaper?

"Cloudy" Proof



- If the path to G is the next shortest path, the path to P must be at least as long. Therefore, any path through P to G cannot be shorter!

Inside the Cloud (Proof)

- Everything inside the cloud has the correct shortest path
- Proof is by induction on the number of nodes in the cloud:
  - Base case: Initial cloud is just the source with shortest path 0
  - Inductive hypothesis: cloud of k-1 nodes all have shortest paths
  - Inductive step: choose the least cost node G → has to be the shortest path to G (previous slide). Add k-th node G to the cloud