Shortest Paths

CSE 373
Data Structures
Lecture 21

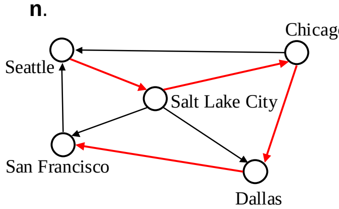
Readings and References

- Reading
  - › Section 9.3 , Section 10.3.4

Some slides based on: CSE 326 by S. Wolfman, 2000

Path

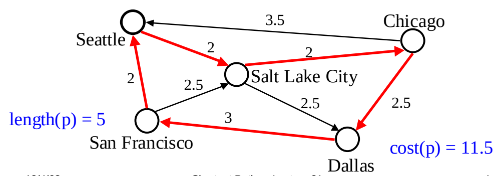
- A *path* is a list of vertices $\{v_1, v_2, \ldots, v_n\}$ such that (v_i, v_{i+1}) is in **E** for all $0 \le i < n$.



p = {Seattle, Salt Lake City, Chicago, Dallas, San Francisco}

Path cost and 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 (each edge = 1)



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

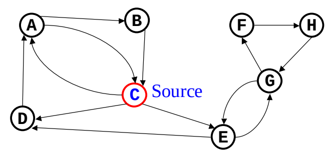
Why study shortest path problems?

- 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

Unweighted Shortest Path Problem

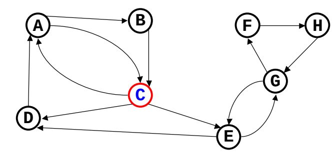
Problem: Given a “source” vertex *s* in an unweighted graph

G = (*V*,*E*), find the shortest path from *s* to all vertices in G



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!)



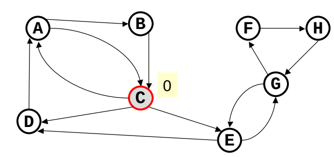
Breadth-First Search Algorithm

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

```
Distance[s] := 0
Enqueue(Q,s); Mark(s)
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;
         Enqueue(Q,Y); Mark(Y);
```

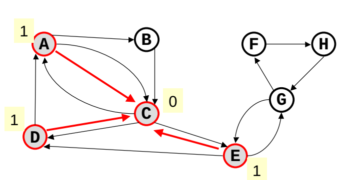
- Running time = O(|*V*| + |*E*|)

Shortest Path



Q = C

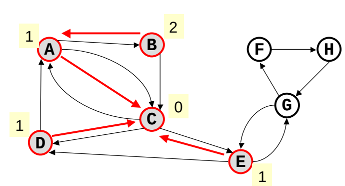
Shortest Path



Q = A D E

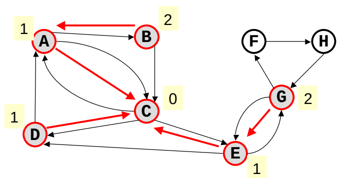
Previous pointer

Shortest Path



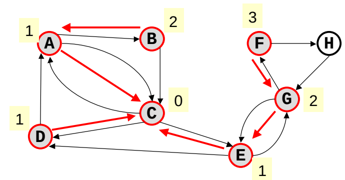
Q = D E B

Shortest Path



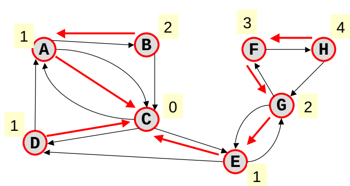
Q = B G

Shortest Path



Q = F

Shortest Path



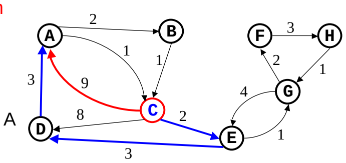
Q = H

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 from C to A:
Cà A (cost = 9)

Minimum Cost Path = Cà Eà Dà A (cost = 8)



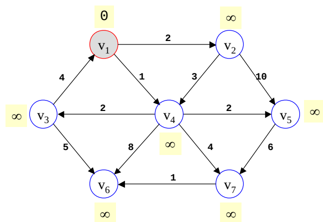
Dijkstra's Algorithm for Weighted Shortest Path

- 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 Shortest Path Algorithm

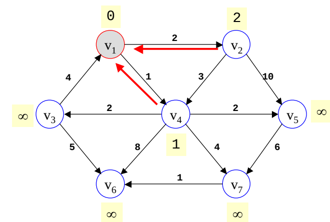
- Initialize the cost of s to 0, and all the rest of the nodes to ∞
- Initialize set S to be ∅
  - S is the set of nodes to which we have a shortest path
- While S is not all vertices
  - Select the node A with the lowest cost that is not in S and identify the node as now being in S
  - for each node 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

A weighted directed graph



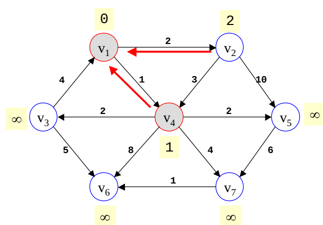
Pick vertex not in S with lowest cost.

A weighted directed graph



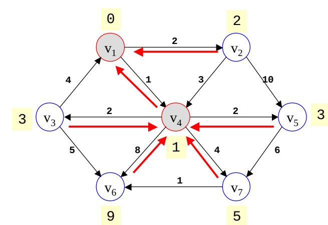
Update neighbors

A weighted directed graph



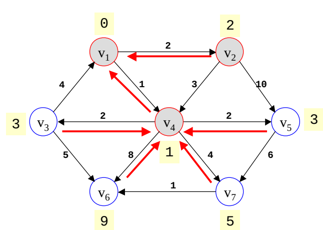
Pick vertex not in S with lowest cost

A weighted directed graph



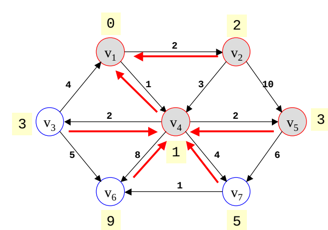
Update neighbors

A weighted directed graph



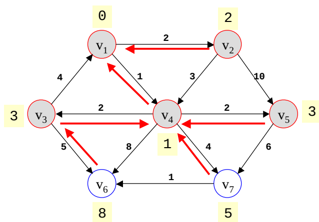
Pick vertex not in S with lowest cost and update neighbors

A weighted directed graph



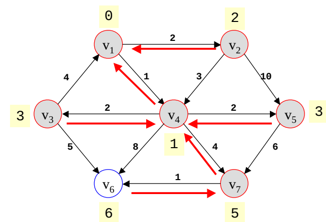
Pick vertex not in S with lowest cost and update neighbors

A weighted directed graph



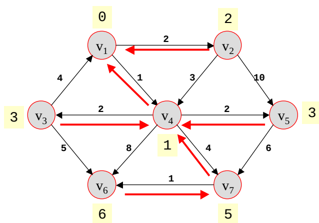
Pick vertex not in S with lowest cost and update neighbors

A weighted directed graph



Pick vertex not in S with lowest cost and update neighbors

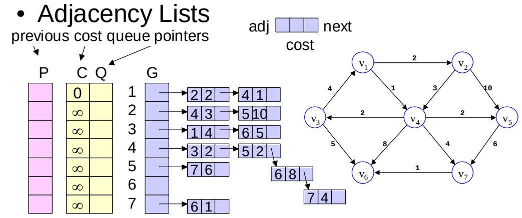
A weighted directed graph



Pick vertex not in S with lowest cost and update neighbors

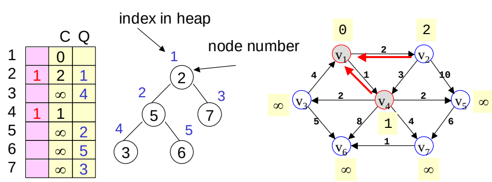
Data Structures

- Adjacency Lists



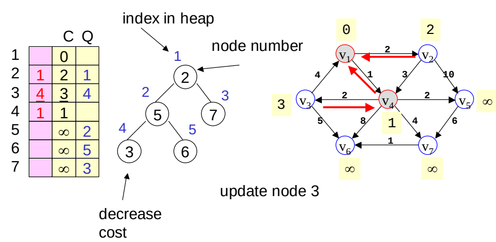
Priority queue for finding finding and deleting lowest cost vertex and for decreasing costs (Binary Heap works)

Priority Queue

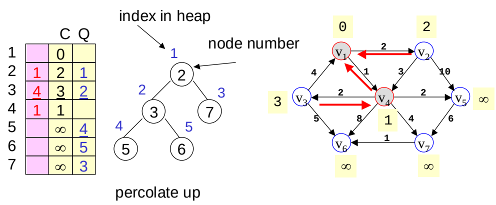


Before the update, but after find min.

Priority Queue



Priority Queue



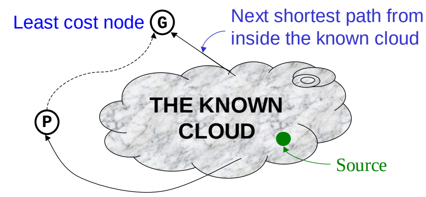
Time Complexity

- 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.

Does It Always Work?

- 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!

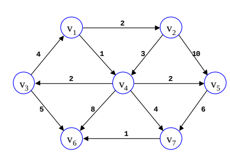
Inside the Cloud (Proof)

- 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 à has to be the shortest path to G (previous slide). Add k-th node G to the cloud

All Pairs Shortest Path

- Given a edge weighted directed graph G = (V,E) find for all u,v in V the length of the shortest path from u to v. Use matrix representation.

C	1	2	3	4	5	6	7
1	0	2	:	1	:	:	:
2	:	0	:	3	10	:	:
3	4	:	0	:	:	5	:
4	:	:	2	0	2	8	4
5	:	:	:	:	0	:	6
6	:	:	:	:	:	0	:
7	:	:	:	:	:	1	0



: = infinity

Matrix Representation

- C[i,j] = the cost of the edge (i,j)
  - › C[i,i] = 0 because no cost to stay where you are
  - › C[i,j] = infinity (:) if no edge from i to j.

C	1	2	3	4	5	6	7
1	0	2	:	1	:	:	:
2	:	0	:	3	10	:	:
3	4	:	0	:	:	5	:
4	:	:	2	0	2	8	4
5	:	:	:	:	0	:	6
6	:	:	:	:	:	0	:
7	:	:	:	:	:	1	0

Floyd – Warshall Algorithm

```
All_Pairs_Shortest_Path {
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      C[i,j] := min(C[i,j], C[i,k] + C[k,j]);
}

Note x + : = : by definition
```

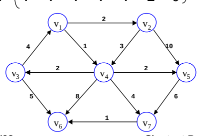
On termination C[i,j] is the length of the shortest path from i to j.

The Computation

C	1	2	3	4	5	6	7
1	0	2	:	1	:	:	:
2	:	0	:	3	10	:	:
3	4	:	0	:	:	5	:
4	:	:	2	0	2	8	4
5	:	:	:	:	0	:	6
6	:	:	:	:	:	0	:
7	:	:	:	:	:	1	0

➡

C	1	2	3	4	5	6	7
1	0	2	3	1	3	6	5
2	9	0	5	3	5	8	7
3	4	6	0	5	4	5	6
4	6	8	2	0	2	5	4
5	:	:	:	:	0	7	6
6	:	:	:	:	:	0	:
7	:	:	:	:	:	1	0



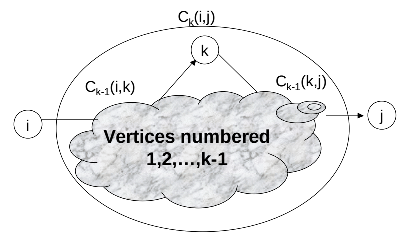
Proof of Correctness

- After the k-th time through the loop C[i,j] is the length of the shortest path that only passes through vertices numbered 1,2,…,k.
  - › Let C_k[i,j] be C[i,j] after k time through the loop.
- Base case: k = 0. C_0[i,j] is the cost of an edge that does not pass through any vertices.

Inductive Step

- Assume true for k-1.
  - › A shortest path from i to j that only goes through vertices 1,2, …, k does not go through vertex k at all.
    - $C_k[i,j] = C_{k-1}[i,j]$
  - › All shortest paths from i to j that only goes through vertices 1,2, …, k must go through vertex k.
    - $C_k[i,j] = C_{k-1}[i,k] + C_{k-1}[k,j]$

Cloud Argument



Time Complexity of All Pairs Shortest Path

- n is the number of vertices
- Three nested loops. $O(n^3)$
  - › Shortest paths can be found too (see the book).
- Repeated Dijkstra's algorithm
  - › $O(n(n + m)\log n)$ (= $O(n^3 \log n)$ for dense graphs).
  - › Run Dijkstra starting at each vertex.
  - › Dijkstra also gives the shortest paths not just their lengths.