Lecture 22: Shortest Paths

(CLRS 24.0, 24.3)

June 20th, 2002

1 Shortest Paths

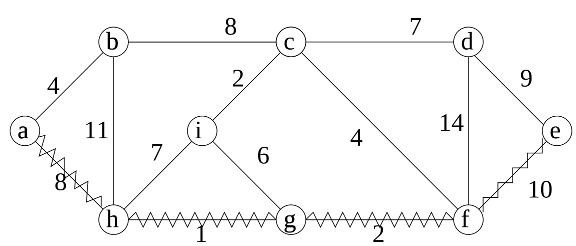
- We will now consider a problem related to minimum spanning trees; shortest paths
  - We already discussed how BFS can be used to find shortest paths if the length of a path is defined to be the number of edges on it
  - In general we have weights on edges and we are interested in shortest paths with respect to the sum of the weights of edges on a path

    Example: Finding shortest driving distance between two addresses (lots of www-sites with this functionality). Note that weight on an edge (road) can be more than just distance (weight can e.g. be a function of distance, road condition, congestion probability, etc).

- Formal definition of shortest path: $G = (V, E)$ weighted graph. Weight of path $P =< v_0, v_1, v_2, \cdots, v_k >$ is $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. Shortest path $\delta(u, v)$ from u to v has weight

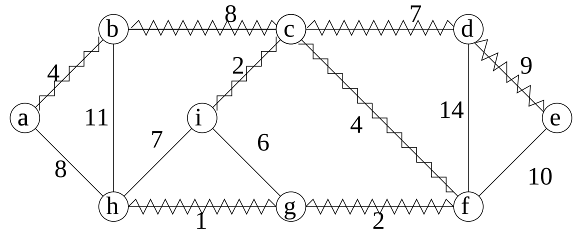
$$\delta(u,v) = \begin{cases} \min\{w(P) : P \text{ is path from } u \text{ to } v\} & \text{If path exists} \\ \infty & \text{Otherwise} \end{cases}$$

  Example: Shortest path from a to e (of length 21)



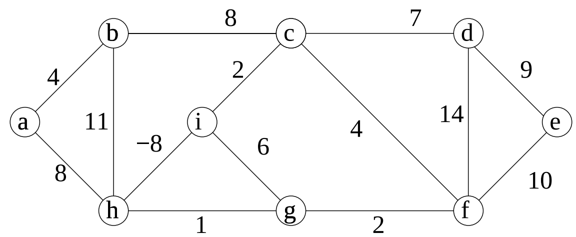
- Note:
  - If $P =< u = v_0, v_1, v_2, \cdots, v_k = v >$ is shortest path from u to v then for all $i < k$ $P' =< u = v_0, v_1, v_2, \cdots, v_i >$ is shortest path from u to v_i
  - Shortest path is not necessarily part of minimum spanning tree.

    Example: Minimum spanning tree for example graph:



- No (unique) shortest path exists if graph has cycle with negative weight

  Example: If we change weight of edge (h, i) to -8, we have a cycle (i,h,g) with negative weight (-1). Using this we can make the weight of path between a and e arbitrarily low by going through the cycle several times



  On the other hand, the problem is well defined if we let edge (h, i) have weight -7 (no negative cycles)

- We will *only* consider graphs with non-negative weights

- Different variants of shortest path problem:
  - *Single pair shortest path*: Find shortest path from u to v
  - *Single source shortest path (SSSP)*: Find shortest path from source s to all vertices $v \in V$
  - *All pair shortest path (APSP)*: Find shortest path from u to v for all $u, v \in V$

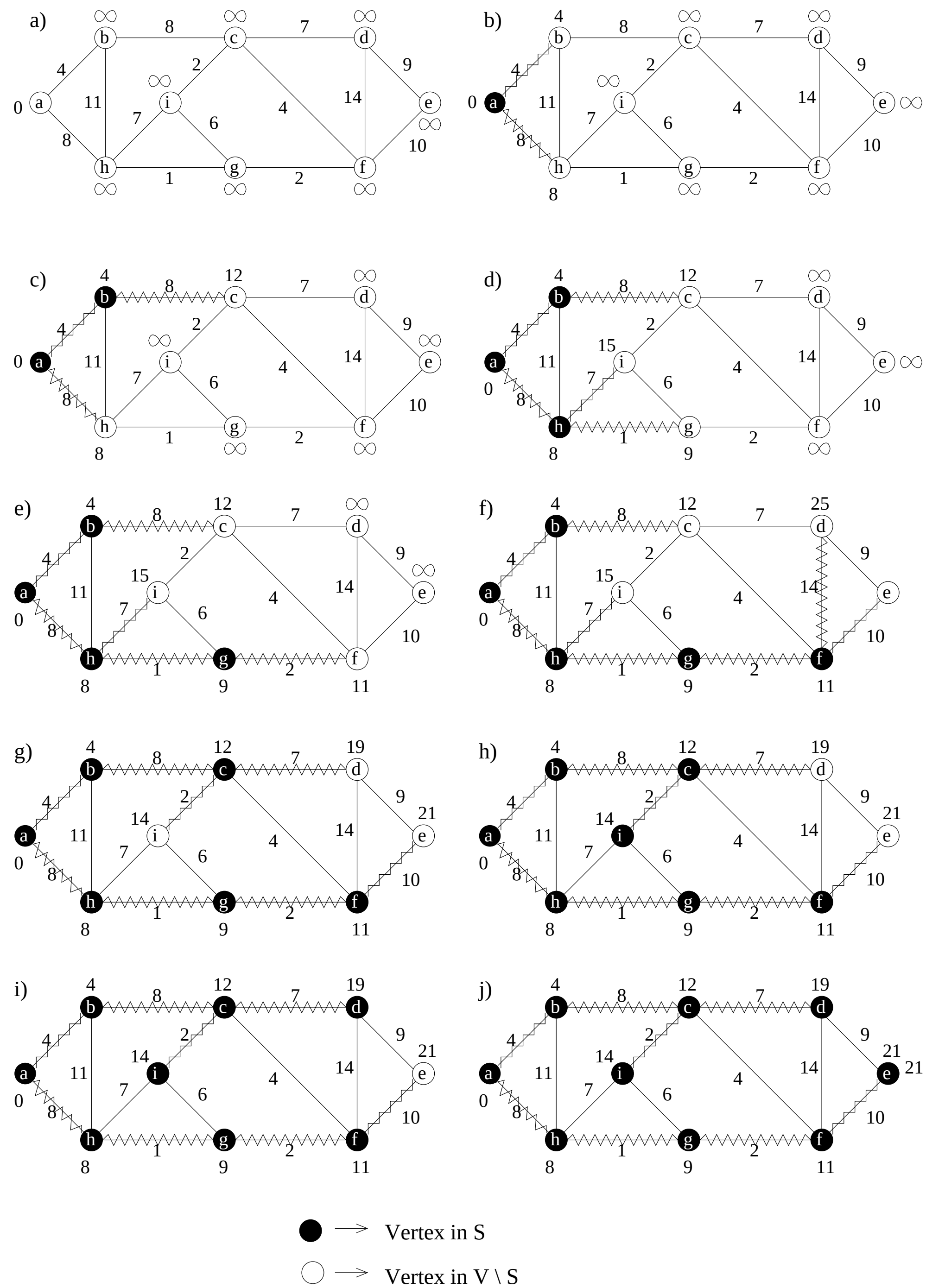
- Note:
  - No algorithm is known for computing a single pair shortest path better than solving the ("bigger") SSSP problem
  - APSP can be solved by running SSSP $|V|$ times
    $\Downarrow$
    We will concentrate on SSSP problem

2 SSSP for graphs with non-negative weights—Dijkstra's algorithm

- Recall Prim's greedy minimum spanning tree algorithm:
  - Grows tree out from source s; repeatedly add minimum edge out of tree
  - Correct by "cut theorem"
  - Implemented using priority queue on vertices not yet in the tree
- Dijkstra's greedy algorithm for SSSP works almost the same way:
  - Grow set (tree) S of vertices we know the shortest path to; repeatedly add new vertex v that can be reached from S using one edge. v is chosen as the vertex with the minimal path weight among paths $< s = v_0, v_1, \cdots v_i, v >$ with $v_j \in S$ for all $j \leq i$
  - Implemented using priority queue on vertices in $V \setminus S$.

```
Dijkstra(s)
    FOR each v ∈ V DO
        d[v] = ∞
        INSERT(Q, v, ∞)
    OD
    S = ∅
    d[s] = 0
    CHANGE(Q, s, 0)
    WHILE Q not empty DO
        u = DELETEMIN(Q)
        S = S ∪ {u}
        FOR each e = (u, v) ∈ E with v ∈ V \ S DO
            IF d[v] > d[u] + w(u, v) THEN
                d[v] = d[u] + w(u, v)
                CHANGE(Q, v, d[v])
                visit[v] = u
            FI
        OD
    OD
```

- Example:



- Analysis:
  - While loop runs $|V|$ times $\Rightarrow$ we perform $|V|$ DELETEMIN operations
  - We perform at most one CHANGE operation for each of the $|E|$ edges
    $\Downarrow$
    $O((|E| + |V|) \log |E|) = O(|E| \log |V|)$ running time

- Note:
  - Running time like Prim's minimal spanning tree algorithm
  - Algorithm computes *shortest path tree* (stored using visit[v]) which can be used to find actual shortest paths
  - Algorithm works for directed graphs as well
  - Like Prim's algorithm, Dijkstra's algorithm can be improved to $O(|V| \log |V| + |E|)$ using another heap (Fibonacci heap)

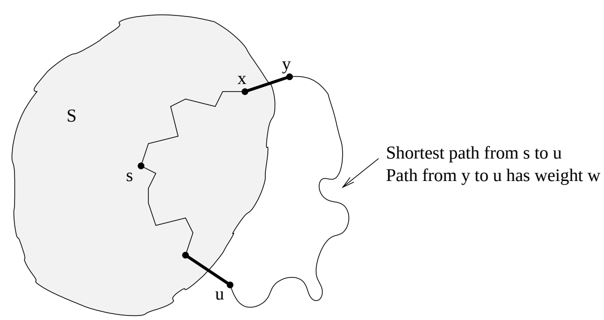
- Correctness:
  - We prove correctness by induction on size of S
  - We will prove that after each iteration of the while-loop the following *invariant* holds:
    a) $v \notin S \Rightarrow d[v]$ is length of shortest path from s to v among path of the form $< s, v_o, v_1, \ldots, v_k, v >$ where $v_1, v_2, \ldots, v_k \in S$
    b) $v \in S \Rightarrow d[v] = \delta(s, v)$ ($\delta(s, v)$ is length of shortest path from s to v)

    $\Downarrow$

    When algorithm terminates ($S = V$) we have solved SSSP
  - Proof:

    Invariant trivially holds initially ($S = \emptyset$). To prove that invariant holds after one iteration of while-loop, given that it holds before the iteration, we need to prove that after adding u to S:

    a) $d[v]$ correct for all $(u, v) \in E$ where $v \notin S$
       · Easily seen to be true since $d[v]$ explicitly updated by algorithm (all the new paths to v of the special type go through u)
    b) $d[u] = \delta(s, u)$
       · Assume $d[u] > \delta(s, u)$, that is, the found path is not the shortest
       · Consider shortest path to u and edge (x, y) on this path where $x \in S$ and $y \notin S$ (such an edge must exist since $s \in S$ and $u \notin S$)



       · We chose u such that $d[u]$ was minimized $\Rightarrow d[y] > d[u] \Rightarrow w$ must me $< 0 \Rightarrow$ contradiction since all weights are non-negative (note that we use that $d[y]$ is shortest path to y)

3 All pairs shortest path (APSP)—non-negative weights

- In the APSP problem, we want to compute the shortest path between any two vertices $u, v \in V$
  - Note that the output is of size $O(|V|^2)$ so we cannot hope to design a better than $O(|V|^2)$ time algorithm
- We can solve the problem simply by running Dijkstra's algorithm $|V|$ times $\Rightarrow$ $O(|V| \cdot |E| \log |V|)$ algorithm
  - In the worst case (dense graph) this is $O(|V|^3 \log |V|)$
- We can obtain a much simpler $O(|V|^3)$ algorithm by working on adjacency matrix A:

```
FOR k = 1 to |V| do
    FOR i = 1 to |V| DO
        FOR j = 1 to |V| DO
            IF A[i,j] > A[i,k] + A[k,j] THEN
                A[i,j] = A[i,k] + A[k,j]
            FI
        OD
    OD
OD
```

- Correctness:
  - We prove correctness by induction
  - We will prove that after each iteration of the k-loop the following *invariant* holds:
    After the k'th (out of $|V|$) iterations, $A[i,j]$ contains the length of shortest path from v_i to v_j that (apart from v_i and v_j) only contains vertices of index at most k
    $\Downarrow$
    When algorithm terminates we have solved APSP
  - Proof:
    * Invariant holds initially (we start with adjacency matrix A).
    * When "adding" vertex with index k we explicitly check all new paths between v_i and v_j through v_k for all $|V|^2$ pairs.
- Note:
  - We can easily produce adjacency-matrix from adjacency list in $O(|V^2|)$ time
  - Algorithm runs in $O(|V|^3)$ time, even if the graph is sparse. Using algorithm based on Dijkstra's algorithm we will get much better performance for sparse graphs.
  - Using more efficient heap, algorithm based on Dijkstra's algorithm can be improved to $O(|V|^2 \log |V| + |V| \cdot |E|) = O(|V|^3)$