CPS260/BGT204.1 Algorithms in Computational Biology	**September 17, 2003**

Lecture 7: Minimum Spanning Trees, Shortest Path

Lecturer: Pankaj K. Agarwal *Scribe: Joe Volpe*

Graphs are one of the most fundamental ways to represent data, so it is important to understand them and how to work with them. Continuing our discussion of graphs from previous lectures, we discuss two topics in this lecture:

- Minimum spanning trees
- Shortest paths

Let $G = (V, E)$ be a weighted graph, with $w : E \to \Re$ being the weight function.

7.1 Minimum spanning trees

Given a graph $G = (V, E)$, a spanning tree $T = (V', E')$ of G is a tree with $E' \subseteq E$. Recall that a graph is a tree with no cycles. If G is connected, a spanning tree always exists. The weight of T is the sum of the weights of its edges, i.e.,

$$w(T) = \sum_{i=1}^{j} w(e) \tag{7.1}$$

A minimum spanning tree of G is a spanning tree with the minimum weight. Note that a minimum spanning tree is not unique.

7.1.1 Motivation for computing MST

MST-type problems show up in biological situations like clustering, where genes that appear to have have similar functionality are grouped together. Using MST algorithms, these groups can be analyzed to find distances not only between the like genes, but between groups of like genes as well.

7.1.2 Algorithms for MST

There are two popular algorithms for tackling the MST problem, Kruskal's and Prim's. In this lecture, we only describe Kruskal's. However, both algorithms are based on the following two ideas:

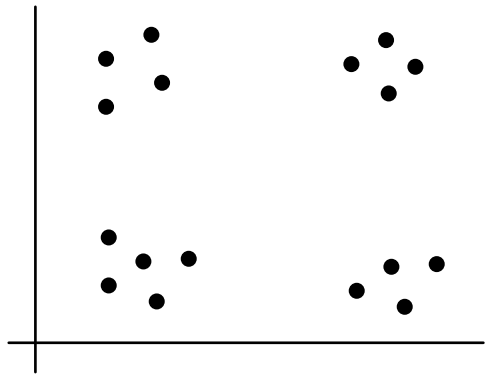

Figure 7.1: Clustered genes

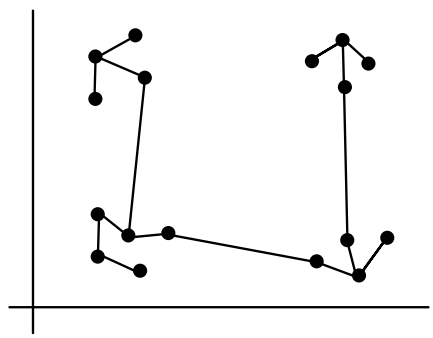

Figure 7.2: Clustered genes in a MST

(i) Let T be a MST, and let $e = u, v$ be an edge of T. $T \setminus \{e\}$ consist of two connected components. Let V_1, V_2 be the set of vertices in each of these connected components. Then:

$$w(e) = \min_{x \in V_1, y \in V_2} w(x, y) \tag{7.2}$$

(ii) Suppose e=$\{u$,$v\} \notin T$. Let $e_1, e_2, \ldots, e_k$ be the unique path in T from u to v. Then:

$$w(e) \geq \max_{1 \leq i \leq k} w(e_i) \tag{7.3}$$

Kruskal's algorithm is a greedy algorithm that builds a MST by adding one edge at a time. It sorts the edges in nondescending order of their weight, and considers the edges in this order. The algorithm maintains a colleciton of trees at any time,each of which is a subtree of the final MST. An edge is added to the tree if it does not create a cycle, i.e. it connects two different subtrees. The so-called *union-find* data structure is used to determine if the edge connects two different subtrees. The algorithm is thus:

```
KRUSKAL(V, E)
    SORT EDGES E BY WEIGHT;
    INITIALIZE E*=∅;
    CONSIDER e ∈ E IN ORDER;
        IF e FORMS A CYCLE WITH E* THEN
            DO NOTHING;
        ELSE ADD e TO E*, (e → E*)
    RETURN E*;

    OUTPUT = T* = (V, E*)
```

Kruskal's algorithm has a running time of *O*(E lg V).

7.2 Shortest Paths

Let $G = (V, E)$, $w : E \to \Re$, be a weighted, driected graph. The weight of a path T'=$V_0 \ldots V_k$ is

$$w(T') = \sum_{i=0}^{k} w(v_{i-1}, v_i) \tag{7.4}$$

The distance between u and v in G is

$$d_G(u, v) = \min_{T':u \to v} w(T') \tag{7.5}$$

A shortest path from u to v in G denoted as $T'_G(u, v)$, is a path from u to v whose weight is $d_G(u, v)$.

7.2.1 Motivation for understanding shortest-paths problems

There are several types of shortest-path problems given a graph G=(V,E):

- *Single-source shortest paths:* These problems focus on finding the shortest path from a given *s*ource vertex *s* to each vertex *v*, *v* ∈ V.
- *Single-destination shortest paths:* These problems deal with finding a shortest path to from a *d*estination vertex *d* to each vertex *v*. This is the reverse of the single-source problem above.
- *Single-pair shortest-path:* This is the basic problem – find the shortest path from *u* to *v*.

- *All-pairs shortest-path:* This problem involves finding a shortest path from *u* to *v* for every pair of (*u*, *v*) in G. This algorithm can be built by running the single source algorithm once from each vertex, but a faster solution typically exists.

Shortest-paths problems appear in a wide variety of cases. An obvious example is for driving directions from one city to another. In biological circustances, shortest-path algorithms are used to help determine the acestral origin for a protein.

7.2.2 Properties of shortest paths

Theorem 1 *Subpaths of shortest paths are shortest paths.*
This is because if there was some subpath which was **not** *a shortest path, we could substitute the shorter subpath and create an even shorter total path.*

Theorem 2 $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$ *where* $\delta(u,v) \equiv$ *weight of a shortest path from* u *to* v.
This is because the shortest path $u \to \cdots \to v$ *is no longer than any other path* $u \to \cdots \to v$. *In particular, it is no longer than the path concatenating the shortest path* $u \to \cdots \to x$ *with the shortest path* $x \to \cdots \to v$.

Also note that the shortest-path problem is only well-defined as long as the graph has no **negative weight cycles**. Otherwise we could always shorten the path by traversing the negative weight cycle an additional time.

7.2.3 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest paths problem on a weighted directed graph $G = (V, E)$ for the case in which all weights are nonnegative.

This algorithm maintains a set of vertices whose final shortest-path weights from the source s have already been determined. the algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. Thus the algorithm uses relaxation to progressively decrease an estimate $d[v]$ on the weight of the shortest path from the source s to each vertex $v \in V$. In the following implementation, we use a minimum priority queue Q keyed on their d values, which supports the following operations:

(i) Extract-min: Return the minimum element in Q and delete it from Q.

(ii) Decrease-key: Decrease the weight (value of key) of an item in Q.

```
DIJKSTRA(G, w, s)
    FOR each v ∈ V
        DO d[v] ← ∞
    ENDFOR
    d[s] ← 0
    S ← ∅
    Q ← V
    WHILE Q ≠ ∅
        DO u ← Extract − Min(Q)
        S ← S ∪ {u}
        FOR each v ∈ Adj[u]
            DO IF d[v] > d[u] + w(u, v)
            THEN d[v] ← d[u] + w(u, v)
            FI
        ENDFOR
    ENDWHILE
```

The working of Dijkstra's Algorithm is depicted by figures 7.3 and 7.4.

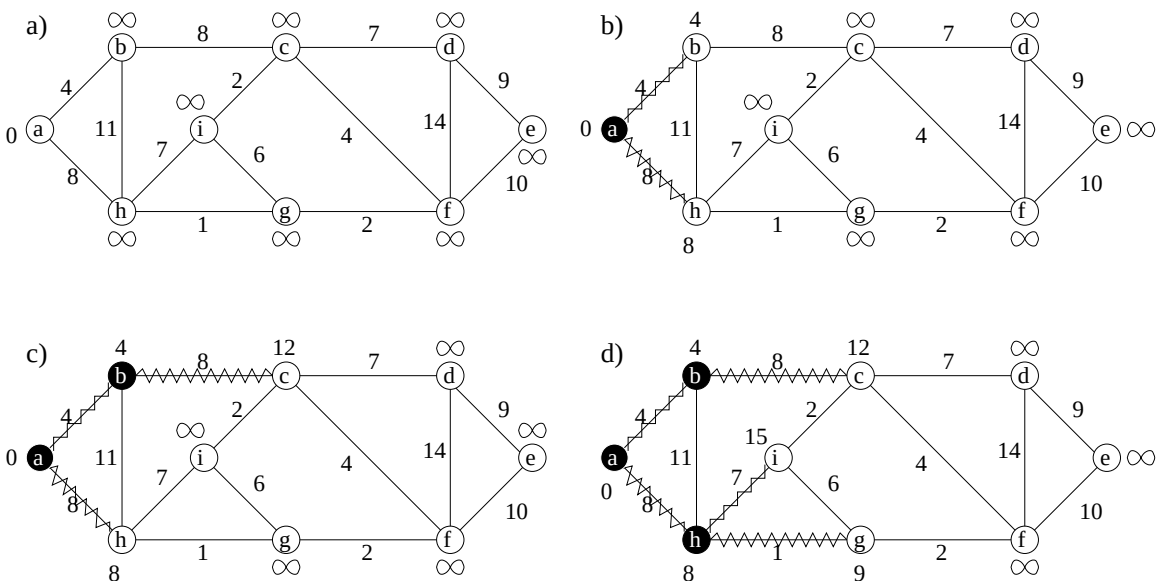

Figure 7.3: Dijkstra's Algorithm

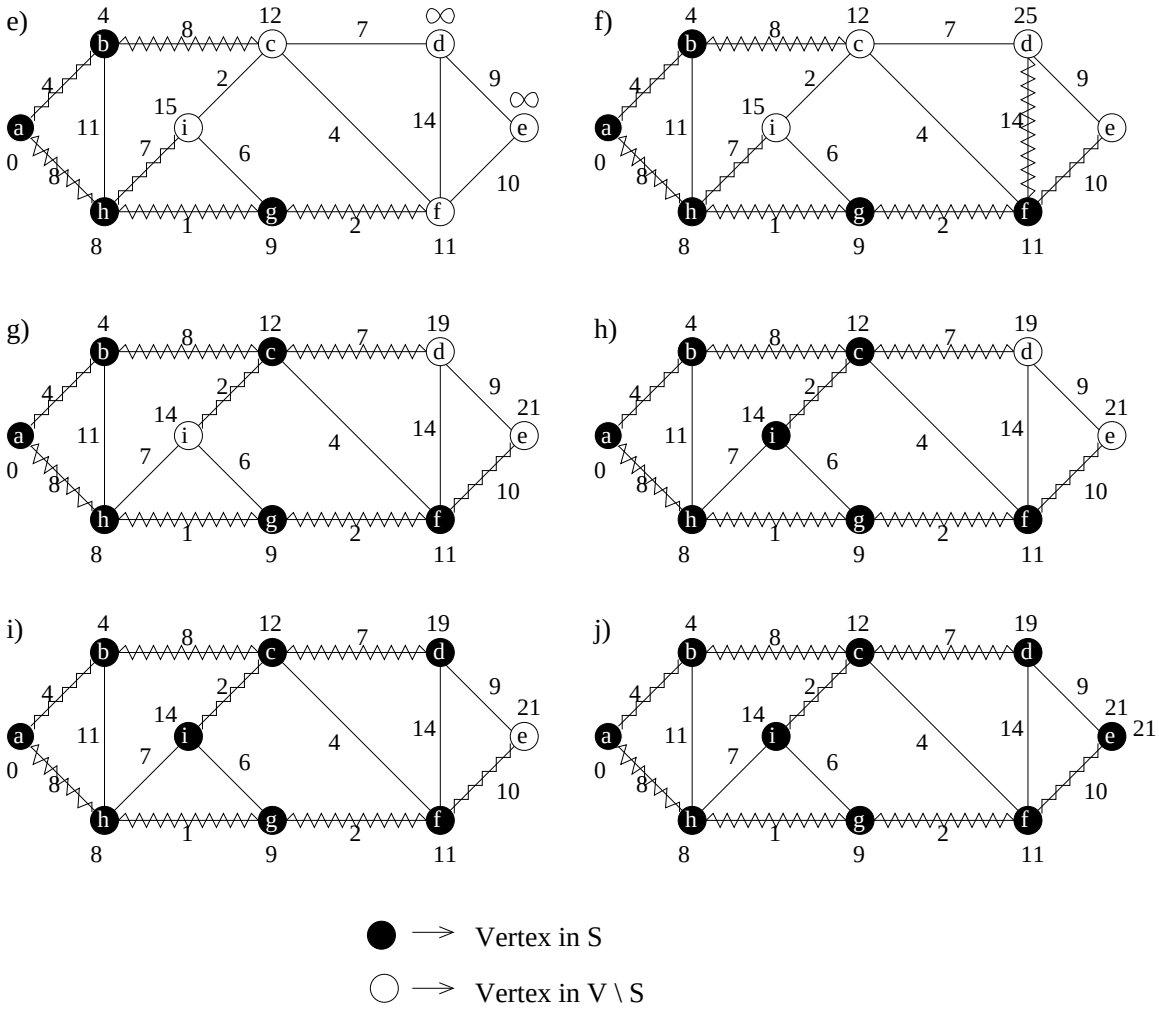

Figure 7.4: Dijkstra's Algorithm (contd.)

Dijkstra's Algorithm: Running-Time Analysis

- Extract-Min executed V times
- Decrease-Key executed E times

Running Time $= V \cdot T_{Extract-Min} + E \cdot T_{Decrease-Key}$

The running time depends on the implementation of Q. Here, are two bounds depending on the implementation. See [CLR] for the details.

Priority queue	TExtract-Min	TDecrease-Key	Total Time
array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\|E\| \log \|V\|)$

7.3 All pairs shortest path (APSP)

- In the APSP problem, we want to compute the shortest path between any two vertices $u, v \in V$. We will only consider graphs that have no negative weight edges.
  - The output has 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
  $\Longrightarrow \quad O(|V| \cdot |E| \log |V|)$ algorithm.
  - In the worst case (dense graph) the running time is $O(|V|^3 \log |V|)$.
- The Floyd-Warshall algorithm (see CLRS Exercise 25.2-4) runs in only $O(|V|^3)$ time 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 outer loop on k, the following *invariant* holds:
    After the kth iteration (out of $|V|$), $A[i,j]$ contains the length of the shortest path from v_i to v_j that (apart from v_i and v_j) only contains vertices of index at most k.
    $\Longrightarrow$ When algorithm terminates we have solved APSP.
  - **Proof:**
    * Invariant holds initially for $k = 0$ (i.e., we start with adjacency matrix A, and the only allowed path from v_i to v_j is the edge from v_i to v_j).
    * When "adding" vertex with index k, we explicitly check all new paths between v_i and v_j that pass through v_k, for all $|V|^2$ pairs of v_i and v_j.

    ■

- 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.