W4231: Analysis of Algorithms

10/28/1999 (revised 11/8/99)

- Shortest Paths
- Cuts and Flow

Weighted Graphs

Having in mind the shortest paths problem,

we want to consider graphs with *weights* associated to edges. Weights can model the cost or the time needed to go from one node to another or to connect them.

The weights are part of the instance. Can be fields in the adjacency lists or entries in the adjacency matrix.

The weight of an edge (u,v) is typically denoted by $w_{u,v}$.

Paths in weighted graphs

In a weighted graph (be it directed or undirected) the *length* of a path $v_1 v_2 \cdots v_k$

is the sum of the weights of the edges that make up the path $\sum_i w_{v_i,v_{i+1}}$.

Then the case of unweighted graphs can be seen as the special case of weighted graphs where all weights are 1.

Shortest Paths

Given a graph $G = (V, E)$ with non-negative weights on the edges, and two points s and t, we want to find the shortest path connecting s to t.

Generalizations:

1. **Single source shortest paths**. Given s, find for every $u \in V - \{s\}$ the shortest path between s and u.

2. **All pairs shortest paths**. Find the shortest path between any to vertices $u, v \in V$.

The fastest known algorithms for one pair also work for the single source problem, so we will consider only it and the all-pairs one. The all-pairs problem will have to wait until we introduce **dynamic programming**.

Single Source — General Idea

For starters, we show how to only compute the distances. General idea:

- The algorithm works in phases.

- At any phase, it maintains a vector $d[\cdot]$ such that $d[u]$ is always an **upper bound** on the distance between s and u.

- It also maintains a subset S of the vertices, with the property that $d[u]$ is the right distance from s to u for all the vertices u of S.

- At the end, S contains all the vertices, and so $d[\cdot]$ contains the right distances.

Beginning

$d[s] = 0$,

$d[v] = \infty$ for $v \neq s$.

S only contains s.

One Step

Find the vertex u of G not in S with the smallest value of $d[u]$. Include u in S.

For all the nodes $v \notin S$ such that there is an edge between u and v, update $d[]$ as follows:

$$d[v] := \min\{d[v], d[u] + w_{u,v}\}$$

Correctness

Induction on the number of iterations of the algorithm. Inductive step:

Consider the vertex v^* in the graph not in S which is closest to s.

The shortest path from s to v^* only uses elements of S (otherwise there would be some closer element to s outside S).

Therefore it is true that

$$dist(s, v^*) = \min_{u \in S, (u,v^*) \in E} \{dist(s, u) + w_{u,v^*}\}$$

But we know:

- For all vertices u of S, $dist(s, u) = d(u)$, by inductive hypothesis.
- For all vertices $v \notin S$,

$$\begin{aligned} dist(s, v) \leq d(v) &\leq \min_{u \in S, (u,v) \in E} \{d(u) + w_{u,v}\} \\ &= \min_{u \in S, (u,v) \in E} \{dist(u) + w_{u,v}\} \end{aligned}$$

So at the end of the current step, v is included in S and $d(v) = dist(u, v)$.

Constructing the Tree

- We also maintain a partial tree T all the elements of S. T is the tree of shortest paths from s to the elements of S.
- T is represented by means of a vector $P[]$ of predecessors.
- At the end, the tree reaches all the vertices, and $d[\cdot]$ contains the right distances.

Algorithm

```
ShortestPath(s)
  Initialize P[v] := ∅ for all v
  Initialize d[s] := 0 and d[v] = ∞ for v ≠ s
  CreateQueue(Q)
  Insert(v, Q) for all v ∈ V // The queue is keyed by d[]
  while Q is not empty do
    u := DeleteMin(Q)
    for all v such that (u, v) ∈ E do
      if d[v] > d[u] + w_{u,v} then
        d[v] := d[u] + w_{u,v}
        // Previous step requires a DecreseKey operation in the queue
        P[v] := u
```


Running Time

We do n `insert` and `deletemin` on the heap, and m `decrease-key`.

- Running time $O(n^2)$ is possible implementing the queue with an array. This dates back to Dijkstra (1959).
- $O((m+n)\log n)$ with a heap (1964).
- With Fibonacci heaps, total time is $O(m + n\log n)$. This is optimum for this algorithm in the comparison model [Fredman and Tarjan (1987)].

- If weights are integers that fit into one RAM word, then one can use asymptotically better priority queues. $O(m\sqrt{\log n})$, randomized algorithm using *fusion trees* [Fredman and Willard (1993)]. . . . Raman (1996) $O(m + n\sqrt{\log n \log\log n})$ deterministic.

  Departing from Dijkstra's algorithm, Thorup (1997) gives a $O(m+n)$ deterministic algorithm.

- If weights are integers less than U, then van Emde Boas (1977) gets $O(m \log\log U)$ . . . Raman (1996) randomized algorithm $O(n(\log U)^{1/4} + m)$.

Directed Acyclic Graphs

It is possible to compute single-source shortest paths in a directed acyclic graph in $O(m+n)$ time.

First do topological sort. Initialize $d[s] := 0$, $d[v] = \infty$.

Then: for all the u that are after s in the sort (respecting the order, starting from s); for all the v such that $(u,v) \in E$,

$d[v] := \min\{d[u] + w_{u,v}, d[v]\}$.

Max Flow

Network flow problems model situations where we want to send a commodity from a source to a destination using a network of transportation that has certain capacity limitations. The goal is to maximize the rate of shipment (regardless of delays).

"commodities" can also be packets to be transmitted across a network, or oil to be pumped through pipes.

The problem has a simple graph-theoretic definitions and very diverse applications.

Network

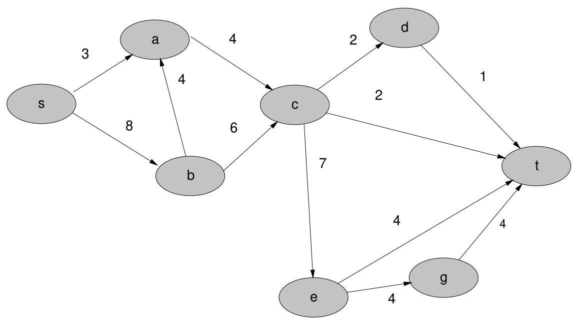
For our purposes a **network** is a **weighted directed** graph G with non-negative weights associated to the edges.

The weights are called **capacities**. For two vertices u and v, $c_{u,v}$ is the rate at which "shipment" or "communication" from u to v can happen. If there is no edge between u and v, we assume by convention that $c_{u,v} = 0$. Note that we can have $c_{u,v} \neq c_{v,u}$.

Two special vertices are called the “source” and the “sink” and denoted by s and t respectively. s has no incoming edge and t has no outgoing edge.

A network is totally described by G, by the capacities, and by the vertices s and t.

Example



Flow

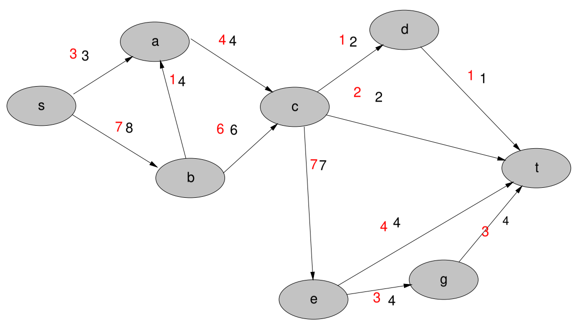
A **flow** is a specification of how much “shipment” do we have for each pair of adjacent vertices.

We can view a flow as a function $f : V \times V \to \mathbf{R}$.

A flow has to satisfy a few properties to be admissible:

- Never exceed the capacity: for every two vertices u, v, it must be $0 \leq f(u,v) \leq c_{u,v}$ [Capacity constraint];
- Never destroy nor create commodities in the intermediate nodes: for every v, $\sum_u f(u,v) = \sum_z f(v,z)$, unlesss $u = s$ or $u = t$. [Conservation constraint]

Example



Cost of the Flow

Note that $\sum_u f(s,u) = \sum_u f(u,t)$, i.e. the flow that gets out of s is equal to the flow that gets into t.

This is implied by the conservation constraint. The proof will come in a minute.

The value $\sum_u f(s,u) = \sum_u f(u,t)$ is called the cost of the flow, and is denoted by $cost(f)$.

Given a network we want to find the flow of maximum cost among those that satisfy the capacity constraints and the conservation constraints.

Cuts in Networks

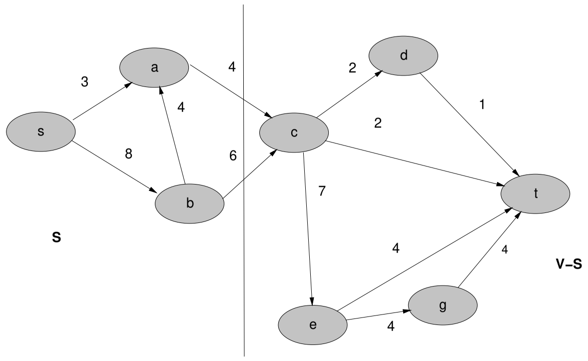
A **cut** in a network is a partition of the set of vertices into two subsets S and $V - S$, with the requirement that $s \in S$ and $t \in V - S$.

The **capacity** of a cut is

$$\sum_{u \in S} \sum_{v \in V-S} c_{u,v}$$

the total capacity of edges that go from a vertes in S to a vertex in $V - S$.

Example: a cut of capacity 10



Capacity of a Cut is an Upper Bound on Cost of a Flow

Whatever goes from s to t must pass through (some of) the edges in the cut, and the total amount of shipment cannot exceed the sum of capacities.

A Flow of Cost 10 is Optimal for Our Network