

6.046 Recitation 7 Handout

March 21, 2008

1 Shortest Paths Algorithms Review

Setting	Weights	Principle	Algorithm
Single source	unit	Greedy	BFS: $O(V + E)$
Single source	nonnegative	Greedy	Dijkstra: $O(E + V \lg V)$
Single source	general	$ V - 1$ passes	Bellman-Ford: $O(VE)$
All pairs	general	DP on length	Matrix mult: $O(V^3 \lg V)$
All pairs	general	DP on vertices	Floyd-Warshall: $O(V^3)$
All pairs	general	Reweigh	Johnson: $O(VE + V^2 \lg V)$

Reminder: Greedy approach does not work for general edge weights. Taking a path that begins with something that doesn't look as good now (i.e. is not the minimum adjacent edge) may lead to a very good (negative) edge later, so we need to take all those possibilities into account.

1.1 A Linear Program

Linear programs (LPs) are problems that can be written in the form of an objective function to be maximized (or minimized) and a set of constraints:

$$\begin{aligned} \max \quad & c \cdot x \\ & Ax \leq b \end{aligned}$$

where x is the vector of variables you are solving for, A is a constant matrix, and b and c are constant vectors. An optimal solution is a vector x for which the constraints hold and there is no way to improve the objective function, and a feasible solution is a vector x for which the value of the objective function is not optimal, but all the constraints are satisfied.

Single-source shortest paths can be cast as an LP using distances $d[v]$ to a particular vertex v as the variables:

$$\begin{aligned} \max \quad & \sum_{v \in V} d[v] \\ & d[v] - d[u] \leq w(u, v) \quad \forall (u, v) \in E \\ & d[s] = 0 \end{aligned}$$

The Bellman-Ford algorithm solves this linear program by setting all the distances to infinity, then decreasing them until all the constraints are satisfied. No distance is decreased more than it must be, so the objective function is satisfied over the constraints.

Side note: In general linear programs can be hard to solve. There is still no known strongly polynomial algorithm that solves it. Something that is strongly polynomial does not depend on the size of the input numbers. For example, merge sort is strongly polynomial because it only depends on n and not the size of the numbers it's sorting. In contrast, counting sort and radix sort are both weakly polynomial time algorithms because it depends on the size of the numbers (in terms of digits, etc.) it's sorting.

1.2 Floyd-Warshall Algorithm

Idea: Use dynamic programming by solving the subproblem of finding the shortest path using a subset of the vertices.

Define

$$a_{ij} = \begin{cases} w(i, j) & \text{if the edge } (i, j) \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

Also define $c_{ij}^{(k)}$ = weight of a shortest path from i to j with intermediate vertices (not including i and j) belonging to the set $\{1, 2, \dots, k\}$.

Thus the distance from i to j is going to be $\delta(i, j) = c_{ij}^{(n)}$, and we have the base cases of $c_{ij}^{(0)} = a_{ij}$.

Dynamic Program: Calculate

$$c_{ij}^{(k)} = \min \left(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \right)$$

which compares the *old* shortest path that doesn't include node k to a *new* one that is constructed out of the shortest paths to and from k . Easy to compute $c_{ij}^{(k)}$ if all $c_{ij}^{(k-1)}$ values have been calculated!

Pseudocode:

Floyd-Warshall:

```
for k <- 1 to n
  do for i <- 1 to n
    do for j <- 1 to n
      do if c_(ij) > c_(ik) + c_(kj)
        then c_(ij) <- c_(ik) + c_(kj)
```

(Superscripts can be ignored because the outer loop for k keeps calculations in order.)

1.3 Johnson's

Idea: Add height offset to each node so that we can adjust edge weights to be all nonnegative. We will use Bellman-Ford to calculate the new heights and then use Dijkstra's from each node to give us a running time of $O(VE + V(E + V \lg V))$ or $O(VE + V^2 \lg V)$.

We give each node u a height $h(u)$ and each edge (u, v) a new weight $w_h(u, v) = w(u, v) + h(u) - h(v)$. Any path from v_1 to v_k will have added weight of $h(v_1) - h(v_k)$ (internal offsets along the intermediate path will cancel each other out).

To make the new weights nonnegative, we can cast the problem as one of satisfying the type of constraints $h(v) - h(u) \leq w(u, v)$, which we saw earlier as part of a linear program and can be solved using Bellman-Ford.

2 Depth-First Search

Main idea: Explore as deeply in the graph as you can. Backtrack only when necessary.

Pseudocode for directed and undirected graphs:

DFS(G):

```
for each vertex u in V
  do color[u] <- white
time <- 0
for each vertex u in V
  do if color[u] = white
    then DFS-Visit(u)
```

```

DFS-Visit(u):
  color[u] <- gray
  d[u] <- time <- time + 1
  for each vertex v in Adj[u]
    do if color[v] = white
       then DFS-Visit(v)
  color[u] <- black
  f[u] <- time <- time + 1

```

Each color has a meaning:

- white - initial state; not yet visited
- gray - visited; not all neighbors visited and explored
- black - visited; all neighbors also visited and explored

Timestamps stored:

- $d[u]$ - discovery timestamps
- $f[u]$ - finishing timestamps

Running time analysis: DFS-Visit only gets called exactly once for each vertex. When called on a vertex v , DFS-Visit examines each outgoing edge from v . Summing over all vertices gets us $O(E)$ time. Thus DFS takes $O(V + E)$ time in total.

2.1 Edge Classification

We can classify edges based on colors of nodes when the edge is traversed.

1. Tree edge: encounter a new vertex: gray to white
2. Back edge: from descendant to ancestor: gray to gray
3. Forward edge: nontree edge from ancestor to descendant: gray to black
4. Cross edge: all other edges: gray to black

Cross edges can happen if for example there are a series of nodes 1, 2, and 3, there are directed edges $1 \rightarrow 2$, $1 \rightarrow 3$, and $3 \rightarrow 2$, and node 1 is visited first, coloring it gray. If node 2 is visited next, it gets colored black, and then node 3 is visited, coloring that gray. At that point, DFS-Visit attempts to visit node 2, so it goes from gray to black without visiting a descendant in the DFS.

2.2 Directed Acyclic Graphs (DAGs)

Definition: A DAG is a directed graph where there are no cycles.

Theorem: A directed graph is acyclic if and only if a DFS yields no back edges.

Proof: Clearly, a DAG will have no edges from a node to its ancestor or else there would be a cycle.

If a graph G does have a cycle, say we visit vertex u on this cycle first. The DFS from there must necessarily visit itself. The edge from the last node before reaching u again will be a back edge.

2.3 Topological Sort

A “sort” on a DAG $G = (V, E)$ that sorts the vertices in an order such that if there is an edge (u, v) in the graph, u appears before v in the order.

Topological-Sort(G):

 Call DFS(G)

 As each vertex finishes, insert it into front of a linked list

 Return the linked list

Example: Imagine a flow chart for classes and edges between them indicating whether a class has to be taken before another. For example, there is an edge from 6.042 to 6.046. You shouldn't find any cycles in this type of a graph since it would be hard to take 6.046 if you had to take 6.046 as a prerequisite. This is therefore a DAG. If we run Topological-Sort on it, we'll get an ordering of classes such that no class precedes a prerequisite in the list.

Correctness:

Claim: $(u, v) \in E$ implies $f[u] > f[v]$

Proof: When the edge (u, v) is explored, u must be gray. There are three cases for v :

- v is gray: implies (u, v) is a back edge; contradiction!
- v is white: v becomes descendant of u , so $f[v] < f[u]$
- v is black: v already finished before u , so $f[v] < f[u]$

Running Time: Essentially running time of DFS, or $O(V + E)$