COS 423 **Notes on Shortest Paths**
Spring 2009

1. The Labeling and Scanning Algorithm

The *labeling and scanning algorithm* for the single source shortest path problem maintains a *tentative distance* $d(v)$ and a *tentative parent* $p(v)$ for each vertex v. In addition, each vertex is (implicitly) in one of three states: *unlabeled*, *labeled*, and *scanned*. Initially $d(s) = 0$ and $d(v) = \infty$ for $v \neq s$, where s is the source vertex, and $p(v) = \text{null}$ for every vertex. We denote the set of labeled vertices by L; initially L={s}. The unlabeled vertices are exactly those with infinite distance; the scanned vertices are those with finite distance but not in L. The algorithm repeats the following step until L is empty:

Scan Step: Delete some vertex v from L. For each edge (v, w), if $d(w) > d(v) + c(v, w)$, replace $d(w)$ by $d(v) + c(v, w)$, replace $p(w)$ by v, and add w to L.

Each time a vertex is added to L, its tentative distance decreases. The algorithm maintains the invariant that if $d(v)$ is finite, there is a path from s to v of length $d(v)$. (Prove this by induction.) It also maintains the invariant (a) $d(v) \geq d(p(v)) + c(p(v), v)$. (Prove this by induction.)

We shall study the behavior of the algorithm by examining the structure of the graph defined by the parent pointers. This graph spans the vertices with finite distance (those labeled or scanned) and consists of at most one tree (rooted at s) and zero or more 1-trees. (A 1-*tree* is a tree with one additional edge, and exactly one cycle, such that every vertex in the tree is reachable from any vertex on the cycle. Such a graph is obtained from a tree by adding an edge from any vertex to the root; deleting any edge on the cycle of a 1-tree produces a tree.) Initially the graph consists of a single tree T rooted at s, containing just s. As scan steps are executed, T grows and changes. Suppose w is a vertex added to L as a result of the examination of an edge (v, w). If w was previously unlabeled, T grows by one vertex (w). If w was already in T, and v was not a descendant of w in T, then T is restructured: the subtree of T rooted at w is disconnected from the previous parent of w and reconnected to v. T remains a tree containing all the vertices with finite distance unless and until the third case occurs: w was already in T, and v was one of its descendants. In this case, changing the parent of w to v creates a cycle. This cycle must be negative, as the following argument shows: let l be the length of the path in T from w to v. Just before the update of $d(w)$ caused by the examination of (v, w), we have $d(w) > d(v) + c(v, w)$ and $d(v) \geq d(w) + l$; we obtain the latter inequality by summing invariant (a) over the edges on the path in T from w to v. Combining these inequalities gives $0 > c(v,w) + l$; the right side is the length of the cycle formed by making v the parent of w.

Thus the algorithm maintains a tree rooted at s that spans the vertices of finite distance until it creates a cycle, which is a negative cycle. Once this happens, the set of parent pointers need not define a tree, but the argument above can be extended to show that any cycle defined by parent pointers must be negative. (Show this.)

Lemma 1: If the labeling and scanning algorithm halts, then, for any vertex v, $d(v)$ is the actual distance from s to v (infinite if v is not reachable from s).

Proof: If v is not reachable from s, then $d(v)$ will remain infinite until the termination of the algorithm, since it cannot be set to a finite value without a path from s to v (of length the new value of $d(v)$) being discovered. Suppose v is reachable from s, and let p be a shortest path from s to v. One can show by induction on the number of edges on p that $d(v)$ is the length of p. (Verify this.)

Thus if the algorithm stops, it computes correct distances. If there is a negative cycle reachable from s, then any vertex on the cycle is not at finite distance from s, and the algorithm runs forever. One can show that, if there are no negative cycles, the algorithm stops after making at most 2^m additions to L. (Prove this.)

2. Efficient Scanning Orders

Depending on the properties of the graph and the edge lengths, different scan orders make the labeling and scanning algorithm efficient.

2.1 General Graphs and Lengths: Breadth-First Scanning

In the case of a general graph with possibly negative edge lengths, maintaining L as a queue gives good worst-case performance. Vertices are removed from the front of L and added to the back of L. There is some ambiguity in this rule. If a vertex w that is currently on L has its distance changed, we can either leave it in its current position in L or move it to the back. Since the former requires less updating than the latter, and since it simplifies the representation of L (explain this), the former seems preferable. But the analysis to follow applies to both variants, and indeed holds for the most general case in which the decision whether to move w or not is made arbitrarily each time the question arises.

We define *passes* over the queue as follows. Pass zero consists of the initial scan of s. For $k > 0$, pass k is defined inductively to consist of the scans of vertices added to the back of L during pass $k-1$. We can prove by induction on k that, for any vertex v, if there is a shortest path from s to v of k edges, then $d(v)$ will be the actual distance from s to v after pass $k-1$. (Show this.) Thus the algorithm terminates after at most $n-1$ passes, or else runs forever. With an appropriate, simple implementation the time per pass is $O(m)$. (Give such an implementation.) Thus the algorithm runs in

$O(nm)$ time if there are no negative cycles, or even if there are negative cycles if we count passes and stop after pass $n-2$.

If L remains non-empty after pass $n-2$, then the graph defined by the parent pointers contains a cycle (which must be negative) and retains this property henceforth; that is, it cannot revert to a tree. To prove this, we define a value $k(v)$ for each vertex v by initializing $k(v)=0$ for each vertex v, and setting $k(w)=k(v)+1$ each time $d(w)$ is updated *and* w *is added to the end of L*. With this definition, each time a vertex v is removed from the front of L, $k(v)$ is the number of the current pass. (Why?) Also, the algorithm maintains the invariant that $k(v)-1 \le k(p(v))$. (Why?) If vertex s is ever added to L a second time, it acquires a parent, and henceforth the graph of parent pointers must always contain a cycle. Otherwise, once pass $n-2$ is completed with L non-empty, there must always be a vertex v with $k(v) \ge n-1$ (since L must remain non-empty). Following parent pointers backward from v must eventually repeat a vertex and hence traverse a cycle. The only other possibility is that s is reached first, but $k(s)=0$ and each step back decreases k by at most 1, so a vertex must repeat before s can be reached.

Thus if we include a count of passes and terminate after pass $n-2$, we can extract a negative cycle merely by following parent pointers from any vertex still on L.

2.2 Eager Detection of Negative Cycles

Even though it is easy to extract a negative cycle once the breadth-first scanning algorithm exceeds its pass count bound, there are applications, such as currency arbitrage, in which the goal is to find a negative cycle rather than one or more shortest paths. For such an application we would like to stop the scanning process and return a negative cycle as soon as the algorithm creates one. There are two straightforward ways to do this, one of which is more obvious but less efficient than the other.

The high-level idea is to add a test after each tightening of an edge (v,w) (updating $d(w)$ as a result of examining (v,w)) of whether w is an ancestor of v in the tree defined by the parent pointers. The most obvious way to test this is to follow parent pointers from v until reaching w (giving a negative cycle) or reaching s (making v the parent of w does not create a cycle). The problem with this method is that it can take $\Omega(n)$ time per cycle test, and, indeed, one can construct examples for which breadth-first scanning augmented by this form of cycle-testing takes $\Omega(n^2 m)$ time, increasing the worst-case time bound by a factor of n.

Rather than starting from v and examining ancestors, it is better to start at w and examine descendants. That is, to test for a negative cycle we traverse the subtree rooted at w. If this subtree contains v, we have found a negative cycle. If not, making v the parent of w does not create a cycle. This method is better than searching through the ancestors of v, because, if a negative cycle is not detected we know that every proper

descendant of w has incorrect distance label (because there is a shorter path to it through the edge (v, w)), and we can pay for an unsuccessful search by disassembling the subtree rooted at w.

We need to augment the representation of the tentative shortest path tree, since parent pointers do not support traversal of subtrees. It would suffice to maintain the set of children of each node, but it is simpler, and more efficient, to store a list of the tree nodes in preorder. Since we need to delete nodes from this list, it seems that we need to make the list doubly-linked. (Or not? Explore this issue.) In addition, we need to be able to mark nodes as deleted from the tree. After the algorithm tightens an edge (v, w), if w is not marked, it marks w and initializes x to be the preorder successor of w. Then it repeats the following step until x is undefined or $p(x)$ is unmarked: if $x = v$, stop and report a negative cycle; otherwise, mark x, delete x from the preorder list, and replace x by its preorder successor. Finally, whether or not w was marked initially, it unmarks w and inserts w after v in the preorder list (deleting w from its previous position). The update of $d(w)$ also causes w to be added to L if it is not on L already.

Each step in a subtree traversal is paid for by the marking of a node; only one node (w) can be unmarked per tightening, so the subtree traversals are paid for by the edge tightenings. The algorithm maintains the invariant that the unmarked nodes form a tree rooted at s, the tentative shortest path tree. Instead of deleting each descendent x of w from the preorder list as it is visited, we can repair the preorder list after the traversal by making the final x the preorder successor of the original preorder predecessor of w, thereby excising all the marked vertices from the list. If we remember the predecessor of w, we can mark a node by setting its predecessor pointer to null.

It is useful in this algorithm to modify the scanning order. First, it is a good idea to delete vertices from L as they are marked, since scanning them when they have obsolete distance labels is not useful. The tradeoff is that this requires making L doubly linked. Or, when deleting a marked vertex from L we can ignore it instead of scanning it. A more drastic version of the same idea is to recompute the distance label of every vertex visited during a subtree traversal, decreasing it by the improvement in $d(w)$. Then we need to add a vertex w to L either when it gets a smaller distance label as a result of the tightening of an arc (v, w), or when w is marked and an arc (v, w) with $d(w) = d(v) + c(v, w)$ is examined. With this method, whenever a vertex is scanned, its best-available distance is used, and the overhead is still constant per tightening. (Both of these heuristics can change the scanning order, but not the $O(nm)$ bound for breadth-first scanning.)

Exercise: Verify the correctness of all variants of the algorithm. Explore efficient implementations, with the goal of minimizing both the time and space used. For example, if L is implemented as a ring (circular list), $x \in L$ if and only if it has a non-null L-pointer.

2.3 Shortest-First Scanning

If all arcs have non-negative lengths, a better scanning order is shortest-first: among vertices on L, delete one with minimum tentative distance. With this method, when a vertex is scanned its distance is correct, and each vertex is scanned only once. We can prove this by proving a stronger invariant: all scanned vertices have *d*-values no greater than those of all labeled vertices. This is certainly true initially, since there are no scanned vertices. Suppose it is true just before the scanning of a vertex v. Since v has minimum distance among all labeled vertices, making it scanned preserves the invariant. Suppose scanning v tightens an arc (v, w). Then $d(w) > d(v) + c(v,w) \geq d(v)$ before the tightening, which implies by the invariant (before the scan) that w is unlabeled or labeled, not scanned. After the tightening, w is labeled and $d(w) = d(v) + c(v, w) \geq d(v)$, so the invariant is true after the tightening. By induction, the invariant holds throughout the computation.

Shortest-first scanning was proposed by Dijkstra. To efficiently implement the algorithm, we represent L by a heap. There are at most n insert and n delete-min operations on L, and at most m decrease-key operations; tightening an arc results either in an insertion into L (if w is unlabeled) or a decrease-key (if w is labeled). The algorithm in Dijkstra's paper in effect represents the heap as an unordered set; each insertion or decrease-key takes $O(1)$ time, and each delete-min takes $O(n)$ time (via a traversal of the entire set), giving an overall time bound of $O(n^2)$. Using a classical heap implementation such as an implicit binary heap results in a time bound of $O(m \log n)$. Using a Fibonacci heap or an equivalently fast heap implementation such as a rank-pairing heap improves the running time to $O(n \log n + m)$, matching Dijkstra's bound on dense graphs and the bound for classical heaps on sparse graphs, and beating both asymptotically on graphs of intermediate density. The $O(n \log n + m)$ bound is best possible for any implementation of shortest-first scanning that uses binary comparisons, because every arc must be examined to obtain a correct solution, and one can reduce sorting n numbers to a run of shortest-first scanning on a sparse graph. The bound can be improved if one uses the power of random access, however.

Another thing worth noting is that the vertices are scanned in non-decreasing order by distance. This can be proved along with the invariant given above. That is, L forms a *monotone* heap in that successive deletions are of larger-and-larger elements. Monotone heaps are easier to implement than general heaps, at least if one uses bit manipulation or related techniques. See "hot queues", which give one way to beat the sorting lower bound.

2.4 Topological Scanning

If the graph contains no cycles, whether or not there are any negative arc lengths, a third scanning order is best: scan the vertices in topological order; that is, in an order such that if (v, w) is an arc, v occurs before w. A graph is acyclic if and only if it has a topological order. There are (at least) two algorithms for computing a toplogical order in linear time.

The first repeatedly deletes a vertex with no incoming arcs; the second uses depth-first search. Given a topological ordering, topological scanning will compute shortest paths from s in linear time. By taking the negatives of the arc lengths, the same method will compute longest paths. This is what has been called PERT analysis in the operations research literature.

3. All Pairs

One way to compute shortest paths for all pairs of source and destination vertices is to use dynamic programming, which results in a very simple $O(n^3)$-time algorithm called the Floyd-Warshall method. I shall describe a bare-bones version that just computes distances, not shortest paths; it can easily be augmented to compute paths. The method needs $O(n^2)$ space. Initialize $d(v,v) = 0$ for every vertex v; $d(v,w) = c(v,w)$ for every arc (v,w); $d(v,w) = \infty$ for $v \neq w$, (v,w) not an arc. Then perform the following triple loop: for each vertex v do for each vertex u do for each vertex w do if $d(u,w) > d(u,v) + d(v,w)$ replace $d(u,w)$ by $d(u,v) + d(v,w)$. The algorithm maintains the invariant (in the absence of negative cycles) that, after each execution of the outer loop, $d(u,w)$ is the length of a shortest path from u to w having as intermediate vertices only vertices for which the outer loop has been executed. Also, there is a negative cycle if and only if $d(v,v) < 0$ for some vertex v by the end of the algorithm.

Although this algorithm is very simple, it does not exploit sparsity very well. To exploit sparsity better, one can adapt Gaussian elimination to compute all-pairs shortest paths. First, one computes the equivalent of an LU factorization, and then, for each source, one does in effect two single-source shortest path computations on acyclic graphs, the equivalent of backsolving.

Another way to solve the all-pairs problem is to solve n single-source problems. If there are negative arc lengths (but no negative cycles), this method becomes much more efficient by solving one single-source problem with the original lengths and then using the computed distances to transform the arc lengths so that they are all non-negative, without affecting shortest paths. Specifically, for any real-valued function f on the vertices, define the *transformed length* $c'(v,w)$ of an arc (v,w) with respect to f to be $c(v,w) + f(v) - f(w)$. Observe that if p is any path from x to y, $c'(p) = c(p) + f(x) - f(y)$, since all the values of f on intermediate vertices along the path contribute to $c'(p)$ both positively and negatively, and cancel out. That is, the transformed cost of a path is the same as the original cost, plus a value that depends only on the source and destination vertices. This means that shortest paths remain shortest (and cycle lengths do not change). Having computed distances with respect to a set of transformed arc lengths, we can map each such distance back to the original lengths in constant time (by subtracting f of the source and adding f of the destination). Such a function f is in fact the set of dual variables in the linear-programming sense for the shortest path problem.

A function f that makes all transformed lengths non-negative (assuming no negative cycles) can be computed as follows. Assume all vertices are reachable from s. (If not, add a dummy source with zero-length arcs to all vertices.) Solve a single-source shortest path problem with source s, and let d be the resulting distance function. Then $d(w) \le d(v) + c(v,w)$ for every arc (v,w); that is, $c'(v,w) = c(v,w) + d(v) - d(w) \ge 0$. Thus d is a suitable f.

By doing one run of breadth-first scanning, then using the resulting distances to transform the arc costs to make them non-negative, and then doing n runs of shortest-first scanning, we can solve the all-pairs problem in $O(nm + n^2 \log n)$ time, which beats the Floyd-Warshall bound except for very dense graphs.

4. Single Pairs: Heuristic Search

One of the most important forms of the shortest path problem is the single-pair problem. Consider a situation in which we are given a huge graph, possibly only specified implicitly (an algorithm is given that will compute the arcs leaving a specified vertex, along with their lengths), and we wish to solve one or more single-pair problems on-line. In such a situation we would like to find a shortest path without exploring the entire graph, in sub-linear time. We may (or may not) be able to do some off-line preprocessing to compute some global information about the graph that helps to speed up distance queries. This is the "driving directions" problem: we are given the graph explicitly and queries come on-line; we can do preprocessing to speed up the queries. A different version of the problem, in which there is only a single query and the graph is given implicitly, was considered in the classical artificial intelligence literature many years ago and gave rise to an important idea, *heuristic search*.

In heuristic search, we are given some easy-to-compute estimate $e(v)$ of the distance from a given vertex to the destination. We use shortest-first scanning, but instead of scanning a labeled vertex with minimum $d(v)$ we scan a labeled vertex with minimum $d(v) + e(v)$: this is an estimate of the length of the shortest path from s to t that goes through v, and we prefer vertices estimated to be on short paths. As intuition, observe that if the e-values are exact distances and there are no ties, then heuristic search will scan exactly the vertices on the shortest path from s to t, and no others. In general the worse the estimate e, the more vertices will be scanned.

For arbitrary estimators e there is no guarantee that heuristic search has the same behavior as shortest-first search in that vertices are only scanned once. But we can place a natural condition on e that does guarantee this. The condition is another application of the arc-length transformation discussed in Section 3. Heuristic search is equivalent to shortest-first search using the transformed arc costs $c'(v,w) = c(v,w) - e(v) + e(w)$. (Verify this.) Thus if (*) $e(v) \le c(v,w) + e(w)$ for every arc (v,w), heuristic search will scan each vertex at most once, and we can stop the algorithm as soon as t is deleted from L, at which time its distance will be correct. We call an estimate that satisfies (*) a *safe*

estimate. We can assume that $e(t) = 0$; if not, we can use $e'(v) = e(v) - e(t)$ in place of e without affecting the behavior of the algorithm. Given $e(t) = 0$, if e is a safe estimate, then $e(v)$ is a lower bound on the actual distance from v to t. (Verify this.) The identically zero function is a (trivial) safe estimate. One can show that if e and f are safe estimates and $e(v) \geq f(v)$ for every vertex v, then heuristic search using e will scan a subset of the vertices scanned by heuristic search using f, if ties are broken in the same way. (Verify this.) If a non-safe estimate is used, one has a choice: stop when t is first deleted from L, resulting in a not-necessarily shortest path from s to t, or continue scanning. The heuristic search literature discusses the use of non-safe estimates as well as safe ones; the trade-off is that a non-safe estimate may result in fewer scans but a less accurate solution (if scanning stops when t is deleted from L). It is an interesting, challenging, and problem-dependent task to devise an accurate but easy-to-compute estimate for use in heuristic search. In the case of road networks, one proposed estimate is based on precomputing shortest paths to a collection of sentinal vertices; if u is a sentinal, $d(v,u) - d(t,u)$ is a safe estimator. (Verify this.) Sentinal-based estimates can be combined (how?); the more sentinals, the better the estimate, but the more costly the precomputation and the computation of the estimate.

Another useful technique in the single-pair problem is *bidirectional search*: search from s toward t and from t backward toward s concurrently, stopping when the two searches meet. One must take care with this method to get the stopping rule correct, especially in the case of bidirectional heuristic search.