1 Shortest paths (continued)

1.1 Properties of shortest paths

The Shortest Path Problem (henceforth SP problem) on a digraph $G = (V, A)$ involves finding the least cost path between two given nodes $s, t \in V$. Each arc $(i, j) \in A$ has a weight assigned to it, and the cost of a path is defined as the sum of the weights of all arcs in the path.

Proposition 1. *If the path $(s = i_1, i_2, \ldots, i_h = t)$ is a shortest path from node s to t, then the subpath $(s = i_1, i_2, \ldots, i_k)$ is a shortest path from source to node i_k, $\forall\ k = 2, 3, \ldots, h-1$. Furthermore, for all $k_1 < k_2$ the subpath $(i_{k_1}, i_{k_1+1}, \ldots, i_{k_2})$ is a shortest path from node i_{k_1} to i_{k_2}.*

Proof. The properties given above can be proven by a contradiction-type argument. □

This property is a rephrasing of the principle of optimality (first stated by Richard Bellman in 1952), that serves to justify dynamic programming algorithms.

Proposition 2. *Let the vector $\vec{d}$ represent the shortest path distances from the source node. Then*

1. *$d(j) \leq d(i) + cost(i, j),\ \forall\, (i, j) \in A$.*
2. *A directed path from s to k is a shortest path if and only if $d(j) = d(i) + cost(i, j),\ \forall\, (i, j) \in P$.*

The property given above is useful for backtracking and identifying the tree of shortest paths.

Given the distance vector $\vec{d}$, we call an arc *eligible* if $d(j) = d(i) + \text{cost}(i, j)$. We may find a (shortest) path from the source s to any other node by performing a breadth first search of eligible arcs. The graph of eligible arcs looks like a tree, plus some additional arcs in the case that the shortest paths are not unique. But we can choose one shortest path for each node so that the resulting graph of all shortest paths from s forms a tree.

Aside: When solving the shortest path problem by linear programming, a basic solution is a tree. This is because a basic solution is an in independent set of columns; and a cycle is a dependent set of columns (therefore "a basic solution cannot contain cycles").

Given and undirected graph with nonnegative weights. The following analogue algorithm solves the shortest paths problem:

The String Solution: Given a shortest path problem on an undirected graph with non-negative costs, imagine physically representing the vertices by small rings (labeled 1 to n), tied to other rings with string of length equaling the cost of each edge connecting the corresponding rings. That is, for each arc (i, j), connect ring i to ring j with string of length $\text{cost}(i, j)$.

Algorithm: To find the shortest path from vertex s to all other vertices, hold ring s and let the other nodes fall under the force of gravity.

Then, the distance, $d(k)$, that ring k falls represents the length of the shortest path from s to k. The strings that are taut correspond to eligible arcs, so this arcs will form the tree of shortest paths.

1.2 Alternative formulation for SP from s to t

We have already seen how to model the single source shortest path problem as a Minimum Cost Network Flow problem. Here we will give an alternative mathematical programming formulation for the SP problem.

$$\begin{array}{lcl} \min & & d(t) \\ d(j) & \leq & d(i) + c_{ij} \ \forall (i,j) \in A \\ d(s) & = & 0 \end{array}$$

Where the last constraint is needed as an anchor, since otherwise we would have an infinite number of solutions. To see this, observe that we can rewrite the first constraint as $d(j) - d(i) \leq c_{ij}$; thus, given any feasible solution $\mathbf{d}$, we can obtain an infinite number of feasible solutions (all with same objective value) by adding a constant to all entries of $\mathbf{d}$.

This alternative mathematical programming formulation is nothing else but the dual of the minimum-cost-network-flow-like mathematical programming formulation of the shortest path problem.

1.3 Shortest Paths in a DAG

We consider the single source shortest paths problem for a directed acyclic graph G.

We address the problem of finding the SP between nodes i and j, such that $i < j$, assuming the graph is topologically ordered to start with. It is beneficial to sort the graph in advance, since we can trivially determine that there is no path between i and j if $i > j$; in other words, we know that the only way to each node j is by using nodes with label less than j. Therefore, without loss of generality, we can assume that the source is labeled 1, since any node with a smaller label than the source is unreachable from the source and can thereby be removed.

Given a topological sorting of the nodes, with node 1 the source, the shortest path distances can be found by using the recurrence,

$$d(j) \ = \ \min_{\substack{(i,j)\in A \\ 1\leq i<j}} \{d(i) + c_{ij}\} .$$

The validity of the recurrence is easily established by induction on j. It also follows from property 2. The above recursive formula is referred to as a dynamic programming equation (or Bellman equation).

What is the complexity of this dynamic programming algorithm? It takes $O(m)$ operations to perform the topological sort, and $O(m)$ operations to calculate the distances via the given recurrence (we obviously compute the recurrence equation in increasing number of node labels). Therefore, the distances can be calculated in $O(m)$ time. Note that by keeping track of the i's that minimize the right hand side of the recurrence, we can easily backtrack to reconstruct the shortest paths.

It is important how we represent the **output** to the problem. If we want a list of all shortest paths, then just writing this output takes $O(n^2)$ (if we list for every node its shortest path). We can do better if we instead just output the shortest path tree (list for every node its predecessor). This way the output length is $O(m)$.

Longest Paths in DAGs:
If we want instead to find the longest path in a DAG, then one possible approach would be to

multiply by -1 all the arc costs and use our shortest path algorithm. However, recall that solving the shortest path problem in graphs with negative cost cycles is "a very hard problem". Nevertheless, if the graph where we want to find the longest path is a DAG, then it does not have cycles. Therefore by multiplying by -1 the arc costs we cannot create negative cycles. In conclusion, our strategy will indeed work to find the longest path in a DAG. Alternatively we can find the longest path from node 1 to every other node in the graph, simply by replacing *min* with *max* in the recurrence equations give above.

1.4 Applications of Shortest/Longest Path

Maximizing Rent

See the handout titled *Maximizing Rent: Example*.

The graph shown in Figure 1 has arcs going from the arrival day to the departure day with the bid written on the arcs. It is particularly important to have arcs with cost 0 between consecutive days (otherwise we would not allow the place to be empty for some days–which is NOT our objective, what we want is to maximize the rent obtained!). These arcs are shown as dotted arcs in the graph. To find the maximum rent plan one should find the longest path in this graph. The graph has a topological order and can be solved using the shortest path algorithm. (Note that the shortest and longest path problems are equivalent for acyclic networks, but dramatically different when there are cycles.)

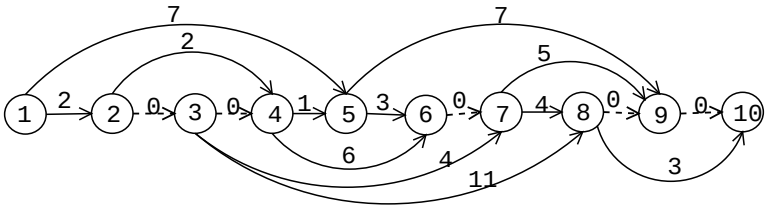

Figure 1: Maximizing rent example