Orf 467 Transportation Systems Analysis

Fall 2016/17

Class #12

Part 2: Planning and Analysis Tools of Transportation Demand and Investment

Networks:

Definitions and notation:

Directed graphs and Networks: A *directed graph*, G = (N, A) consists of a set of N Nodes and a set of A arcs whose elements are ordered pairs of distinct nodes. A *directed network* is a directed graph whose nodes and/or arcs have associated numerical values.
Undirected graphs and networks: A directed graph with arcs having unordered pairs of distinct nodes.
Tails and Heads: A directed arc (i,j) has two end points i and j. We refer to i ("A" node) as the tail node and j ("B" node) as the head node. Arc (i,j) *emanates* from i and *terminates* at j. Arc (i,j) is incident to nodes i and j. It is *outgoing* from i and *incoming* to j. Whenever an arc $(i,j) \in A$, then node j is adjacent to i.
Degrees: The *indegree* of a node is the number on incoming arcs to that node and the *outdegree* is the number of its outgoing arcs. The *degree* of a node is the sum of the in- and out- degrees.
Adjacency list: The *arc adjacency list*, A(i) of a node i is the set of arcs emanating from that node. $A(i) = \{ (i,j) \in A : j \in N \}$. The *node adjacency list*, N(i) is the set of nodes adjacent to that node; $N(i) = \{ j \in N: (i,j) \in A \}$
Multiarcrs and Loops: *Multiarcs* are two or more arcs having the same tail and head nodes. A *loop* is an arc whose tail node is the same as its head node.
Subgraph: A graph G' = (N', A') is a *subgraph* of G = (N,A) if $N' \subseteq N$ and $A' \subseteq A$. A graph G' = (N', A') is a *spanning subgraph* of G = (N,A) if N' = N and $A' \subseteq A$.
Walk: A *walk* in a directed graph is a subgraph of G containing a connected sequence of nodes (without any mention of arcs) or a connected sequence of arcs (without any mention of nodes).
Directed Walk: A *directed* walk is an "oriented" version of a walk in that any two consecutive nodes on the walk must be a member of the set of arcs $(i_k, i_{k+1}) \in A$.
Path: A path is a walk without any repetition of nodes. We can partition the arcs of a path into two groups : forward arcs and backward arcs. An arc (i,j) in the path is a *forward arc* if the path visits node i prior to visiting node j , and is a *backward arc* otherwise.
Directed Path: A *directed path* is a directed walk without any repetition of nodes. We can store a path easily by defining a *predecessor index*, *pred (j)* for every node j in the path.
Cycles: A *cycle* is a path $i_1 - i_2 - i_3 - \dots i_r - i_1$. Cycles can be directed.
Acyclic Graph: A graph is *acyclic* if it contains no cycles.
Connectivity: We will say that two nodes, i and j , are *connected* if the graph contains at least one path from node i to node j. A graph is *connected* if every pair of nodes is connected; otherwise it is *disconnected*.
Strong connectivity: A connected graph is strongly connected if it contains at least one *directed* path from every node to every other node.
Cut: A cut is a partition of the node set N into two parts, S and $\underline{S} = N - S$. Each cut defines a set of arcs consisting of those arcs that have one endpoint in S and the other in $\underline{S}$.
s – t Cut: This cut has node $s \in S$ ant $t \in \underline{S}$.
Tree: A *tree* is a connected graph that contains no cycles.
A tree of n nodes contains exactly n – 1 arcs.
A tree has at least two *leaf* nodes (nodes with degree 1)
Every two nodes of a tree are connected by a unique path.
Forest: A graph that contains no cycles is a *forest*. Alternatively, a forest is a collection of trees.
Subtree: A connected subgraph of a tree is a *subtree*.

Rooted trees: A rooted tree is a tree with a specially designated node called a *root*. Rooted trees are viewed as hanging from the root.
Directed-out-tree: every path from node s is a directed path

Network Flow Problems:

• **Minimum cost flow problem:**

Minimize $\sum_{(i,j) \in A} C_{ij} X_{ij}$

Subject to: $\sum_{\{j:(i,j) \in A\}} X_{ij} - \sum_{\{j:(j,i) \in A\}} X_{ji} = b(i)$
(flow in = flow out ; mass balance)

$l_{ij} \leq X_{ij} \leq u_{ij}$ for all $(i,j) \in A$
(bounded flow)

Can be rewritten in matrix notation as:
Minimize CX
Subject to: $\mathrm{N}\ X = b$
$l \leq X \leq u$
where N is an n x m node – arc incidence matrix for which each column of N_{ij} corresponds with the variable X_{ij}. It has a +1 in the i^{th} row, -1 in the j^{th} row; the rest of its entries are zero.

• Special cases:

Shortest Path: b(source) = 1 and b(sink) = -1 all others are zero.

Shortest Path Solution algorithms:

Label setting algorithms

• *Dijkstra's Algorithm* Magnanti Ch 4.4 pp 107
Finds the shortest path from the source node *s* to all other nodes in the network with non-negative arc lengths.

It maintains a distance label *d(i)* for each node *i*, which is the upper bound on the shortest path length to node *i*. At any intermediate step, it divides the nodes into two groups: those that are permanently labeled, *S* , and those that are temporarily labeled, $\underline{S}$. The distance to any node in *S* is the shortest from the source. The basic idea of Dijkstra is to fan out from the source, *s* , and permanently label nodes in the order of their distance from node *s*. Initially s is assigned a distance of zero and all others a very large distance. It selects node *i*, with the minimum temporary label, makes it permanent and scans out from that node, using A(*i*) (the node-arc adjacency matrix) to update the distance labels of adjacent nodes. The algorithm terminates when it has designated all nodes as permanent. The algorithm relies on the fact that we can always designate the node with minimum temporary label as permanent.

```
begin
    $S$ : = empty;   $\underline{S}$ : = $N$;
    $d(i)$ : = $\infty$ for each node $i \in N$;
    $d(s)$ : = 0 and $pred(s)$ : = 0;
while $|S| < n$ do
begin   (node selection operation)
 let $i \in \underline{S}$ be a node for which $d(i) = \min\{ d(j) : = j \in \underline{S}\}$;
 $S : = S \cup \{i\}$
 $\underline{S} : = \underline{S} - \{i\}$
 for each $(i,j) \in A(i)$ do  (distance update operation)
          if $d(j) > d(i) + C_{ij}$ then $d(j) = d(i) + C_{ij}$ and $pred(j) = i$;
 end;
end;
```

Running Times:

1. Node Selection: performed n times. Each time requires the scan of each temporarily labeled node: Worst case: $n + (n-1) + (n-2) + \ldots + 1 = O(n^2)$
2. Distance update: performed $|A(i)|$ times for the node i. Overall this is performed $\sum_{i \in N} |A(i)| = m$.

Since each distance update operation requires O(1) time, this step requires only O(<u>m</u>) total time for updating all of the distance labels.

Thus Dijkstra is of $O(n^2)$. Bottleneck is node selection.

Dial's Implementation: Magnanti Ch 4.6 pp 113

Focuses on node selection: realizes that if C is length of longest link, then max distance is nC. Then make nC+1 buckets and The contents of bucket k are *content(k).* Whenever we update the distance label of node i from d_1 to d_2 we move node i from *contents* (d_1) to *contents*(d_2). This can be done with a doubly-linked list (each cell has two links, one to the preceding cell and the other to the succeeding cell. Each "row" of the list 3 elements: *data*, *llink* (left link), *rlink* (right link)). We then need to only scan up to the first non-empty bucket and make permanent all contents, then perform the update operation. We use a doubly linked list to perform the operation.

Since no link is larger than C, then $d(j) = d(i) + c_{i,j} \leq d(i) + C$ Consequently, we actually only need C + 1 buckets, and we can arrange then cyclically.

Radix Heap Implementation: (one of many heap implementations) Magnanti Ch 4.7 pp 115 and 4.8 pp 116, also Ahuja

Use a Heap (or priority queue) data structure (for example store nodes of a heap as a rooted tree). The Radix Heap implementation modifies Dial's by using 1 + [log(nC)] buckets, numbered 0, 1, 2, 3, …, K .

- Each bucket has a *range( )*; *range(k)* is the range of bucket *k*.
- Temporary node *i* is stored in bucket *k* id $d(i) \in range(k)$.
- Permanent nodes are not stored. Let *content(k)* denote the nodes in bucket *k*.
- The algorithm changes the ranges of the buckets dynamically, when it does it redistributes the nodes in the buckets.
- The range of the Kth bucket is $[2^{K-1}, 2^K - 1]$. (widths are 1,1, 2, 4, 8, 16, ….). Whenever the algorithm finds nodes with a minimum distance label are in a bucket with range larger than 1, it examines all nodes in the bucket to identify the one with minimum distance label. Then the algorithm redistributes the bucket ranges and shifts each node in the bucket to the lower-indexed bucket. (Since the radix heap contains K buckets, a node can shift at most K times and consequently, the algorithm will examine will examine any node at most K times. Hence, the total number of examinations is $O(nK)$) .

Label-Correcting algorithms Magnanti Ch 5 pp 133

Label-correcting algorithms maintain a distance $d(j)$ label for every node $j \in N$. At intermediate stages of computations, the distance label $d(j)$ is an upper bound on the shortest path distance to the source node , and at termination it is the shortest path distance. *Pred(j)* are also saved at each stage. The necessary conditions for optimality are

$$d(j) \leq d(i) + C_{ij} \qquad \text{for all } (i,j) \in A$$

Algorithm *label-correcting*;
begin

```
    $d(s)$ : = 0 and $pred(s)$ : = 0;
$d(j)$ : = ∞ for each node j ∈ $N - \{s\}$;
        **while** some arc( i,j ) satisfies $d(j) > d(i) + C_{ij}$ **do**
            **begin**
                $d(j)$ : = $d(i) + C_{ij}$ and $pred(j) = i$;
 **end**;
**end**;
```

The key to label-correcting algorithms is to process "some arc (i,j)"!

Dequeue Implementation

Suppose we maintain a LIST of arcs that might violate the optimality condition. If LIST is empty, then we are optimal. Otherwise we examine LIST to select an arc, say *(i,j)*, violating it's optimality condition. We remove the arc from LIST and update *d(j)* and *pred(j)* if the optimality condition is violated. Any decrease in *d(j)* reduces the length of all paths passing through *j*. Therefore, if *d(j)* decreases, we must add arcs in *A(j)* to the set of LIST … we add all arcs emanating from *j* ! This suggests that LIST actually contains a list of nodes with the property that if an arc *(i,j)* violates the optimality condition, then LIST must contain node *j*. The question is where do we add the node to LIST, front or back? The "Pape" implementation uses a dequeue data structure that permits us to add and delete elements form the front as well as the rear of the LIST. The dequeue implementation always selects nodes from the front of the dequeue, but adds nodes to either the front or the rear. If the node has been in the LIST before, then it adds it to the front; otherwise it adds it to the rear.

Traffic Assignment:

- Constant link costs (linear networks):
  - "All-or-nothing" assignments
  - Multiple path assignments:
    - K-shortest paths
    - "Essentially-equal" shortest path
- Volume-dependent link costs
  - $\$_Q = \$_0 \{1 + a (Q / Q_{max})^b\}$

    where

    $\$_Q$ = link cost at traffic flow q

    $\$_0$ = "zero flow" link cost

    Q = traffic flow (veh/hr.)

    Q_{max} = practical capacity

    a , b are parameters

  - System-optimum assignments

User-optimum assignments