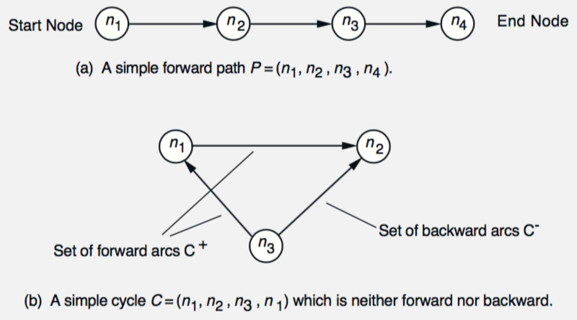
ECE 602 – Section 8
Network Optimization (Part 1)

- A *directed graph* $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ consists of a set of *nodes* $\mathcal{N}$ and a set of *arcs* $\mathcal{A}$.
- In general, an arc (i, j) is viewed as an ordered pair (i.e., *outgoing* from node i and *incoming* to node j).
- A graph is said to be *complete* if it contains all possible arcs.
- A *path* P is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ and a related sequence of $k - 1$ arcs such that the i-th arc is either (n_i, n_{i+1}) (*forward* arc) or (n_{i+1}, n_i) (*backward* arc).
- A path is called *simple* if it contains neither repeated arcs nor repeated nodes.

- A *cycle* is a path for which the start and end nodes are the same.
- A *Hamiltonian cycle* is a simple forward cycle containing all the nodes of $\mathcal{G}$.
- A graph that contains no simple cycles is said to be *acyclic*.



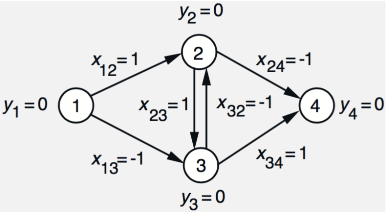
- $\mathcal{G}$ is *connected* if for each i and j, there is a path starting at i and ending at j. If such a path is forward, then $\mathcal{G}$ is *strongly connected*.
- $\mathcal{G}' = (\mathcal{N}', \mathcal{A}')$ is a *subgraph* of $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ if $\mathcal{N}' \subset \mathcal{N}$ and $\mathcal{A}' \subset \mathcal{A}$.
- A *tree* is a connected acyclic graph.
- A *spanning tree* of $\mathcal{G}$ is a subgraph of $\mathcal{G}$, which is a tree and includes all the nodes of $\mathcal{G}$.

- Given a graph $(\mathcal{N}, \mathcal{A})$, a set of *flows* $\{x_{ij} \mid (i,j) \in \mathcal{A}\}$ is referred to as a *flow vector*.
- The *divergence vector* y associated with a flow vector x is defined as

$$y_i = \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ji}, \qquad \forall i \in \mathcal{N}$$

- If $y_i > 0$, then i is a *source*. If $y_i < 0$, then i is a *sink*. If $y_i = 0$ for all i, then x is a *circulation*.
- Every divergence vector y must satisfy

$$\sum_{i\in\mathcal{N}} y_i = 0$$



- The *minimum cost flow problem* can be formulated as follows:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ji} = s_i, \quad \forall i \in \mathcal{N} \\
& l_{ij} \leq x_{ij} \leq u_{ij}, \quad \forall (i,j) \in \mathcal{A}
\end{aligned}
$$

- Here a_{ij} are *cost coefficients*, while l_{ij} and u_{ij} are *flow bounds*. Also, s_i (resp. $-s_i$) is referred to as the *supply* (resp. *demand*) of node i.

- The constraints are known as the *conservation of flow constraints* and the *capacity constraints*, respectively.

- If there exists at least one feasible flow vector, the minimum cost flow problem is called feasible.

Example: Shortest path problem

- Given a pair of nodes, the *shortest path problem* is to find a forward path that connects these nodes and has minimum cost (*path length*).
- The problem of finding the shortest path from node s to node t can be defined as:

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij} x_{ij}$$

$$\text{subject to} \quad \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ji} = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{if } i = t \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ij} \geq 0, \quad \forall (i,j) \in \mathcal{A}$$

- It can be shown that if this problem has an optimal solution, then the latter has the form of

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ belongs to } P \\ 0 & \text{otherwise} \end{cases}$$

with the corresponding path P being the shortest.

- The *assignment problem* consists in assigning n objects to n persons, with a_{ij} being a *value* for matching person i with object j.
- Our goal is to maximize the total benefit of the assignment.
- Any assignment can be associated with $\{x_{ij} \mid (i,j) \in A\}$, where $x_{ij} = 1$ if person i is assigned to object j and $x_{ij} = 0$ otherwise.
- The assignment problem can then be formulated as

$$
\begin{aligned}
\text{maximize} \quad & \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} = 1, \quad \forall i = 1,\ldots,n \\
& \sum_{\{i|(i,j)\in\mathcal{A}\}} x_{ij} = 1, \quad \forall j = 1,\ldots,n \\
& 0 \leq x_{ij} \leq 1, \quad \forall (i,j) \in \mathcal{A}
\end{aligned}
$$

- The optimal solution of the above ("relaxed") problem can be shown to satisfy $x^*_{ij} \in \{0,1\}$.

- In the *max-flow problem*, the objective is to move as much flow as possible from s (source) into t (sink).
- We want to find a flow vector that makes the divergence of all nodes other than s and t equal to 0 while maximizing the divergence of s.
- The problem can be formulated as follows:

$$
\begin{aligned}
&\text{maximize} \;\; x_{ts} \\
&\text{subject to} \quad \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j \mid (i,j)\in\mathcal{A}\}} x_{ji} = 0, \quad \forall i \in \mathcal{N}\backslash\{s,t\} \\
&\qquad\qquad\quad \sum_{\{j \mid (s,j)\in\mathcal{A}\}} x_{sj} = \sum_{\{i \mid (i,t)\in\mathcal{A}\}} x_{it} = x_{ts} \\
&\qquad\qquad\quad l_{ij} \le x_{ij} \le u_{ij}, \quad \forall (i,j) \in \mathcal{A} \text{ with } (i,j) \neq (t,s)
\end{aligned}
$$

where we introduced an artificial arc (t,s) with cost -1.

- At the optimum, the flow x_{ts} equals the maximum flow that can be sent from s to t (with the artificial arc (s,t) removed).

- A more general version of the minimum cost flow problem arises when the cost function is *convex* rather than linear.

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} f_{ij}(x_{ij}) \\ \text{subject to} \quad & \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ji} = s_i, \quad \forall i \in \mathcal{N} \\ & x_{ij} \in X_{ij}, \quad \forall (i,j) \in \mathcal{A} \end{aligned}$$

  where f_{ij} are convex functions and X_{ij} are convex intervals.

- This problem is commonly referred to as the *separable convex cost network flow problem*.

- More generally, such problems can be represented as

$$\begin{aligned} & \text{minimize} \;\; f(x) \\ & \text{subject to} \;\; x \in F \end{aligned}$$

  where F is a convex subset of flow vectors in a graph and f is a convex function over F.

- In the *matrix balancing problem*, the goal is to find an $m \times n$ matrix X that has given row and column sums, and is close to a given matrix M.
- Such a problem can be formulated in terms of a graph consisting of m sources and n sinks.
- In this case, $\mathcal{A}$ consists of the pairs (i, j) for which x_{ij} of X is allowed to be nonzero.

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} w_{ij}(x_{ij} - m_{ij})^2 \\
\text{subject to} \quad & \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} = r_i, \quad \forall i = 1, \ldots, m \\
& \sum_{\{i|(i,j)\in\mathcal{A}\}} x_{ij} = c_j, \quad \forall j = 1, \ldots, n
\end{aligned}$$

- Example: prediction of the distribution matrix X of telephone traffic between m origins and n destinations (based on historic data given by M).

- Objective: to find a minimum mileage/cost tour that visits each of N given cities exactly once and returns to the origin.
- Essentially, the problem is to find a Hamiltonian cycle with minimum sum of arc costs.
- Let $x_{ij} \in \{0, 1\}$ be the flow of arc (i, j), indicating whether or not it is part of the tour.
- Then, the problem can be formulated as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in\mathcal{T}} a_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j=1, j\neq i}^{N} x_{ij} = 1, \quad \forall i = 1, \ldots, N \\
& \sum_{i=1, i\neq j}^{N} x_{ij} = 1, \quad \forall j = 1, \ldots, N
\end{aligned}$$

and that the subgraph with node set $\mathcal{N}$ and arc set $\{(i, j) \mid x_{ij} = 1\}$ is connected.

- The last constraint makes the problem very difficult to solve.

- The *shortest path problem* is a classical and important combinatorial problem.

- Given a directed graph $\mathcal{G}$, the length of a forward path $(i_1, i_2, \ldots, i_k)$ is defined as

$$\sum_{n=1}^{k-1} a_{i_n i_{n+1}}$$

- The path is called *shortest* if it has minimum length over all forward paths with the same origin and destination nodes.

- The shortest path problem appears in a large variety of contexts (e.g., communication, routing in data networks, scheduling and sequencing, project management, paragraphing, etc.)

- We focus on algorithms for a single origin/all destinations problem.
- Many such algorithms maintain and adjust a vector $(d_1, \ldots, d_N)$, with d_j being the *label of node* j.
- Using the labels is motivated by the following optimality condition.

Proposition

Let $d_1, d_2, \ldots, d_N$ be scalars satisfying $d_j \leq d_i + a_{ij}$ for all $(i, j) \in \mathcal{A}$ and let P be a path starting at a node i_1 and ending at a node i_k. If $d_j = d_i + a_{ij}$ for all arcs $(i, j) \in P$, then P is a shortest path from i_1 to i_k.

- The above conditions are called the *complementary slackness (CS) conditions for the shortest path problem.*
- In fact, one can show that the scalars d_i are related to dual variables.

- A generic shortest path method consists in successively selecting (i, j) such that $d_j > d_i + a_{ij}$, and then setting $d_j := d_i + a_{ij}$.
- The iterations are continued until the CS condition $d_j \leq d_i + a_{ij}$ is satisfied for all arcs (i, j).

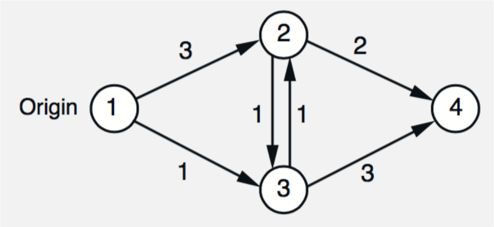
ALGORITHM: Generic shortest path algorithm

starting with $V = \{1\}$, $d_1 = 0$, $d_2 = \ldots = d_N = \infty$
repeat

1. Remove a node i from *the candidate list* V.
2. For each outgoing arc $(i, j) \in \mathcal{A}$: if $d_j > d_i + a_{ij}$, set $d_j := d_i + a_{ij}$.
3. Add j to V if it does not already belong to V.

until V is empty.

- Essentially, the method finds successively better paths from the origin to various destinations.



Iteration #	Candidate List V	Node Labels	Node out of V
1	$\{1\}$	$(0,\infty,\infty,\infty)$	1
2	$\{2,3\}$	$(0,3,1,\infty)$	2
3	$\{3,4\}$	$(0,3,1,5)$	3
4	$\{4,2\}$	$(0,2,1,4)$	4
5	$\{2\}$	$(0,2,1,4)$	2
	$\emptyset$	$(0,2,1,4)$	

- The algorithm terminates if and only if there is no path that starts at 1 and contains a cycle with negative length.
- The generic algorithm is guaranteed to terminate if: 1) all cycles have nonnegative lengths and 2) $\exists$ a path from node 1 to every node j.
- Upon termination, all labels d_j are equal to the corresponding shortest distances, and satisfy $d_1 = 0$ and

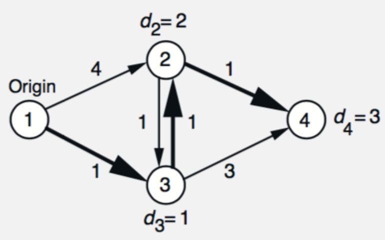
$$d_j = \min_{(i,j)\in\mathcal{A}} \{d_i + a_{i,j}\}, \quad \forall j \neq 1$$

- This is known as *Bellman's equation*.
- It means that if P_j is a shortest path from 1 to j, and $i \in P_j$, then the portion of P_j from 1 to i, is a shortest path from 1 to i.

ALGORITHM: Reconstructing the shortest path

starting with optimal labels $(d_1, d_2, \ldots, d_N)$
repeat

1. $\forall j \neq 1$, select (i, j) that attains minimum in $d_j = \min_{(i,j)\in\mathcal{A}}\{d_i + a_{ij}\}$.
2. Consider the subgraph consisting of the resulting $N-1$ arcs.
3. For any j, start from j and follow the corresponding arcs of the subgraph *backward* until node 1 is reached.



The above subgraph is known as *a shortest path spanning tree*.

- There are many implementations of the generic algorithm.
- They differ in how they select i to be removed from V.
- Broadly speaking, we have *label setting methods* and *label correcting methods*.
- There are several worst-case complexity bounds for both groups of methods.
- In practice, when the arc lengths are nonnegative, the best label setting methods and the best label correcting methods are competitive.
- As a general rule, a sparse graph favours the use of a label correcting over a label setting method
- Label correcting methods are more general, since they do not require nonnegativity of the arc lengths.

ALGORITHM: Dijkstra algorithm

starting with $V = \{1\}$, $d_1 = 0$, $d_2 = \ldots = d_N = \infty$
repeat

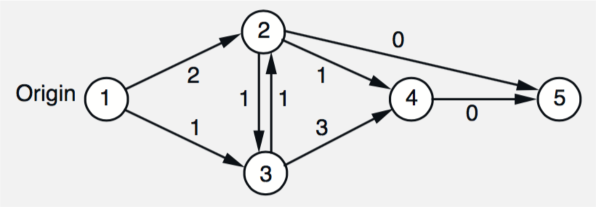
1. Remove from V a node i such that $d_i = \min_{j \in V} d_j$.
2. For each outgoing arc $(i, j) \in \mathcal{A}$: if $d_j > d_i + a_{ij}$, set $d_j := d_i + a_{ij}$.
3. Add j to V if it does not already belong to V.

until V is empty.

For any iteration of the algorithm and $W = \{i \mid i < \infty, i \notin V\}$, we have:

1. No node belonging to W at the start of the iteration will enter V during the iteration.
2. At the end of the iteration, we have $d_i \leq d_j$ for all $i \in W$ and $j \notin W$.
3. Assuming $a_{ij} \geq 0$, once a node enters W, it stays in W and its label does not change further (hence W is called a set of *permanently labeled nodes*).
4. The best estimates of the worst-case running time are $\mathcal{O}(A + N \log N)$ and $\mathcal{O}(A + N\sqrt{\log C})$, where C is the range of a_{ij}.

Example



Iteration #	Candidate List V	Node Labels	Node out of V
1	$\{1\}$	$(0, \infty, \infty, \infty, \infty)$	1
2	$\{2, 3\}$	$(0, 2, 1, \infty, \infty)$	3
3	$\{2, 4\}$	$(0, 2, 1, 4, \infty)$	2
4	$\{4, 5\}$	$(0, 2, 1, 3, 2)$	5
5	$\{4\}$	$(0, 2, 1, 3, 2)$	4
	$\varnothing$	$(0, 2, 1, 3, 2)$	

- Such methods use simpler rules for removal of the nodes from the candidate list V (hence less overhead).
- Yet, this is done at the expense of multiple entrances of nodes in V.
- All of these methods use some type of a *queue* to maintain V (in fact, the methods differ in the way the queue is structured).
- The simplest (Bellman-Ford) label correcting method uses a *first-in first-out* rule to update the queue.
- The running time of the Bellman-Ford method is $\mathcal{O}(NA)$.
- Alternative methods include the D'Esopo-Pape algorithm, the SLF and LLL algorithms, the threshold algorithm, and their variations.

- When using the label setting method, we can stop it when the destination t becomes permanently labeled.
- If t is closer to the origin than many other nodes, the saving in computation time will be significant.
- Another possibility is to use a *two-sided label setting method*.
- In this case, when a node gets permanently labeled from both sides, the labeling can stop.
- A shortest path can then be obtained by combining the forward and backward paths of each labeled node and by comparing the resulting origin-to-destination paths.
- Unfortunately, the approach does not work when there are multiple destinations.
- Some adaptations of label correcting methods are available as well.

- The algorithm maintains a path $P = ((s, i_1), (i_1, i_2), ..., (i_{k-1}, i_k))$ with no cycles, and modifies P using two operations, *extension* and *contraction*.
- If i_{k+1} is not on P and (i_k, i_{k+1}) is an arc, an *extension* of P by i_{k+1} replaces P by $((s, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_{k+1}))$.
- If P does not consist of just the origin node s, a *contraction* of P replaces P by $((s, i_1), (i_1, i_2), \ldots, (i_{k-2}, i_{k-1}))$.
- For each i, we introduce *the price p_i of node i*.
- The algorithm maintains a price vector p satisfying

$$p_i \leq a_{ij} + p_j, \qquad \text{for all arcs } (i, j)$$
$$p_i = a_{ij} + p_j, \qquad \text{for all arcs of } P$$

- It is equivalent to the CS condition, if p_i is viewed as the negative of d_i.

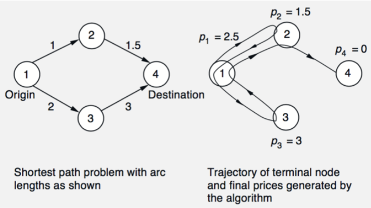
- We assume that $a_{ij} > 0$ and the initial pair (P, p) satisfies CS (e.g., $P = (s)$ and $p_i = 0$ for all i).
- We also assume that all cycles have positive length (can be relaxed).
- The algorithm iteratively transforms a pair (P, p) satisfying CS into another pair satisfying CS.

ALGORITHM: Auction algorithm

starting with (P, p) satisfying CS
repeat

1. Let i be the terminal node of P.
2. **if** $p_i < \min_{j|(i,j)\in\mathcal{A}}\{a_{ij} + p_j\}$
   set $p_i := \min_{\{j|(i,j)\in\mathcal{A}\}}\{a_{ij} + p_j\}$ and **contract** P (if $i \neq s$).
   else
   extend P by j_i where $j_i := \arg\min_{\{j|(i,j)\in\mathcal{A}\}}\{a_{ij} + p_j\}$.
   end

until j_i is the destination t.



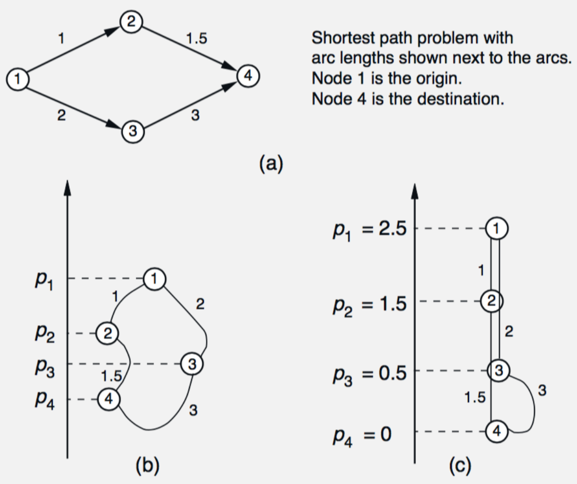
Iteration #	**Path P prior to iteration**	**Price vector p prior to iteration**	**Type of action during iteration**
1	(1)	(0, 0, 0, 0)	contraction at 1
2	(1)	(1, 0, 0, 0)	extension to 2
3	(1, 2)	(1, 0, 0, 0)	contraction at 2
4	(1)	(1, 1.5, 0, 0)	contraction at 1
5	(1)	(2, 1.5, 0, 0)	extension to 3
6	(1, 3)	(2, 1.5, 0, 0)	contraction at 3
7	(1)	(2, 1.5, 3, 0)	contraction at 1
8	(1)	(2.5, 1.5, 3, 0)	extension to 2
9	(1, 2)	(2.5, 1.5, 3, 0)	extension to 4
10	(1, 2, 4)	(2.5, 1.5, 3, 0)	stop

One can see that the terminal node traces the tree of shortest paths from s to the nodes that are closer to s than the given destination t.

Proposition

If there is at least one path from s to t, the auction algorithm terminates with a shortest path s to t. Otherwise the algorithm never terminates and $p_s \to \infty$.

- A drawback of the auction algorithm is that its running time depends on the arc lengths (particularly bad performance for graphs involving a cycle with relatively small length).
- It is possible to turn the algorithm into one that is polynomial by using an additional *reduction* operation.
- The reduction allows deleting some unnecessary arcs without affecting the shortest distance from s to t.
- Running the algorithm until every destination becomes the terminal node of the path allows dealing with the case of multiple destinations (single origin).



- In **(b):** The CS condition $p_i - p_j \leq a_{ij}$ clearly holds for all (i, j).
- In **(c):** We have $p_i - p_j = a_{ij}$ for all the “tight strings".

- Consider the *all-pairs* shortest path problem.
- Starting with

$$D_{ij}^0 = \begin{cases} a_{i,j} & \text{if } (i,j) \in \mathcal{A} \\ \infty & \text{otherwise} \end{cases}$$

for each i and j and each $k = 0, 1, \ldots, N-1$, generate sequentially

$$D_{ij}^{k+1} = \begin{cases} \min\{D_{ij}^k, D_{i(k+1)}^k + D_{(k+1)j}^k\} & \text{if } i \neq j \\ \infty & \text{otherwise} \end{cases}$$

- D_{ij}^k gives the shortest distance from i to j using only nodes from 1 to k as intermediate nodes.
- Thus, D_{ij}^N gives the shortest distance from i to j (total running time is $\mathcal{O}(N^3)$).
- Unfortunately, the Floyd-Warshall algorithm cannot take advantage of sparsity of the graph.
- In such a case, it is typically better to apply a single origin/all destinations algorithm separately for each origin.