MATH 566 – Lecture 1 (08/26/2014)

This is Math 466/566 – Network Optimization.

I'm Bala Krishnamoorthy. Call me 'Bala'.

Check out the syllabus on the course web page.

A field that is closely related to network optimization is graph theory. While we will use several results that are also typically presented in a similar graph theory course, we will concentrate more on the "network" aspects.

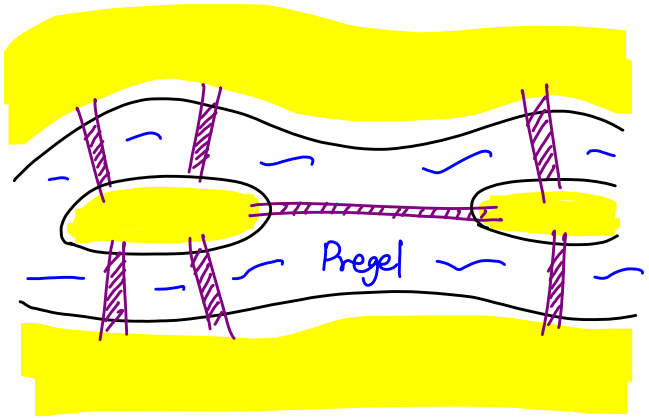
One key difference between "graphs" and "networks", as used conventionally, is that the arcs (or edges) in networks have capacities. In other words, one could think of them as pipes with limits on how much fluid could be sent through them at a time. On the other hand, edges in "graphs" are typically "lines" drawn between the points representing the nodes (or vertices). Thus, we will adopt a "flow" point of view.

Another difference between our approach and that of a typical graph theory class is that we will emphasize efficient algorithms to solve the optimization problems on networks.

We will also explore numerous applications from various fields, including science, engineering, and business, of the different network problems we will study.

The Königsberg problem

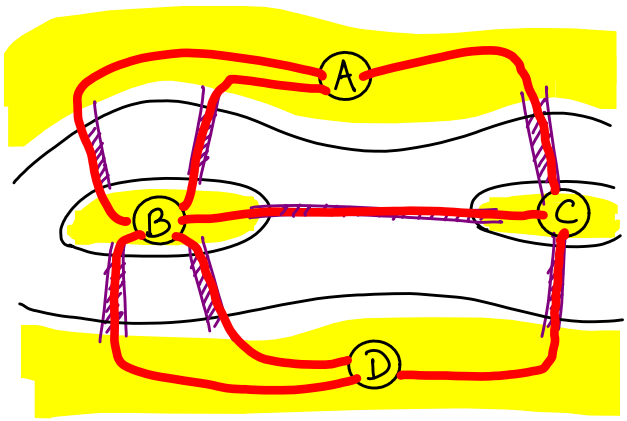
We start will a historical note. The first paper in graph theory (and, by extension, in network optimization) was published by Euler in 1736. He was visiting Königsberg, near Warsaw in Poland. The river Pregel flows through the town, and had seven bridges across it.

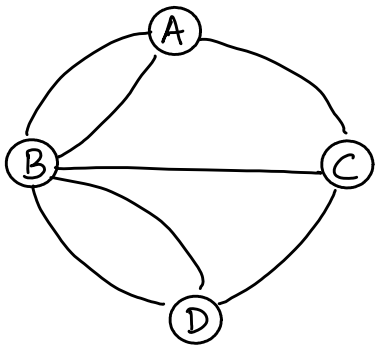


The citizens asked him if one could cross each of the seven bridges exactly once, and return to the starting point?

It was generally considered impossible to do. Euler characterized when it would be possible.

Here is a model for this problem. We represent each land mass by a node, and each bridge by an edge connecting the nodes representing the two land masses in question.





graph

We can restate the question as follows. In the graph, is it possible to start at A, say, traverse each edge exactly once, and return to A?

Since we return to where we started, such a "walk" is a cycle. Such a cycle, which traverses each edge exactly once, is called an Eulerian cycle. Hence we can ask "Does there exist an Eulerian cycle in the graph"?

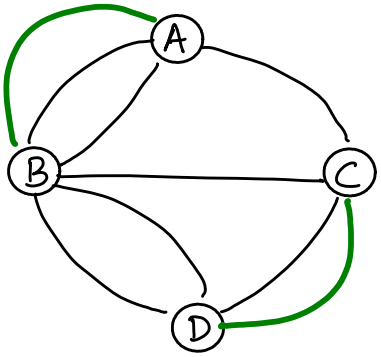
Theorem An undirected graph has an Eulerian cycle iff

* every node has an even degree, and
* the graph is connected, i.e., every pair of nodes has a "path" between them.

The degree of a node is the number of edges connected to it. We will not get into the details of this result right now. But one could get the intuition behind the even degree requirement. We should be able to "come in" to a node on an edge, and "go out" on another edge to a different node.

Many results (theorems) we present in this course will have a similar nature – intuitive to grasp and understand, and easy to state (may be not always easy to prove ☺)!

In the present case, if there were two more bridges as shown here, there would exist an Eulerian cycle.



Notice that all nodes have even degrees now (A, C, D-4, B-6).

A Related problem (Hamiltonian cycle problem): Does there exist a "cycle" that visits every node exactly once? This is the traveling salesman problem (TSP).

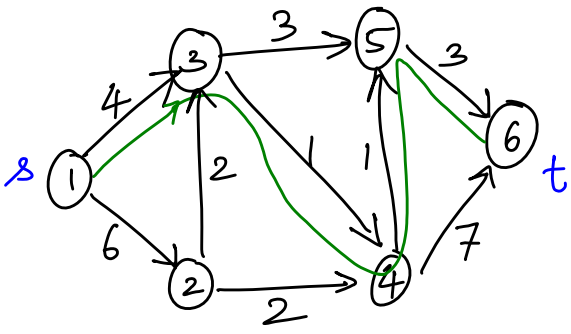
There are many applications of the TSP. It is one of the most well-studied network optimization problems. One typical application is in chip design, where the robotic arm printing the circuit pattern on the chip has to be programmed to finish printing the entire circuit in an optimal fashion in order to minimize the time spent.

We will now introduce several network optimization problems, which we will study in detail.

Network flow problems

① Shortest path problem

objective: find a path of minimum cost (length) from an origin (or source) node s to a destination (or sink) node t in a network with node set N and **directed** are set A.



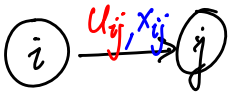
notation:

the shortest path is indicated in green here

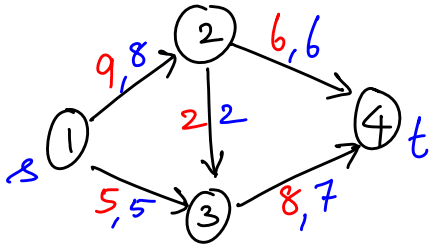
A typical application is driving directions on Google Maps. The costs c_{ij} could just be the distances between the cities modeled by the nodes, or could capture other relevant info such as traffic, tolls, etc.

(2) Maximum flow problem (or max-flow)

Find a "feasible" solution that sends maximum flow from a source node s to a sink node t, while honoring the capacities of the arcs.



Example



The maximum flow sends 13 units from s to t here.

Typical examples include traffic flow management – in road transportation networks or in computer networks.

Notice that we are working with directed arcs (or edges). This will be the default mode going onward, unless mentioned otherwise.

③ Minimum cost flow problem (min cost flow problem)

Objective: determine a least cost shipment of commodity through a network in order to satisfy the demands at certain nodes using supply available at other nodes, while honoring capacities of the arcs.

Typical example: shipping of a commodity from warehouses to retailers.

We now introduce standard notation and formulate the network flow problems as optimization problems.

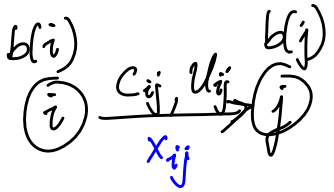
Notation

$G=(N,A)$ → directed graph, N: set of nodes, A: set of arcs.

G for "graph".

$|N|=n$ (# nodes)
$|A|=m$ (# arcs)

c_{ij}: unit cost on arc (i,j)
u_{ij}: capacity of arc (i,j)



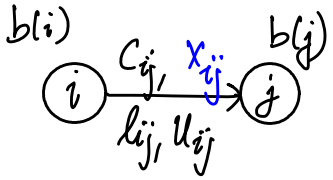
$b(i)$: supply/demand at node i.

$>0 \Rightarrow$ supply
$<0 \Rightarrow$ demand (demand of $-b(i)$ at that node)
$=0 \Rightarrow$ transshipment node.
↳ everything that comes in goes out.

MATH 566 – Lecture 2 (08/28/2014)

Today: * mathematical model for min cost flow
- shortest path and max flow as special cases
* two more problem classes
* modeling applications.

Recall notation: $G=(N,A)$, $|N|=n$, $|A|=m$



Here is the optimization model for min-cost flow:

minimize

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \quad \text{(total cost)} \qquad (1)$$

subject to

s.t.

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b(i) \quad \forall\, i \in N \qquad (2)$$

for all.

outflow − inflow = supply/demand

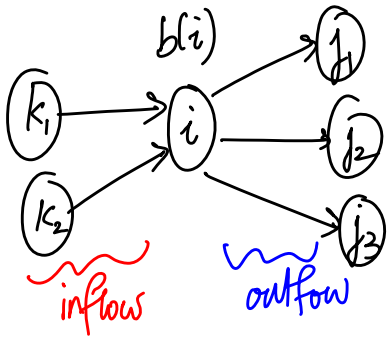
$$l_{ij} \le x_{ij} \le u_{ij} \quad \forall\, (i,j) \in A \qquad (3)$$

Mathematical notation
- $\in$: "element of"
- $\forall$: "for all"
- $\sum$: sum

In words, we want to minimize the total cost (1), subject to meeting the supply/demand constraints at each node (2), and the bounds on flow in each arc (3).

(1) models the total cost incurred. On arc (i,j), one unit of flow incurs a cost of c_{ij}. Hence x_{ij} units incur $c_{ij}x_{ij}$. Summing up all such cost terms (over all arcs in the arc set A) gives the total cost. The goal is to find a flow, i.e., x_{ij} values, that minimizes the total cost.

The flow must satisfy constraints (2) and (3). (2) are the flow balance constraints, which you could think of as "conservation of flow (or mass)."



There is no flow "lost" or "appearing out of the blue". In words, "what comes in + what is produced or used = what goes out".

The flow must honor the lower and upper bounds on each arc, as specified by constraints (3). The default value for all l_{ij} is zero, and all u_{ij} is $+\infty$ (or, some large number).

We also assume here that $\sum_{i \in N} b(i) = 0$, i.e., total supply is equal to total demand. There are generalizations where this condition might not hold, when (2) may not be written as '=' for all i.

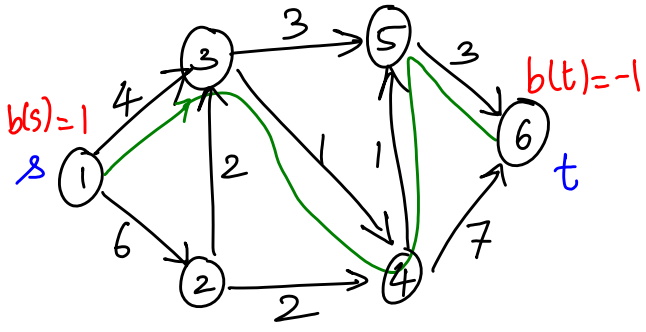
This model for min cost flows is an instance of linear optimization (or linear programming). This problem is one of the most general network flow problems. While we write it down as a linear optimization problem, we will not solve it as so – most network flow problems are easier than general linear optimization problems. We will discuss special algorithms that work more efficiently on these problems (than on linear optimization problems).

We now show that the shortest path and the max flow problems are special cases of the min cost flow problem.

Shortest path as a min-cost flow problem

When we model an instance of mincost flow, we need to specify the network, i.e., what the nodes and arcs are, and then specify all associated parameters – c_{ij}, l_{ij}, u_{ij}, for arcs (i,j), and $b(i)$ for nodes i.

Consider the example for shortest path from Lecture 1.



In general, we set

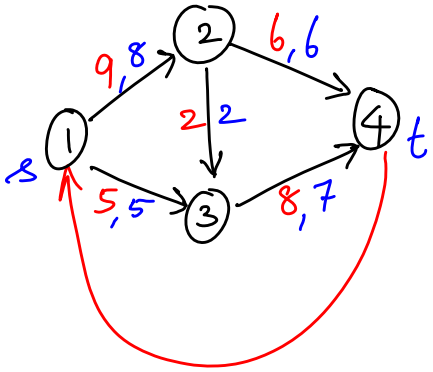
$b(s)=+1$, $b(t)=-1$, $b(i)=0$ otherwise,

and $l_{ij}=0$, $u_{ij}\geq 1$ for all (i,j).

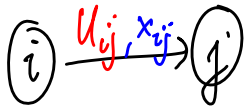
The c_{ij}'s are already given. Finding the shortest path from s to t is equivalent to sending one unit of flow (think one car, for instance) from s to t at the least cost.

Max-flow as min-cost flow problem

In the previous instance (shortest path), we did not have to add any extra nodes or arcs. To model the max flow problem as a min cost flow problem, we add a single extra arc.



$c_{ts}=-1$, $u_{ts}=\infty$



In general, we add arc (t,s) with $c_{ts}=-1$, $u_{ts}=\infty$. We set all $b(i)=0$, and $c_{ij}=0$ for the original network.

Since $c_{ts}=-1$ (we could set it to any value <0, while keeping all other $c_{ij}=0$), min cost flow will try to send as much flow as possible on arc (t,s). And all flow on (t,s) enters node s. Further, this flow is also equal to the net flow out of node t. In other words, the flow on (t,s) is indeed the maximum flow from s to t. Since $u_{ts}=\infty$, the value of the max flow is determined by the u_{ij} values of the original network, as it should be.

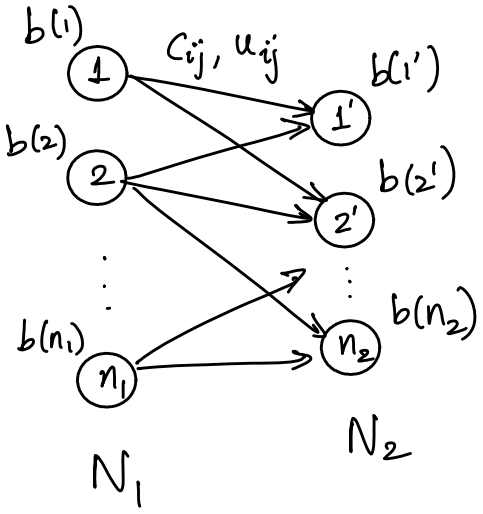
Notice that the flow just circulates around the network here. There is no flow coming in or going out at any of the nodes. This is an instance of the **circulation problem**, which is the min cost flow problem with $b(i)=0$ for all nodes i. This is the 4th network flow problem class (after shortest path, max flow and min cost flow).

→ min cost circulation, to be exact.

A typical example of the circulation problem is the scheduling of airplanes operated by a firm, e.g., Alaska Airlines. Each city/destination served is a node, and the total number of planes remains same within the network. If a service is required between, say, Seattle and Spokane, the l_{ij} for that arc is set to 1, ensuring one plane flies from Seattle to Spokane.

(5) Transportation problem

This is a special case of mincost flow where the node set $N = N_1 \cup N_2$, ("union") with N_1 being supply nodes and N_2 being demand nodes. Hence, $b(i) > 0 \ \forall i \in N_1$, and $b(j) < 0 \ \forall j \in N_2$. Further, all arcs $(i,j) \in A$ have $i \in N_1$ and $j \in N_2$. The arcs have c_{ij}, l_{ij}, and u_{ij} specified.



Notice that there are no arcs within N_1 or within N_2.

We assume that $\sum_{i \in N_1} b(i) = -\sum_{j \in N_2} b(j)$,

i.e., total supply = total demand.

Such an instance is a **balanced transportation problem**. In the unbalanced case, there might be penalties for unmet demand or for unused supply.

The default example is that of shipping of a commodity between warehouses and retailers, with the former set forming the supply nodes, and the retailers forming the demand nodes. If there is an option to ship the commodity from warehouse i to retailer j, arc (i,j) is included in the network. For instance, if there is a truck available to ship items from the Seattle warehouse to the Vancouver store; the capacity and cost associated with the truck are modeled as u_{ij} (and l_{ij}, if there is a minimum), and c_{ij}.

Read section 1.3 on modeling applications as various network flow problems. We will discuss one such example now.

The seat sharing problem

→ Ahuja, Magnanti, Orlin (the text)

(AMO 1.8, page 21) Several families are planning a shared car trip on scenic drives in the Cascades in Washington. To minimize the possibility of any quarrels, the organizers of the trip want to assign individuals to cars so that **no two members of a family are in the same car**. Formulate this problem as a network flow problem.

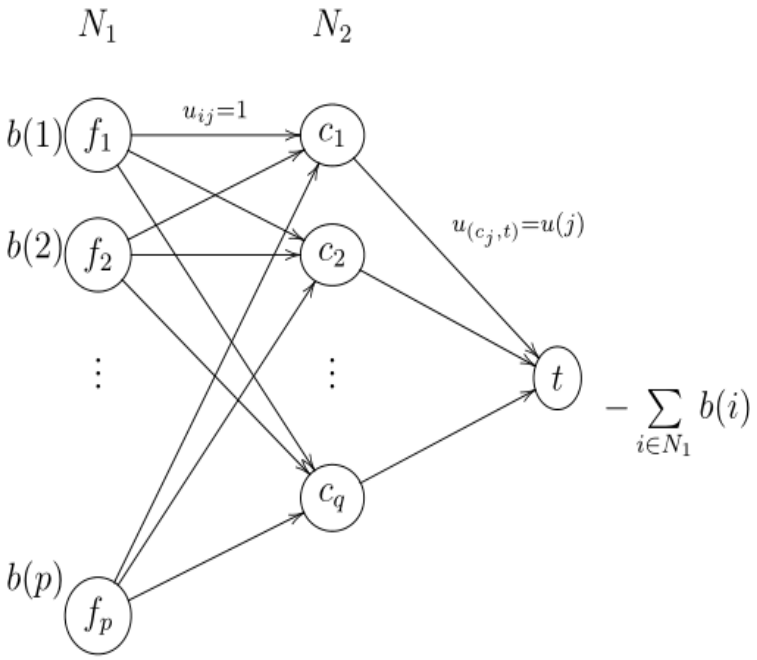
We need to specify the network, and also all the associated data, i.e., $G=(N,A)$, $b(i)$, c_{ij}, l_{ij}, u_{ij}.

We could model each family as a supply node, supplying as many people as there are family members. Similarly, each car could be a demand node, demanding as many people as there are seats.

The details are reproduced here from the handout posted on the course web page.

Assume that there are p families, and that the family i has $b(i)$ members (for $1 \leq i \leq p$). Also, assume that there are q cars, and that car j can hold up to $u(j)$ persons (for $1 \leq j \leq q$). To start with, assume that there is enough room to accommodate all families, i.e., $\sum_{1 \leq j \leq q} u(j) \geq \sum_{1 \leq i \leq p} b(i)$.
A setup similar to that of the transportation problem can be imagined here, with two subsets of nodes – the **family** nodes being the supply nodes ($N_1 = \{f_1, \ldots, f_p\}$), and the **car** nodes being the demand nodes ($N_2 = \{c_1, \cdots, c_q\}$). The supply on the family node f_i is $b(i)$. To model the restriction that no two persons from one family should be assigned to the same car, we draw one arc from each family node f_i to every car node c_j with upper bound $u_{ij} = 1$. Thus, there will be a total of pq such arcs.

One could put a demand of $-u(j)$ on the car node c_j if $\sum_{j \in N_2} u(j) = \sum_{i \in N_1} b(i)$, and the model will be complete. The more realistic situation occurs when there are more seats available in the cars than there are people to be accommodated, and hence we may not be able to satisfy certain demands completely (in other words, certain cars will have unfilled seats). To model this situation, the car nodes are set as transshipment nodes. We add another node t and draw arcs from each car node c_j to t with an upper bound of $u(j)$. Then we put $b(t) = -\sum_{i \in N_1} b(i)$ to complete the network flow model. This construction makes sure that we can assign up to (or possibly less than) $u(j)$ persons in car c_j, and at the same time, all the family members are assigned to some car.



MATH 566 – Lecture 3 (09/02/2014)

To describe a network flow problem formulation, you must describe the network structure, i.e., what the nodes and arcs are. You must also specify the parameter values $b(i)$, l_{ij}, u_{ij}, and c_{ij}. The default values can be taken as $b(i)=0$, $l_{ij}=0$, $u_{ij}=\infty$, $c_{ij}=0$. Thus, you could mention just the non-default values.

More on the seat sharing problem

The max # of people seated in car C_j from family $f_i = 1$.
Model this situation by setting $u_{ij}=1$ for arc (f_i, C_j).

More generally, car C_j can set at most u_j people.
Model this situation by setting $u_{\{C_j,t\}} = u_j$ for arc (C_j, t).

In general, max limits (or bounds) are typically modeled by setting them as upper bounds on the appropriate arcs. Similarly, lower bounds could be used to ensure that a commodity, say, is used at least to certain extent.

Today:
* Definitions
* Network representations

Definitions for Networks

We have seen some of them informally, today we will list them formally.

$G = (N, A)$ is the **network** with node set N and arc set A.

"network" and "graph" used interchangeably

$|N| = n$ (# nodes) $\quad N = \{1, 2, \ldots, n\}$

$|A| = m$ (# arcs) $\quad A = \{(1,2), (1,3) \ldots (i,j) \ldots\}$

Directed network: edges or arcs of the network are directed

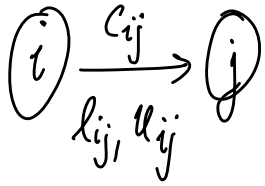
(i,j) $\quad$ 

$(i,j) \neq (j,i)$ in a directed network.

In an undirected network, $(i,j) = (j,i)$, i.e., there is no arrow.

We assume the network is directed by default.

node i: tail
node j: head



(i,j) is an outgoing arc of node i, and an incoming arc of node j.

If $(i,j) \in A$, then j is **adjacent** to i. Notice i is not adjacent to j because of (i,j).

By default, (i,j) being directed implies only one-way traffic from i to j. But, more generally, we could first push flow from i to j, some of which is then sent back to i.

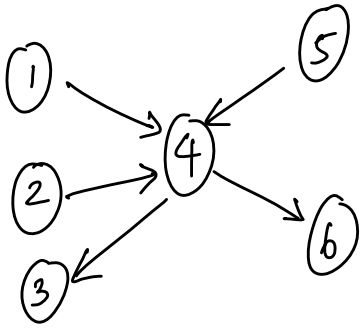
Degree

indegree (i) = # arcs incoming to node i.

outdegree (i) = # arcs outgoing from node i.

degree (i) = indegree (i) + outdegree (i).

Example:



indegree $(4) = 3$

outdegree $(4) = 2$

degree $(4) = 5$

Adjacency List

$A(2) = \{4\}$, $A(5) = \{(5,4)\}$

Arc adjacency list $A(i) = \{(i,j) \in A, j \in N\}$ e.g., $A(4) = \{(4,3),(4,6)\}$

Node adjacency list $A(i) = \{j \in N : (i,j) \in A\}$ e.g., $A(4) = \{3,6\}$

Notation is confusing! A : set of arcs

$A(i)$: node/arc adjacency list of node i.

It'll be evident from the context which one is it!

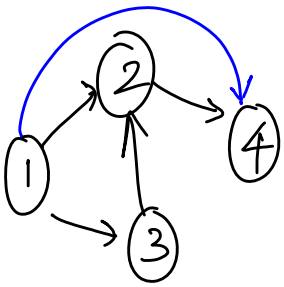
Notice that $\sum_{i \in N} |A(i)| = m$ (the total # arcs).

Subgraph A graph $G' = (N', A')$ with $N' \subseteq N$, $A' \subseteq A$, is a subgraph of G.

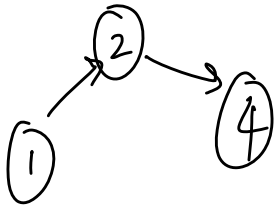
($\subseteq$: subset of or equal to)

An **induced subgraph** is a subgraph that contains each arc of A with both endpoints in N', i.e., each such arc is in A'.

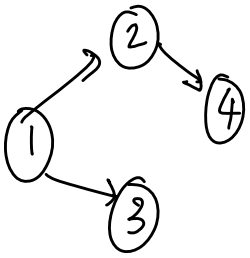
A **spanning subgraph** is a subgraph with $N' = N$.



G



G' is not induced, if (1,4) is included



G'' is a spanning subgraph

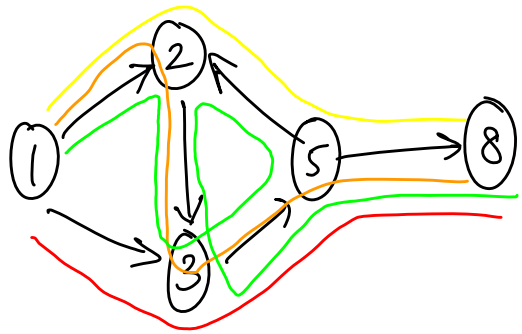
A **walk** in $G = (N, A)$ is a subgraph G_1 made of a sequence of nodes and arcs $i_1 - a_1 - i_2 - a_2 - \dots a_{r-1} - i_r$

("hyphen")

with $a_k = (i_k, i_{k+1})$ or $a_k = (i_{k+1}, i_k)$ for $1 \le k \le r-1$.

So, walk is not necessarily directed.

A **directed walk** is the oriented version of the walk with $a_k = (i_k, i_{k+1})$.



1-2-5-8 is a walk

1-2-3-5-8 is a directed walk

1-2-3-5-2-3-5-8 is also a directed walk

Notice that a node could be visited more than once in a walk.

A **path** is a walk without repetition of nodes.

e.g., 1-2-5-8 is a path.

A **directed path** is a directed walk without repetition of nodes.

e.g., 1-2-3-5-8 is a directed path here.
Similarly, 1-3-5-8 is a directed path

Arc (i,j) in a path is a **forward arc** if i is visited before j, and is a **backward arc** if j is visited before i.

In 1-2-5-8, $(1,2)$ and $(5,8)$ are forward arcs
$(5,2)$ is a backward arc.

A directed path has no backward arcs.

If (i,j) is in a directed path, then i is the **predecessor** of node j, denoted as $\text{pred}(j) = i$.

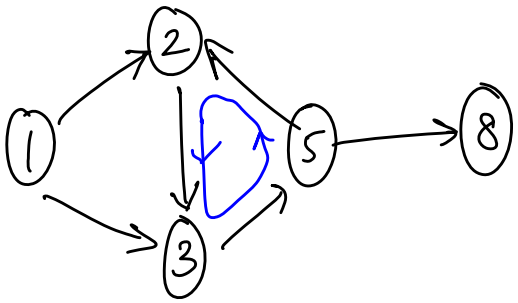
1-3-5-8:

$\text{pred}(8) = 5$
$\text{pred}(5) = 3$
$\text{pred}(3) = 1$
$\text{pred}(1) = 0$ ← marks the beginning of the directed path

A **cycle** is a path $i_1 - i_2 - \cdots i_k$ with arc (i_k, i_1) or (i_1, i_k), denoted as $i_1 - i_2 - \cdots i_k - i_1$ (or $i_k - i_1 - i_2 \cdots - i_k$).

A **directed cycle** is directed path $i_1 - i_2 - \cdots - i_k$ with arc (i_k, i_1), denoted $i_1 - i_2 - \cdots i_k - i_1$.

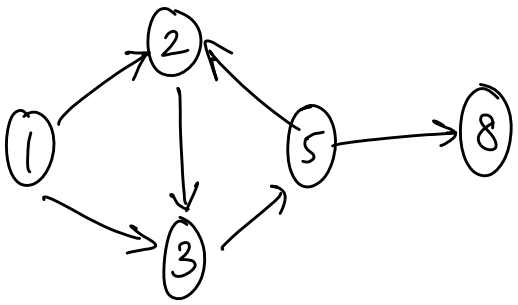
2-3-5-2 is a directed cycle.



An **acyclic network** is a network with no directed cycles.

Nodes i and j are **connected** if there is a path from i to j.
A network is connected if every pair of nodes is connected.

A network is **strongly connected** if there is a directed path from every node to every other node.



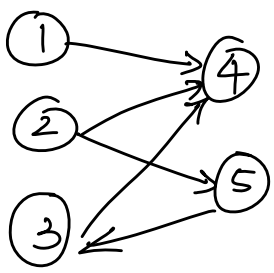
This network is connected, but not strongly connected.

Bipartite graph

$G=(N,A)$ where $N=N_1 \cup N_2$. and $(i,j) \in A$ has

$i \in N_1, j \in N_2$ or $i \in N_2, j \in N_1$

e.g.



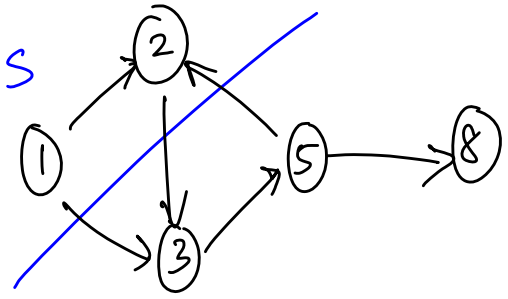
MATH 566 – Lecture 4 (09/04/2014)

More definitions on networks

A **cut** is a partition of the node set N into two parts S and $\bar{S} = N - S$.

↑ complement of S

Example:

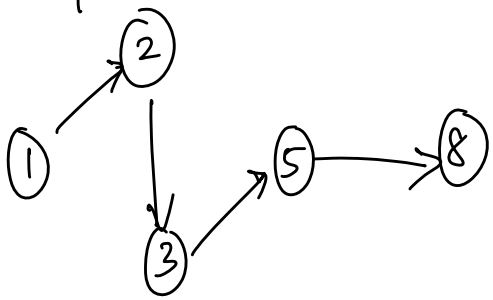


$S = \{1,2\}$
$\bar{S} = \{3,5,8\}$

Here, the set of **arcs in the cut** are $\{(1,3),(2,3),(5,2)\}$. In general, the arcs in the cut are $(i,j) \in A$ with $i \in S, j \in \bar{S}$ or $i \in \bar{S}, j \in S$.

A **tree** is a connected network that contains no cycle.

Example:



is a tree, and is also a subgraph of the network shown above.

Notice that we avoid both directed and undirected cycles in trees.

Spanning tree : A tree T is a **spanning tree** of a graph G if T is a spanning subgraph of G.

The tree shown above is a spanning tree, for example.

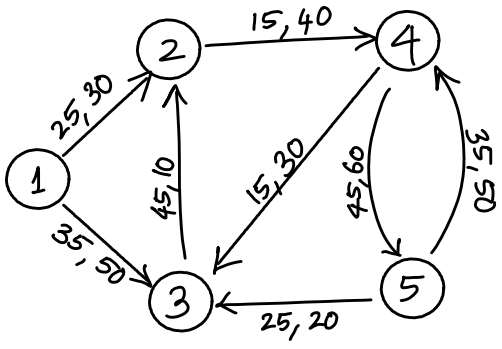
Network Representations

We need efficient representations that store, and can be used for efficient computation using, all data related to the network, i.e., its structure as well as c_{ij}, l_{ij}, u_{ij}, $b(i)$, etc.

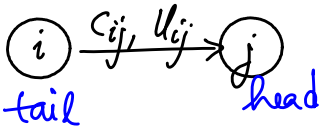
We explore several options for representing networks now.

Node-arc incidence matrix N

This is an $n \times m$ matrix with one row for each node and one column for each arc, with entries in $\{-1, 0, 1\}$. We illustrate the matrix on an example network now.



(AMO Fig 2.14, pg 32)

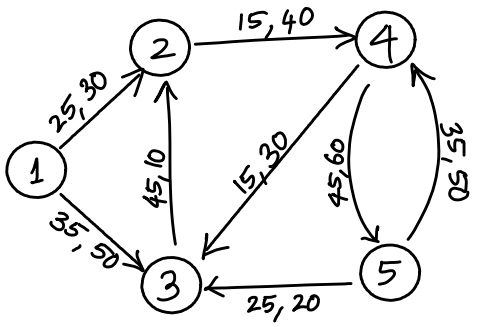


we use the lexicographic ordering for arcs by default

$$N = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} (1,2) & (1,3) & (2,4) & (3,2) & (4,3) & (4,5) & (5,3) & (5,4) \end{array} \\ \begin{bmatrix} 1 & 1 & & & & & & \\ -1 & & 1 & -1 & & & & \\ & -1 & & 1 & -1 & & -1 & \\ & & -1 & & 1 & 1 & & -1 \\ & & & & & -1 & 1 & 1 \end{bmatrix} \end{array}$$

other entries are all zeros

We store l_{ij}, u_{ij} and c_{ij} as m-vectors.



$L = [0 \ 0 \ \cdots \ 0]$ (8-vector of zeros)

$U = [30 \ 50 \ 40 \ 10 \ 30 \ 60 \ 20 \ 50]$

$C = [25 \ 35 \ 15 \ 45 \ 15 \ 45 \ 25 \ 35]$

The book uses L, U, and C to denote these vectors, but we will introduce general notation for vectors soon.

The $b(i)$'s are stored as an n-vector.

Properties of the node-arc incidence matrix N

* each column has one $+1$ and one -1, in the rows corresponding to the tail and the head nodes, respectively.
* # $+1$'s in row i = outdegree(i)
  # -1's in row i = indegree(i)
  } # nonzeros in row i = degree(i)
* the total # nonzeros is $2m$,
  i.e., N is a very sparse matrix!

At the same time, N is a convenient representation of the network structure especially for proving various properties related to the network. For instance, the linear optimization model for min-cost flow can be written in matrix-vector form using this matrix.

Matrix version of the min-cost flow problem

lower case letters with a bar represent vectors in my scribes.

Recall the linear optimization model for the min-cost flow problem:

Notation: $\bar{x}, \bar{c}, \bar{l}, \bar{u}$, are vectors representing $x_{ij}, c_{ij}, l_{ij}, u_{ij}$

$$\left.\begin{aligned} \min \quad & \sum_{(i,j)\in A} c_{ij} x_{ij} \\ \text{s.t.} \quad & \sum_{(i,j)} x_{ij} - \sum_{(j,i)} x_{ji} = b(i) \quad \forall i \in N \\ & l_{ij} \le x_{ij} \le u_{ij} \quad \forall (i,j) \in A \end{aligned}\right\}$$

$$\begin{aligned} \min \quad & \bar{c}^T \bar{x} \\ \text{s.t.} \quad & N\bar{x} = \bar{b} \\ & \bar{l} \le \bar{x} \le \bar{u} \end{aligned}$$

($\bar{c}^T$: transpose; $\bar{b}$: vector of $b(i)$'s)

all vectors, by default, are assumed to be column vectors. Hence we write $\bar{c}^T x$.

Notation in **my notes**:
lower case letters represent 1D variables, e.g., x, y, θ, α, etc.
lower case letters with bars represent n-D vectors: $\bar{x}, \bar{\beta}, \bar{z}$, etc.
Upper case letters represent matrices or sets, e.g, A, B, U, X, etc.

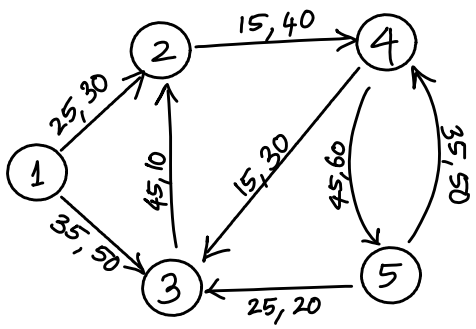
We now introduce another matrix representing the network.

Node-Node adjacency matrix $\mathcal{H}$

This is an $n \times n$ matrix $\mathcal{H} = [h_{ij}]$, with

$$h_{ij} = \begin{cases} 1 & \text{if } (i,j) \in A \\ 0 & \text{otherwise} \end{cases}$$

→ row of the tail node i and column of head node j.



$$\mathcal{H} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Properties of the node-node incidence matrix $\mathcal{H}$

* each arc is represented by a 1 in $\mathcal{H}$.
* the # 1's in row i = outdegree (i)
  the # 1's in column i = indegree (i)
* $\mathcal{H}$ has n^2 elements, m of which are nonzero.
  So, $\mathcal{H}$ is an efficient data structure when $m >> n$.
  ↑ "much greater than"

We store l_{ij}, u_{ij}, and c_{ij} as separate $n \times n$ matrices L, U, and C.

Unless the network is really dense, i.e., has lots of arcs connecting the given nodes, $\mathcal{H}$ could have a lots of zeros (and so will L, U, and C). Hence we look for even more efficient data structures.

It must be mentioned that there are standard ways of handling sparse matrices, e.g., the sparse package in MATLAB. But we would ideally like to avoid the overhead associated with such operations by using more efficient representations.

Adjacency lists

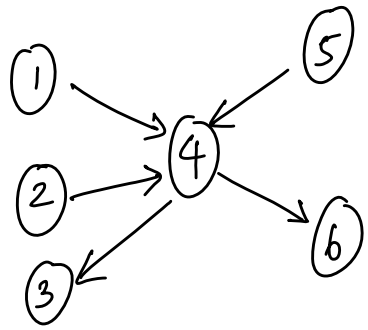
We already saw $A(i)$: the out arc list of node i, represented either as a set of arcs, or their head nodes.

$A(i)$: outarc adjacency list

Similarly, we can define **inarc adjacency lists**:

$$AI(i) = \{(j,i) \in A\}.$$

e.g., $AI(4) = \{(1,4), (2,4), (5,4)\}$

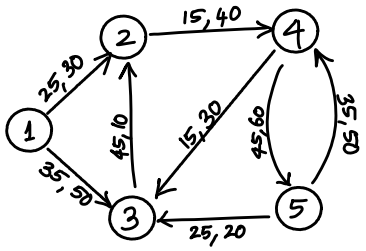


$A(4) = \{(4,3), (4,6)\}$

We describe two data structures to store lists of data:

Array and **linked list of cells**

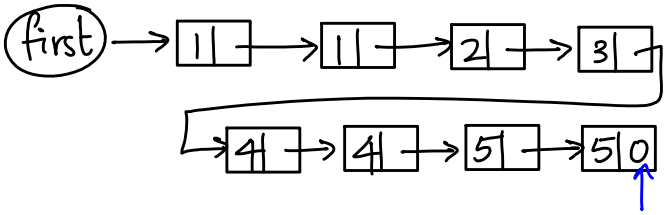
* vector
* stores numbers consecutively as a list
* size is known beforehand, and cannot be changed dynamically
* simple, and easy to use



e.g., list of tail nodes of arcs.

	(1,2)	(1,3)	(2,4)	(3,2)	(4,3)	(4,5)	(5,3)	(5,4)
$\bar{t}$ =	1	1	2	3	4	4	5	5

* each cell stores a value and a "pointer" to the next cell
* uses memory very efficiently
* size can be changed easily - can insert or delete cells at any position

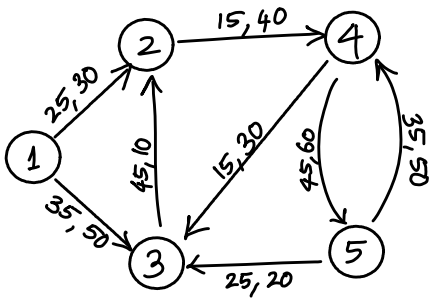


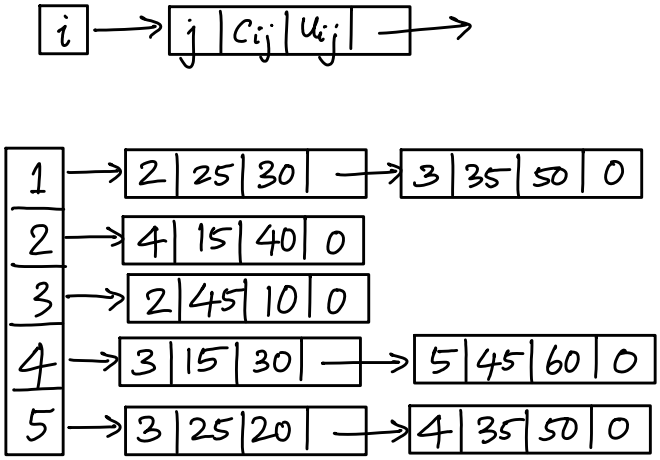
ZERO or NULL signifying end of the list

In MATLAB, the size of arrays could be changed dynamically. In other words, MATLAB is doing the background work for you. But efficient implementations in C/C++, say, using linked lists often run much faster, especially on large problem instances.

For instance, we could use linked lists to store the outarc lists for each node, while storing related parameters such as c_{ij}, u_{ij}, etc., for an efficient storage of the network.

e.g.,





For ease of use, we typically consider a matrix notation, storing the data for all the arcs in an $m \times 4$ or $m \times 5$ matrix.

T	H	COST	UB
1	2	25	30
1	3	35	50
2	4	15	40
3	2	45	10
4	3	15	30
4	5	45	60
5	3	25	20
5	4	35	50

If l_{ij}'s are also given, we get another column (hence, it will be an $m \times 5$ matrix in that case).

The b(i) values are stored as an n-vector separately.

Notice that the information is stored in the order of the arcs. To extract the outarc list efficiently from this matrix notation, we use one more vector named the **point**.

Most algorithms run steps of operations on the outarcs or inarcs of each node. Hence it is important to have these lists readily available.

	T	H	COST	UB
1 →	1	2	25	30
2	1	3	35	50
3 →	2	4	15	40
4 →	3	2	45	10
5 →	4	3	15	30
6	4	5	45	60
7 →	5	3	25	20
8	5	4	35	50

point(i)

1	1
2	3
3	4
4	5
5	7
6	9

For $1 \le i \le n$, point(i) stores the row # (index) of the first arc (in lexicographic order) that has i as its tail. point(n+1) is always initialized to m+1.

With this point vector, the outarcs of node i are in rows indexed point(i) to point(i+1) − 1, for $1 \le i \le n$.

For instance, outarcs of node 4 are in rows point(4) = 5 to point(5) − 1 = 7 − 1 = 6, i.e., in rows 5 and 6 of the matrix.

The m×4 (or m×5) matrix of arcs data along with the point vector is called the **forward star representation** of the network.

MATH 566 – Lecture 5 (09/09/2014)

Next week (Sept 16 & 18): Lectures originate in Pullman.

Forward star representation – review from last lecture.

In practice, it is convenient to create and store A(i) and AI(i) lists. MATLAB allows one to change sizes of matrices on the fly. Of course, MATLAB is doing the work for us under the surface!

netdata.m

```
%% Math 566 (Fall 2014)
%% Octave code to extract out- and in-arc lists
%% from the forward star representation

%T---TAIL,  H---HEAD
%DEGO---degree_out,    DEGI---degree_in
%n---number of nodes,  m---number of arcs
%A{i}---out-arc list at node i, cell array
%AI{i}---in-arc list at node i, cell array

DEGO=zeros(1,n);
DEGI=DEGO;
A{1,n} = [];
AI{1,n} = [];
for k=1:m
  i=T(k);
  j=H(k);
  pos=DEGO(i)+1;
  DEGO(i)=pos;
  A{i}(pos)=k;

  posI=DEGI(j)+1;
  DEGI(j)=posI;
  AI{j}(posI)=k;
end%for
```

Example1.m

```
data=[1  1 2 25 30
      2  1 3 35 50
      3  2 4 15 40
      4  3 2 45 10
      5  4 3 15 30
      6  4 5 45 60
      7  5 3 25 20
      8  5 4 35 50 ];

 % data=data(3:end,:)
 T=data(:,2);
 H=data(:,3);
 m=length(T);
 n=max(max(T), max(H));

 netdata
 DEGO
 celldisp(A)
 DEGI
 celldisp(AI)
```

Output from Octave

```
octave:2> example1
DEGO =
   2   1   1   2   2

A{1} =
   1   2
A{2} =
  3
A{3} =
  4
A{4} =
   5   6
A{5} =
   7   8

DEGI =
   0   2   3   2   1

AI{1} =
[](0x0)
AI{2} =
   1   4
AI{3} =
   2   5   7
AI{4} =
   3   8
AI{5} =
  6
```

More commands

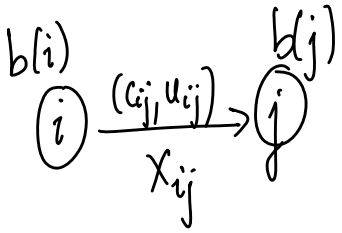
```
octave:5> data
data =
    1    1    2   25   30
    2    1    3   35   50
    3    2    4   15   40
    4    3    2   45   10
    5    4    3   15   30
    6    4    5   45   60
    7    5    3   25   20
    8    5    4   35   50
octave:6> T=data(:,2);
octave:7> H=data(:,3);
octave:8> T
T =
   1
   1
   2
   3
   4
   4
   5
   5
octave:9> H
H =
   2
   3
   4
   2
   3
   5
   3
   4
octave:10> m=length(T)
m =  8
octave:11> n=max(max(T), max(H))
n =  5
octave:12> 1:m
ans =
   1   2   3   4   5   6   7   8
```

We will illustrate many algorithms using simple commands/functions in MATLAB as shown above.

Network transformations

we discuss several ways to transform networks. The goal is to make the network amenable to algorithms requiring certain structure, while not changing the problem entirely.

Recall the model for min-cost flow:



$$\min \sum_{(i,j)\in A} c_{ij} x_{ij}$$

$$\text{s.t.} \sum x_{ij} - \sum x_{ji} = b(i) \quad \forall i$$

$$l_{ij} \le x_{ij} \le u_{ij} \quad \forall (i,j) \in A.$$

(1) Removing nonzero lower bounds

$$l_{ij} - l_{ij} \le x_{ij} - l_{ij} \le u_{ij} - l_{ij}$$

$$0 \le x'_{ij} \le u'_{ij}$$

$$x_{ij} - l_{ij} = x'_{ij}$$

$$\text{so, } x_{ij} = x'_{ij} + l_{ij}$$

The flow balance constraint for node i becomes

$$\sum_{(i,j)} (x'_{ij} + l_{ij}) - \sum_{(j,i)} (x'_{ji} + l_{ji}) = b(i)$$

"implies" $\Rightarrow$

$$\sum_{(i,j)} x'_{ij} - \sum_{(j,i)} x'_{ji} = \underbrace{b(i) - \sum_{(i,j)} l_{ij} + \sum_{(j,i)} l_{ji}}_{\text{this is the modifide } b(i)}$$

② Arc reversal

Used for reversing negative costs.

$$x_{ij} = u_{ij} - x_{ji}$$

Logic: first send u_{ij} units from ⓘ to ⓙ, and max out arc (i,j). We modify $b(i)$ and $b(j)$ to reflect this change. Then pull back x_{ji} units from j to i.

Contribution to total cost: $c_{ij}x_{ij}$ (in original network)

$$= c_{ij}(u_{ij} - x_{ji}) = \underbrace{c_{ij}u_{ij}}_{\text{constant}} - c_{ij}x_{ji}$$

$$= \text{constant} + (-c_{ij})x_{ji}$$

Having one or more constant terms added to the objective function does not change the problem or its optimal solution.

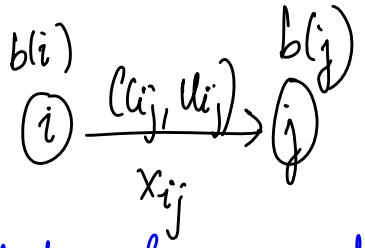
The motivation for arc reversal is the possibility of applying an algorithm which assumes all c_{ij} are non-negative.

③ Removing upper bounds

We assume $l_{ij} = 0 \ \forall (i,j)$.

We could convert finite u_{ij}'s to ∞ by transforming the network as follows.

Let's concentrate on (i,j) alone.



set of constraints for arc (i,j) alone

$$x_{ij} = b(i) \quad \text{—— (1)}$$
$$-x_{ij} = b(j) \quad \text{—— (2)}$$
$$x_{ij} \le u_{ij}$$

$$\Rightarrow x_{ij} + s_{ij} = u_{ij} \quad \text{—— (3)}$$

s_{ij}: "slack" variable

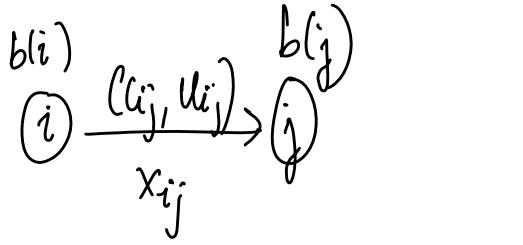
Notice that we have an equation for each node (flow balance), and a "$\le$" inequality for each arc. To remove u_{ij}, we convert its "$\le$" to an "$=$" → i.e., we add a new node and associated arcs

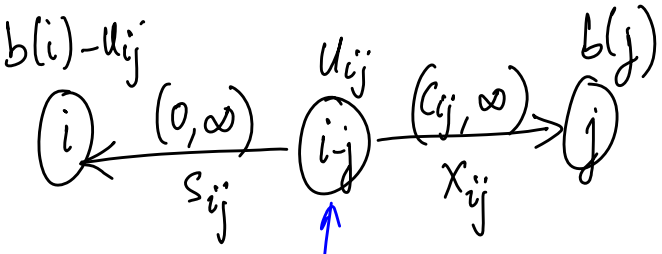
Subtract (3) from (1) to get (1'): $-s_{ij} = b(i) - u_{ij}$ —— (1')

Take the system (1'), (2), (3):

$$-s_{ij} = b(i) - u_{ij} \quad \text{— (1')}$$
$$-x_{ij} = b(j) \quad \text{—— (2)}$$
$$x_{ij} + s_{ij} = u_{ij} \quad \text{—— (3)}$$

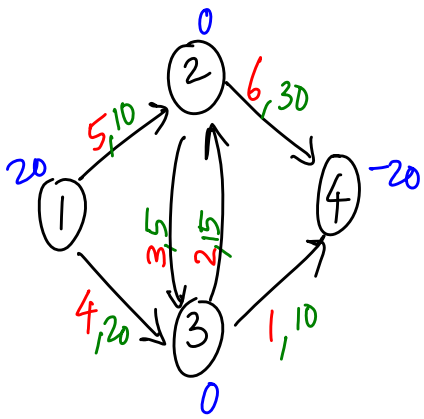
book does this to add i-j as a demand node

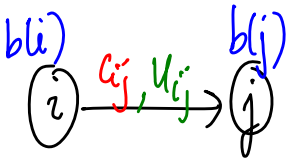


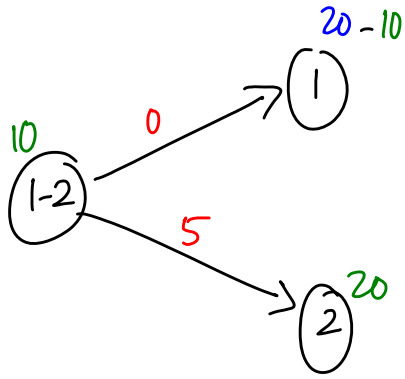


Notice that each flow term x_{ij} appears twice in the system of flow balance constraints – once as $+x_{ij}$ (i.e., as outflow) in the equation for node i, and once as $-x_{ij}$, i.e., as inflow, in the equation for node j. We modified the system of equations (1), (2), (3) so that the terms x_{ij} and s_{ij} appear twice each in the modified system.

We could apply this transformation on a entire network.







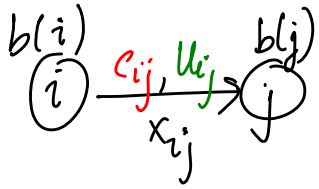
3

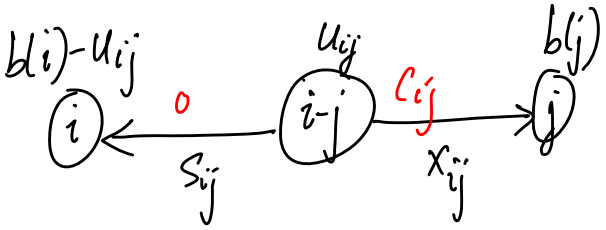
4

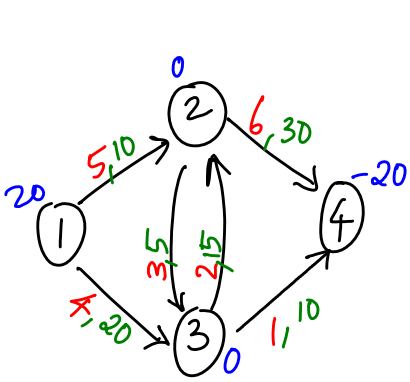
We'll finish this exercise in the next lecture...

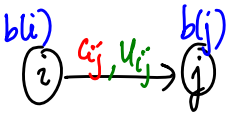
MATH 566 – Lecture 6 (09/11/2014)

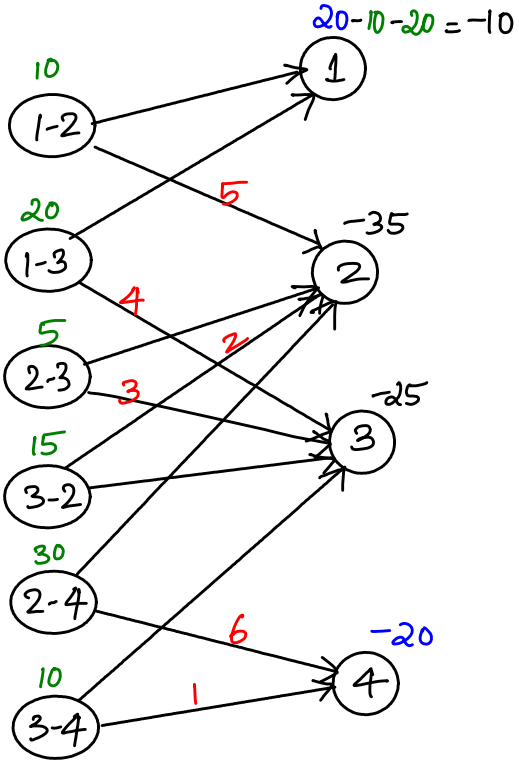
Removing upper bounds











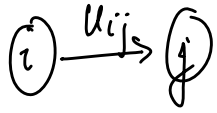
The overall change to b(i) will be

$$b(i) \leftarrow b(i) - \sum_{(i,j)\in A} u_{ij}$$

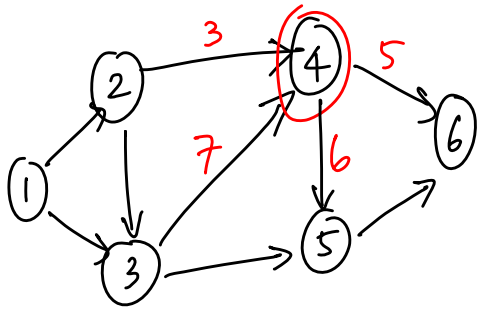
The new network is bipartite, with the original nodes all in N_2 and the new nodes (modeling the arc upper bounds) being put in N_1.

One could use this class of transformations to obtain a bipartite structure. The motivation is the possible design/application of algorithms that work better on bipartite graphs.

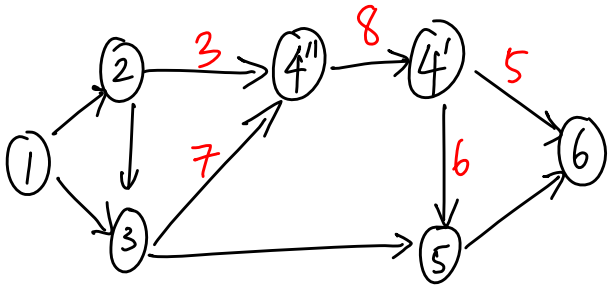
(4) Node splitting



We illustrate the operations on a small example.



Say the "capacity" of node 4 is 8.



If there is a unit cost c_4 for flow through node 4, you can put that cost on arc $(4'', 4')$.

In general, if node i is split into i'' and i' with the arc (i'', i') added, we can reconfigure the arcs incident to node i in the original network as follows. In-arcs of i of the form (j, i) are replaced by (j, i''), while out-arcs of i of the form (i, k) are replaced by (i', k).

Intuition: Consider traffic flow through a road network modeled as a network. There could be a cost for going through the node modeling a city with downtown, for instance, but there is no cost for bypassing the city.

Computational Complexity

Q: How to measure efficiency of an algorithm?

We do "worst case analysis", i.e., analyse the worst possible performance, so that we can guarantee it will perform at least as well on all instances.

Example: Add two $m \times n$ matrices A, B to get matrix C.

```
for i = 1 to m
    for j = 1 to n
        C_ij := A_ij + B_ij
    end
end
```

(assignment)

types of operations:
addition (+)
assignment (:=)
comparisons (for $i = \ldots$, $j = \ldots$)

What is the "running time" of this algorithm?

→ The number of steps (or operations) as a function of m & n.

→ Assume each operation takes a constant amount of time.

The exact time taken for an operation might vary from one computer to another. Hence we want to estimate the number of operations, and ignore the actual "time" as a constant multipliers.

We want to measure the running time for the worst case — so that we can **guarantee** the algo will perform as well on all instances.

Here, the running time includes

* $c_1 mn$ for additions
* $c_2 mn$ for assignments
* $c_3 mn$ for comparisons

} we want to ignore the constants c_1, c_2, c_3, c_4

i.e., $c_4 mn$ overall time.

We say the algorithm runs in $O(mn)$ time. ← "big-O" asymptotic upper bound

Def An algorithm is said to run in $O(f(n))$ time if for some numbers c (>0, real) and n_0, the time taken by the algorithm is at most $cf(n)$ for all $n \geq n_0$.

$O(\cdot)$ as a function gives the most dominant term in the input, e.g.,

$$O(100n + n^2 + 0.0001 n^3) = O(n^3).$$

$O(\cdot)$ gives a convenient way to ignore the constant terms in the polynomial expressions.

Size of a problem: # bits needed to store the problem i.e., represent all data of the problem.

For instance, let ~~2^{50}~~ $0 < A_{ij} < 2^{51}$ then each A_{ij} takes up to 51 bits.

If each element needs k bits, then the algo uses $O(mnk)$ storage. Typically, $K = O(\log(A_{max}))$, where

$$A_{max} = \max_{i,j} \{|A_{ij}|, |B_{ij}|\}$$

while $O(\cdot)$ gives upper bound, we also talk about lower bound.

Big Ω and big Θ notations

Def An algorithm is said to run in $\Omega(f(n))$ time if for some numbers c' (>0, real) and n_0', for all instances with $n \geq n_0$, the algorithm takes at least $c'f(n)$ time on **some** problem instance.

Notice the difference in the definitions of $O(\cdot)$ and $\Omega(\cdot)$.

$O(f(n))$: **every** instance takes at most $cf(n)$ time.

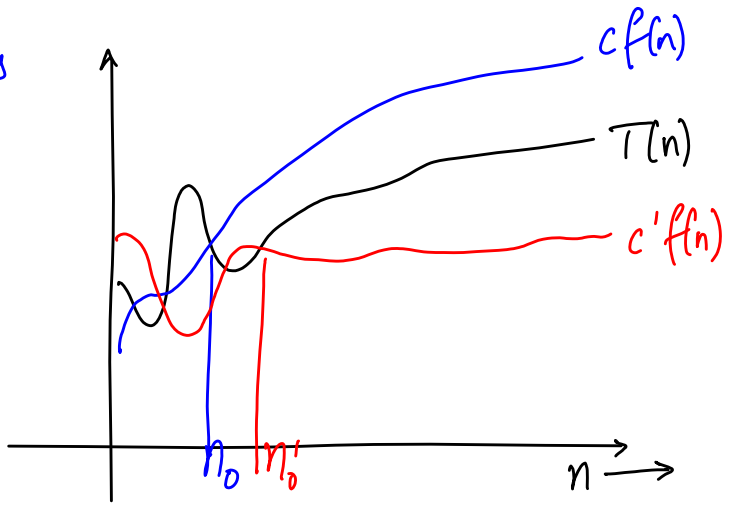
$\Omega(f(n))$: **some** instance takes at least $c'f(n)$ time.

Def An algorithm is said to be $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$.

↳ ≡ said to run in $\Theta(f(n))$ time

Here is a typical function for running time and how it compares with lower and upper bounds as n becomes large.

$T(n)$: running time



Example $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

To show this result, we want to find $c>0, c'>0$ and $n_0>0$ such that

$$c'n^2 \le \frac{1}{2}n^2 - 3n \le cn^2 \quad \forall\ n \ge n_0.$$

$$\Rightarrow \quad c' \le \frac{1}{2} - \frac{3}{n} \le c$$

↳ at $n=6$, we get $\frac{1}{2} - \frac{3}{n} = 0$. We can look at n that are 7 or higher for possible choices.

We can choose $c = \frac{1}{2}$, $n_0 = 7$, $c' = \frac{1}{14}$.

We are interested in **'polynomial time'** algorithms, i.e., algos whose running times are bounded by polynomial functions of problem parameters – $n, m, \log C, \log U$, etc.

e.g., $O(mn)$, $O(n^2)$, $O(m + n\log C)$, $O(mn + n^2 \log^2 U)$.

Here, C : largest arc cost (in absolute value)

U : largest upper bound

An algorithm is said to be a **strongly polynomial time** algorithm if the running time is bounded by a polynomial function of m and n (so, $\log C$, $\log U$ terms do not appear).

Exponential time algorithms : $O(f(n))$ where $f(n)$ is exponential in $m, n, \log C, \log U$.

$f(m, n, C, U)$ to be exact

e.g., $O(2^n)$, $O(n!)$, etc.

e.g., algorithms for most general cases of the traveling salesman problem are exponential time.

pseudo-polynomial time algorithm : $O(f(n))$ where f is polynomial in m and n, but not so in C, U.

e.g., $O(mn^2C)$, $O(m + nU)$, etc.

We will start discussing network algorithms in the next lecture...

MATH 566 – Lecture 7 (09/16/2014)

Search algorithms → critical to many network flow algorithms

We will discuss

* generic search
* breadth-first search (BFS)
* depth-first search (DFS)

INPUT: $G=(N,A)$ and a node s

OUTPUT: $\{j \in N \mid$ ("such that") a directed that exists in G from s to $j\}$.

– set of nodes reachable from s.

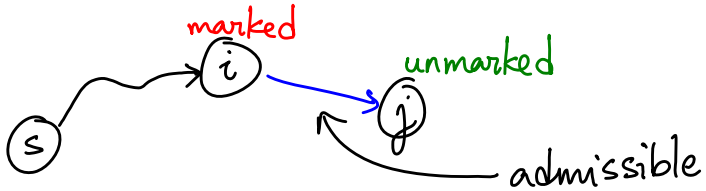
At any stage of the algorithm, a node is either **marked** or **unmarked**.

marked: is reachable from s

unmarked: status yet to be determined

Critical step: If node i is marked, node j is unmarked, and $(i,j) \in A$, then mark j.

Such an arc is said to be **admissible**.



Use pred(i) indices to store directed paths from s to i, and store traversal order of nodes in order(i).

```
algorithm search;
begin
    unmark all nodes in N;
    mark node s;
    pred(s) : = 0;
    next : = 1;
    order(s): = 1;
    LIST : = {s}
    while LIST ≠ ∅ do
    begin
        select a node i in LIST;
        if node i is incident to an admissible arc (i, j) then
        begin
            mark node j;
            pred( j) : = i;
            next : = next + 1;
            order( j) : = next;
            add node j to LIST;
        end
        else delete node i from LIST;
    end;
end;
```

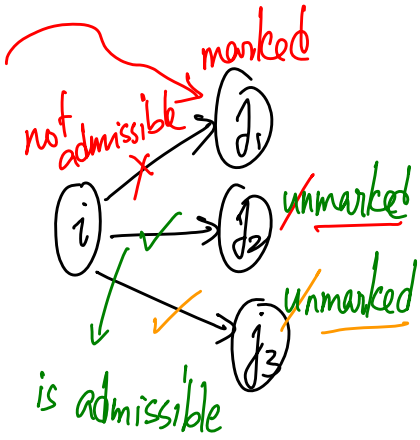
Figure 3.4 Search algorith

initialization

counter

or order(s) = next;

can be stored in a binary array

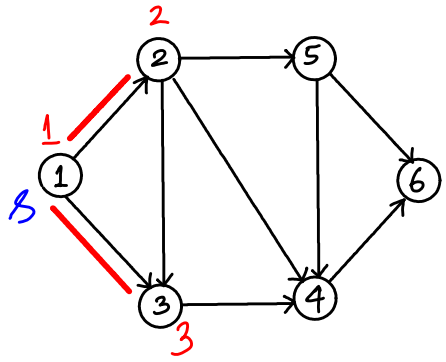


We could keep track of the arcs in A(i) (out-arc list) that are being examined using a pointer/indicator current arc.

Depending on how we maintain and update the LIST, we get different versions of the generic search.

If we maintain LIST as a queue, i.e., select nodes from front and add nodes to the back, we get the **breadth-first search** (BFS). The nodes are handled in a first-in first-out (FIFO) order.

Example (AMO-Figure 3.5)



order (i)

Initialization

indicator for whether nodes are marked (1) or not (0)

Mark = [1 2 3 4 5 6 / 1 0 0 0 0 0]

Mark(s) = 1;

pred(s) = 0;

next = 1;

order(s) = next;

LIST = [s] = [1];

$A(1) = \{(1,2), (1,3)\}$

$A(2) = \{(2,3), (2,4), (2,5)\}$

$A(3) = \{(3,4)\}$

$A(4) = \{(4,6)\}$

$A(5) = \{(5,4), (5,6)\}$

$A(6) = \emptyset$

} out arc lists

Iteration 1

i = 1.

(1,2) is admissible: (i, j) = (1,2)

current arc = (1,2),

mark[head(current arc)];

mark = [1 2 3 4 5 6 / 1 1 0 0 0 0]

pred(2) = 1

(in general, pred(j) = i)

next = next + 1 = 2

order(2) = next

LIST = [1 2]

(1,2) is now inadmissible

Iteration 2 (i = 1)

current_arc ← (1,3)

Mark = [1 2 3 4 5 6 / 1 1 1 0 0 0]

pred(3) = 1

next = 3

order(3) = next

LIST = [1 2 3]

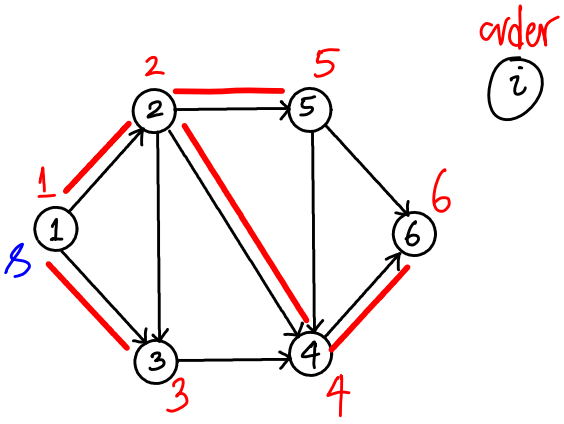
(1,3) is now inadmissible

Iteration 3 ($i=1$)

no admissible arcs out of node i

So, delete i from LIST

LIST = [2 3] → [2 3 4] → [2 3 4 5] → [3 4 5] → [4 5] → [4 5 6] → [5 6] → [6] → ϕ

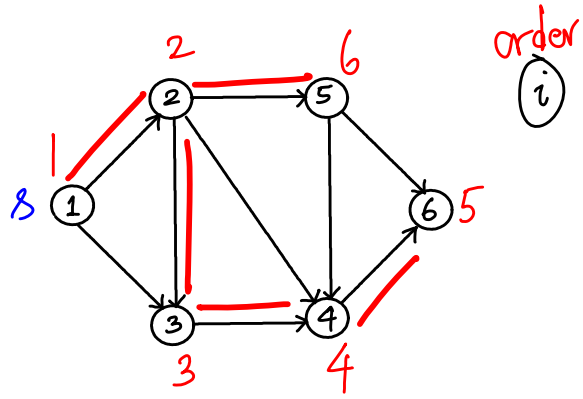


We could halt the algorithm as soon as we have marked all nodes. But it could happen that not all nodes are reachable from s, in which you run till you have no more admissible arcs, and LIST is empty.

We get a search tree, which is called the BFS tree.

Property In the BFS tree, the directed path from s to any node i contains the smallest number of arcs among all directed paths from s to i, i.e., it is the shortest path in # arcs.

Depth-First Search We maintain LIST as a stack, i.e., in the last-in first-out (LIFO) order.



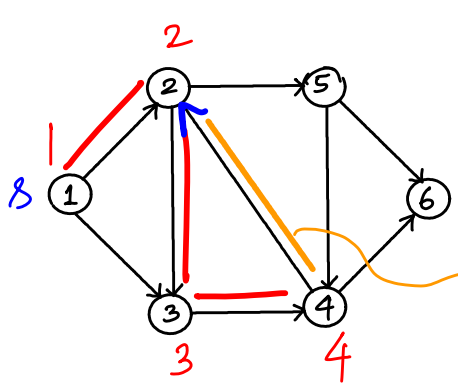
LIST = [1] → [2 1] → [3 2 1]

→ [4 3 2 1] → [6 4 3 2 1]

→ [4 3 2 1] → [5 2 1]

Notice that DFS identifies a path from $s=1$ to 6 of 4 arcs, while BFS identifies one with 3.

DFS identifies directed cycles quickly.



replace (2,4) with (4,2) here, for illustration.

first inadmissible arc identifies a directed cycle.

The first inadmissible arc in DFS identifies a direct cycle. BFS would also identify such a cycle, but it might not be as quick as DFS.

Reverse search Find all nodes from which we can reach node t, i.e., there is a directed path to node t.

Start with LIST = [t], examine AI(t), and span out "backwards" examining AI(i) lists.

Strong Connectivity $G = (N, A)$ is strongly connected if there is a directed path from every node i to every node j.

Choose any node s, apply forward search from s and reverse search to s. If we get spanning trees in both cases, the network is strongly connected.

It is sufficient to do the forward and reverse searches from a single node s (any node) – why?

MATH 566 – Lecture 8 (09/18/2014)

Running time of the (generic) search algorithm.

```
algorithm search;
begin
    unmark all nodes in N;
    mark node s;
    pred(s) := 0;
    next := 1;
    order(s) := 1;
    LIST := {s}
    while LIST ≠ ∅ do
    begin
        select a node i in LIST;
        if node i is incident to an admissible arc (i, j) then
        begin
            mark node j;
            pred(j) := i;
            next := next + 1;
            order(j) := next;
            add node j to LIST;
        end
        else delete node i from LIST;
    end;
end;
```

Figure 3.4 Search algorith

→ $O(n)$

Constant # operations

$|A(i)|$ times

constant time for each operation

Within the **while** loop, a node gets added or deleted from LIST. — $2n$ operations i.e., $O(n)$ operations.

We examine in the worst case $\sum |A(i)| = m$ arcs for admissibility. — $O(m)$ operations.

Overall, the # operations is $O(m+n) = O(m)$. Hence we say that (generic) search runs in $O(m)$ time. The complexity is the same for BFS and DFS.

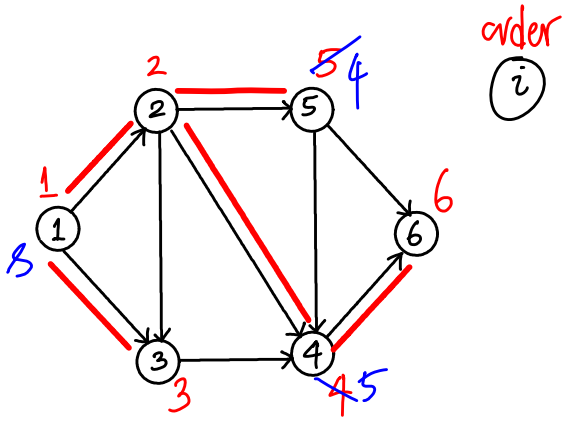
We assume $m > n$ (typically). Given n nodes, we could have up to n^2 directed arcs. If there are more nodes than arcs, many nodes might be isolated, and hence would not affect algorithms as much as the connected components.

Topological Ordering

We use order(i) as a label for node i. For many instances, a set of labels that mark all arcs in an orderly fashion, e.g., low to high label, is desirable.

Def A labeling is a **topological ordering** if $\forall (i,j) \in A$, $\text{order}(i) < \text{order}(j)$.

Recall the order generated by BFS on the example network.

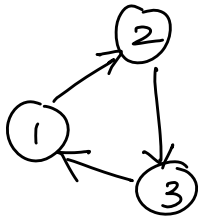


This ordering is not topological – see (5,4).

But we can swap the labels for nodes 4 and 5 to get a topological ordering.

Not all graphs are guaranteed to have a topological ordering. Directed cycles created obstructions.

In the absence of directed cycles, it turn out we can come up with a topological ordering.



has no topological ordering

In fact, we can prove the following result:

network is acyclic $\Leftrightarrow$ it has a topological ordering.

We outline the results that give one direction of the above equivalence.

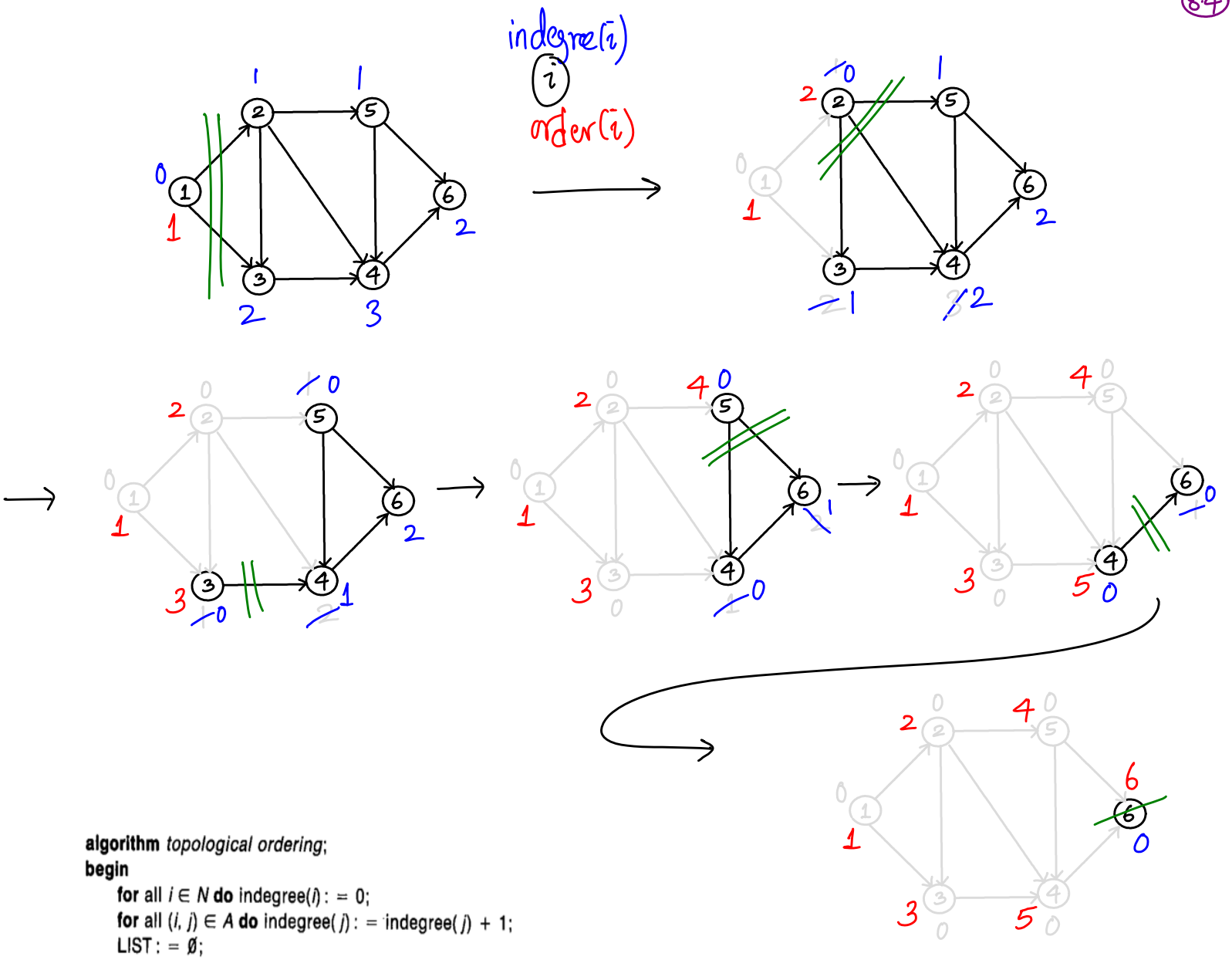
Lemma If outdegree$(i) \geq 1 \ \forall i \in N$, then the first inadmissible arc of DFS determines a directed cycle.

Corollary 1 If G has no directed cycle, there is at least one node with zero outdegree, and at least one node with zero indegree.

Corollary 2 If G has no directed cycle, one can label the nodes so that order$(i) <$ order(j) for all $(i,j) \in N$.

Idea for algorithm: Start with nodes that have indegree=0. Assign order=1. Delete these nodes, and arcs going out from them. Adjust node degrees. Assign order=2 for nodes with indegree=0 now. Repeat this process.

When implementing, you do not have to delete the nodes and arcs. Maintaining and modifying the indegree list is sufficient!




```
algorithm topological ordering;
begin
    for all i ∈ N do indegree(i) := 0;
    for all (i, j) ∈ A do indegree(j) := indegree(j) + 1;
    LIST := ∅;
    next := 0;
    for all i ∈ N do
        if indegree(i) = 0 then LIST := LIST ∪ {i};
    while LIST ≠ ∅ do
    begin
        select a node i from LIST and delete it;
        next := next + 1;
        order(i) := next;
        for all (i, j) ∈ A(i) do
        begin
            indegree(j) := indegree(j) − 1;
            if indegree(j) = 0 then LIST := LIST ∪ {j};
        end;
    end;
    if next < n then the network contains a directed cycle
    else the network is acyclic and the array order gives a topological order of nodes;
end;
```

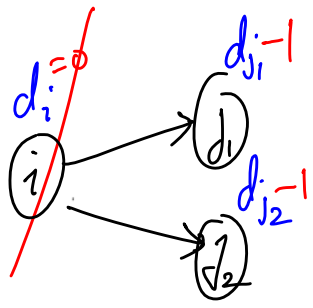
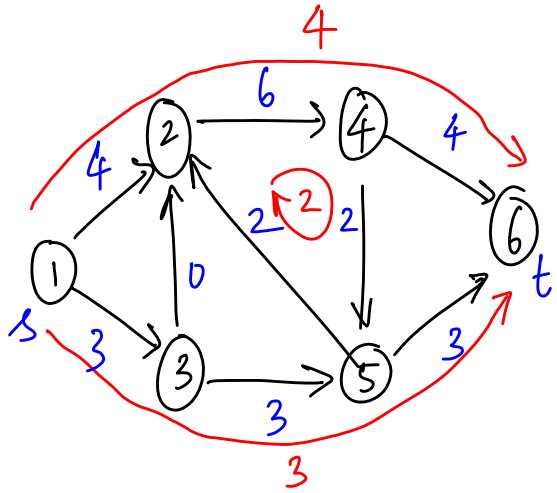


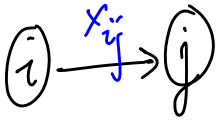
Figure 3.8 Topological ordering algorithm.

The topological ordering algorithm also runs in $O(m)$ time.

Flow Decomposition

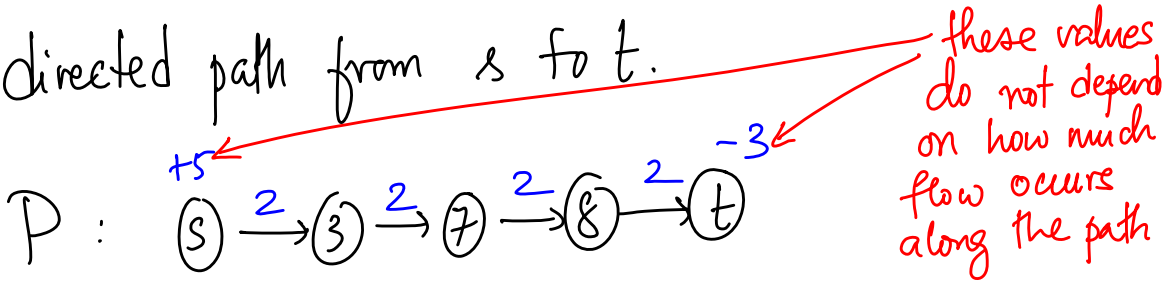
So far, we have considered flow in arcs, using the x_{ij} variables. Alternatively, we could consider flow in paths from supply to demand nodes, and flow in cycles.





Flow along two paths
1-2-4-6 and 1-3-5-6,
and one cycle 2-4-5-2.

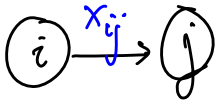
Let P be a directed path from s to t.

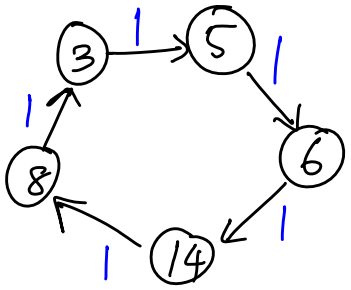


To describe a flow in path P, we specify that all intermediate nodes conserve flow, i.e., inflow = outflow is satisfied within that path.

Notice that these intermediate nodes could participate in more than one path at a time.

Similarly, to describe the flow in a cycle W, we specify that all nodes satisfy outflow = inflow.





We denote by $f(P)$ the flow in path P, and by $f(W)$ the flow in cycle W. We also specify paths and cycles in terms of indicator variables for the arcs, δ_{ij}.

$$\delta_{ij}(P) = \begin{cases} 1 & \text{if } (i,j) \in P \\ 0 & \text{otherwise} \end{cases} \qquad \delta_{ij}(W) = \begin{cases} 1 & \text{if } (i,j) \in W \\ 0 & \text{otherwise.} \end{cases}$$

Given the flow in a set of paths $\mathcal{P}$ and set of cycles $\mathcal{W}$, we can get the corresponding flow in arcs directly using these indicator variables.

$$x_{ij} = \sum_{P \in \mathcal{P}} f(P) \cdot \delta_{ij}(P) + \sum_{W \in \mathcal{W}} f(W) \cdot \delta_{ij}(W) \quad \forall (i,j) \in A.$$

Consider the arc (2,4) in the example above. (2,4) is part of the path 1-2-4-6 and the cycle 2-4-5-2. Hence

$$x_{24} = f(1\text{-}2\text{-}4\text{-}6) + f(2\text{-}4\text{-}5\text{-}2) = 4 + 2 = 6.$$

Here is the more interesting question, though.

Can we go the other way? Can we convert an arc flow to a collection of path and cycle flows?

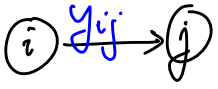
Indeed we can! We can use search repeatedly to find directed paths from supply to demand nodes, as well as directed cycles, to push flow along.

We describe an algorithm that performs such a flow decomposition. At any stage, we work with an intermediate flow y_{ij}, and the associated network. We start with $y_{ij} = x_{ij}$, and update the network as we push parts of y_{ij} along paths and cycles (both directed).

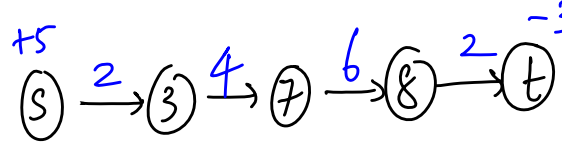
We define the capacity of an s-t path with intermediate flow y_{ij} and supply/demand $b(i)$ as follows.

$$\Delta(P) = \min\{b(s), -b(t), \{y_{ij} \mid (i,j) \in P\}\}.$$

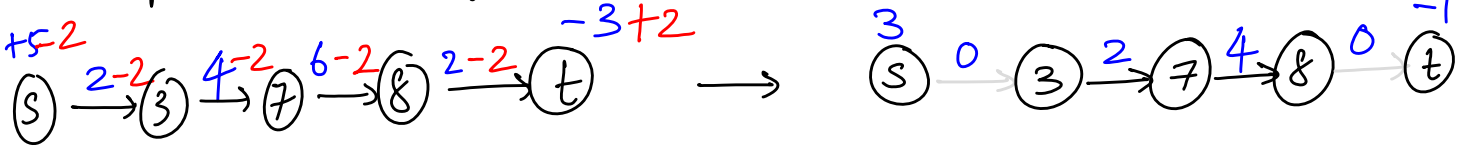
Example:



Capacity of path: P:



Here, $\Delta(P) = 2$. We could push $\Delta = 2$ units along P, and update the y_{ij} and $b(i)$ values as follows.

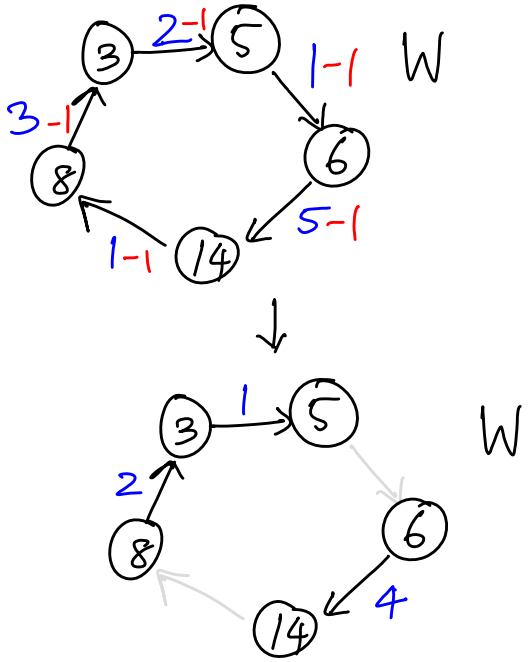


Notice that arcs $(s,3)$ and $(8,t)$ can now be removed from the current network.

We similarly define the capacity of a cycle:

$$\Delta(W) = \min\{y_{ij} \mid (i,j) \in W\}.$$

Here, $\Delta(W)=1$. We can push $\Delta=1$ units along this directed cycle, and update the network.



We ran DFS repeatedly – if we find a directed cycle, we push its capacity of flow. Else if we find a path from a supply to a demand node, we push its capacity. We update the intermediate flows and the network, and continue. See the algorithm in the handout posted on the course web page.

MATH 566 – Lecture 9 (09/23/2014)

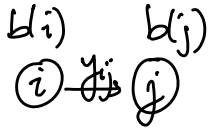
Flow decomposition – Given arc flow x_{ij}, find equivalent flows in paths $\mathcal{P}$ and directed cycles $\mathcal{W}$.

Update in $b(s), b(t)$ when sending flow along a path P.

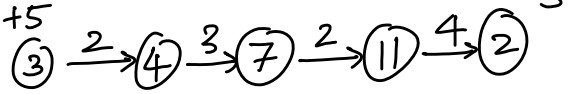
$$b(s) = b(s) - \Delta(P)$$
$$b(t) = b(t) + \Delta(P)$$

$$\Delta(P) = \min\{b(s), -b(t), y_{ij} \mid (i,j) \in P\}$$

We are trying to account for all y_{ij} and $b(i)$ for supply/demand nodes, by assigning amounts out of y_{ij} and $b(i)$ into $f(P)$ and $f(W)$. Ultimately, we want to get $b(i)=0 \ \forall i$, and $y_{ij}=0 \ \forall (i,j)$, at which point, $f(P)$ and $f(W)$ for $P \in \mathcal{P}$ and $W \in \mathcal{W}$ account for all the flows.

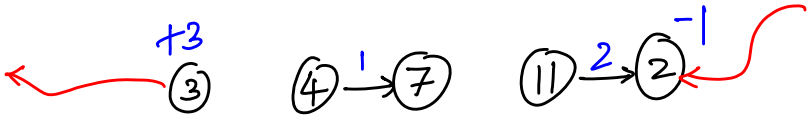


Example



Here, $\Delta = 2$.

After sending flow of $\Delta = 2$ along $P = 3\text{-}4\text{-}7\text{-}11\text{-}2$, we get



There should be other path flows to take off $b(3) = 3$ now.

Flow decomposition algorithm

We work with intermediate flow y, and push it all on the flows on paths and cycles.

Notation:

y – intermediate flow
$G(y) = (N(y), A(y))$ – graph corresponding to flow y
$A(y) = \{(i,j) \in A \mid y_{ij} > 0\}$ (arcs with positive flow in y)
$N(y) = \{i \mid (i,j) \in A(y) \text{ or } (j,i) \in A(y)\}$ (nodes incident to arcs in $A(y)$)
$\mathcal{S} = \{i \mid b(i) > 0\}$ (supply nodes)
$\mathcal{D} = \{i \mid b(i) < 0\}$ (demand nodes)
for $P \in \mathcal{P}$, $\Delta(P) = \min\{b(s), -b(t), \min\{y_{ij} \mid (i,j) \in P\}\}$ (capacity of path)
for $W \in \mathcal{W}$, $\Delta(W) = \min\{y_{ij} \mid (i,j) \in W\}$ (capacity of cycle)

Algorithm

$y := x$, $\mathcal{P} = \emptyset$, $\mathcal{W} = \emptyset$, assign $A(y), N(y), \mathcal{S}$, and $\mathcal{D}$. (Initialization)
begin
while $y \neq 0$ **do**
begin
  $s =$ **Select**(y)
  Search(s, y)
  if cycle W found **then do**
  begin
    $\mathcal{W} = \mathcal{W} \cup \{W\}$
    $f(W) = f(W) + \Delta(W)$
    $y_{ij} = y_{ij} - \Delta(W) \ \forall \ (i,j) \in W$
    update $A(y), N(y)$
  end
  if path P found **then do**
  begin
    $\mathcal{P} = \mathcal{P} \cup \{P\}$
    $f(P) = f(P) + \Delta(P)$
    $y_{ij} = y_{ij} - \Delta(P) \ \forall \ (i,j) \in P$
    $b(s) = b(s) - \Delta(P)$ (s starting node of P)
    $b(t) = b(t) + \Delta(P)$ (t ending node of P)
    update $A(y), N(y), \mathcal{S}, \mathcal{D}$
  end
end

From the handout given on the course web page.

Subroutines:

Select(y)
  if $\mathcal{S} \neq \emptyset$, **then** choose $s \in \mathcal{S}$;
  else choose $s \in N(y)$;

Search(s, y)
  Do DFS starting with node s until finding a cycle W in $G(y)$
  or a path P in $G(y)$ from node s to a node $t \in \mathcal{D}$

Flow decomposition theorem

Any nonnegative feasible flow $\bar{x}$ (vector) $(x_{ij} \; \forall (i,j) \in A, \; b(i) \; \forall i \in N)$ can be decomposed into a

(a) sum of flows in paths directed from supply nodes to demand nodes, and

(b) sum of flows along directed cycles.

It will have at most $n+m$ paths and cycles, out of which at most m are cycles.

Corollary Any circulation $\bar{x}$ can be decomposed into the sum of flows around at most m directed cycles.

Notice that we could use the bounds of $n+m$ on the number of cycles and paths for doing the complexity analysis of the flow decomposition algorithm.

We will use the flow decomposition theorem in the proofs of certain results on the shortest path problem, and also later on for other results.

The Shortest Path Problem (Chapter 4 in AMO)

Reminder: length of a path P is the sum of the lengths of arcs in P.

We consider the following general version of the problem:

Given: $G=(N,A)$ with costs c_{ij} and a source node s.
Goal: find the shortest length directed path from s to every $i \in N \setminus \{s\}$.

Assumptions

1. c_{ij} are integers. → helps with complexity analysis.
2. There is a directed path from s to all $i \in N \setminus \{s\}$.
   If such a path is not present for some node $i \in N \setminus \{s\}$, we could add arc (s,i) with $c_{si} = +\infty$ (or some large number).
3. There is no negative cost cycle in G.
   If there is such a cycle, going around it infinitely many times will decrease the total cost without limits! We will have to then impose extra restrictions that we use each arc at most once. But such restrictions make the problem hard to solve.

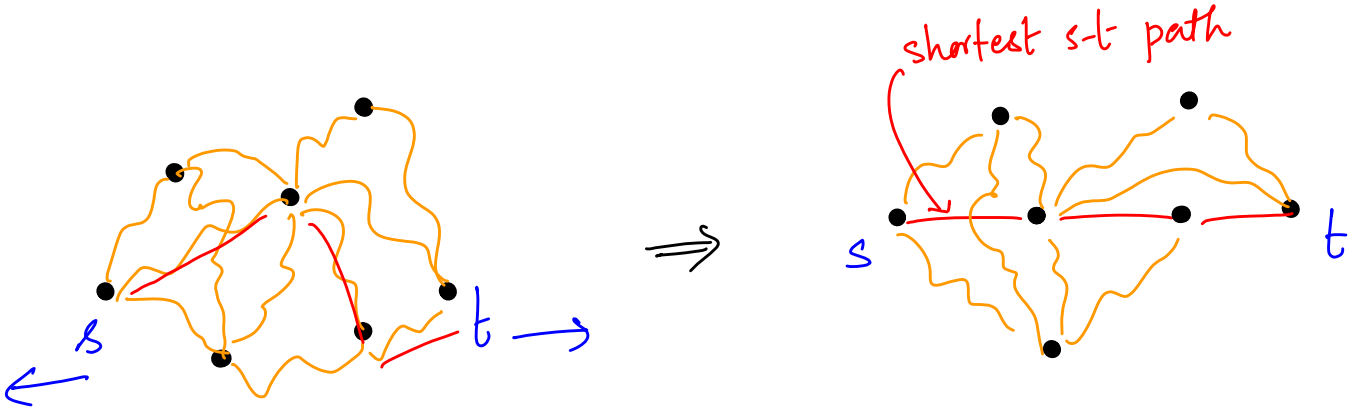
We would need some $c_{ij} < 0$ for negative cycles to exist. But even with some $c_{ij} < 0$, we could have no negative cycles.

To start with, we assume $c_{ij} \geq 0 \; \forall (i,j) \in A$.

The string model

Consider making a network model out of beads and strings, with one bead for each node, and a string connecting beads i and j if $(i,j) \in A$. The length of the string is c_{ij}. Assume the strings are nonelastic.

Consider the s-t version of the shortest path problem. Imagine pulling apart the beads corresponding to the nodes s and t.



The set of strings that become taut first represents a shortest s-t path in the network.

Applications of the Shortest Path problem

Recall: TeX word spacing problem (AMO 1.7).

The book describes many more applications. See AMO Section 4.2.

The Knapsack problem

A hiker wants to decide which items to include in her knapsack. She has to choose from n items. Item i has weight w_i lbs and utility u_i. The knapsack can carry no more than W lbs. The objective is to maximize total utility. We will model this problem as a shortest path problem.

Example $n=4$, $W=6$

i	1	2	3	4
w_i	4	2	3	1
u_i	40	15	20	10

Modeled here as a longest path problem.

Notation:

(i^k) : items $1, 2, \ldots, i$ use up k lbs (of total weight)

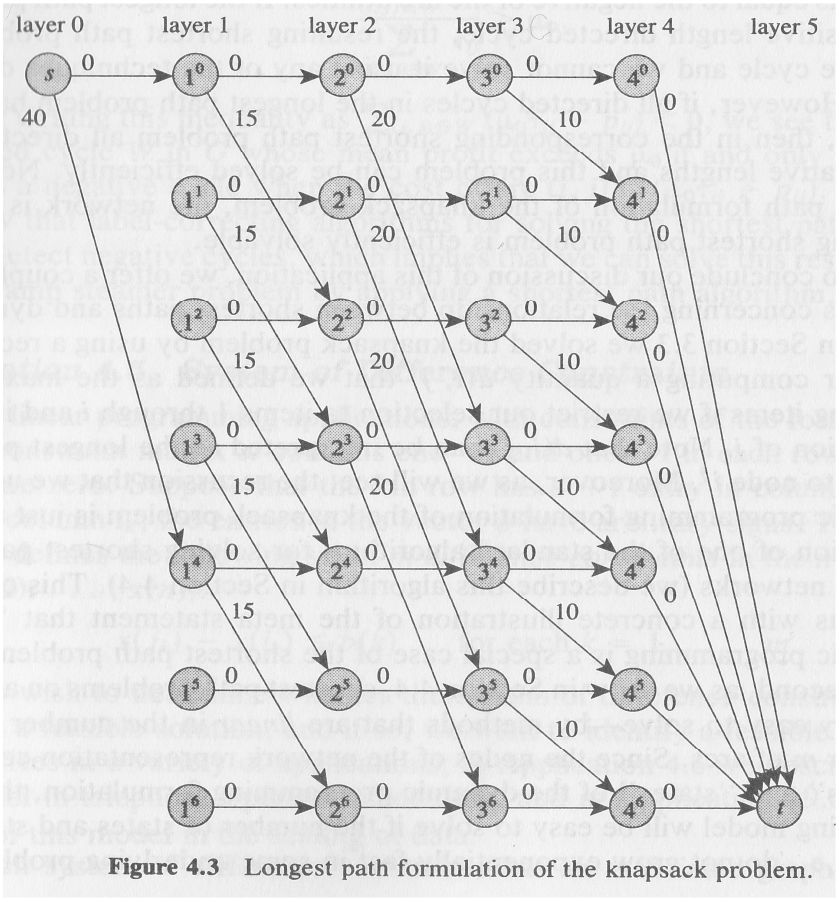
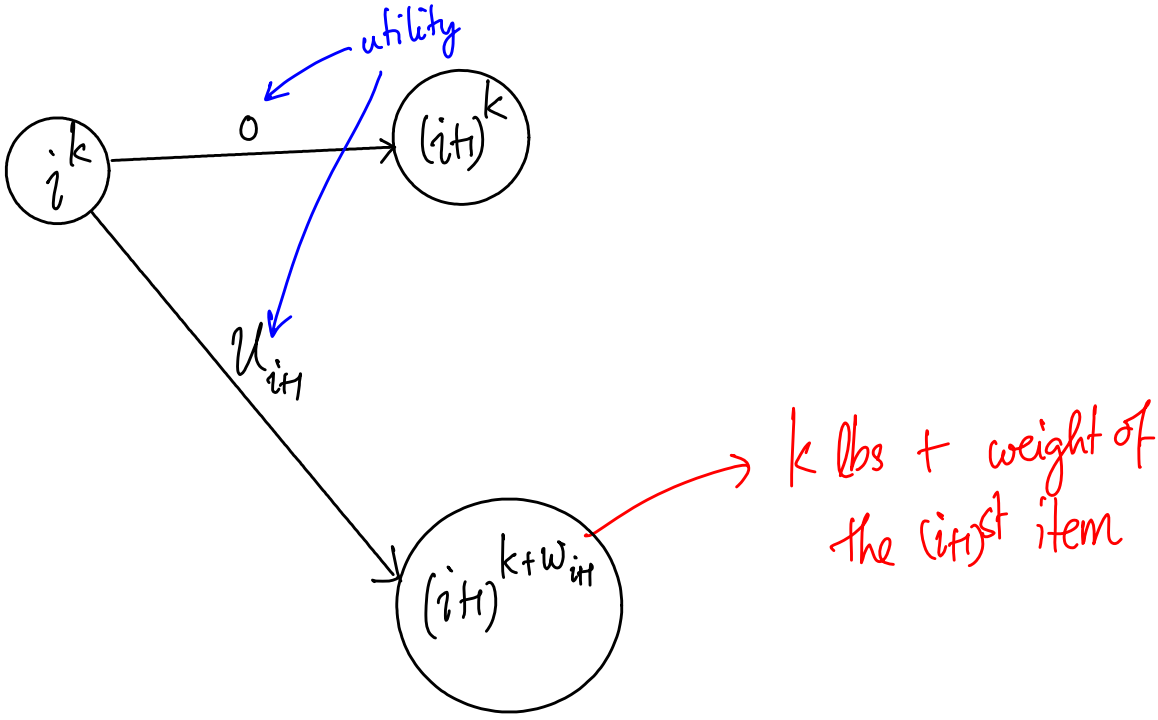


Figure 4.3 Longest path formulation of the knapsack problem.

From any node (i,k), we have two arcs going out, corresponding to us picking item $(i+1)$, or leaving it out.



The longest path problem can be converted to a shortest path problem by negating the assigned c_{ij} values.

The knapsack problem has many applications, including in cryptography, multiprocessor scheduling, fair division, etc. The model illustrated here (with layers corresponding to each item and levels corresponding to the total weight) is a typical instance of dynamic programming. Hence, under mild assumptions, many dynamic programming problems could be cast as shortest path problems.

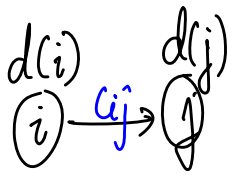
Algorithms for the shortest path (SP) problem

Let $d(\cdot)$ denote a vector of temporary distance labels.

$d(i)$ = length of some path from s to i.

$d(i)$ is an upper bound on the shortest path length (from s to i).

A key step in most SP algorithms



Procedure UPDATE(i)

for each $(i,j) \in A(i)$ do

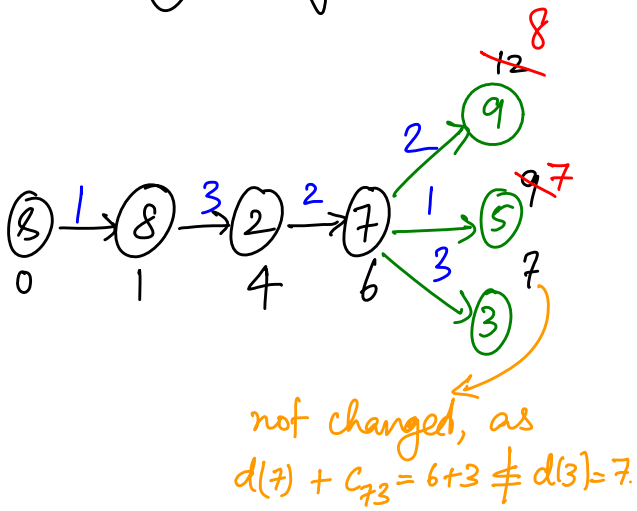
if $d(j) > d(i) + c_{ij}$ then

$d(j) := d(i) + c_{ij}$;

pred(j) := i;

end_if

end_for



When the $d(i)$'s have become "permanent", they represent the shortest path distances from s to each node i.

MATH 566 – Lecture 10 (09/25/2014)

The UPDATE step: if $d(j) > d(i) + c_{ij}$ then

$d(j) := d(i) + c_{ij}$;

$pred(j) := i$;

Dijkstra's algorithm

S: set of "permanent" nodes ($d(\cdot)$ is permanent)

$\bar{S}$: set of nodes with "temporary" $d(i)$'s

```
algorithm Dijkstra;
begin
    S := ∅; S̄ := N;
    d(i) := ∞ for each node i ∈ N;
    d(s) := 0 and pred(s) := 0;
    while |S| < n do
    begin
        let i ∈ S̄ be a node for which d(i) = min{d(j) : j ∈ S̄};
        S := S ∪ {i};
        S̄ := S̄ − {i};
        for each (i, j) ∈ A(i) do
            if d(j) > d(i) + c_ij then d(j) := d(i) + c_ij and pred(j) := i;
    end;
end;
```

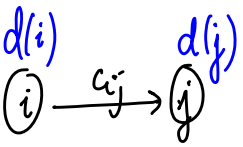
} → node selection

} UPDATE

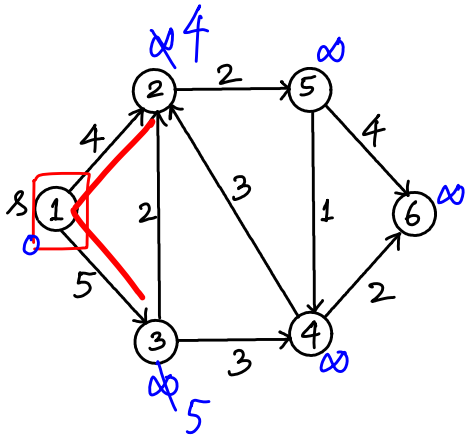
Figure 4.6 Dijkstra's algorithm.

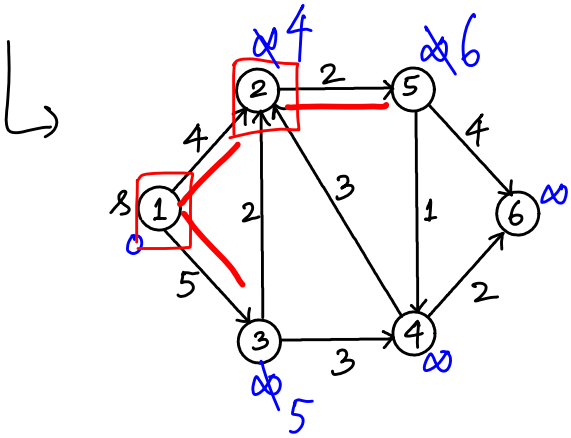
Pick a node i with smallest $d(i)$ from $\bar{S}$, make it permanent, i.e, move it to S. Then run UPDATE on the arcs in $A(i)$.

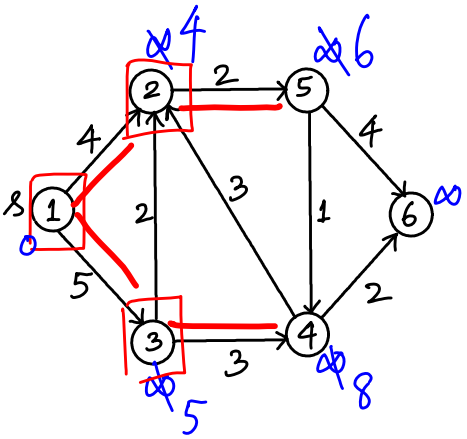
An example

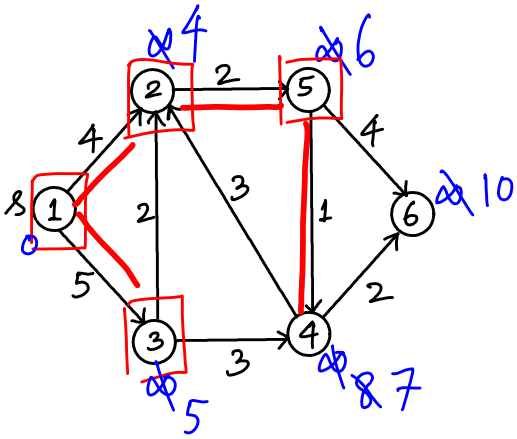


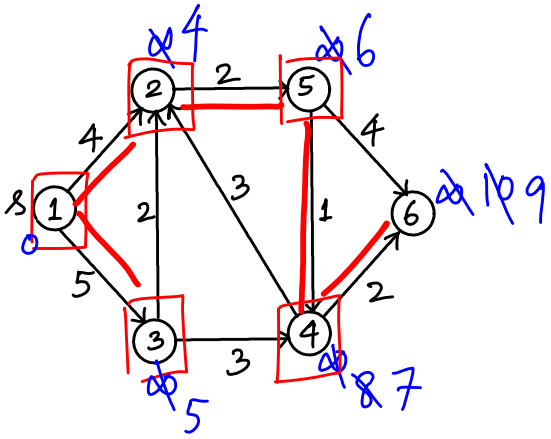
(i) : $i \in S$.

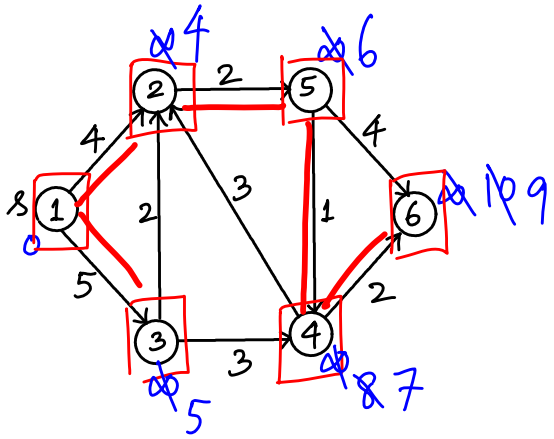












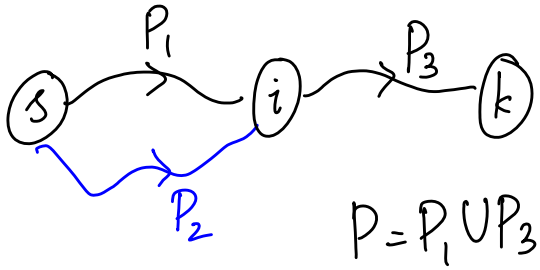
We get the shortest path tree (maintained using pred(i) indices).

Correctness of Dijkstra's algorithm

We first prove certain properties of shortest paths.

Property 1 If P is a shortest path from s to a node k, then any subpath of P from s to a node $i \in P$ is a shortest path from s to i.

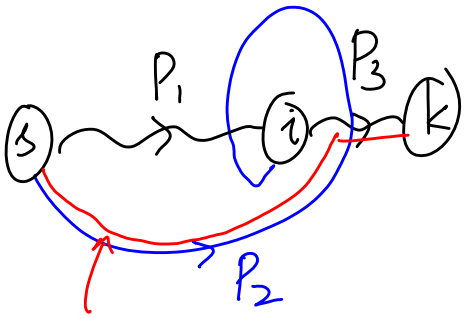
Let the shortest path P from s to k be broken into P_1 from s to node i and P_3 from i to k. The property is saying that P_1 is an SP from s to i.



$P = P_1 \cup P_3$ v/s $P_2 \cup P_3$

Assume there is another path P_2 from s to i that is shorter than P_1. Then $P_2 \cup P_3$ is a directed walk from s to k.

As an application of flow decomposition, any directed walk can be broken into a union of directed paths and directed cycles.



the directed path part of the directed walk

Since all $c_{ij} \geq 0$, the directed cycles will all have positive total costs, and hence the directed paths as part of the decomposition will have a total cost that is smaller than that of P — a contradiction.

Property 2

Let $d(\cdot)$ be the shortest path distances. A directed path P from s to k is a shortest path iff $d(j) = d(i) + c_{ij}$ for all $(i,j) \in P$.

($\Rightarrow$) Let P be a shortest path. Using Property 1 repeatedly, we get $d(j) = d(i) + c_{ij} \quad \forall (i,j) \in P$.

($\Leftarrow$) $d(j) = d(i) + c_{ij} \quad \forall (i,j) \in P$.

Let P be $s = i_1 - i_2 - \cdots - i_h = k$. With $d(i_1) = d(s) = 0$, we can add the above equations for all arcs in P:

$$d(k) = d(i_h) = d(i_{h-1}) + c_{i_{h-1} i_h}$$

$$= d(i_{h-2}) + c_{i_{h-2}, i_{h-1}} + c_{i_{h-1}, i_h}$$

$$\vdots$$

$$= \sum_{(i,j) \in P} c_{ij} \quad (\text{as } d(i_1) = d(s) = 0).$$

Hence the length of path P is equal to $d(k)$, which is the shortest path distance. So, P is a shortest path.

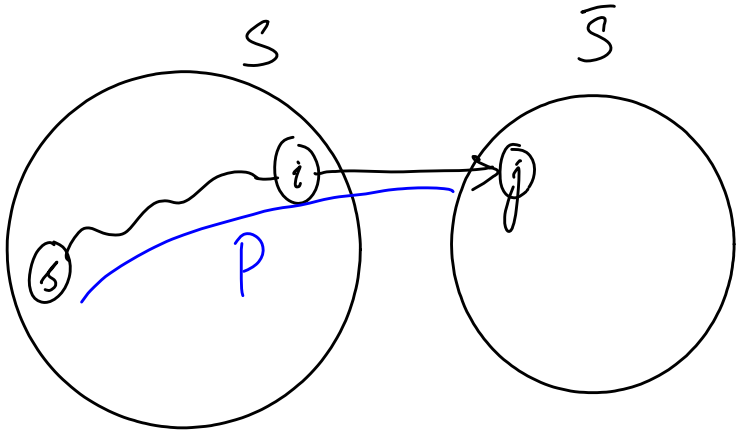
Back to correctness of Dijkstra's algorithm

Recall, S is the set of nodes labeled permanently. We use induction on the cardinality of S.

"At" any iteration

(1) $\forall i \in S$, $d(i)$ is optimal, ie., $d(i)$ does not increase and $d(i) \le d(j)\ \forall j \in \bar{S}$, as well as that $d(i)$ is the shortest path length from s to i.

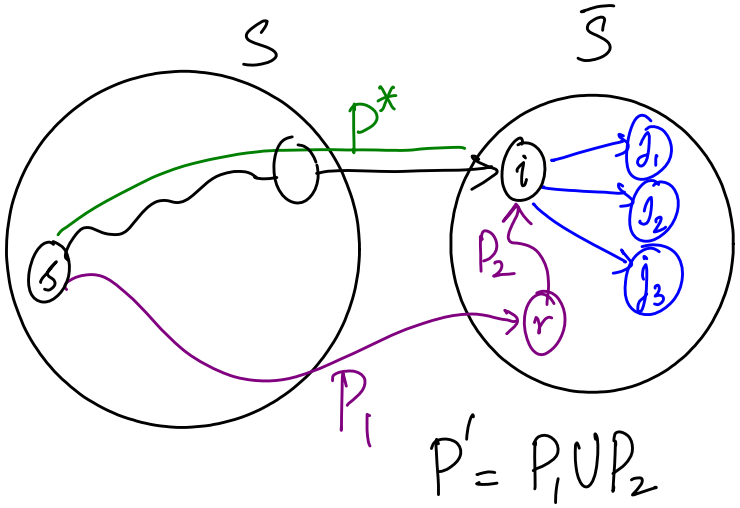
(2) $\forall j \in \bar{S}$, $d(j)$ is the length of the shortest path P from s to j such that all nodes $k \in P$ satisfy $k \in S \cup \{j\}$. (if there is no such path, $d(j) = \infty$).



We assume results (1) and (2) hold true for $|S| = k-1$.

In the k^{th} iteration, say we choose node $i \in \bar{S}$ with $d(i) = \min_{j \in \bar{S}} d(j)$, and move it to S.

We prove that results (1) and (2) hold after this move and the associated UPDATES also.



By induction assumptions, $d(i)$ is the length of a shortest path from s to i (through P^*). To show $d(i)$ is optimal, we show the length of any other path P containing at least one other node $r \in \bar{S}$ is $\geq d(i)$.

$P' = P_1 \cup P_2$ length $(P_1) = d(r)$. length $(P_2) \geq 0$.

By how we chose i (as $\min_{j \in \bar{S}} d(j)$), $d(i) \leq d(r)$.

So length of $P' \geq d(i)$.

We will finish the proof in the next lecture.

The approach used in this proof is a typical one used to prove correctness of similar algorithms. We first assert one or more properties that hold true at any iteration, which guarantee optimality up to that instance. Then we show that the operations of the current iteration preserve these properties.

MATH 566 – Lecture 11 (09/30/2014)

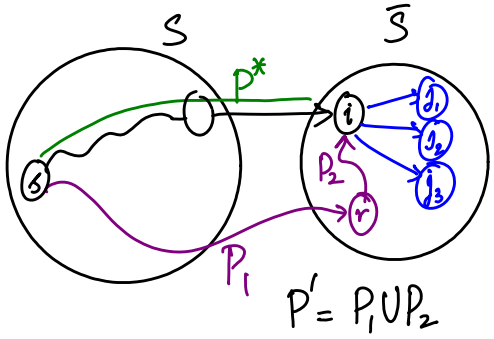
Proof of correctness of Dijkstra's algorithm (continued...)

Recall:

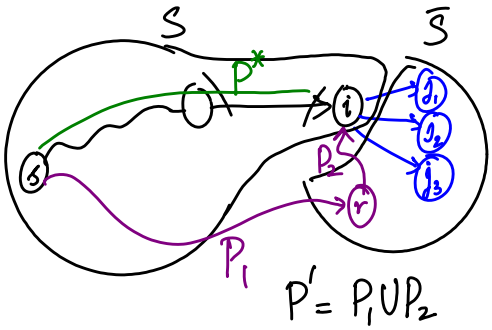
"At" any iteration

(1) $\forall i \in S$, $d(i)$ is optimal, ie., $d(i)$ does not increase and $d(i) \le d(j)\ \forall j \in \bar{S}$, as well as that $d(i)$ is the shortest path length from s to i.

(2) $\forall j \in \bar{S}$, $d(j)$ is the length of the shortest path P from s to j such that all nodes $k \in P$ satisfy $k \in S \cup \{j\}$. (if there is no such path, $d(j) = \infty$).



after iteration:



Continuing the inductive argument, after UPDATE(i), $d(j)$ gets updated to $d(i) + c_{ij}$ $\forall (i,j) \in A(i)$.

After these distance updates, $d(j) \ge d(i)$ holds for all $i \in S$, $j \in \bar{S}$ (recall that we did not change $d(i)$ as part of the update). Also, path from s to j will now satisfy $d_\ell = d_k + c_{k\ell}$ for all (k,ℓ) in the path (by induction hypothesis). Hence, the result (2) above also holds.

We are selecting nodes from $\bar{S}$ in the increasing order of shortest path distances from s.

Complexity of Dijkstra's algorithm

```
algorithm Dijkstra;
begin
    S := ∅; S̄ := N;
    d(i) := ∞ for each node i ∈ N;
    d(s) := 0 and pred(s) := 0;
    while |S| < n do
    begin
        let i ∈ S̄ be a node for which d(i) = min{d(j) : j ∈ S̄};
        S := S ∪ {i};
        S̄ := S̄ − {i};
        for each (i, j) ∈ A(i) do
            if d(j) > d(i) + c_ij then d(j) := d(i) + c_ij and pred(j) := i;
    end;
end;
```

} node selection

} UPDATE

Figure 4.6 Dijkstra's algorithm.

(1) The UPDATE step runs $\sum_{i \in N} |A(i)| = m$ times, i.e., it takes $O(m)$ time.

(2) Node selection: selects one node n times (at most). Each time, scan all nodes in $\bar{S}$ for smallest $d(j)$. So, the running time is $n + (n-1) + (n-2) + \cdots$, i.e., $O(n^2)$.

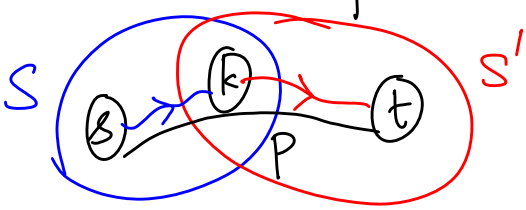
Theorem Dijkstra's algorithm solves the shortest path problem (with $c_{ij} \geq 0$) in $O(n^2)$ time.

Variations of Dijkstra's algorithm

1. Reverse Dijkstra's algorithm: Given a terminus node t, find the shortest paths from all $i \in N \setminus \{t\}$ to t. ($c_{ij} \geq 0$, $\exists$ a directed path from i to t $\forall i \in N \setminus \{t\}$ - if not, add (i,t) with $c_{it} = \infty$).

   We reverse the sense of Dijkstra's algorithm here by examining $AI(i)$ lists. Maintain lists S' and $\bar{S}'$ of permanently and temporarily marked nodes. Pick $i \in \bar{S}'$ with $d(i) = \min_{j \in \bar{S}'} \{d(j)\}$, and move it to S'. Then do UPDATE on $AI(i)$. So, if $d(j) > c_{ji} + d(i)$ for $(j,i) \in AI(i)$, then $d(j) := c_{ji} + d(i)$.

2. Bidirectional Dijkstra's algorithm for the shortest s-t path: Run forward Dijkstra from s and reverse Dijkstra from t. When the same node gets permanently labeled by both runs, we have a shortest s-t path.



This approach often works well in practice, even though the worst case complexity is $O(n^2)$.

Dial's implementation (of Dijkstra's algorithm)

The bottleneck step for the default Dijkstra's algorithm is the node selection step. Dial's implementation improves the time for node selection by storing nodes with finite temporary labels in a sorted fashion.

Property The permanent distance labels maintained by Dijkstra's algorithm are nondecreasing.

Recall, $C = \max_{(i,j)\in A} (c_{ij})$ ($c_{ij} \geq 0$ here)

Dial's implementation creates <u>buckets</u> from 0 to nC, as nC is the largest finite $d(j)$ possible.

* $\text{BUCKET}(k) = \{j \in \bar{S} \mid d(j) = k\}$.

Store BUCKETS in increasing order of k.

* Whenever $d(\cdot)$ is updated, update BUCKETs too.
* FINDMIN → procedure to look for the first nonempty BUCKET, and delete nodes in this BUCKET (after permanently labeling them).

If we maintain each BUCKET as a doubly linked list, we can insert and delete nodes in $O(1)$ time.
↳ constant time

These UPDATE operations take $O(m)$ time.

Complexity

BUCKETs needed = $O(nC)$

time for updates on BUCKETs = $O(nC)$ → maintenance

UPDATEs = $O(m)$

time for FINDMIN = $O(nC)$

total running time is $O(m+nC)$. ← pseudopolynomial algorithm

Could be improved — need not store nC BUCKETs, C would do.

Label Correcting Algorithms for SPP (Chapter 5 AMO)

Dijkstra's algo is a label setting algorithm.

Optimality Conditions

We use distance labels $d(\cdot)$, but now specify conditions that $d(i)$ must satisfy to be optimal, i.e., to represent SP distances from s to i.

Recall: $d(j)$ is the length of some s-j path, and is hence an upper bound on the SP length.

Theorem (5.1) $d(j)\ \forall\ j \in N$ represent shortest path distances iff

$$d(j) \le d(i) + c_{ij} \quad \forall (i,j) \in A \qquad (1)$$

(1) the SP optimality conditions

iff: necessary and sufficient

Proof ($\Rightarrow$) Let $d(j)$ be shortest path distances from s to j. Assume (1) is not true. Hence $d(j) > d(i) + c_{ij}$ for some $(i,j) \in A$. Then we can improve the shortest path length from s to j by taking the path from s to i, and (i,j), which gives a contradiction.

($\Leftarrow$) Let $d(j)$ be some distance labels from s to j satisfying (1).

So, $d(j)$s are upper bounds on lengths of SPs from s to j.

Consider any path P from s to j. Add (1) for each arc $(i,k) \in P$. We get

$$d(j) \leq \sum_{(i,k) \in P} c_{ik} \quad \text{(since } d(s) = 0\text{)}.$$

Hence $d(j)$ is a lower bound on the length of any s-j path.

$\Rightarrow$ $d(j)$ is the length of an s-j SP.

MATH 566 – Lecture 12 (10/02/2014)

Next Thursday, Oct 9, is the midterm exam.
- in class, closed book, no electronic devices

Topics from chapters 1-4.

Practice midterm posted

Hw6 (3 problems from chapter 4) is due on Tuesday, Oct 7.

Shortest path optimality conditions

$d(\cdot)$ represent SP distances iff $d(j) \le d(i) + c_{ij}\ \forall (i,j) \in A$.

These conditions apply even when (some) $c_{ij} < 0$.

We now assume c_{ij} could be negative, but there still do not exist any negative cost cycles. Later on, we will describe how our algorithms could detect negative cycles, if present.

Generic label-correcting algorithm for SP

* maintain $d(\cdot)$ and $pred(\cdot)$
* successively update $d(\cdot)$ until they satisfy all optimality conditions.

$d(j) = \infty \ \forall\, j \in N;$

$d(s) = 0;$

$pred(s) = 0;$

while $d(j) > d(i) + c_{ij}$ for some $(i,j) \in A$ do

begin

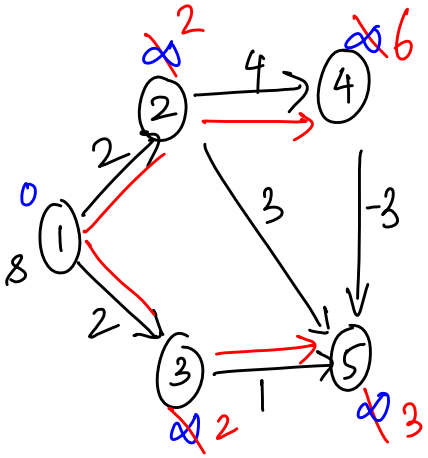
$d(j) = d(i) + c_{ij};$

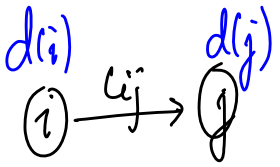
$pred(j) = i;$

end

end

Example:





We could, alternatively, have updated $d(5)$ to $d(2) + c_{25} = 5$ before checking $(3,5)$. In that case, we would set $d(5) = 3$ after checking $(3,5)$ or $(4,5)$.

Proof of finiteness (Show that the algorithm terminates after a finite number of iterations)

Assume c_{ij} are integers.

* $-nC \leq d(j) \leq nC$ $\qquad C = \max_{(i,j)\in A} \{|c_{ij}|\}$

* In each iteration (of the while loop), at least one $d(j)$ is decreased by at least 1 unit.

  $\Rightarrow$ any label $d(j)$ is updated at most $2nC$ times.

* total number of distance updates is $2n^2C$.

$\Rightarrow$ algorithm performs $O(n^2C)$ iterations.

If c_{ij} are not integral, we have to use a different proof, but the algorithm is still finite.

The complexity of the algorithm is $O(2^n)$.

↪ We will not describe the details. Instead, we talk about efficient, i.e., polynomial time, implementations of the algorithm.

What if there are negative cycles?

The algorithm is no longer finite, but we can stop when any $d(j) < -nC$.

Are there polynomial time versions of this algorithm? **Yes!**

Modified label-correcting algorithm

How to select the arcs violating optimality conditions efficiently?

* maintain a LIST of nodes
* if (i,j) violates the optimality condition, then LIST must contain i.
* when you update $d(j)$, add j to LIST.

```
LIST = {s};
while LIST ≠ ∅ do
    remove i from LIST;
    for all (i,j) ∈ A(i) do
        if d(j) > d(i) + c_ij then
            d(j) = d(i) + c_ij;
            pred(j) = i;
            LIST = LIST ∪ {j};
        end_if
    end_for
end_while
```

Complexity

* $d(j)$ is updated at most $2nC$ times.
  After the update, j is added to LIST. Arcs in $A(j)$ are scanned later

* total # scannings $\leq \sum_{j \in N} 2nC\,|A(j)|$

$\Rightarrow$ modified label correcting algorithm runs in $O(mnC)$ time.

This is a pseudopolynomial time algorithm. We wonder — is there is a polynomial time implementation?

Can be bad when C is exponentially large.

FIFO implementation of label-correcting algorithm

* "pass": scan all arcs in A, update $d(j)$ if $d(j) > d(i) + c_{ij}$.

* Do n passes, or stop if no $d(j)$s change in a pass, whichever comes first.

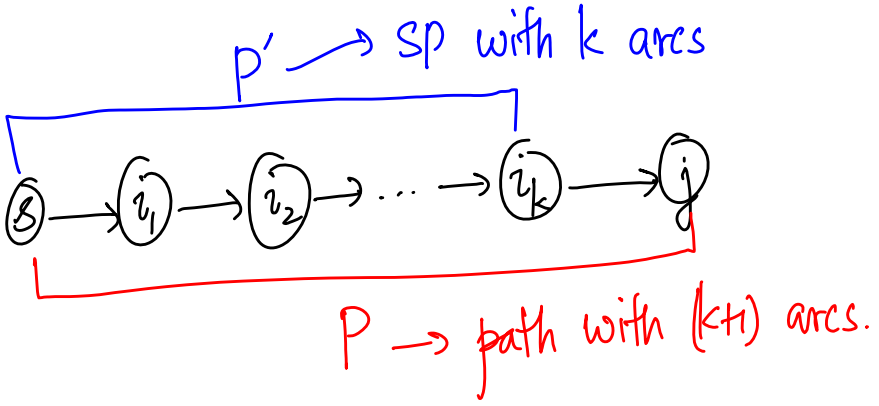
Theorem 5.3 The FIFO label correcting algorithm solves SPP in $O(mn)$ time, or it shows that there exists a negative cycle in the network.

Proof Each pass takes $O(m)$ time (at most m updates)

To show correctness, we argue that the algorithm performs at most $(n-1)$ passes.

Claim At the end of the k^{th} pass, the algorithm computes SP distances for all nodes connected to s by a shortest path consisting of k or fewer arcs.

Induction Assume claim holds after k passes.



After k passes, $d(i_k) = \sum_{(i,\ell) \in p'} c_{i\ell}$, which is the SP distance from s to i_k

In pass $(k+1)$, we update $d(j) = d(i_k) + c_{i_k j}$

$\Rightarrow$ Claim holds after $(k+1)$ passes.

Any SP from s to j has at most $(n-1)$ arcs. Hence, $d(j)$ is optimal for all $j \in N$ after $(n-1)$ passes.

In the n^{th} pass, if there is an update then there is a negative cycle. Else SPP is solved.

Maintain LIST in modified label correcting algorithm as a queue — gives a running time of $O(mn)$.

Treating LIST as a stack (LIFO) works well often in practice but is bad in terms of worst case complexity.

Detecting negative cycles

1. Stop if $d(j) < -nC$.

2. Run FIFO label correcting algorithm, and stop if a node is scanned at least n times.

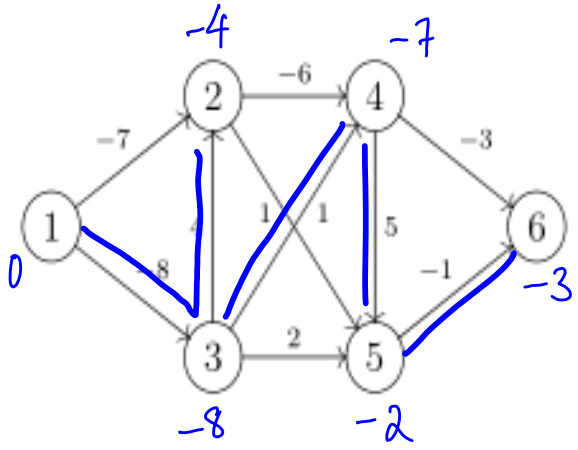
3. Keep track of the number of arcs in SP from s to each node j, and stop if you find more than $(n-1)$ arcs in such a path.

MATH 566 – Lecture 13 (10/07/2014)

Review for midterm – Practice midterm exam

#3) Longest path problem – best to eyeball the longest paths for each node.

d(i)
(i)

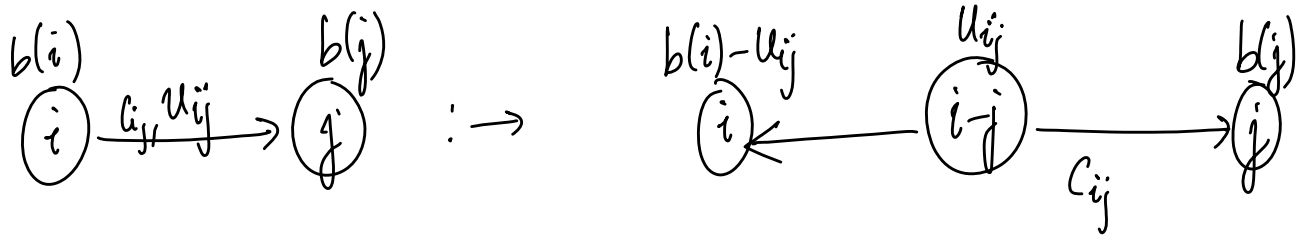


You could indicate the longest path tree and the d(i) values. No other details are necessary here.

Provide justifications where needed!

4. In the transportation problem, $N = N_1 \cup N_2$ with $b(i) > 0 \ \forall i \in N_1$ (supply nodes) and $b(j) < 0 \ \forall j \in N_2$ (demand nodes).

We can remove arc capacities by adding nodes ((i-j) for $(i,j) \in A$), as $u_{ij} < \infty$. This transformation gives the bipartite structure.



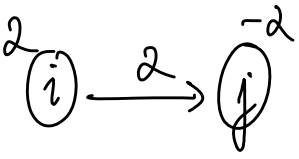
$$b(i) \leftarrow b(i) - \sum_{(i,k)\in A} u_{ik}$$

According to the problem, $\sum_{(i,k)\in A} u_{ik} \geq b(i)$, hence $b(i) \leq 0$ $\forall\, i \in N$ after the transformation.

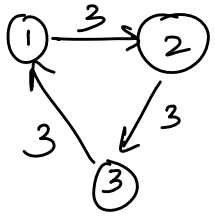
The new nodes (i-j) will be the supply nodes, and original nodes $i \in N$ will be demand nodes.

5 (a) (iv) as DFS and SP use different criteria.

(b) (iv) Depends on the network.

e.g.1 

decomposition into a unique path flow.

or 

→ decomposition into a unique cycle flow.

Review $O(\cdot), \Omega(\cdot), \Theta(\cdot)$ notations!

Use definitions given in class.

6.(a) $f(n) + g(n) = \Theta(\min\{f(n), g(n)\})$

False. The $O(\cdot)$ part does not hold. For instance,

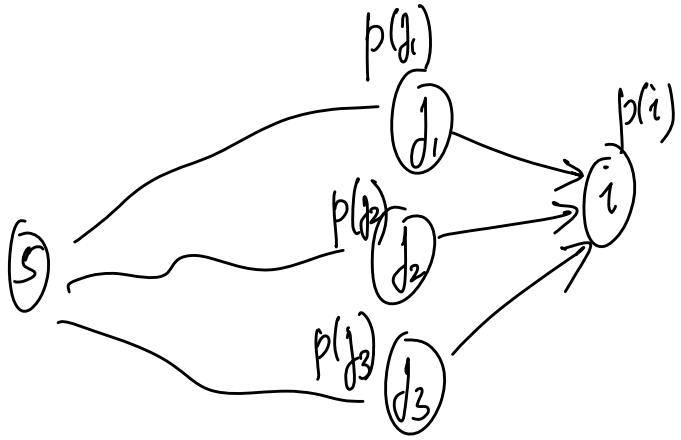
$f(n) = n^2$, $g(n) = n$, so $\min\{f(n), g(n)\} = g(n)$ $\forall n \geq 2$.

But $n^2 + n \neq O(n)$.

Networks are assumed to be directed by default.

(b) $p(i)$ = # distinct s-i paths

acyclic network $\Rightarrow$ topological ordering! $\Rightarrow$ $O(m)$ time!!



set $p(i) = \sum_{(k,i) \in A} p(k)$.

In formulating problems as network flow problems, never use x_{ij}s (flow variables) to define the parameters $c_{ij}, u_{ij}, b(i)$, etc.

8. Look for suggestive structures in problem – for $l_{ij}, u_{ij}, b(i), c_{ij}$, etc.

Here, $y_i \in [0, v_i]$ → suggests an arc with lower bound of 0 and upper bound v_i.

Similarly, $x_i \in [l_i, u_i]$ suggests an arc with lower bound of l_i and upper bound of u_i.

We have bounds and costs, but apparently no supply/demand. So go with circulation problem!

Here is an outline of the structure of various problem classes we have introduced.

Problem classes	cost (c_{ij})	bounds (l_{ij}, u_{ij})	supply/demand ($b(i) \neq 0$)
shortest path	✓	X	X
max flow	X	✓	X
min-cost flow	✓	✓	✓
circulation	✓	✓	X (all $b(i)=0$)
assignment	✓	X $\longleftrightarrow$ X (implied)	✓
transportation	✓	✓	✓

we have $N = N_1 \cup N_2$,

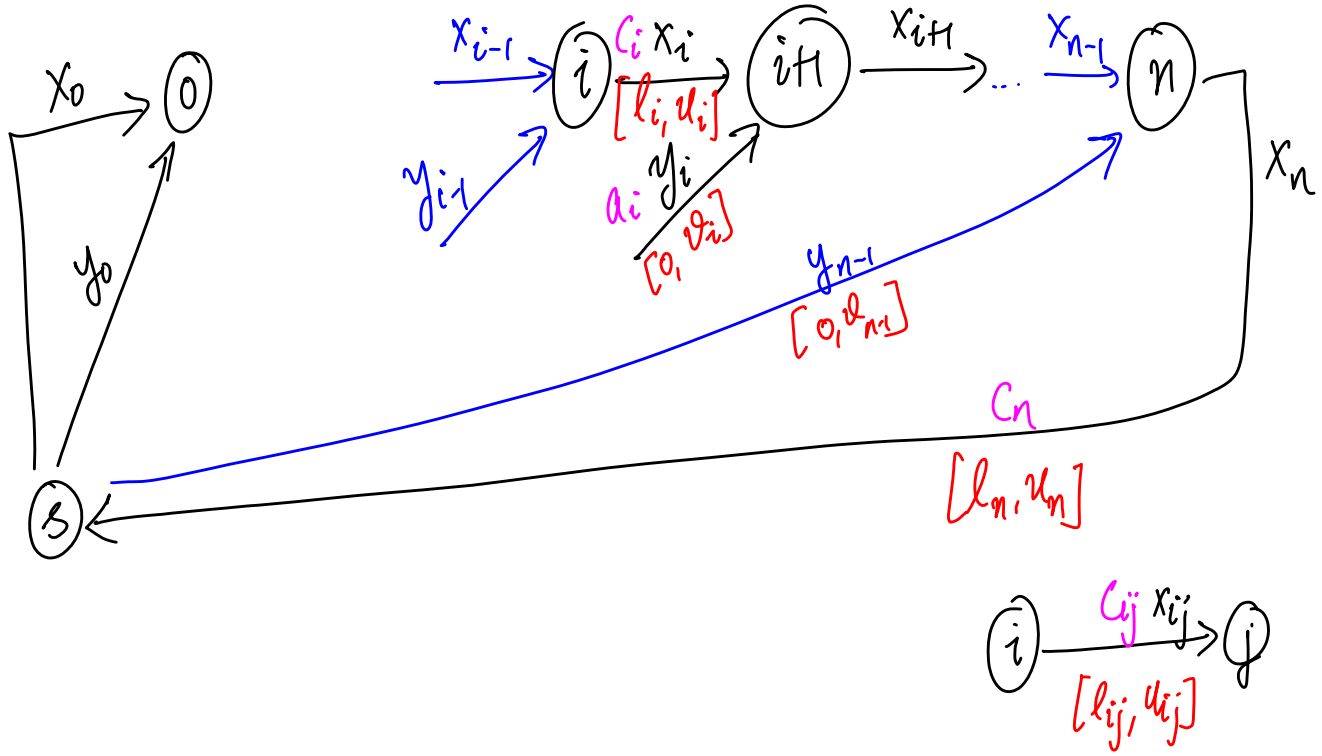
$u_{ij} = 1 \ \forall (i,j)$

with $N = N_1 \cup N_2$ where

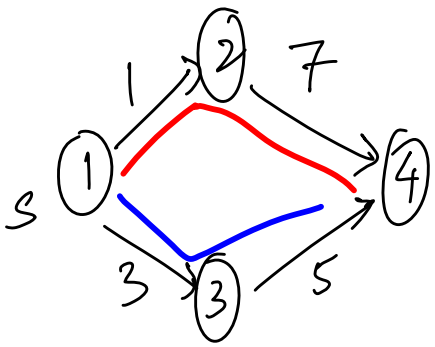
$b(i) > 0 \ \forall i \in N_1$

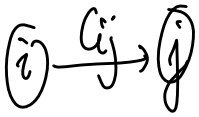
$b(j) < 0 \ \forall j \in N_2$

$x_{i+1} = x_i + y_i$ suggests a flow balance equation.



(7). Notice that we could get a composite total cost for an SP using two sets of prime numbers that are also relatively prime.





two SPs from 1 → 4

MATH 566 – Lecture 15 (10/14/2014)

All-pairs shortest path problem

Find shortest paths from every node i to every node j.

* Can solve n standard shortest path problems, with each node i as the source.

Would be nice if we could use Dijkstra's algo (→ as it is faster), but what about negative c_{ij}'s?

✓ Use **reduced costs**!

$d(j) \le d(i) + c_{ij} \quad \forall (i,j) \in A$ are the SP optimality conditions

Let $c_{ij}^d = c_{ij} + d(i) - d(j) = \left(c_{ij} + d(i)\right) - d(j)$

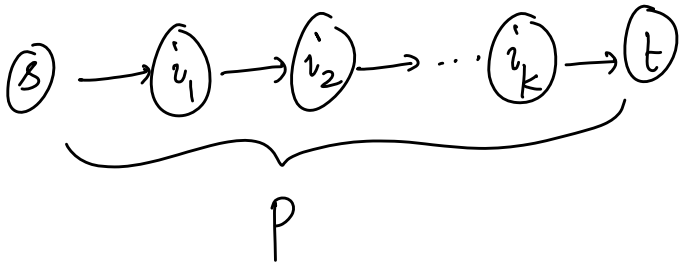
More generally, for a set of node potentials $\pi(i)$, $i \in N$, $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$. Here, we are setting $\pi(i) = -d(i)$.

Notice that if $d(\cdot)$ are SP distances, $c_{ij}^d \ge 0$.

But, could we use c_{ij}^d in place of c_{ij}? Yes!

Property A shortest path w.r.t. c_{ij} is also a shortest path w.r.t c_{ij}^d for a given vector $d(\cdot)$.

Proof



$$\sum_{(i,j)\in P} c_{ij}^{d} = \sum_{(i,j)\in P} c_{ij} + \underbrace{d(s) - d(t)}_{\text{constant if } d(\cdot) \text{ is fixed.}}$$

All-pairs SP algorithm

1. Solve SP with $s=1$ (say). Let $d(j)$ = shortest path length from $s=1$ to j, $\forall j \in N$.
2. For $s = 2$ to n, solve SP w.r.t c_{ij}^{d}.

Complexity

Step 1: FIFO-label correcting algorithm - $O(mn)$

Step 2: Use radix heap implementation of Dijkstra for each SP calculation. $O(m + n\log C)$ for each run from $s=2$ to n. → see AMO for details!

$\Rightarrow$ Overall, complexity is $O(mn + n(m + n\log C))$

(n → $(n-1)$ to be exact)

$= O(mn + n^2 \log C)$.

All-pairs SP optimality Conditions

Assume there are no negative cost cycles.

Let $d[i,j]$ = length of some finite walk from i to j.

Thus, $d[i,j]$ is an upper bound on the SP length from i to j.

Also, let $d[i,i] = 0 \ \forall \ i \in N$, and

$d[i,j] \leq c_{ij} \ \forall \ (i,j) \in A$.

Theorem 5.5 $d[i,j]$ represent shortest path lengths from i to j

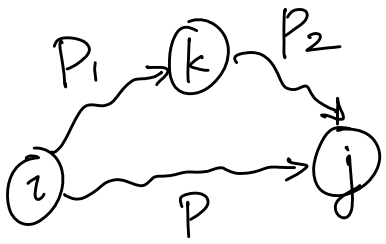
iff $d[i,j] \leq d[i,k] + d[k,j] \ \forall \ i,k,j$.

all possible triplets of nodes

Idea of Proof

($\Rightarrow$) Let $d[i,j]$ represent SP distances from i to j, $\forall i,j \in N$.

Assume $d[i,j] > d[i,k] + d[k,j]$ for some set of nodes i,j,k.



$P_1 \cup P_2$ is a directed walk from i to j. This walk could be decomposed into a union of paths and cycle, from which we could obtain a path from i to j whose total cost is $\leq d[i,k] + d[k,j]$ (as we do not have negative cycles). But this path contradicts the optimality of $d[i,j]$. □

The reverse direction ($\Leftarrow$) is similar to the corresponding proof in the case of the standard SP optimality conditions.

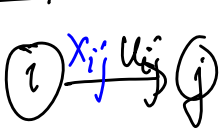
Floyd-Warshall algorithm - repeatedly check for violations of the all-pairs SP optimality conditions, and UPDATE if violations are found. Runs in $O(n^3)$ time.

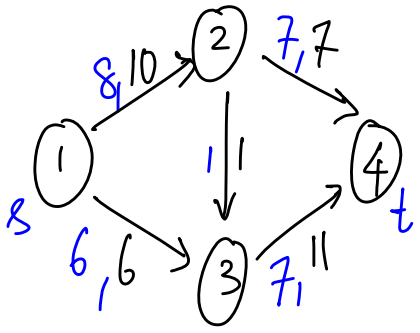
have to check all triplets i,j,k.

The Maximum Flow Problem (Chapter 6)

Goal: Send as much flow as possible, i.e., without exceeding arc capacities, from node s to node t in a capacitated network.

Example





max flow here is 14 units

Do no write the optimization model when asked to formulate an instance of the max flow problem!

Optimization Model

$$\max \quad v$$

$$\text{s.t.} \quad \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = \begin{cases} v & \text{for } i = s \\ 0 & \text{for } i \in N \setminus \{s,t\} \\ -v & \text{for } i = t \end{cases}$$

in place of $b(i)$, which are data/constants. v is a variable here!

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

$v = 14$ in the above example.

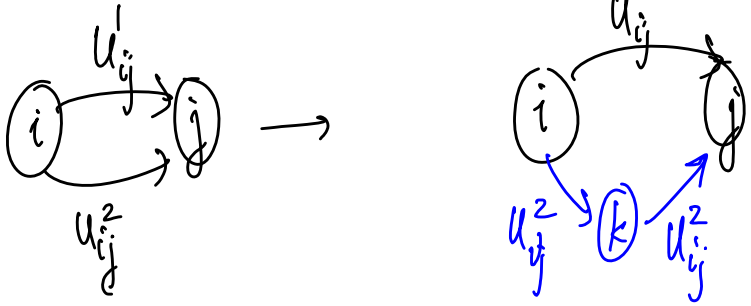
Assumptions

1. $G = (N, A)$ is directed.
2. u_{ij} are non negative integers.
3. G does not contain a directed P from s to t with $u_{ij} = \infty \; \forall (i,j) \in P$.
4. When $(i,j) \in A$, then (j,i) is also present.
   If $(j,i) \notin A$ originally, add (j,i) with $u_{ji} = 0$.

We will see in the next lecture that most algorithms for max flow try to push flow "up" an arc, and possibly pull back some of the flow. Hence we assume (i,j) & $(j,i) \in A$ in "preparation", so to speak.

5. There are no parallel arcs.

   If there are parallel arcs, then use extra nodes to avoid them being parallel.



Applications

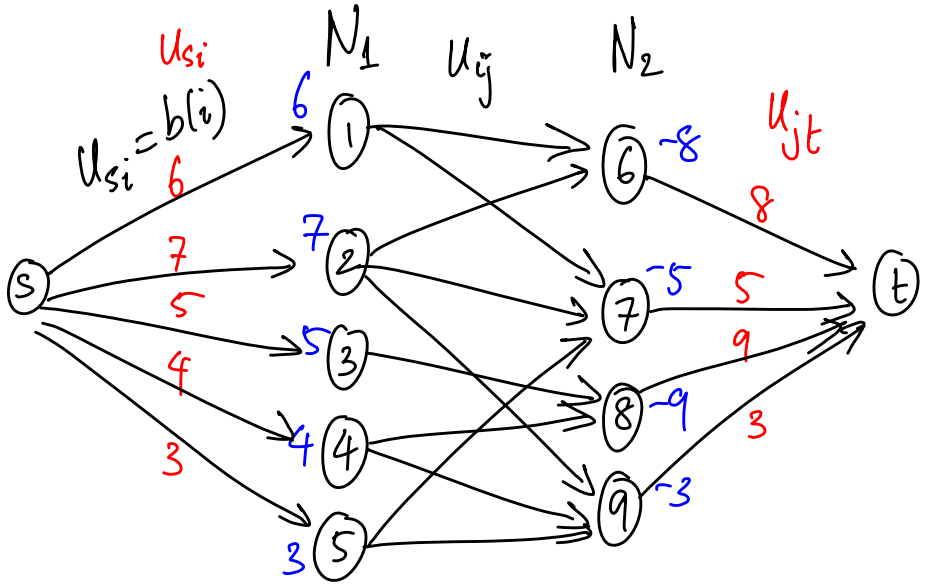
1. **Feasibility problem**: find a feasible flow given $G(N, A)$ with $b(i)$'s and u_{ij}'s.

For instance, the transportation problem (warehouses and retailers).

Can we route all the supply from nodes in N_1 to nodes in N_2 while honoring u_{ij}?

We model this problem as an instance of max-flow.

We add nodes s and t, and arcs $(s,i)\ \forall i \in N_1$, $(j,t)\ \forall j \in N_2$. We then set $u_{si} = b(i)\ \forall i \in N_1$, and $u_{jt} = -b(j)\ \forall j \in N_2$.



If the max flow saturates all arcs $(s,i)\ \forall i \in N_1$ (equivalently, if saturates all arcs $(j,t), j \in N_2$), then there exists a feasible flow. The x_{ij}'s for (i,j) in the original network give the feasible flow.

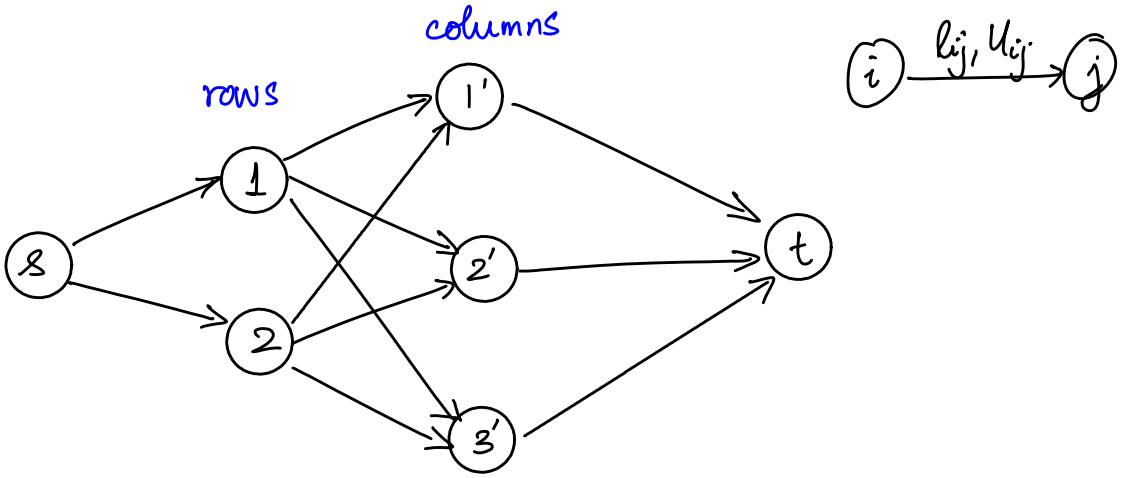
2. Matrix rounding problem

Given: $p \times q$ matrix $D = [d_{ij}]$ and row sums α_i $(1 \le i \le p)$, and column sums β_j $(1 \le j \le q)$. $d_{ij}, \alpha_i, \beta_j \in \mathbb{R}_{\ge 0}$

↙ nonnegative real numbers

	1′	2′	3′	α_i ↓
1	3.1	6.8	7.3	17.2
2	9.6	2.4	0.8	12.8
$\beta_j \rightarrow$	12.7	9.2	8.1	

Goal: Round each element d_{ij} and α_i, β_j such that row (column) sums of rounded elements equal the round row (column) sums.



For (s, i), $i \in$ Rows, we set the lower and upper bounds as $\lfloor \alpha_i \rfloor$ and $\lceil \alpha_i \rceil$. For (j, t), $j \in$ Columns, we set the bounds as $\lfloor \beta_j \rfloor$ and $\lceil \beta_j \rceil$ (floor and ceil).

↙ round down ↘ round up

Similarly, we have $(i,j) \in A$ for every element (i,j) in D, whose bounds are set to $\lfloor d_{ij} \rfloor$ and $\lceil d_{ij} \rceil$.

We then solve the s-t max flow problem. In the optimal solution, every flow x_{ij} will be equal to either l_{ij} or u_{ij}, thus providing a consistent rounding.

MATH 566 – Lecture 16 (10/16/2014)

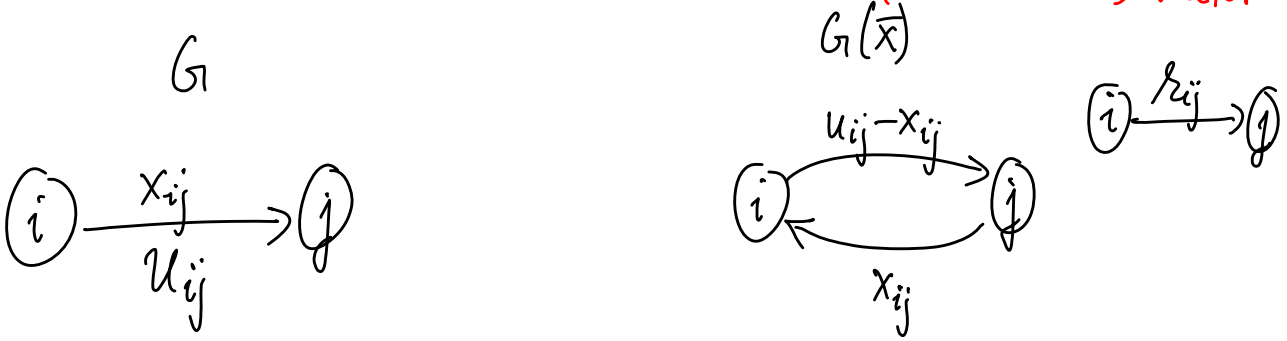
Offer for changing weight of midterm exam

* If you score 90%+ on the remaining homeworks, that high score will count for 10% (out of the 25%) weight of the midterm.
* If you score 90%+ on the project, that score will replace 10% (out of 25%) of your midterm weight.

Residual Network ("remaining flow" network)

Algorithms for max flow will employ the concept of residual networks.

$\bar{x}$ → vector of flows x_{ij}
"bar" ⇒ vector



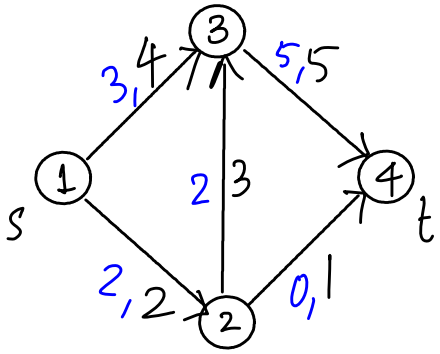
Notice that the residual network is defined for a particular flow x.

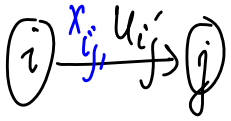
The **residual capacity** of arc (i,j) is defined as

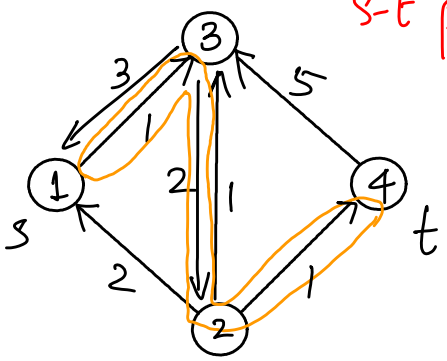
$$r_{ij} = u_{ij} - x_{ij} + x_{ji}.$$

The maximum additional flow that can be sent from i to j using (i,j) and by reversing flow in (j,i).

The residual network $G(\bar{x})$ has the same node set as G, and all arcs (i,j) with $r_{ij} > 0$.







$G(x)$

this is an augmenting path

If there is a (directed) path from s to t in $G(\bar{x})$, we could push flow along this path. This idea is central to max flow algorithms — start with some flow $\bar{x}$, find $G(\bar{x})$, and send flow along s-t paths in $G(\bar{x})$.

Def A path P from s to t in $G(x)$ is an **augmenting path**.

How much flow could we send from s to t along an augmenting path?

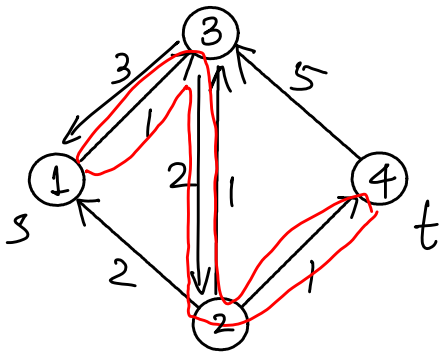
The **residual capacity** of an augmenting path P is

$$\delta(P) = \min_{(i,j)\in P} \{r_{ij}\}.$$

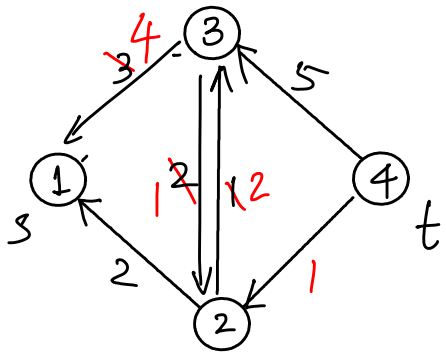
Note that $\delta(P) > 0$ for any augmenting path P, as $r_{ij} > 0$.

To **augment** along P is to send $\delta(P)$ units of flow along each arc of the path P, and modify $\bar{x}$ (the flow) as well as $G(\bar{x})$, i.e., r_{ij}'s, accordingly.

Back to the example:



$G(x)$



$P = 1\text{-}3\text{-}2\text{-}4$, $\delta(P) = 1$

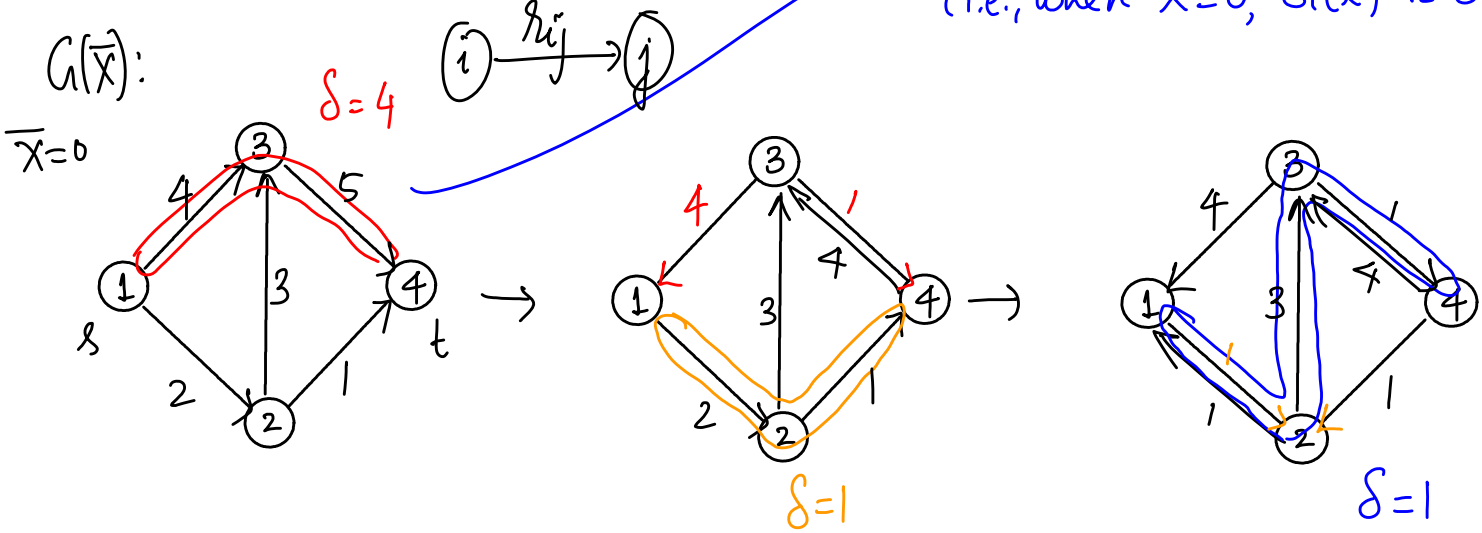
There are no more augmenting paths in $G(x)$, so the corresponding flow is maximum.

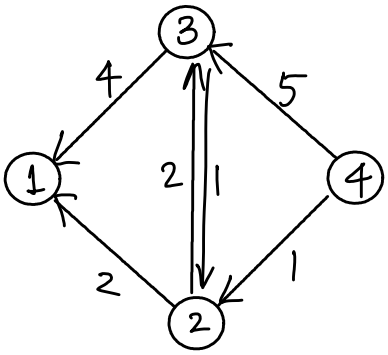
The generic augmenting path algorithm (Ford-Fulkerson)

```
begin
  x̄ = 0;
  while G(x̄) has a path P from s to t do
  begin
    augment δ(P) units of flow along P;
    Update G(x̄) and x̄;
  end
end_while
```

Notice that $G(\bar{0}) = G$ itself (i.e., when $\bar{x}=0$, $G(\bar{x})$ is G).

$G(\bar{x})$:





No more augmenting paths, hence flow is maximum

How do we recover the original (max) flows once the algorithm terminates? Recall that we have

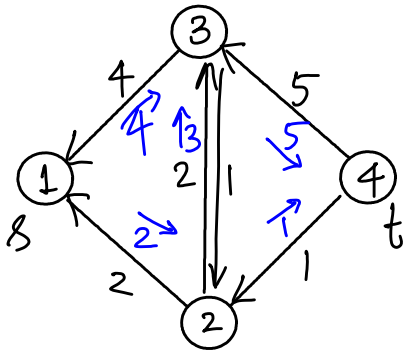
$$r_{ij} = u_{ij} - x_{ij} + x_{ji}$$

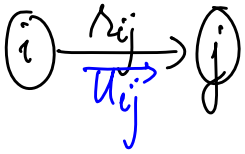
$$\Rightarrow \quad x_{ij} - x_{ji} = u_{ij} - r_{ij}.$$

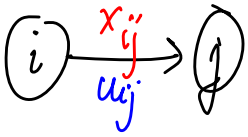
If $u_{ij} \geq r_{ij}$, then set $x_{ij} = u_{ij} - r_{ij}$, $x_{ji} = 0$.

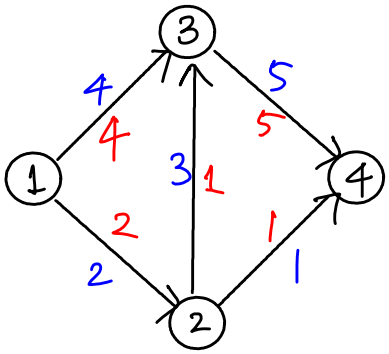
else, set $x_{ij} = 0$ and $x_{ji} = r_{ij} - u_{ij}$.

↳ $u_{ij} < r_{ij}$









$v = 6$ (4+2 out of ① or 5+1 into ④)

↳ value of max flow

We first prove finiteness of the generic algorithm, and then consider details of implementation (to get polynomial time implementations).

Proof of finiteness of the algorithm

Recall, we assume u_{ij} are integers (≥ 0).

1. Lemma The residual capacities are integral after each augmentation.

2. The capacity $\delta(p)$ of each augmenting path p is at least 1.

3. Augmenting along a path p reduces r_{si} for some arc (s,i) by at least 1 unit.

4. Augmentation does not increase r_{si} for any (s,i).

5. $\sum_{(s,i)\in A} r_{si}$ keeps decreasing, and is bounded below by 0 (as $\sum r_{si} \geq 0$).

Hence the # augmentations $= O(nU)$, where $U = \max_{(i,j)\in A}\{u_{ij}\}$. Here, we could specify $U = \max_{(s,i)\in A}\{u_{si}\}$.

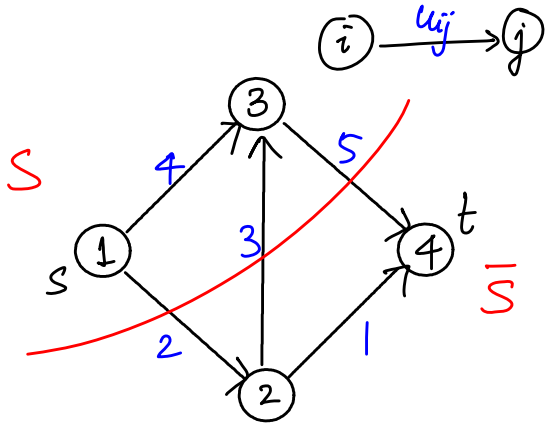
So, we could have an arc with a much larger capacity further down the network (i.e., farther away from s). But since we are sending flow out of s, we may not be able to use this high-capacity arc to its full.

How do we know a flow is maximum?

1. No augmenting path in $G(x)$
2. Max-flow min-cut theorem (duality)

Def An **s-t cut** is a partition of the node set N into disjoint sets $S, \bar{S}$ such that $s \in S$, $t \in \bar{S}$.

A **forward arc** of the cut is an arc $(i,j) \in A$ with $i \in S$, $j \in \bar{S}$. Similarly, a **backward arc** of the cut is an arc $(j,i) \in A$ with $j \in \bar{S}$, $i \in S$.



$S = \{1, 3\}$, $\bar{S} = \{2, 4\}$.

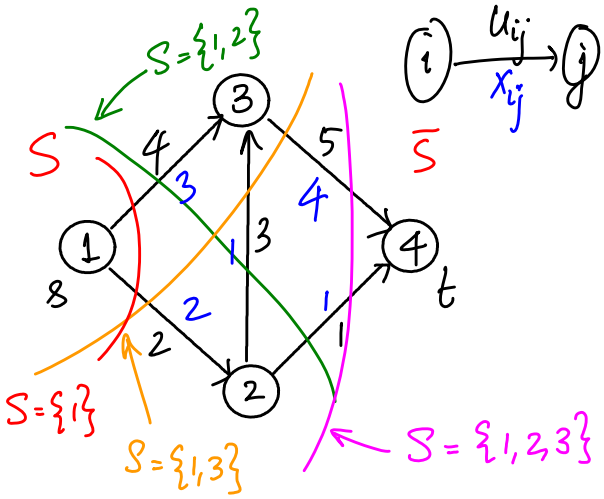
The **capacity** of a cut is $U[S, \bar{S}] = \sum_{\substack{(i,j) \in A \\ i \in S \\ j \in \bar{S}}} u_{ij}$.

Sum of the capacities of all forward arcs.

MATH 566 – Lecture 17 (10/21/2014)

Capacity of cut $[S,\bar{S}]$:

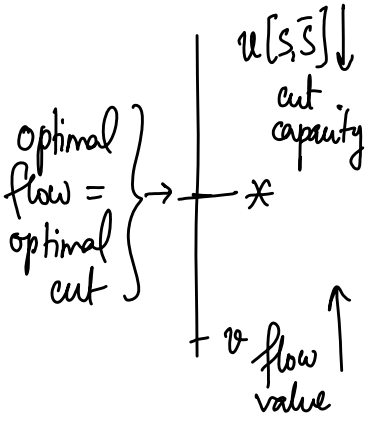
$$u[S,\bar{S}] = \sum_{\substack{i\in S,\\ j\in\bar{S}}} u_{ij}$$



Property 6.1 The value v of any feasible flow $\bar{x}$ is at most $u[S,\bar{S}]$ of any s-t cut $[S,\bar{S}]$.

Def The flow across s-t cut $[S,\bar{S}]$ is

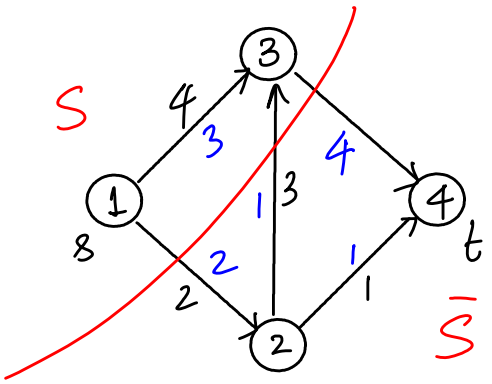
$$F_{\bar{x}}[S,\bar{S}] = \sum_{i\in S, j\in\bar{S}} x_{ij} - \sum_{i\in S, j\in\bar{S}} x_{ji}$$



Example: In the picture

$$F_{\bar{x}}[S,\bar{S}] = x_{12} + x_{34} - x_{23} = 2+4-1 = 5.$$

$\{1,3\}$ $\{2,4\}$



Claim 1 For an s-t cut $[S, \bar{S}]$, $F_{\bar{x}}[S, \bar{S}] = v$.

Proof Add the flow balance equations for all $i \in S$.

Recall that $\sum_{(s,j) \in A} x_{sj} - \sum_{(j,s) \in A} x_{js} = v$ is the flow balance equation for s.

We will be left with arc flows x_{ij} for $i \in S, j \in \bar{S}$, and $-x_{ji}$ for $i \in S, j \in \bar{S}$. The flows within S will cancel.

Claim 2 $F_{\bar{x}}[S, \bar{S}] \leq u[S, \bar{S}]$ for s-t cut $[S, \bar{S}]$.

For $i \in S, j \in \bar{S}$, $x_{ij} \leq u_{ij}$, and $x_{ji} \geq 0$.

So $$F_{\bar{x}}[S, \bar{S}] = \sum_{i \in S, j \in \bar{S}} x_{ij} - \sum_{i \in S, j \in \bar{S}} x_{ji}$$

$$\leq \sum_{i \in S, j \in \bar{S}} u_{ij} - \sum_{i \in S, j \in \bar{S}} 0$$

$$= u[S, \bar{S}].$$

The Max-Flow Min-Cut Theorem (MFMC theorem)

Optimality conditions for max flows.

The following statements are equivalent.

(1) The flow $\bar{x}$ is maximum.
(2) There is no augmenting path in $G_1(\bar{x})$.
(3) There is an s-t cut $[S, \bar{S}]$ whose capacity is equal to the value v of the flow $\bar{x}$.

Corollary (MCMF Theorem). The maximum flow value is the capacity of the minimum cut.

Proof (1) $\Rightarrow$ (2) (Equivalently, not(2) $\Rightarrow$ not(1)).

If there is an augmenting path in $G_1(\bar{x})$, then $\bar{x}$ is not maximum.

(3) $\Rightarrow$ (1) $v = F_{\bar{x}}[S, \bar{S}]$ (by Claim 1).

(3) $\Rightarrow v = u[S, \bar{S}]$. But By claim 2, $v \leq u[S, \bar{S}]$.

Hence v is maximum.

(2) $\Rightarrow$ (3)

(2) says there is no augmenting path in $G_1(\bar{x})$.

Let $S = \{ i \in N \mid i$ is reachable from s in $G_1(\bar{x}) \}$

$\bar{S} = N \setminus S$ (complement of S).

→ there is a directed path from s to i

Hence, there is no arc in $G_1(\bar{x})$ from S to $\bar{S}$.

$\Rightarrow$ For $i \in S$, $j \in \bar{S}$, $x_{ij} = u_{ij}$, and similarly,

$i \in S$, $j \in \bar{S}$, $x_{ji} = 0$.

$$\Rightarrow F_{\bar{x}}[S, \bar{S}] = \sum_{i \in S, j \in \bar{S}} x_{ij} - \sum_{i \in S, j \in \bar{S}} x_{ji}$$

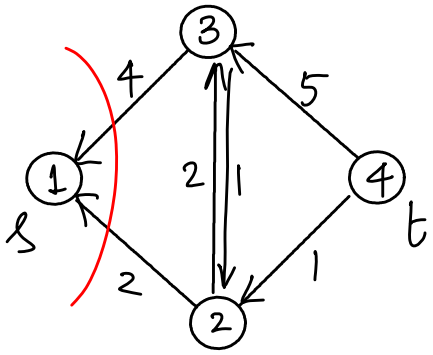
$$= \sum_{i \in S, j \in \bar{S}} u_{ij} - \sum_{i \in S, j \in \bar{S}} 0 = u[S, \bar{S}].$$

□

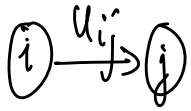
Property To obtain a min-cut from a max flow $\bar{x}$, set $S = \{$all nodes reachable from s in $G_1(\bar{x})\}$.

Example (from Lecture 16)

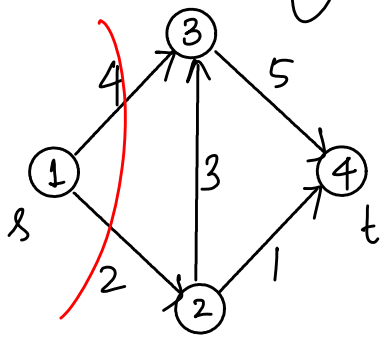
$G(\bar{x})$ with no augmenting paths



Here, we can set $S = \{1\}$, $\bar{S} = \{2,3,4\}$.



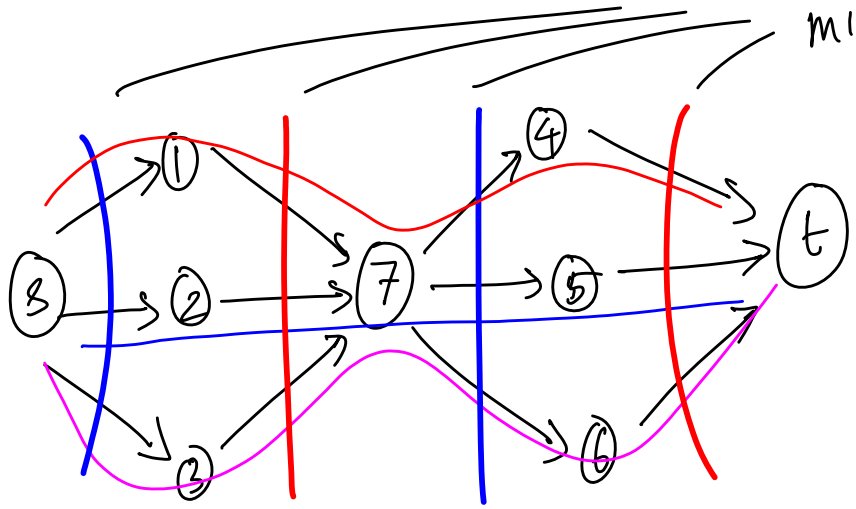
$u[S,\bar{S}] = 4+2 = 6.$



Applications of Max-Flow Min-Cut Theorem

Consider a communication network in which you want to have alternative routes to send traffic from s to t. Similar scenario arises in road transportation networks.

Q: What is the maximum number of arc-disjoint paths from s to t?



min cuts → all have 3 arcs

There are 3 arc-disjoint s-t paths here.

Arc disjoint s-t paths: Two s-t paths are **arc-disjoint** if they do not share any arcs. They could share nodes.

Theorem Let $G = (N, A)$ be a directed graph. The number of arc disjoint paths from s to t is equal to the minimum number of arcs whose removal from G leaves no directed s-t paths.

This is a direct application of the MCMF theorem. If one were to try and prove it using other approaches, it becomes quite hard!

Node-disjoint paths

Two s-t paths are **node-disjoint** if they do not share any nodes except s and t themselves.

In the above example, no two s-t paths are node-disjoint, as node ⑦ is shared by all of them.

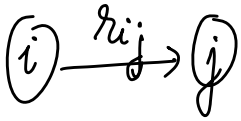
Theorem Let $G=(N,A)$ be a directed network with no (s,t) arc. The maximum number of node-disjoint s-t paths in G is equal to the minimum number of nodes whose removal from G disconnects all s-t directed paths.

In the above example, this number is 1 – remove node ⑦.

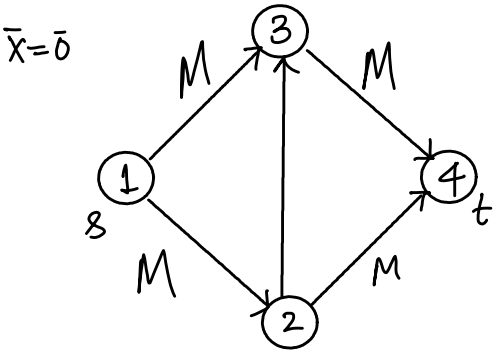
We can use node-splitting, and then apply MCMF to get this result. Removing node ⓘ is equivalent to removing arc (i', i'').

MATH 566 – Lecture 18 (10/23/2014)

Generic augmenting path algo: $\bar{x}=0$, find $G(\bar{x})$, find s-t augmenting paths in $G(\bar{x})$, augment ...

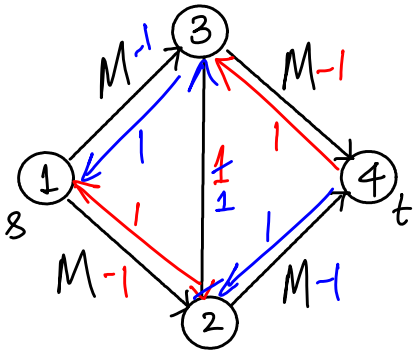


A pathological example:



Choose $P_1 = 1\text{-}2\text{-}3\text{-}4$, $\delta(P)=1$,
then $P_2 = 1\text{-}3\text{-}2\text{-}4$, $\delta(P)=1$, and again P_1, and so on.

We could repeatedly choose P_1 and P_2, sending 1 unit of flow in each augmentation. We end up doing $2M$ augmentations.



Instead, we could augment along $1\text{-}2\text{-}4$ and then $1\text{-}3\text{-}4$ (M units each) and be done in 2 augmentations!

Also, if data (i.e., u_{ij}'s) are irrational, the generic algo might converge to a suboptimal solution.

We need to choose augmenting paths carefully!

1. "Largest" augmenting path algo: select path P with largest $\delta(P)$.
2. Shortest augmenting path algo: select P with smallest # arcs.

We consider option 2 first.

Shortest augmenting path algorithm for max flow

We use distance labels $d : N \to \mathbb{Z}_{\geq 0}$.

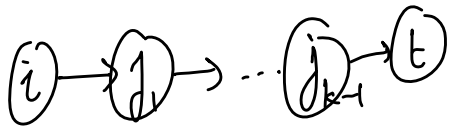
(N: set of nodes)

Def A set of distance labels $d(\cdot)$ is **valid** w.r.t. flow $\bar{x}$ if

(1) $d(t) = 0$.

(2) $d(i) \leq d(j) + 1 \quad \forall (i,j) \in G(\bar{x})$.

Property 7.1 If the distance labels are valid, then $d(i)$ is a **lower bound** on the length (in # arcs) of the SP from i to t.



Proof Add validity conditions over any path of length k from i to t.

We get $d(i) \leq d(t) + k = k$.

($d(t) = 0$)

Valid for all k. Hence $d(i)$ is a lower bound.

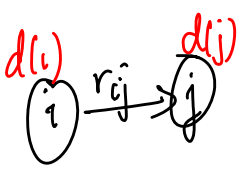
Property 7.2 If $d(s) \geq n$, there is no directed s-t path.

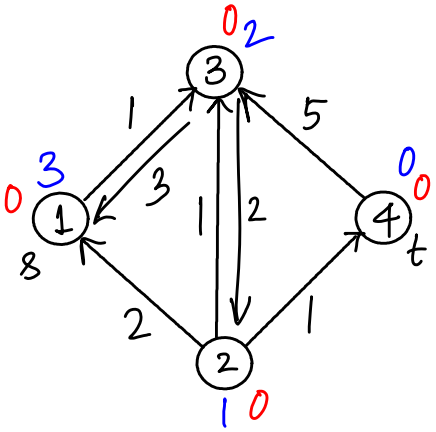
Def An arc $(i,j) \in G(\bar{x})$ is **admissible** if $d(i) = d(j) + 1$.
A path from s to t consisting only of admissible arcs is an **admissible path**.

Property 7.3 An admissible path P from s to t is a shortest augmenting path.

As we will have $d(s) = k$ for the #arcs $= k$ in the path. Since $d(s)$ is a lower bound on all possible lengths k (Property 7.1), equality implies optimality.

Def A set of distance labels $d(\cdot)$ is **exact** if $d(i)$ is the length of the SP from i to t (in # arcs).





$d = [0\ 0\ 0\ 0]$ is valid

$d = [3\ 1\ 2\ 0]$ is valid and exact

We could think about $d(i)$ as the height above ground level that node i has to be raised for flow to happen "freely". t is at ground level ($d(t)=0$), and s need not be raised above $(n-i)$ height levels above the ground.

Shortest augmenting path algorithm

```
algorithm shortest augmenting path;
begin
    x := 0;
    obtain the exact distance labels d(i);
    i := s;
    while d(s) < n do
    begin
        if i has an admissible arc then
            begin
                advance(i);
                if i = t then augment and set i = s
            end
        else retreat(i)
    end;
end;
```

We could use reverse BFS from t to obtain the exact distance labels

Figure 7.5 Shortest augmenting path algorithm.

```
procedure advance(i);
begin
    let (i, j) be an admissible arc in A(i);
    pred(j) := i and i := j;
end;
```

in $G(\bar{x})$

```
procedure retreat(i);
begin
    d(i) := min{d(j) + 1 : (i, j) ∈ A(i) and r_ij > 0};
    if i ≠ s then i := pred(i);
end;
```

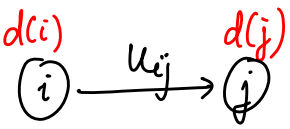
in $G(\bar{x})$

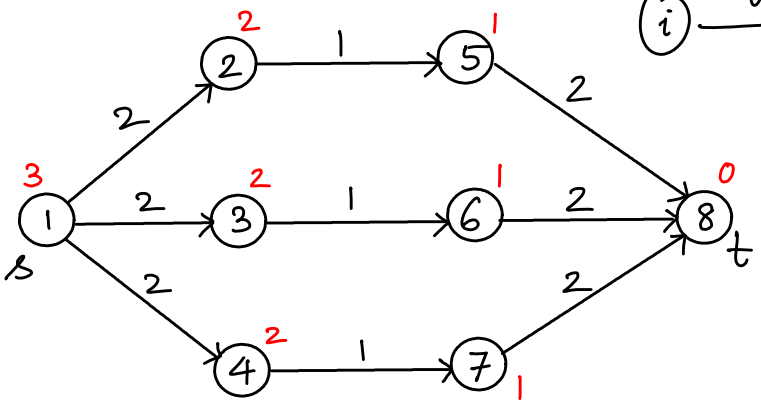
this is the **relabel** operation

This is the one step where we update $d(\cdot)$ during the run of the algorithm.

```
procedure augment;
begin
    using the predecessor indices identify an augmenting
    path P from the source to the sink;
    δ := min{r_ij : (i, j) ∈ P};
    augment δ units of flow along path P;
end;
```

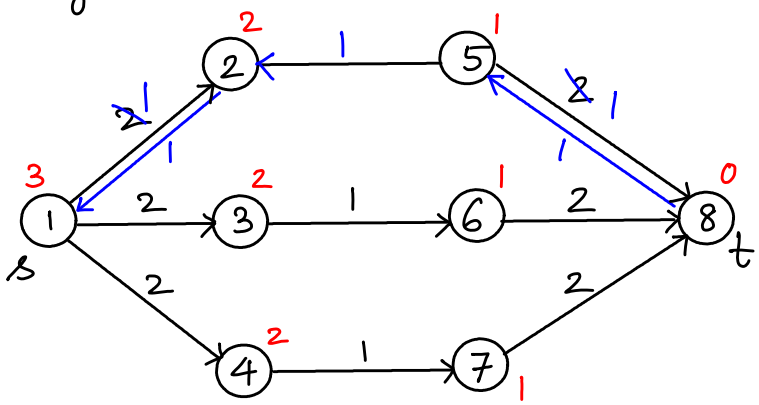
Example





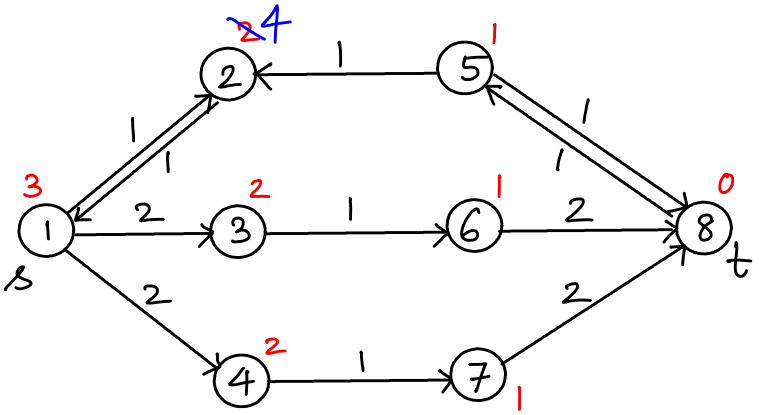
As a convention, we will examine ares in A(i) in the lexicographic order.

Iteration 1 $P_1 = 1\text{-}2\text{-}5\text{-}8$; augment $\delta(P_1) = 1$ units.



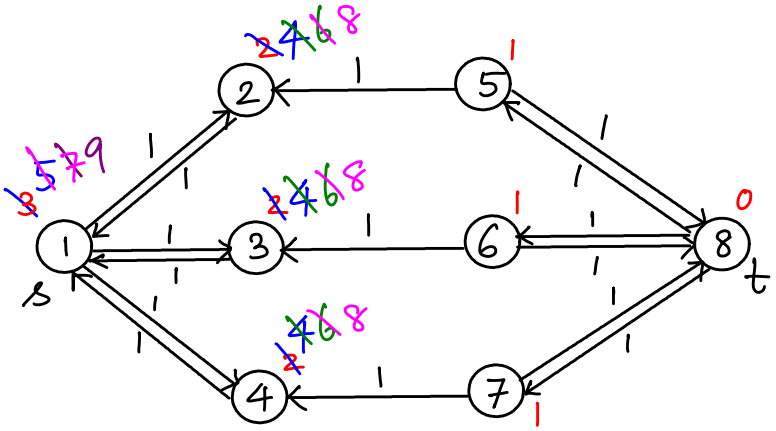
We advance from $s=1$ to 2 to 5 to 8, identifying an admissible path.

Iteration 2:



We advance from $s=1$ to 2. But there are no admissible arcs out of 2. So we relabel node 2 (to $d(1)+1=4$), and retreat back to $s=1$.

In **Iteration 3**, (1,2) is no longer admissible ($d(1)=3 \neq d(2)+1=5$). We choose (1,3), and advance all the way to 8 via 1-3-6-8, and augment $\delta=1$ unit. Subsequently, we augment along $\delta=1$ unit along 1-4-7-8.



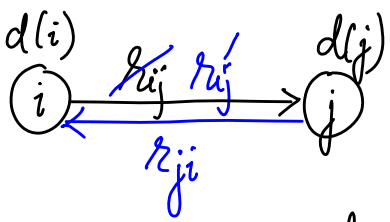
The algorithm repeatedly relabels $s=1$ and 2, 3, 4 using a series of retreat operations until $s=1$ is relabeled from 7 to 9. (notice $n=8$ here). The algorithm stops at this point.

When augmenting, add new arcs to the $A(i)$ lists, and remove arcs that are saturated from $A(i)$ lists.

Correctness

We show that distance labels remain valid after each augmentation and after each relabeling.

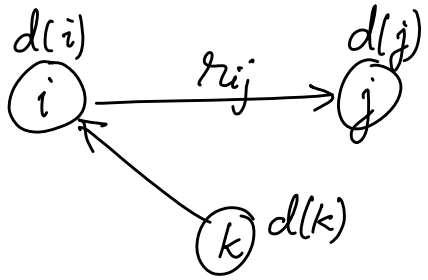
Augmentation Bottleneck arc(s) disappear from $G_r(\bar{x})$, and we can add (or create) (j,i) to $G_r(\bar{x})$ with $r_{ji} > 0$.



$d(i) = d(j) + 1$ (admissible arc)

Consider the general situation where (i,j) is not saturated by the augmentation. Thus, r_{ij} is reduced to r'_{ij} (with $r'_{ij} < r_{ij}$), and $r_{ji} > 0$ is added (i.e., (j,i) is added to $G_r(\bar{x})$ anew).

For (j,i), we need $d(j) \le d(i)+1$ for validity. But this condition holds, as $d(j) = d(i) - 1$ now!



Relabeling

We relabel when there is no admissible arc (i,j).

$$\Rightarrow \quad d(i) \neq d(j) + 1 \quad \forall (i,j) \in G(\bar{x}).$$

$$\Rightarrow \quad d(i) < d(j)+1 \quad \forall (i,j) \in G(\bar{x}),$$ due to validity of distance labels

$$\Rightarrow \quad d(i) < \min_j \{ d(j)+1 \mid (i,j) \in G(\bar{x}) \}.$$

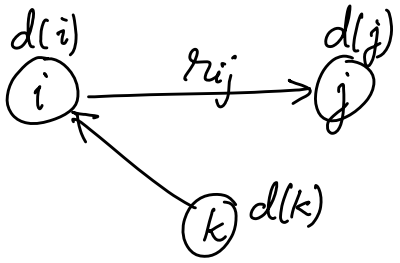
Relabel: $d'(i) = \min_j \{ d(j)+1 \mid (i,j) \in G(\bar{x}) \}.$

The relabeling thus strictly increases $d(i)$.

MATH 566 – Lecture 19 (10/28/2014)

Correctness of shortest augmenting path algorithm for max-flow

Proof continued ...



Relabeling

d(i) strictly increases due to the relabeling operation.

Consider $(k,i) \in G_l(\bar{x})$. $d(k) \le d(i)+1$ holds previously. Since d(i) strictly increases, $d(k) \le d(i)+1$ holds after the relabeling. □

Complexity of Shortest Augmenting Path algorithm

Outline (Corresponding results in AMO mentioned alongside)

1. number of relabels $= O(n^2)$ → Lemma 7.9(a)
   time for relabels $= O(nm)$ → property 7.7

2. number of augmentations $= O(nm)$ → Lemma 7.9(b)
   ↳ uses Lemma 7.8
   time for each augmentation $= O(n)$
   Hence, total time for augmentations $= O(n^2m)$ – Theorem 7.10

3. time spent looking for augmentations

total # retreat operations = $O(n^2)$ (= # relabels)

total # advance operations = $O(n^2 + n^2 m)$

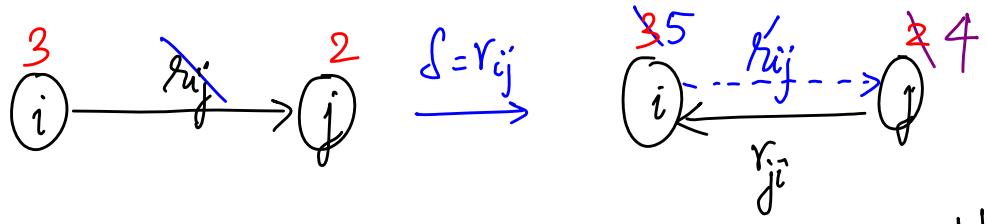
n^2: # retreats; $n^2 m$: $n \times$ (# augmentations)

Details

1. $d(i) \leq n-1 \quad \forall i \in N$. If $d(i)$ goes above $n-1$ for some intermediate node, we can ignore that node from further computation.

Each relabel of node i strictly increases $d(i)$. So we perform at most n relabels of node i. Hence we do $O(n^2)$ relabels overall.

To relabel node i, we examine each arc in $A(i)$ once. If arc (i,j) is inadmissible, it remains so until $d(i)$ is increased by relabeling. Hence, the total time for all relabels $= O\left(n \sum_i |A(i)|\right) = O(nm)$.

2. 

Let the algorithm saturate (i,j) in the first push. We could saturate (i,j) again in a subsequent augmentation. But for this step to happen, both $d(i)$ and $d(j)$ must have increased (by 2 each). Since $d(i)$ is at most n ($n-1$ to be exact), we can saturate (i,j) at most $\frac{n}{2}$ times, i.e., $O(n)$ times.

Each augmentation saturates at least one arc. Hence the total # augmentations $= O(mn)$.

$O(mn)$: # arcs × # saturations

Time for each augmentation $= O(n)$ → there are at most $n-1$ arcs in each augmenting path

Hence, time for all augmentations $= O(n \cdot mn) = O(n^2 m)$.

4. Each retreat operation removes one arc from a partially admissible path, while each advance operation adds an arc to such a partial s-t path.

total # retreats $= O(n^2)$ = # relabels.

total # arcs in all augmentations $= O(n^2m)$

$O(mn)$ augmentations $\times$
$O(n)$ arcs per augmentation

Hence the total # advances $= O(n^2 + n^2m)$

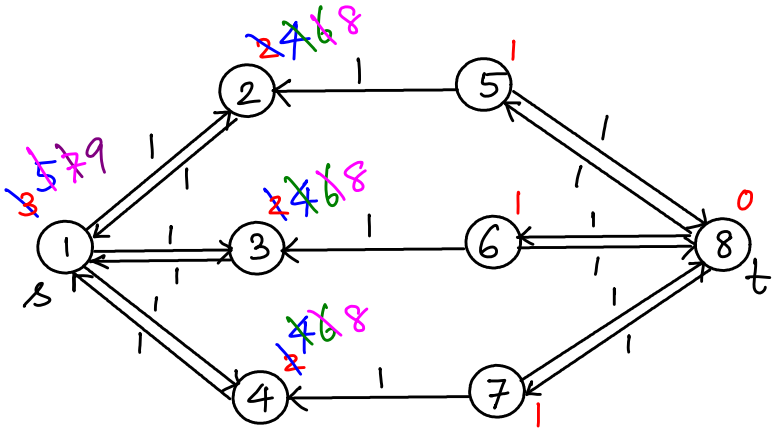
retreats # arcs in augmentations

Notice that we have to make advances first before we could retreat. In the case of augmentations, we can count the advances alone, as there are no retreats in this process.

Hence, the overall running time is $O(n^2m)$.

We could do practical improvements to the running time of this algorithm.

Consider the instance from the last lecture. Ideally, the algorithm could stop after the three augmentations as the flow is already maximum. But it performs a number of relabels before terminating.



Could identify a mincut $[S, \bar{S}]$ with $S = \{1,2,3,4\}$ as soon as $d(4)$ gets relabeled to 4 from 2.

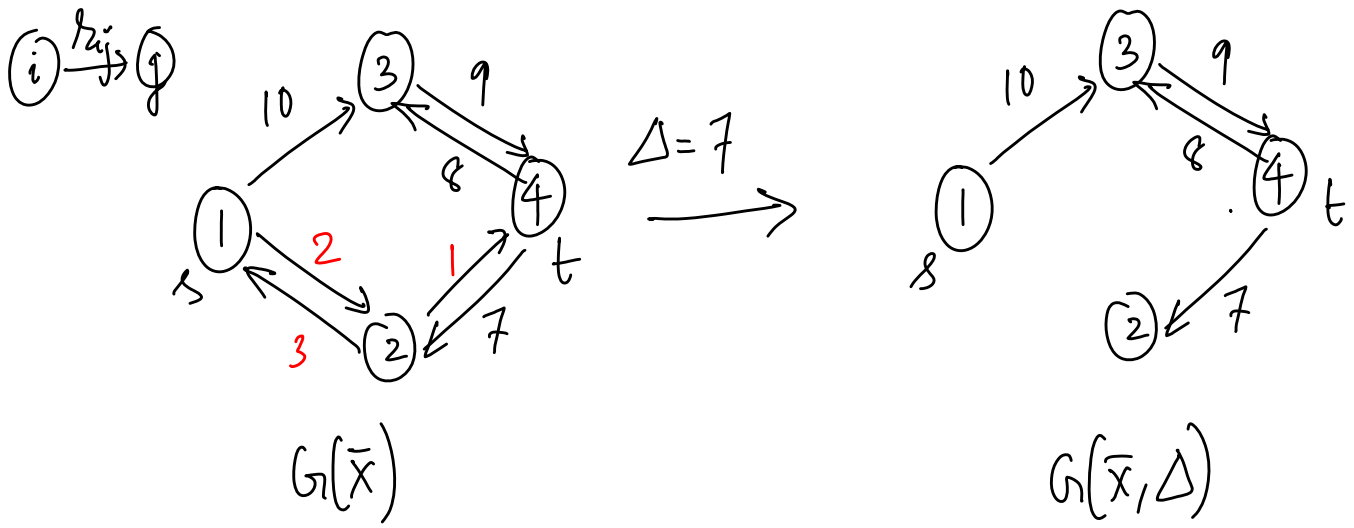
The Capacity-scaling Algorithm

Motivated by the "largest" augmenting path algorithm idea, we augment along a path with "sufficiently large" capacity in each step.

Def For any Δ (fixed), we define

$$G(\bar{x}, \Delta) = \{(i,j) \in G(\bar{x}) \mid r_{ij} \geq \Delta\}.$$

A flow $\bar{x}$ is **Δ-maximum** if there is no augmenting path in $G(\bar{x}, \Delta)$, i.e., there is no augmenting path with capacity Δ or more.



$G(\bar{x})$ $\qquad$ $G(\bar{x}, \Delta)$

Notice that for $\Delta = 1$, $G(\bar{x}, \Delta) = G(\bar{x})$ itself.

Push flow along large Δ-capacity augmenting paths, then <u>decrease Δ</u> until we get to $\Delta = 1$.

↳ we scale Δ by $\frac{1}{2}$

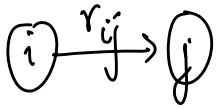
The capacity scaling algorithm

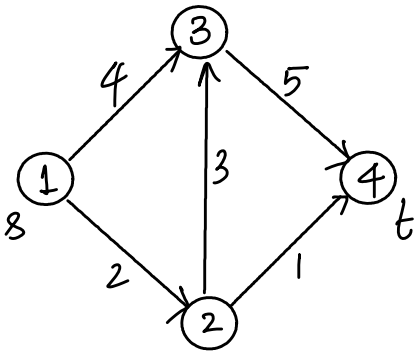
```
algorithm capacity scaling;
begin
    x := 0;
    Δ := 2^⌊log U⌋;
    while Δ ≥ 1 do
    begin
        while G(x, Δ) contains a path from node s to node t do
        begin
            identify a path P in G(x, Δ);
            δ := min{r_ij : (i, j) ∈ P};
            augment δ units of flow along P and update G(x, Δ);
        end;
        Δ := Δ/2;
    end;
end;
```

} Δ-phase

Figure 7.3 Capacity scaling algorithm.

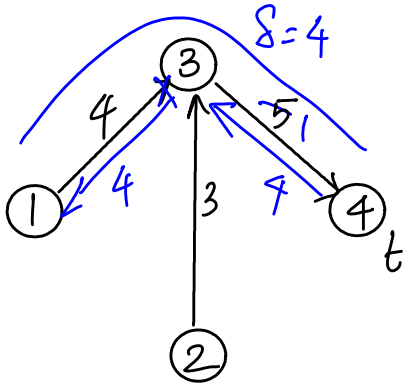
Example





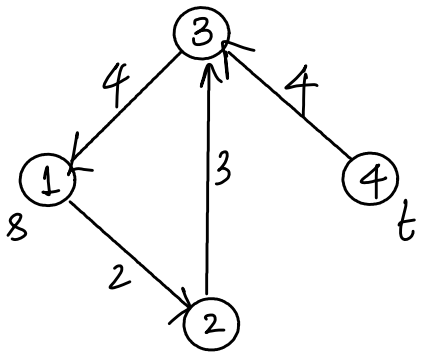
$\Delta = 5$ has only $(3,4)$ in $G(\bar{x}, \Delta)$;

$\Delta = 5/2$ gives $G(\bar{x}, \Delta)$: $\delta = 4$ units are pushed.

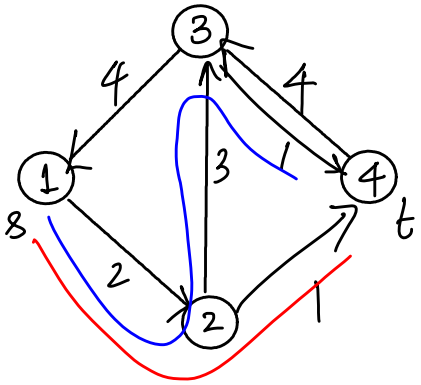


$\Delta = (5/2)/2 = \frac{5}{4}$ gives

There are no augmenting paths.



$\Delta = 1$ (we could go with $\Delta = \frac{5}{8}$, but its equivalent to $\Delta = 1$) gives



$\delta = 1$

$\delta = 1$

Two more augmentations terminates the run.

Initially, Δ is at most U, and Δ is halved in each phase; we stop when $\Delta = 1$. So we do $O(\log U)$ scaling phases.

Overall, the complexity is $O(m^2 \log U)$.

↙ work in each scaling phase.

MATH 566 – Lecture 20 (10/30/2014)

On the shortest augmenting path algo implementation

keep track of the augmenting path in terms of arc indices (apart from the usual pred indices).

Use predarc(i) for $i \in N$.

If you advance path P from node i using arc k to node j,

set predarc(j) = k.

P = [P, k];

Then i = Head(k); % advancing from current i to j = head(k)

If i == t

X(P) = X(P) + Delta;

⋮

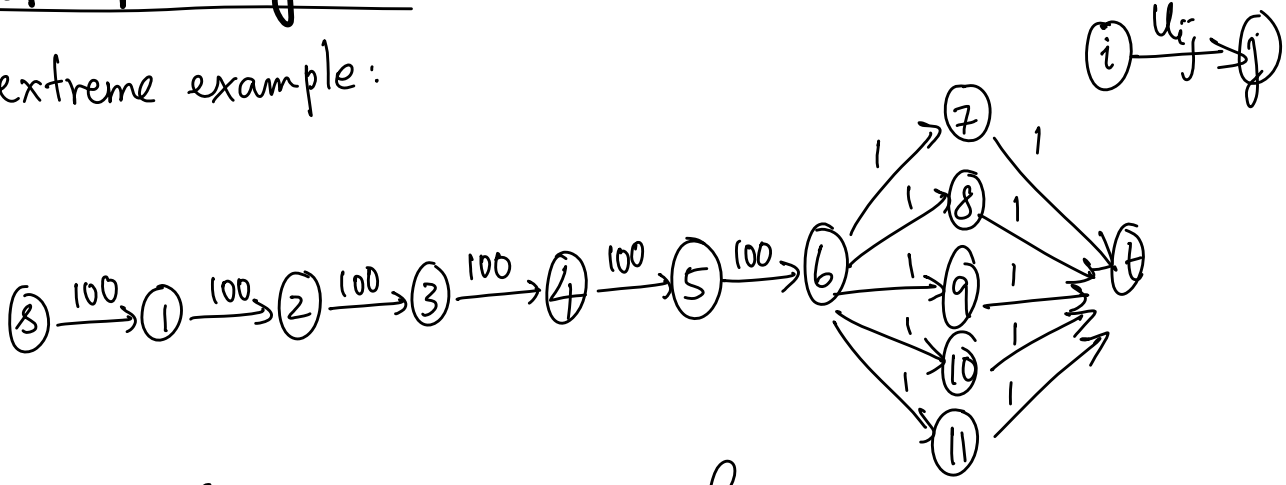
X is an m-vector of flows

start with X = 0.

To handle r_{ij}'s, one option is to work directly with x_{ij}'s. From current node i, examine A(i), the out arc list, for arcs (i,j) with $x_{ij} < u_{ij}$. Also examine AI(i), the in-arc list, for arcs (j,i) with $x_{ji} > 0$.

Preflow Push Algorithms

An extreme example:



The max flow value is $v=5$ here,
but the SAP algo will push one unit each along the partial path $s-6$ several times before pushing to t.
If we could instead push 5 units all in one from s to node 6, the run would be more efficient.

IDEA: At intermediate stages, we permit more inflow than outflow at nodes $i \in N \setminus \{s,t\}$.

So, flow balance constraints are possibly violated in between, but are satisfied at the end.

Def A **preflow** is $\bar{x} : A \to \mathbb{R}^m_{\geq 0}$ such that

$0 \leq x_{ij} \leq u_{ij} \quad \forall (i,j) \in A$ and

$$e(i) = \sum_{(j,i) \in A} x_{ji} - \sum_{(i,j) \in A} x_{ij} \geq 0 \quad \forall i \in N \setminus \{s,t\}.$$

$e(i)$: excess; $\sum_{(j,i) \in A} x_{ji}$: inflow; $\sum_{(i,j) \in A} x_{ij}$: outflow

We typically want $e(s) < 0$ and $e(t) > 0$, but we do not consider $e(s)$ and $e(t)$ in our algorithm explicitly.

Def A node $i \in N \setminus \{s,t\}$ with $e(i) > 0$ is an **active node**.

s, t are never active.

IDEA Push flow from active nodes toward the sink t, relying on $d(\cdot)$, and try to achieve flow balance at all nodes except s, t. If there is excess flow left in the middle nodes that cannot be pushed to t, push them back to s.

The generic preflow-push algorithm

```
algorithm preflow-push;
begin
    preprocess;
    while the network contains an active node do
    begin
        select an active node i;
        push/relabel(i);
    end;
end;
```

Figure 7.12 Generic preflow-push algorithm.

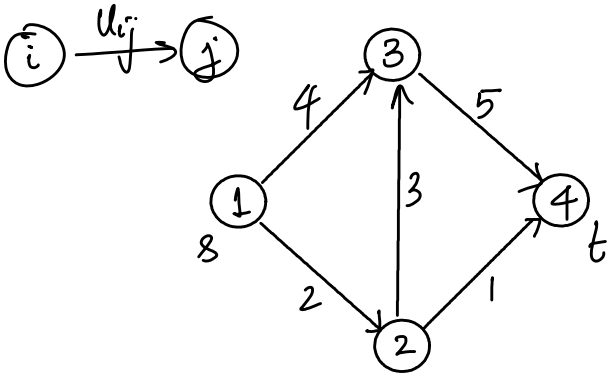
```
procedure preprocess;
begin
    x := 0;
    compute the exact distance labels d(i);
    x_sj := u_sj for each arc (s, j) ∈ A(s);
    d(s) := n;
end;
```

→ j are all active nodes now

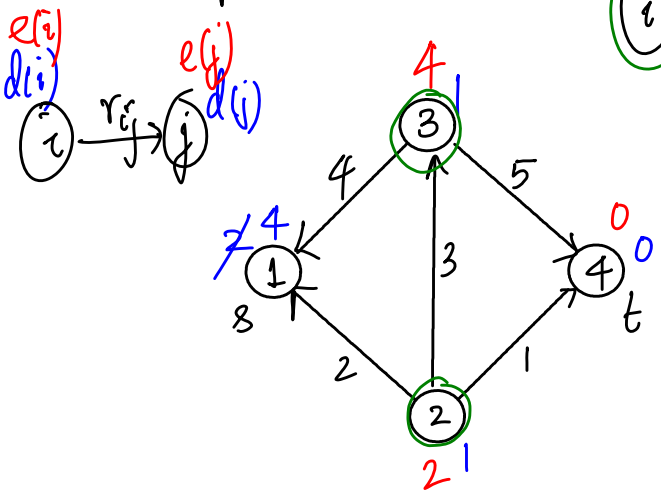
```
procedure push/relabel(i);
begin
    if the network contains an admissible arc (i, j) then
        push δ := min{e(i), r_ij} units of flow from node i to node j
    else replace d(i) by min{d(j) + 1 : (i, j) ∈ A(i) and r_ij > 0};
end;
```

Water pipe network analogy: We first raise s to level n above the ground, and push as much flow "downstream" along all arcs going out of s. We keep pushing flow along downhill pipes from nodes that have excess flow. If there is excess, but no downhill pipe, then we raise the node (relabel operation). There may be downhill pipes now directed toward the source. We stop when all excess has been pushed to either the sink or back to the source.

Example

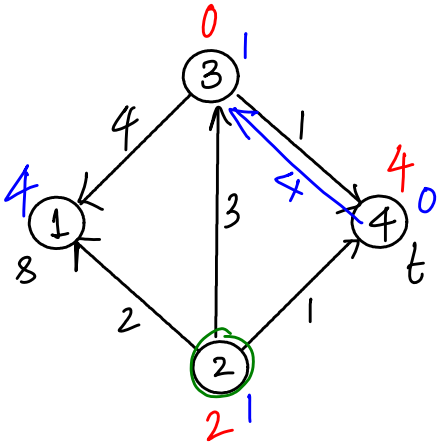


Preprocessing

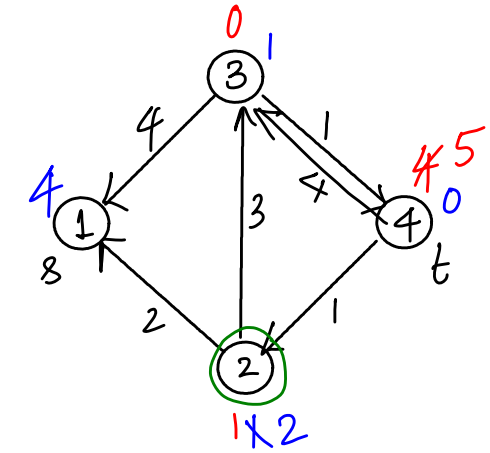


2, 3 are active now

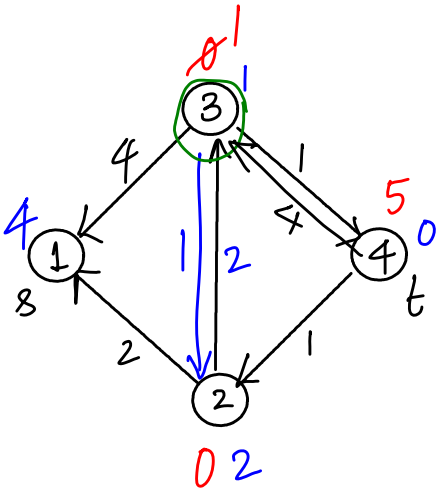
pick ③, (3,4) is admissible, so push $\delta = \min\{e(3), r_{34}\} = 4$.



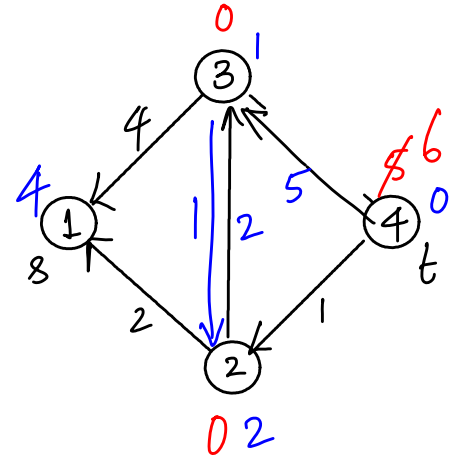
3 is not active. 2 is still active; (2,4) is admissible.
Push $\delta = \min\{e(2), r_{24}\} = 1$.



Node 2 is still active, but there are no admissible arcs, so we relabel. $d(2) = \min\{d(3), d(1)\} + 1 = 2$, after which $(2,3)$ is admissible. We push $\delta = \min\{e(2), r_{23}\} = 1$.



Node 3 is active now, with $e(3) = 1$.
Node 2 is inactive.
From 3, $(3,4)$ is admissible, so push $\delta = \min\{e(3), r_{34}\} = 1$.



No more active nodes, so flow is maximum, with $v = 6$.

The generic preflow push algorithm runs in $O(n^2m)$ time
- same complexity as the shortest augmenting path algorithm.

The preflow-push algorithm was proposed by Goldberg and Tarjan.

Implementation - FIFO preflow-push algorithm

* Whenever you select an active node i, keep pushing flow until $e(i)$ becomes zero (i.e., i is inactive), or node i is relabeled.
  - This whole process is termed **node examination** of i.

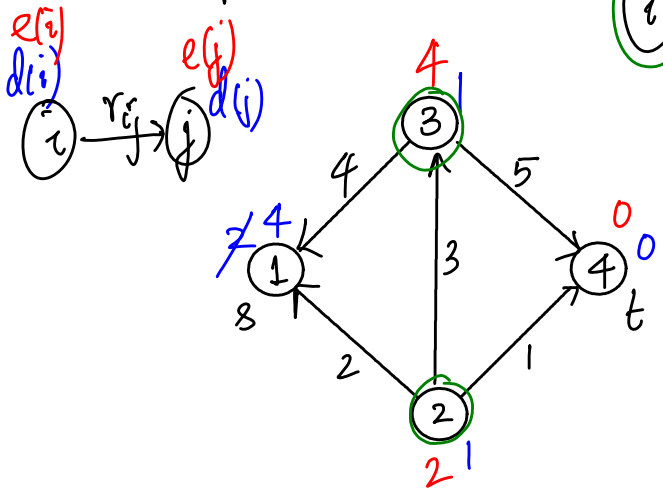
* Examine nodes in a FIFO order, say, maintain a LIST.

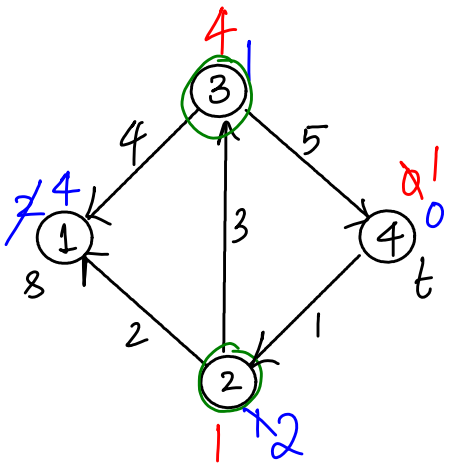
* Add newly active nodes as well as relabeled nodes to the back of LIST.

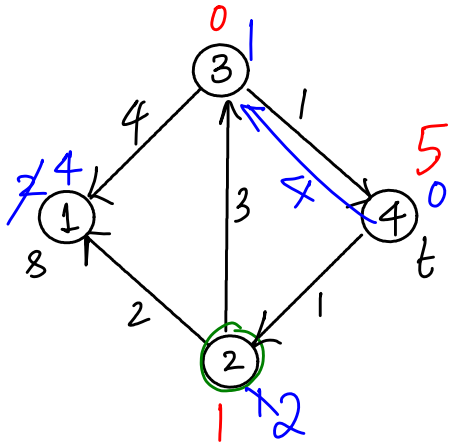
* pick the active node from the front of LIST, and do node examination.

Example revisited — now using FIFO implementation

Preprocessing







LIST = [~~2~~ 3] list of active nodes

pick 2, push $\delta = 1$. Then relabel 2.

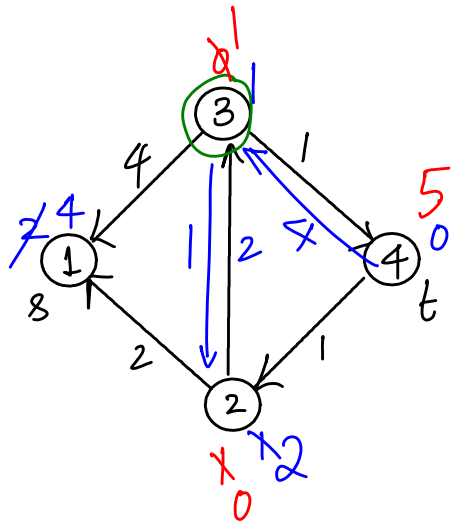
LIST = [~~3~~ 2]

pick 3, push $\delta = 4$.

LIST = [~~2~~]

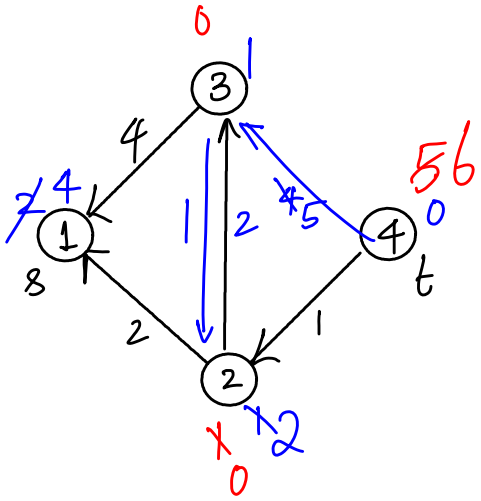
pick 2, push $\delta = 1$ along (2, 3)

3 is active, so LIST = [3].



LIST = [3]

pick 3, push $\delta = 1$ along $(3, 4)$



No more active nodes, so flow is maximum. Notice also that since there are no active nodes, we do not have a preflow anymore — we have a flow!

MATH 566 – Lecture 21 (11/04/2014)

The excess scaling algorithm

Work with active nodes, i.e., using preflow-push ideas, and push large quantities, but not too large!

Let $e_{max} = \max_{i \in N} \{e(i)\}$. We systematically reduce e_{max} to zero (after starting with $2\bar{u}$).

Large excess nodes are nodes j with $e(j) \geq \frac{\Delta}{2}$.

```
algorithm excess scaling;
begin
    preprocess;                                  → same as in preflow-push algo
    Δ := 2^⌈log U⌉;                              → need strictly larger than ū at start
    while Δ ≥ 1 do
    begin (Δ-scaling phase)
        while the network contains a node i with a large excess do
            begin
                among all nodes with a large excess, select a node i with
                the smallest distance label;
                perform push/relabel(i) while ensuring that no node excess exceeds Δ;
            end;
        Δ := Δ/2;
    end;
end;
```

→ $\delta = \min \{e(i), r_{ij}, \Delta - e(j)\}$

Figure 7.18 Excess scaling algorithm.

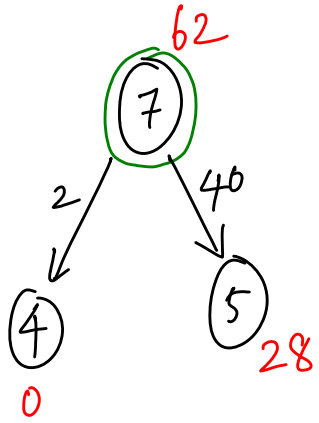
We want to push from a large excess node with the smallest $d(i)$ so as to get as much flow toward t as quickly as possible.

Push large quantity of flow, but not too large:

$$\delta = \min\{e(i), r_{ij}, \Delta - e(j)\}.$$

ensures that $e(j) \le \Delta$ after the push

Illustration of Δ-scaling phase



$\Delta = 64$

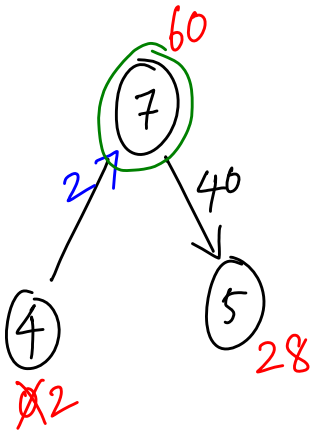
(64-scaling phase)

7 is a large excess node ($e(7) \ge \frac{\Delta}{2} = 32$)

$\Delta - e(j)$

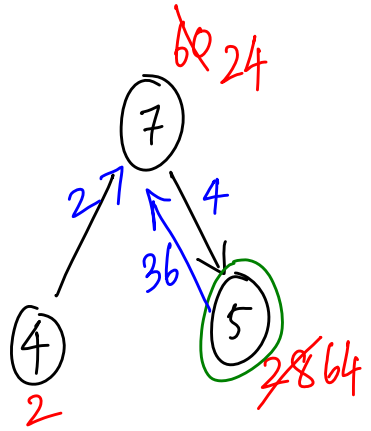
Push $\delta = \min\{62, 2, 64\}$

$= 2$ units along $(7,4)$.



Then push $\delta = \min\{60, 40, 36\}$ ($64-28$, Δ, $e(5)$)

$= 36$ units along $(7,5)$



Now, 5 becomes a large excess node (in the 64-scaling phase).

In another variation, we choose a large excess node with the largest (instead of smallest) distance label. Intuitively, the idea is that we have to push flow out of excess nodes any way, and have to deal with nodes at all height levels, i.e., distance labels, eventually. So we might as well start farther away from t.

The book has a lot more variants described of these algorithms, along with details of implementation.

Minimum Cost Flow (MCF) Problem

Recall: $G = (N, A)$

costs: c_{ij} $\forall (i,j) \in A$ $\quad C = \max_{(i,j)\in A} \{|c_{ij}|\}$

capacities: u_{ij} $\forall (i,j) \in A$ $\quad U = \max_{(i,j)\in A} \{u_{ij}\}$

supply/demand: $b(i)$ $\forall i \in N$

Recall the linear optimization model for MCF:

$$\min \quad z = \sum_{(i,j)\in A} c_{ij} x_{ij} \quad \rightarrow \text{total cost}$$

$$\text{s.t.} \quad \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b(i) \quad \forall i \in N \quad \leftarrow \text{flow balance constraints}$$

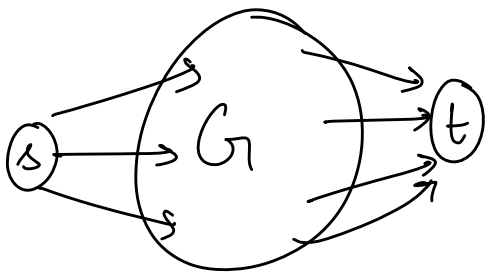
$$0 \le x_{ij} \le u_{ij} \quad \forall (i,j) \in A \quad \leftarrow \text{bound constraints}$$

We will not discuss applications of MCF here. The AMO book lists several of them in section 9.2.

Assumptions

1. $l_{ij} = 0 \quad \forall (i,j) \in A$.
2. All data is integral (c_{ij}, u_{ij}, $b(i)$).
3. The network is directed.
4. $\sum_{i \in N} b(i) = 0$ (i.e., total supply = total demand).
   Else, we will not have a feasible solution.
5. The MCF problem has a feasible solution.

Can use an s-t max flow model to find one.



Add s,t, arcs (s,i) for $i \in N$ with $b(i) > 0$, with $u_{si} = b(i)$, and arcs (j,t) for $j \in N$ with $b(j) < 0$, with $u_{jt} = -b(j)$.

If an s-t max flow saturates all these extra arcs, it gives a feasible solution.

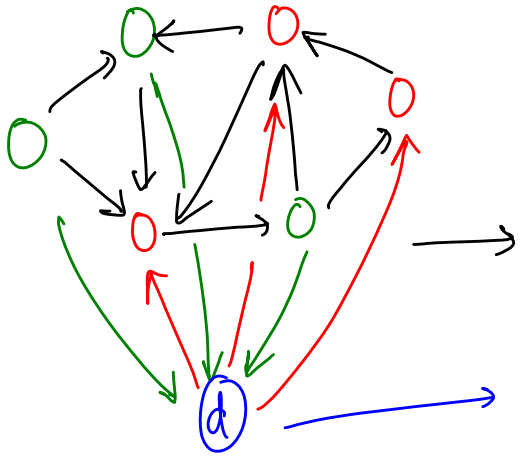
6. There is an uncapacitated directed path between every pair of nodes i, j each with large costs.

large capacity

If there is no arc $(i,j) \in A$, add it with $u_{ij} = \infty$ and $c_{ij} = +\infty$.

Another approach for ensuring feasible solutions – use artificial arcs

G



○ → supply node $(b(i) > 0)$

○ → demand node $(b(i) < 0)$

→ artificial arcs.

→ dummy node

Add arcs (i,d) $\forall i \in N$ with $b(i) > 0$, and (d,j) $\forall j \in N$ with $b(j) < 0$, with $c_{id} = c_{dj} = M$ $(+\infty)$ as well as $u_{id} = u_{dj} = M$ $(+\infty)$.

If the optimal solution uses any of the artificial arcs, then the original MCF problem is infeasible. Else, i.e., if $x_{id} = x_{dj} = 0 \; \forall i, j \in N$, in the optimal solution, we get an MCF.

7. $c_{ij} \geq 0$.

Else, we could use arc reversal. But this transformation assumes $u_{ij} < \infty$. If indeed some $u_{ij} = +\infty$, we can instead use $u_{ij} = B = \sum u_{kl} + \sum_{b(i)>0} b(i)$.

B acts like infinity; $\sum u_{kl}$: all arcs with finite u_{kl}

8. We assume there is no negative cost cycle of infinite capacity.

Else, the problem is unbounded.

We could replace $u_{ij} = +\infty$ with B as above, and then watch for the total cost exceeding some bound.

Notice that we get Assumption 8 when Assumption 7 is satisfied. At the same time, it is helpful to list them both separately.

Residual network

$G(\bar{x})$: For (i,j), we set

$$r_{ij} = u_{ij} - x_{ij}$$ and has cost c_{ij}

For (j,i) $r_{ij} = x_{ij}$ and has cost $-c_{ij}$.

Recall that in max flow, we did not have to work with costs (or, all $c_{ij} = 1$). Hence we could combine the residual capacities arising from different arcs. In particular, we had $r_{ij} = (u_{ij} - x_{ij}) + x_{ji}$. But in MCF, the costs are different, and hence we cannot combine.

Similarly, we could have multiple arcs between i and j with different costs.

MATH 566 – Lecture 22 (11/06/2014)

Residual network for MCF

$r_{ij} = u_{ij} - x_{ij}$ with cost c_{ij}

$r_{ji} = x_{ij}$ with cost $-c_{ij}$

Reduced costs

Recall: $d(j) \leq d(i) + c_{ij}$ $\forall (i,j)$ were the optimality conditions for the shortest path problem.

Equivalently, we can write $c_{ij}^d = c_{ij} + d(i) - d(j) \geq 0 \; \forall (i,j)$

For a set of node potentials $\pi(i)$, $i \in N$, $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ is the **reduced cost w.r.t.** $\bar{\pi}$. → vector of all node potentials

In the SP case, we take $\pi(i) = -d(i)$.

We describe optimality conditions for MCF using $G_1(\bar{x})$ and some node potentials $\bar{\pi}$.

Can design algorithms to test for violations of the optimality conditions.

1. Negative cycle optimality conditions

Theorem 9.1 A feasible flow $\bar{x}$ is an optimal solution of the MCF problem iff there is no negative cost cycle in $G(\bar{x})$.

Proof ($\Rightarrow$) If there is a negative cycle in $G(\bar{x})$, we can augment flow along it to reduce total cost. Hence $\bar{x}$ is not optimal.

($\Leftarrow$) Let $\bar{x}$ be a feasible flow and $G(\bar{x})$ has no negative cycles. We need to argue that $\bar{x}$ is optimal. Assume $\bar{x}^*$ is an optimal flow, and $\bar{x}^* \neq \bar{x}$. Then we can decompose $\bar{x}^* - \bar{x}$ into at most m cycles. The total cost of all these cycles is $\bar{c}^T(\bar{x}^* - \bar{x})$.

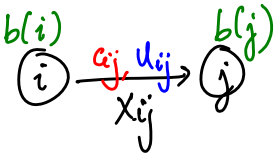
↳ vector of costs

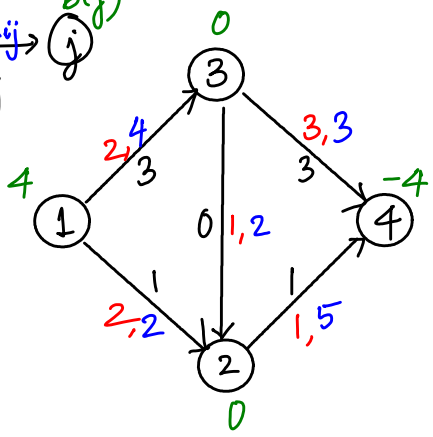
But since $G(\bar{x})$ has no negative cycles, $\bar{c}^T(\bar{x}^* - \bar{x}) \geq 0$, which gives $\bar{c}^T\bar{x}^* \geq \bar{c}^T\bar{x}$. Also, $\bar{x}^*$ is optimal, hence $\bar{c}^T\bar{x}^* \leq \bar{c}^T\bar{x}$, $\Rightarrow \bar{c}^T\bar{x}^* = \bar{c}^T\bar{x}$, i.e., $\bar{x}$ is also an optimal flow.

Negative cycle canceling Algorithm

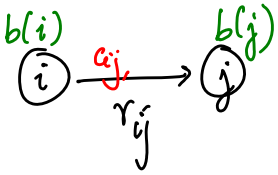
* Start with a feasible flow $\bar{x}$.
* Use the FIFO label-correcting algorithm to detect negative cycles in $G(\bar{x})$.
* Augment along negative cycle, update $\bar{x}$, $G(\bar{x})$.

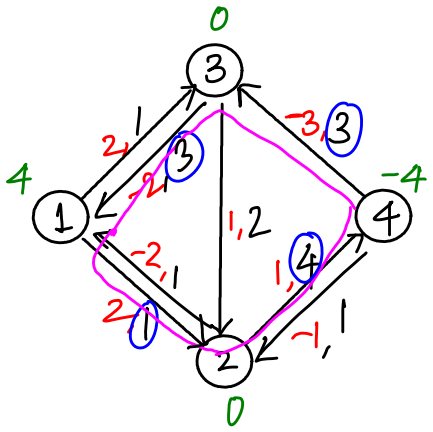
Example





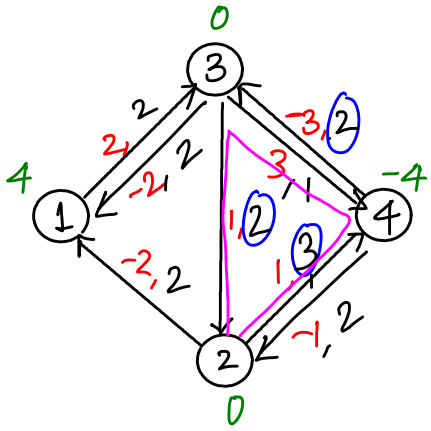
$G(\bar{x})$





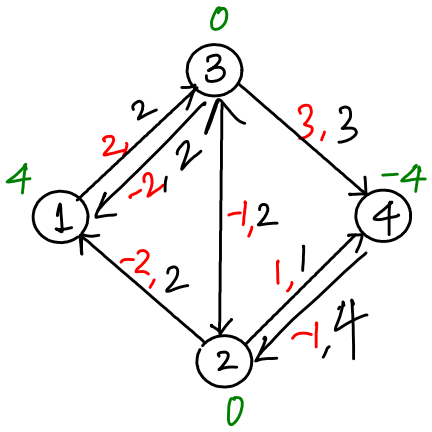
$W_1 = 1\text{-}2\text{-}4\text{-}3\text{-}1$, $\delta(W_1) = 1$

$c(W_1) = -2 \longrightarrow$ total cost of W_1



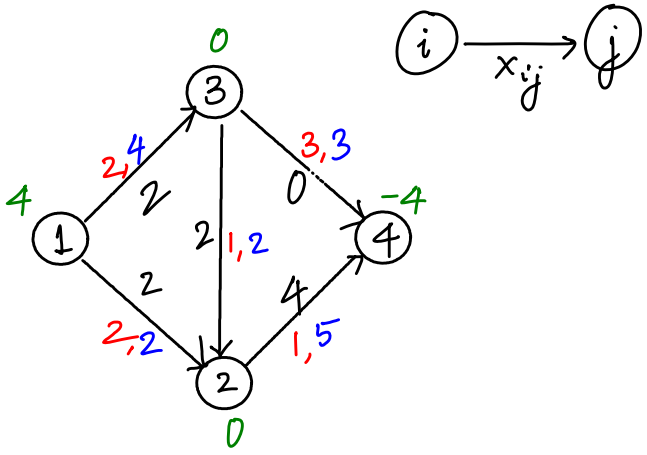
$W_2 = 2\text{-}4\text{-}3\text{-}2$, $\delta(W_2) = 2$

$c(W_2) = -1$



No more negative cycles, hence flow is optimal.

optimal flow:



Theorem 9.10 If u_{ij} and $b(i)$ are integral, the negative cycle canceling algorithm maintains an integral solution, i.e., flow $\bar{x}$, after each iteration.

Proof The initial feasible flow is integral, as it could be found by solving a max-flow problem. At each iteration, δ is integral as well.

Finiteness

Theorem The negative cycle canceling algorithm terminates after a finite number of iterations, assuming all data is finite and integral.

Proof

We can use $mC\bar{U}$ and $-mC\bar{U}$ as the upper and lower limits on the total cost $\bar{c}^T\bar{x}$ $\left(\sum_{(i,j)} c_{ij}x_{ij}\right)$.

So the total cost can change by at most $2mC\bar{U}$.

In each iteration, the total cost decreases by at least 1 unit. Hence the # iterations $= O(mC\bar{U})$.

FIFO label-correcting algorithm in each iteration (to find negative cycle) takes $O(mn)$ time.

Hence, the generic negative cycle canceling algorithm runs in $O(m^2nC\bar{U})$ time.

2. Reduced Cost Optimality Conditions

Recall: $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ is the reduced cost of arc (i,j) w.r.t node potentials $\bar{\pi}$.

With $\pi(i) = -d(i)$, we had seen $c_{ij}^{d} \geq 0$ are the shortest path optimality conditions.

Property 9.2

(a) For a directed path P from node k to node l,

$$\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l).$$

(b) For a directed cycle W, $\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij}$

Implications

1. Potentials π do not change the shortest path between k and l.
2. A directed negative cycle W w.r.t. c_{ij} is also a directed negative cycle w.r.t. c_{ij}^{π}

(Reduced cost optimality conditions) A feasible solution $\bar{x}$ is an optimal solution to the MCF problem iff some node potentials π satisfy

$$c_{ij}^{\pi} \geq 0 \quad \forall (i,j) \in G(\bar{x}).$$

We can argue that the negative cycle optimality conditions are equivalent to the reduced cost optimality conditions.

Proof ($\Rightarrow$) Let $c_{ij}^{\pi} \geq 0 \ \forall (i,j) \in G(\bar{x})$. Hence for every cycle W in $G(\bar{x})$, we have $\sum_{(i,j)\in W} c_{ij}^{\pi} \geq 0$, which by property 9.2(b) gives $\sum_{(i,j)\in W} c_{ij} \geq 0$.

($\Leftarrow$) $G(\bar{x})$ has no negative cycle. We can use $\pi(i) = -d(i)$, where $d(i)$ is the shortest path distance from node 1 to node i. We get

↳ can be any node

$c_{ij}^{\pi} \geq 0$ as the SP optimality conditions for $G(\bar{x})$.

Notice that by specifying the reduced cost optimality conditions on $G(\bar{x})$, we take into account the flow $\bar{x}$ (and with it the bounds u_{ij}) as well as the costs c_{ij}.

Interpretation $c_{ij}^{\pi} \geq 0$ with $\pi(i) = -d(i)$, i.e., $d(j) \leq d(i) + c_{ij}$.

Let $d(i) = -\pi(i)$ be the cost for buying 1 unit of the commodity at node i, and c_{ij} is the cost of shipping one unit from i to j.

The optimality conditions specify that the cost of buying one unit at node j cannot be more than that of buying it at node i and shipping it to j.

3. Complementary Slackness Optimality Conditions

$\bar{x}$ is optimal w.r.t. $\bar{\pi}$ iff

1. If $c_{ij}^{\pi} > 0$ then $x_{ij} = 0$
2. If $c_{ij}^{\pi} < 0$ then $x_{ij} = u_{ij}$
3. If $0 < x_{ij} < u_{ij}$, then $c_{ij}^{\pi} = 0$

MATH 566 – Lecture 23 (11/13/2014)

Review of Min-cost flow (MCF) problem

For flow $\bar{x}$, the residual network $G_r(\bar{x})$ has

$r_{ij} = u_{ij} - x_{ij}$ with cost c_{ij} and

$r_{ji} = x_{ij}$ with cost $-c_{ij}$.

For node potentials $\bar{\pi}$ ($\pi(i)$ is the potential of node i), the reduced costs are $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$.

Optimality conditions

1. Negative cycle optimality conditions:
   $\bar{x}$ is optimal for MCF iff there are no negative cost cycles in $G_r(\bar{x})$.
2. Reduced cost optimality conditions:
   $\bar{x}$ is optimal for MCF iff $c_{ij}^{\pi} \geq 0 \; \forall (i,j) \in G_r(\bar{x})$ for some node potentials $\bar{\pi}$.

Notice that both the above optimality conditions are specified on $G_r(\bar{x})$.

3. Complementary Slackness Optimality conditions

(CSC for complementary slackness conditions)

CSC are specified on the original network, as opposed to on $G(\bar{x})$.

CSC $\bar{x}$ is a feasible flow, $\bar{\pi}$: node potentials.

$(\bar{x}, \bar{\pi})$ is optimal for MCF iff the following conditions hold:

1. If $c_{ij}^{\pi} > 0$ then $x_{ij} = 0$;
2. If $c_{ij}^{\pi} < 0$ then $x_{ij} = u_{ij}$; and
3. If $0 < x_{ij} < u_{ij}$, then $c_{ij}^{\pi} = 0$.

Proof: We show that the reduced cost optimality conditions are equivalent to the CSC.

1. $c_{ij}^{\pi} > 0 \Rightarrow r_{ji} = 0$ (or $(j,i) \notin G(\bar{x})$).

as $c_{ji}^{\pi} = -c_{ij}^{\pi} < 0$, which would violate the reduced cost optimality conditions.

Hence $x_{ij} = 0$.

2. $c_{ij}^{\pi} < 0 \Rightarrow r_{ij} = 0 \Rightarrow x_{ij} = u_{ij}$.

due to reduced cost optimality conditions

3. $0 < x_{ij} < u_{ij} \Rightarrow r_{ij} > 0$ and $r_{ji} > 0$.

By reduced cost optimality conditions, we need $c_{ij}^{\pi} \geq 0$ and $c_{ji}^{\pi} = -c_{ij}^{\pi} \geq 0$, i.e., $c_{ij}^{\pi} = 0$.

For 3., the reverse implication need not hold, i.e., $c_{ij}^{\pi} = 0 \not\Rightarrow 0 < x_{ij} < u_{ij}$. If $c_{ij}^{\pi} = 0$, x_{ij} does not contribute to the total cost. So we could have $x_{ij} = 0$, but notice that this condition is not implied.

Successive shortest path algorithm for MCF

We will use a combination of the concepts employed in the preflow-push and shortest augmenting path algos for max-flow.

The idea is to maintain optimality in terms of the reduced cost optimality conditions, and strive for feasibility.

We maintain bounds $(0, u_{ij})$, but violate flow balance in the intermediate steps.

Def A flow $\bar{x}$ that satisfies the flow bounds $(0, u_{ij})$, but not necessarily the flow balance constraints in an MCF problem is called a **pseudo flow**.

The **imbalance** at node i is $e(i) = b(i) + \sum_{(j,i)\in A} x_{ji} - \sum_{(i,j)\in A} x_{ij}$

supply/demand — inflow — outflow

If $e(i) > 0$, i is an **excess** node (belongs to set **E**).
If $e(i) < 0$, i is a **deficit** node (belongs to set **D**).

Note: $$\sum_{i\in N} e(i) = \sum_{i\in N} b(i) = 0.$$

Also, $$\sum_{i\in E} e(i) = -\sum_{j\in D} e(j)$$

If the network has an excess node, it must also have a deficit node.

The successive shortest path algorithm maintains optimality as $c^{\bar{x}}_{ij} \geq 0 \;\forall (i,j) \in G(\bar{x})$, and strives for feasibility, i.e., $e(i) = 0 \;\forall i \in N$.

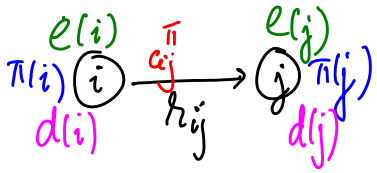
```
algorithm successive shortest path;
begin
    x := 0 and π := 0;
    e(i) := b(i) for all i ∈ N;
    initialize the sets E := {i : e(i) > 0} and D := {i : e(i) < 0};
    while E ≠ ∅ do
    begin
        select a node k ∈ E and a node l ∈ D;
        determine shortest path distances d(j) from node k to all
            other nodes in G(x) with respect to the reduced costs c^π_ij;
        let P denote a shortest path from node k to node l;
        update π := π − d;
        δ := min[e(k), − e(l), min{r_ij : (i, j) ∈ P}];
        augment δ units of flow along the path P;
        update x, G(x), E, D, and the reduced costs;
    end;
end;
```

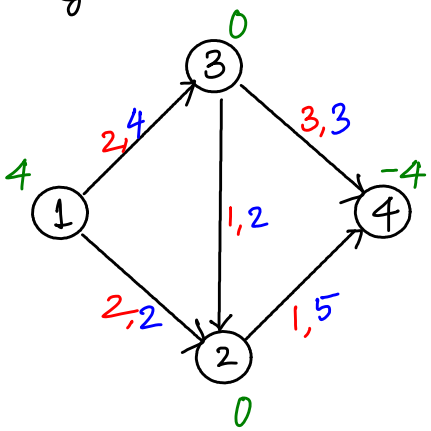
$r_{ij} = u_{ij}$ and $c^{\pi}_{ij} = c_{ij}$ at start

Figure 9.9 Successive shortest path algorithm.

Unlike the max-flow problem, we do not have a pair of nodes s and t here. Instead, we have collections of supply and demand nodes, and we could satisfy the demand at a demand node using the supply from any of the supply nodes.

Example

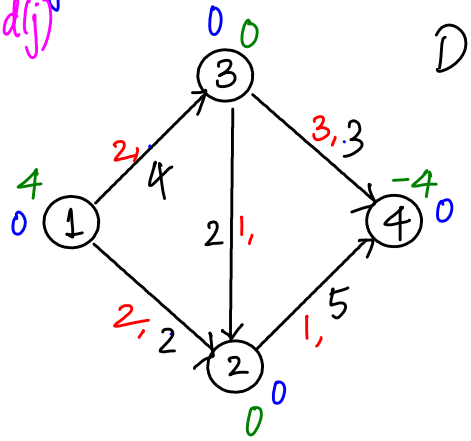




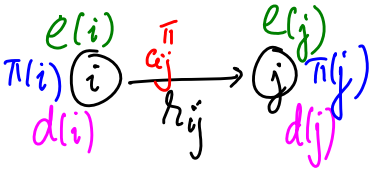
Original network

$E = \{1\}$

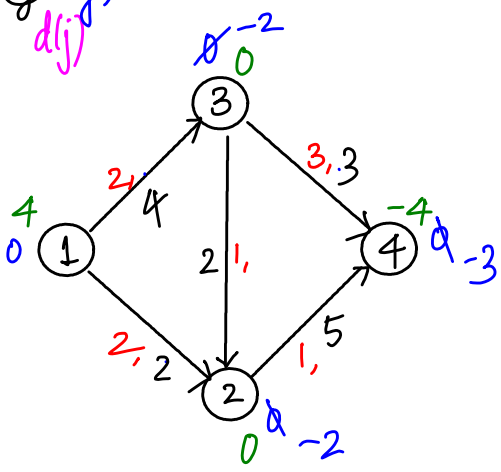
$D = \{4\}$



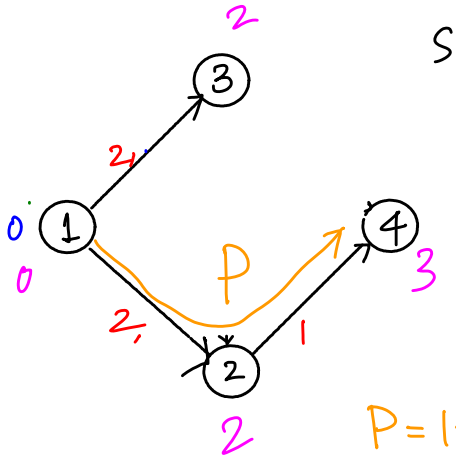
residual network at start



$k=1,\ l=4$



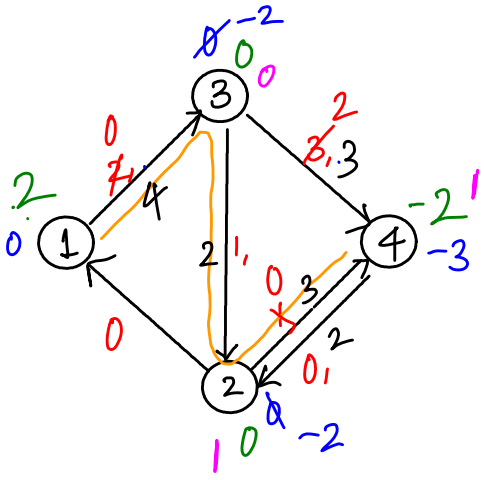
sp-tree (from $k=1$).



$P = 1\text{-}2\text{-}4,$

$\delta(P) = \min\{4, -(-4), 2, 5\} = 2$

$b(i),\ -b(4),\ r_{12},\ r_{24}$



$E=\{1\},\ D=\{4\}$

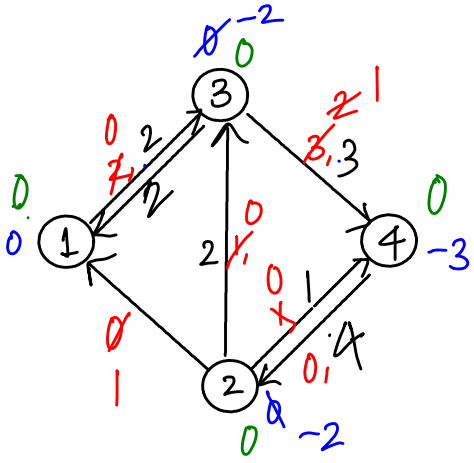
$P = 1\text{-}3\text{-}2\text{-}4,\ \delta(P) = \min\{2, -(-2), 4, 2, 3\} = 2.$

P is also the sp tree (from $k=1$)

$c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$

update

$(\bar{\pi} = \bar{\pi} - \bar{d})$



$E = D = \phi$ now

hence flow is optimal

Lemma 9.11 $\bar{x}$: pseudo flow, $\bar{\pi}$: node potentials. $\bar{x}$ satisfies $c_{ij}^{\bar{\pi}} \geq 0 \ \forall (i,j) \in G(\bar{x})$. $d(j)$: SP distances from some node s (in E) in $G(\bar{x})$ to node j with $c_{ij}^{\bar{\pi}}$ as costs. Then

(a) Let $\bar{\pi}' = \bar{\pi} - \bar{d}$. Then $c_{ij}^{\pi'} \geq 0$.

(b) $c_{ij}^{\pi'} = 0 \ \forall (i,j)$ in a shortest path from s to another node.

Proof $\bar{d}$: SP distances w.r.t. $c_{ij}^{\bar{\pi}}$.

$$\Rightarrow \quad d(j) \leq d(i) + c_{ij}^{\bar{\pi}} \quad \forall (i,j) \in G(\bar{x}).$$

But $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$, so

$$d(j) \leq d(i) + c_{ij} - \pi(i) + \pi(j)$$

$$\Rightarrow \quad c_{ij} - \underbrace{[\pi(i) - d(i)] + [\pi(j) - d(j)]}_{\pi'(i),\ \pi'(j)} \geq 0$$

$$\Rightarrow \quad c_{ij} - \pi'(i) + \pi'(j) \geq 0 \quad \Rightarrow \quad c_{ij}^{\pi'} \geq 0.$$

(b) $c_{ij}^{\pi} = 0$ along a shortest path, as

$$d(j) = d(i) + c_{ij}^{\pi} \quad \text{for } (i,j) \in \text{Shortest path.}$$

Thus, augmenting along the shortest path maintains the optimality conditions. When we augment along (i,j), we add r_{ji} in $G_1(\bar{x})$, for which $c_{ji}^{\pi} = -c_{ij}^{\pi}$. Since $c_{ij}^{\pi} = 0$, we avoid the possibility that $c_{ji}^{\pi} < 0$.

Hence, we maintain optimality $\left(c_{ij}^{\pi} \geq 0 \; \forall (i,j) \in G(\bar{x})\right)$ all along, and get feasibility when $E = D = \emptyset$.

MATH 566 – Lecture 24 (11/18/2014)

The minimum spanning tree (MST) Problem (Chapter 13-AMO)

$G=(N,A)$

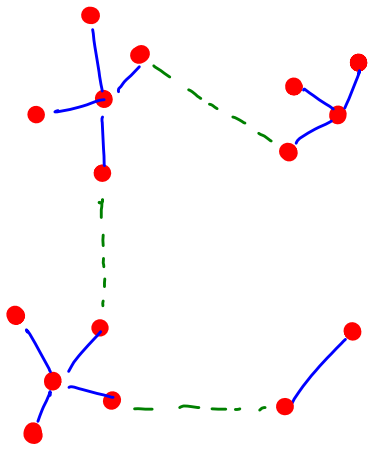
We consider **undirected** networks here.

Recall, a spanning tree T of G is a connected, acyclic subgraph of G which spans all nodes in N.

If $|N|=n$, then T has $n-1$ edges (or arcs).

A **minimum spanning tree** (MST) is a spanning tree that has the smallest total cost.

An application – Clustering (or cluster analysis)



Can start building an MST by assembling smaller trees. Stop at a specified cut off (for C_{ij}). The connected components form the clusters.

Optimality Conditions

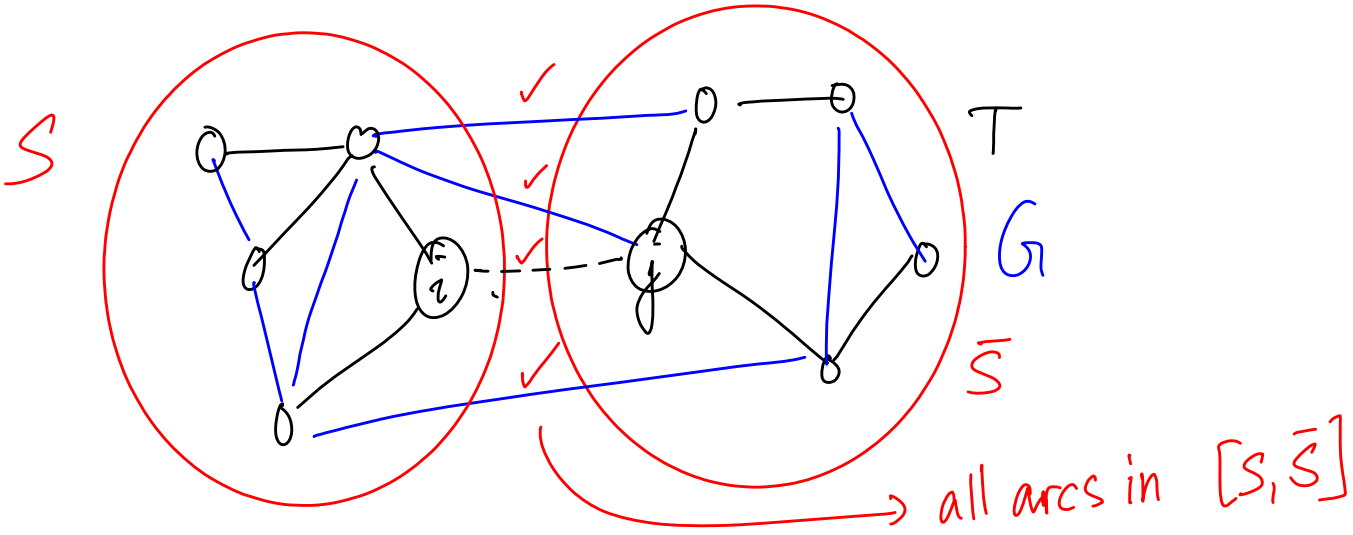
* Cut optimality conditions
* path optimality conditions

Notation: Arcs in the spanning tree T are **tree arcs**, and arcs not in T (but in G) are **nontree arcs**.

Observations

1. For every nontree arc (k,l), there is a unique path in T connecting k and l.

2. Deleting a tree arc (i,j) from spanning tree T divides N into disjoint subsets $S, \bar{S}$. The arcs (k,l) of G with $k \in S, l \in \bar{S}$ constitute a cut $[S,\bar{S}]$.



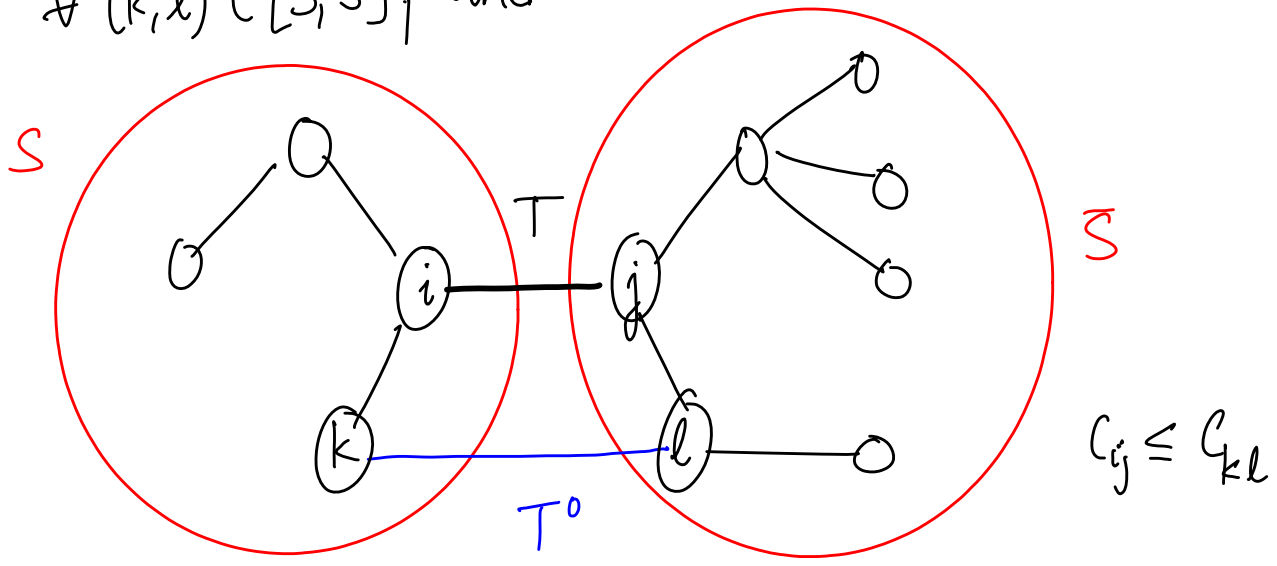
Cut Optimality Conditions

A spanning tree T is an MST iff $\forall (i,j) \in T$, $c_{ij} \leq c_{kl}$ for every $(k,l) \in [S, \bar{S}]$, where $[S, \bar{S}]$ is the cut formed by deleting (i,j) from T.

Proof

($\Rightarrow$) Assume T is an MST but $c_{ij} > c_{kl}$. Then we can replace (i,j) by (k,l) in T to obtain another spanning tree with smaller total cost than T, which contradicts the minimality of T.

($\Leftarrow$) we assume T satisfies $c_{ij} \leq c_{kl}$ $\forall (i,j) \in T$ and $\forall (k,l) \in [S, \bar{S}]$, and show that T must be an MST.



Let T^0 be an MST and $T^0 \neq T$.

$\Rightarrow \exists (i,j) \in T$ which is not in T^0.

$\Rightarrow \exists (k,l) \in T^0$ such that $k \in S$, $l \in \bar{S}$, where $[S, \bar{S}]$ is the cut obtained by deleting (i,j) from T.

By assumption, $C_{ij} \leq C_{kl}$

But since T^0 is an MST, we must have $C_{kl} \leq C_{ij}$.

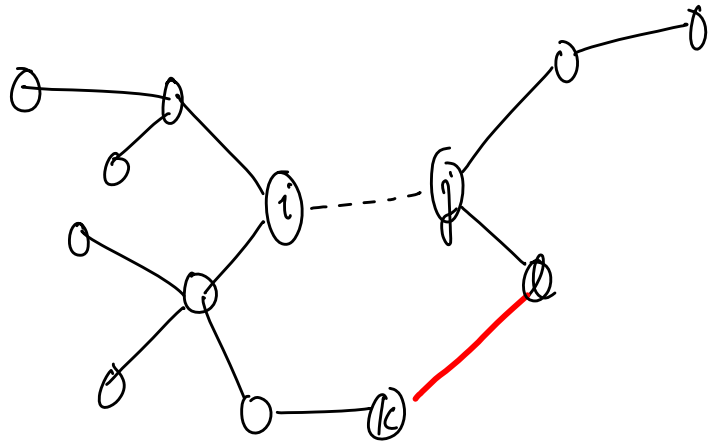
Hence $C_{ij} = C_{kl}$. So we can replace (i,j) in T by (k,l) without changing total cost of T. Repeat these replacement operations until $T = T^0$. Since the total cost remains unchanged, T must be an MST as well.

Path Optimality Conditions

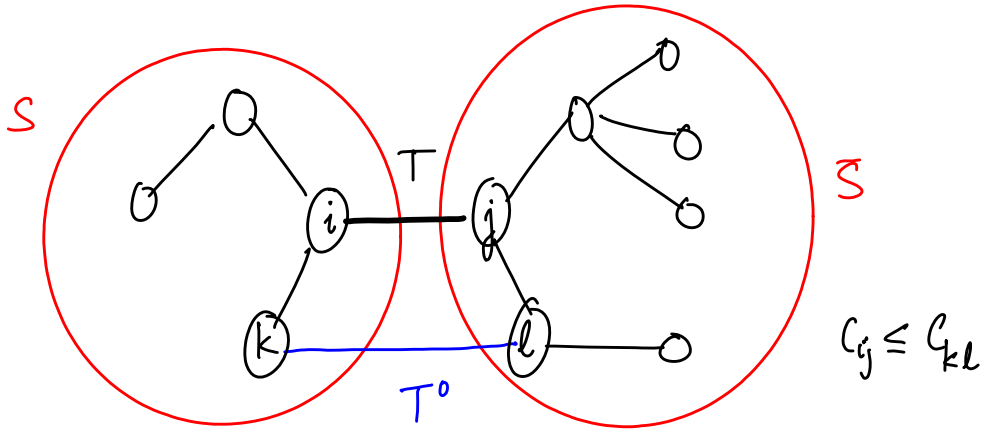
A spanning tree T is an MST iff for every non tree arc (k,l) in G, $C_{ij} \leq C_{kl}$ for each (i,j) in the path connecting k and l in T.

Proof

($\Rightarrow$) If T is an MST and $c_{ij} > c_{kl}$ for some nontree arc (k,l) and (i,j) in the path connecting k and l in T, then we can replace (i,j) with (k,l) in T and lower its total cost.



($\Leftarrow$) We start with a spanning tree T satisfying path optimality conditions, and show that it satisfies the cut optimality conditions.



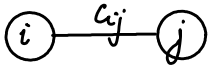
$(k,l) \in [S, \bar{S}]$. There is a unique path in T connecting k and l, and (i,j) is the only arc in T connecting S and $\bar{S}$, so (i,j) must be in this path.

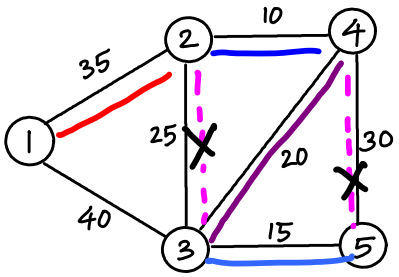
Path optimality conditions $\Rightarrow c_{ij} \leq c_{kl}$. This condition holds for every $(k,l) \in [S, \bar{S}]$, i.e., the cut optimality conditions hold. So T is an MST.

Kruskal's algorithm (for MST)

* uses path optimality conditions
* builds spanning tree by adding one arc at a time.
* uses a sorted LIST of arcs (sorted ascendingly in the order of c_{ij}'s)
* Checks whether adding an arc creates a cycle
  - if no, add to tree
  - if yes, discard arc from LIST.

Example



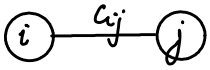


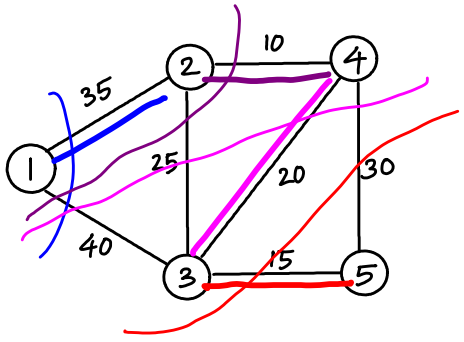
Arcs considered in the order indicated by going from blue to red.

Prim's Algorithm

* uses cut optimality conditions
* builds a spanning tree by fanning out from a single node, adding one arc at a time.
* maintains a spanning tree of the subset S of nodes, and adds $(i,j) \in [S, \bar{S}]$ with the smallest c_{ij} to tree.

Same example





Let $S = \{1\}$ at start.

$S \to \{1\} \to \{1,2\} \to \{1,2,4\}$

$\to \{1,2,4,3\} \to N$

Sollin's algorithm → combines Kruskal's and Prim's algos

Back to Kruskal's algorithm - Complexity

Sort m arcs (using c_{ij}'s) – $O(m \log m) = O(m \log n^2)$

$= O(m \log n)$.

How to find if adding (i,j) creates a cycle?

→ use UNION-FIND data structure

Operations

ADD(i): Add $\{i\}$ as a singleton set

FIND(i): Find (the "name" of) the set containing i

UNION(A,B): replace A, B by $A \cup B$.

more details to follow in the next lecture.

MATH 566 – Lecture 25 (11/20/2014)

Kruskal's algorithm for MST – Complexity

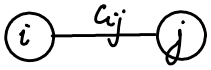
* Sorting all arcs using c_{ij}'s – $O(m \log n)$.
* Detecting cycles

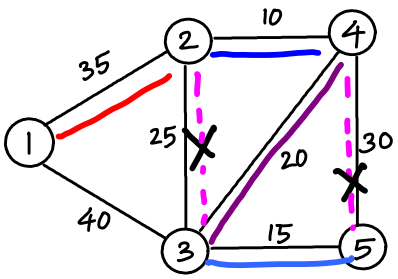
  LIST (of nodes in the tree) is a forest at any intermediate step. Let the subtrees be $N_1, N_2, N_3, \ldots$ (forest: collection of disjoint trees)

  For each arc (k,l), check if k and l belong to the same subtree N_i. If YES, (adding (k,l) will create a cycle) discard (k,l). If NO, merge N_k and N_l into one set. We have to do $O(n)$ work for each arc (check at most n nodes twice, once each for k and l).

  There are m arcs, so overall complexity is $O(mn)$.

The Union-Find data structure allows one to implement these operations more efficiently.





Arcs considered in the order indicated by going from blue to red.

Operations in Union-Find data structure are ADD(i), FIND(i), and UNION(A, B).

↳ returns "name" of set containing i. The "name" could be the smallest i included in that set.

Start by calling ADD(i) $\forall i \in N$.

Sort arcs, and consider arcs one by one.

(2,4) FIND(2) ≠ FIND(4)
2 4

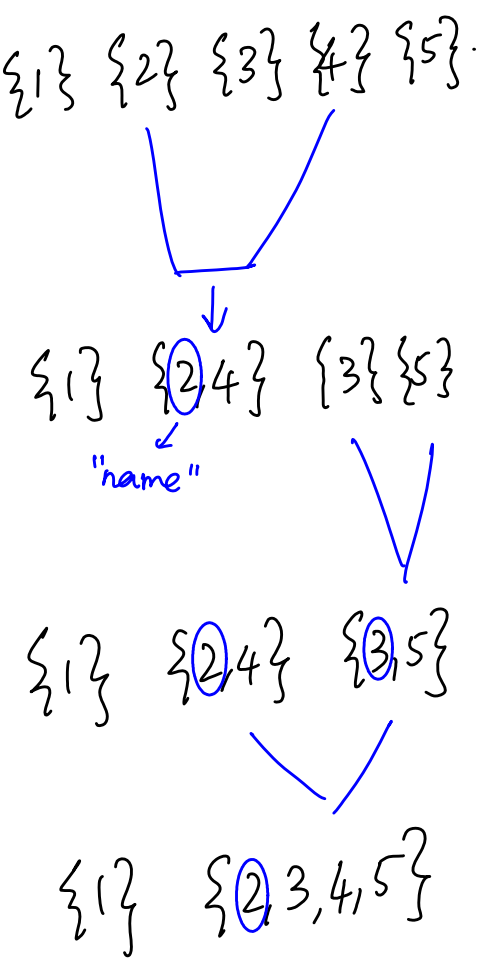
So UNION(FIND(2), FIND(4))

(3,5) FIND(3) ≠ FIND(5), so do UNION

(3,4) FIND(3) = 3, FIND(4) = 2

do UNION

N_i's



(2,3) FIND(2) = FIND(3) = 2

So discard (2,3)

(4,5) FIND(4) = FIND(5) = 2

So discard (4,5)

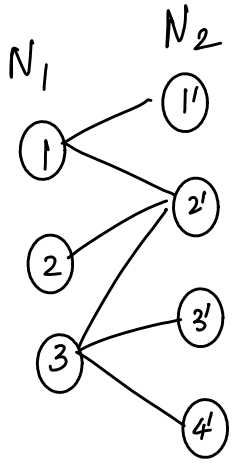
(1,2) → do UNION {1,2,3,4,5}

Can implement using linked lists, and do not have to search O(n) nodes for each k and l. This search can be done in log(n) time. So complexity is O(m log n).

Assignments and Matching Problems (AMO chapter 12)

↳ also called bipartite matching problem

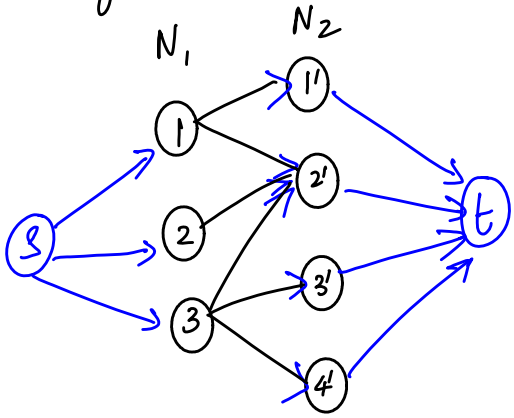
1. Bipartite cardinality matching problem: $G = (N_1 \cup N_2, A)$ where A has undirected arcs (i,j) with $i \in N_1$, $j \in N_2$. The goal is to match as many nodes in N_1 with unique nodes in N_2.

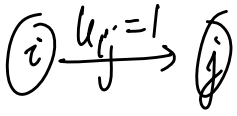


We can model this problem as a max flow problem on a **simple network**, in which every arc (i,j) has $u_{ij} = 1$, and each node i has either indegree$(i) \le 1$ or outdegree$(i) \le 1$.

We first transform G to a directed network G' with (i,j) always going from $i \in N_1$ to $j \in N_2$. Add nodes s, t, arcs $(s,i)\ \forall i \in N_1$, $(j,t)\ \forall j \in N_2$, and set $u_{ij} = 1\ \forall (i,j)$.

G'





We can show that a matching of cardinality k in G corresponds to a flow with value k in G'.

($\Rightarrow$) straightforward. → set $x_{si} = x_{ij} = x_{jt} = 1 \ \forall (i,j) \in$ Matching. Then value $v = k$.

($\Leftarrow$) Given a flow with value k in G', we use flow decomposition to obtain k path flows of the form $s-i-j-t$ with flow value 1 each. We match each such i and j to define the matching.

Max flow on simple networks can be solved in $O(m\sqrt{n})$ time (AMO Chapter 8). Hence bipartite matching can be solved in $O(m\sqrt{n})$ time as well.

→ the problem is easier than the general max-flow problem.

2. Bipartite Weighted Matching Problem

$G = (N_1 \cup N_2, A)$, $|N_1| = |N_2| = n$. c_{ij}'s are costs

Goal is to find a perfect matching, i.e., match all nodes, of minimum total weight.

We assume G is directed. If not, direct arcs to go from N_1 to N_2.

This is a special case of the min cost flow problem. The linear optimization model is given below.

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{(i,j)} x_{ij} = 1 \quad \forall\, i \in N_1$$

$$-\sum_{(i,j)} x_{ij} = -1 \quad \forall\, j \in N_2$$

$$0 \le x_{ij} \le 1$$

↳ UB of 1 is implied for each (i,j) by the flow balance constraints above and $x_{ij} \ge 0$.

We can adapt algos for MCF to the assignment problem.

Successive shortest path algorithm

Recall: $(\bar{x}, \bar{\pi})$ are optimal iff $c_{ij}^{\bar{\pi}} \geq 0 \quad \forall (i,j) \in G(\bar{x})$

Here, augmenting 1 unit corresponds to assigning one additional node in N_1.

If $S(n,m,C)$ is the complexity for solving the SP instances, the assignment problem can be solved in $O(n\,S(n,m,C))$.

Relaxation algorithm: Allow nodes in N_2 to be over- and under-assigned (under assigned means not assigned). Then pick node $k \in N_2$ that is over assigned, find SP distances from k to all nodes in $G(\bar{x})$ with $c_{ij}^{\bar{\pi}}$, pick a node $l \in N_2$ that is underassigned, and augment 1 unit from k to l.

Stable Marriage Problem

n men, n women, each person provides a ranking of the n members of the other sex (from most to least preferred).

A (man, woman) pair is **unstable** for a given matching if they are not married, but they prefer each other to their current spouses.

More on this problem after the break...!

MATH 566 – Lecture 26 (12/02/2014)

Stable Marriage Problem

n men, n women

A (man, woman) pair is **unstable** if they are not married, but prefer each other to their current spouses.

The stable marriage problem tries to identify a perfect matching that is **stable**, i.e., it has no unstable pairs.

Q: Does a stable perfect matching always exist? YES!

We describe an $O(n^2)$ algorithm to construct such a matching.

This is the **propose-and-reject** algorithm, which is a greedy algo.

Input Two $n \times n$ matrices of rankings by the men and women. The rankings are integers in $[1, n]$ (each person produces a permutation of $1 \ldots n$, for instance.

Set-up The algo maintains a LIST of unassigned men, and each man has a current-woman, who is the woman in his priority list to whom he will propose next.

* At each step, pick a man from LIST, say Bill.
* Bill proposes to his current-woman, say, Helen.
* If Helen is unassigned, then she accepts, and [Bill, Helen] are engaged.
* If Helen is already engaged to Frank, she selects her preferred man (from among Frank and Bill), and rejects the other.
* Rejected man is added back to LIST, and he designates the next woman in his preferred list as his current-woman.
* Go until LIST is empty, and then engaged couples are married.

Correctness If Dave prefers Laura over wife (as decided by the matching), he would've proposed to Laura earlier, and Laura would've rejected Dave in favor of someone she likes more. Since no woman switches to a man she prefers less, Laura prefers her husband to Dave. Hence [Dave, Laura] is not an unstable pair.

Hence the matching found is stable.

Def A [man, woman] pair are called **stable partners** if they are matched in some stable matching.

Notice there can be many stable (perfect) matchings.

Lemma 12.4 In the propose-and-reject (P&R) algorithm, a woman never rejects a stable partner.

proof Let M^* be the stable matching given by the P&R algorithm. Suppose Lemma is false, i.e., women do reject stable partners. Let the first time a woman rejects a stable partner be when Joan rejects Dave.

Let M° be a stable matching in which [Dave, Joan] is a stable pair.

Let the rejection happen because Joan is engaged to Steve, whom she prefers to Dave. Since this is the first such rejection of a stable partner by a woman in M^*, no other woman rejected a stable partner before.

So, Steve prefers Joan to all other stable partners.

In M^0, let [Steve, Sue] be a stable pair, and be matched.

Look at the pair [Steve, Joan] → not matched in M^0.

- Steve prefers Joan to Sue
- Joan prefers Steve to her partner

Hence, [Steve, Joan] is an unstable pair, and hence M^0 is not a stable matching – a contradiction.

We can argue that the P&R algo creates a **man-optimal matching**, in which each man is married to his best possible **stable** partner. This result follows from the fact that men propose to women in decreasing order of their preferences, and no woman ever rejects a stable partner.

This is a somewhat surprising result – if each man is given his best stable partner independently, we get a stable matching!

On the other hand, we can show that the P&R algo produces a **woman-minimal matching**, i.e., each woman gets her worst **stable** partner.

Complexity of P&R algorithm

Each woman either

(1) gets her first proposal, or → n times in total

(2) rejects a man. → at most $(n-1)$ times for each woman.

Step (2) is the bottleneck, taking $O(n^2)$ time. The data input is $O(n^2)$ (two $n \times n$ matrices of preferences). Hence the P&R algo is optimal, since handling the data itself takes $O(n^2)$ time.

In other words, we cannot design an algorithm that runs in time faster than $O(n^2)$.

Back to non-bipartite (cardinality) matching.

Undirected network $G = (N, A)$, $|N| = n$.

Hence the size of any matching $\leq \left\lfloor \frac{n}{2} \right\rfloor$

↙ the arcs that identify the matched pairs

Did course evaluations... more on matching in next lecture...

MATH 566 – Lecture 27 (12/04/2014)

Hw12 posted (due in a week).

Nonbipartite Cardinality Matching

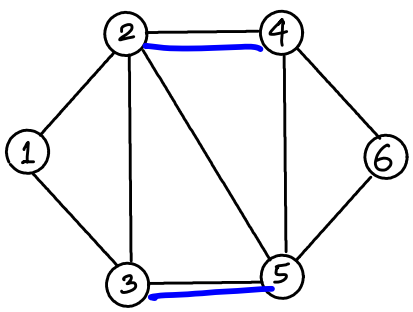
$G=(N,A)$ is an undirected network. We could maintain **node adjacency lists** here, i.e.,

$$A(i) = \{j \in N \mid (i,j) \in A\}.$$

A **matching** M of $G=(N,A)$ is a subset of arcs such that no two arcs in the subset are incident to the same node. Arcs in M are matched arcs, rest are unmatched arcs. Similarly, we defined matched and unmatched nodes.

Size of any matching is at most $\lfloor \frac{n}{2} \rfloor$.

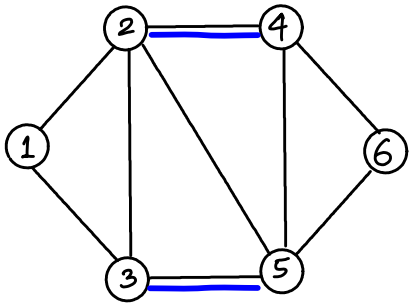
Example



$M=\{(2,4),(3,5)\}$ is a matching

We use augmenting paths (and cycles) to increase the cardinality of a matching – similar to max-flow.

Def A path $P = i_1 - i_2 - \cdots - i_k$ is an **alternating path** w.r.t. a matching M if every consecutive pair of arcs in P contains one arc in M and one not in M.



e.g., $1-2=4-6$
$1-3=5-2=4-6$
} odd alternating path

$3=5-2=4-6$ → even alt. path

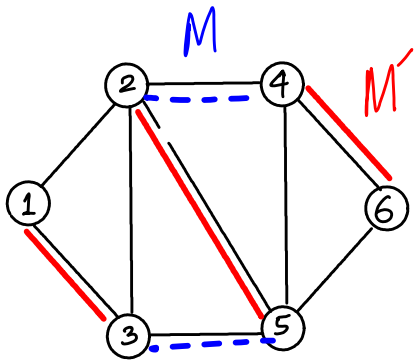
An alternating path is an odd (even) alternating path if it has an odd (even) # arcs.

An alternating cycle is an alternating path that starts and ends at the same node.

For instance, $2=4-5=3-2$ is an odd alternating cycle.

Def An odd alternating path w.r.t matching M is an **augmenting path** if its first and last nodes are unmatched in M.

We can swap the matched and unmatched arcs in an augmenting path to get matching M' with $|M'| = |M| + 1$.



$1 - 3 = 5 - 2 = 4 - 6$ is an augmenting path.

M' has a cardinality of 3, while M had cardinality 2.

We use set theoretic notation to describe these operations

The **symmetric difference** of two sets S_1, S_2 is

$$S_1 \oplus S_2 = (S_1 \cup S_2) - (S_1 \cap S_2).$$

eg., $\{2,3,5,6\} \oplus \{3,6,8,4\} = \{2,4,5,8\}$

collection of elements in either set which are not in both of them.

Here, we look at arcs that are in one matching, but not in both.

In the network above, we found $M \oplus P$, where P is the augmenting path.

Property 12.6 Let P be an augmenting path w.r.t. matching M. Then $M \oplus P$ is a matching with size $|M|+1$.

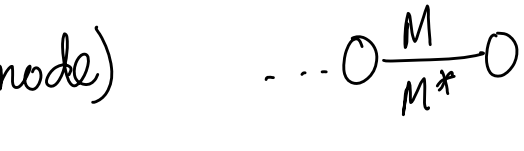
All nodes matched in M remain matched in $M \oplus P$, and two additional nodes are matched.

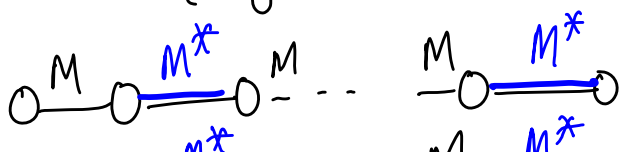
Def An **augmentation** of a matching M is to find an augmenting path P and find $M \oplus P$.

This suggests an augmenting path algorithm for cardinality matching. Start with some matching, pick a node that is not matched, find an augmenting path starting at that node, and augment.

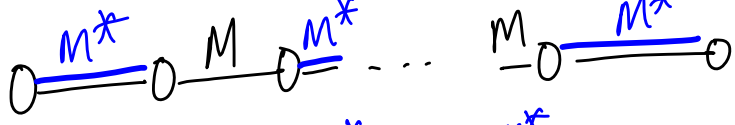
But what if we cannot find an augmenting path for an unmatched node? We need a couple more results to depict how to handle this situation.

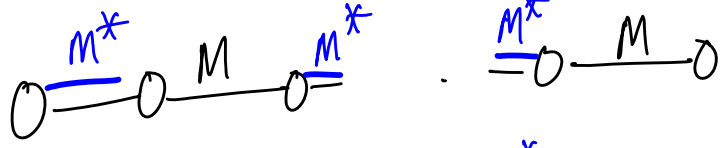
Property 12.7 Let M, M^* be two matchings. Then $M \oplus M^*$ defines $G' = (N, M \oplus M^*)$, a subgraph of G, with the property that every component of G' is one of the following six types.

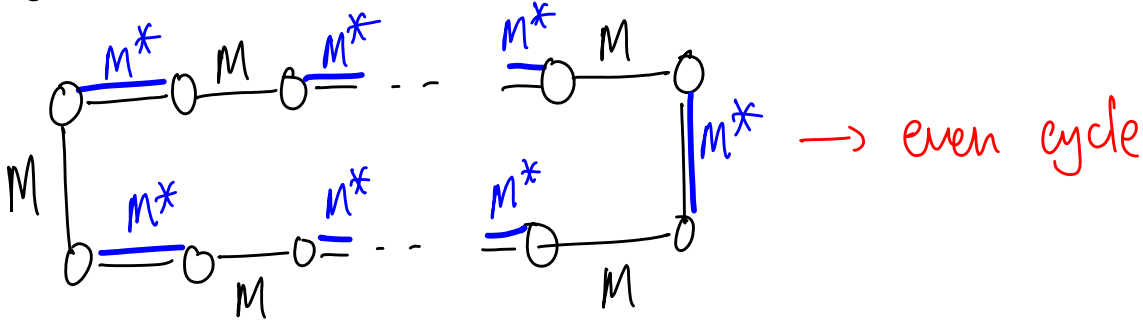
(a) 

(b) 

(c) 

(d) 

(e) 

(f) 

This result follows from the fact that in G', each node has degree 0, 1, or 2, and the above six are the only possible configurations.

The augmenting path algorithm depends on the following result.

Theorem 12.8 If node p is unmatched in matching M, and M contains no augmenting path starting at p, then node p is unmatched in some maximum matching.

Proof: Let M^* be a maximum matching. If p is not matched in M^*, we are done. So, assume p is matched in M^*. Consider $M \oplus M^*$. This subgraph will have components as described in Property 12.7. Since p is unmatched in M, p has to be the starting node in a component of the form (d) or (e).

There is no augmenting path starting at p in M, so we are left with (e), the even path P.

Then, $M' = M^* \oplus P$ is also a maximum matching, as P is an even path, and p is unmatched in M'.

Hence, we can describe the algorithm as follows.

- Start with any matching (e.g., zero matching).
- pick a node that is unmatched.
- search for augmenting path starting with that node.
  if found augment;
  else delete this node and all arcs incident to it.

Unfortunately, this algo is guaranteed to work only for bipartite networks ☹!

MATH 566 – Lecture 28 (12/09/2014)

The traveling salesman problem (TSP)

$G = (N, A)$ directed graph with costs c_{ij} on edge (i, j).

Objective: find a Hamiltonian tour of minimum total cost.

↳ a tour that visits every node exactly once

Assumption 1: Costs are symmetric $c_{ij} = c_{ji}$.

The asymmetric version is much harder to solve!

P and NP: classes of problems

shortest path, max flow, MCF, MST, etc.

The problems we have studied so far belong to P, which is the class of problems for which there are polynomial time algorithms.

TSP is NP-hard, i.e., there is no polynomial time algo known to solve TSP.

So we consider heuristics and approximation algorithms.

↙

no guarantees on solution quality, but might work well in practice.

Approximation algorithms A is an ε-approximation algorithm if for $\varepsilon > 0$, on any instance I, the solution S given by A satisfies

$$\text{cost}(S) \leq (1+\varepsilon)\,\underbrace{\text{Opt}(I)}_{\text{cost of an optimal solution for } I.}$$

Another notation: δ-approximation algs for $\delta > 1$, with $\delta = 1+\varepsilon$.

A lower bound for TSP

A minimum spanning tree (MST) gives a lower bound for the optimal traveling salesman (TS) tour.

If we delete one arc from any Hamiltonian tour, we get a spanning tree. So

$$\text{Opt}_{TSP}(I) \geq \text{Cost}(MST).$$

The goal is to find also upper bounds. This is a typical approach for optimization problems — find lower and upper bounds for the optimal value. If these bounds coincide, that common value is indeed optimal.

Heuristics
- construction → build Hamiltonian tour from scratch
- improvement → improve existing Hamiltonian tour

Insertion Heuristics

1. Nearest insertion heuristic (greedy).
   Insert j into partial tour W such that $d(j,W) = \min_{l \notin W} d(l,W)$.
   shortest distance from l to any node in W

2. Farthest insertion heuristic.
   Insert j into W s.t. $d(j,W) = \max_{l \notin W} d(l,W)$.

   Idea: We have to include every node ultimately, so might as well include the farthest one first.

3. Random insertion heuristic.
   Pick $j \notin W$ randomly and add to W.

(iv) Cheapest insertion heuristic (somewhat greedy)

Select $k \notin W$ such that $c_{ik} + c_{kj} - c_{ij}$ is smallest (could search for this k for every consecutive pair (i,j) in W).

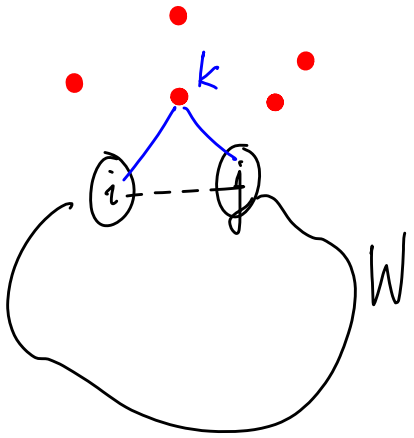
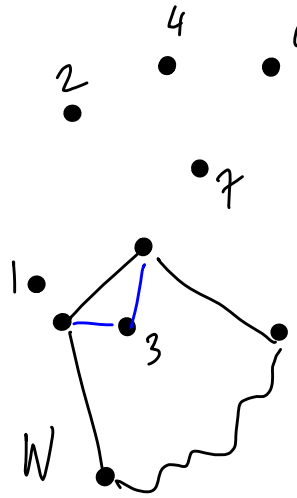


Illustration → Assume (i,j) exists for every pair of nodes i and j, and that c_{ij} is the Euclidean length between the nodes.



nearest insertion: picks 1.
farthest insertion: picks 9.
cheapest insertion: picks 3.

Average quality on ≈ 40 benchmark instances

nearest insertion – 20%
farthest insertion – 10%
cheapest insertion – 17%
random insertion – 11%

pay high price for being greedy

Improvement Heuristics

Assumption 2: Costs satisfy the triangle inequality, i.e.,

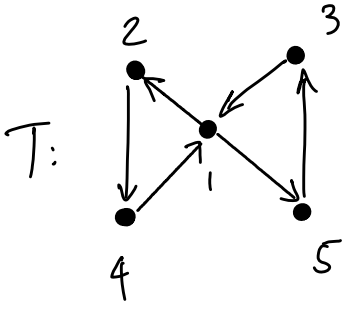
$$c_{ij} + c_{jk} \geq c_{ik}.$$

(we are essentially working with points on the 2D-plane, and c_{ij} = Euclidean distance between i and j).

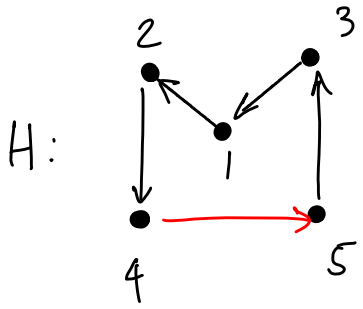
Also assume (i,j) is present for all nodes i & j.

Equivalently, we just consider points on the plane, and pick undirected straight line segments as the arcs between them

If the triangle inequality holds, we can improve any tour to a Hamiltonian tour by skipping nodes already visited.



tour T: 1, 2, 4, 1, 5, 3, 1 (the second 1 is crossed out)



H: 1, 2, 4, 5, 3, 1

Generic algorithm

Input Tour $T = v_{i_0}, v_{i_1}, \ldots, v_{i_n}$ (not necessarily Hamiltonian).

Initialization $P = \{v_{i_0}\}$ (path)

$T = T \setminus \{v_{i_0}\}$ (remove v_{i_0} from T)

```
while T ≠ ∅
    Let v be the first node in T
    if v ∉ P
        P = P ∪ {v};   (add v to end of P)
    end_if
    T = T \ {v};
end_while
```

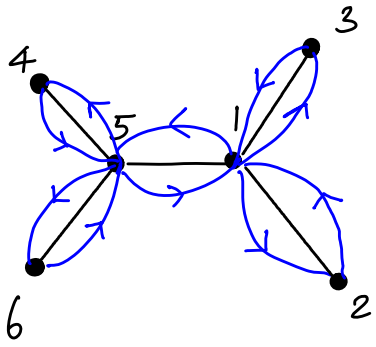
Close path P (if needed) to obtain Hamiltonian tour H.

We consider a couple ways to generate "good" tours, on which we could apply the generic algorithm.

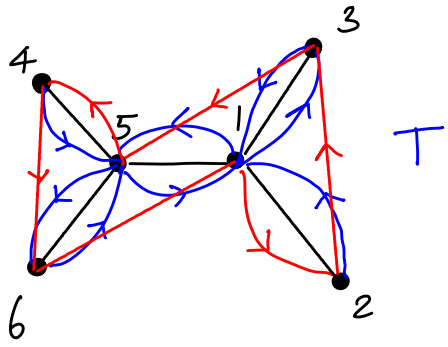
(i) Double-tree algorithm

Let T^* be an MST of G. Start with tour $T = \text{double}(T^*)$, and apply generic algorithm to T to get Hamiltonian tour H.

T^* (MST)



T



T

$T = 1, 2, 1, 3, 1, 5, 4, 5, 6, 5, 1$

$H = 1, 2, 3, 5, 4, 6, 1$

Let H^* be an optimal Hamiltonian tour.

$\underline{\text{Cost}}(T^*) \leq \text{Cost}(H^*)$ → sum of C_{ij}'s in T^*

$\Rightarrow \text{Cost}(T) = 2\,\text{Cost}(T^*) \leq 2\,\text{cost}(H^*)$

$\Rightarrow \text{cost}(H) \leq \text{Cost}(T) \leq 2\,\text{cost}(H^*)$.

So the double-tree algorithm gives a Hamiltonian tour that has at most double the total cost of an optimal Hamiltonian tour.

Hence, the double-tree algorithm is a 1-approximation algorithm ($\varepsilon = 1$).

MATH 566 – Lecture 29 (12/11/2014)

(ii) Christofides' algorithm

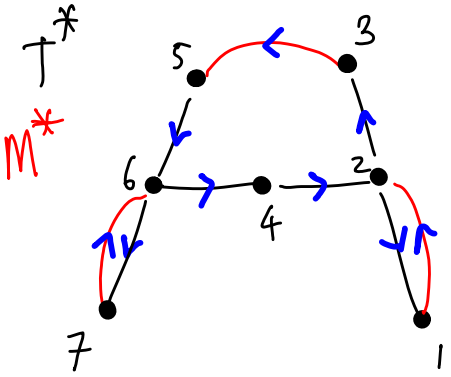
Recall, T^* is an MST. Let $N_{odd} \subseteq N$ be the set of nodes in $G=(N,A)$ with odd degrees in T^*. Let M^* be a minimum weight matching on N_{odd}.

Match each node in N_{odd} with another node in N_{odd}, total cost is the sum of c_{ij}'s of arcs used in the matching.

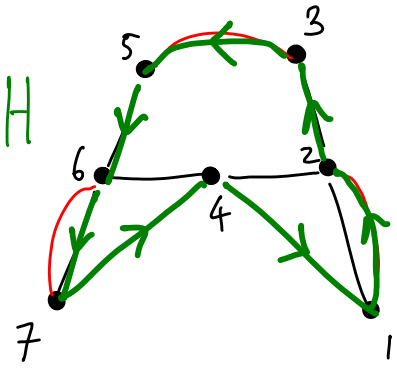
Notice that $|N_{odd}|$ is even, as the number of odd degree nodes in a tree is even.

We construct tour T as $T^* \cup M^*$. Notice that all nodes in $T^* \cup M^*$ have even degrees

We now apply the generic algorithm to convert T to a Hamiltonian tour H.



Tour T (>)
= 1, 2, 3, 5, 6, 7, 6, 4, 2, 1



Claim $\text{Cost}(M^*) \leq \frac{1}{2}\text{cost}(H^*)$

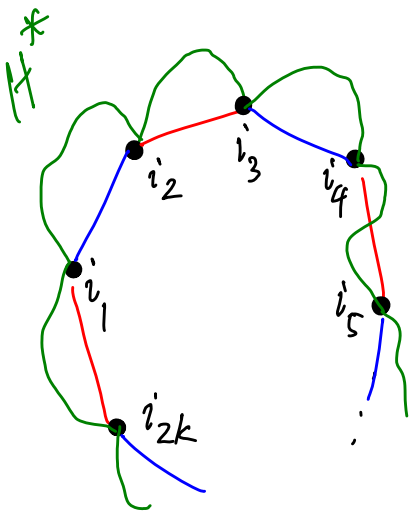
Notice, first, that we pick in M^* at most half the number of edges in H^*. ($|N_{odd}| \leq n$, and we are matching nodes in N_{odd} in pairs).

Proof Let $N_{odd} = \{i_1, i_2, \ldots, i_{2k}\}$, and let $i_1, i_2, \ldots, i_{2k}$ be the order in which these nodes appear in H^*.

Consider two matchings

$M_1 = \{(i_1, i_2), (i_3, i_4), \ldots, (i_{2k-1}, i_{2k})\}$ and

$M_2 = \{(i_2, i_3), (i_4, i_5), \ldots, (i_{2k}, i_1)\}$.



(even-degree nodes not shown here)

$M_1 \cup M_2$ gives a subtour, and not necessarily a Hamiltonian tour.

By triangle inequality, $\text{Cost}(H^*) \geq \text{Cost}(M_1) + \text{Cost}(M_2)$

$\geq 2\,\text{Cost}(M^*)$

$\Rightarrow \text{Cost}(M^*) \leq \frac{1}{2}\,\text{Cost}(H^*)$.

We have already seen that $\text{Cost}(T^*) \leq \text{Cost}(H^*)$.

Hence, for tour H given by the Christofides' algorithm,

$$\text{Cost}(H) \leq \text{Cost}(T^*) + \text{Cost}(M^*) \leq \frac{3}{2}\,\text{Cost}(H^*).$$

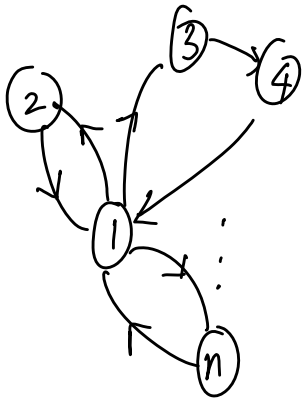
So, Christofides' algorithm is a $\frac{1}{2}$-approximation algorithm ($\varepsilon = \frac{1}{2}$) for Euclidean TSP.

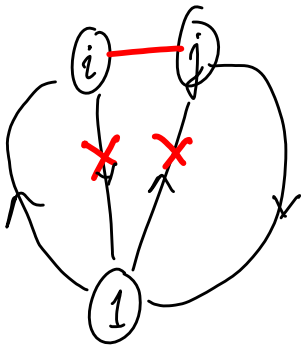
In practice, the average quality of Christofides' algorithm is $\sim 10\%$ (above the optimal solution).

We describe a few more improvement heuristics.

(iii) Savings heuristic

Start with a bunch of subtours going from home node, say node 1, to other (may be more than one) nodes, and coming back to home node.



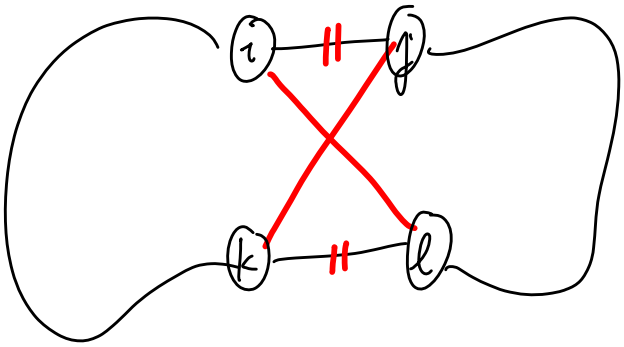


merge two subtours rooted at 1.

Choose i, j, such that the savings $C_{1i} + C_{1j} - C_{ij}$ is maximum.

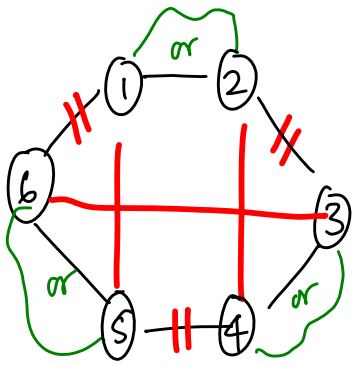
Average quality of savings algorithm is 10%.

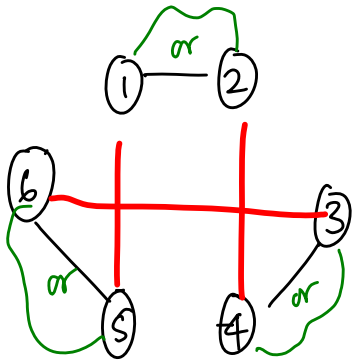
(iv) 2-opt interchange



Do the interchange when $(C_{ij} + C_{kl}) - (C_{il} + C_{kj}) > 0$.

(v) 3-opt interchange





In the general case, (1,2) could be a sequence of edges.

Need $\underbrace{(C_{23} + C_{45} + C_{61}) - (C_{15} + C_{24} + C_{36})}_{\text{Savings}} > 0$

Average quality of 2-opt and 3-opt interchange heuristics $\approx$ 4% each.