MATH 566: Lecture 1 (08/20/2024)

This is Math 466/566 Network Optimization
I'm Bala Krishnamoorthy (Call me Bala).

Today
* syllabus, logistics
* The Königsberg bridges problem
* Network flow problems

A field that is closely related to network optimization is graph theory. While we will use several results that might also typically be presented in a graph theory course (Math 453/553), we will concentrate more on the "network" aspects.

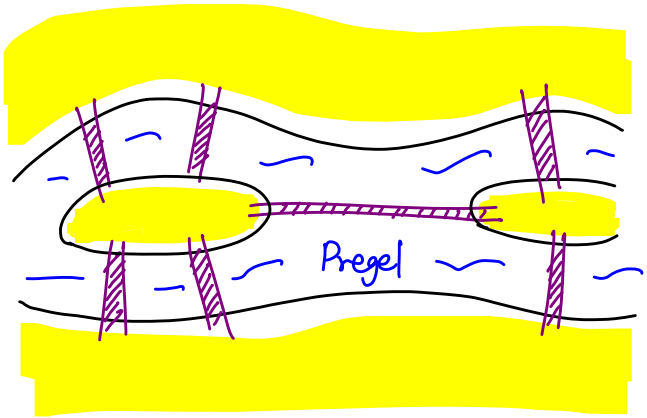
One key difference between "graphs" and "networks" as used conventionally, is that the arcs (or edges) in networks have capacities. In other words, one could think of them as pipes with limits on how much fluid could be sent through them at a time. On the other hand, edges in "graphs" are typically "lines"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

We will also emphasize **implementation of several algorithms**. Thus, there will be a sizeable programming component in this class.

The Königsberg Bridges Problem

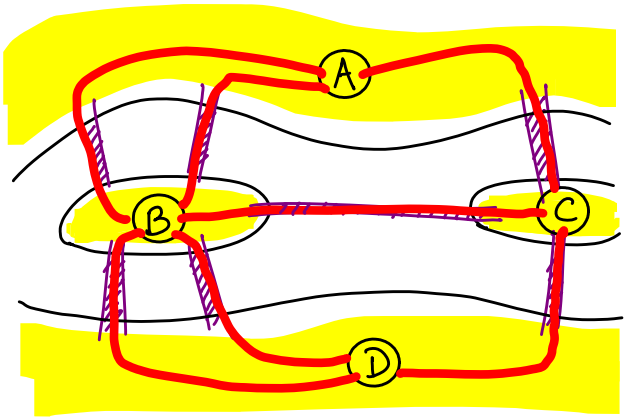
We start will a historical note. The first paper in graph theory (and, by extension, in network optimization) was published by Euler in 1736. He was visiting Königsberg in Prussia (now in Kaliningrad) The river Pregel flows through the town, and had seven bridges across it.

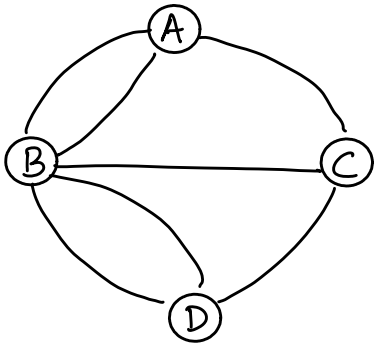


The citizens asked him if one could cross each of the seven bridges exactly once, and return to the starting point?

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"?

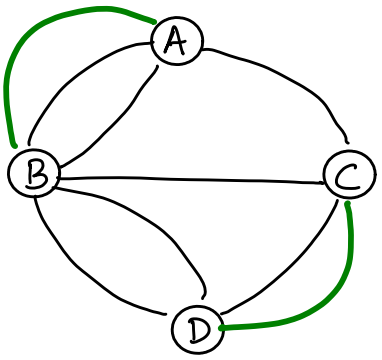
Theorem An undirected graph has an Eulerian cycle iff

* 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.

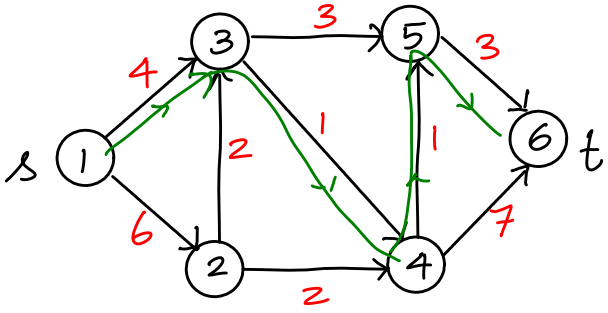
One more point about notation. In graph theory, it is common to call a graph as $G = (V, E)$, where V is the set of vertices and E the set of edges (undirected, by default). We will use the notation $G = (N, A)$ to denote a network, where N is the set of nodes and A is the set of arcs, which are directed by default.

Network Flow Problems

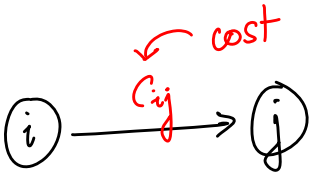
① Shortest path problem

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 arc set A.

Here is an example:



Notation:



optimal or shortest path is shown in green

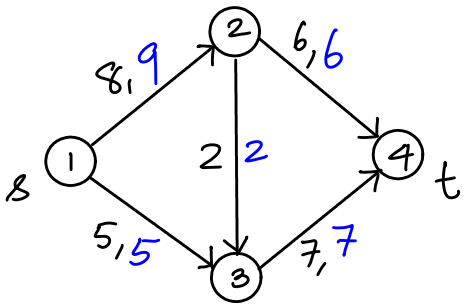
A typical application is driving directions on Google Maps. The costs c_{ij} could just be the distances between the cities modeled by the nodes, or could capture other relevant info such as traffic, tolls, etc.

Notice that the only parameter specified for the arcs is the cost, and the nodes themselves have no additional attributes (except for two of them being specified as s and t).

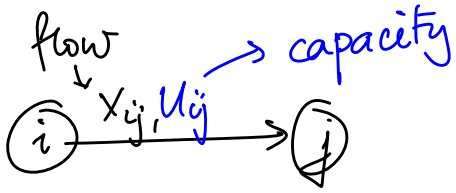
(2) Maximum flow problem (or max-flow)

Objective: Find a "feasible" flow (i.e., a solution that honors capacity limits on arcs) that sends maximum flow from a source node s to a sink node t.

Example:



notation:



max flow value $v = 13$.

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

Notice that we are working with directed arcs. This will be the default mode going forward, unless mentioned otherwise.

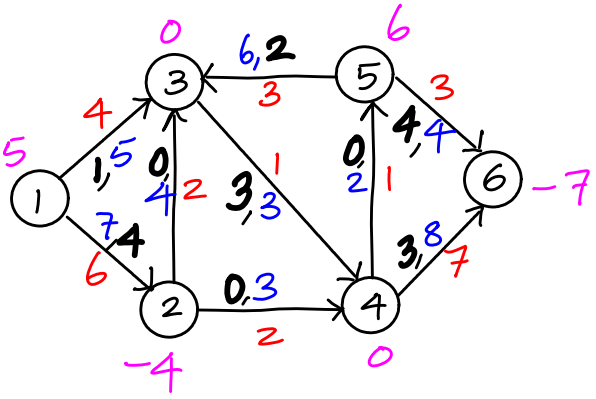
In the above instance, notice that the total flow coming into nodes 2 and 3 is equal to the total flow going out of that node. These nodes are "transshipment nodes", i.e., there is no loss or addition of material in these nodes. For the node s, we assume there is an infinite supply of material to be sent to node t, honoring the capacities.

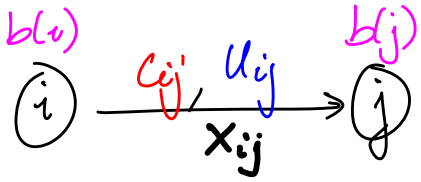
We point out that the set of parameters specified in max flow include u_{ij}'s, the capacities on arcs. Notice there are no costs (c_{ij}) specified (unlike in the shortest path case).

(3) Minimum cost flow (min-cost flow problem)

Objective: determine a least-cost shipment of commodity through a network in order to satisfy demands at certain nodes using supply at certain other nodes.

An illustration:





b(i): supply/demand

Nodes 1, 5: supply nodes (S)
2, 6: demand nodes (D)
3, 4: transshipment nodes.

We assume $\sum_{i \in N} b(i) = 0$ by default. In other words, the total supply in all supply nodes (S) is equal to the total demand in all demand nodes (D). The goal is to ship the supply from S nodes to D nodes at the minimum total cost while honoring capacity limits on arcs.

In min-cost flow, we specify costs (c_{ij}), capacities (u_{ij}), and supply/demand values at nodes (b_i).

MATH 566: Lecture 2 (08/22/2024)

Today:
* Formal definition of MCF
  - SP, MF as cases of MCF
* circulation, transportation, seat-sharing problem

We now specify the mathematical model for the MCF problem. Even though we introduce the model now, we will not solve the problems using this model – we will present efficient algorithms that exploit the network structure better (which the model does not do).

Notation

→ G for "graph"

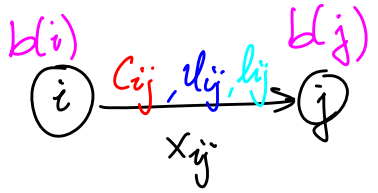
$G = (N, A)$; G: directed network, N: set of nodes, A: set of directed arcs.

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

c_{ij} : unit cost on arc $(i,j) \in A$ (can be $>0, <0$, or $=0$)

u_{ij} : capacity of $(i,j) \in A$ (>0) → upper bound

l_{ij} : lower bound of $(i,j) \in A$ (≥ 0).



By default, $l_{ij} = 0$, unless mentioned otherwise.

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

$> 0 \Rightarrow$ i is a supply node

$< 0 \Rightarrow$ i is a demand node → demand of $-b(i)$ at node i

$= 0 \Rightarrow$ i is a transshipment node

$b(i), c_{ij}, l_{ij}, u_{ij}$: data (known with certainty). These parameters do not depend on x_{ij}'s, which are variables.

Goal: Find flow x_{ij} $\forall (i,j) \in A$ such that bounds are satisfied, supply/demand is "met" at each node $i \in N$, and overall cost is minimum.

We assume $\sum_{i \in N} b(i) = 0$ by default, i.e., total supply = total demand.

Here is the optimization model for the min-cost flow problem:

$$\min \left\{ \sum_{(i,j) \in A} c_{ij} x_{ij} \right\} \quad \text{(1)}$$

(minimize; total cost)

subject to (s.t.)

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

(for all)

outflow − inflow = supply/demand

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

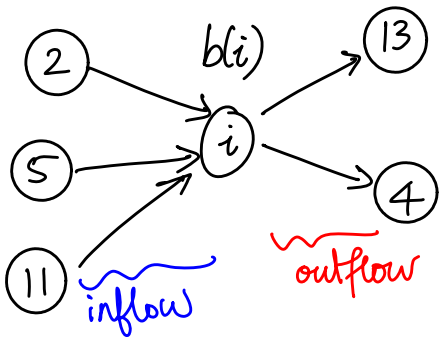
Mathematical notation: $\in$: "element of", $\forall$: "for all", $\sum$: sum.

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).

This is a linear optimization problem (or a linear program, LP). But we won't solve these instances the same way we solve LPs. These problems are simpler than the general LPs — the underlying network structure allows one to design efficient algorithms, which run faster than default algos to solve LPs!

(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., all x_{ij} values, that minimizes the total cost.

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



$x_{i,13} + x_{i,4}$ } outflow

$x_{2i} + x_{5i} + x_{11,i}$ } inflow

$b(i)$: supply/demand

outflow − inflow = supply/demand

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. If u_{ij} is not specified, it is taken as $+\infty$ (some large number in practice).

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.

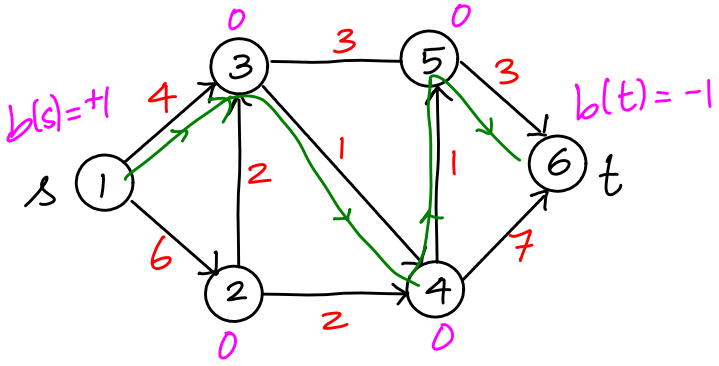
We now show that the shortest path and the max flow problems are special cases of the min cost flow problem.

(*) When "formulating" or "modeling" a problem as a network flow problem, you need to specify the network $G=(N,A)$, i.e., what is the node set, the arc set, and what are the associated parameters $(b(i), l_{ij}, u_{ij}, c_{ij})$. ($M$ some big >0 number in practice)

Default values: $b(i)=0$, $l_{ij}=0$, $c_{ij}=0$, $u_{ij}=+\infty$.

Shortest Path Problem as a min-cost flow Problem

Consider the example for shortest path from Lecture 1.

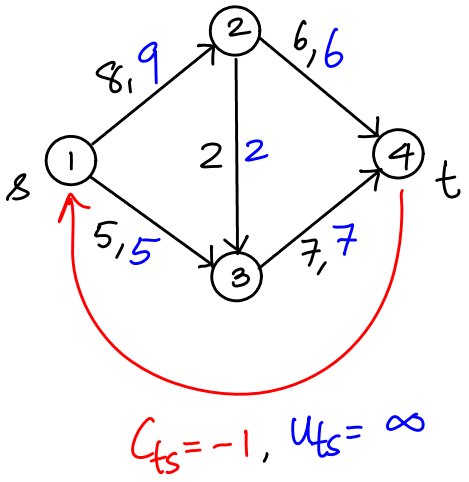


Use same network $G=(N,A)$ as in the shortest path problem. We set $b(s)=+1$, $b(t)=-1$, $b(i)=0$ $\forall i \neq s,t$, $i \in N$. Then we set $l_{ij}=0$, $u_{ij} \geq 1$ $\forall (i,j) \in A$.

Solving the MCF instance here is equivalent to sending one unit of flow from s to t at the least cost. Think of sending one car from the origin to destination. To detail the argument, we can show $x_{ij}=1$ or $x_{ij}=0$ here (there are some deeper assumptions critical here). Then argue that the $(i,j) \in A$ with $x_{ij}=1$ precisely gives the shortest path.

Max Flow as special case of MCF

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.



Add (t,s) to A, with $c_{ts}=-1$, $u_{ts}=+\infty$.

Set $l_{ij}=0 \quad \forall (i,j)\in A$ (including l_{ts})

$c_{ij}=0 \quad \forall (i,j)\in A \neq (t,s)$.

$b(i)=0 \quad \forall i \in N$.

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.

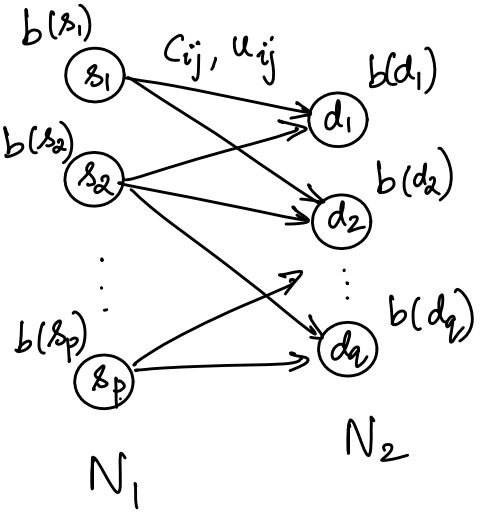
Notice that the flow just circulates around the network here. There is no flow coming in or going out at any of the nodes. This is an instance of the **circulation problem**, which is the min cost flow problem with $b(i)=0$ for all nodes i. This is the fourth network flow problem class (after shortest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

There are no transshipment nodes in the standard transportation problem. Indeed, if there were a node with zero supply within N_1, we could remove it without changing the main problem. Similar is the case with a zero demand node in N_2.

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 problem — the seat sharing problem — in detail in the next lecture. A handout on this problem is posted on the course web page.

Seat Sharing Problem

(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.

Let there p families, with b_i members for $1 \le i \le p$. And let there be q cars, with u_j seats, for $1 \le j \le q$. We start with a node for each family and a node for each car. Let $N_1 = \{f_1, \dots, f_p\}$ be the set of family nodes and $N_2 = \{c_1, \dots, c_q\}$ be the car nodes.

This situation appears to be close to a transportation problem setting. But there is a catch — each car has some set of seats that need not **all** be filled! In that sense, we cannot model the # seats of car c_j as its demand. We need to somehow model it as a capacity of the node.

More in the next class...

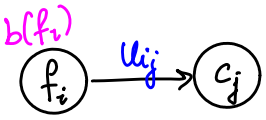
MATH 566: Lecture 3 (08/27/2024)

Today: * seat sharing problem
* assignment problem
* definitions

Seat Sharing Problem

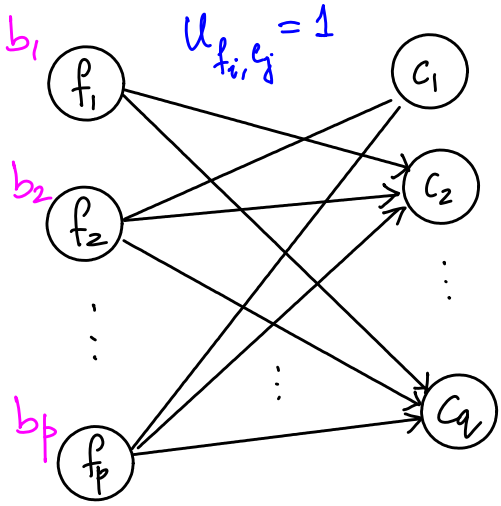
(AMO 1.8, page 21) Several families are planning a shared car trip on scenic drives in the Cascades in Washington. To minimize the possibility of any quarrels, the organizers of the trip want to assign individuals to cars so that **no two members of a family are in the same car**. Formulate this problem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

Let there p families, with b_i members for $1 \le i \le p$. And let there be q cars, with u_j seats, for $1 \le j \le q$. We start with a node for each family and a node for each car. Let $N_1 = \{f_1, \dots, f_p\}$ be the set of family nodes and $N_2 = \{c_1, \dots, c_q\}$ be the car nodes.



N_1 N_2

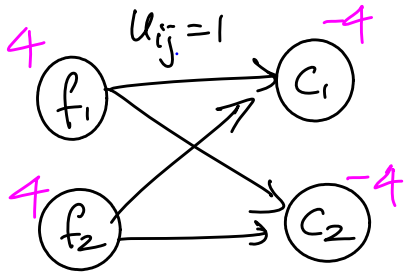
We set $b(f_i) = b_i$, i.e., the supply of family node f_i as b_i.



To capture the restriction that no two members of a family can be seated in one car, we add arcs from each f_i to each c_j, and set their capacities to 1.

Notice that we add an arc from each f_i to each c_j — as a member from any family could be sent to any car. We add a total of pq arcs, each with $u_{(f_i, c_j)} = 1$.

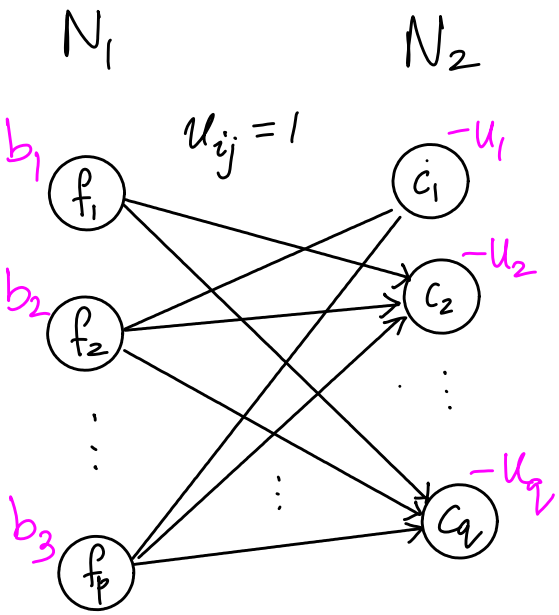
Assumption We assume there exists a feasible assignment of people to cars. There could be trivial instances where we cannot seat all members without avoiding clashes, e.g., see the network below:



Let the capacities of cars c_1 and c_2 be 4 each. We can set them as demands, but there is no feasible arrangement that avoids in-family clashes!

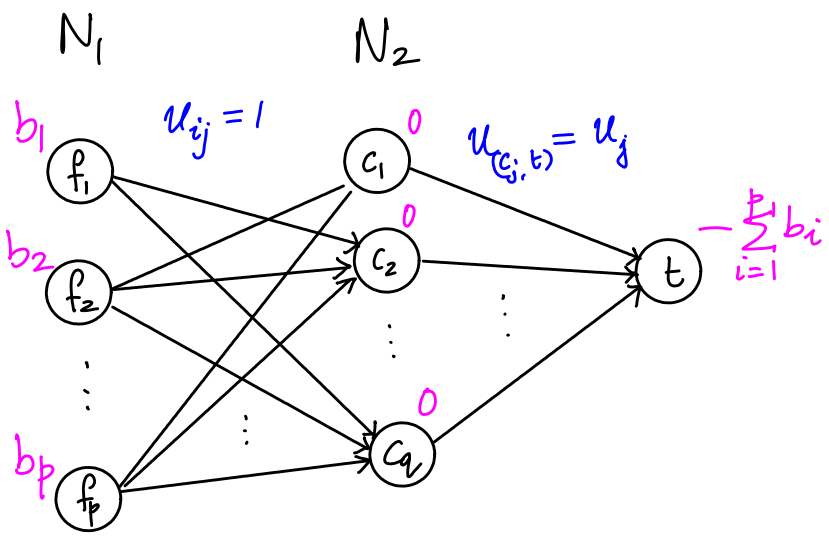
In fact, finding if a given network has a feasible flow can itself be modeled as another network flow problem! More on this topic in a bit...

Back to the problem now.



If we assume further that $\sum b_i = \sum u_j$, then we could set $b(c_j) = -u_j$, and we have a valid MCF problem (as a transportation feasibility problem, since no c_{ij}'s are used).

But more generally, there could be more total seats than there are people to be seated, i.e., $\sum u_j > \sum b_i$. Hence we consider the following more general model.

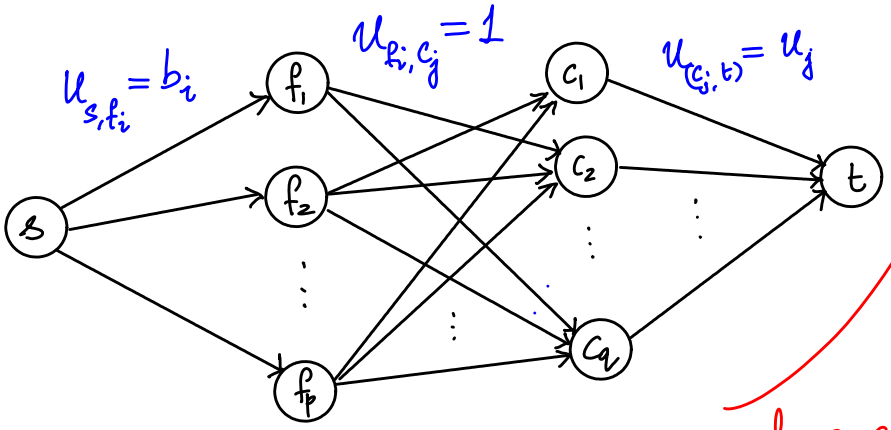


We add a terminus node t, and arcs (c_j, t) with $u_{(c_j,t)} = u_j$, the # seats available in car c_j. We set $b(j) = 0$ for all $j \in N_2$, i.e., car nodes are transshipment nodes. And we set $b(t) = -\sum_i b_i$.

By setting $b(t) = -\sum_{i \in N_1} b(i)$ (i.e., $-\sum_{\text{families}} (\#\text{ members})$), we ensure that all members from each family is assigned to some car.

This more general model allows one to accommodate some other variations/generalizations — see the problem in Homework 1.

The original problem could be modeled also as a max flow problem:



If the value of the max flow is $\sum_{f_i \in N_1} b_i$, then there is a feasible seating arrangement, and that is specified by the x_{f_i,c_j} values that are 1.

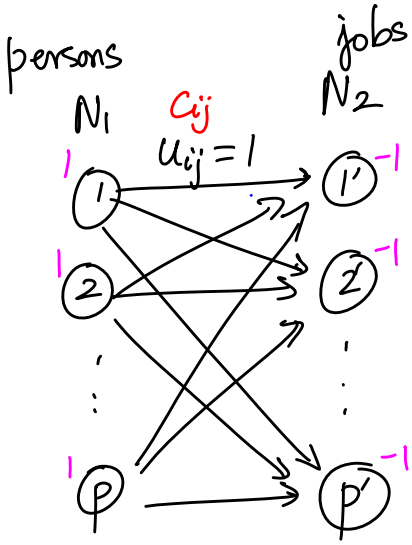
so, each arc (s, f_i) is saturated

We saw shortest path (SP), max-flow (MF), min-cost flow (MCF), circulation, and transportation problems. Here is one more class of problems.

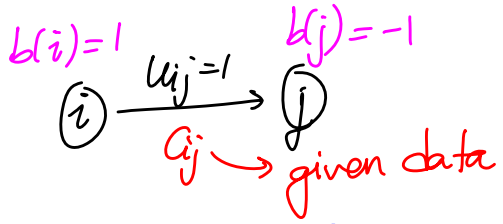
6. Assignment problem

Given p persons and p jobs, the goal is to assign each person to one job, such that the overall cost of the assignments is minimized. You are also given costs c_{ij} for assigning person i to job j.

We can model this problem as a case of MCF with node set $N = N_1 \cup N_2$. The network has a bipartite structure, similar to the case of the transportation problem.



$(i,j) \in A$ has $i \in N_1$ and $j' \in N_2$



We need not specify $u_{ij} = 1$ here! Could leave them as $u_{ij} = +\infty$, the default values, since $b(i) = 1 \ \forall i \in N_1$ and $b(j') = -1 \ \forall j' \in N_2$.

When formulating a network flow problem, you must describe the structure of the network, i.e., specify the node and arc sets N and A, and specify all associated parameters — $b(i) \ \forall i \in N$, $c_{ij}, l_{ij}, u_{ij} \ \forall (i,j) \in A$.

Default values: $b(i) = 0$, $c_{ij} = 0$, $l_{ij} = 0$, $u_{ij} = +\infty$.
If not specified, a parameter is assumed to be at its default value.

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) $\quad N=\{1,2,\dots,n\}$

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

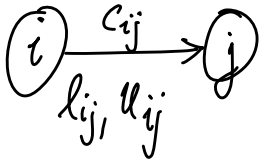
Directed network: edges or arcs of the network are directed

(i,j) 

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

We assume the network is directed by default.

node i: **tail**
node j: **head**



also, outarc — also, inarc

(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).

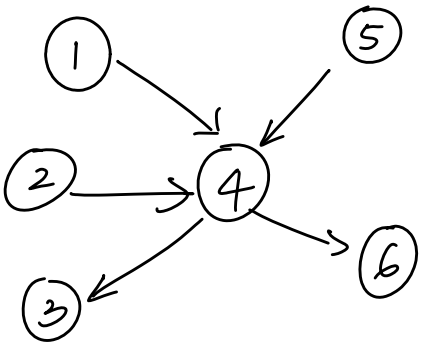
Degree

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 → perhaps the first data structure for networks!

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

Notice that $|A(i)|$ = outdegree of i, and

$$\sum_{i \in N} |A(i)| = m \quad \text{(total \# arcs)}.$$

With arcs numbered $1, 2, \ldots, m$, it is more efficient to store the corresponding arc numbers (or indices), instead of pairs (i,j).

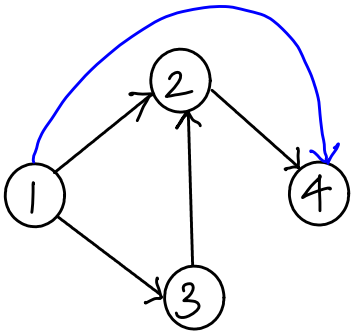
Note that $A(i)$ defines adjacency based on outarcs. We can equivalently define inarc lists $AI(i)$. In practice, it often helps to initialize and use both sets of lists.

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

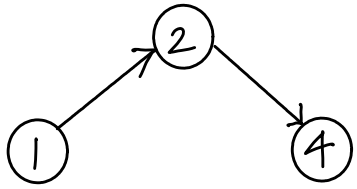
An **induced subgraph** is a subgraph that contains each arc of A with nodes in N', i.e., each such arc is present in A'.

A **spanning subgraph** has $N' = N$.

Example

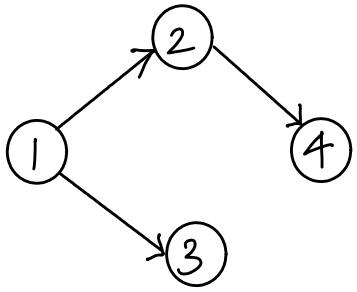


G



G' is an induced subgraph of G.

But is not induced if (1,4) were present



G'' is a spanning subgraph

MATH 566: Lecture 4 (08/29/2024)

Today: * More definitions
* Network representations

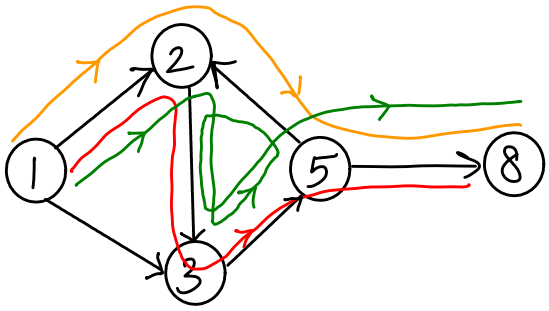
Definitions (contd..)

A **walk** in $G=(N,A)$ is a subgraph of G made of a sequence of nodes and arcs $i_1-a_1-i_2-a_2-\cdots a_{r-1}-i_r$ with $a_k=(i_k,i_{k+1})$ or $a_k=(i_{k+1},i_k)$ for $1\le k\le r-1$.

Thus, a walk need not necessarily be directed.

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

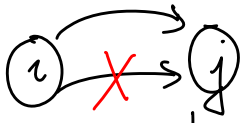
Here are some examples.



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

But 1-2-3-5-2-3-5-8 may not be considered a (directed) walk under this definition since arcs are repeated.

But some other book allows repeated arcs in walks. Also, look up "trail" and "circuit".

We assume there are no duplicate arcs: 

Notice that nodes could be repeated in a walk.

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

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

An 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.

For instance, in 1-2-5-8, $(1,2)$ and $(5,8)$ are forward arcs, while $(5,2)$ is a backward arc.

A directed path has no backward arcs.

Convention: A path with backward arcs called a **non**-directed path; an **un**directed path will be a path in an undirected graph.

If (i,j) is in a directed path, i is the **predecessor** of j, and we denote it by $\text{pred}(j) = i$.

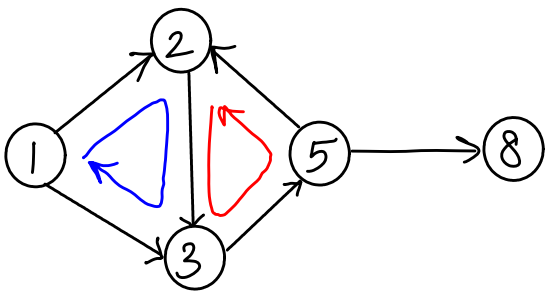
For the directed path 1-2-3-5-8, we set

$\text{pred}(8) = 5$
$\text{pred}(5) = 3$
$\text{pred}(3) = 2$
$\text{pred}(2) = 1$
$\text{pred}(1) = 0$ → marks the start of the directed path

We will use pred(·) indices as the standard data structure to represent paths – we will use a pred vector of length n.

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

A **directed cycle** is a directed path $i_1 - i_2 - \cdots - i_r$ with arc (i_r, i_1)



1-2-3-1 is a cycle
2-3-5-2 is a directed cycle.

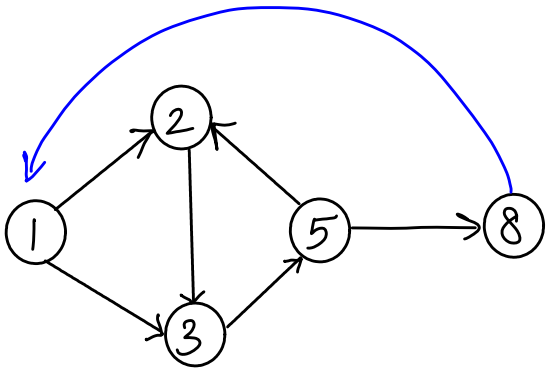
Many problems are more complicated when the network has directed cycles.

An **acyclic graph** is a graph with no directed cycles. So, non-directed cycles are permitted here. We get a tree if we avoid those cycles as well, assuming the graph is connected.

Connectivity

Nodes i and j are **connected** if there is a path from i to j. The (entire) graph is connected if every pair of vertices is connected. Else it is **disconnected**.

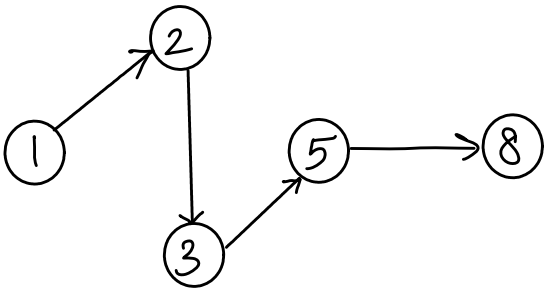
A graph is **strongly connected** if there is a directed path from every node i to every node j.



G is connected but not strongly connected. Adding $(8,1)$ makes it strongly connected.

A **tree** is a connected graph with no cycles. So, we avoid both directed and non-directed cycles in a tree.

e.g., T

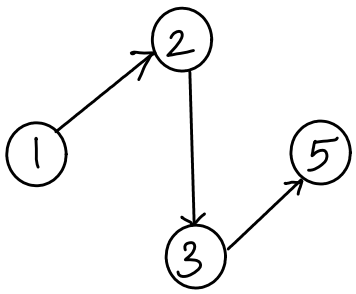


is a tree of the graph G introduced previously.

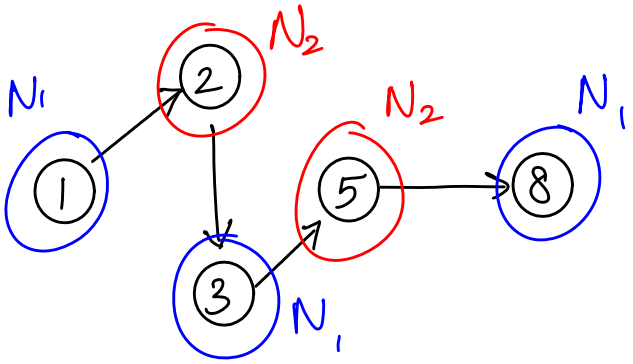
A tree T is a **spanning tree** of graph G if T is a spanning subgraph of G.

For instance, T' is not a spanning tree of G.

T'



Bipartite graph $G=(N,A)$ is **bipartite** if we can partition the node set N into subsets N_1 and N_2 such that $N = N_1 \cup N_2$, and each $(i,j) \in A$ has either $i \in N_1, j \in N_2$ or $i \in N_2, j \in N_1$.

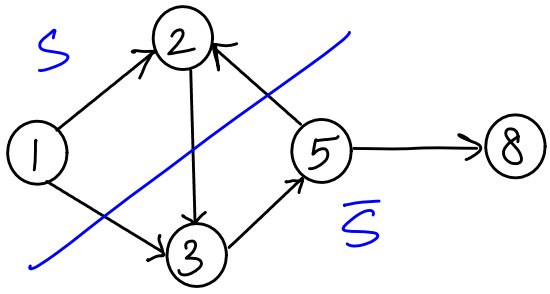


Note: A bipartite graph need not always be "drawn" in an obvious bipartite fashion, e.g., in the case of transportation problems.

In fact, one could prove that every tree is a bipartite graph — think about it!

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

Each cut defines a set of arcs $(i,j) \in A$ with $i \in S, j \in \bar{S}$ or $i \in \bar{S}, j \in S$.
forward arcs; backward arcs



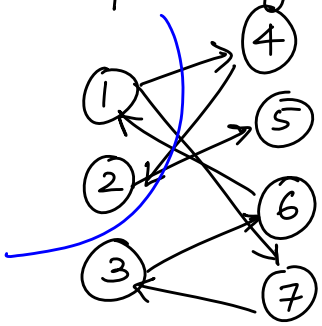
$S = \{1,2\}$, $\bar{S} = \{3,5,8\}$

$(1,3), (2,3)$: forward arcs

$(5,2)$: backward arc

Notice that we could talk about cuts for a bipartite graph just the same as in a general graph.

We will talk more about cuts when we introduce max flow in detail.

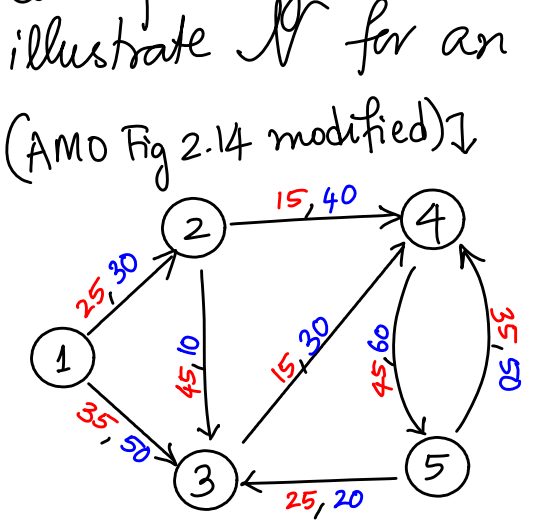


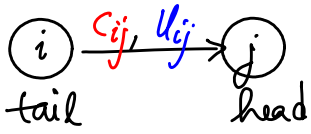
Network Representations

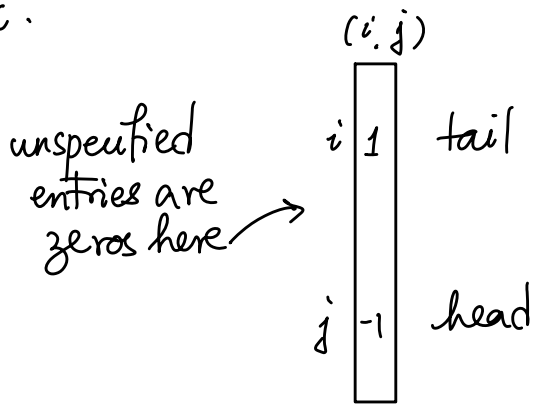
We need to represent $G = (N, A)$, along with data, i.e., $b(i)$, l_{ij}, u_{ij}, c_{ij}, etc. The ultimate goal is to be able to access and use all this info efficiently in algorithms. We consider several options for representing networks now.

Node-arc incidence matrix N

N is an $n \times m$ matrix with one row for each node, and one column for each arc, with entries in $\{-1, 0, 1\}$. The column of N corresponding to arc (i, j) is shown to the right. We also illustrate N for an example network.







$$N = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ (1,2) & (1,3) & (2,3) & (2,4) & (3,4) & (4,5) & (5,3) & (5,4) \end{array} \\ \begin{bmatrix} 1 & 1 & & & & & & \\ -1 & & & 1 & 1 & & & \\ & -1 & -1 & & & & -1 & \\ & & & & -1 & -1 & 1 & & -1 \\ & & & & & & -1 & 1 & 1 \end{bmatrix} \end{array}$$

↑ unspecified entries are zeros.

Lexicographic or dictionary ordering of arcs. This will be the default way in which we order arcs

AMO uses c, u, etc to denote these vectors

$\bar{c} = [25 \quad 35 \quad 45 \quad 15 \quad 15 \quad 45 \quad 25 \quad 35]$

$\bar{u} = [30 \quad 50 \quad 10 \quad 40 \quad 30 \quad 60 \quad 20 \quad 50]$

$\bar{b}$ → a 5-vector of supply/demand values.

My notation: lower case letters with a bar are used to denote vectors, e.g., $\bar{x}$, $\bar{c}$, $\bar{u}$, $\bar{b}$, etc.

Properties of $\mathcal{N}$

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

At the same time, $\mathcal{N}$ 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

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

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

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

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

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

$$\bar{x} = \begin{bmatrix} \vdots \\ x_{ij} \\ \vdots \end{bmatrix}$$

$$\min \quad \bar{c}^T \bar{x} \qquad (T \rightarrow \text{transpose})$$

$$\text{s.t.} \quad \mathcal{N}\bar{x} = \bar{b} \qquad (\bar{b} \rightarrow \text{vector of } b(i)\text{'s})$$

$$\bar{l} \le \bar{x} \le \bar{u}$$

vectors are columns by default, hence the transpose here.

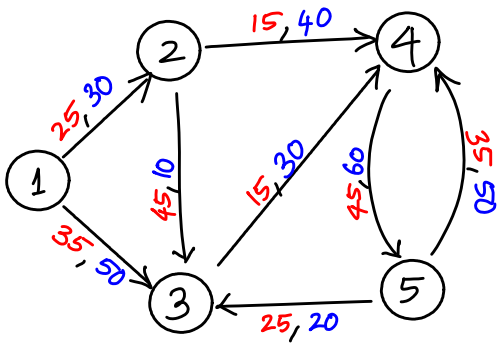
We now introduce another matrix representing the network.

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

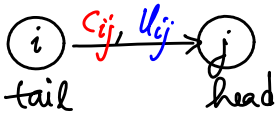
This is an $n \times n$ matrix $\mathcal{H} = [h_{ij}]$, with

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

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



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



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 j = indegree (j)

  } scan row i to get outarcs of node i; scan column j to get inarcs of node j.

* $\mathcal{H}$ has n^2 elements, m of which are nonzero.
  So, $\mathcal{H}$ is an efficient data structure when $m \gg n$. ("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 Octave/Matlab. But we would ideally like to avoid the overhead associated with such operations by using more efficient representations to start with.

We will talk about efficient ways of representing such lists in general, and (out)arc lists in particular, in the next lecture.

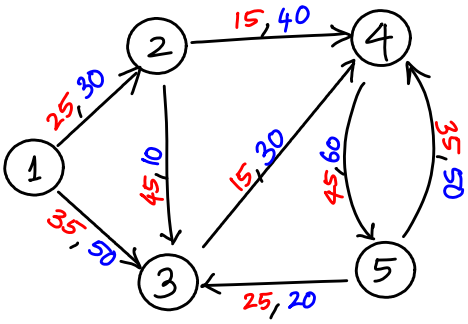
MATH 566: Lecture 5 (09/03/2024)

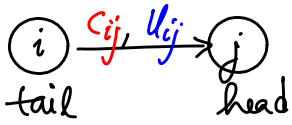
Today: * forward star representation
* network transformations

Forward Star Representation

A list of info about arcs, set as a table. There is one row for each arc $(i,j) \in A$, and 3–5 columns listing the tail (i), head (j), and c_{ij}, l_{ij}, and u_{ij} values.

Illustration





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

Assume $l_{ij} = 0 \quad \forall (i,j) \in A$

The main matrix is $m \times 3$, or $m \times 4$ or $m \times 5$, depending on which of the parameters c_{ij}, l_{ij}, and u_{ij} are given. The $b(i)$ values are stored in a separate n-vector.

Notice that the information is stored in the order (lexicographic 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, or easily extractable.

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

point(i)

1	1
2	3
3	5
4	6
5	7
(n+1) → 6	9 ← m+1 (#arcs+1)

point(n+1) = m+1 (always)

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

The outarcs of node i are in rows indexed point(i) to point(i+1)-1, for $1 \le i \le n$.

For instance, for node 2, point(2) = 3, point(2+1) = point(3) = 5. Hence, the outarcs of node 2 are in rows 3 to 5-1, i.e., rows 3 and 4.

The matrix (mx3, mx4, ...) of arcs list along with the point vector (and b(·) vector) is the **forward star representation** of G.

Note: If node i has no outarcs, we set point(i) = point(i+1), so that the indices of outarcs of node i is an empty set.

Here is a quick example: we reverse arc (3,4) in the example network. Rest of the network remains unchanged. Note that node 3 now has no outarcs. The corresponding point(·) vector is shown to the right.

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

point(i)

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

Notice how the outarcs of node 2 are still recorded correctly: arcs point(2) = 3 to point(3) − 1 = 4.

Equivalently, we could have a matrix of inarcs along with the associated parameters (c_{ij}, u_{ij}, l_{ij}), and create the **rpoint** vector, which is the **reverse point** vector. The in-arcs of node i are precisely the arcs indexed rpoint(i) to rpoint(i+1) − 1.

For small or moderately sized networks, it may be simpler to initialize the A{·} lists as a cell array (outarc list). For large networks, the use of point is recommended.

Here are some basic Matlab commands and files

netdata.m

```
%% Math 566 (Fall 2024)
%% Matlab code to extract out- and in-arc lists
%% from the forward star representation

%T: TAIL,  H: HEAD
%DEGO: outdegree,  DEGI: indegree
%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

```
%% Math 566 (Fall 2024)
%%
%% Network from Lecture 4, which is a slighty modified
%% version of the network in AMO Figure 2.13, page 32.

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

% The first column is just the arc index (or number)
% It is redundant info, and is listed her just for
% the sake of convenience.

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

netdata

DEGO
celldisp(A)
```

Output from Matlab:

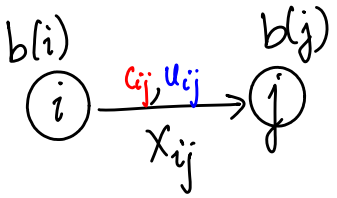
```
% Matlab session
% Lecture 5, Sep 03, 2024

>> example1
DEGO =
    2    2    1    1    2
A{1} =
    1    2
A{2} =
    3    4
A{3} =
    5
A{4} =
    6
A{5} =
    7    8
DEGI =
    0    1    3    3    1
AI{1} =
    []
AI{2} =
    1
AI{3} =
    2    3    7
AI{4} =
    4    5    8
AI{5} =
    6
>>
```

We will implement several algorithms we discuss in class in Matlab.

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

(We typically take $l_{ij} = 0$. But if $l_{ij} > 0$, we could transform the problem to an equivalent problem with $l_{ij} = 0$).

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

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

x'_{ij}: new flow; u'_{ij}: new u_{ij}

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

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

$u'_{ij} = u_{ij} - l_{ij}$.

The flow balance constraint for node i becomes

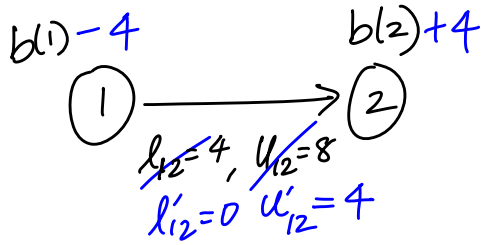
$$\sum_{(i,j)} (x'_{ij} + l_{ij}) - \sum_{(j,i)} (x'_{ji} + l_{ji}) = b(i)$$

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

$b'(i)$: new $b(i)$ value

The $b(i)$ values are updated as follows. We **subtract** the l_{ij} values for all **outarcs** of node i, and **add** the l_{ji} values for all **inarcs** of node i.

Here is an illustration on a single arc:
If we have to send at least 4 units along $(1,2)$, we could imagine that much flow being "taken out of" $b(1)$, and being added to $b(2)$.



What about the objective function?

$$\sum_{(i,j)} c_{ij} x_{ij} = \sum_{(i,j)} c_{ij} (x'_{ij} + l_{ij}) = \sum_{(i,j)} c_{ij} x'_{ij} + \sum_{(i,j)} c_{ij} l_{ij}$$

constant; does not depend on x_{ij} values.

Adding a constant to the objective function does not change the optimal solution. Hence we will indeed find the optimal solution to the original problem.

Once you have the optimal solution x'_{ij}, we can recover the corresponding optimal solution x_{ij} by computing $x'_{ij} + l_{ij}$.

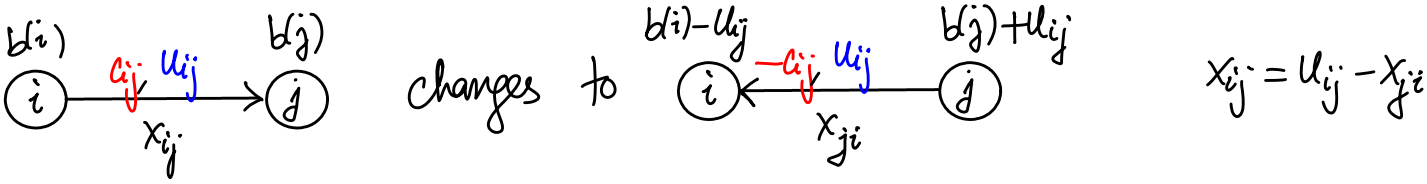
MATH 566: Lecture 6 (09/05/2024)

Today: * Network transformations
- arc reversal, removing upper bounds, node splitting
* complexity of algorithms

---

(2) Arc reversal

Can be used to handle negative c_{ij}'s.



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

Logic: First send u_{ij} from i to j to saturate (i,j). Modify $b(i)$ and $b(j)$ to reflect this change. Then we pull back x_{ji} units from j to i. Since u_{ij} is constant, the only variable contribution to the cost will be from x_{ji}, which flows in the opposite direction.

Contribution to total cost: $c_{ij} x_{ij}$

$$= c_{ij}(u_{ij} - x_{ji}) = c_{ij} u_{ij} - c_{ij} x_{ji}$$

($c_{ji} = -c_{ij}$)

$$= \text{constant} + c_{ji} x_{ji}$$

By setting $c_{ji} = -c_{ij}$, we get a positive cost when $c_{ij} < 0$.

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

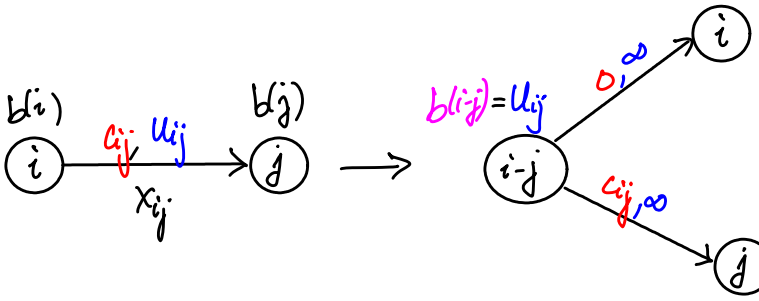
Q: What if all c_{ij} are < 0 to start with? Could we add a constant value to all c_{ij}'s so that they all become ≥ 0?

This transformation is a bit tricky! Think about it  ?!

③ Removing upper bounds

($l_{ij} = 0 \ \forall (i,j)$ is assumed)

Want all $u_{ij} = +\infty$.



We present a complementary but equivalent formulation to that given in AMO.

Intuitively, each node i has a flow balance equation with $b(i)$ in the rhs. Arc (i,j) has an upper bound that appears as $\leq u_{ij}$. Hence to remove this bound, we equivalently add an extra node $i\text{-}j$ ("i-hyphen-j") and set $b(i\text{-}j) = u_{ij}$.

Let's concentrate on x_{ij} alone in the model:

$$x_{ij} = b(i) \quad \text{——— (1)}$$

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

$$x_{ij} \leq u_{ij}$$

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

(s_{ij}: "slack")

But now, s_{ij} appears only once in system (1), (2), (3), and x_{ij} appears 3 times. We want each flow appearing twice, with +1 and -1.

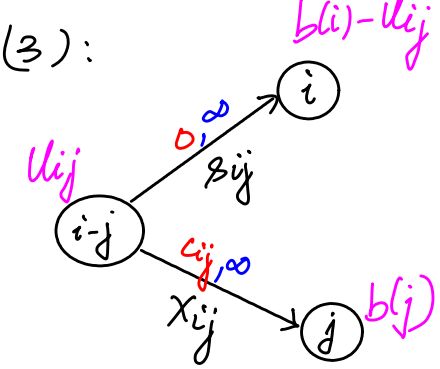
So we do (1) − (3) to get (1′): $-s_{ij} = b(i) - u_{ij}$ ——— (1′)

We consider the equivalent system (1′), (2), (3):

$$-s_{ij} = b(i) - u_{ij} \quad \text{——— (1′)}$$

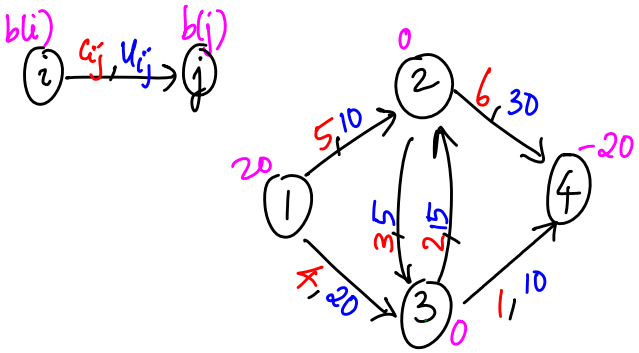
$$-x_{ij} = b(j) \quad \text{——— (2)}$$

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

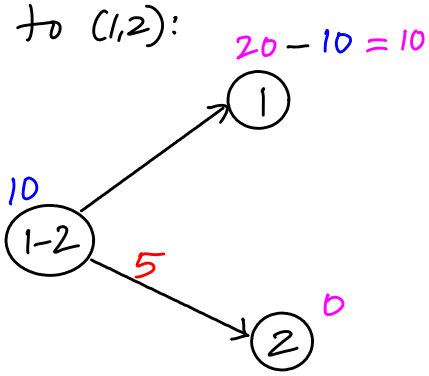


x_{ij} and s_{ij} are outflows from a new node with $b(\cdot) = u_{ij}$; and they flow into j and i, respectively.

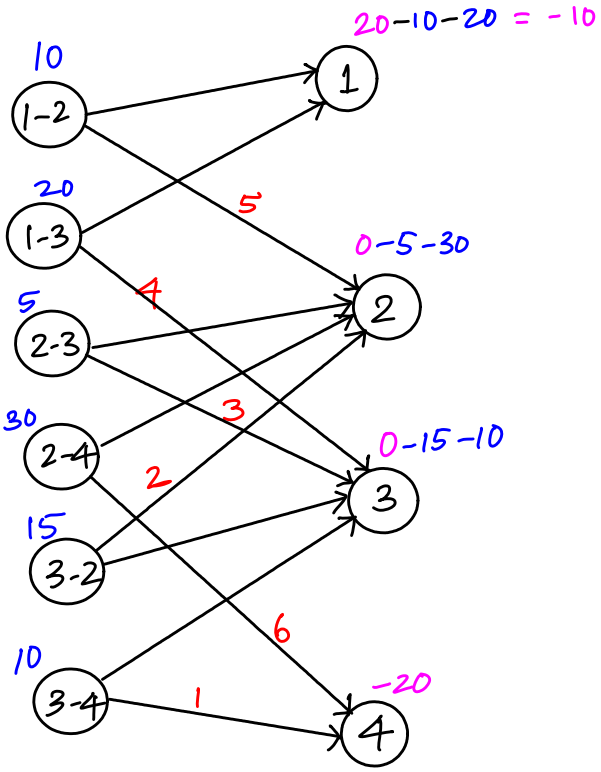
We could apply this transformation on multiple arcs together. Consider the following network:



Here is the transformation applied to (1,2):



We aggregate the transformations to each arc to get the final network. Unmentioned costs are 0 (and, of course, all upper bounds are $+\infty$).



Here are some points to note:

* The overall change to b(i) is

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

← all outarcs of node i

* 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. This result holds in general as long as all original u_{ij} are finite to start with.

* The flow $x_{i-j,j}$ in the new network is equal to the flow x_{ij} in the original network. Further notice that the contribution to total cost from flow on (i,j) in the original network is captured still as $c_{ij} x_{ij}$ in the new network.

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.

At the same time, we might not apply this transformation if directed paths from i to j are of critical interest — notice that there is no directed path from any node $i \in N_2$ to any other node $j \in N_2$ in the new network.

4. Node splitting

Split node i into two: i'', i', and add arc (i'', i').

Replace

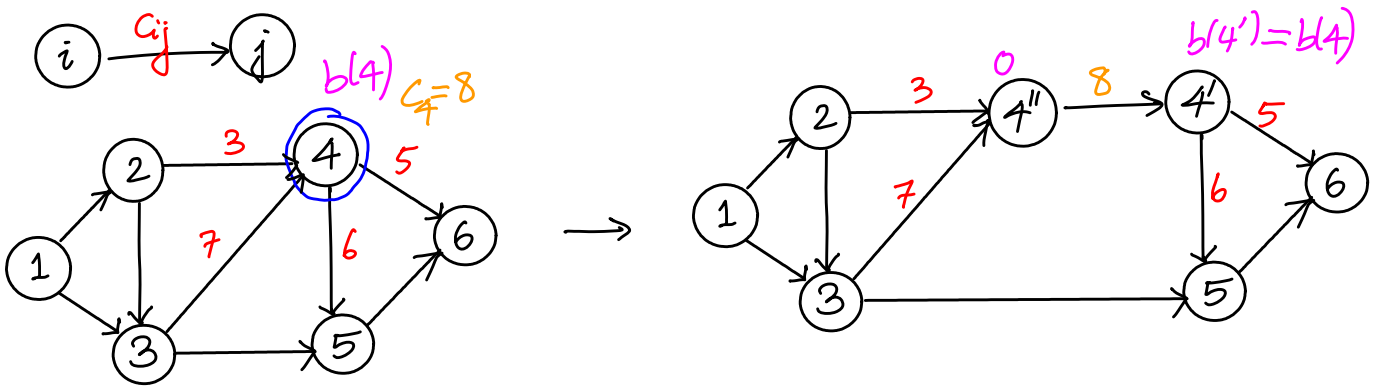
Steps Replace node i with two nodes i'' and i', add arc (i'', i').

Replace arc (j, i) with (j, i'') → inarc of node i

Replace arc (i, k) with (i', k) → outarc of node i

For the arc replacements, we transfer the arc parameters (cost, capacities) to the new arcs unchanged.

We illustrate the transformation on a small example. Say node 4 has a unit cost $c_4 = 8$ (for flow through the node). We split node 4 as shown below.



We assume that $b(4)$ does not incur cost c_4. If it were that case, we set $b(4'') = b(4)$ and $b(4') = 0$.

Intuition: Consider traffic flow through a road network modeled as a directed 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. Similarly, there might be a limit on how much traffic could go through downtown.

We summarize the various classes of network flow problems as well as network transformations we have seen so far.

Problem classes

* shortest path
* max flow
* min-cost flow (most general)
* circulation
* transportation
* assignment

We will see minimum spanning trees as a separate class of problems later on. Similarly, matching (in general setting) is a different problem class.

Transformations

* adding nodes/arcs
* removing lower bounds
* arc reversal
* removing upper bounds
* node splitting

All transformations were defined for the min-cost flow problem.

Computational Complexity (for dummies)

Q. How do we measure efficiency of an algorithm?

We do "worst case analysis", i.e., analyze 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 multiplier. Further, our goal is to study how the algorithm performs as the size of the problem grows. As such, we want to compare the performance of the algorithm on different instances on the same machine, i.e., we do not want to be confused by the relative efficiencies of different machines.

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

(more in the next lecture...)

MATH 566: Lecture 7 (09/10/2024)

Today: * Computational complexity
* Search algorithms – generic search

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

→ could be $f(n,m,p,\dots)$ for multiple dimensions

We can define the notion of $O(\cdot)$ for functions:

Def A function $f(n)$ is $O(g(n))$ if there exist numbers $c>0$ and $n_0>0$ such that $f(n) \leq c\,g(n) \quad \forall\, n \geq n_0$.

→ Formally, $O(g(n))$ is the set of all such functions. Hence we say $f(n)$ is **in** $O(g(n))$.

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

$$O(100n + n^2 + 0.00001 n^3) = O(n^3).$$

$O(\cdot)$ gives a convenient way to ignore coefficients and lower order terms in polynomial expressions.

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

For instance, let $0 \leq A_{ij} < 2^{51}$; then each A_{ij} takes upto 51 bits.

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

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

→ for adding two matrices.

Big Ω and big Θ notations

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

Def An algorithm is said to run in $\Omega(f(n))$ time if for some numbers c' (>0, real) and $n_0'>0$, 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 (for $n \geq n_0$)

$\Omega(f(n))$: **some** instance takes at least $c'f(n)$ time (for $n \geq n_0'$).

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

Note that the corresponding definition(s) for functions are a bit different.

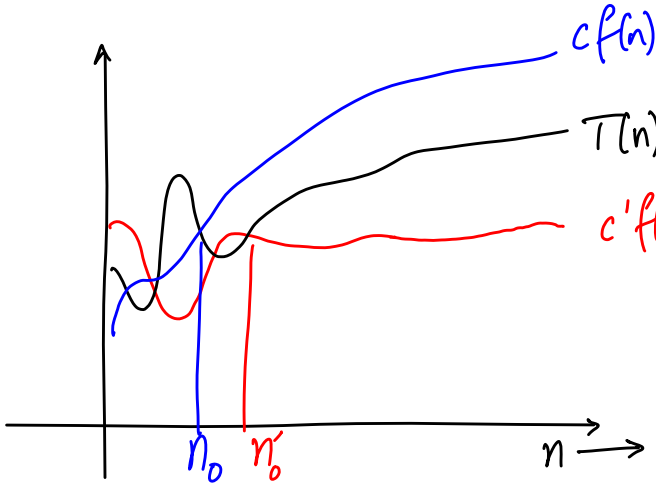
Def A function $f(n)$ is $\Omega(g(n))$ if there exist numbers $c'>0$ and $n_0'>0$ such that $f(n) \geq c'g(n)$ $\forall$ $n \geq n_0'$.

Def A function $f(n)$ is $\Theta(g(n))$ if there exist numbers $c, c'>0$ and $n_0>0$ such that $c'g(n) \leq f(n) \leq cg(n)$ $\forall$ $n \geq n_0$.

↳ can start with n_0, n_0' and take $\max(n_0, n_0')$ here.

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

{One n_0 is enough; we can use the larger of n_0 and n_0'}

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** where the worst-case complexity is bounded by a polynomial function of problem parameters $n, m, \log C, \log U$, where $C = \max_{(i,j)\in A} |C_{ij}|$, $U = \max U_{ij}$.

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

Strongly polynomial time algorithm: worst case complexity is bounded by a polynomial in only m and n, i.e., it is independent of $\log C, \log U$.

e.g., $O(m^2 n)$.

(strongly poly-time algos $\subset$ poly-time algos)

Exponential time algorithm: running time is $O(f(\cdot))$ where $f(\cdot)$ is exponential in $m, n, \log C, \log U$ → recall: size of problem varies as log of c_{ij}, etc.

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

e.g., an exact algorithm for the traveling salesman problem (TSP).

Pseudopolynomial time algorithm: running time is bounded by a polynomial in m, n, C, U. → and **not** $\log C, \log U$.

e.g., $O(m + nC)$. (pseudo poly-time algos $\subset$ exponential time algos)

Search Algorithms

Goal: find all nodes in G that satisfy a property.

e.g., 1. find all nodes in $G = (N, A)$ that are reachable by directed paths from node s;

2. find all nodes that can reach node t via directed paths.

We will discuss
- *generic search
- * breadth-first search (BFS)
- * depth-first search (DFS)

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

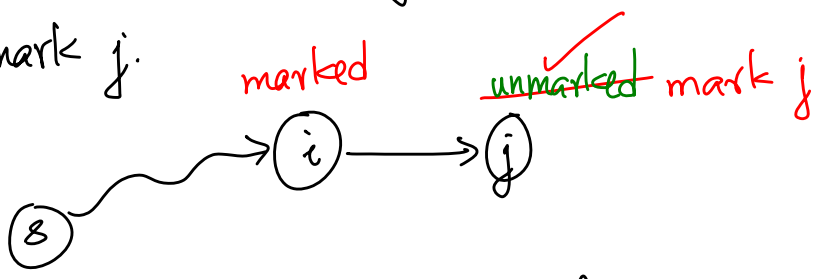
OUTPUT: $\{j \in N \mid$ a directed path exists in G from s to $j\}$.

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

marked → is reachable from s

unmarked → status of reachability from s 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 (i,j) where i is marked and j is unmarked is said to be **admissible**.

We use **pred(i)** indices to store directed paths from s to i, and the order of traversal of the nodes in **order(i)**.

Here is the pseudocode from AMO:

```
algorithm search;
begin
    unmark all nodes in N;
    mark node s;
    pred(s) := 0;
    next := 1;
    order(s) := next;
    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

maintain a binary n-vector (array)

initialization

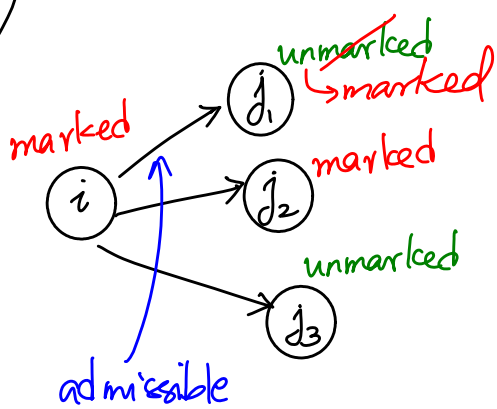
counter

order(s)=s; works only for s=1.

order(s)=2 ⇒ node 5 is the 2nd node visited

look at A(i) list

→ Run through A(i) list. The **current arc** (i,j) is the next candidate arc from A(i).



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

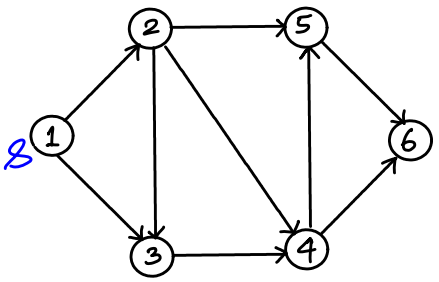
If we maintain LIST as a queue, i.e., select node i from the front, add node j to back of LIST we get **breadth-first search** (BFS). The nodes are examined in a **first-in first-out** (FIFO) order.

MATH 566: Lecture 8 (09/12/2024)

Today: * BFS, DFS examples
* complexity of search
* topological ordering

Example

Slightly modified version of AMO Fig 3.5.



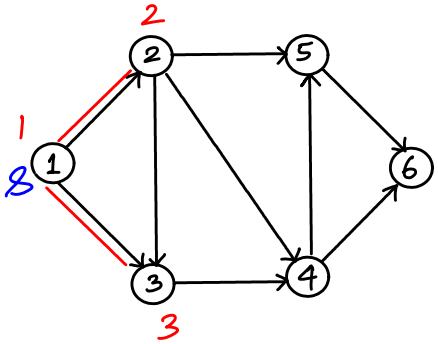
order: i

Outarc lists

$A(1) = \{(1,2), (1,3)\}$
$A(2) = \{(2,3), (2,4), (2,5)\}$
$A(3) = \{(3,4)\}$
$A(4) = \{(4,5), (4,6)\}$
$A(5) = \{(5,6)\}$
$A(6) = \emptyset$

Initialization

Mark = [1 2 3 4 5 6
1 0 0 0 0 0]
Pred = [M M M M M M]
Order = [1 0 0 0 0 0]
LIST = ∅ → [s]
next = 1
Mark(s) = 1 ← mark s
Pred(s) = 0



the search tree is shown in red (after Iteration 2 here)

Iteration 1

i=1.
Look at (i,j) = (1,2) ← current arc
it is admissible as Mark(i)=1 & Mark(j)=0.

Mark = [1 1 0 0 0 0]
Pred = [0 1 M M M M]
Order = [1 2 0 0 0 0]
next → 2
LIST = [1 2]

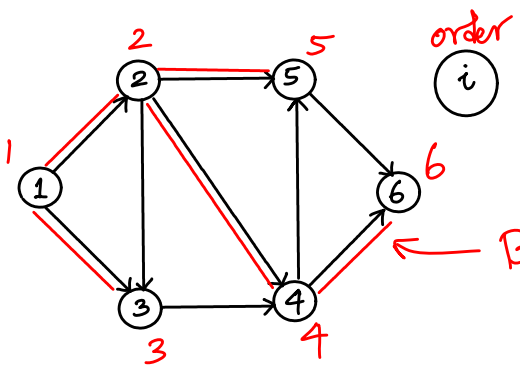
in general, we set pred(j) = i;

Iteration 2

i=1 → taken from front of LIST
(1,2) is now inadmissible, so move on.
current arc ← (1,3) → admissible

Mark = [1 1 1 0 0 0]
Pred = [0 1 1 M M M]
Order = [1 2 3 0 0 0]
next → 3
LIST = [1 2 3]

BFS Example (continued)



Iteration: 0 1 2 3 4 5 6 7 8 9 10 11 (initialization)

(Changes made in each iteration are color-coded accordingly)

Mark = [∅ ∅ ∅ ∅ ∅ ∅]

pred = [M M M M M M]

order = [∅ ∅ ∅ ∅ ∅ ∅]

LIST = [1] → [1 2] → [2 3] → [2 3] → [2 3 4] → [3 4 5] → [3 4 5] → [4 5] → [4 5 6] → [5 6] → [6]

Iterations 9, 10, 11: delete nodes 4, 5, 6

We get a search tree, called the breadth-first search (BFS) tree.

We could stop the algorithm once we have marked all nodes here. But, of course, all nodes may not be reachable from an s in every instance. For completeness, we do want to run the steps till LIST is indeed empty.

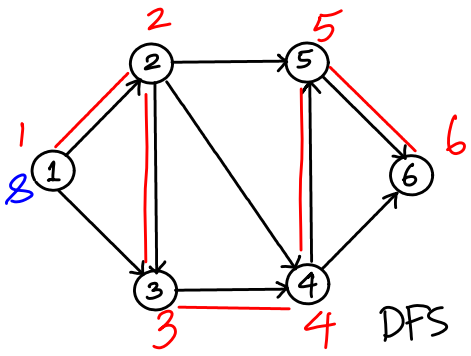
Property In the BFS tree, the directed path from s to i contains the smallest number of arcs among all directed s-i paths, i.e., it is a shortest path in terms of # arcs.

→ the sp here may not be unique, but # arcs is minimum

Depth-First Search (DFS)

Maintain LIST as a stack, i.e., in a **last-in first out** (LIFO) order.

Here is how DFS will proceed on the same instance:



LIST: [1] → [2 1] → [3 2 1] → [4 3 2 1] → [5 4 3 2 1] → [6 5 4 3 2 1] → ∅

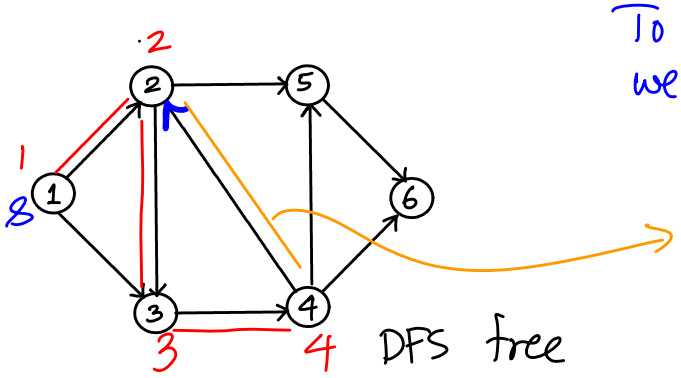
delele node in this order

Note that while the DFS tree is different from the BFS tree, order vectors turn out to be the same. But that is just a coincidence on this small network!

For instance, the search path to node 6 in DFS has 5 arcs, while the corresponding path in BFS is only 3 arcs long.

DFS identifies directed cycles quickly.

To include a directed cycle (for illustration), we replace (2,4) with (4,2).



The first inadmissible arc in DFS identifies a directed cycle. BFS could identify the directed cycle as well, but it could take longer than DFS.

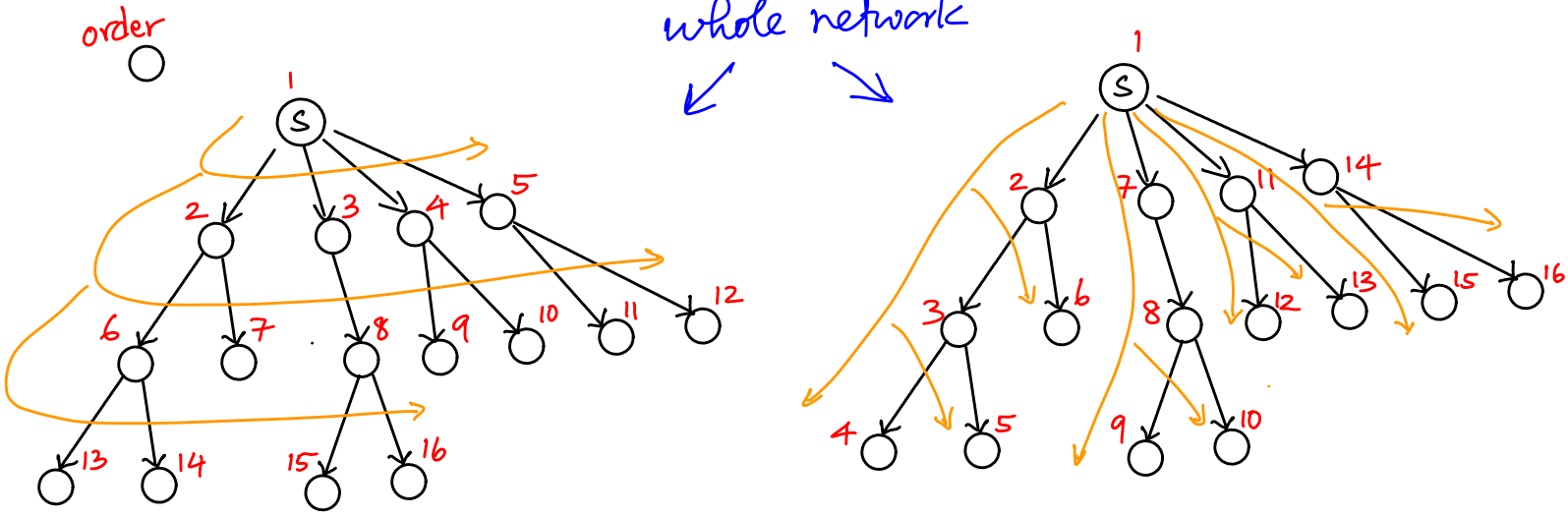
Q. How do BFS and DFS compare on large networks in general?

Here is a comparison of how these two searches visit nodes in a typical network. Notice how DFS dives deep fast.

BFS tree

only the search trees are shown here — not the whole network

DFS tree



BFS spreads at each "level" (or depth, in terms of # arcs from s) completely before descending to next level. DFS, on the other hand, dives to the bottom-most level along each "branch" before spreading wide to the next branch.

Complexity (Running time) of (Generic) Search algorithm

```
algorithm search;
begin
    unmark all nodes in N;
    mark node s;
    pred(s) := 0;
    next := 1;
    order(s) := next;
    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 ops

Within the **while** loop, a node gets added and deleted from LIST once, giving $2n$ operations, i.e, $O(n)$ time.

We examine, in the worst case, $\sum_{i \in N} |A(i)| = m$ arcs for admissibility, taking $O(m)$ time.

$\Rightarrow$ The overall running time is $O(m+n) = O(m)$, as $m \geq n$ typically.

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.

Notice that we do not have to look at all arcs repeatedly within the while loop. Indeed, each arc has to be examined for admissibility only once. If an arc becomes inadmissible at some point, it will never become admissible again. We could hence just run through $A(i)$ for each node i, examining the arcs just once each.

In practice, we can examine all outarcs of current node i in a unified manner, identify the admissible arcs from among them, and mark their head nodes in a unified manner as well.

We now look at a variation and an application of search.

Reverse Search Find all nodes from which one can reach a node t.
(reach → via directed path)

In generic search, start with LIST = [t] (instead of [s]), and examine AI(·) lists. From node j, (i,j) is **admissible** if j is marked and i is unmarked. Rest of the search process applies in this case as well.

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

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

$\Rightarrow$ Strong connectivity can be tested in $O(m)$ time.
(Two searches from any one node $\Rightarrow O(2m) = O(m)$ time.)

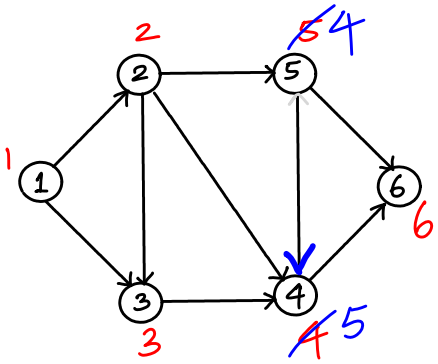
If is indeed sufficient to do this paired search from a single node (s). For any node pair i,j, we could go from i to s and s to j.

Topological Ordering

We use order(i) as labels for nodes. It is often desirable to assign order(·) such that every arc goes from a lower order node to a higher order node.

Def A labeling order(·) is a **topological ordering** if $\forall (i,j) \in A$, we have order(i) < order(j).

Consider the example we used to illustrate BFS and DFS:



The ordering we got for BFS is a topological ordering here.

But if we had (5,4) instead of (4,5) the ordering is not topological. But if we swap order(4) and order(5) now, we again get a topological ordering.

MATH 566: Lecture 9 (09/17/2024)

Today: * algo for topological ordering
* flow decomposition

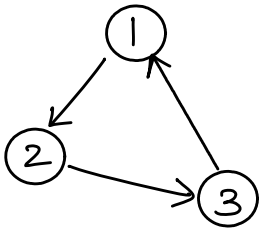
Topological Ordering

Recall the definition from the last lecture...

Def A labeling order(·) is a **topological ordering** if $\forall (i,j) \in A$, we have order(i) < order(j).

We now consider the problem of constructing a topological ordering for a given network, if possible, or certify that a topological ordering does not exist. From that point of view, can we characterize when a network is guaranteed to have a topological ordering?

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



No topological ordering is possible

But if G has no directed cycles, we can always find a topological ordering.

In fact, we can prove the following stronger result:

network is acyclic $\iff$ 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$(j) \ \forall (i,j) \in A$.

Idea for algorithm Start with nodes that have indegree $= 0$. Assign order $= 1$. Delete these nodes and all arcs going out of these nodes. Adjust node indegrees. Assign order $= 2$ for all nodes with indegree $= 0$ now. Repeat till network is empty.

In practice, we do not actually delete the nodes and arcs — just keeping track of indegrees (and decreasing them as we proceed) will be sufficient.

Pseudocode for topological ordering (from AMO):

```
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;
```

we assume indegrees are not known

counter

different from search!

≡ "deleting" (i, j)

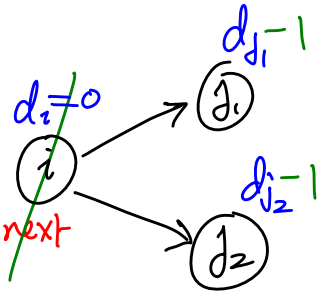


Figure 3.8 Topological ordering algorithm.

Note the similarities to the generic search algorithm. But also note that each node selected from LIST is immediately removed/deleted from LIST here.

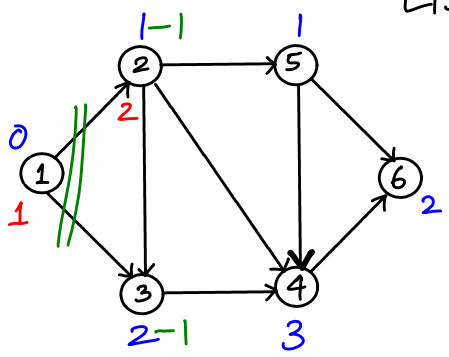
Using similar arguments as used for search, we can see that topological ordering runs in $O(m)$ time.

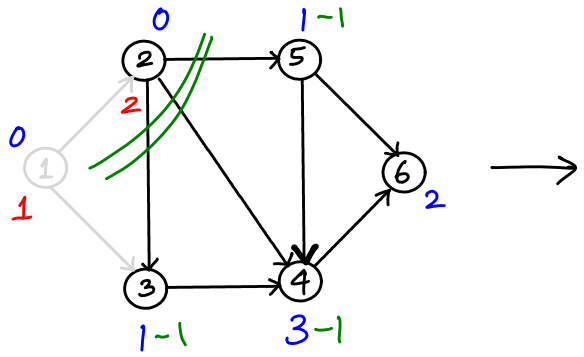
Illustration

We show the steps of topological ordering on a slightly modified network from the one we have seen previously — we include (5,4) in place of (4,5), hence the BFS order(·) is not a topological ordering.

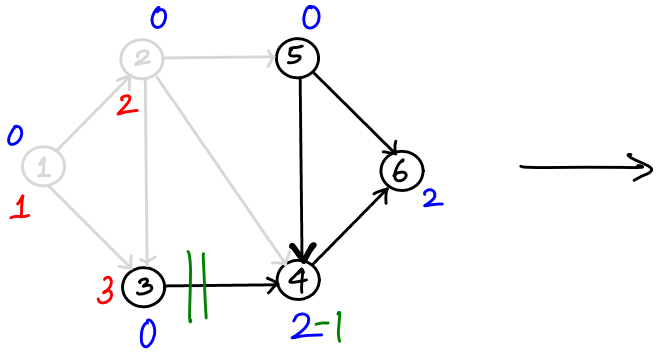
indegree
i
order

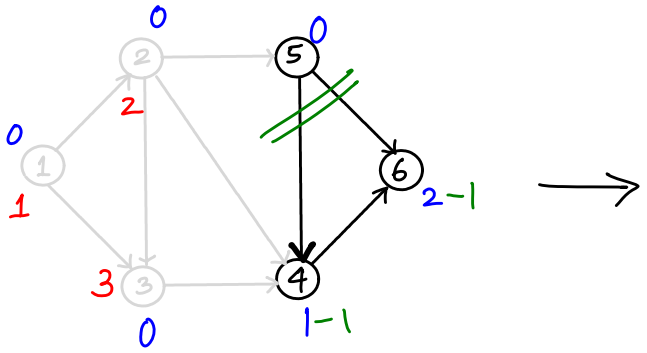
LIST: [1] → [7]



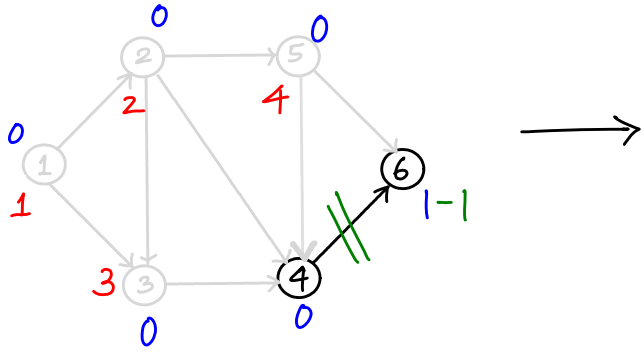


LIST: [3 5] → [5] → [4]

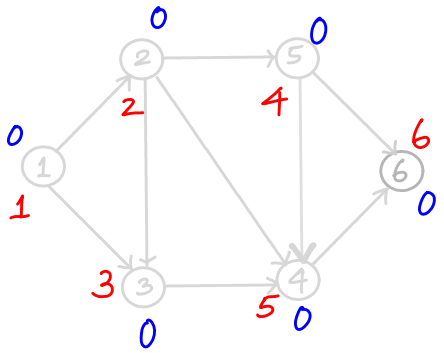




LIST: [4] → [6]

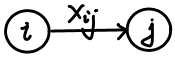


LIST: [6]

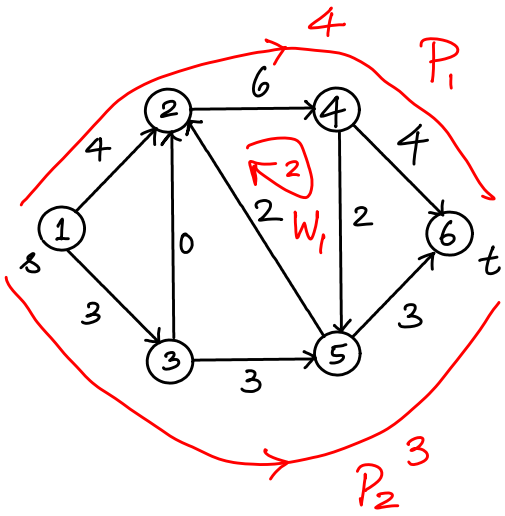


Flow Decomposition

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.



Consider this example network, with flows on arcs x_{ij} given. We assume x_{ij} values satisfy flow balance as well as bound constraints.



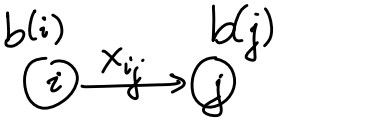
Instead, we could consider flows along two paths and a cycle.

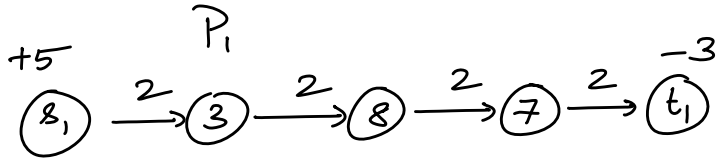
P_1: 1-2-4-6 sending 4 units

P_2: 1-3-5-6 sending 3 units

W_1: 2-4-5-2 circulating 2 units.

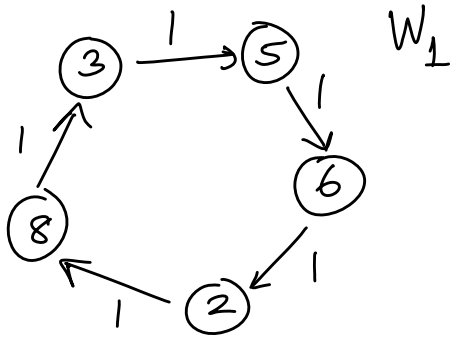
Let P_1 be a path from s_1 to t_1 as shown here:





We specify that the intermediate nodes in P_1 conserve flow, i.e., inflow = outflow is satisfied when restricted to the path. Note that each node and arc in P_1 could be part of other paths and/or cycles. In particular, the value $b(s_1) = 5$ is not determined just by the flows in the arcs in P_1.

To describe flow along a cycle W_1, we specify that every node in W_1 satisfies inflow = outflow.



We denote by $f(P)$ and $f(W)$ the flow in path P and cycle W.
We also specify paths and cycles using indicator variables δ_{ij} for each arc (i,j).

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

Claim Given flows in a set of paths $\mathcal{P}$ and set of cycles $\mathcal{W}$, we can get the flows in arcs using the δ_{ij} indicators:

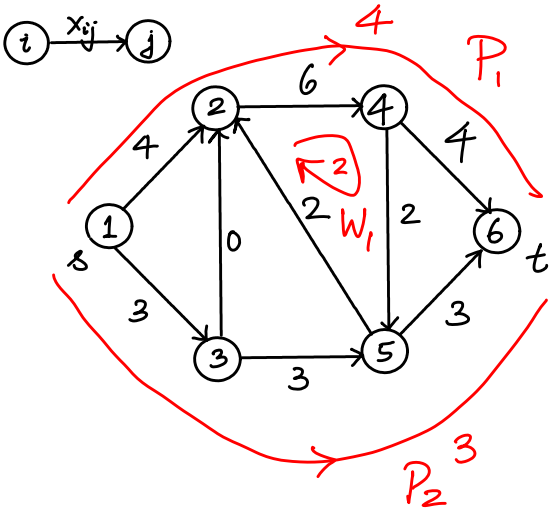
$$x_{ij} = \sum_{P \in \mathcal{P}} f(P)\delta_{ij}(P) + \sum_{W \in \mathcal{W}} f(W)\delta_{ij}(W) \quad \forall (i,j) \in A.$$

Consider the example again. Here, $f(P_1) = 4$, $f(P_2) = 3$, and $f(W_1) = 2$.

Arc $(2,4)$ is part of $P_1 = 1\text{-}2\text{-}4\text{-}6$ and $W_1 = 2\text{-}4\text{-}5\text{-}2$.

Indeed, we get

$$x_{24} = f(P_1) + f(W_1) = 4 + 2 = 6.$$



The more interesting question is, given a set of arc flows (x_{ij}'s), can you find an equivalent collection of path and cycle flows?

Yes! We repeatedly search for paths/cycles, and send flows along them until all arc flows are exhausted.

Flow decomposition algorithm

We work with intermediate flows y_{ij}. We start with $y_{ij} = x_{ij}$. We then search to find a path P from a supply node to a demand node, or a cycle W. We then push flow along P or W, update y_{ij}'s, $b(i)$'s, as well as the associated network. We repeat till $y_{ij} = 0 \ \forall (i,j) \in A$ and $b(i) = 0 \ \forall i \in N$.

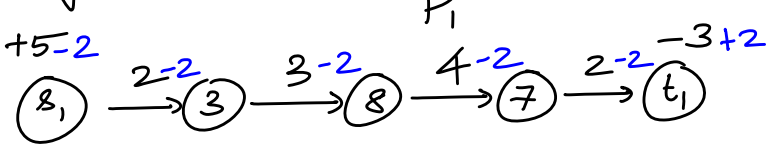
How much can we push? Consider the path $P = s - i_1 \cdots i_r - t$.

We define $\Delta(P) = \min\{b(s), -b(t), y_{ij} \ \forall (i,j) \in P\}$, and set $f(P) = \Delta(P)$.

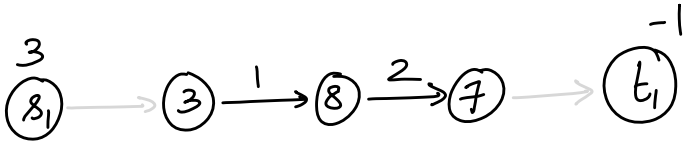
($\Delta(P)$ → "capacity" of P)

e.g., let P_1 be

P_1



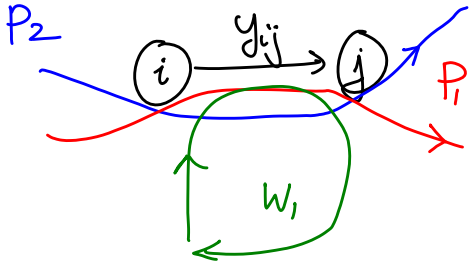
$\Delta(P_1) = \min\{5, 3, 2, 3, 4, 2\} = 2.$



Set $f(P_1) = \Delta(P_1) = 2$.

After the update (to y_{ij}, $b(i)$), the arcs $(s_1, 3)$ and $(7, t_1)$ are no longer considered in the network.

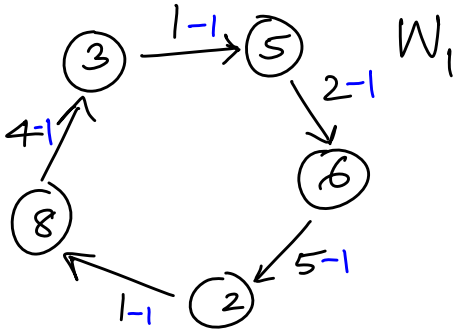
Note that (i,j) can be part of multiple paths and/or cycles, but it is always a forward arc in each path and cycle.



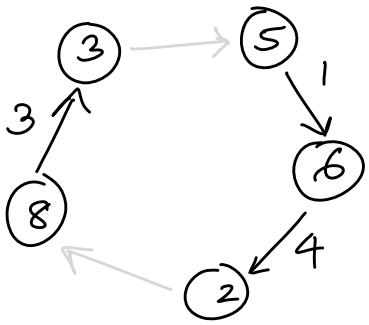
For a cycle W, we set $\Delta(W) = \min\{ y_{ij} \mid (i,j) \in W \}$.
↳ capacity of cycle W

Here, $\Delta(W_1) = 1$.

We set $f(W_1) = \Delta(W_1) = 1$.



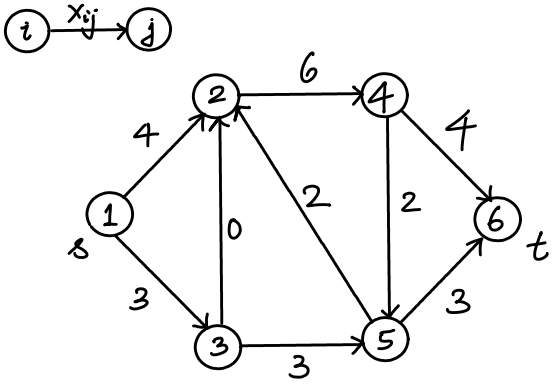
After update, $(3,5)$ and $(2,8)$ are removed from further consideration.

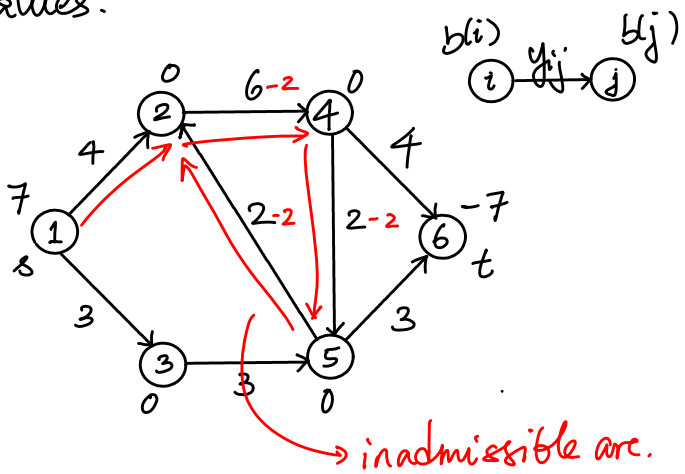


We ran DFS repeatedly. If we find a directed cycle, we push its capacity of flow. Else if we find a path from a supply to a demand node, we push its capacity. We update the intermediate flows and the network, and continue. See the algorithm in the handout posted on the course web page.

Illustration

Consider the input flow. We start by initializing $b(i)$ and y_{ij} values using x_{ij} values.





Notice we do not have (3,2) in the starting network, as $y_{32} = 0$ to start with.

We maintain sets of supply and demand nodes. $\mathcal{S}$, $\mathcal{D}$.

$\mathcal{S} = [1]$, $\mathcal{D} = [6]$ at the start.

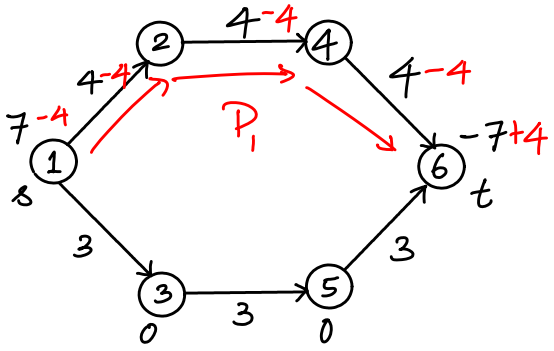
Start with $s = 1$ (taken from $\mathcal{S}$).

We perform DFS. We identify cycle $W_1 = 2\text{-}4\text{-}5\text{-}2$. We set

$\mathcal{W} = \{W_1\}$. $\Delta(W_1) = \min\{6, 2, 2\} = 2$. Set $f(W_1) = 2$.

The updated network is shown here.

We still have $\mathcal{S} = [1]$, $\mathcal{D} = [6]$.



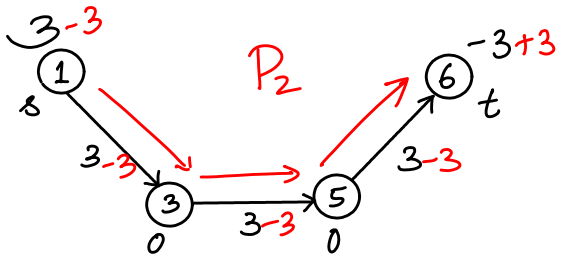
Starting with $s = 1$, we do DFS.

We identify $P_1 = 1\text{-}2\text{-}4\text{-}6$, with

$\Delta(P_1) = \min\{7, 4, 4, 4, 7\} = 4$.

We set $\mathcal{P} = \{P_1\}$, $f(P_1) = 4$.

$S = [1]$, $\mathcal{D} = [6]$



Start with $s=1$, do DFS. We find $P_2 = 1\text{-}3\text{-}5\text{-}6$ with

$$\Delta(P_2) = \min\{3, 3, 3, 3, 8\} = 3.$$

We set $\mathcal{P} = \mathcal{P} \cup \{P_2\} = \{P_1, P_2\}$, and $f(P_2) = 3$.

After this iteration, all $b(i) = 0$, $y_{ij} = 0$. Thus we have decomposed the arc flow into flows along paths P_1, P_2, and cycle W_1.

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 flows.

The input flow $\bar{x} = [x_{ij}]$ (vector of x_{ij} values) is assumed to satisfy flow balance, and bounds. But it need not necessarily be an optimal flow — notice that we do not worry about costs in this decomposition.

Similarly, the bounds (u_{ij}) do not play a direct part in the flow decomposition. Naturally, $f(P)$ and $f(W)$ cannot exceed y_{ij}, and hence x_{ij} values.

MATH 566: Lecture 10 (09/19/2024)

Today: * Shortest path problem: – assumptions, analogy
– application
– Dijkstra's algo

We finish the discussion on flow decomposition with a theorem:

Flow Decomposition Theorem

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

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

(b) the sum of flows around directed cycles.

The decomposition will have <u>at most</u> $m+n$ directed paths and cycles, out of which at most m are cycles.

<u>IDEA</u> Each step of the flow decomposition algorithm identifies a path P or a cycle W. Pushing $\Delta(P)$ or $\Delta(W)$ (capacity of P or W) will necessarily zero out at least one $b(i)$ or one y_{ij}.

Corollary Any circulation $\bar{x}$, i.e., flow $\bar{x}$ with $b(i)=0 \; \forall i$, can be decomposed into the sum of flows around at most m directed cycles.

Notice that we could use the bounds of $m+n$ 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.

Shortest Path (SP) Problem

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

We study the following generalization of the SP problem: → as previously introduced

Given: $G=(N,A)$ with costs c_{ij} for $(i,j)\in A$, and a source node s.

Goal: find shortest length directed paths from s to all $j \in N/\{s\}$.

Assumptions

1. c_{ij} are integers.
   For ease of analysis of complexity of algorithms.

2. There is a directed path from s to each $i \in N/\{s\}$.

   If there is no such path to a particular node j, we add an extra arc (s,j) with $c_{sj} = +\infty$. If such an arc is part of the SP tree, we know that node is in fact not reachable from s via a directed path.

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. Under this assumption, each arc would be used at most once in the SP tree.

In general, we do permit some c_{ij}'s to be negative, as long as there are no negative cost cycles.

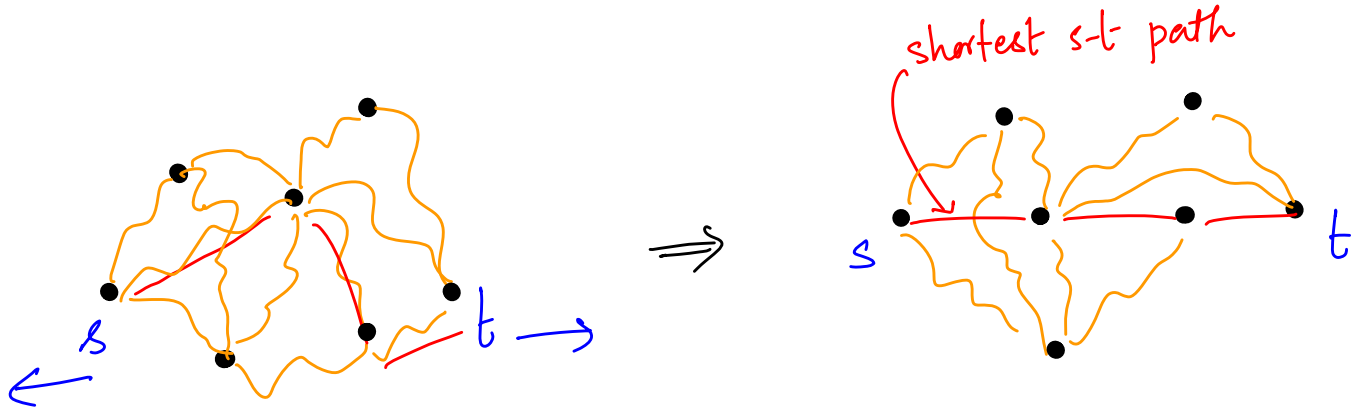
We first consider the case where all $c_{ij} \geq 0$. This case will be easier than the general case that allows $c_{ij} < 0$. As it turns out, we would use similar ideas to solve both cases.

An Analogy: The string model

Assume $c_{ij} \geq 0$.

Make a model of G with beads and strings. Each node gets a bead, and each arc gets an inelastic string of length c_{ij}.

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 (or tight) first represents an s-t shortest path.

Applications of the Shortest Path Problem

Recall: TeX paragraph spacing problem (AMO Problem 1.7). Section 4.3 in AMO describes many more applications modeled as SP problems. We consider one case here.

The knapsack Problem

A hiker wants to decide which items to include in her knapsack. She has to choose from n items. The knapsack can carry up to W lbs. Item i has weight w_i and utility u_i. The goal is to maximize total utility while not exceeding total weight W. 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

Notation

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

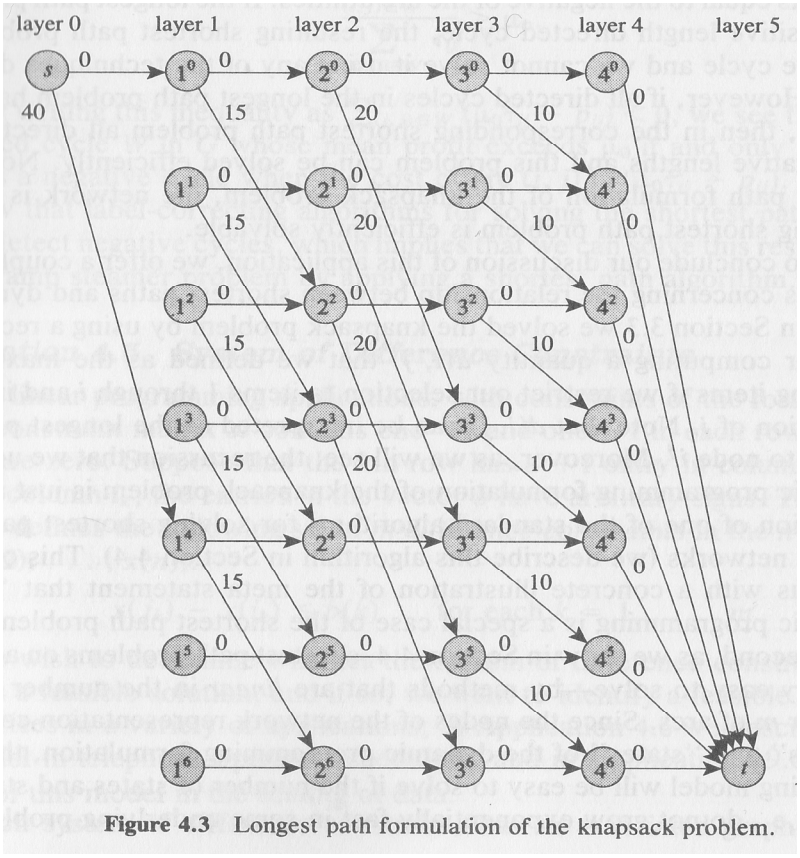
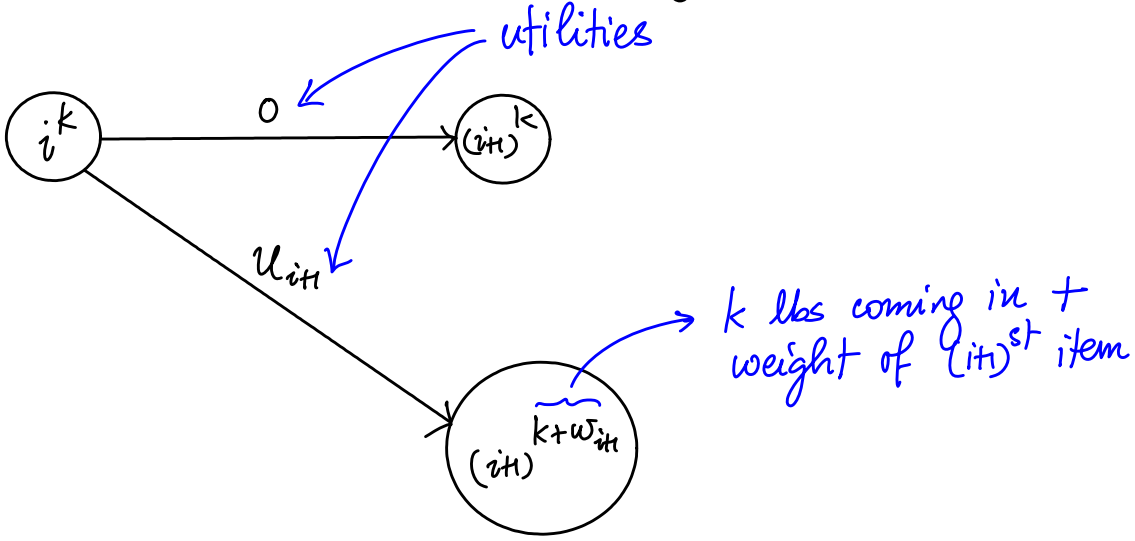


Figure 4.3 Longest path formulation of the knapsack problem.

The figure illustrates a longest path formulation of this problem.

From node (i^k), we have two outarcs, corresponding to the hiker including item $i+1$ in the knapsack, or leaving it out.



From s, we add $(s, 1^0)$ with utility 0, and $(s, 1^{w_1})$ with utility u_1. Finally, we connect all nodes from Layer n (last layer), i.e., nodes n^w for $w = 1, 2, \ldots, W$, to t with zero utility arcs.

Find the s-t longest path. When $u_i \geq 0 \; \forall i$, we can negate all utilities and solve the s-t SP problem.

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 Shortest Path Problem (Recall, we assume $c_{ij} \geq 0 \; \forall (i,j)$ for now)

Let $d(\cdot)$ denote a vector of temporary distance labels for the nodes. Thus, $d(i)$ = length of **some** path from s to i. Hence, $d(i)$ is an upper bound on the SP length from s to i.

A key step in all SP algorithms

Procedure UPDATE(i)

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

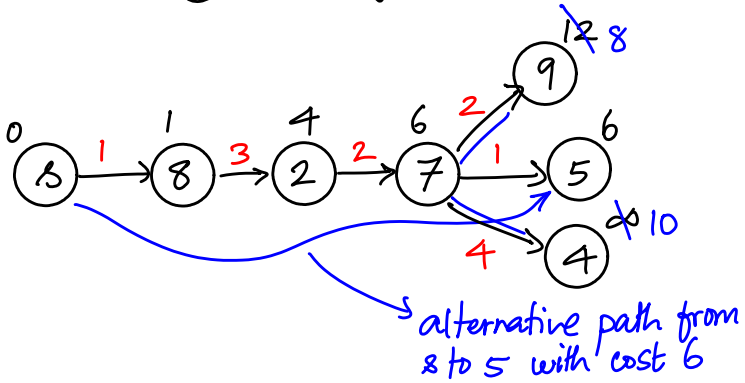
    if $d(j) > d(i) + c_{ij}$ then

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

      pred(j) := i;

    endif

  endfor



UPDATE(7): we update $d(4)$ and $d(9)$, but do not change $d(5)$.

Note the similarity to the process of looking for admissibility of arcs from node i, and marking the heads of admissible arcs.

When the $d(i)$'s become permanent, they represent SP distances from s to i.

We now present the first algorithm for SP problems where $c_{ij} \geq 0$ for all (i,j).

Dijkstra's algorithm (pronounced 'Daikstraa')

We maintain and update two (sub) sets of nodes.

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

$\bar{S} = N \setminus S$: set of nodes with temporary $d(i)$'s.

```
algorithm Dijkstra;
begin
    S := ∅; S̄ := N;
    d(i) := ∞ for each node i ∈ N;          } initialization
    d(s) := 0 and pred(s) := 0;
    while |S| < n do                        → run till every d(i) is made permanent
    begin
        let i ∈ S̄ be a node for which d(i) = min{d(j) : j ∈ S̄};   } node selection
        S := S ∪ {i};                       } make i "permanent"
        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;   } UPDATE(i)
    end;
end;
```

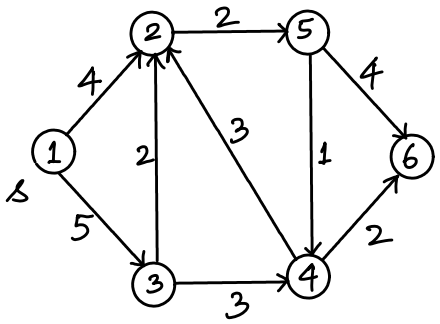
Figure 4.6 Dijkstra's algorithm.

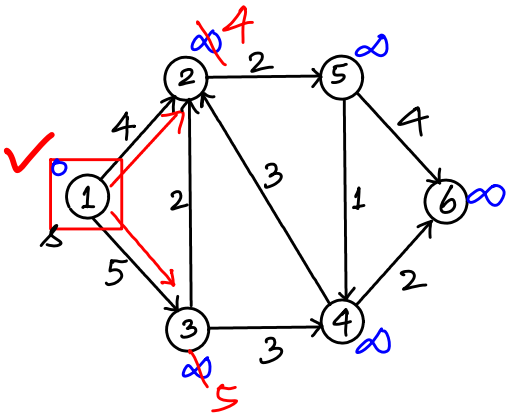
In words, pick node i with smallest $d(\cdot)$ from $\bar{S}$, make it permanent, i.e., move it to S. Then run UPDATE(i) by exploring the out arcs of node i.

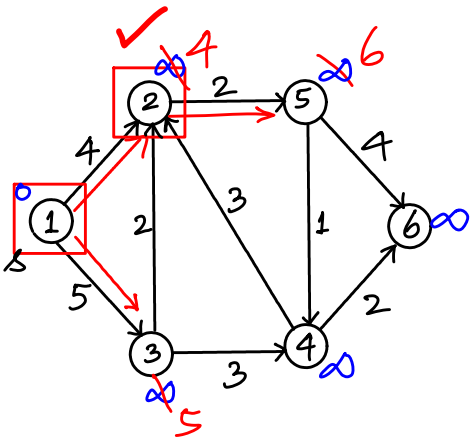
Illustration

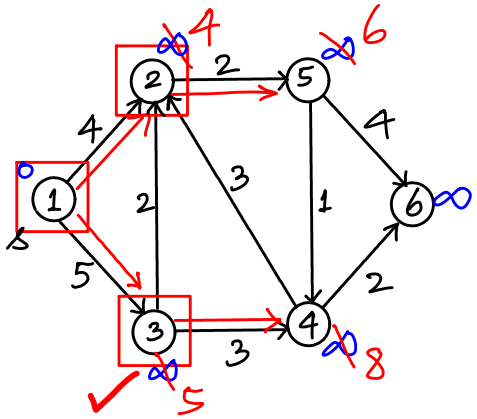
✓ : latest node that's made permanent

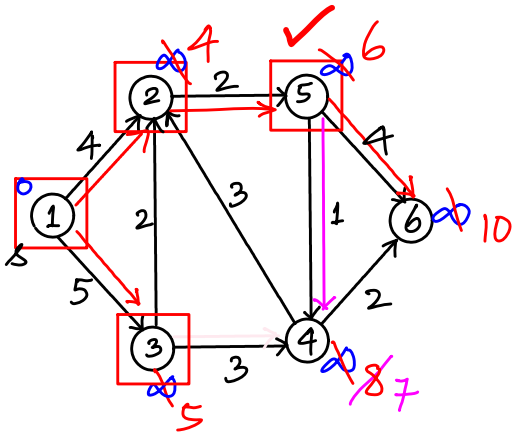
[i] : node i is made permanent

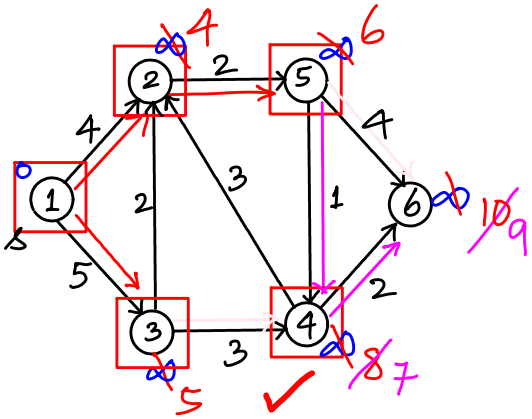


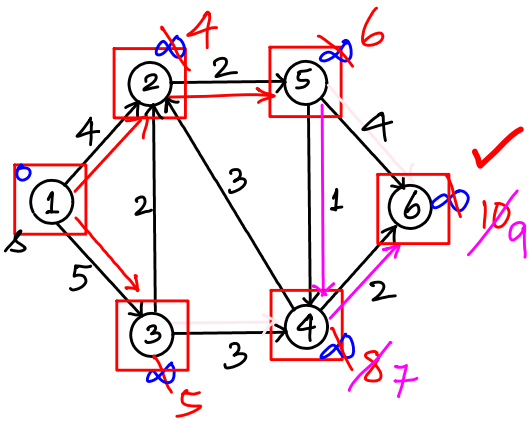




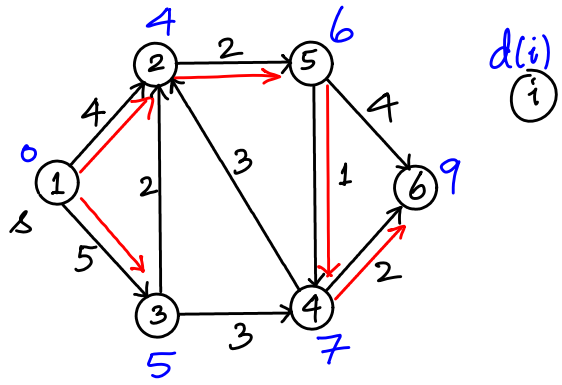








Here is the shortest path tree:



Notice that the SP tree could be different from the BFS tree. For instance, we could get to 4 from 1 via 3 (1-3-4) in BFS, but we go 1-2-5-4 in the SP tree.

MATH 566: Lecture 11 (09/24/2024)

Next Thursday, Oct 3, is the midterm exam
— watch out for an email with more details on the same.

Today: * Dijkstra's algorithm
- correctness
- complexity
- variants

Correctness of Dijkstra's Algorithm

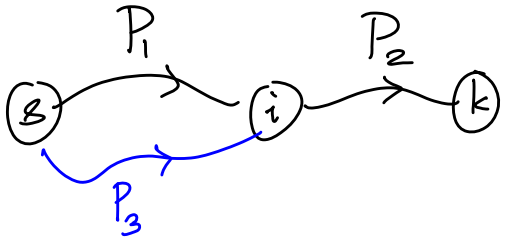
We first prove some properties of shortest paths. We will use them to subsequently prove the correctness of Dijkstra's algorithm.

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

Proof

Let the shortest path from s to k be broken into subpaths P_1 from s to i, and P_2 from i to k.

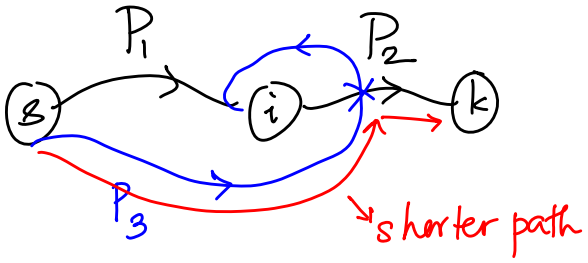
The property is saying that P_1 is then an SP from s to i.



$P = P_1 \cup P_2$ vs $P_3 \cup P_3$

To show the property, we consider an alternative path P_3 from s to i that is possibly shorter than P_1. At the same time, such a path when combined with P_2 (from i to k) need not still be a path!

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



By applying flow decomposition, this directed walk can be broken into a union of directed paths and directed cycles.

As all $c_{ij} \geq 0$, the directed cycles have nonnegative costs, and hence the directed paths in the decomposition will give a shorter s-k path (than P). Essentially, we can "short-cut" the cycles, and just use the paths. But this contradicts the optimality of P!

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 $(i,j) \in P$.

Proof

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

($\Leftarrow$) Let $d(j) = d(i) + c_{ij}$ $\forall (i,j) \in P$. We will show P is an SP.

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

$$d(k) = d(i_h) = d(i_{h-1}) + c_{i_{h-1}, i_h}$$
$$= d(i_{h-2}) + c_{i_{h-2}, i_{h-1}} + c_{i_{h-1}, i_h}$$
$$\vdots$$
$$= \sum_{(i,j) \in P} c_{ij} \quad (\text{as } d(i_1) = d(s) = 0).$$

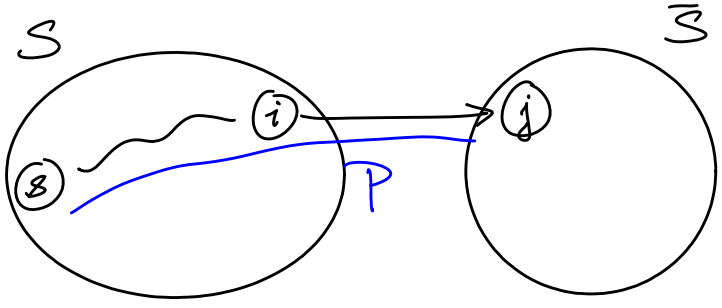
Hence the length of path P is equal to $d(k)$, the shortest path distance from s to k. So P is a shortest s-k path. □

We now consider the correctness of Dijkstra's algorithm. To prove the same, use induction on $|S|$, the subset of permanently labeled nodes.
($|S|$: cardinality of S, i.e., # nodes in S.)

At any iteration, the following two conditions hold.

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

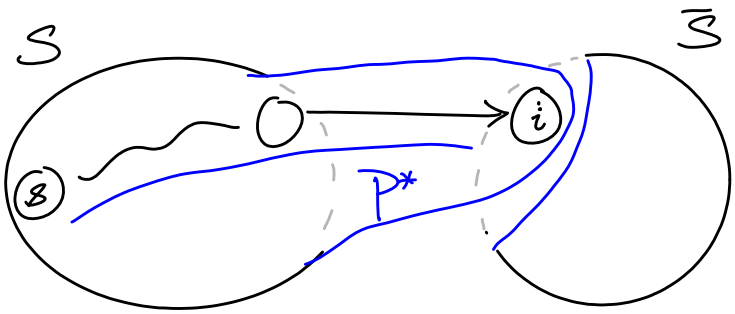
(2) $\forall j \in \bar{S}$, $d(j)$ is the length of a shortest path P from s to j such that all nodes $k \in P$ satisfy $k \in S \cup \{j\}$. If there is no such path, then $d(j) = +\infty$.



In words, the shortest path P sits completely within S, except for the last arc, where it jumps from some node $i \in S$ to $j \in \bar{S}$ via arc (i,j).

We assume properties (1) and (2) hold for $|S| = k-1$. We show they continue to hold when $|S| = k$, i.e., when we move the next node from $\bar{S}$ to S.

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



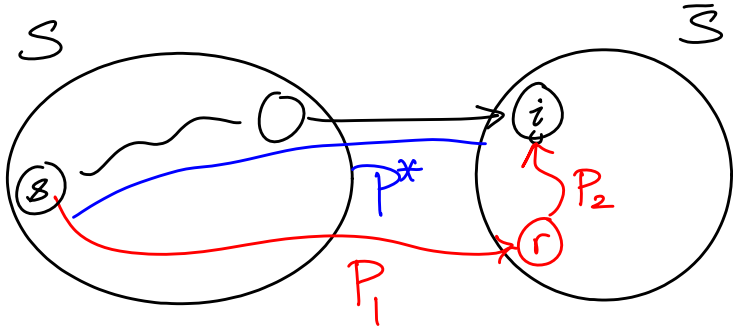
Here is how S, $\bar{S}$ look after (before) this step

Recall the steps: * Node_Selection (move i to S from $\bar{S}$)
* UPDATE (i)

We show that conditions (1) and (2) hold after both the steps (in the k^{th} iteration).

By induction assumption, $d(i)$ 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 s-i path containing at least one other node $r \in \bar{S}$ is at least as big as $d(i)$.

Consider $P = P_1 \cup P_2$ as shown.



$\text{length}(P_1) = d(r)$

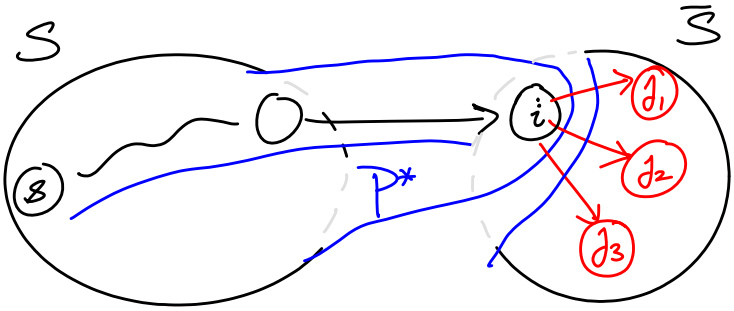
$\text{length}(P_2) \geq 0$

By the way we choose i (as $\underset{j \in \bar{S}}{\text{argmin}} \{d(j)\}$), $d(i) \leq d(r)$.

$\Rightarrow \text{length}(P) = \text{length}(P_1) + \text{length}(P_2) \geq d(i)$.

Hence condition (1) holds for $|S| = k$ as well.

To show condition (2) also holds, we consider the UPDATE(i) operations.



After UPDATE(i), $d(j)$ gets possibly updated to $d(i) + c_{ij}$ for $(i,j) \in A(i)$.

After these updates, $d(j) \geq d(i) \; \forall j \in \bar{S}$.

Also, the path from s to j will satisfy $d_\ell = d_k + c_{k\ell}$ for all (k,ℓ) in the path (by induction hypothesis). Hence Result (2) also holds.

□

In other words, Dijkstra's algorithm picks nodes from $\bar{S}$ in the increasing order of their shortest path lengths from s.

Complexity of Dijkstra's Algorithm

The bottleneck steps are node selection and UPDATE(i) in each iteration.

```
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};   } Node selection
        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;   } UPDATE(i)
    end;
end;
```

Figure 4.6 Dijkstra's algorithm.

1. The UPDATE steps (overall) run $\sum_{i \in N} |A(i)| = m$ times, i.e., it takes $O(m)$ time. → steps within each UPDATE take constant time each.

2. Node selection: Select a node in each step — n times. Each time, compare with $d(j)$ $\forall j \in \bar{S}$. Hence the running time is $O(\underbrace{n + n-1 + n-2 + \cdots}_{\frac{n(n+1)}{2}}) = O(n^2)$.

Theorem 4.4 Dijkstra's algorithm solves the shortest path problem with $c_{ij} \geq 0$ $\forall (i,j) \in A$ in $O(n^2)$ time.

Variants of Dijkstra's Algorithm

1. Reverse Dijkstra's algorithm: Given a terminus node t, find SPs from all $i \in N \setminus \{t\}$ to node t.

(Assumptions: * $c_{ij} \geq 0 \ \forall (i,j) \in A$

* $\exists$ directed path from i to t $\forall i \in N \setminus \{t\}$)

→ if not, add arc (i,t) with $c_{it} = +\infty$

We reverse the sense of Dijkstra's algorithm. We maintain lists S' and $\bar{S}'$ of permanently and temporarily labeled nodes.

Pick $i \in \bar{S}'$ with $d(i) = \min_{j \in \bar{S}'} \{d(j)\}$ and move it to S'.

→ in-arc list.

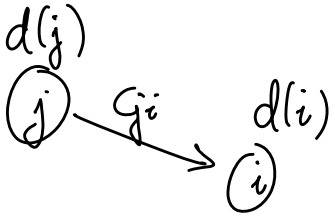
Then do UPDATE(i) on $AI(i)$.

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

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

pred$(i) = j$;

end-if



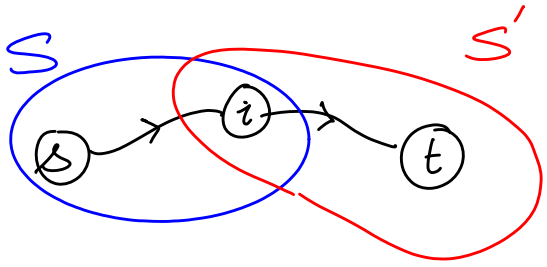
2. Bidirectional Dijkstra's algorithm for the shortest s-t path

This is the original version of the shortest path problem, where both a source and sink node s and t are given.

Run forward Dijkstra from s and reverse Dijkstra to t.

When the same node gets permanently labeled by both runs, we have a shortest s-t path.

We run the next iteration for the forward Dijkstra and then the next iteration of reverse Dijkstra. When the forward SP tree and the reverse SP tree intersect, we get a shortest s-t path.



This approach is still $O(n^2)$ time, but could be quite fast in practice.

MATH 566: Lecture 12 (09/26/2024)

Today: * Dial's implementation of Dijkstra
* SP optimality conditions
* generic label correcting algorithm

Dial's Implementation

Bottleneck of Dijkstra's algorithm is the node selection steps ($O(n^2)$).
Dial's implementation stores nodes with finite temporary labels in a sorted fashion.

Property The permanent distance labels maintained by Dijkstra are non-decreasing.

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

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

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

  We store BUCKETs in increasing order of k.

* Whenever $d(\cdot)$ is updated, update BUCKETs too.

* FINDMIN : procedure to look for the first non-empty BUCKET, and delete node(s) in that BUCKET (after making them permanent).

Insertions/deletions of nodes from a BUCKET can be done in $O(1)$ time (by maintaining BUCKETs as doubly linked lists). ↳ constant

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

Complexity

BUCKETs needed = $O(nC)$

time for UPDATES = $O(nC)$

UPDATES = $O(m)$

time for FINDMIN = $O(nC)$

total running time is $O(m+nC)$. → pseudopolynomial time algo

Could be improved to $O(m+C)$ — see AMO.

(storing $C+1$ BUCKETs suffices).

Shortest Path Optimality Conditions

(AMO Chapter 5)
Not included in midterm exam

Recall, $d(j)$ represents the length of some path from s to j.
We specify conditions that distance labels $d(j)$ must satisfy for them to represent SP distances.

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

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

Proof ($\Rightarrow$) Let $d(j)$ be the SP length from s to j, $\forall j \in N$.
Assume (1) does not hold. Hence there exists $(i,j) \in A$ such that $d(j) > d(i) + c_{ij}$. Hence we can improve the SP length from s to j by taking the SP from s to i and adding (i,j).
This contradicts optimality of $d(j)$.

($\Leftarrow$) Let $d(\cdot)$ be some path lengths satisfying (1). So, $d(j)$ is an upper bound on the SP length from s to j.

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

$$d(j) \leq \sum_{(i,j) \in P} c_{ij} \qquad (2)$$

(as $d(s) = 0$).

(2) holds for all s-j paths P. Hence $d(j)$ is a lower bound on the SP length from s to j.
$\Rightarrow d(j)$ is exactly the SP length from s to j.

□

The Generic label-correcting algorithm

We assume there are no negative cost cycles, but some (or all) c_{ij}'s could be < 0.

We maintain $d(\cdot)$ and $pred(\cdot)$, and successively update them until optimality conditions are satisfied.

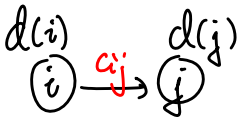
$$d(j) = \infty \quad \forall j \in N$$
$$pred(j) = 0 \quad \forall j \in N$$
$$d(s) = 0$$

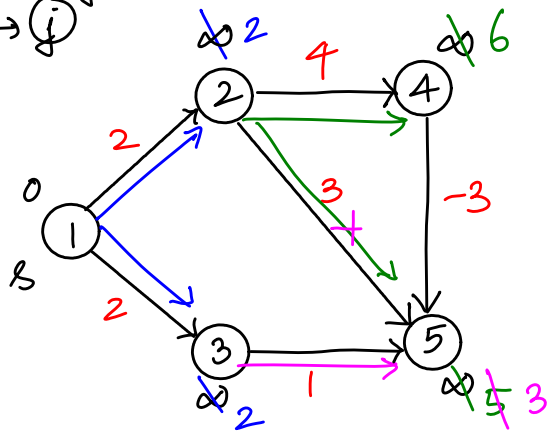
while $d(j) > d(i) + c_{ij}$ for some $(i,j) \in A$ do

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

end

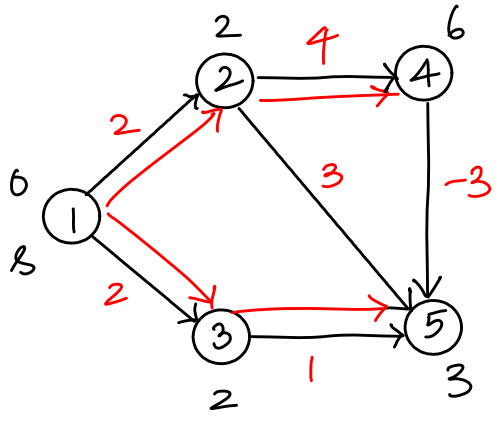
Illustration





Steps: 1, 2, 3

SP tree



$d(j)$ satisfy optimality conditions for all (i,j)

Proof of Finiteness

Assume c_{ij}'s are integers.

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

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

* Any $d(j)$ is updated at most $2nC$ times.

  Hence the total # distance updates is at most $2n^2C$.

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

Complexity : $O(2^n)$ (see AMO for details)

We will talk about polynomial time implementations of the generic label correcting algorithm.

If there are negative cycles, the algorithm is no longer finite. But we can stop as soon as any $d(j) < -nC$.

Modified Label-Correcting Algorithm

Q: How do we select arcs violating (1) efficiently?

* maintain a LIST of nodes, such that
* if (i,j) violates (1), then LIST must contain i;
* When we update $d(j)$, add j to LIST.

```
begin
LIST = [s];
pred(s) = 0;
while LIST ≠ ∅ do
        remove i from LIST
        for all (i,j) ∈ A(i) do
            if  d(j) > d(i) + c_ij
                    d(j) := d(i) + c_ij;
                pred(j) := i;
                LIST := LIST ∪ {j};
            end_if
        end_for
end_begin
```

MATH 566: Lecture 13 (10/01/2024)

Today: * label correcting algo
— poly time implementation
* practice midterm

Complexity of the modified label correcting algorithm

* $d(j)$ is updated at most $2nC$ times. → $-nC \le d(j) \le nC$
* After update of $d(j)$, j is added to LIST, and arcs in $A(j)$ are scanned afterward.
* Total # arc scans $= \sum_{j \in N} 2nC\,|A(j)|$. ($|A(j)|$ = # outarcs of j)

$\Rightarrow$ Modified label correcting algorithm runs in $O(mnC)$ time. → pseudopolynomial time algorithm

We have come from an exponential time algo ($O(2^n)$) to a pseudopolynomial time algo (pseudopolynomial due to the dependence on C, rather than $\log C$). But we now present an implementation that runs in polynomial time — $O(mn)$!

FIFO Implementation of the 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).
* Maintain LIST as a FIFO queue.

In fact, we don't have to examine all arcs in A in each pass! We can take a node i from top/front of the LIST, and examine only its outarcs, i.e., the arcs in $A(i)$. If we update $d(j)$ for any arc $(i,j) \in A(i)$ and $j \notin$ LIST, then we add j to the back of the LIST. See AMO Fig 5.5 (in pg 141) for pseudocode.

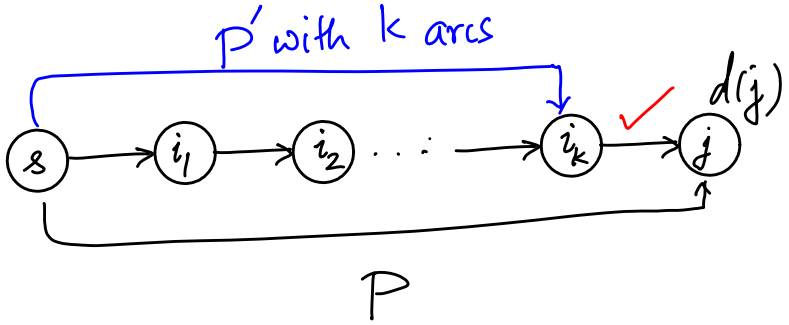
AMO Theorem 5.3 The FIFO label correcting algorithm solves SP in $O(mn)$ time, or else shows that there is a negative cycle.

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

Need to show the algo will run correctly in n passes.

Claim At the end of the k^{th} pass, the algorithm computes SP distances for all nodes whose SP from s has k or fewer arcs.

We use induction to prove this claim. Assume it holds for k passes. Consider node j.



P' is an SP from s to i_k with k arcs. By induction assumption, after k passes, we will have

$$d(i_k) = \sum_{(i,j)\in P'} c_{ij}.$$

In the $(k+1)$-st pass, we update $d(j)$ to $d(i_k) + c_{i_k,j}$. Hence $d(j)$ will be the SP distance from s to j with $k+1$ arcs.
$\Rightarrow$ The claim holds.

Note that if $d(j)$ is not updated in the $(k+1)$-st pass, node j will (trivially) be not a candidate to consider here.

Since any SP from s to j has at most $(n-1)$ arcs, the $d(j)$'s are optimal after $(n-1)$ passes. But if there is an update in the n^{th} pass, there exists a negative cycle. Else, the SP problem is solved.

In the modified label correcting algorithm, if we maintain LIST as FIFO queue, we get the $O(mn)$ running time. □

If we maintain LIST as a LIFO queue, it might work better in practice, especially on large instances. But we need the FIFO handling to prove the $O(mn)$ running time.

Detecting Negative Cycles

1. Stop if $d(j) \leq -nC$.
2. Run the FIFO label correcting algorithm, and stop if a node is scanned at least n times.
3. Keep track of the # arcs in an SP from s to j, $j \in N$, and stop if any SP has more than $(n-1)$ arcs.

Review for Midterm

For Problems 1-3, just eyeball solutions! No proof of correctness is required, nor are any evidence of use of any algorithms.

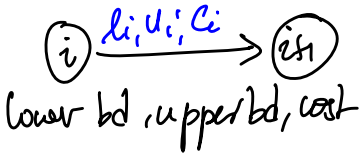
6. FALSE. $O(\cdot)$ does not hold, e.g., $f(n)=n^2$, $g(n)=n$.

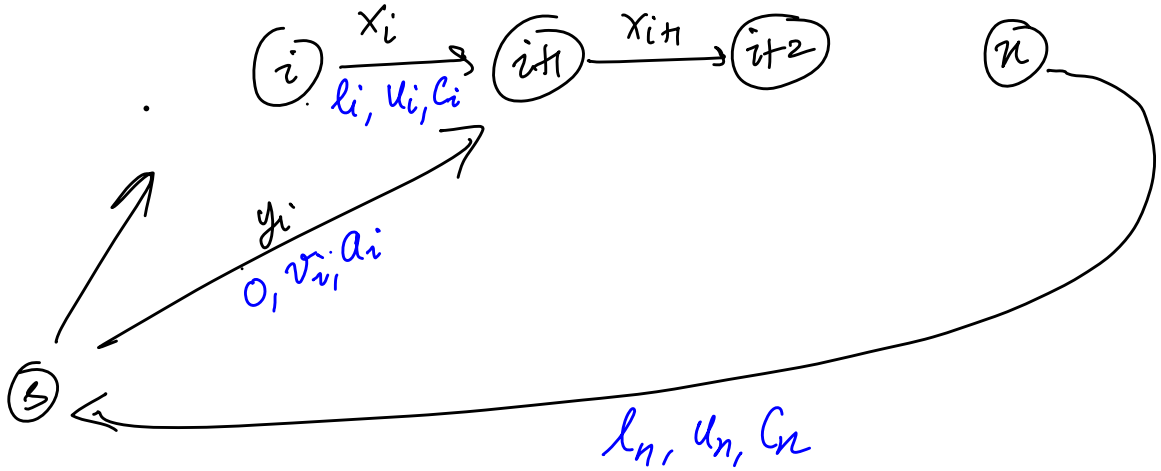
8. $y_i \in [0, v_i] \Rightarrow 0 \le y_i \le v_i$ (lower bound, upper bound)

$x_i \in [l_i, u_i] \Rightarrow l_i \le x_i \le u_i$

All #s (v_i, l_i, u_i)'s suggest lb/ub for arcs. Then there are costs c_{i+1}, a_i. But no supply/demand!

$\Rightarrow$ Circulation model ✓!





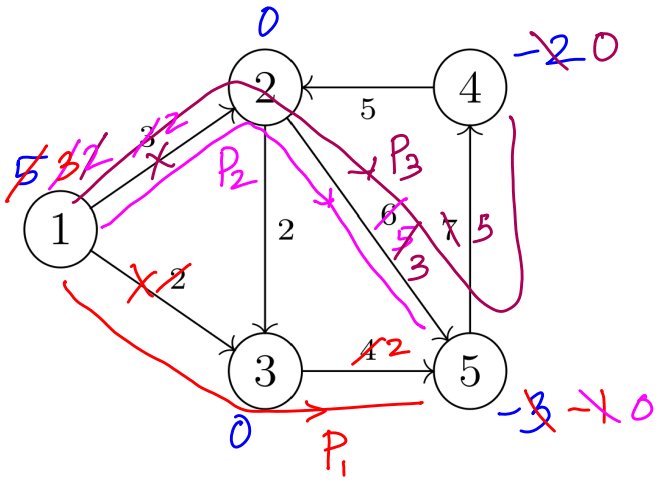
See Solutions for details!

2. Flow decomposition problem

Depending on the order in which you choose paths and cycles, you can get different decompositions.

b(i)

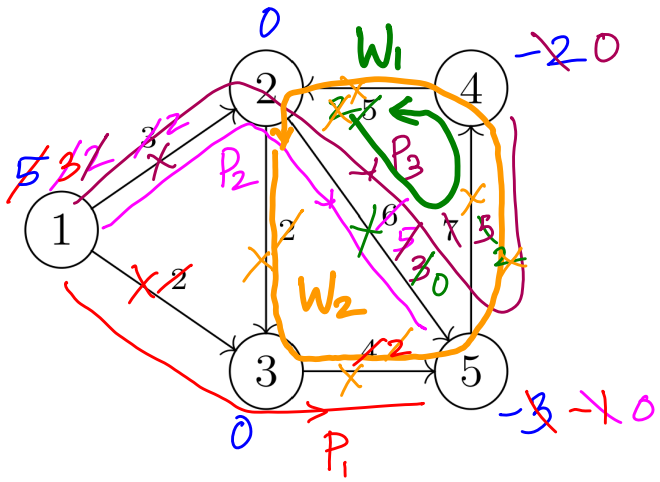
(i)



$\Delta(P_1) = 2$
$\Delta(P_2) = 1$
$\Delta(P_3) = 2$

Identify the Supply and Demand subsets of nodes Here, $\mathcal{S} = \{1\}$ and $\mathcal{D} = \{4, 5\}$.

If we identify paths first, one option is to get P_1, P_2, P_3 as shown.



$\Delta(W_1) = 3$
$\Delta(W_2) = 2$

We could then identify cycles W_1 and W_2 as shown.

Naturally, alternative choices exist!

MATH 566: Lecture 15 (10/08/2024)

OFFER to make-up midterm scores:

* If you score $\geq 92\%$ in the remaining Homeworks, that hw score will replace your midterm score.
* If you score $\geq 85\%$ (but $< 92\%$), then the midterm score will count for 10% of total score. → instead of 20%.

Today:
* all pairs SP
* max flow

All Pairs Shortest Path Problem

Goal: Find SP from i to j $\forall i,j \in N$.

We could solve n SP instances with $s=i$ $\forall i \in N$.

Assumptions

* c_{ij} can be < 0
* no negative cycles

But could we somehow use Dijkstra's algo (it is more efficient)? Note that some c_{ij} could be < 0 to start with.

Yes! We can use **reduced costs**:

Let $\bar{c}_{ij}^{d} = c_{ij} + d(i) - d(j)$ for given $d(i), i \in N$.

Recall: SP optimality conditions: $d(j) \leq d(i) + c_{ij}$.

$\Rightarrow$ If SP optimality conditions are satisfied by $d(\cdot)$, then $\bar{c}_{ij}^{d} \geq 0$.

Def More generally, for a set of node potentials $\pi(i)$ (vector $\bar{\pi}$), the reduced costs are

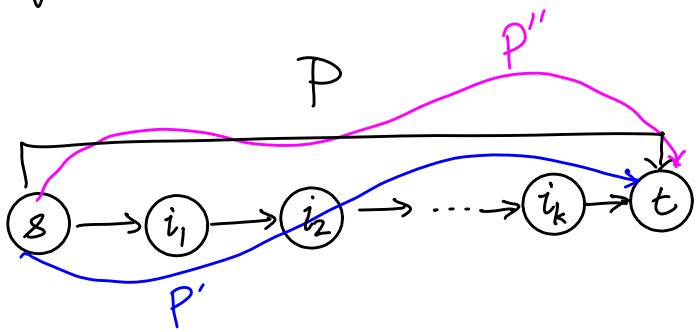
$$c_{ij}^{\bar{\pi}} = c_{ij} - \pi(i) + \pi(j).$$

We will use reduced costs again (in MCF, ...)

Here, we use $\bar{\pi} = -\bar{d}$. So, could we use $c_{ij}^{\bar{d}}$ instead of c_{ij}?

Property A shortest path w.r.t. c_{ij} is also a shortest path w.r.t $c_{ij}^{\bar{d}}$ for a given vector $\bar{d}$ (of $d(i)$'s).

Proof



The ordering of all s-t paths $(P, P', P'', \ldots)$ in terms of their total cost remains the same!

$$\sum_{(i,j)\in P} c_{ij}^{\bar{d}} = \sum_{(i,j)\in P} c_{ij} + \underbrace{d(s) - d(t)}_{\text{constant for a given } \bar{d}}$$

The cost of any s-t path is shifted by only a constant, hence the property holds. □

All-Pairs SP Algorithm

1. Solve SP with $s = 1$ (say). Let $d(j)$ be the SP distance from s to j, $j \in N$.
2. For $s = 2 \ldots n$, solve $SP^{\bar{d}}$ with $c_{ij}^{\bar{d}}$.

Complexity

Step 1. $O(mn)$ FIFO label correcting algorithm

Step 2. Each SP computation takes $O(m + n \log C)$.
(Using Dial's or radix heap implementation of Dijkstra's algorithm)

Overall: $O(mn + n(m + n \log C))$ → (n-1), to be precise

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

For small values of C, this algorithm is essentially quadratic, and not cubic.

For the default SP (single source) case, we considered Dijkstra's algo first, and then optimality conditions. We follow the same trend for all pairs SP.

All Pairs Shortest Path Optimality Conditions

Let $d[i,j]$ be the length of some finite walk from i to j. So, $d[i,j]$ is an upper bound for the SP length from i to j.

We also let $d[i,i] = 0 \ \forall i \in N$, and $d[i,j] \leq c_{ij} \ \forall (i,j) \in A$.

↳ follows from the assumption that there are no negative cycles.

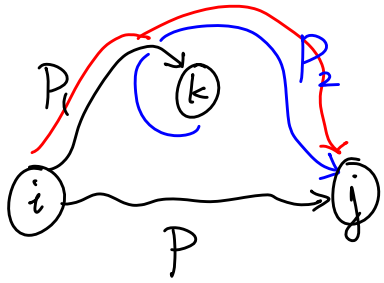
↳ else the arcs themselves are the shortest paths — somewhat trivial!

AMO Theorem 5.5 $d[i,j]$ represent shortest path lengths from i to j iff $d[i,j] \leq d[i,k]+d[k,j] \quad \forall i,j,k$.

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 triplet of nodes i,j,k.



$P_1 \cup P_2$ is a directed walk from i to j. This walk can be decomposed into a union of paths and cycles, from which we could obtain a path from i to j whose total length is $\leq d[i,k]+d[k,j]$ (as there are no negative cycles).

This path contradicts the optimality of $d[i,j]$.

The reverse direction ($\Leftarrow$) follows similar arguments to those used in the corresponding proof for the default SP case (single source). □

Floyd-Warshall algorithm for all pairs SP

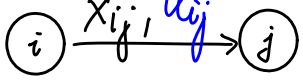
Repeatedly check for violations of all pairs SP optimality conditions, and UPDATE when violations are found.

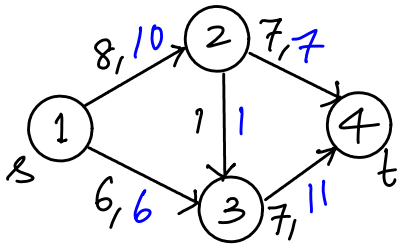
Complexity is $O(n^3)$. → check all triplets of nodes i,j,k. Note that this complexity term does not depend on m (# arcs).

Maximum Flows (AMO Chapter 6)

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

Here is an illustration:





value of flow here is

$$v = 8+6 = 7+7 = 14.$$

Consider the mathematical model (linear program):

v → value of flow

$$\max \quad v$$

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

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

Compare this model with the default min-cost flow model. Note that v is a variable here! All nodes except s and t are transshipment nodes. And we do not worry about costs c_{ij}'s.

Assumptions

1. $G_R = (N, A)$ is directed.

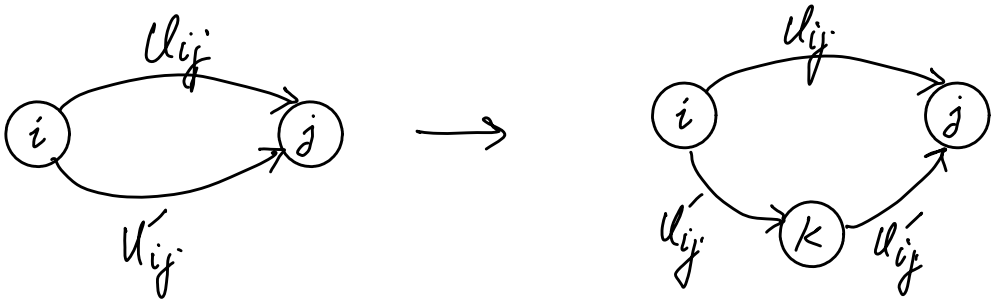
2. $u_{ij} \in \mathbb{Z}_{\geq 0}$ (nonnegative integers) → makes complexity analysis easier.

3. G_R does not contain a directed s-t path P with $u_{ij} = \infty \ \forall (i,j) \in P$. → else, we can keep pushing arbitrarily large amounts of flow from s to t

4. When $(i,j) \in A$, (j,i) is also present in A. Else, we add (j,i) with $u_{ji} = 0$.

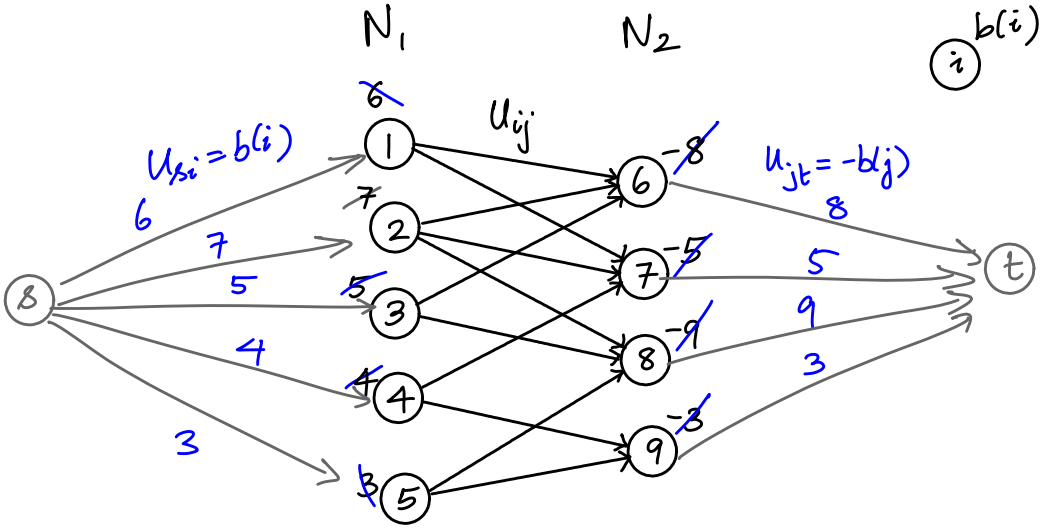
We will see soon that most algorithms for max flow try to push flow "up" an arc, and possibly pull back some of the flow. Hence we assume (i,j) & $(j,i) \in A$ in "preparation", so to speak.

5. There are no parallel (forward) arcs. If such arcs are present, use extra node(s) to "split" all but one of them.



Applications of Max Flow (Read AMO Chapter 6 Section 3!)

Feasibility problem: Find a feasible flow in $G=(N,A)$ with $b(i)$'s (with $\sum b(i)=0$) and u_{ij}'s. For instance, consider the transportation problem. The question is to find if there is a way to ship from supply nodes to demand nodes to satisfy all demands. This is the feasibility version, before the costs are considered. We formulate this problem as a max flow problem. Here is an example.



Add nodes s,t, arcs (s,i) with $u_{si}=b(i)$ $\forall i\in N_1$, (j,t) with $u_{jt}=-b(j)$, $j\in N_2$. We then solve the max flow problem. If the max flow saturates all arcs (s,i), $i\in N_1$ (equivalently, all arcs (j,t), $j\in N_2$), then there exists a feasible flow. The x_{ij} values for $(i,j)\in A$ in the input network give a feasible flow.

MATH 566 : Lecture 16 (10/10/2024)

Today: * max flow application: matrix rounding
* residual network
* max flow algorithm

Matrix Rounding Problem

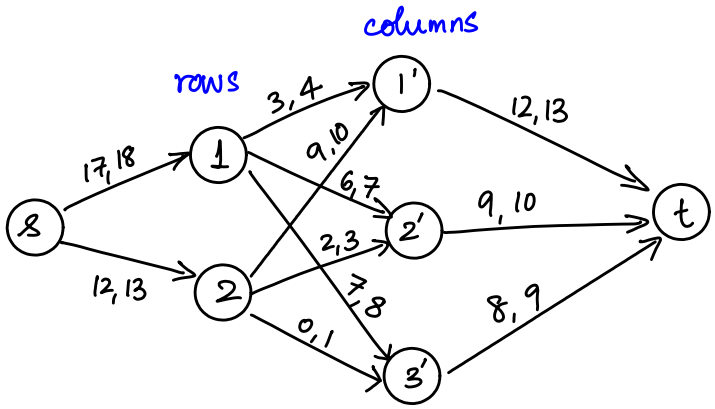
Given $m \times n$ matrix, $D = [d_{ij}]$ row sums α_i $(1 \le i \le m)$, and column sums β_j $(1 \le j \le n)$, where $d_{ij}, \alpha_i, \beta_j \in \mathbb{R}$, the goal is to round each element d_{ij} and row & column sums (α_i, β_j), such that the row sums of the rounded elements equal the rounded row sums, and similarly for column sums. For instance, consider the 2×3 matrix D:

	1'	2'	3'	α_i
1	3.1	6.8	7.3	17.2
2	9.6	2.4	0.8	12.8
β_j	12.7	9.2	8.1	

Note that we can round each number up or down — e.g., 3.1 can be rounded to 3 or to 4.

$(\lfloor 3.1 \rfloor = 3, \lceil 3.1 \rceil = 4)$

We formulate this problem as a max flow problem as follows.



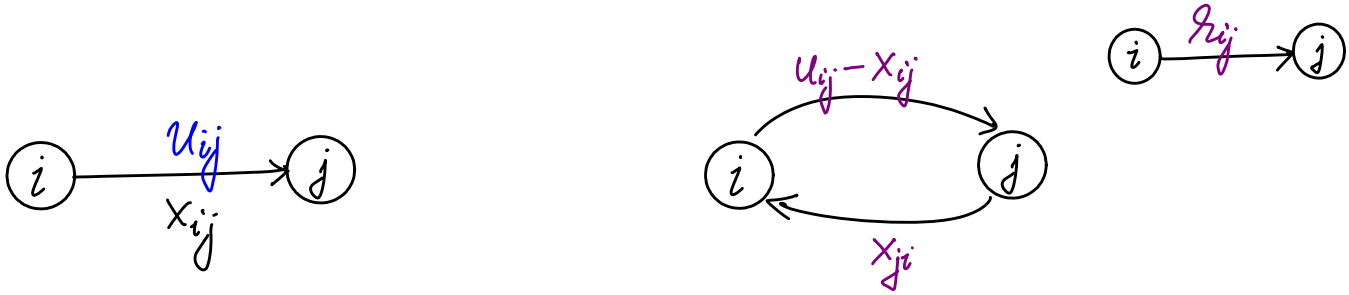
Here,
$N_1 = \{1, \dots, m\}$: row nodes
$N_2 = \{1', 2', \dots, n'\}$: column nodes

We add s, t, arcs (s, i), $i \in N_1$ (row nodes), (j, t) $\forall j \in N_2$ (column nodes), and set $\ell_{si} = \lfloor \alpha_i \rfloor$, $u_{si} = \lceil \alpha_i \rceil$, $\ell_{jt} = \lfloor \beta_j \rfloor$, $u_{jt} = \lceil \beta_j \rceil$. We also have (i, j), $1 \le i \le m$, $1 \le j \le n$, with $\ell_{ij} = \lfloor d_{ij} \rfloor$ (floor), $u_{ij} = \lceil d_{ij} \rceil$ (ceil).

The x_{ij}'s in a max flow will be at l_{ij} or u_{ij}, which are all integers. Further, flow balance at nodes in N_1 and N_2 impose the row and column sums, giving a consistent rounding.

Residual Network ("Remaining flow" network)

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



The diagram shows the components of the residual network corresponding to arc (i,j). Repeat for all arcs $(i,j) \in A$ to obtain $G(\bar{x})$, the residual network for the flow $\bar{x}$.

Notice that the residual network is defined for a particular flow $\bar{x}$.

The **residual capacity**, r_{ij}, of arc (i,j) is given as

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

This is the maximum additional flow we can send from i to j using arcs (i,j) and (j,i).

The residual network corresponding to flow $\bar{x}$ is $G(\bar{x})$, and it contains all (i,j) with $r_{ij} > 0$ on N, the same node set as the input graph G.

Example: Consider the network G with a flow (x_{ij} values) given

G with flow $\bar{x}$ $\qquad$ $G(\bar{x})$

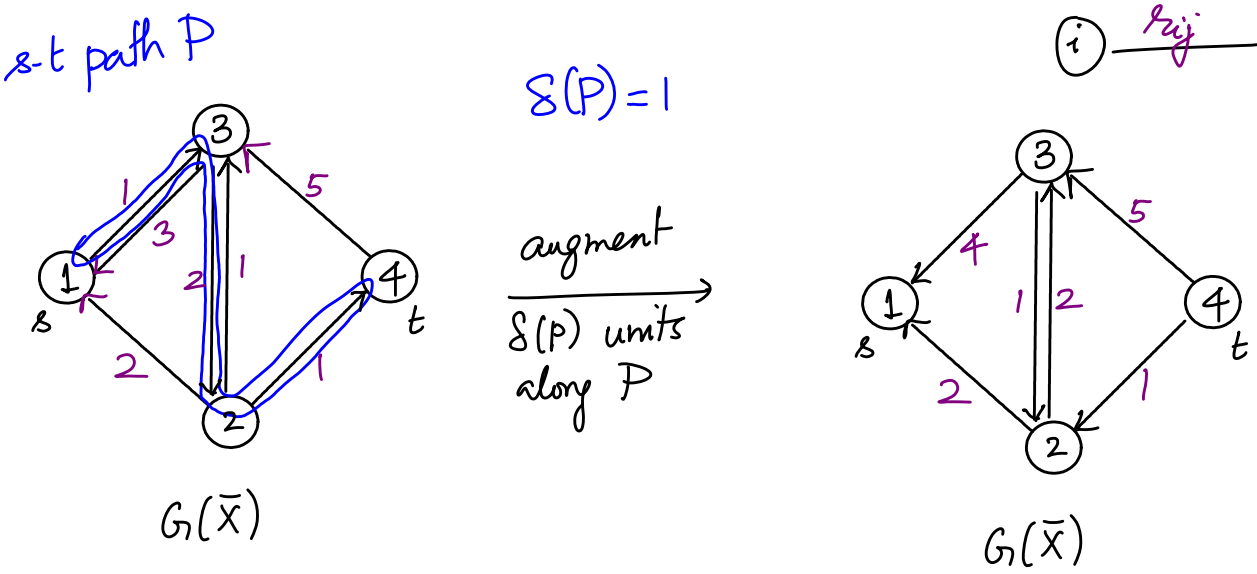
If there is a (directed) path from s to t in $G(\bar{x})$, we could push flow along this path. This idea is central to max flow algorithms — start with some flow $\bar{x}$, find $G(\bar{x})$, and send flow along s-t paths in $G(\bar{x})$. How much flow could we send from s to t along such an s-t path? We formalize these ideas into definitions now.

Def A directed path P from s to t in $G(\bar{x})$ is an **augmenting path**. The **residual capacity** of the augmenting path P is $\delta(P) = \min_{(i,j)\in P} \{r_{ij}\}$.

Note that $\delta(P) > 0$, as $(i,j) \in P \Rightarrow r_{ij} > 0$ by definition.

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

Let's consider the example again.



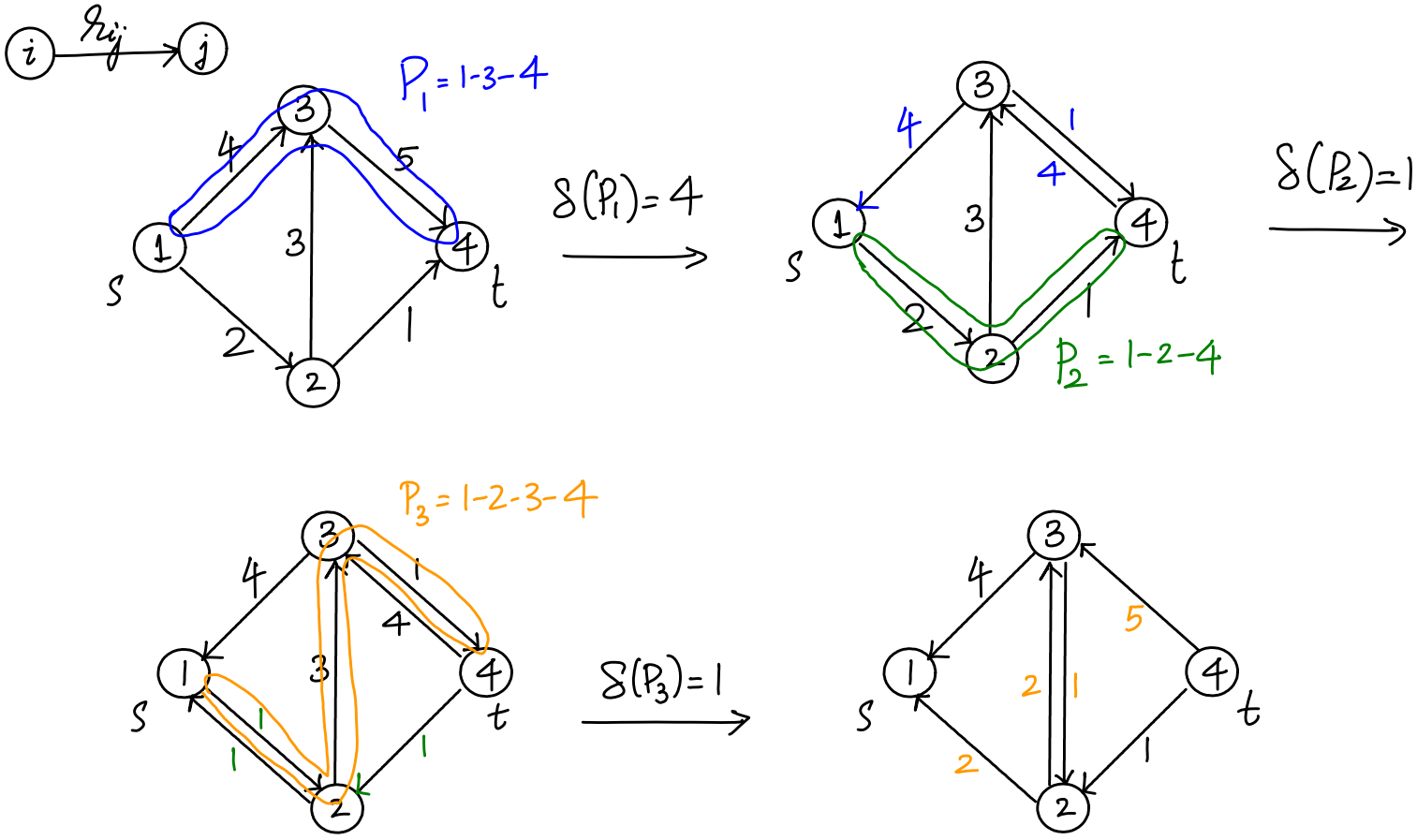
Now there are no more s-t paths in $G(\bar{x})$, which tell us that $\bar{x}$ is optimal!

The Generic Augmenting Path Algorithm (Ford-Fulkerson)

Assume $l_{ij} = 0 \ \forall (i,j) \in A$.

```
begin
    x̄ := 0̄;
    initialize G(x̄);
    while G(x̄) has a path P from s to t do
        augment δ(P) units of flow along P;
        update x̄, G(x̄);
    end_while
end_begin
```

Example We start with $\bar{x}=\bar{0}$, i.e., the zero flow. Notice that $G(\bar{x})=G$ when $\bar{x}=\bar{0}$, assuming all $u_{ij}>0$.

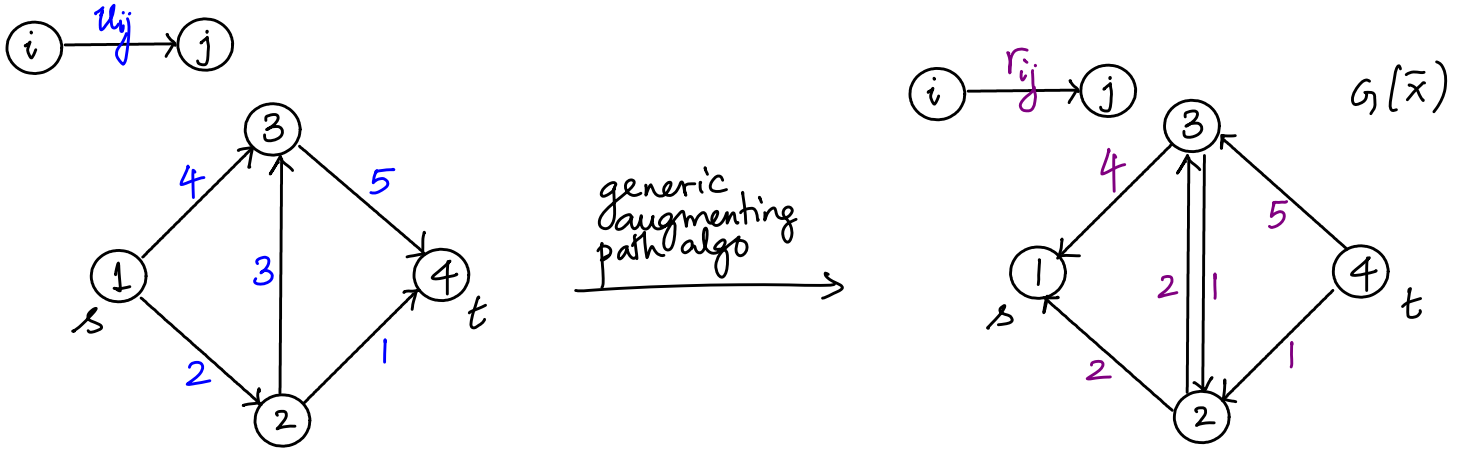


There are no more augmenting paths, and hence the flow is maximum. Notice that the value of the maximum flow is $\delta(P_1)+\delta(P_2)+\delta(P_3)=4+1+1=6$.

MATH 566: Lecture 17 (10/15/2024)

Today: * finiteness of augmenting path algo
* max-flow min-cut theorem (MFMC)

To finish the example on augmenting path algorithm, we consider the following:
How to obtain optimal x_{ij}'s from final $G(\bar{x})$ once the algorithm terminates?



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

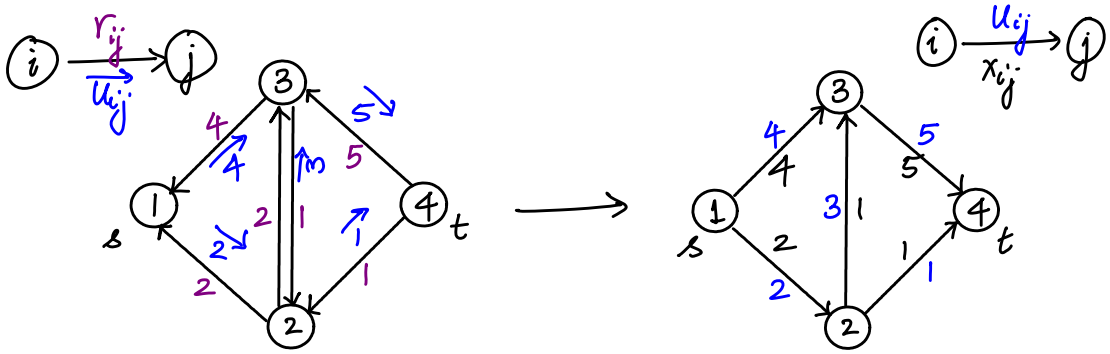
$\Rightarrow x_{ij} - x_{ji} = u_{ij} - r_{ij}$

→ Indeed, only one of x_{ij} and x_{ji} will be > 0, not both!

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

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

($u_{ij} < r_{ij}$)



Notice we obtain $x_{23} = 1$ from $r_{23} = 2$ or, equivalently from $r_{32} = 1$.

Proof of Finiteness of the Generic Augmenting Path algorithm

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

Recall, we assume $u_{ij} \in \mathbb{Z}_{\geq 0}$ (non negative integers). We now assume all u_{ij}'s are also finite. → The more general case when some u_{ij}'s are ∞ can be handled similarly.

1. **Lemma** The residual capacities (r_{ij}) are integral after each iteration.
2. The capacity of each augmenting path is at least 1 (as all $r_{ij} \geq 1$).
3. Augmenting along a path P decreases r_{si} for some arc (s,i) by at least one unit.
4. Augmentation never increases r_{si} for any i.
5. $\sum_{(s,i)\in A} r_{si}$ keeps decreasing, and is lower bounded by zero. $\left(\sum_{(s,i)\in A} r_{si} \geq 0\right)$.

finite!

$\Rightarrow$ The # augmentations $= O(nU)$ where $U = \max_{(i,j)\in A} \{u_{ij}\}$.

In fact, we can say the # augmentations $= O(nU_s)$, where $U_s = \max_{(s,i)\in A} \{u_{si}\}$.

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 when a flow $\bar{x}$ is optimal?

1. No augmenting path in $G(\bar{x})$.
2. Max-flow Min-Cut (MFMC) theorem (duality)

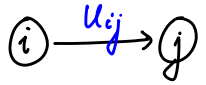
} Max-flow optimality conditions

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

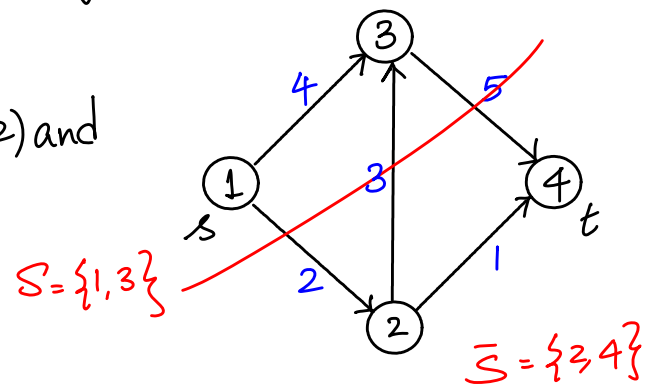
$\Rightarrow (S \cup \bar{S} = N,\ S \cap \bar{S} = \emptyset)$

A **forward arc** is $(i,j) \in A$ with $i \in S$, $j \in \bar{S}$.

A **backward arc** is $(i,j) \in A$ with $i \in \bar{S}$, $j \in S$.



Example For $S = \{1,3\}$, $\bar{S} = \{2,4\}$, (1,2) and (3,4) are forward arcs, and (2,3) is a backward arc.



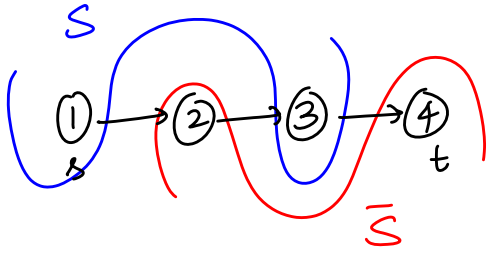
Def The **capacity** of an s-t cut $[S, \bar{S}]$ is

$$u[S,\bar{S}] = \sum_{\substack{i \in S, j \in \bar{S} \\ (i,j) \in A}} u_{ij} \qquad (1)$$

→ sum of all forward arc capacities

e.g., $u[S,\bar{S}] = u_{12} + u_{34} = 2 + 5 = 7$, above

We are not assuming anything about the connectivity of subgraphs defined by S and $\bar{S}$, e.g., the following scenario is perfectly valid:



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

We present two claims, from which the above Property follows.

Def The **flow across a cut** $[S,\bar{S}]$ is

$$F_{\bar{x}}[S,\bar{S}] = \sum_{i\in S, j\in\bar{S}} x_{ij} - \sum_{i\in S, j\in\bar{S}} x_{ji}. \qquad (2)$$

$S_1 = \{1,3\}$

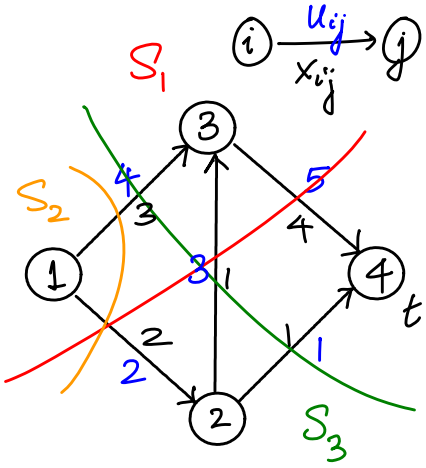
$F_{\bar{x}}[S_1,\bar{S}_1] = 2+4-1 = 5$
$x_{12}+x_{34}-x_{23}$

$S_2 = \{1\}$

$F_{\bar{x}}[S_2,\bar{S}_2] = 3+2 = 5$
$x_{13}+x_{12}$

$S_3 = \{1,2\}$

$F_{\bar{x}}[S_3,\bar{S}_3] = 3+1+1 = 5$
$x_{13}+x_{23}+x_{24}$



Notice $F_{\bar{x}}[S,\bar{S}]$ = value of flow in each case. Indeed, this is not a coincidence!

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

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

Recall, $\sum_{(s,j)\in A} x_{sj} - \sum_{(j,s)\in A} x_{js} = v$ is the equation for $i=s$. Overall, we will be left with flows x_{ij} and $-x_{ji}$ for $i\in S, j\in\bar{S}$, and the flows within S will cancel in $\pm$ pairs. □

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

Proof For $i \in S$, $j \in \bar{S}$, $x_{ij} \le u_{ij}$, and $x_{ji} \ge 0$ hold.

$$\text{So (2)} \Rightarrow \quad F_{\bar{x}}[S,\bar{S}] = \sum x_{ij} - \sum x_{ji} \le \sum_{i\in S, j\in\bar{S}} u_{ij} - \sum 0$$

$$= u[S,\bar{S}], \text{ by (1).} \qquad \square$$

Note that Property 6.1 follows directly from Claims 1 and 2 above.

The Max-Flow Min-Cut Theorem (MFMC)

We state a more general theorem, from which MFMC follows as a corollary.

$= * \;\Big|\; u[S,\bar{S}]$ ↓ min cut — optimal — ↑ v max flow

Theorem 6.3 (Optimality Conditions for Max flow)

The following statements are equivalent.

(1) A flow $\bar{x}$ is maximum. → the m-vector of arc flows x_{ij}

(2) There is no augmenting path in $G(\bar{x})$.

(3) There is an s-t cut $[S,\bar{S}]$ whose capacity is equal to the value of the flow $\bar{x}$, i.e., $u[S,\bar{S}] = v$.

Corollary 6.4 (MFMC Theorem) The maximum flow value is equal to the minimum capacity of an s-t cut.

Proof (of Theorem 6.3) (1) $\Rightarrow$ (2). We argue $\neg$(2) $\Rightarrow$ $\neg$(1)

"negation" or "not"

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

(3) $\Rightarrow$ (1)

By (3), we have $v = u[S, \bar{S}]$.

By Claim (1), $v = F_{\bar{x}}[S, \bar{S}]$. Hence $F_{\bar{x}}[S, \bar{S}] = u[S, \bar{S}]$.

By Claim (2), $F_{\bar{x}}[S, \bar{S}] \leq u[S, \bar{S}]$. Hence we have optimality.

(2) $\Rightarrow$ (3) There is no augmenting path in $G(\bar{x})$.

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

$\bar{S} = N \setminus S$.

$\Rightarrow$ There is no arc in $G(\bar{x})$ from S to $\bar{S}$.

forward arcs are all full, and backward arcs have no flow.

So, $i \in S, j \in \bar{S}$ $(r_{ij} = 0) \Rightarrow x_{ij} = u_{ij}$, and

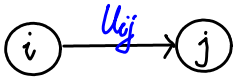
$i \in S, j \in \bar{S}$ $(r_{ji} = u_{ji}) \Rightarrow x_{ji} = 0$.

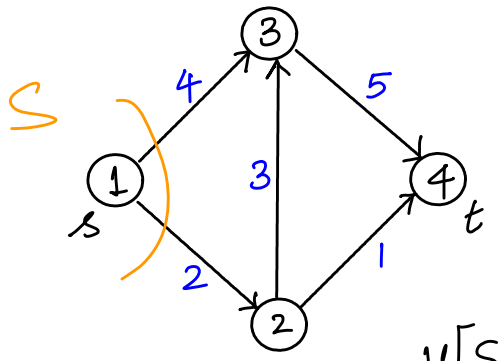
$$\Rightarrow F_{\bar{x}}[S, \bar{S}] = \sum_{i \in S, j \in \bar{S}} x_{ij} - \sum_{i \in S, j \in \bar{S}} x_{ji}$$

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

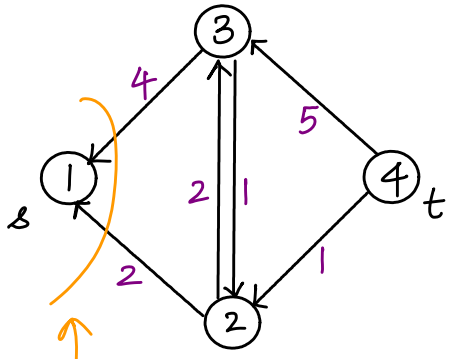
□

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





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



The only node reachable from s here is s itself.

MATH 566 : Lecture 18 (10/17/2024)

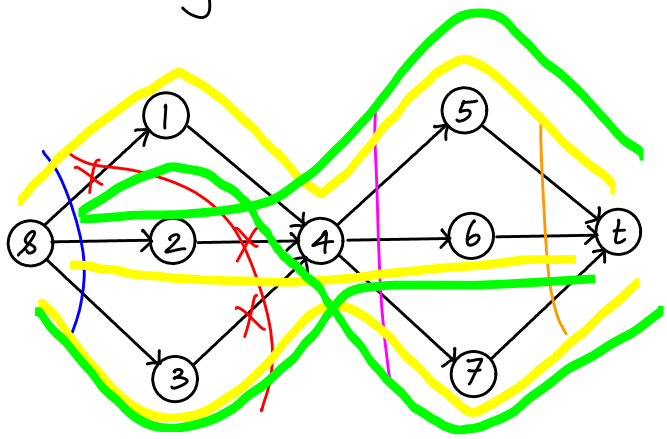
Today: * application of MFMC in network reliability
* augmenting path algorithm

Applications of MFMC in Network Reliability

In a communication network, we might want to have more than one s-t path to send info/packets. We want to count the number of arc-disjoint s-t paths in G. Same scenario arises in road networks too.

Q. what is the maximum number of arc-disjoint paths from s to t?

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



There are 3 arc-disjoint paths (2 sets of 3 such paths are highlighted).

Several min-cuts are also shown.

Theorem The maximum number of arc-disjoint s-t paths in a directed network G is equal to the minimum number of arcs up on whose removal there is no directed s-t path.

Proof Set $u_{ij} = 1 \ \forall (i,j) \in A$, and apply MFMC theorem! □

It is much harder to prove this result from first principles, or using other techniques. But it follows directly from the application of MFMC!

We consider also a stronger notion of independence of paths — being node-disjoint.

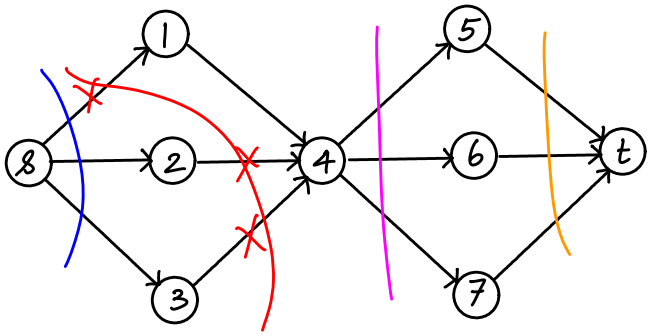
Node-disjoint Paths

Def Two s-t paths are **node-disjoint** if the only nodes they share are s and t.

There are no groups of node-disjoint s-t paths in the previous example except for isolated s-t paths. Notice that each s-t path goes through node 4.

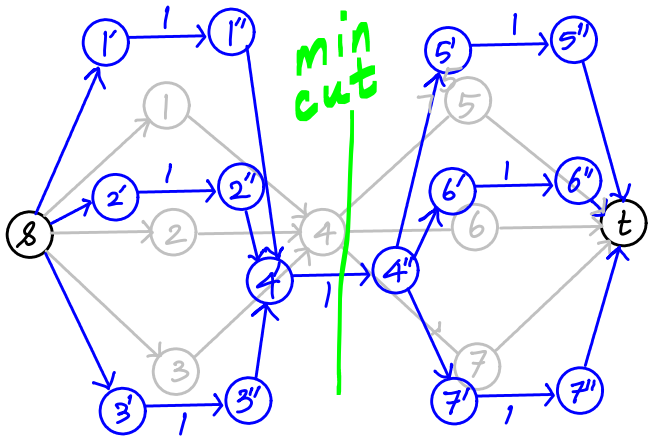
Theorem Let G have not no (s,t) arc. The maximum number of node-disjoint s-t paths is equal to the minimum number of nodes whose removal leaves no directed s-t paths.

Proof Can use node-splitting. Removing node i is then equivalent to removing arc (i', i''). Set $u_{i',i''} = 1$, and apply MFMC. Set $u_{ij} = \infty$ for all other arcs (i,j). □



Here, the min # nodes whose removal disconnects s and t is 1 — remove node 4.

Equivalently, the # node-disjoint s-t paths is also 1.



We now return to algorithms for max flow. We first describe polynomial implementations of the augmenting path algorithm.

The Generic Augmenting Path Algorithm (Ford-Fulkerson)

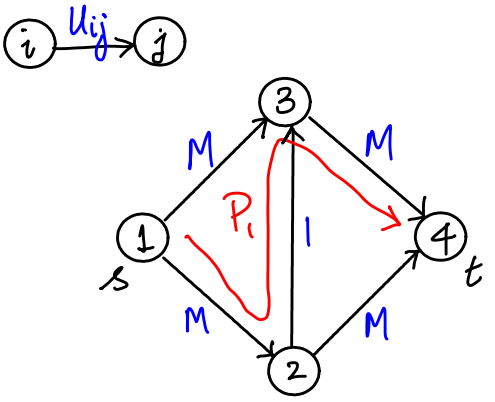
Assume $l_{ij} = 0 \ \forall (i,j) \in A$.

```
begin
    x̄ := 0̄;
    initialize G(x̄);
    while G(x̄) has a path P from s to t do
        augment δ(P) units of flow along P;
        update x̄, G(x̄);
    end_while
end_begin
```

We saw this algorithm terminates in $O(nU)$ augmentations. In the worst case, we perform updates on every arc following each augmentation. Hence the algorithm runs in $O(mnU)$ time.

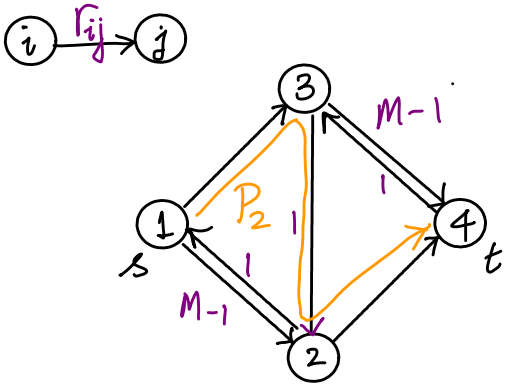
A pathological example

Consider the network on the right.



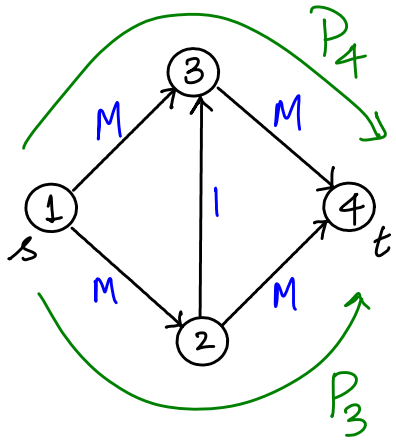
We start with $\bar{x} = \bar{0}$, and can choose $P_1 = 1\text{-}2\text{-}3\text{-}4$, with $\delta(P_1) = 1$, and augment along P_1.

Then we can choose $P_2 = 1\text{-}3\text{-}2\text{-}4$ with $\delta(P_2) = 1$, and augment. Then pick P_1, and alternately P_2, and so on. In this way, we will find the max-flow in $2M$ augmentations.



But we could have found the max flow in just 2 augmentations by choosing $P_3 = 1\text{-}2\text{-}4$, $\delta(P_3) = M$ and $P_4 = 1\text{-}3\text{-}4$, $\delta(P_4) = M$.

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



We need to select augmenting paths carefully!

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

We will look at option 2 first.

Shortest Augmenting Path (SAP) Algorithm

* Use distance labels $d(i)$; time for each augmentation: $O(n)$
* total number of augmentations : $O(mn)$

  Total complexity : $O(n^2 m)$.

Def (Distance labels)

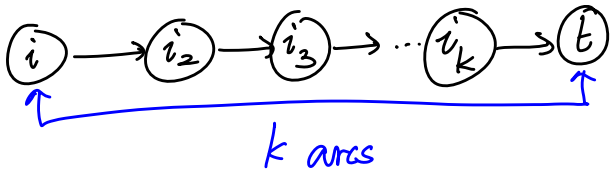
A set of distance labels $d: N \to \mathbb{Z}^n_{\geq 0}$ is **valid** with respect to flow $\bar{x}$ if

(1) $d(t) = 0$, and

(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 terms of # arcs) of the SP from i to t in $G_1(\bar{x})$.

Proof We add the validity conditions for all arcs in any i-t path P, say, with k arcs.



We get

$$d(i) \leq d(t) + k = k \quad (\text{as } d(t)=0).$$

This result holds for all i-t paths, and hence $d(i)$ is a lower bound. □

Property 7.2 If $d(s) \geq n$, there is no directed s-t path in $G_1(\bar{x})$.

Intuition for $d(i)$: Think of $d(i)$ as the height above ground level that node i has to be raised for flow to happen "freely". Node t is at ground level ($d(t)=0$), and s need not be raised above level $n-1$ from the ground.

We now look for candidate arcs in $G_1(\bar{x})$ that could be part of augmenting path(s). Recall the notion of admissible arcs in search (BFS/DFS) — we redefine admissibility here.

Def An arc $(i,j) \in G(\bar{x})$ is **admissible** if $d(i) = d(j)+1$.

A path from s to t consisting of only admissible arcs is an **admissible path**.

MATH 566: Lecture 19 (10/22/2024)

Today: * Shortest augmenting path algorithm

Recall Distance labels $d(\cdot)$ are valid if $d(i) \le d(j)+1 \ \forall (i,j) \in G_f(\bar{x})$, $d(t)=0$.

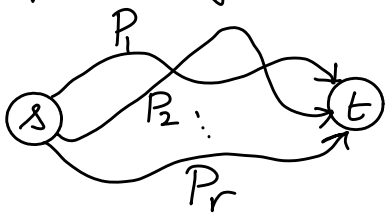
We now look for candidate arcs in $G_f(\bar{x})$ that could be part of augmenting path(s). Recall the notion of admissible arcs in search (BFS/DFS) — we redefine admissibility here.

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

We formalize the notion that admissible paths are precisely augmenting paths.

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

Proof Add admissibility conditions over all arcs in P. We get $d(s)=k$, where P has k arcs. Since $d(s)$ provides a lower bound on the length of arcs (in terms of # arcs) by Property 7.1, equality here implies optimality. □

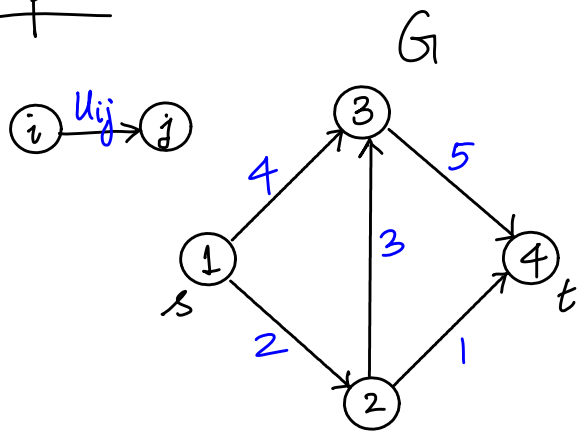


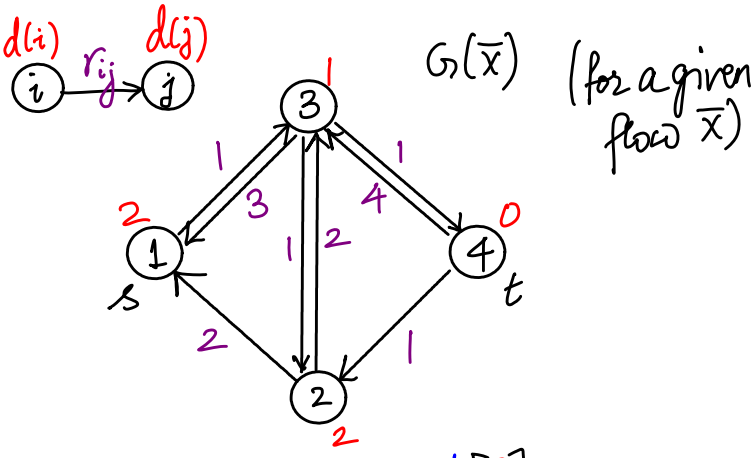
$$\left.\begin{array}{l} d(s) \le k_1 \\ d(s) \le k_2 \\ \vdots \end{array}\right\} \Rightarrow d(s) \le k = \min_{i=1\dots r}\{k_i\}$$

We need one more definition before looking at some examples.

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

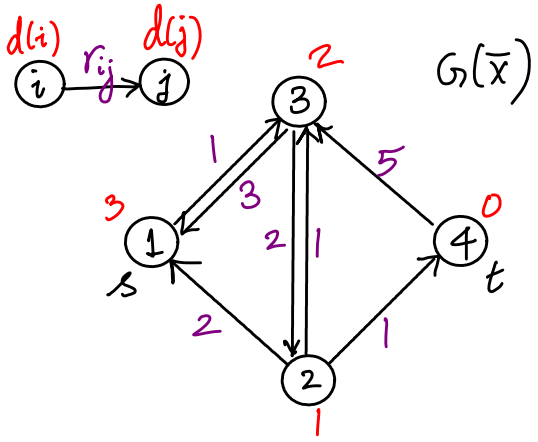
Examples



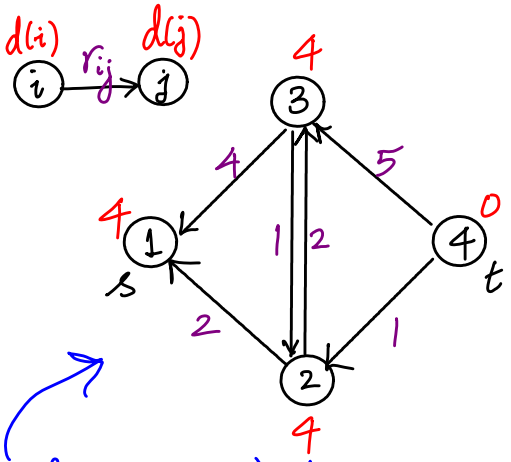


Note that $\bar{d} = \begin{matrix}1\\2\\3\\4\end{matrix}\begin{bmatrix}0\\0\\0\\0\end{bmatrix}$ is valid, but is not exact, while $\bar{d} = \begin{matrix}1\\2\\3\\4\end{matrix}\begin{bmatrix}2\\2\\1\\0\end{bmatrix}$ is exact.

Let's consider a couple more flows (on the same network):



$\bar{d} = \begin{matrix}1\\2\\3\\4\end{matrix}\begin{bmatrix}3\\1\\2\\0\end{bmatrix}$ is exact.



This is $G(\bar{x})$ for the max flow.

Here, there is no directed path from any $i \neq t$ to t in $G(\bar{x})$. Setting $d(i) = n$ is sufficient (rather than setting $d(i) = \infty$), as we know the largest meaningful value for $d(i)$ is $n-1$.

If $d(i)$ goes above n at any point of the SAP algorithm, we can ignore node i from further consideration.

Recall! Intuition for $d(i)$: Think of $d(i)$ as the height above ground level that node i has to be raised for flow to happen "freely". Node t is at ground level ($d(t)=0$) and s need not be raised above level $n-1$ from 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;
```

⇒ $G(\bar{x}) = G$ at start

→ reverse BFS to t

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;
```

→ look at $A(i)$ in $G(\bar{x})$

→ $d(i) = d(j)+1$ for $(i,j) \in G(\bar{x})$

```
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 $G(\bar{x})$

→ $d(i) < d(j)+1 \ \forall (i,j) \in G(\bar{x})$ (as $d(i) \le d(j)+1$ by validity, and no admissible arc ⇒ $d(i) = d(j)+1$ does not hold)

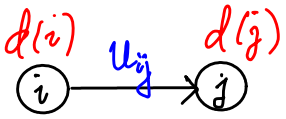
relabel operation

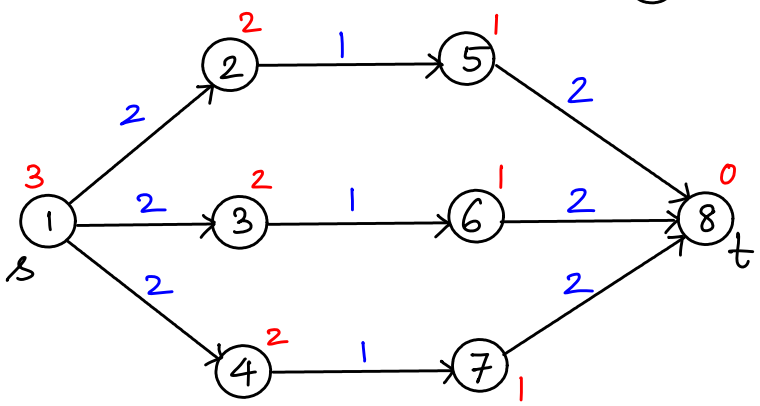
This is the one step where we update $d(i)$. The $d(i)$ value strictly increases by a relabel.

```
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;
```

→ also update $G(\bar{x})$

Illustration

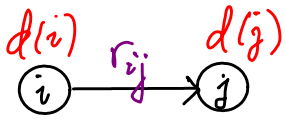


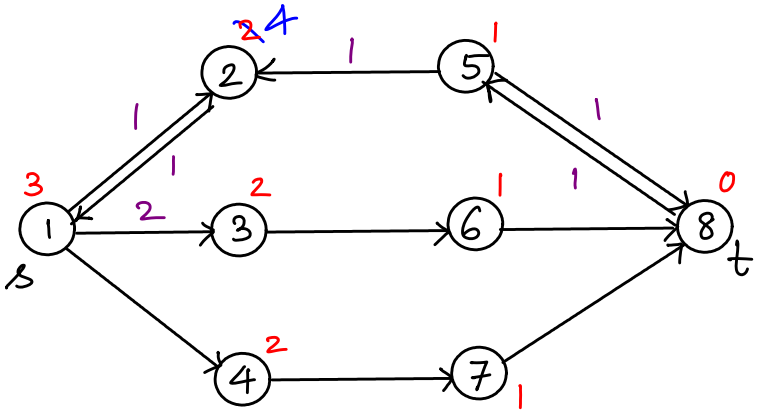


We examine outarcs $(A(i)$ in $G(\bar{X}))$ in lexicographic order, by default.

Iteration 1

$P_1 = 1\text{-}2\text{-}5\text{-}8$, augment $\delta(P_1)=1$ unit. In detail, we advance from $s=1$ to 2 to 5 to $8=t$, identifying the augmenting path P_1.





Iteration 2

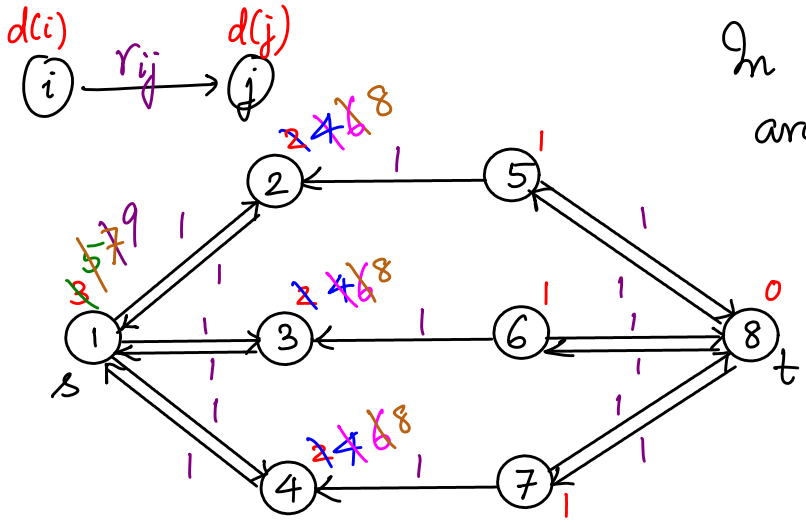
Advance from $s=1$ to 2. But there are no admissible arcs out of 2. So we relabel $d(2)$ to $d(1)+1=4$, and retreat back to $s=1$.

Iteration 3 $(1,2)$ is no longer admissible, but $(1,3)$ is. We advance to t via $P_2 = 1\text{-}3\text{-}6\text{-}8$, augment $\delta(P_2)=1$, as we did for P_1.

In Iteration 4, we advance along $(1,3)$, and then relabel 3 before retreating back to $s=1$.

In Iteration 5, we repeat steps similar to Iteration 3 to augment along $P_3 = 1\text{-}4\text{-}7\text{-}8$ $(\delta(P_3)=1)$.

In Iteration 6, we advance on $(1,4)$, relabel 4, and retreat to $s=1$.



In Iteration 7, there are no admissible arcs out of $s=1$. So, relabel node $s=1$:

$$d(1) = \min_{j=2,3,4} \{d(j)+1\} = 5.$$

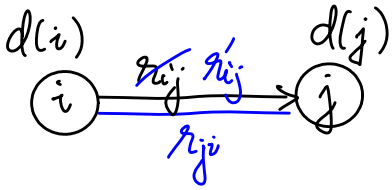
We could have stopped after Iteration 6, but that's just because of the special structure of the network. The algorithm relabels $s=1$ and 2, 3, 4 nodes using a series of retreat operations until $d(s) = 9 > n = 8$, when it terminates.

Proof of Correctness

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

Augmentation Bottleneck arc(s) disappear from $G(\bar{x})$, we decrease some r_{ij}, and we add (j,i) to $G(\bar{x})$ (with $r_{ji} > 0$).

Consider the more general situation where we push some flow forward along arc (i,j), but not saturate it. Thus, r_{ij} is decreased to $r'_{ij} < r_{ij}$, and (j,i) is added to $G(\bar{x})$.



$d(i) = d(j)+1$, as $(i,j) \in G(\bar{x})$ is an admissible arc.

For (i,j), validity is maintained ($r'_{ij} < r_{ij}$).

For (j,i), we need $d(j) \le d(i)+1$. But this condition holds, as $d(j) = d(i)-1$, and hence satisfies the validity condition.

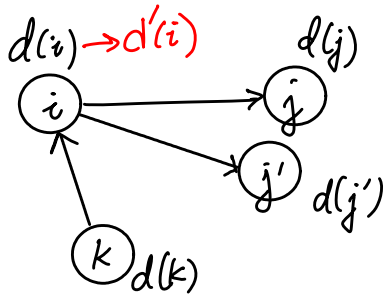
MATH 566 : Lecture 20 (10/24/2024)

Today: * Complexity of SAP algorithm
* Capacity scaling algorithm

Correctness of SAP Algorithm (continued)

* We saw that $d(i)$'s remain valid after augmentations.
* Relabeling

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



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

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

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

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

Hence the relabeling strictly increases $d(i)$, and

$d'(i) \leq d(j)+1$ still holds for all $(i,j) \in G(\bar{x})$.

Now Consider $(k,i) \in G(\bar{x})$. $d(k) \leq d(i)+1$ previously. ($d(i)$ increased to $d'(i)$)

Since we strictly increased $d(i)$ by relabeling, $d(k) \leq d'(i)+1$ holds as well. □

Note that while we start with exact distance labels, we do not strive to maintain exactness at each step — we make sure labels remain valid after each iteration. In particular, we do not recalculate exact distance labels after each augmentation. The exactness of distance labels would hold at the end, though.

Complexity of Shortest Augmenting Path Algorithm

Outline (Corresponding results from AMO listed 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 = $O(n)$
   ↳ each augmenting path has at most $n-1$ arcs

   $\Rightarrow$ Total time for augmentations = $O(n^2m)$ — Theorem 7.10

Recall that we might advance and retreat several times before finding each augmenting path. We need to count all that effort!

3. Time Spent looking for augmentations

   Total number of retreat operations = $O(n^2)$ (= number of relabels)

$\Rightarrow$ Total number of advance operations = $O(n^2 + n^2m)$

↳ Each retreat operation cancels out a previous advance operation. We end back at $i = s$ (when $d(s) \geq n$).

We will now discuss details of each result.

Details of Complexity Analysis of SAP Algorithm

1. $d(i) \le n-1 \quad \forall i \in N$.

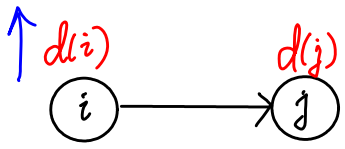
After each relabel, $d(i)$ strictly increases. Hence we relabel node i at most n times.

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

n nodes, $\le n$ relabels per node

Each arc in $A(i)$ is examined once per relabel.

If (i,j) is inadmissible, $d(i) < d(j)+1$, and it remains inadmissible until i is relabeled.



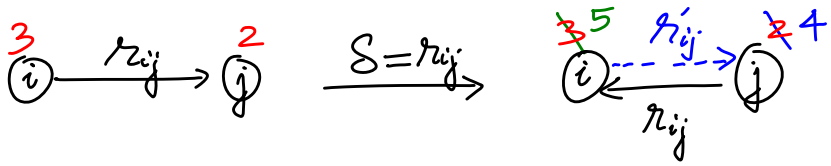
Total time for all relabels $= O(n \sum_i |A(i)|) = O(nm)$.

We might conclude that the time for all relabels is $O(n^2 m)$, counting it as (#nodes) × (max #relabels) × (# arcs per relabel) $= O(n \times n \times m)$. But we can be more careful in counting. Let r_i be the number of times node i is relabeled. For each relabel, the # arcs involved is $|A(i)|$. Hence the total time for all relabels is

$$O\left(\sum_{i \in N} r_i |A(i)|\right) = O\left(\sum_{i \in N} n |A(i)|\right) = O\left(n \sum_{i \in N} |A(i)|\right) = O(nm).$$

This result is given as AMO Property 7.7 with $r_i = k$.

2. Consider the following example.



Initially, $d(i) = 3 = d(j)+1 = 2+1$, and (i,j) is admissible. We then have to push flow backward, i.e., along (j,i), for which $d(j)$ has to be increased to $d(i)+1 = 4$. Subsequently, $d(i)$ is increased to the new $d(j)+1 = 4+1 = 5$, so that (i,j) can be saturated again.

Let the algorithm saturate (i,j) in one 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. We know

$d(i) \leq n$ ($n-1$ to be exact), and hence

(i,j) can be saturated at most $\frac{n}{2}$ times, i.e., $O(n)$ times.

There are m arcs in total, and each augmentation will saturate at least one arc.

$\Rightarrow$ total # augmentations $= O(mn)$ (# arcs; # saturations of one arc)

Time for each augmentation $= O(n)$ $\rightarrow$ $\leq n-1$ arcs per augmenting path

$\Rightarrow$ Total time for all augmentations $= O(n \times mn) = O(n^2 m)$.

3. 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 $= O(n^2 m)$

$O(nm)$ augmentations $\times$ $O(n)$ arcs per s-t path

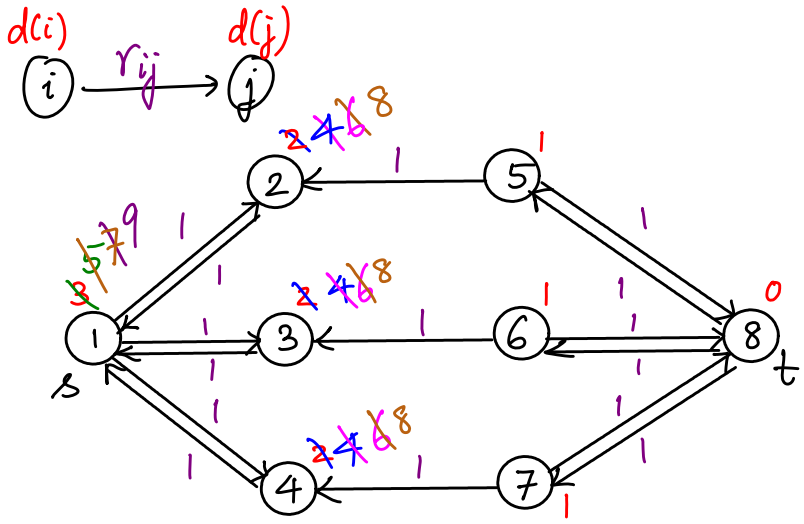
$\Rightarrow$ Total # advances $= O(n^2 + n^2 m) = O(n^2 m)$

n^2: # retreats; $n^2 m$: # 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. In other words, we make as many advances as the number of retreats, in addition to the advances associated with augmentations.

$\Rightarrow$ Overall running time $= O(n^2 m)$. □

We could implement practical improvements. 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.



Here, we could try to identify a min cut $[S, \bar{S}]$, with $S = \{1, 2, 3, 4\}$ as soon as $d(4)$ is relabeled from 2 to 4.

Capacity Scaling Algorithm

IDEA: Motivated by the "largest capacity" augmenting paths example, we augment along paths with "sufficiently large" capacity, rather than finding the largest capacity path.

Def (Δ-**residual network**) For any fixed $\Delta \geq 0$, we let

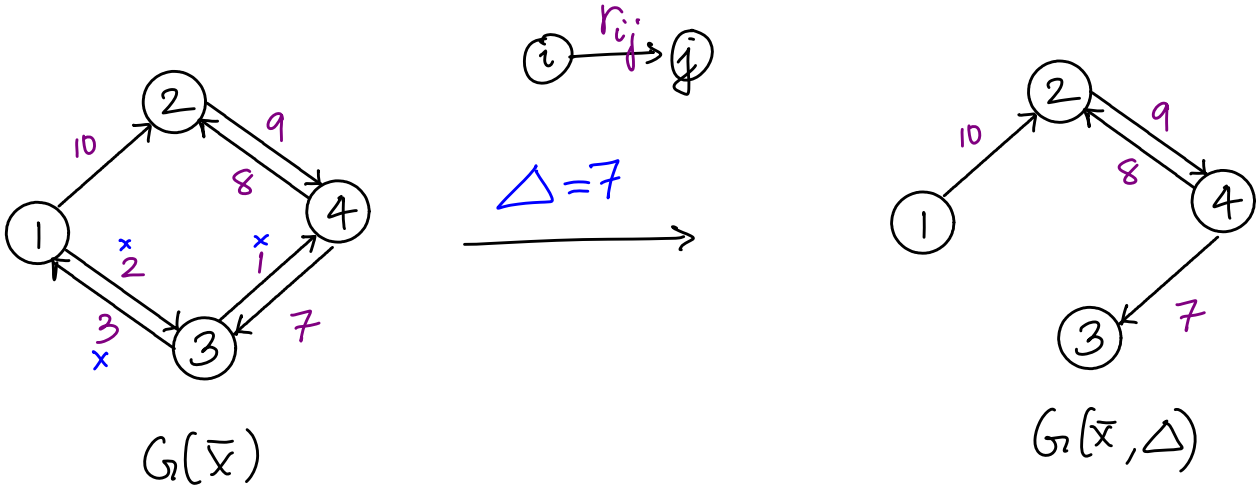
$$G(\bar{x}, \Delta) = \{(i,j) \in G(\bar{x}) \mid r_{ij} \geq \Delta\}.$$

→ arcs in $G(\bar{x})$ with residual capacity $\geq \Delta$.

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

Note that a flow $\bar{x}$ is maximum if it is 1-maximum, i.e., for $\Delta = 1$.

Here is an example.



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

The idea is to push flow along paths with capacity at least Δ for a large value of Δ, and then scale Δ by $\frac{1}{2}$ repeatedly, until we get to $\Delta = 1$.

Here is the 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;
```

→ largest power of 2 $\le U$

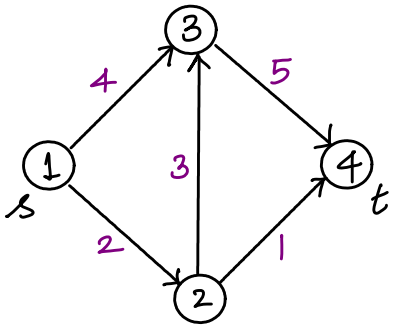
} Δ-phase (one scaling phase)

Figure 7.3 Capacity scaling algorithm.

Example

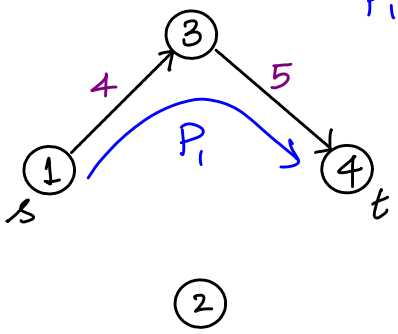
We start with $\bar{x} = \bar{0}$.

$G(\bar{x})$ $\quad$ $(i) \xrightarrow{r_{ij}} (j)$



$\Delta = 4$ →

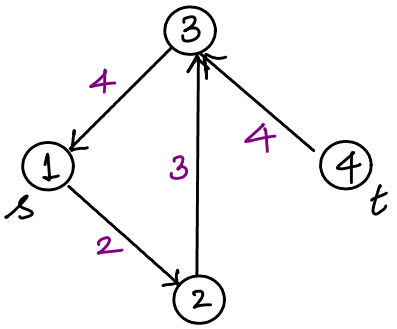
$G(\bar{x}, \Delta)$



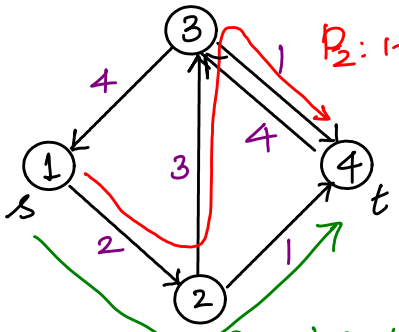
$P_1 = 1\text{-}3\text{-}4$, $\delta(P_1) = 4$

No more s-t paths. ⇒ $\Delta \leftarrow \frac{\Delta}{2}$, i.e., $\Delta = 2$.

$\Delta = 2$ →



→ $\Delta = 1$



$P_3 = 1\text{-}2\text{-}4$

$\delta(P_3) = 1$

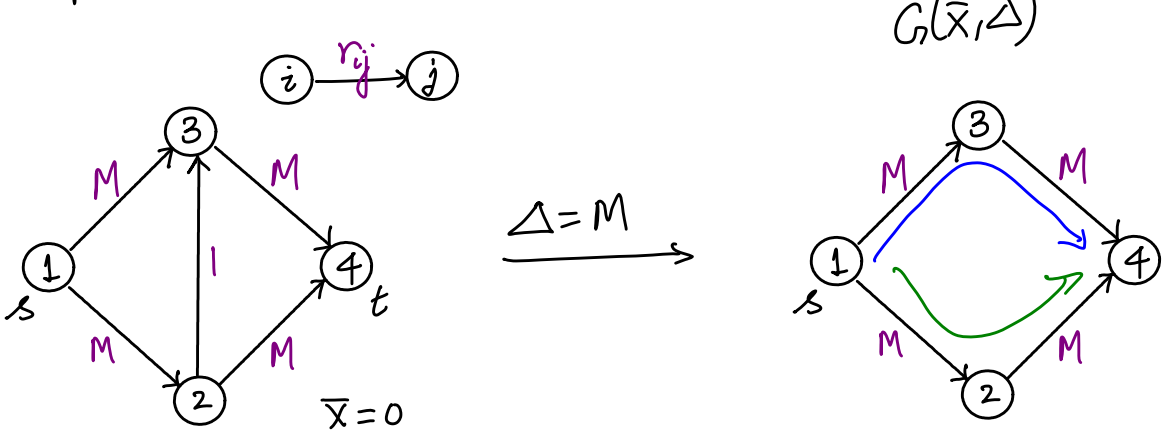
There is no s-t path in $G(\bar{x}, \Delta)$, so $\Delta \leftarrow \frac{\Delta}{2}$, i.e., $\Delta = 1$.

We find two s-t paths, and get no more s-t paths after augmenting along P_2 and P_3, i.e., we have a max-flow.

MATH 566: Lecture 21 (10/29/2024)

Today: * Complexity of capacity scaling algorithm
* Preflow-Push algorithm

Capacity Scaling on pathological example



We find max flow in the first scaling phase itself, after augmenting along two paths of residual capacity equal to $\Delta = M$.

Complexity

* Initially, $\Delta \leq U$.
* Δ is halved after each scaling phase
* Stop when $\Delta < 1$.

$\Rightarrow O(\log U)$ scaling phases.

* Each scaling phase has $O(m)$ augmentations
  - each augmentation saturates at least one arc.
  - can find augmenting path using search ($O(m)$).

  Time per augmentation $= O(n)$ → at most $n-1$ arcs in an augmenting path

$\Rightarrow$ Time per scaling phase $= O(m(m+n)) = O(m^2)$

$\Rightarrow$ Overall running time $= \underline{O(m^2 \log U)}$.

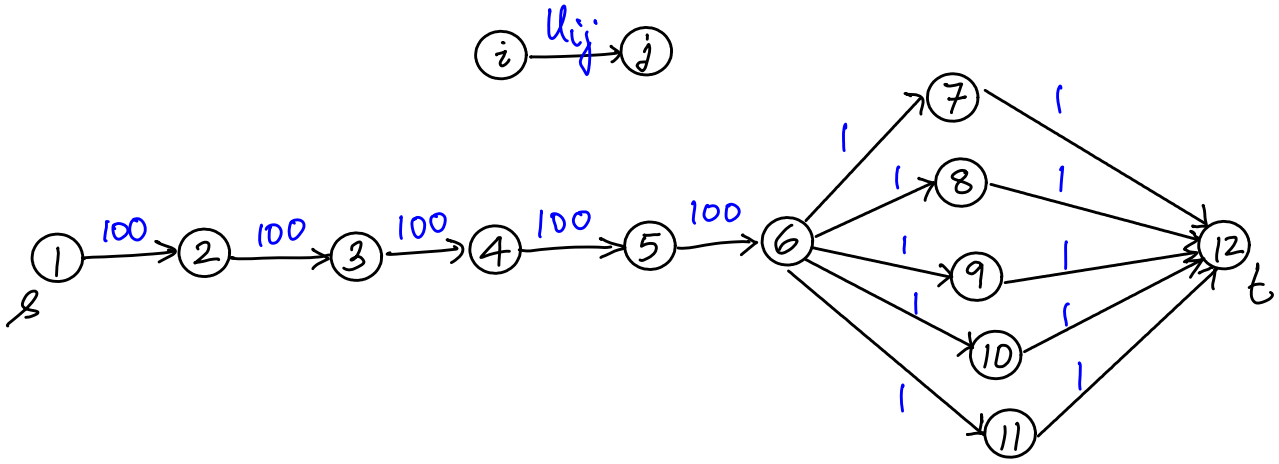
AMO Theorem 7.4

→ Could improve to $O(mn \log U)$ using d(i)'s: combine SAP ideas with capacity scaling.

Compare this worst-case complexity with that of the shortest augmenting path algorithm — $O(n^2m)$. When U is small, the $\log U$ factor could be much smaller than n, the capacity scaling algorithm could be more efficient than the SAP algorithm.

Preflow-Push Algorithms

Here is another pathological instance.



The maximum flow has value $v=5$ here. But any augmenting path algorithm would push 1 unit of flow from s to node 6 repeatedly before pushing it to t along one of the five are-disjoint paths from 6 to t. It would be better if we could push 5 units of flow from s to 6 at start, and then push this 5 units out to t along the disjoint paths in the end.

To push "excess" flow in this fashion, we allow flow balance constraints to be violated in the intermediate steps.

In a bigger network, it may not be feasible/easy to identify that the total capacity from ⑥ to ⑫ is actually 5. We may want to be greedy and push 100 units from ⑤ to ⑥. Then we'll have to push back the excess 95 units from ⑥ back to ①!

Preflow: At **intermediate stages**, we permit more inflow than outflow at nodes other than s and t. Eventually, we want flow balance to hold at all nodes.

Def A **preflow** is $\bar{x} \in \mathbb{R}^m_{\geq 0}$ (x_{ij} for all $(i,j) \in A$) such that $\bar{0} \leq \bar{x} \leq \bar{u}$ ($0 \leq x_{ij} \leq u_{ij}$) and

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

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

$e(i)$ is the **excess flow** at node i, defined as inflow − outflow.
We do not consider $e(s)$ and $e(t)$.

Def A node $i \in N \setminus \{s,t\}$ with $e(i) > 0$ is an **active node**.
s and t are never active.

IDEA: Push flow from active nodes toward t using $d(i)$'s.
Try to achieve flow balance at all nodes except s and t.
If there is excess flow left in intermediate nodes, then push that flow back to s.

Recall: distance labels $d(i)$

* valid: $d(i) \leq d(j)+1 \quad \forall (i,j) \in G(\bar{x})$
* exact: $d(i)$ = SP distance from i to t in $G(\bar{x})$ (w.r.t. # arcs)
* admissible: $(i,j) \in G(\bar{x})$ is admissible if $d(i) = d(j)+1$ (and $r_{ij} > 0$ by definition)

Generic Preflow-Push Algorithm (Goldberg-Tarjan)

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

↳ node examination

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;
```

↳ reverse BFS to t in $G(\bar{x})$

↳ create active nodes $\{j \mid (s,j) \in A\}$

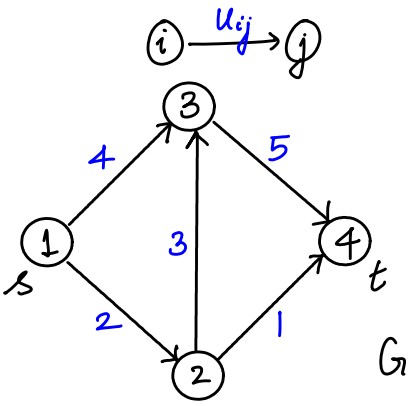
```
procedure push/relabel(i);
begin
    if the network contains an admissible arc (i, j) then
        push δ := min{e(i), r_ij} units of flow from node i to node j
    else replace d(i) by min{d(j) + 1 : (i, j) ∈ A(i) and r_ij > 0};
end;
```

↗ $d(i) = d(j)+1$, $r_{ij} > 0$

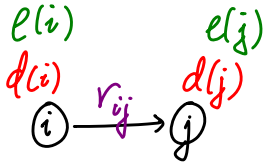
↳ relabel

Recall pipe network analogy: first raise node s to level n (above ground) and push as much flow "downstream" as possible in the pipes going out of s. We keep pushing flow "downhill" from nodes that have excess flow. If there is excess in an intermediate node but no "downhill" pipe, then raise that node. The new "downhill" pipes might take flow back toward s. Keep pushing one arc at a time till all excess in intermediate nodes is cleared.

Example

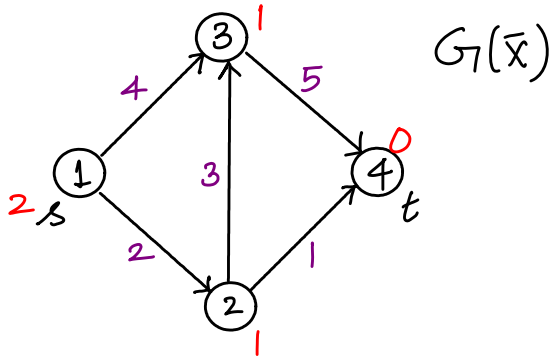


Let's consider our favorite instance.

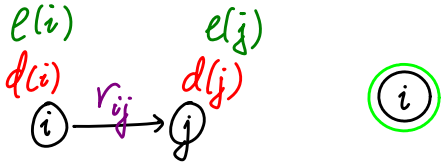


Preprocessing

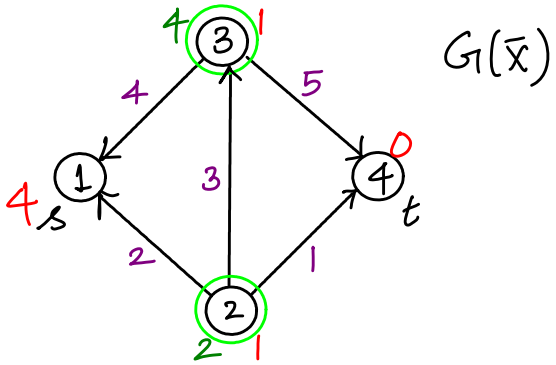
Start with $\bar{x}=\bar{0}$, find exact $d(i)$'s in $G(\bar{x})$.



Saturate all (s,j), and set $d(s)=n$.

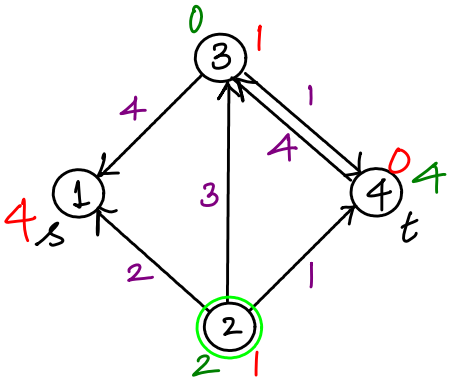


$\textcircled{i}$: i is active



Nodes 2, 3 are active now.

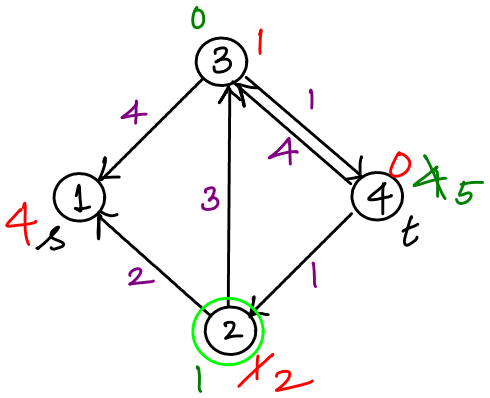
Let's choose 3. Arc (3,4) is admissible, as $d(3)=1=d(4)+1=0+1$. We push $\delta=\min\{e(3), r_{34}\}=\min\{4,5\}=4$.



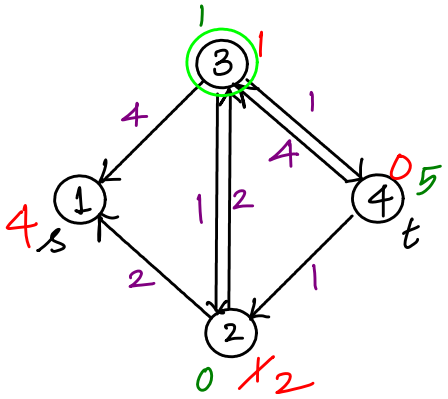
3 is not active, but 2 is still active.

pick node 2, (2,4) is admissible.

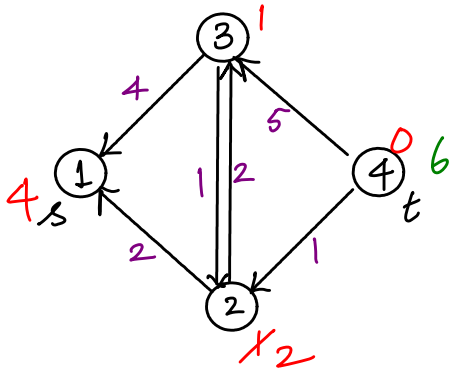
We push $\delta=\min\{e_2, r_{24}\}=\min\{2,1\}=1$.



2 is still active, but has no admissible arcs. So we relabel node 2 to $\min\{d(3), d(1)\} + 1 = 1 + 1 = 2$. After relabel, (2,3) is now admissible, so we push $\delta = \min\{e(2), r_{23}\} = \min\{1, 3\} = 1$.



2 is no longer active, but 3 is. (3,4) is admissible, so we push $\delta = \min\{e(3), r_{34}\} = \min\{1, 1\} = 1$.



There are no more active nodes. So flow is maximum, with value

$v = 5+1 = 4+2 = 6$.

coming into t — going out of s

Complexity of generic preflow push algorithm $= O(n^2 m)$ — same as that of SAP algorithm (see AMO for details).

MATH 566: Lecture 22 (10/31/2024)

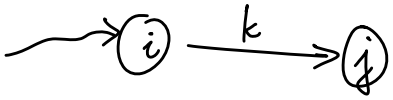
Today: * FIFO Preflow-Push algorithm
* Excess Scaling algorithm

Implementation of SAP algorithm

May be easier to work with x_{ij}'s in G itself, rather than $G(\bar{x})$.

Start with $\bar{x}=\bar{0}$;

Use arc-predecessor indices (apart from usual pred). Here, predA(j)=k means the arc leading into node j in your path is $k \in \{1,2,\ldots,m\}$.

So $H(k)=j$ (head of arc k is j). 

When working with G (and $\bar{x}$), we need to use both A{i} and AI{i} for candidate admissible arcs out of node i. Check them separately.

```
jj = A{i};
admjj = jj(find(x(jj) < u(jj)));
admjj = admjj(find(d(H(admjj))+1 == d(i)));
```
} outarcs

```
ji = AI{i};
admji = ji(find(x(ji) > 0));
admji = admji(find(d(T(admji))+1 == d(i)));
```
} inarcs

[admjj, admji] could be the collection of admissible arcs. You can pick any one from this list (assuming there are many).

```
admarcs = [admjj, admji]; k = admarcs(1);
```
← picking the first admissible arc from the list

```
i = H(k);
```
→ advance operation

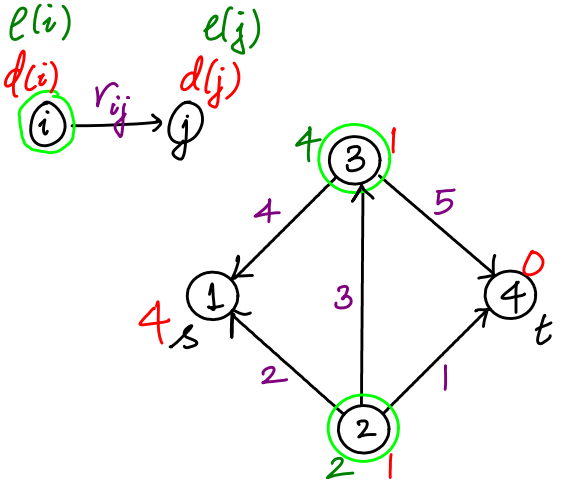
```
predA(i) = k;
```
→ arc predecessor

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 i is relabeled. This whole process is called a **node examination.**
* Examine active nodes in a FIFO order.
  - maintain a LIST of active nodes, remove the first node from the front of LIST to examine.
  - add newly active nodes to the back of LIST.
* If you relabel node i, add it to the back of LIST.
  So, do not push from node i immediately after relabeling it.
  $\rightarrow d(i) = \min\{d(j)+1 \mid (i,j) \in G(\bar{x})\}$
* Complexity : $O(n^3)$ (AMO Theorem 7.17).

Back to our Example

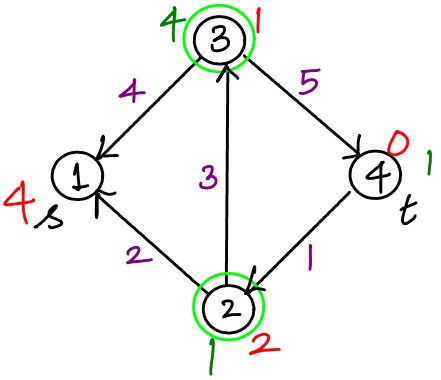
After preprocessing!



LIST = [~~2~~ 3]

examine 2 : push $\delta = \min\{e(2), r_{24}\}$

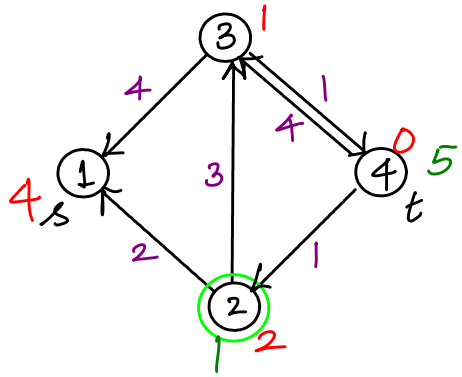
$= \min\{2, 1\} = 1.$



2 is still active, but no admissible arcs exist. So relabel, and add 2 to the back of LIST.

LIST = [~~3~~ 2]

Node examination of 3 now:
(3,4) is admissible. Push
$\delta = \min\{e(3), r_{34}\} = \min\{4, 5\} = 4$.



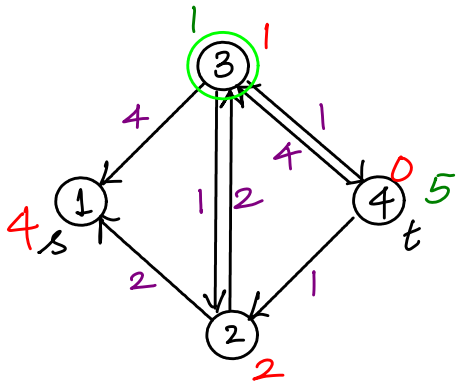
LIST = [~~2~~]

Examine node 2.
Push $\delta = \min\{e(2), r_{23}\} = 1$ along (2,3). 2 becomes inactive, but 3 is active — added to LIST.

LIST = [~~3~~]

Examine node 3.

Push $\delta = \min\{e(3), r_{34}\} = \min\{1, 1\} = 1$ along (3,4).



LIST is empty now, so flow is maximum.

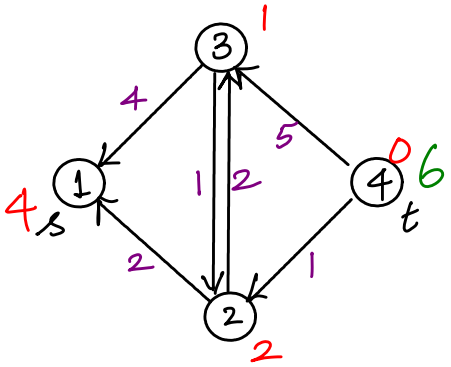
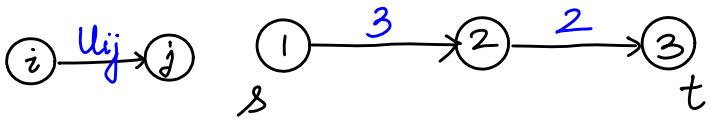
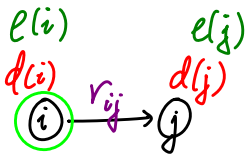
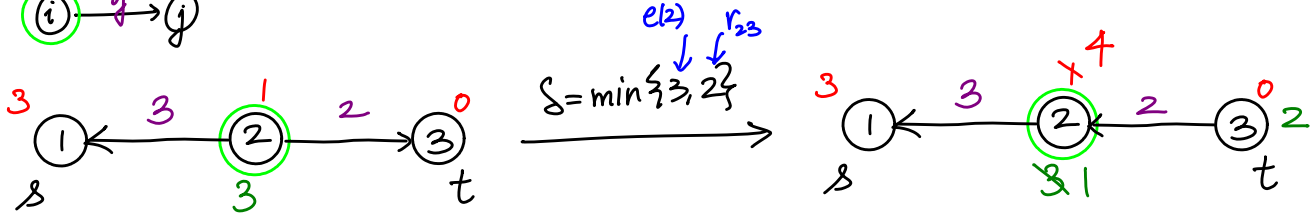


Illustration of pushing flow back to s

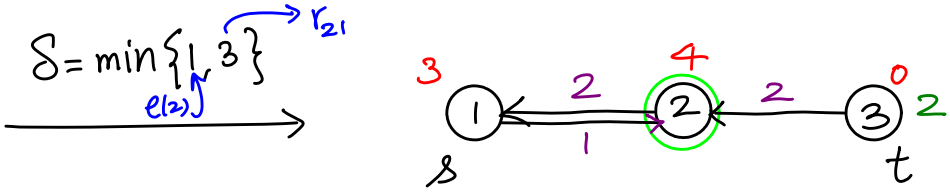
In the previous examples, we did not push any flow back to the source. But that sort of push-back happens routinely in more general networks. Here is a minimal example.







relabel 2, then push 1 unit back to s=1



Excess Scaling Algorithm

→ Idea: Have to get rid of all excess. Might as well push from large excess nodes. Similar to capacity scaling, but applied to preflow-push.

Let $e_{max} = \max_{i \in N \setminus \{s,t\}} \{e(i)\}$.

Start with large e_{max}, push from "large excess" nodes, while systematically reducing e_{max} to 0.

Def For a given Δ, **large excess nodes** are $j \in N \setminus \{s,t\}$ with $e(j) \geq \frac{\Delta}{2}$.

IDEA: Push flow from large excess nodes, but not too much!

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

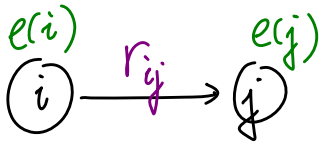
$e(j)$ stays $\leq \Delta$ after push!

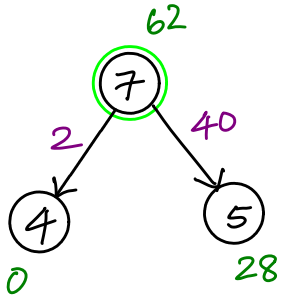
```
algorithm excess scaling;
begin
    preprocess;                                   → same as in preflow push
    Δ := 2^⌈log U⌉;                               → need Δ ≥ U at start
    while Δ ≥ 1 do
    begin (Δ-scaling phase)
        while the network contains a node i with a large excess do   e(i) ≥ Δ/2
            begin
                among all nodes with a large excess, select a node i with
                the smallest distance label;      → pick node closest to t
                perform push/relabel(i) while ensuring that no node excess exceeds Δ;
            end;                                  → δ = min{e(i), r_ij, Δ - e(j)}
        Δ := Δ/2;
    end;
end;
```

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. At the same time, we do not want to just "kick the can down the road", i.e., push all excess from a node to another node just down the hill. The idea is to keep all the excesses in check, while still working on the large excess nodes.

Illustration of Δ-scaling phase



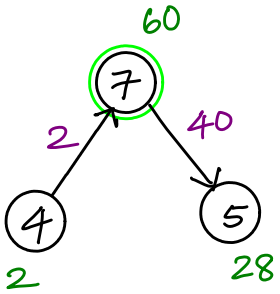


$\Delta = 64$ (64-scaling phase)

7 is a large excess node. We push along (7, 4) first. We push

$\delta = \min\{62, 2, 64\}$. ($64 = \Delta - e(4)$)

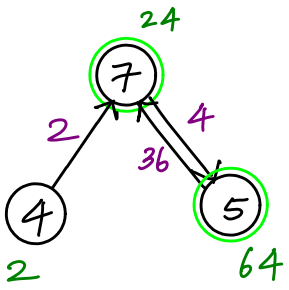
$e(7), r_{74}, \Delta - e(4)$



Then push $\delta = \min\{e(7), r_{75}, \Delta - e(5)\}$

$= \min\{60, 40, 36\}$ → $64-28$

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



Node 5 is now a large excess node (in the 64-scaling phase), but Node 7 is not.

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. We might as well start farther away from t.

The book has many more variants of these algorithms, along with details of implementation. We talk next about min-cost flow...

MATH 566 : Lecture 23 (11/05/2024)

Today:
* MCF — assumptions
* residual network for MCF
* MCF optimality conditions

The Min-Cost Flow (MCF) Problem

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

Max flow optimality conditions: No augmenting paths in $G(\bar{x})$.

In MCF, we work with costs, capacities, and supplies/demand. We use $C = \max_{(i,j)\in A} \{|c_{ij}|\}$ and $U = \max_{(i,j)\in A} u_{ij}$ in our discussion.

Optimization model (Linear Program):

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

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

outflow; inflow; supply/demand

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

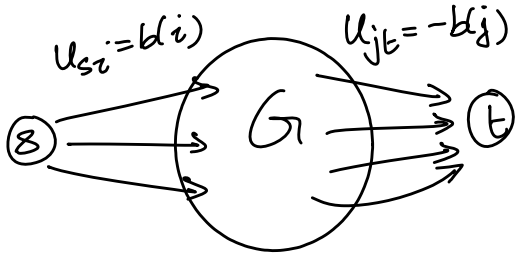
Again, we will **not** directly solve MCF problems as linear programs. The focus will be on efficient algorithms.

AMO describes several applications modeled as MCF — we will not discuss them here.

Assumptions

1. $l_{ij} = 0$ → else remove nonzero lower bounds (network transformations)
2. All data is integral $(c_{ij}, u_{ij}, b(i))$ for complexity analysis purposes
3. The network is directed.
4. $\sum_{i \in N} b(i) = 0$ (total supply = total demand)
   → else, MCF instance is not feasible
5. The MCF problem has a feasible solution.

Recall: Given an MCF problem instance, we can check if it has a feasible flow by solving a max-flow problem.



Add s, t, (s,i) with $u_{si} = b(i)$ for i s.t. $b(i) > 0$ (supply nodes), and (j,t) with $u_{jt} = -b(j)$ for j s.t. $b(j) < 0$ (demand nodes).

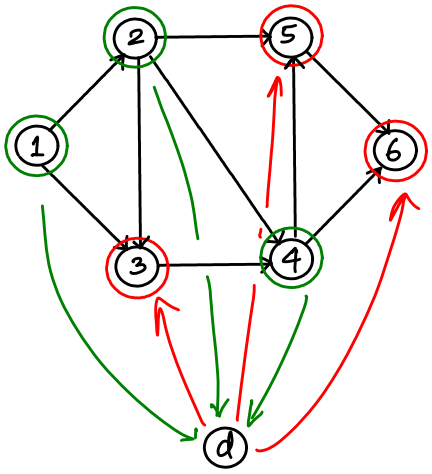
If an s-t max flow saturates all these extra arcs, the original MCF instance has a feasible solution.

Another approach to handle feasibility:

(i) supply node, $b(i) > 0$

(j) demand node, $b(j) < 0$

G



} artificial arcs

d: dummy node (transshipment node) $b(d) = 0$

Add arcs (i,d) $\forall i \in N$ with $b(i) > 0$, and (d,j) $\forall j \in N$ with $b(j) < 0$, with $c_{id} = c_{dj} = +\infty$ and $u_{id} = u_{dj} = +\infty$. Solve MCF on this modified problem. If the optimal solution has any $x_{id} > 0$ or $x_{dj} > 0$, then the original MCF has no feasible solution. If not, x_{ij} $\forall (i,j) \in A$ (in original network G) is an optimal solution to the MCF problem.

This option is motivated by the use of artificial variables in linear programming (the big-M simplex method).

Assumptions (for MCF, continued)

6. There is an uncapacitated directed path between every pair of nodes. Can add extra arc (i,j) with $u_{ij} = \infty$, $c_{ij} = +\infty$ if needed.

7. $c_{ij} \geq 0 \quad \forall (i,j) \in A$ (else, we can use arc reversal)

We need $u_{ij} < \infty$ for this transformation.

If $u_{ij} = \infty$, use $u_{ij} = B > \sum_{\substack{u_{ij} \text{ is} \\ \text{finite}}} u_{ij} + \sum_{b(i)>0} b(i)$.

finite, but large enough to act as $+\infty$.

8. If some $u_{ij} = +\infty$, we assume there is no negative cost cycle of infinite capacity.

If there is such a cycle, the problem is unbounded.
Else, we could replace $u_{ij} = \infty$ with B as shown above.

Notice that we get Assumption 8 when Assumption 7 is satisfied. At the same time, it is helpful to list them both separately.

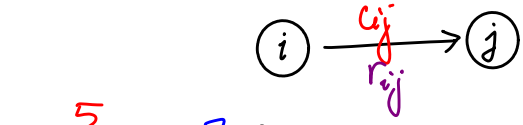
Residual Network for Min-Cost Flow

Recall: residual network for max flow: $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. But in MCF c_{ij} may not be same as c_{ji}, so we cannot combine residual capacities usually.

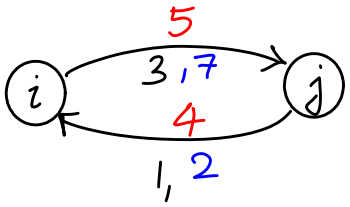
$G(\bar{x})$ for MCF: for arc (i,j) we get

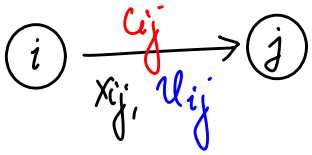
$r_{ij} = u_{ij} - x_{ij}$ with cost c_{ij}, and
$r_{ji} = x_{ij}$ with cost $-c_{ij}$ (or $c_{ji} = -c_{ij}$ here).

Of course, only those arcs (i,j) with $r_{ij} > 0$ appear in $G(\bar{x})$.

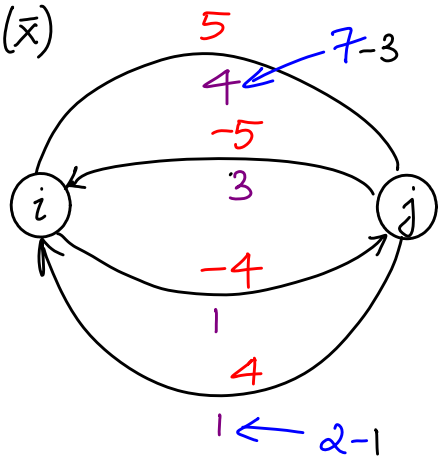


G:





$G(\bar{x})$



Recall that in max flow, 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.

Reduced Costs

Recall, SP optimality conditions: $d(j) \le d(i) + c_{ij} \quad \forall (i,j) \in G_1$.

Equivalently, $c_{ij}^{d} = c_{ij} + d(i) - d(j) \ge 0 \quad \forall (i,j) \in G_1$.

Def For a set of node potentials $\pi(i)$, $i \in N$, $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$ is the **reduced cost** of (i,j) **with respect to** π.

In the SP case, we take $\pi(i) = -d(i)$.

Optimality Conditions for MCF

We extend the SP optimality conditions to $G_1(\bar{x})$ for MCF to define MCF optimality conditions. We then devise algorithms for MCF that check for these optimality conditions repeatedly, and modify $\bar{x}$ and $G_1(\bar{x})$ to correct any violations.

We present three different optimality conditions. Naturally, they are equivalent. The first two are specified on $G_1(\bar{x})$, while the third is specified on G_1 (original network).

1. Negative Cycle Optimality Conditions

AMO Theorem 9.1 A feasible flow $\bar{x}$ is an optimal solution to the MCF problem iff there is no negative-cost cycle in $G(\bar{x})$.

Proof

($\Rightarrow$) If there is a negative cycle $G_f(\bar{x})$, then augment flow along it to reduce total cost. Hence $\bar{x}$ is not an optimal flow. Hence by the contrapositive, if $\bar{x}$ is optimal, $G_f(\bar{x})$ has no negative cycle.

($\Leftarrow$) Let $\bar{x}$ be a feasible flow, and $G_f(\bar{x})$ has no negative cycle. We want to show $\bar{x}$ is optimal.

Assume $\bar{x}^*$ is an optimal flow, and $\bar{x}^* \neq \bar{x}$. Then we can decompose the difference $\bar{x}^* - \bar{x}$ into at most $2m$ cycles. ($G_f(\bar{x})$ has $\leq 2m$ arcs)

The sum of the costs of all these cycles is $\bar{c}^T\bar{x}^* - \bar{c}^T\bar{x}$. (see below for why)

Since $G_f(\bar{x})$ has no negative cycles, it must hold that $\bar{c}^T\bar{x}^* - \bar{c}^T\bar{x} \geq 0 \Rightarrow \bar{c}^T\bar{x}^* \geq \bar{c}^T\bar{x}$. But $\bar{x}^*$ is an optimal solution, i.e., $\bar{c}^T\bar{x}^* \leq \bar{c}^T\bar{x}$. So, $\bar{c}^T\bar{x} = \bar{c}^T\bar{x}^*$.

Hence $\bar{x}$ is also an optimal solution. □

Note: Here $\bar{c}^T\bar{x} = \sum_{(i,j)\in A} c_{ij}x_{ij}$, with $\bar{x} = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ m \end{bmatrix} \begin{bmatrix} x_{ij} \end{bmatrix}$, the vector of flows x_{ij}.

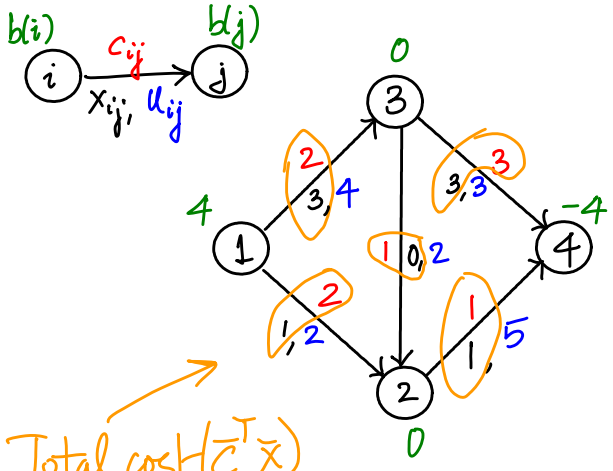
Also, since both $\bar{x}^*$ and $\bar{x}$ are feasible flows, their difference $\bar{x}^* - \bar{x}$ can be decomposed into only cycle flows. Due to the flow balance constraints, and (since $b(i)$ is same for both $\bar{x}^*$ and $\bar{x}$), we cannot get any path flows.

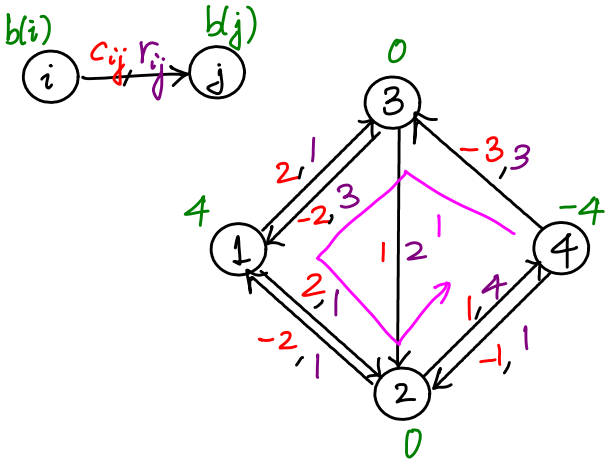
Negative Cycle Canceling Algorithm

* Start with a feasible flow $\bar{x}$.
* Use FIFO label correcting algorithm to identify a negative cycle in $G(\bar{x})$.
* augment, update $\bar{x}$, $G(\bar{x})$; repeat.

Example

G



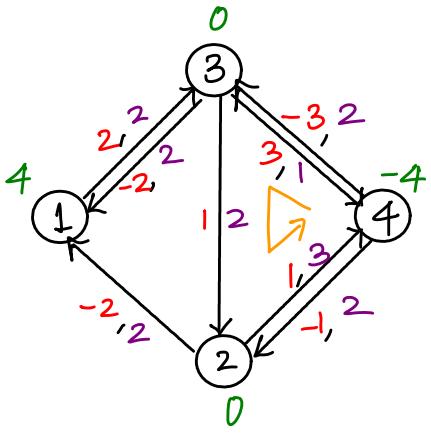


$W_1 = 1\text{-}2\text{-}4\text{-}3\text{-}1$

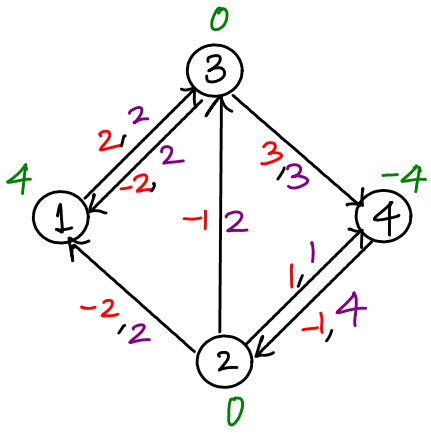
$c(W_1) = -2$

$\delta(W_1) = 1 \quad (r_{12} = 1)$

Augment $\delta = 1$ along W_1.



augment $\delta(W_2) = 2$



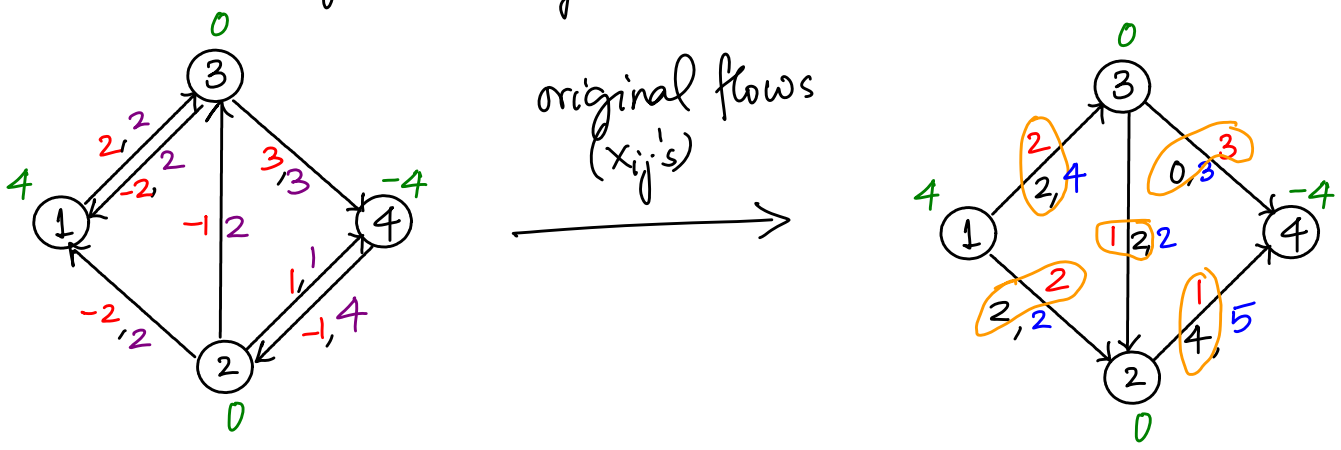
$W_2 = 2\text{-}4\text{-}3\text{-}2$

$c(W_2) = -1, \quad \delta(W_2) = 2$

r_{32} and r_{43}.

No more negative cycles, so flow is optimum.

We recover the original flows x_{ij}, similar to how we did it for max flow:



$\bar{c}^T \bar{x} = 14$ here originally, $\bar{c}^T \bar{x} = 18$.

Finiteness and Complexity

AMO Theorem 9.10 If u_{ij} and $b(i)$ are all integers, the negative cycle canceling algorithm maintains an integral flow (solution) in each iteration.

Proof Initial feasible flow can be found using a max flow, which gives an integral flow. In each iteration, the bottleneck capacity is integral.

Theorem The negative cycle canceling algorithm is finite if all data is finite and integral. (finite → terminates after a finite # iterations)

Proof We use $-mCU$ as a lower bound and mCU as an upper bound for the total cost. Hence, the maximum change in total cost is $2mCU$.

In each iteration, total cost is decreased by at least 1. Hence, the algorithm terminates in at most $2mCU$ augmentations.

FIFO label correcting algorithm in each step takes $O(mn)$ time. Hence the overall time complexity of the negative cycle canceling algorithm is $O(m^2nCU)$. □

MATH 566 : Lecture 24 (11/07/2024)

Today:
* reduced cost optimality conditions
* complementary slackness conditions (CSCS)
* successive SP algo — set up

---

2. Reduced Cost Optimality Conditions

$c_{ij}^{\bar{\pi}} = c_{ij} - \pi(i) + \pi(j)$

In SP, $c_{ij}^{d} \geq 0 \iff d(j) \leq d(i) + c_{ij}$ are the optimality conditions..

AMO Property 9.2

(a) For a path P from node k to l,

$$\sum_{(i,j)\in P} c_{ij}^{\bar{\pi}} = \sum_{(i,j)\in P} c_{ij} - \underbrace{\pi(k) + \pi(l)}_{\text{fixed for a given } \bar{\pi}}.$$

(b) For a directed cycle W, $\sum_{(i,j)\in W} c_{ij}^{\bar{\pi}} = \sum_{(i,j)\in W} c_{ij}$.

Implications

1. A given set of node potentials $\bar{\pi}$ does not change the shortest path from any node k to node l.
2. A directed negative cycle W w.r.t c_{ij} is also a directed negative cycle w.r.t $c_{ij}^{\bar{\pi}}$.

What about the total cost for the MCF problem (under reduced costs)?

With total cost $\bar{c}^T \bar{x} = \sum_{(i,j)\in A} c_{ij} x_{ij}$ and $c^{\bar{\pi}T} \bar{x} = \sum_{(i,j)\in A} c_{ij}^{\bar{\pi}} x_{ij}$,

we would like to get $\bar{c}^T \bar{x} - c^{\bar{\pi}T} \bar{x} = \text{constant}$.

$$\bar{c}^T\bar{x} - \bar{c}^{\bar{\pi}T}\bar{x} = \sum_{(i,j)\in A} (c_{ij} - c_{ij}^{\bar{\pi}})x_{ij}$$

$$= \sum_{(i,j)\in A} (\pi(i) - \pi(j))x_{ij}$$

$$= \sum_{i\in N} \pi(i)\underbrace{\left(\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji}\right)}_{b(i)}$$

$$= \sum_{i\in N} \pi(i)\cdot b(i), \quad \text{a constant for a given } \bar{\pi}.$$

Hence the optimal solutions w.r.t c_{ij} remain optimal w.r.t. $c_{ij}^{\bar{\pi}}$.

Reduced Cost Optimality Conditions: A feasible flow $\bar{x}$ is an optimal solution for the MCF problem iff some node potentials $\bar{\pi}$ satisfy

$$c_{ij}^{\bar{\pi}} \geq 0 \quad \forall (i,j) \in G(\bar{x})$$

→ there exists a set of node potentials $\bar{\pi}$ satisfying $c_{ij}^{\bar{\pi}} \geq 0$.

We now show that the reduced cost optimality conditions are equivalent to the negative cycle optimality conditions.

Proof

IDEA: No negative cycle in $G(\bar{x}) \iff c_{ij}^{\bar{\pi}} \geq 0 \ \forall (i,j) \in G(\bar{x})$.

($\Leftarrow$) Let $c_{ij}^{\bar{\pi}} \geq 0 \ \forall (i,j) \in G(\bar{x})$.

$\Rightarrow$ For every cycle W in $G(\bar{x})$, $\sum_{(i,j)\in W} c_{ij}^{\bar{\pi}} = \sum_{(i,j)\in W} c_{ij} \geq 0$.

$\Rightarrow$ There is no negative cycle in $G(\bar{x})$.

($\Rightarrow$) $G(\bar{x})$ has no negative cycle.

We want to show $c_{ij}^{\bar{\pi}} \geq 0 \ \forall (i,j) \in G(\bar{x})$ for some $\bar{\pi}$. Indeed, we can use $\pi(i) = -d(i)$, where $d(i)$ = shortest path distance from node 1 to node i, $\forall i \in N$.

↳ could use any node

Since $c_{ij}^{\bar{\pi}} = c_{ij} + d(i) - d(j) \geq 0$, we have optimality. □

Economic Interpretation of SP optimality Conditions:

Think of c_{ij} = cost of transporting 1 unit from i to j, and $d(i) = -\pi(i)$ = cost of buying 1 unit at node i. Then $d(j) \leq d(i) + c_{ij}$ says that the cost of buying 1 unit at node j is no more than the cost to buy it at node i and transporting it to node j.

3. Complementary Slackness Optimality Conditions

Let $\bar{x}$ be a feasible flow and $\bar{\pi}$ be a set of node potentials. Then $(\bar{x}, \bar{\pi})$ is optimal iff the following conditions hold.

1. If $c_{ij}^{\bar{\pi}} > 0$ then $x_{ij} = 0$.
2. If $c_{ij}^{\bar{\pi}} < 0$ then $x_{ij} = u_{ij}$.
3. If $0 < x_{ij} < u_{ij}$ then $c_{ij}^{\bar{\pi}} = 0$.

Note

* The CSCs are specified on G_1, and not on $G(\bar{x})$. Recall that the negative cycle and reduced cost optimality conditions are specified on $G(\bar{x})$.

* Condition 3 does not imply the reverse result, i.e., $c_{ij}^{\bar{\pi}} = 0$ does **not** imply $0 < x_{ij} < u_{ij}$.

Intuitively, $c_{ij}^{\bar{\pi}} = 0$ means the flow along (i,j) does not cost anything. Hence it could be any value — in particular, it could be at its lower or upper bound, or in between.

Proof

Show reduced cost optimality conditions $\Leftrightarrow$ CSCs.

We show that reduced cost optimality conditions imply CSCs.

1. If $c^{\bar{\pi}}_{ij} > 0 \Rightarrow r_{ji} = 0$ (or $(j,i) \notin G(\bar{x})$), (assuming reduced cost optimality conditions)

as $c^{\bar{\pi}}_{ji} = -c^{\bar{\pi}}_{ij} < 0$ will violate reduced cost optimality conditions.

But $r_{ji} = 0 \Rightarrow x_{ij} = 0$.

2. If $c^{\bar{\pi}}_{ij} < 0 \Rightarrow r_{ij} = 0 \Rightarrow x_{ij} = u_{ij}$. (violates reduced cost optimality conditions, unless $(i,j) \notin G(\bar{x})$)

3. If $0 < x_{ij} < u_{ij}$ then $r_{ij} > 0$ and $r_{ji} > 0$.

i.e., both (i,j) and $(j,i) \in G(\bar{x})$.

$\Rightarrow c^{\bar{\pi}}_{ij} \geq 0$ and $c^{\bar{\pi}}_{ji} = -c^{\bar{\pi}}_{ij} \geq 0$ according to the reduced cost optimality conditions.

$\Rightarrow c^{\bar{\pi}}_{ij} = 0$.

Successive Shortest Path (SSP) Algorithm for MCF

IDEA Combine tools froms Preflow-push and SAP, apply to MCF now.

* Maintain optimality in terms of reduced cost optimality conditions.
* Strive for feasibility (for flow-balance; bounds hold always).

We present some definitions first.

Def A **pseudoflow** is $\bar{x} \in \mathbb{R}^m$ that satisfies bounds in the MCF formulation, i.e., $0 \le x_{ij} \le u_{ij}$, but not necessarily the flow balance constraints.

Def The **imbalance** at node i is

$$e(i) = b(i) + \sum_{(j,i)\in A} x_{ji} - \sum_{(i,j)\in A} x_{ij}$$

$b(i)$: supply/demand; $\sum_{(j,i)\in A} x_{ji}$: inflow; $\sum_{(i,j)\in A} x_{ij}$: outflow

recall the definition of excess nodes in preflow push for max flow

If $e(i) > 0$, i is an **excess** node, and it belongs to set E, and if $e(i) < 0$, i is a **deficit** node, and it belongs to set D.

Note that $\sum_{i\in N} e(i) = \sum_{i\in N} b(i) = 0$, and

$$\sum_{i\in E} e(i) = -\sum_{j\in D} e(j).$$

Hence, if G has an excess node, it must have a deficit node.

SSP Algo: Maintain optimality as $c_{ij}^{\bar{\pi}} \geq 0 \ \forall (i,j) \in G(\bar{x})$, and strive for feasibility, i.e., $e(i) = 0 \ \forall i \in N$.

We present a key lemma first, which is used in the SSP algorithm.

AMO Lemma 9.11

Let $\bar{x}$ be a pseudoflow, $\bar{\pi}$ a set of node potentials satisfying $c_{ij}^{\bar{\pi}} \geq 0 \ \forall (i,j) \in G(\bar{x})$. Let $\bar{d}$ be the set of SP distances from some node s in $G(\bar{x})$ to all (other) nodes in $G(\bar{x})$ using $c_{ij}^{\bar{\pi}}$ as arc lengths. The following results hold.

(a) Let $\bar{\pi}' = \bar{\pi} - \bar{d}$. Then $c_{ij}^{\bar{\pi}'} \geq 0 \ \forall (i,j) \in G(\bar{x})$; and

(b) $c_{ij}^{\bar{\pi}'} = 0 \ \forall (i,j)$ in a shortest path from s to another node in $G(\bar{x})$.

Proof

(a) $\bar{d}$: SP distances w.r.t. $c_{ij}^{\bar{\pi}}$ in $G(\bar{x})$

$$\Rightarrow \quad d(j) \leq d(i) + c_{ij}^{\bar{\pi}} \ \forall (i,j) \in G(\bar{x}).$$

Using $c_{ij}^{\bar{\pi}} = c_{ij} - \pi(i) + \pi(j)$ gives

$$d(j) \leq d(i) + c_{ij} - \pi(i) + \pi(j)$$

$$\Rightarrow c_{ij} - [\pi(i) - d(i)] + [\pi(j) - d(j)] \geq 0$$

$$\Rightarrow c_{ij}^{\bar{\pi}'} \geq 0 \ \forall (i,j) \in G(\bar{x}), \text{ where } \bar{\pi}' = \bar{\pi} - \bar{d}.$$

Thus, decreasing node potentials $\bar{\pi}$ by SP distances $\bar{d}$ in $G(\bar{x})$ maintains reduced cost optimality.

(b) Along the SP tree in $G(\bar{x})$, $d(j) = d(i) + c_{ij}^{\bar{\pi}}$.

$\Rightarrow \quad c_{ij}^{\bar{\pi}'} = 0 \quad \forall (i,j)$ in the SP tree. $\square$

MATH 566: Lecture 25 (11/12/2024)

Today: * SSP algo, example
* MST, cut/path optimality conditions

Successive Shortest Path (SSP) Algorithm for MCF

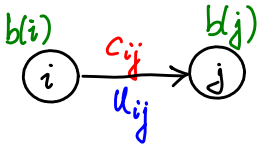
algorithm *successive shortest path*;
begin
  $x := 0$ and $\pi := 0$; → no need to find feasible flow @start
  $e(i) := b(i)$ for all $i \in N$;
  initialize the sets $E := \{i : e(i) > 0\}$ and $D := \{i : e(i) < 0\}$;
  while $E \neq \emptyset$ **do**
  begin
    select a node $k \in E$ and a node $l \in D$;
    determine shortest path distances $d(j)$ from node k to all other nodes in $G(x)$ with respect to the reduced costs c_{ij}^{π}; → ≥ 0, can use Dijkstra (from step 2 onward)
    let P denote a shortest path from node k to node l;
    update $\pi := \pi - d$; → maintain optimality.
    $\delta := \min[e(k), -e(l), \min\{r_{ij} : (i, j) \in P\}]$;
    augment δ units of flow along the path P;
    update x, $G(x)$, E, D, and the reduced costs;
  end;
end;

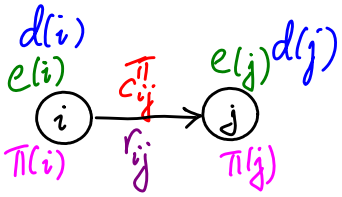
supply nodes
demand nodes

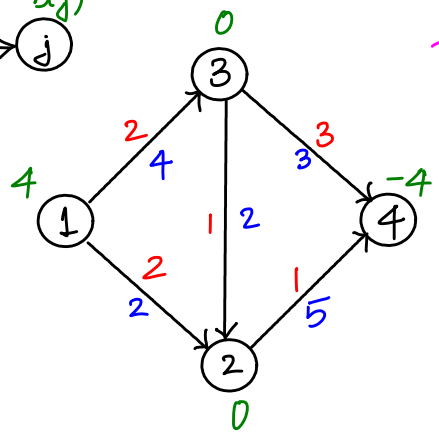
Figure 9.9 Successive shortest path algorithm.

Since we maintain optimality ($C_{ij}^{\bar{\pi}} \geq 0 \ \forall (i,j) \in G_f(\bar{x})$), we can use efficient algorithms — Dijkstra — to compute the SP distance labels in each iteration after the first one.
When we start, $\bar{\pi} = \bar{0}$ and hence $C_{ij}^{\bar{\pi}} = C_{ij}$ itself, which could be < 0. But after the first SP computation, $C_{ij}^{\bar{\pi}} \geq 0$ is maintained (Lemma 9.11).

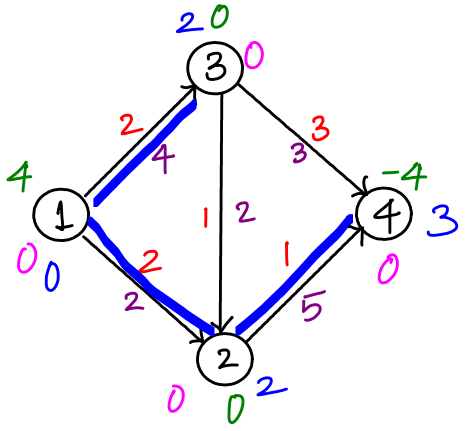
SSP Algorithm: Example







$E=\{1\}$, $D=\{4\}$



Iteration 1 : SP from $k=1$

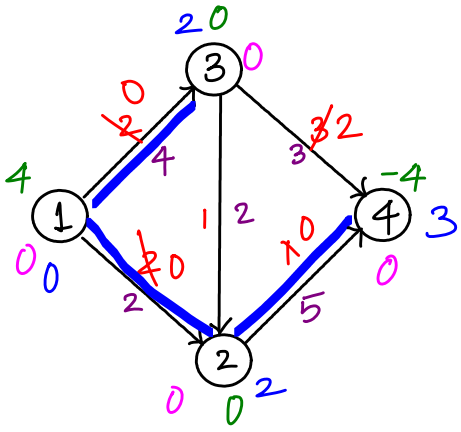
$d(i)$'s

(SP tree shown in **blue**)

$\pi \leftarrow \pi - d$

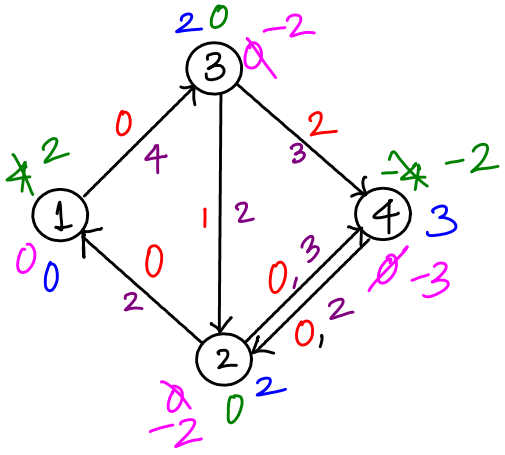
$P_1 = 1\text{-}2\text{-}4$. $\delta = \min\{e(1), -e(4), r_{12}, r_{24}\}$

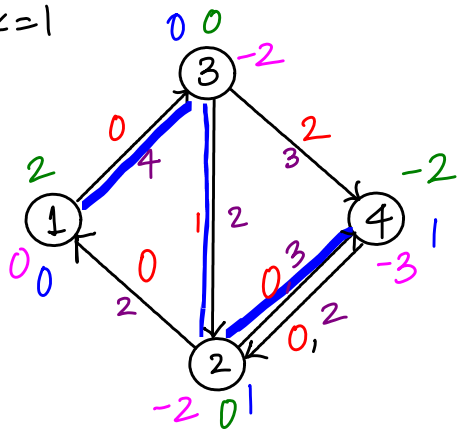
$= \min\{4, 4, 2, 5\} = 2$

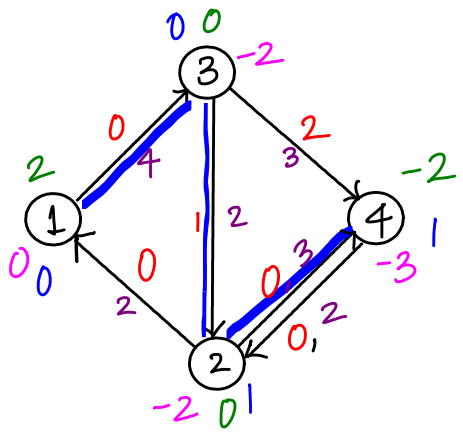


Iteration 2 $E=\{1\}$, $D=\{4\}$.

SP from $k=1$





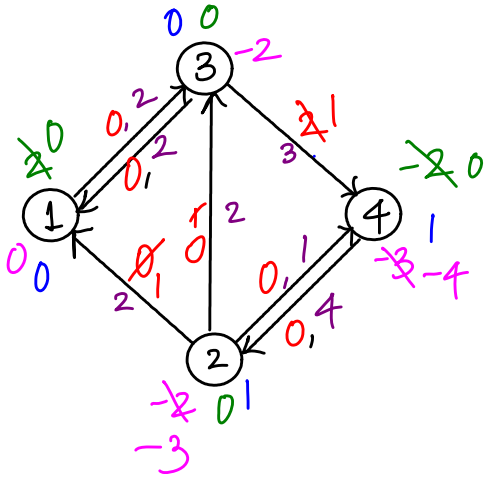


$P_2 = 1\text{-}3\text{-}2\text{-}4$

$\delta = \min\{e(1), -e(4), r_{13}, r_{32}, r_{24}\}$

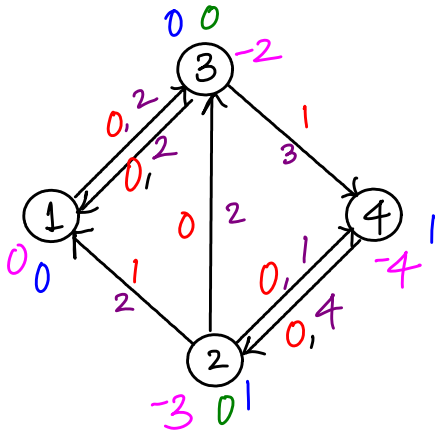
$= \min\{2, 2, 4, 2, 3\} = 2$

$\bar{\pi} \leftarrow \bar{\pi} - \bar{d}$



Flow is optimum, as

$E = \phi = D$ now.



flow in the original network:

$x_{12} = 2,\ x_{13} = 2,\ x_{32} = 2,\ x_{24} = 4$

Minimum Spanning Trees (MST) (AMO Chapter 13)

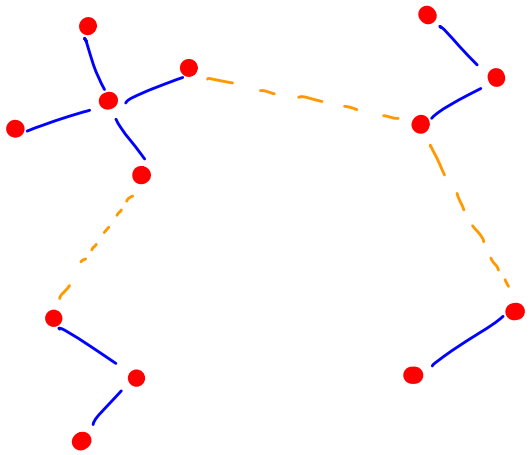
We switch to undirected networks, i.e., arcs are undirected.

$G=(N, A)$, but now A has undirected arcs (or edges).

A **spanning tree** T of G is a connected, acyclic subgraph that spans all nodes in N. T has $n-1$ arcs.

A **minimum spanning tree** (MST) is a spanning tree that has the smallest total cost $\sum_{(i,j)\in T} c_{ij}$.

An application: Cluster analysis. → see AMO for more applications



Can start building MST by assembling smaller trees. Stop at a specified cut-off value (for c_{ij}).

The connected components (subtrees) form the clusters.

Optimality Conditions

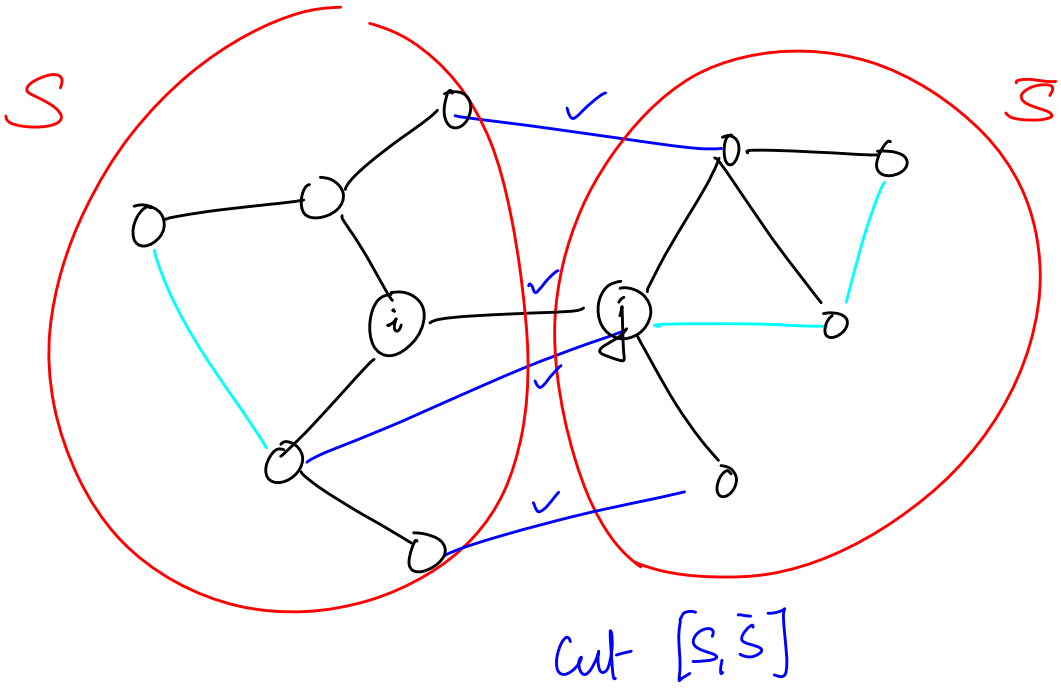
- cut optimality conditions
- path optimality conditions

Notation Arcs in a spanning tree are called tree arcs, while arcs not in the spanning tree are non-tree arcs.

Observation

(1) For every non-tree arc (k, ℓ), there is a unique path in T connecting k and ℓ.

(2) Deleting a tree arc (i, j) from a spanning tree divides N into two disjoint subsets $S, \bar{S}$. The arcs of G (k, ℓ) with $k \in S$, $\ell \in \bar{S}$ form a cut $[S, \bar{S}]$.

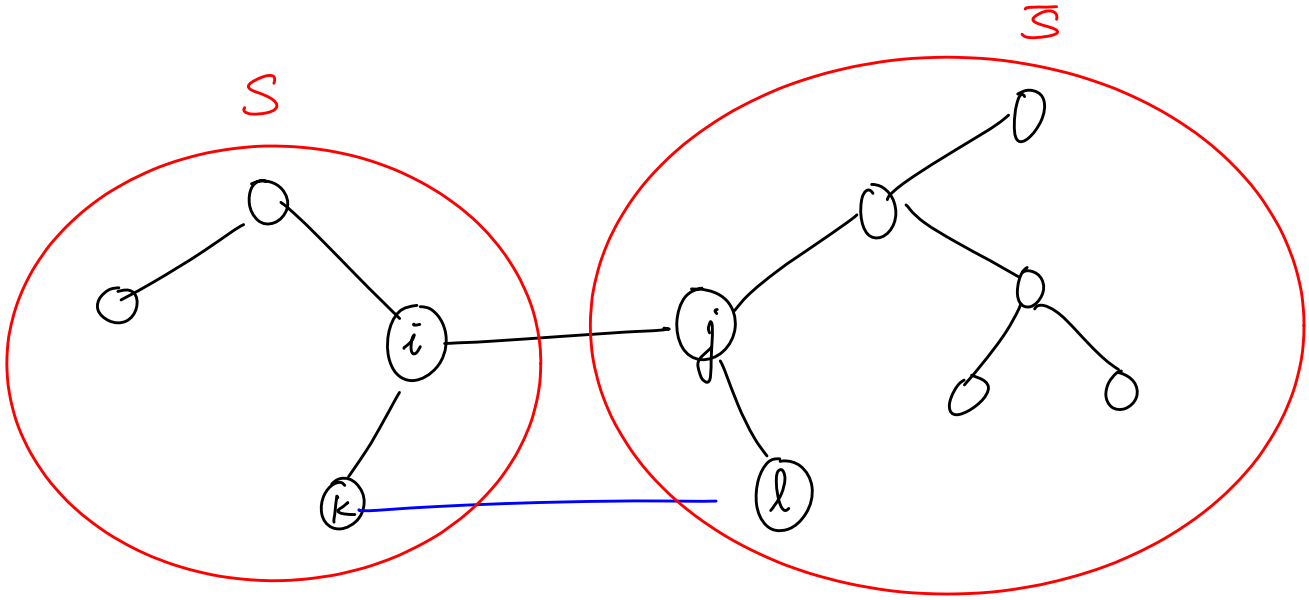


Cut Optimality Conditions

A spanning tree T is an MST iff $\forall (i,j) \in T$, $c_{ij} \leq c_{kl}$ $\forall (k,l) \in [S, \bar{S}]$, where $[S, \bar{S}]$ is the cut formed by deleting (i,j) from T.

Proof

($\Longrightarrow$) Assume T is an MST but $c_{ij} > c_{kl}$ for some $(i,j) \in T$ and $(k,l) \in [S, \bar{S}]$. Then we can replace (i,j) by (k,l) in T to obtain another spanning tree with smaller total cost, contradicting minimality of T.



($\Leftarrow$) Let T be a spanning tree and $c_{ij} \leq c_{kl}$ holds.
We want to show T is an MST.

Let T^0 be an MST and $T^0 \neq T$.

$\Rightarrow \exists (i,j) \in T$ such $(i,j) \notin T^0$.

$\Rightarrow \exists (k,l) \in T^0$ such that $k \in S$, $l \in \bar{S}$, where $[S, \bar{S}]$ is the cut obtained by deleting (i,j) from T.

The pieces in S and $\bar{S}$ need to be connected somehow in T^0.

T satisfies cut optimality conditions $\Rightarrow c_{ij} \leq c_{kl}$.

T^0 is an MST $\Rightarrow c_{kl} \leq c_{ij}$.

$\Rightarrow c_{ij} = c_{kl}$.

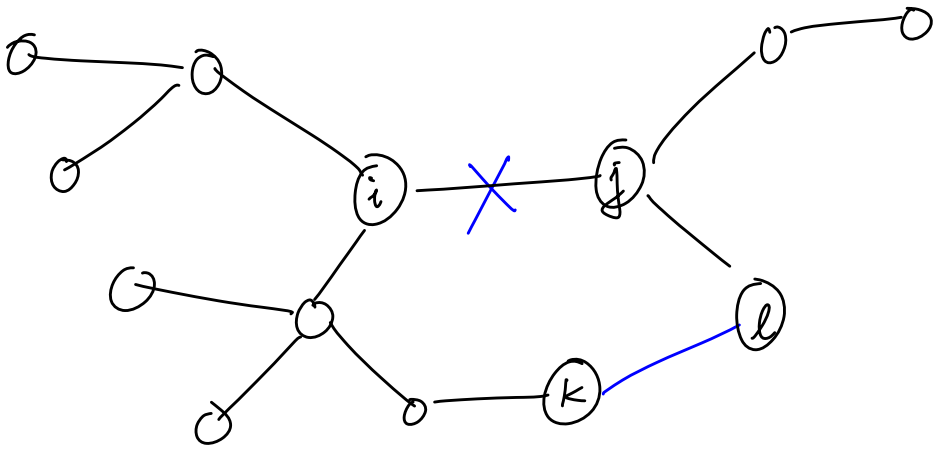
Hence we can replace (i,j) in T with (k,l). Repeat this process until $T = T^0$. We have not changed the total cost all along, so T must be an MST as well.

□

Path Optimality Conditions

A spanning tree T is an MST iff for every non-tree arc $(k,\ell) \in G$, $c_{ij} \le c_{k\ell}$ $\forall (i,j)$ in the path connecting k and ℓ in T.

Proof If T is an MST and $c_{ij} > c_{k\ell}$, then replacing (i,j) with (k,ℓ) gives a contradiction.



($\Leftarrow$) Let T be a spanning tree satisfying the path optimality conditions. We show that T will satisfy the cut optimality conditions.

$(k,\ell) \in [S, \bar{S}] \Rightarrow$ There is a unique path in T connecting k and ℓ. (i,j) is the only arc connecting S and $\bar{S}$. Hence (i,j) is in this path.

Path optimality $\Rightarrow$ $c_{ij} \le c_{k\ell}$. This holds $\forall (k,\ell) \in [S,\bar{S}]$.

$\Rightarrow$ cut optimality conditions hold. $\square$

MATH 566: Lecture 26 (11/14/2024)

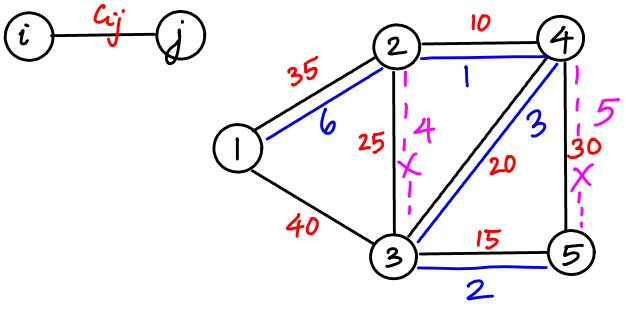
Today: * algos for MST - Kruskal's and Prim's algos
* Assignment and matching

Kruskal's Algorithm for Minimum Spanning Tree (MST)

Uses path optimality conditions

- builds the tree by adding one arc at a time
- uses a sorted LIST of arcs in the increasing order of c_{ij} values
- check whether adding an arc creates a cycle
  * if no, add it to tree
  * if yes, discard arc from LIST.

Example



Order in which arcs are considered indicated as 1, 2, ..., 6.

Arcs (2,3) and (4,5) are not added to the tree (4th and 5th arcs considered), as they would create cycles.

Complexity

Sorting arcs is the bottleneck step.

Sorting m arcs : $O(m \log m) = O(m \log n^2) = O(m \log n)$.

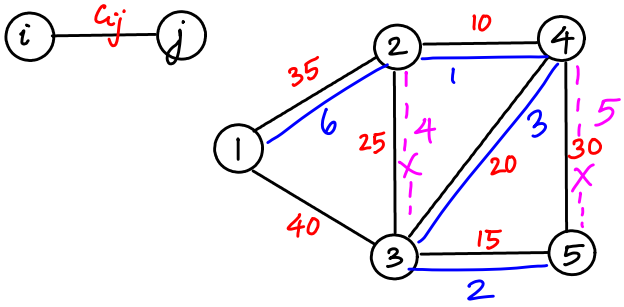
Detecting cycles when adding arcs can be done efficiently using the UNION-FIND data structure for maintaining connected components. There are two main operations/functions:

FIND(i): returns connected component containing node i
each component is represented by a single node in it

UNION(i,j): joins components containing i and j into single component, now represented by i.

We can describe Kruskal's algorithm

```
T := φ;        MST is empty at start
LIST of arcs (k,ℓ) sorted according to c_kℓ (smallest to largest)

for each arc (k,ℓ) ∈ LIST do
    if FIND(k) == FIND(ℓ) then
        discard (k,ℓ);
    else
        UNION(k,ℓ);
        add (k,ℓ) to tree T
    end
end
```



{1} {2} {3} {4} {5}

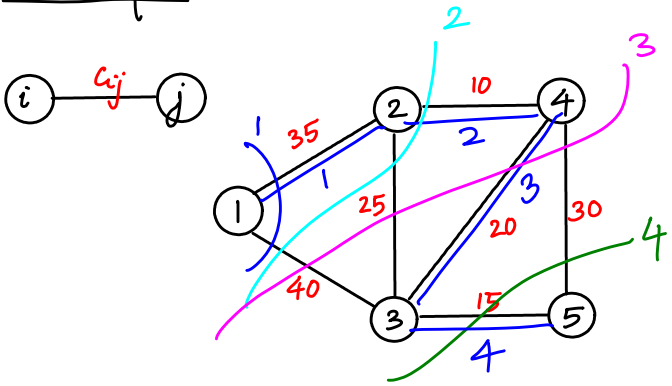
1. (2,4) → {2 4}
2. (3,5) → {3 5}
3. (3,4) → {1}, {2 3 4 5}
4. (2,3) FIND(2) = FIND(3) = 2 ⇒ discard
5. (4,5) FIND(4) = FIND(5) = 2 ⇒ discard
6. (1,2) → {1 2 3 4 5}.

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 N, and adds $(i,j) \in [S, \bar{S}]$ with the smallest c_{ij} to the tree.

Example: cuts in each iteration shown: 1, 2, 3, 4



S = [1] at start

→ [1,2] → [1,2,4]

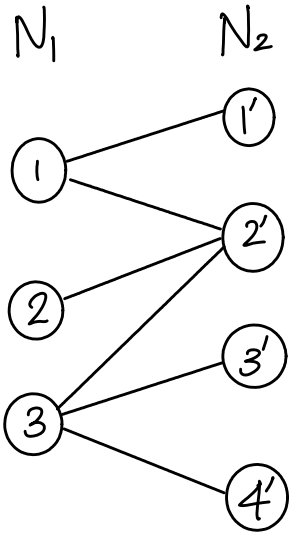
→ [1,2,4,3] →

[1,2,4,3,5]

Solin's algorithm combines Kruskal's and Prim's algorithms.

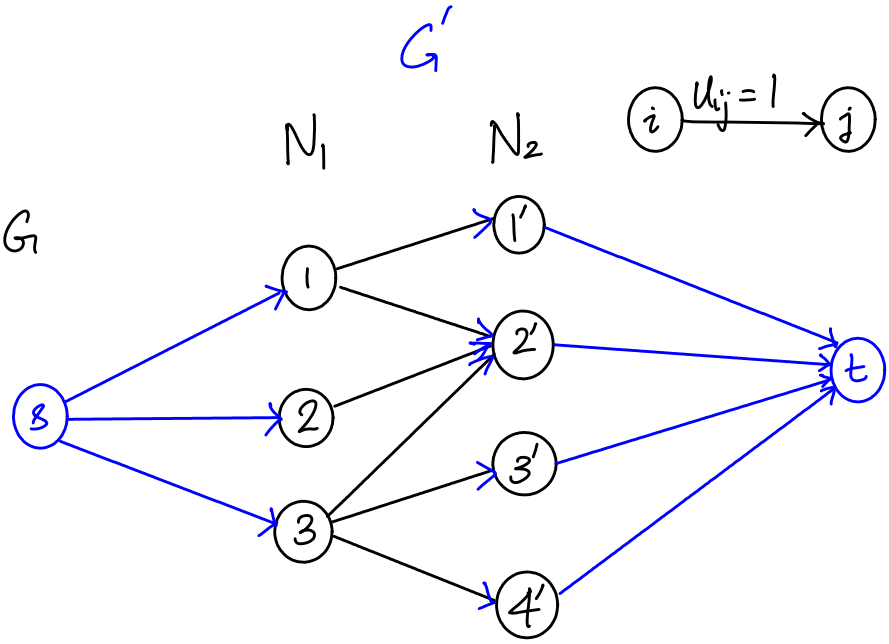
Assignments and Matching Problems (AMO Chapter 12)

1. Bipartite cardinality matching problem:
   $G = (N_1 \cup N_2, A)$ where A has undirected edges (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 $u_{ij} = 1 \; \forall (i,j)$, and each node i has either indegree$(i) \leq 1$ or outdegree$(i) \leq 1$.

Set all arc capacities as 1.



* Start with $G' = G$
* Direct all arcs $(i,j) \in G$ from i to j.
* Add nodes s, t, and arcs $(s,i) \; \forall i \in N_1$ and $(j,t) \; \forall j \in N_2$
* set all arc capacities in G' as 1.

We can show that a matching of cardinality k in G_1 corresponds to an s-t flow in G' with value k.

($\Rightarrow$) Given a matching of cardinality k in G_1, set $x_{si} = x_{ij} = x_{jt} = 1$ in G' $\forall (i,j)$ in the matching. Value of flow $= k$ here.

($\Leftarrow$) 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 $i \in N$, $j \in N_2$ to define the matching. □

Note that these paths will be arc disjoint (they share only the nodes s and t) since all capacities are 1.

Hence solving the cardinality bipartite matching problem on G_1 is equivalent to solving the max flow problem on G'.

The max flow problem on simple networks can be solved in $O(m\sqrt{n})$ time (AMO Chapter 8). Hence the bipartite cardinality matching problem can be solved in $O(m\sqrt{n})$ time.

2. Bipartite Weighted Matching Problem (Assignment Problem)

$G = (N_1 \cup N_2, A)$, $|N_1| = |N_2| = n$, c_{ij}'s are costs on undirected arcs (i,j) with $i \in N_1, j \in N_2$. The goal is to find a perfect matching, i.e., match all nodes in pairs, of minimum total cost.

Here is the optimization model (LP) using variables x_{ij} defined to be $x_{ij} = 1$ if i and j are matched for $i \in N_1$, $j \in N_2$, and $x_{ij} = 0$ otherwise.

linear program

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

$$\text{s.t.} \quad \sum_{(i,j) \in A} x_{ij} = 1 \quad \forall\, i \in N_1$$

($b(i) = 1$; supply nodes)

$$-\left(\sum_{(i,j) \in A} x_{ij} = 1 \right) \quad \forall\, j \in N_2$$

(demand nodes with $b(j) = -1$)

$$0 \le x_{ij} \le 1 \quad \forall (i,j) \in A$$

upper bound is implied by the perfect matching constraints

Special case of the min-cost flow problem with the nodes in N_1 being supply nodes (all with unit supplies) and nodes in N_2 are demand nodes (all with unit demands).

Successive Shortest Path Algorithm (for assignment problem)

Recall $(\bar{x}, \bar{\pi})$ are optimal for MCF iff $c_{ij}^{\bar{\pi}} \geq 0 \; \forall (i,j) \in G(\bar{x})$.

Here, augmenting 1 unit corresponds to assigning one additional node in N_1. We augment 1 unit in each iteration.

If $S(n,m,C)$ is the complexity of solving the SP instances, the assignment problem could be solved in $O(nS(n,m,C))$ time.

A Relaxation Algorithm

Allow nodes in N_2 to be over- or under-assigned. Here, under-assigned means not assigned at-all. We then pick node $k \in N_2$ that is over-assigned, find SP from k to all nodes in $G(\bar{x})$ with $c_{ij}^{\bar{\pi}}$ as costs. Augment 1 unit along shortest path from k to some $l \in N_2$ that is underassigned.

MATH 566: Lecture 27 (11/19/2024)

Today: * stable marriage problem
* non-bipartite cardinality matching

Stable Marriage Problem

A special case of weighted bipartite matching, but we work with rankings rather than c_{ij}'s.

We have two groups (group1, group2) of n persons each. Each person provides a ranking of the n people in the other group, e.g., from most to least preferred.

Def A pair $[p_1, p_2]$ of persons with $p_i \in$ group$_i$, $i=1,2$, is **unstable** if they are not married to each other, but they prefer each other to their spouses.

The goal of stable marriage problem is 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 a **propose-and-reject** algorithm, which is a greedy algorithm.

Input: Two $n \times n$ matrices of rankings by the two groups. The rankings are integers in $[1,n]$, e.g., each person produces a permutation of $1, 2, \ldots, n$.

The set-up The algorithm maintains a LIST of unassigned group 1 members, and each person in LIST has a current_candidate, to whom they propose next.

→ from group 2.

The algorithm

* At each step, pick a person from LIST, say Bill.
* Bill proposes to his current_candidate, say, Helin.
* If Helin is unassigned, the proposal is accepted, and [Bill, Helin] are engaged.
* If Helin is engaged already to France ∈ group 1, Helin selects their preferred partner (between Bill and France), and rejects the other person.
* Rejected person is added back to the LIST, and they designate the next person in their preferred list as their current_candidate.
* Continue until LIST is empty, and then all engaged couples are married.

LIST can be handled as a FIFO or LIFO queue — it does not make a big difference in complexity here.

Correctness

If Dave ∈ group1 prefers Laura ∈ group2 over his spouse (as determined by the algorithm), he would've proposed to Laura earlier, and Laura would have rejected Dave in favor of someone she likes more. Since no person in group2 switches to a partner they prefer less, Laura prefers her spouse to Dave. Hence [Dave, Laura] is not an unstable pair.

Hence the matching found is stable. □

→ Laura must prefer Dave to her spouse as well to create an unstable pair

Def A pair $[P_1, P_2]$, $P_i \in \text{group}_i$, $i=1,2$, are called **stable partners** if they are matched in some stable matching.

Note: There could be many stable (perfect) matchings.

AMO Lemma 12.4 In the propose-and-reject (P&R) algorithm, a person of group2 never rejects a stable partner.

Proof Let M^* be the stable matching given by the P&R algo. Suppose lemma is false, i.e., group2 persons reject stable partners. Let the first such rejection be when Jona rejects Dave.

Let M° be a stable matching in which [Dave, Jona] is a pair.

We will show M° cannot be stable.

Let the rejection (in M^*) happen because Jona is engaged to Steve, and Jona prefers Steve to Dave. Since this is the first such rejection, no other group 2 person rejected a stable partner before. Hence Steve prefers Jona to all other stable partners.

In M°, let [Steve, Sue] be matched.

Look at pair {Steve, Jona}:

- Steve prefers Jona to Sue
- Jona prefers Steve to Dave

Pairs in M°:

[Dave, Jona]
[Steve, Sue]

$\Rightarrow$ {Steve, Jona} is an unstable pair, and hence M° is not a stable matching — a contradiction! □

We can argue that the P&R algo creates a **group-1 optimal** matching, in which every group 1 person is married to their best stable partner. This result follows from the observation that group 1 persons propose to group 2 candidates in decreasing order of preference, and no group 2 person ever rejects a stable partner.

This is a somewhat surprising result — if each group 1 person is given their best stable partner independently, we get a stable matching!

On the other hand, we can also show that the P&R algorithm produces a **group-2 minimal** matching, where each person group 2 gets their least preferred stable partner.

We could reverse the roles of groups 1 and 2 to get a matching that favors group 2 more.

Complexity of P&R Algorithm

Each group 2 person either
(1) gets their first proposal, or → n times in total
(2) rejects a group 1 candidate. → at most (n-1) times for each group 2 person

Step (2) is bottleneck, taking $O(n^2)$ time. The data (input) is $O(n^2)$ to start with (two $n \times n$ matrices). Hence the P&R algorithm is optimal, since handling data itself takes $O(n^2)$ time.

So, we cannot design an algorithm that runs in time strictly smaller than n^2 (in the O-sense).

4. Non-bipartite Cardinality Matching

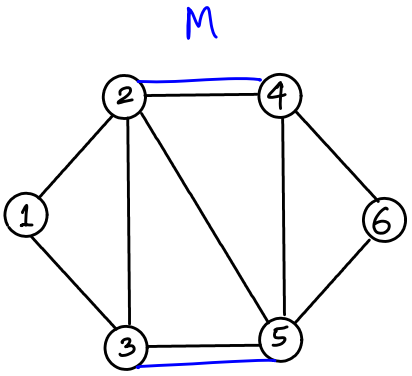
$G=(N,A)$ is an undirected network. We want to match as many nodes as possible uniquely (using edges in A).

A **matching** M of $G=(N,A)$ is a subset of the 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.

If $|N|=n$, the **size** of a matching is at most $\lfloor \frac{n}{2} \rfloor$.

(size = # arcs in the matching)

Example



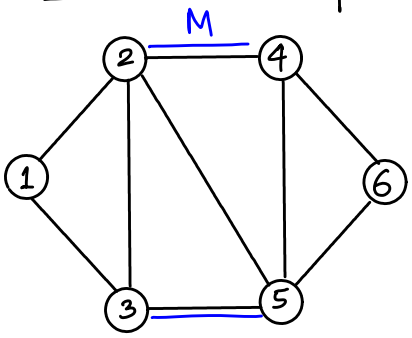
$M=\{(2,4),(3,5)\}$ is a matching of size $|M|=2$. Note that nodes 1, 6 are unmatched here.

We will use "augmenting paths" to increase the cardinality of a matching — more in the next lecture.

MATH 566: Lecture 28 (11/21/2024)

Today: Nonbipartite cardinality matching

Recall: non-bipartite matching

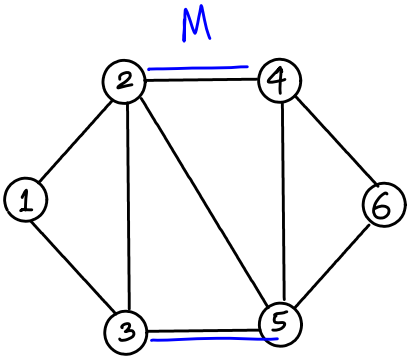


Matching M: subset of arcs of G s.t. any node is incident to at most one arc in M.

Size of matching = # arcs in M (M)
$\leq \lfloor \frac{n}{2} \rfloor$ when $|N| = n$.

Def A path $P = i_1 - \cdots - i_k$ is an **alternating path** w.r.t. a matching M if every pair of consecutive arcs in P contains one arc in M and one not in M.

Def An alternating path is an odd (even) alternating path if it has an odd (even) # arcs.



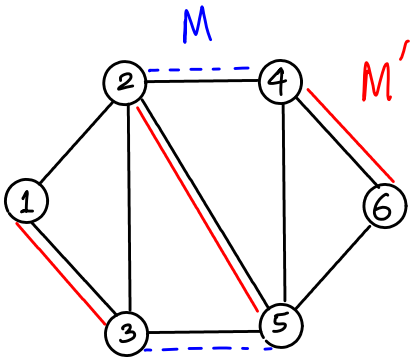
e.g., $1 - 2 = 4 - 6$
$1 - 3 = 5 - 2 = 4 - 6$
} odd alternating paths

$3 - 5 = 2 = 4 - 6$ } even alternating path

An alternating cycle is an alternating path that starts and ends at the same node.

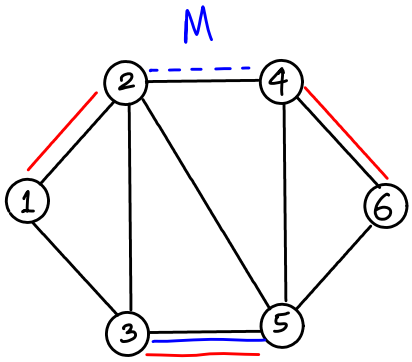
Def An odd alternating path w.r.t. matching M is an **augmenting path** if both its first and last nodes are unmatched in M.

We can swap the matched and unmatched arcs in an augmenting path to get a matching M' with $|M'| = |M|+1$.



$P = 1 - 3 - 5 - 2 - 4 - 6$ is an augmenting path

$|M'| = 3 = |M|+1.$



$P' = 1 - 2 - 4 - 6$

$M' = \{(1,2), (4,6), (3,5)\}$

$|M'| = 3 = |M|+1.$

We use set theoretic notation to describe these operations.

The **symmetric difference** of two sets S_1 and S_2 is denoted

$$S_1 \oplus S_2 = (S_1 \cup S_2) - (S_1 \cap S_2).$$

"oplus"

In words, this is the collection of elements in either set but not in both.

e.g. $\{2,3,5,6\} \oplus \{3,6,8,4\} = \{2,5,8,4\}.$

In the examples, we found $M \oplus P$ (or $M \oplus P'$), where M is a given matching and P is an augmenting path.

AMO 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 $M \oplus P$.

Here is an idea for an augmenting path algorithm for matching:

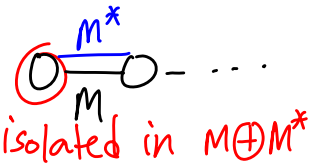
* start with a matching M.
* pick a node i unmatched in M,
  - find augmenting path P starting at i,
  - find $M \oplus P$.
* Repeat.

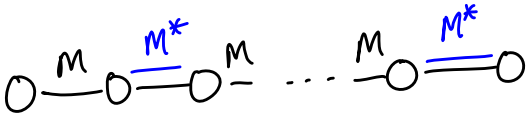
Q. What if we cannot find augmenting path starting at unmatched node i?

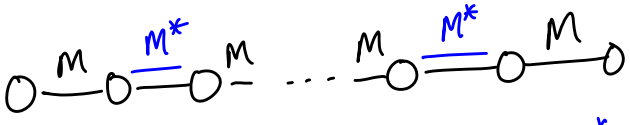
Way Out We can show that there exists a maximal matching M' in which node i is unmatched. So we try to find M', rather than continuing to grow current matching M.

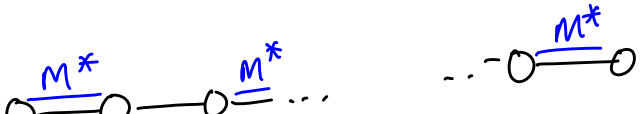
AMO Property 12.7 Let M and M^* be two matchings. Then $M \oplus M^*$ defines $G' = (N, M \oplus M^*)$, a subgraph of G, with the property that every component of G' is one of the following six types.

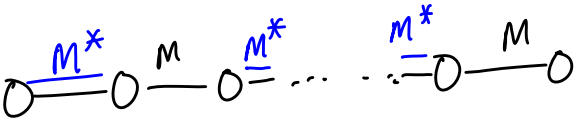
(a) o (degree 0; isolated node)

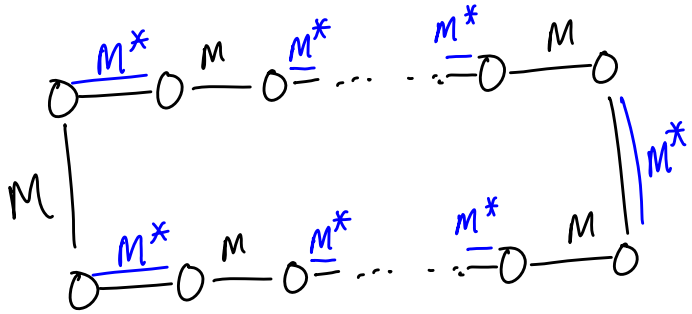


(b)  even

(c)  odd (one arc in M is unpaired with M^*)

(d)  odd (one arc in M^* unpaired)

(e)  even

(f)  even cycle

Note that the starting node (left-most) is matched to M in (b), and in M^* in (e), both even components.

Result follows from the fact that every node in G' has degree 0, 1, or 2, and the above six types are the only possibilities.

The augmenting path algorithm depends on the following theorem.

AMO Theorem 12.8 If a node p is unmatched in matching M, and there is no augmenting path starting at node p, then node p is unmatched in some maximal matching.

Proof Let M^* be a maximum matching. If p is unmatched in M^*, we are done. So, assume p is matched in M^*.

Consider $M \oplus M^*$. This subgraph has components as specified in property 12.7.

Since p is unmatched in M, p has to be the starting node in a component of type (d) or (e). As there is no augmenting path starting at node p in M, the only option is (e), which is even.

Then $M' = M^* \oplus P$ is also a maximum matching, as P is an even path and M^* is maximal. And p is unmatched in M'. □

Note that this is an existence result ☹!
This result does not tell how to find M^* or M'...

Here is an algorithm we could consider:

* start with any matching M (e.g., zero matching)
* pick a node p that is unmatched in M
* search for augmenting path starting at node p.
  - if path found, augment
  - else delete p and all arcs incident to p.

Unfortunately, this algorithm is guaranteed to work only for bipartite networks!

Come to my Math 567 class in Spring (Integer Optimization) to learn how to solve this problem!

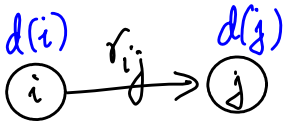
MATH 566: Lecture 29 (12/03/2024)

Today: * On SAP algorithm
* generating random networks for max flow

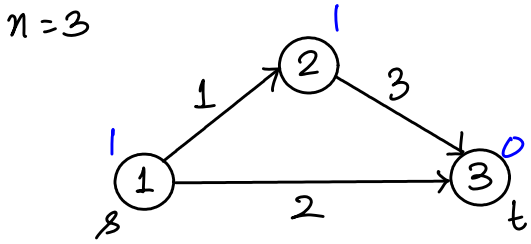
On Shortest Augmenting Path algorithm

Recall that the main "while" loop runs as long as $d(s) < n$ holds. We could encounter a situation where we can't advance from s, i.e., there are no admissible arcs out of s. But there are also no arcs available to relabel. Yet, since $d(s) < n$ still holds, the algorithm is "stuck". Here is an example.

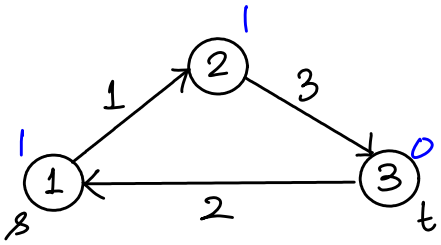
relabel: set $d(i) = \min_{j:(i,j)\in G(\bar{x})} \{d(j)\} + 1$.



$n = 3$



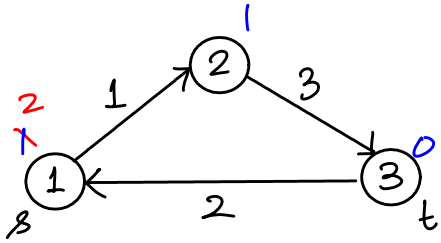
1. Augment along (1,3):

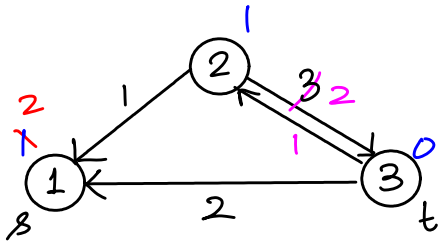


2. Relabel node $s=1$: $d(1) \leftarrow d(2)+1$

$d(s) = 2 < n = 3$ still.

3. Augment along 1-2-3





We can stop here, but $d(s) = 2 < n = 3$, and we cannot relabel $d(s)$ to increase it beyond n.

If you face this situation, just set $d(s) = 2n$ (or $n+1$), for instance.

Test your programs on several small networks, and for different choices of s and t (e.g., $s = t$ should give $v = 0$).

Generating Random Networks for Max Flow

Could justify choosing $s = 1, t = n$ as default.

* For generating "nontrivial" instances, ensure there is an s-t path in the instances.
* Could, alternatively, run the algorithms on two subsets of instances — one with s-t paths, and second without.

So, use an existing random network generator, or generate networks randomly on your own. For each network, test if an s-t directed path exists (using BFS). If yes, put it in Group 1; else in Group 2.

Report statistics (running times) separately for the two groups of networks.

Preflow push **may** take longer time on "infeasible" instances, i.e., from Group 2.

A good approach: "layered pipe" network

* Could have, say, $\frac{n}{3}$ layers of nodes.
* t is at bottom layer (ground, or layer 0).
* s at top layer
* Remaining nodes thrown into the intermediate layers.
* A majority of arcs could go downhill 1 layer,
  - a smaller number go 2 layers downhill,
  - some go 3 layers downhill
* Add some arcs going 1 or 2 layers uphill.
* Check if an s-t path exists → BFS search
  - if yes, put into group 1
  - if no, put into group 2

* Once this structure is built, sample u_{ij}'s from $[1, U]$ for $U \in \mathbb{Z}_{>0}$, and vary U.

---

* Can describe procedure as pseudocode or in words.
* Suggest plotting graphs of average running times...

You can report the average run times as functions of n, m, U, varying one parameter at a time.

When comparing to the worst-case time bound, take the constant of 'O' as 1. For SAP max flow, for instance, plot n^2m. (recall, it's run time is $O(n^2m)$).