MATH 566: Lecture 1 (08/25/2026)

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.

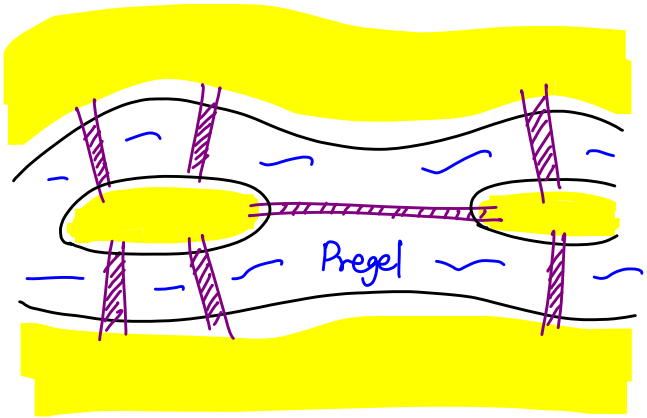
One key difference between "graphs" and "networks" as used conventionally, is that the arcs (or edges) in networks have capacities. In other words, one could think of them as pipes with limits on how much fluid could be sent through them at a time. On the other hand, edges in "graphs" are typically "lines"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

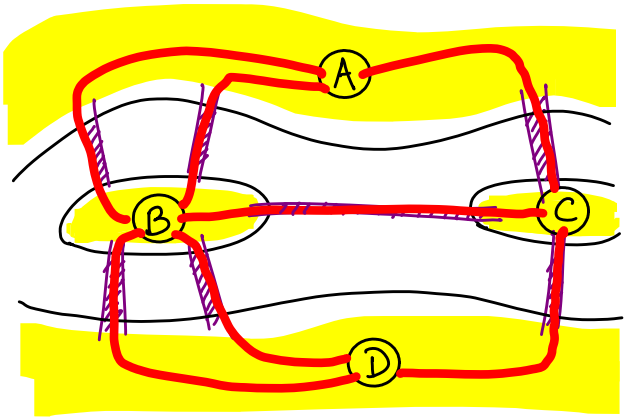
We start will a historical note. The first paper in graph theory (and, by extension, in network optimization) was published by Euler in 1736. He was visiting Königsberg in Prussia (now in Kaliningrad) The river Pregel flows through the town, and had seven bridges across it.

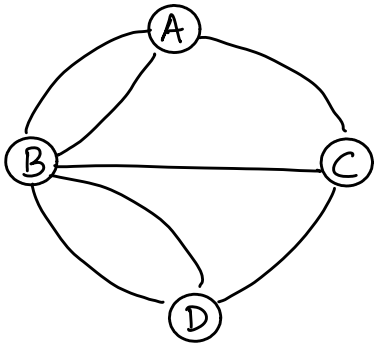


The citizens asked him if one could cross each of the seven bridges exactly once, and return to the starting point?

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"?

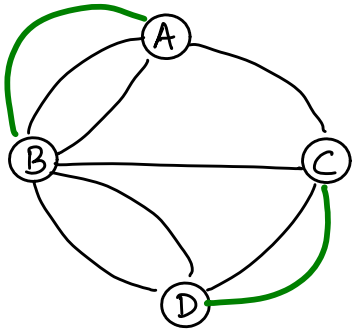
Theorem An undirected graph has an Eulerian cycle iff

* 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.

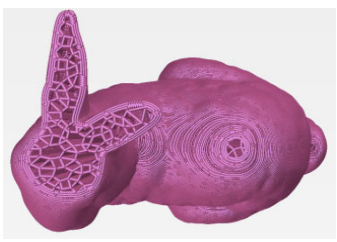
Many results (theorems) we present in this course will have a similar nature – intuitive to grasp and understand, and easy to state (may be not always easy to prove 😐!).

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).

An application of Eulerian cycles in 3D printing: check out https://arxiv.org/abs/1908.07452 !



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.

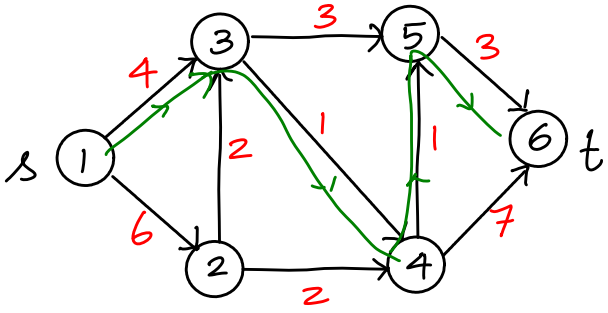
One more point about notation. In graph theory, it is common to call a graph as $G = (V, E)$, where V is the set of vertices and E the set of edges (undirected, by default). We will use the notation $G = (N, A)$ to denote a network, where N is the set of nodes and A is the set of arcs, which are directed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

Network Flow Problems

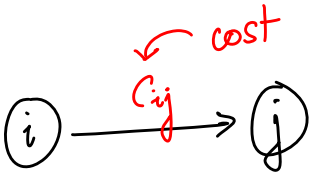
① Shortest path problem

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

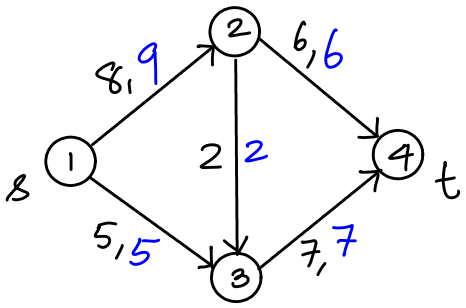
A typical application is driving directions on Google Maps. The costs c_{ij} could just be the distances between the cities modeled by the nodes, or could capture other relevant info such as traffic, tolls, etc.

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).

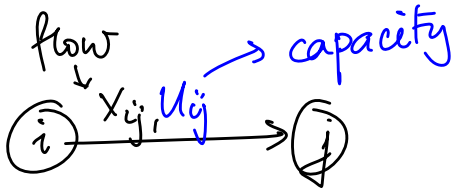
(2) Maximum flow problem (or max-flow)

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 flows (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.

We will present an example of MCF in the next lecture...

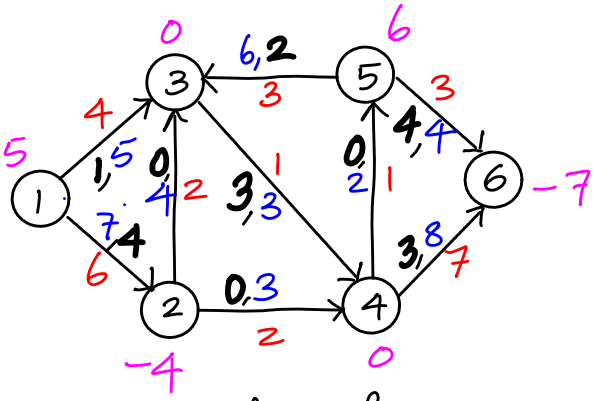
MATH 566: Lecture 2 (08/27/2026)

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

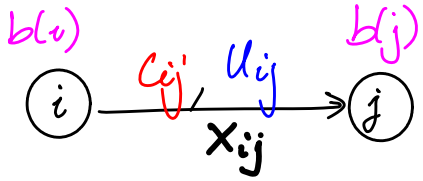
We first finish the introduction to min-cost flow (MCF) problem...

Recall: The objective of MCF is to determine a least cost shipment of a commodity through a network in order to satisfy demands at certain nodes using supply at certain other nodes.

Here is an illustration:



X_{ij}'s shown here form one feasible solution — not necessarily an optimal one.



$b(i)$: 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).

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"

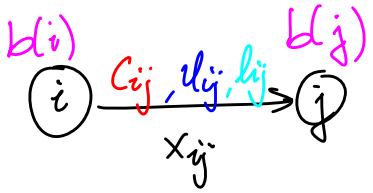
$G = (N, A)$; G: directed network, N: set of nodes, A: set of directed arcs.

$|N| = n$ (# 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

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

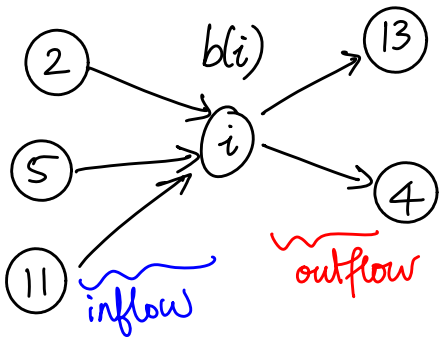
Mathematical notation: $\in$: "element of", $\forall$: "for all", $\sum$: sum.

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.

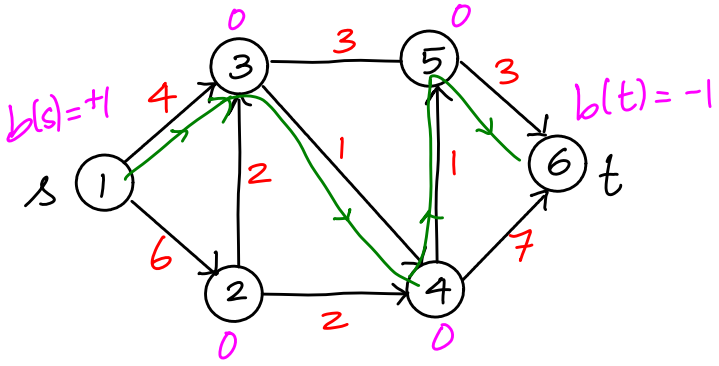
We now show that the shortest path and the max flow problems are special cases of the min cost flow problem.

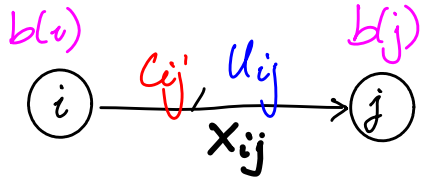
(*) When "formulating" or "modeling" a problem as a network flow problem, you need to specify the network $G=(N,A)$, i.e., what is the node set, the arc set, and what are the associated parameters $(b(i), l_{ij}, u_{ij}, c_{ij})$. (M some big >0 number in practice)

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.



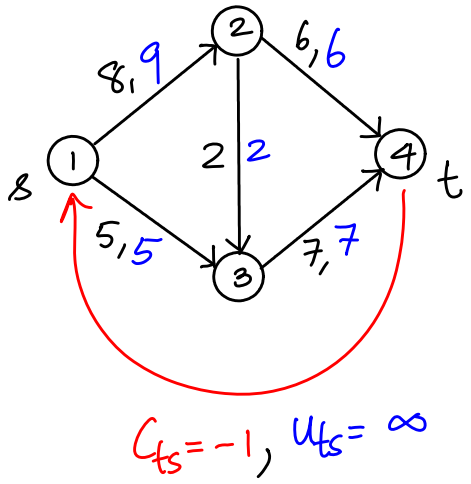


Use same network $G=(N,A)$ as in the shortest path problem. We set $b(s)=+1$, $b(t)=-1$, $b(i)=0$ $\forall i \neq s,t$, $i \in N$. Then we set $l_{ij}=0$, $u_{ij} \geq 1$ $\forall (i,j) \in A$.

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.

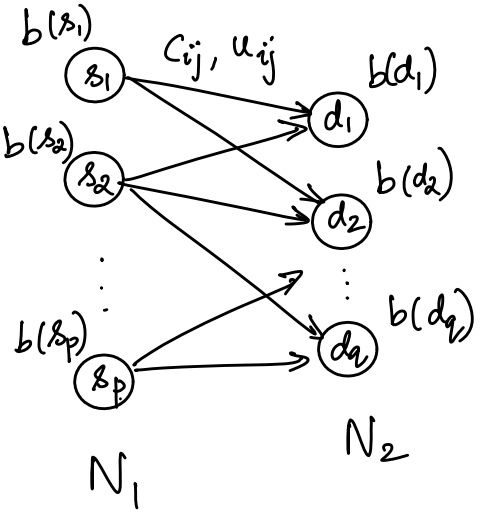
Notice that the flow just circulates around the network here. There is no flow coming in or going out at any of the nodes. This is an instance of the **circulation problem**, which is the min cost flow problem with $b(i)=0$ for all nodes i. This is the fourth network flow problem class (after shortest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

More in the next class...

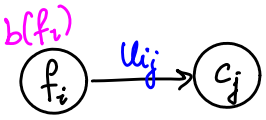
MATH 566: Lecture 3 (09/01/2026)

Today: * seat sharing problem
* assignment problem
* definitions

Seat Sharing Problem

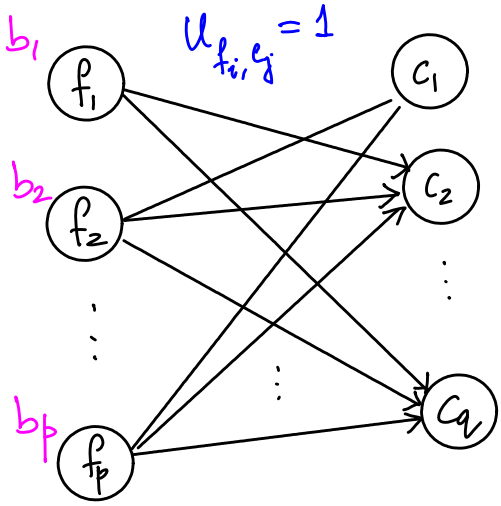
(AMO 1.8, page 21) Several families are planning a shared car trip on scenic drives in the Cascades in Washington. To minimize the possibility of any quarrels, the organizers of the trip want to assign individuals to cars so that **no two members of a family are in the same car**. Formulate this problem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

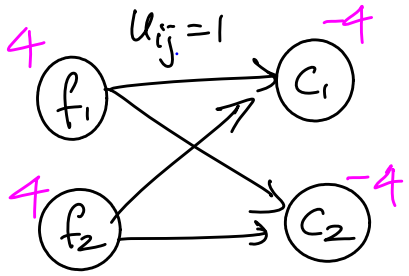
We set $b(f_i) = b_i$, i.e., the supply of family node f_i as b_i.



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 $u_{(f_i, c_j)} = 1$.

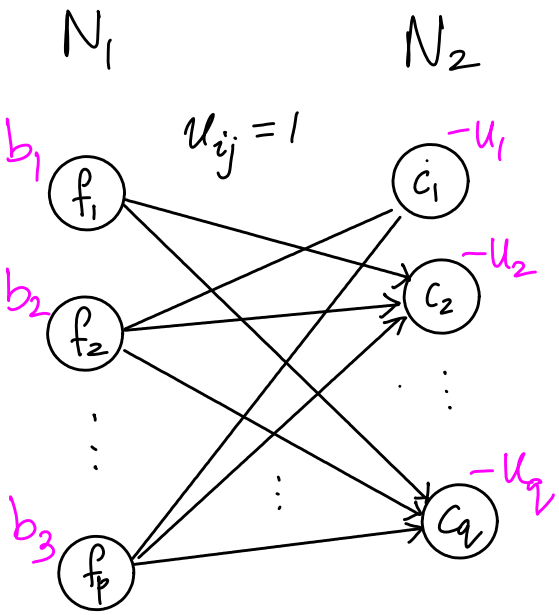
Assumption We assume there exists a feasible assignment of people to cars. There could be trivial instances where we cannot seat all members without avoiding clashes, e.g., see the network below:



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!

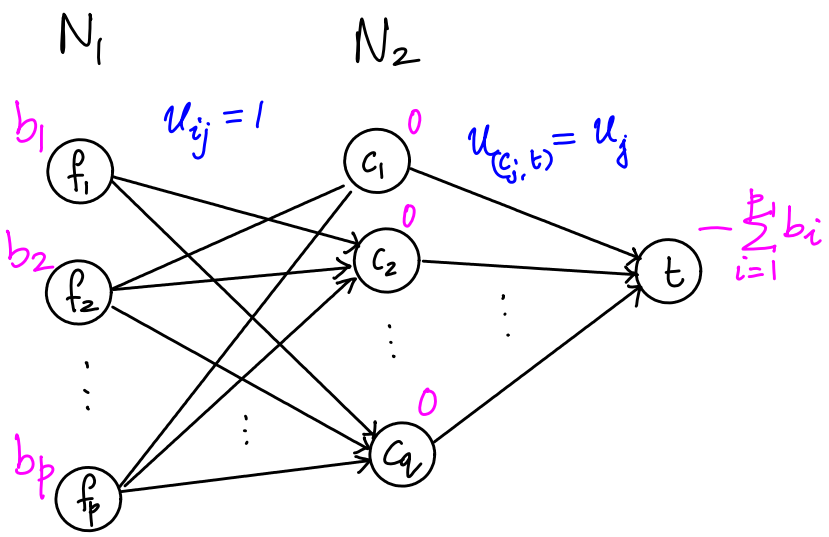
In fact, finding if a given network has a feasible flow can itself be modeled as another network flow problem! More on this topic in a bit...

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.

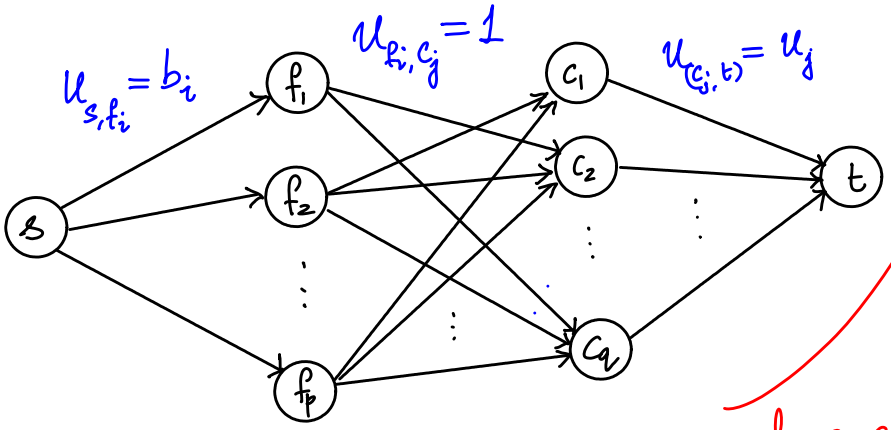


We add a terminus node t, and arcs (c_j, t) with $u_{(c_j,t)} = u_j$, the # seats available in car c_j. We set $b(j) = 0$ for all $j \in N_2$, i.e., car nodes are transshipment nodes. And we set $b(t) = -\sum_i b_i$.

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.

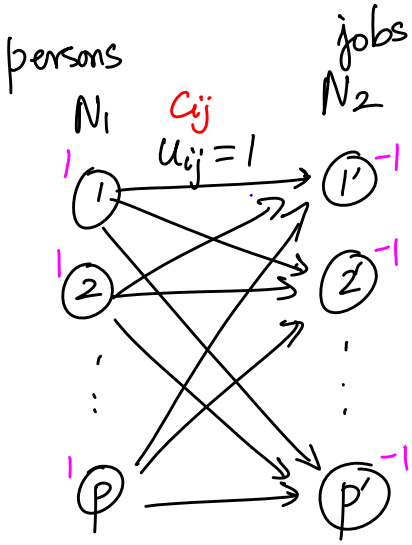
so, each arc (s, f_i) is saturated

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

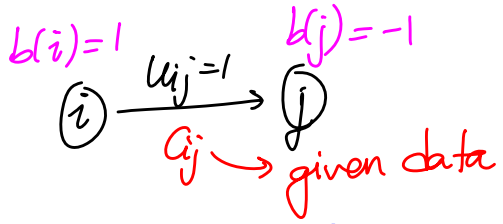
6. Assignment problem

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

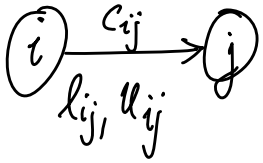
Directed network: edges or arcs of the network are directed

(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).

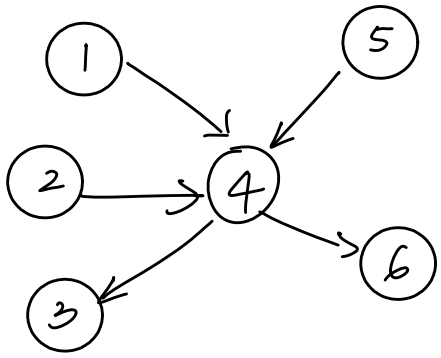
Degree

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).

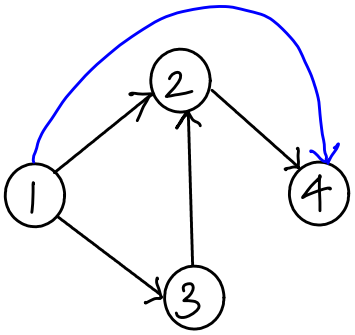
Note that $A(i)$ defines adjacency based on outarcs. We can equivalently define inarc lists $AI(i)$. In practice, it often helps to initialize and use both sets of lists.

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

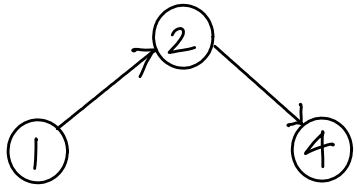
An **induced subgraph** is a subgraph that contains each arc of A with nodes in N', i.e., each such arc is present in A'.

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

Example

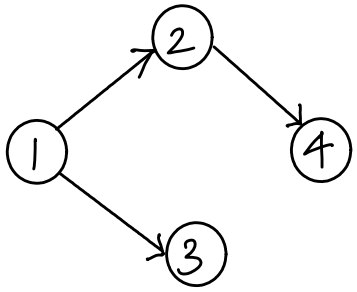


G



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 (09/03/2026)

Today: * More definitions
* Network representations

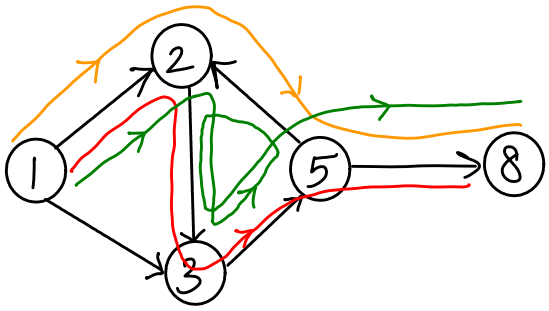
Definitions (contd..)

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 $a_k = (i_k, i_{k+1})$ for $\forall k$.

Here are some examples.

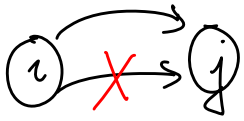


1-2-5-8 is a walk

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

1-2-3-5-2-3-5-8 is also a directed walk, as arcs can be repeated in walks

Some books define a "trail" as a walk that has no repeated arcs (nodes may be repeated).

We assume there are no duplicate arcs: 

.

↳ we will see soon how to handle such cases using network transformations

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 $pred(j) = i$.

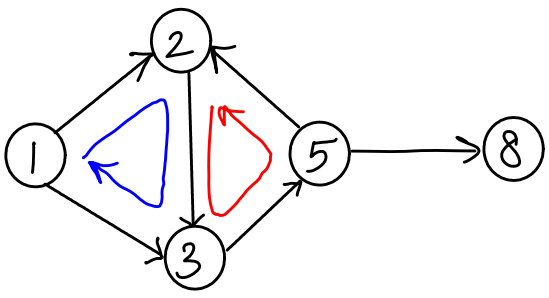
For the directed path 1-2-3-5-8, we set

$pred(8) = 5$
$pred(5) = 3$
$pred(3) = 2$
$pred(2) = 1$
$pred(1) = 0$ → 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.

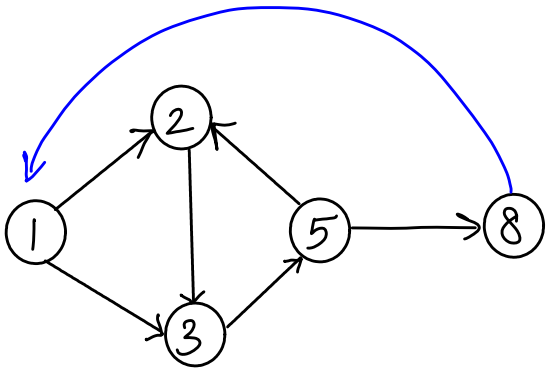
Many problems are more complicated when the network has directed cycles.

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**.

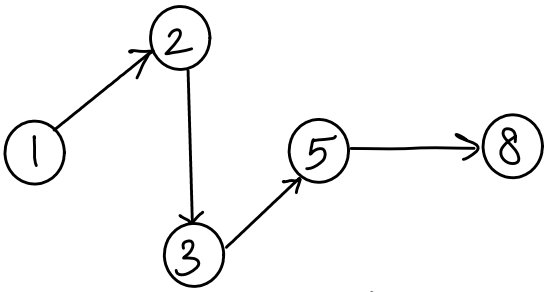
A graph is **strongly connected** if there is a directed path from every node i to every node j.



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

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

e.g., T

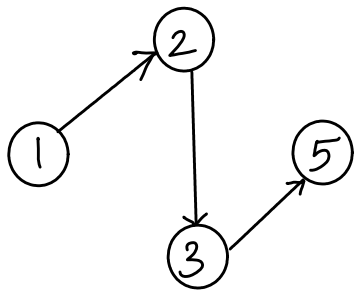


is a tree of the graph G introduced previously.

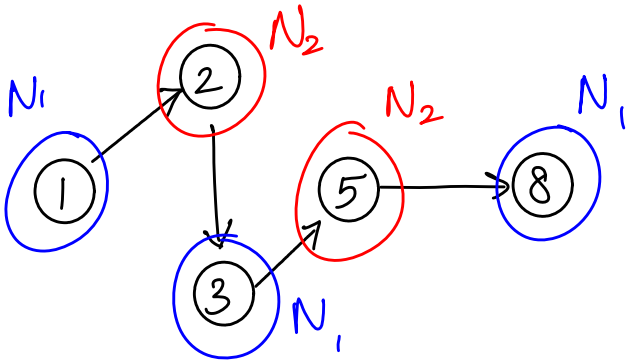
A tree T is a **spanning tree** of graph G if T is a spanning subgraph of G. For instance, T shown above.

But T' here 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 $N = N_1 \cup N_2$ (disjoint union) and each $(i,j) \in A$ has either $i \in N_1, j \in N_2$ or $i \in N_2, j \in N_1$.

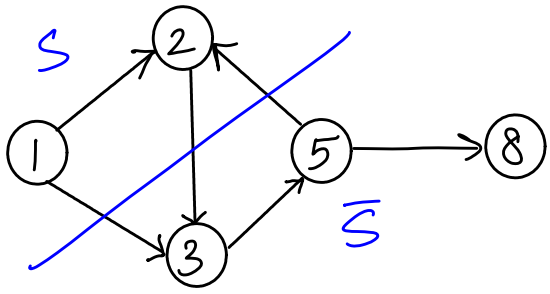


Note: A bipartite graph need not always be "drawn" in an obvious bipartite fashion, e.g., as in the case of transportation problems.

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

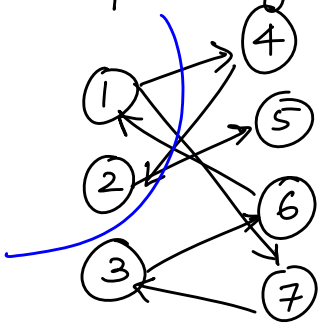


$S = \{1,2\}$, $\bar{S} = \{3,5,8\}$

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

$(5,2)$: backward arc

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



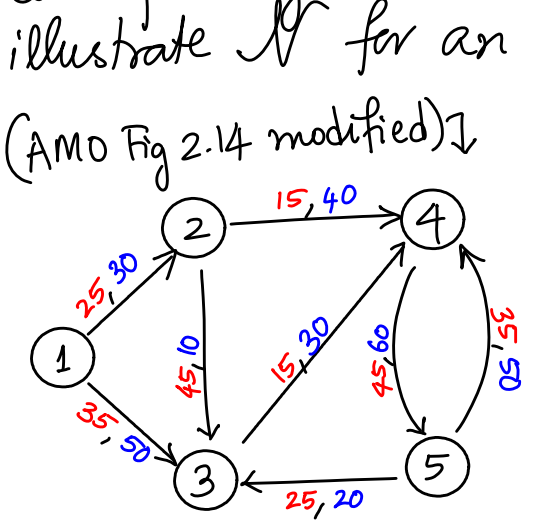
We will talk more about cuts when we introduce max flow in detail.

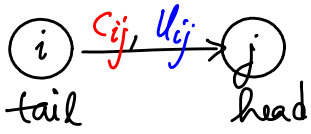
Network Representations

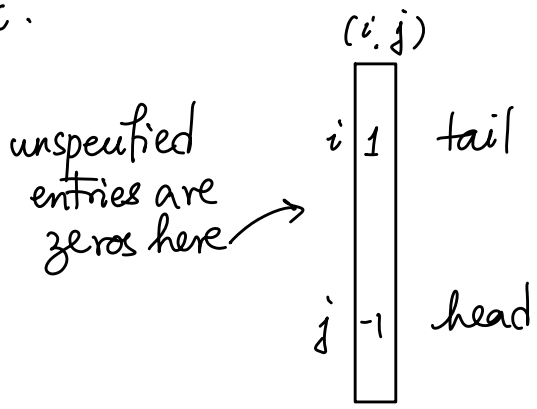
We need to represent $G = (N, A)$, along with data, i.e., $b(i)$, l_{ij}, u_{ij}, c_{ij}, etc. The ultimate goal is to be able to access and use all this info efficiently in algorithms. We consider several options for representing networks now.

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.







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

↑ 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.

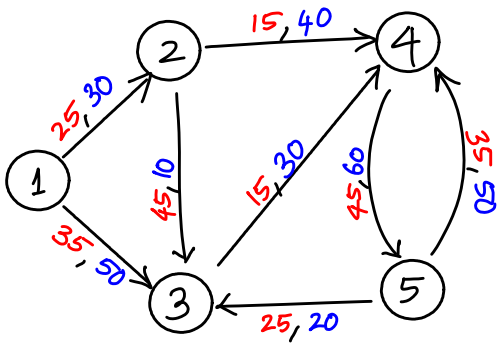
We now introduce another matrix representing the network.

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

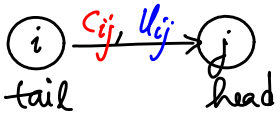
This is an $n \times n$ matrix $\mathcal{H} = [h_{ij}]$, with

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

→ 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 Matlab/Python. 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.

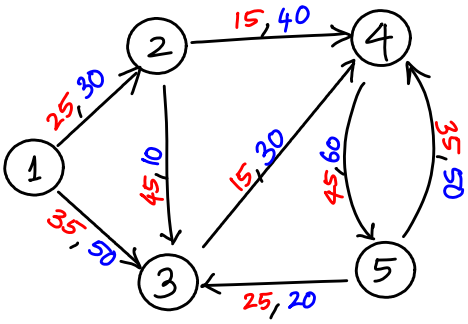
MATH 566: Lecture 5 (09/08/2026)

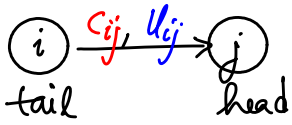
Today: * forward star representation
* network transformations

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 2026)
%% 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 2026)
%%
%% 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)

DEGI
celldisp(AI)
```

Output from Matlab:

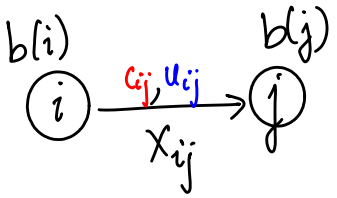
```
% Matlab session
% Lecture 5, Sep 08, 2026

>> 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

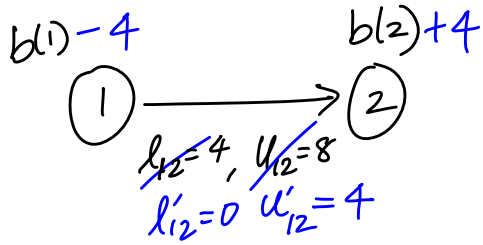
$$\sum_{(i,j)} (x'_{ij} + l_{ij}) - \sum_{(j,i)} (x'_{ji} + l_{ji}) = b(i)$$

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

$b'(i)$: 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 (09/10/2026)

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.

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

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.

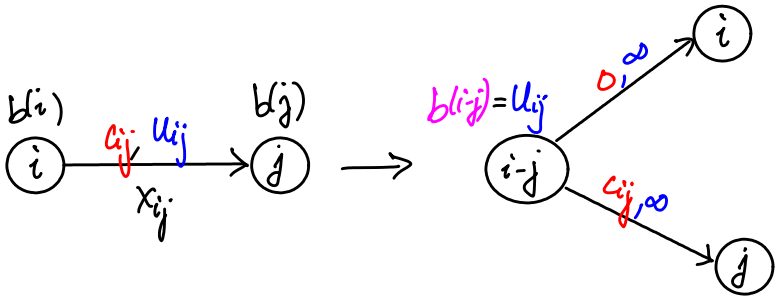
Q: What if all c_{ij} are < 0 to start with? Could we add a constant value to all c_{ij}'s so that they all become ≥ 0?

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

③ Removing upper bounds

($l_{ij} = 0 \ \forall (i,j)$ is assumed)
else we can use the transformation we introduced previously

We 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:

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

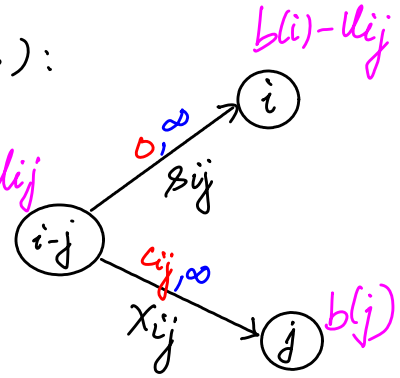
(s_{ij}: "slack")

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, once as outflow and once as inflow.

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

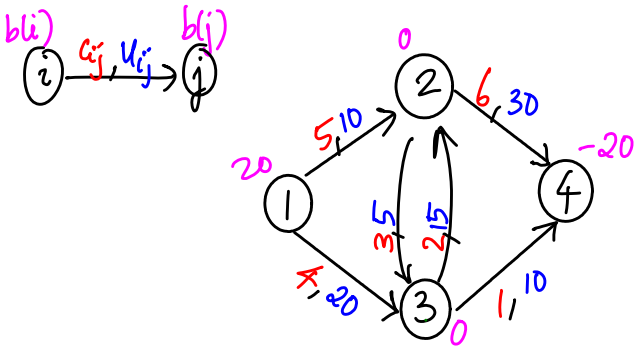
We consider the equivalent system (1'), (2), (3):

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

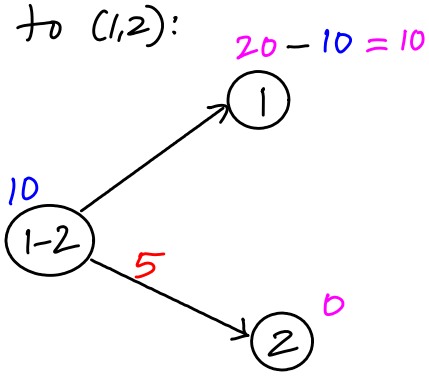


x_{ij} and s_{ij} are outflows from a new node with $b(\cdot) = u_{ij}$; and they flow into j and i, respectively.

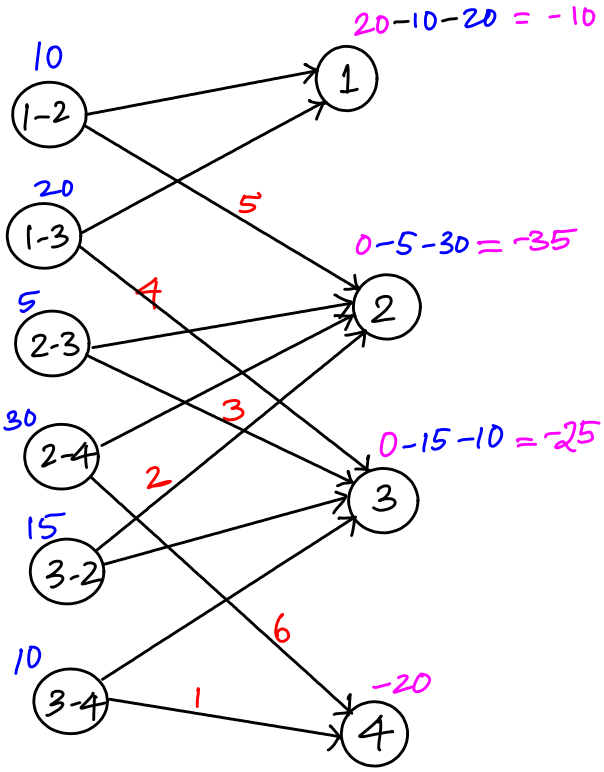
We could apply this transformation on multiple arcs together. Consider the following network:



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 $c_{ij}x_{i-j,j} = c_{ij}x_{ij}$ 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.

We can recover flows from any node i to a node j in the original network from the solution to the MCF problem on the modified network.

4. Node splitting

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

Split → Replace

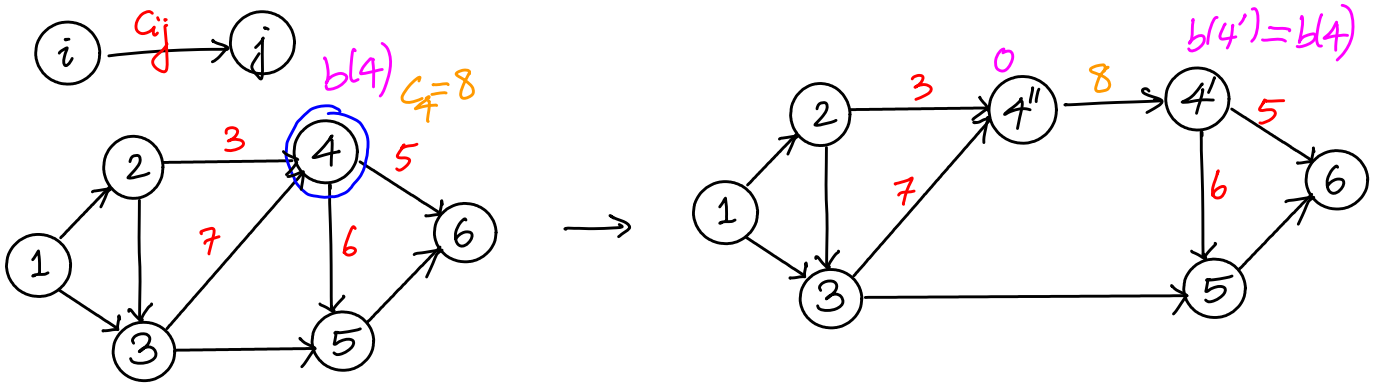
Steps: Replace node i with two nodes i'' and i', add arc (i'', i').

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 = ... j = ...)

What is the "running time" of this algorithm?

→ The number of steps (or operations) as a function of m and 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

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 $f(n, m, p, \ldots)$ 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 $f(n) \leq c\,g(n) \;\forall\; n \geq n_0$.

→ 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.,

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

MATH 566: Lecture 7 (09/15/2026)

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 $f(n,m,p,\ldots)$ 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 $c'(>0, \text{real})$ 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.

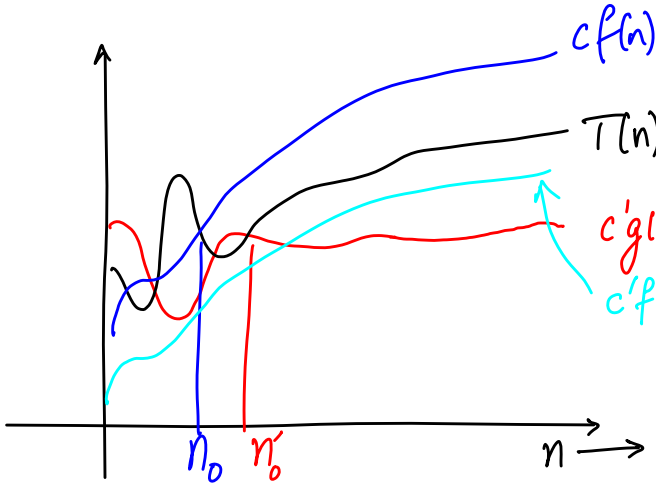
Def A function $f(n)$ is $\Omega(g(n))$ if there exist numbers $c'>0$ and $n_0'>0$ such that $f(n) \geq c'g(n)$ $\forall$ $n \geq n_0'$.

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 $s \in N$.

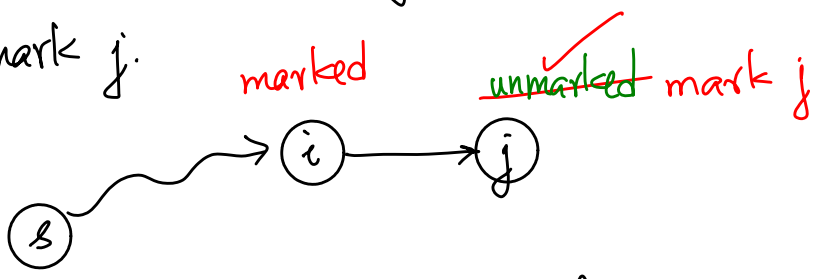
OUTPUT: $\{j \in N \mid$ a directed path exists in G from s to $j\}$.

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:

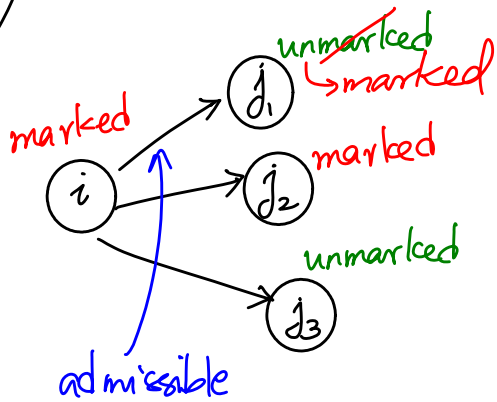
order(s)=s; works only for s=1.

```
algorithm search;
begin
    unmark all nodes in N;        ← maintain a binary n-vector (array)
    mark node s;
    pred(s) := 0;
    next := 1;                    ← counter
    order(s) := next;             → order(s)=2 ⇒ node 5 is the 2nd node visited
    LIST := {s}
    while LIST ≠ ∅ do
    begin
        select a node i in LIST;
        if node i is incident to an admissible arc (i, j) then   → look at A(i) list
        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;
```

initialization

Figure 3.4 Search algorith

→ 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.

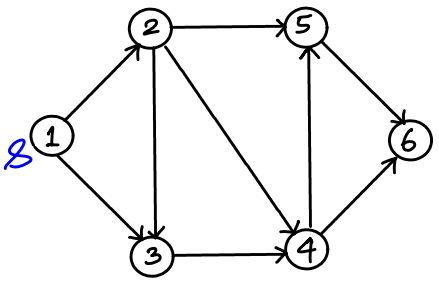
If we maintain LIST as a queue, i.e., select node i from the front, add node j to back of LIST we get **breadth-first search** (BFS). The nodes are examined in a **first-in first-out** (FIFO) order.

MATH 566: Lecture 8 (09/17/2026)

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

Example

Slightly modified version of AMO Fig 3.5.



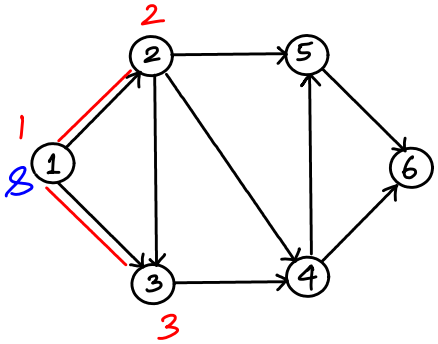
order: i

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 2 3 4 5 6
1 0 0 0 0 0]
Pred = [0 1 M M M M]
Order = [1 2 0 0 0 0]
next → 2
LIST = [1 2]

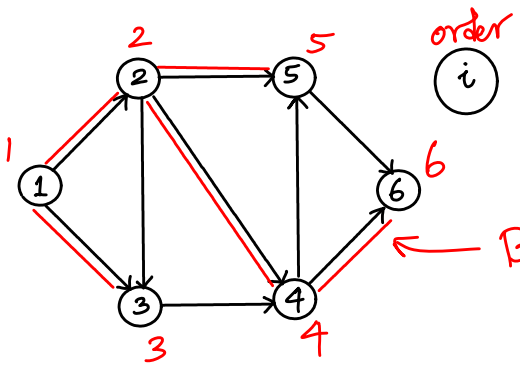
in general, we set pred(j) = i;

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 2 3 4 5 6
1 1 0 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)



Mark = [s, 1, 1, 1, 1, 1]
pred = [N, 0, 1, 1, 2, 2]
order = [1, 2, 3, 4, 5, 6]

LIST = [1] → [1 2] → [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

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

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

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.

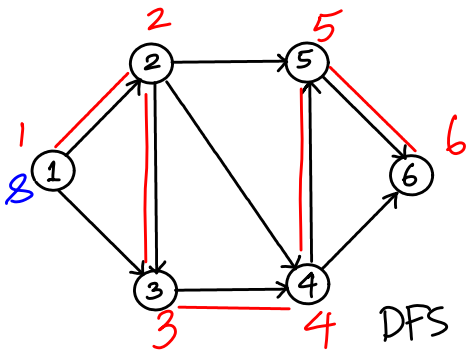
Property In the BFS tree, the directed path from s to i contains the smallest number of arcs among all directed s-i paths, i.e., it is a shortest path in terms of # arcs.

→ 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] → ∅

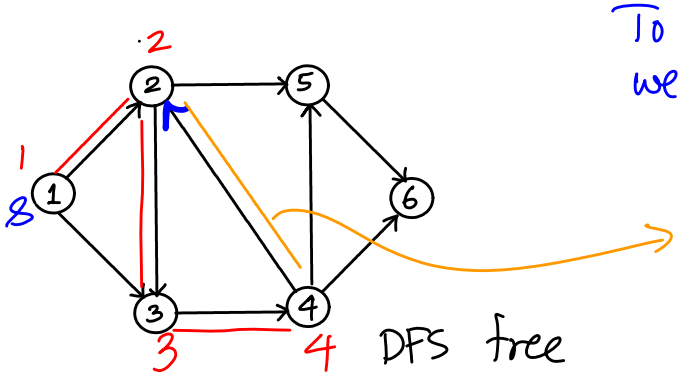
delele node in this order

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.

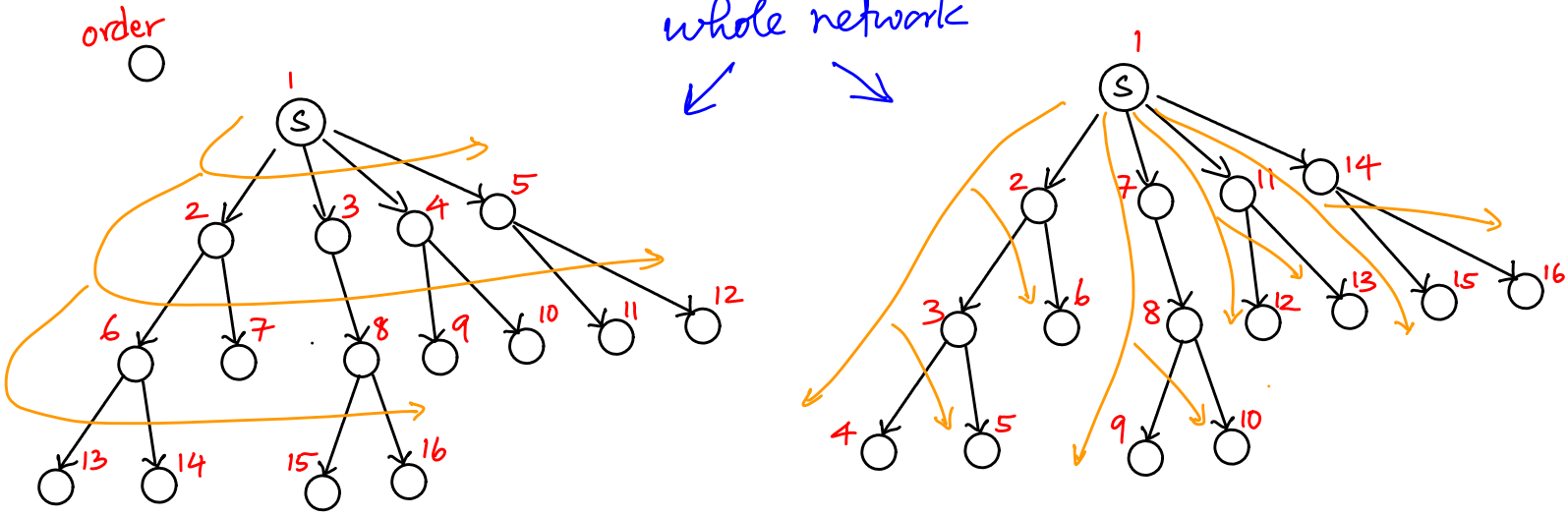
Q. How do BFS and DFS compare on large networks in general?

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 (via directed path) a node t.

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.)

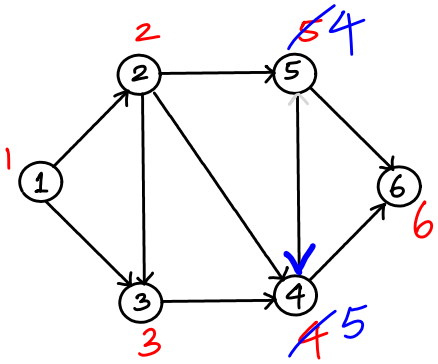
It is indeed sufficient to do this paired search from a single node (s). For any node pair i,j, we could go from i to s and s to j.

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 (09/22/2026)

Today: * algo for topological ordering
* flow decomposition

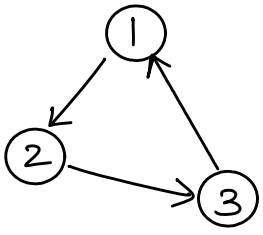
Topological Ordering

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;
```

initializing indegrees

counter

different from search!

≡ "deleting" (i, j)

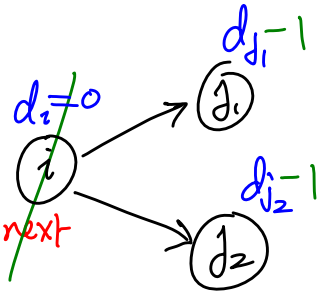


Figure 3.8 Topological ordering algorithm.

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

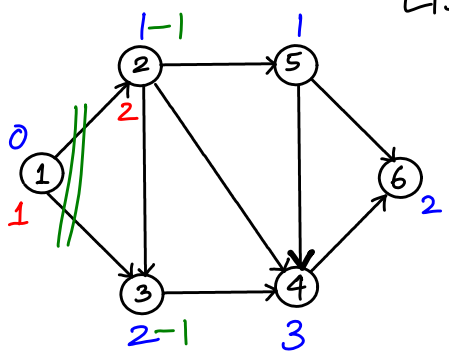
Using similar arguments as used for search, we can see that topological ordering runs in $O(m)$ time.

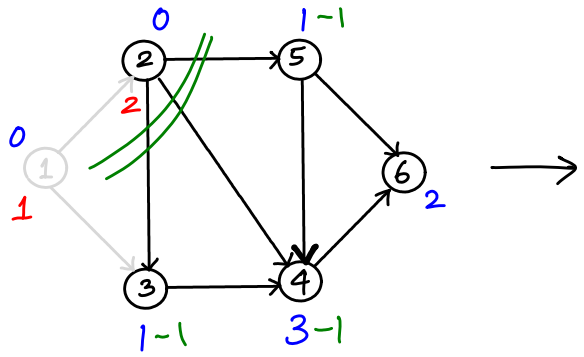
Illustration

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.

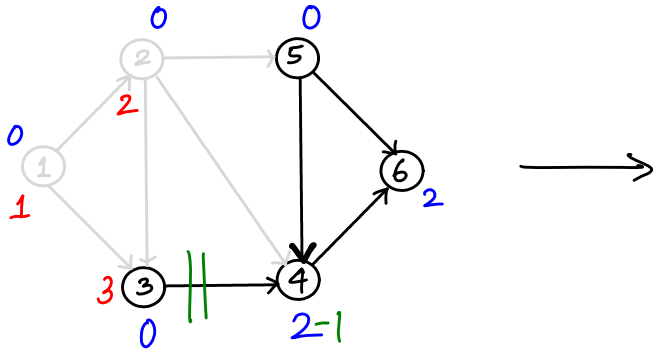
indegree
i
order

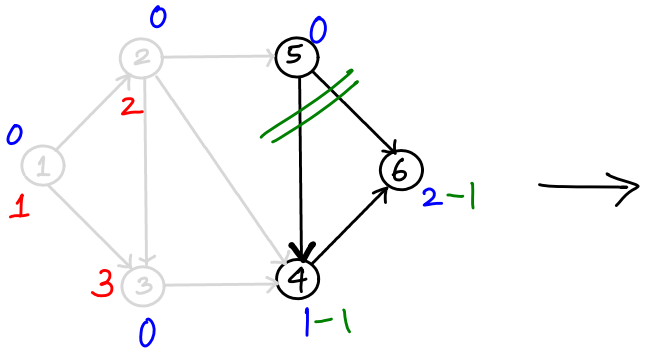
LIST: [1] → [7]



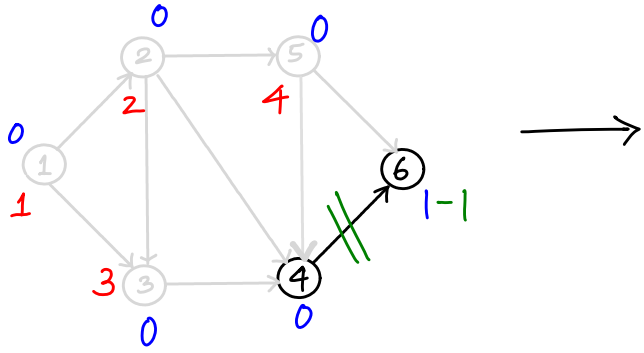


LIST: [3 5] → [5] → [4]

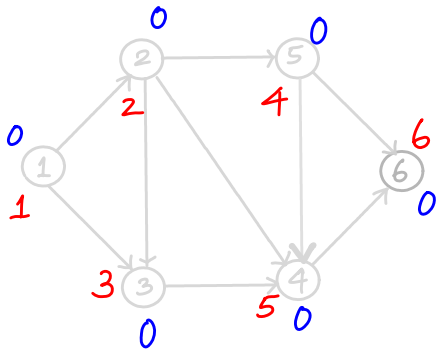




LIST: [4] → [6]

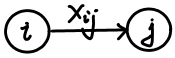


LIST: [6]

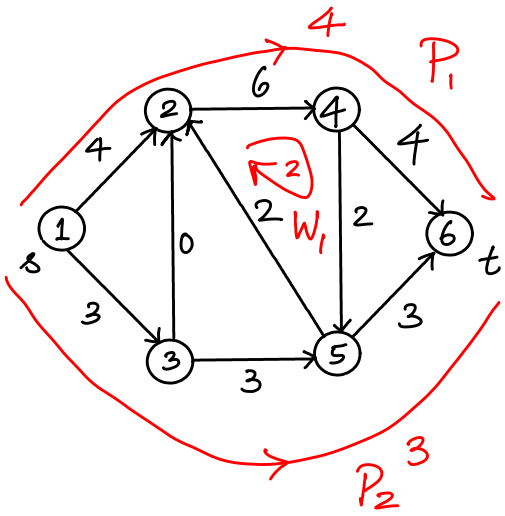


Flow Decomposition

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.



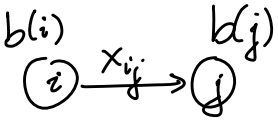
Instead, we could consider flows along two paths and a cycle.

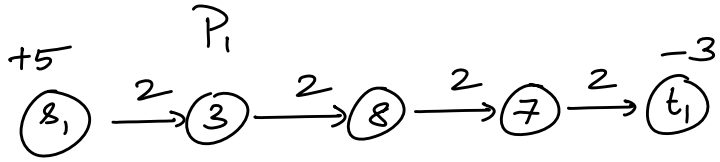
P_1: 1-2-4-6 sending 4 units

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

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

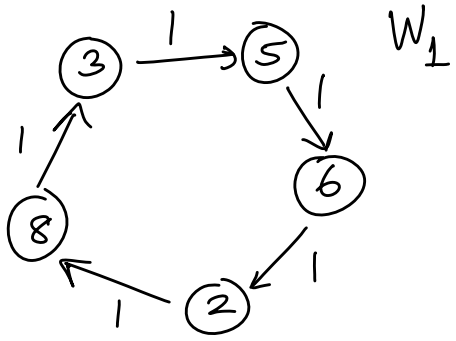
Let P_1 be a path from s_1 to t_1 as shown here:





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:

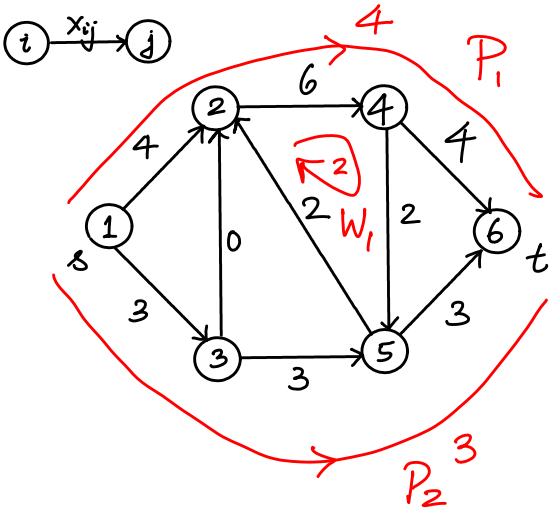
$$x_{ij} = \sum_{P \in \mathcal{P}} f(P)\delta_{ij}(P) + \sum_{W \in \mathcal{W}} f(W)\delta_{ij}(W) \quad \forall (i,j) \in A.$$

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

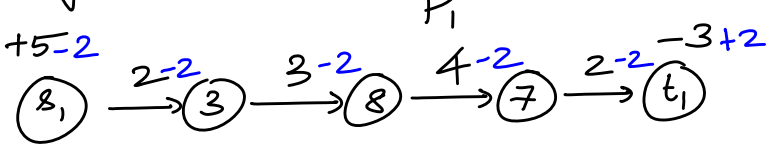
How much can we push? Consider the path $P = s - i_1 \cdots i_r - t$.

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

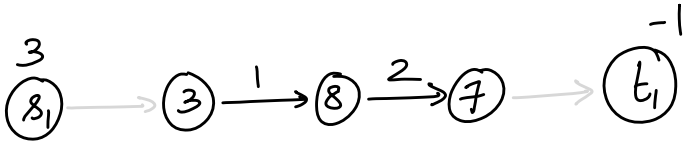
($\Delta(P)$ → "capacity" of P)

e.g., let P_1 be

P_1



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



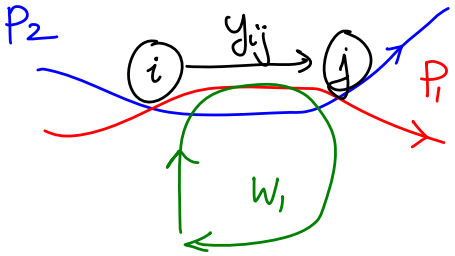
Set $f(P_1) = \Delta(P_1) = 2$.

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

MATH 566: Lecture 10 (09/24/2026)

Today: * flow decomposition
* shortest path problem – assumptions, applications, algo

Recall: Flow decomposition...



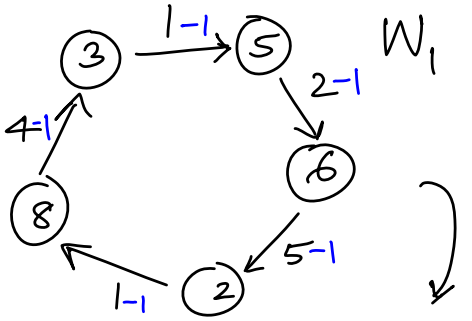
Note that (i,j) can be part of multiple paths and/or cycles, but it is always a forward arc in each path and cycle.

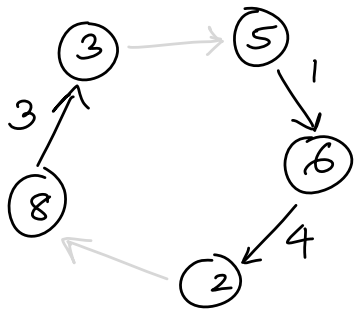
For a cycle W, we set $\Delta(W) = \min\{y_{ij} \mid (i,j) \in W\}$.
↳ capacity of cycle W

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.



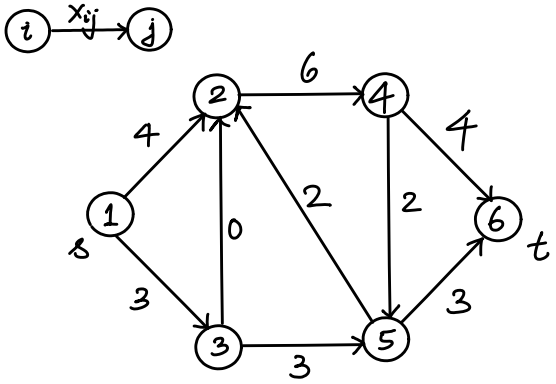


We run DFS repeatedly. If we find a directed cycle, we push its capacity of flow. Else if we find a path from a supply to a demand node, we push its capacity. We update the intermediate flows and the network, and continue. See the algorithm in the handout posted on the course web page.

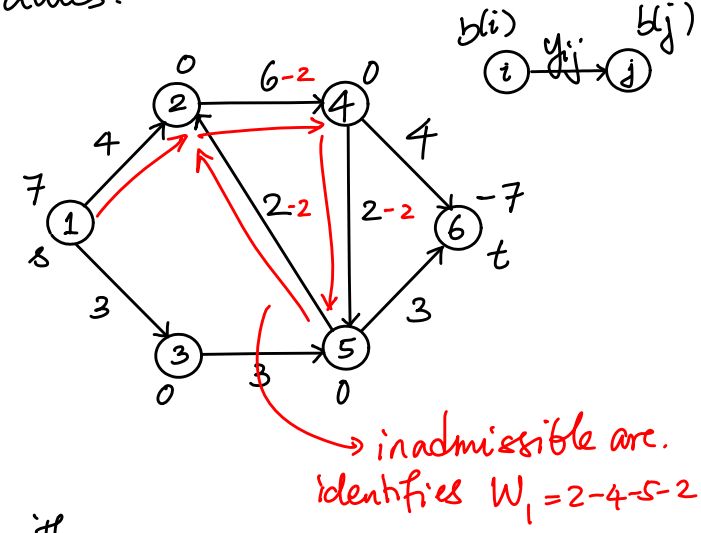
10.2

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.

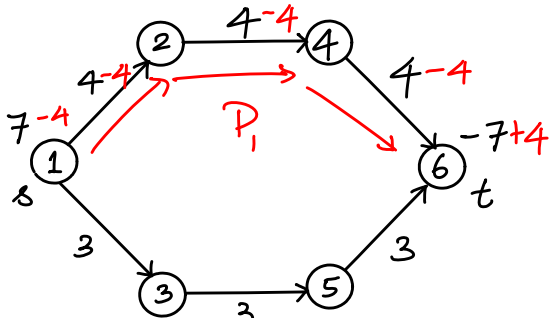
Start with $s = 1$ (taken from $\mathcal{S}$)

We perform DFS. We identify cycle $W_1 = 2-4-5-2$. 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 $\mathcal{S} = [1]$, $\mathcal{D} = [6]$.



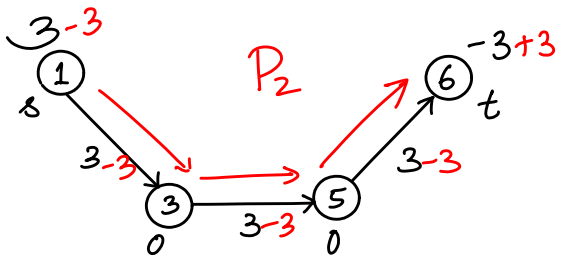
Starting with $s = 1$, we do DFS.

We identify $P_1 = 1-2-4-6$, with

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

10.3

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



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're 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 $f(W)$. Eventually, we want to get $b(i) = 0 \ \forall i \in N$ and $y_{ij} = 0 \ \forall (i,j) \in A$, 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.

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 at most $m+n$ directed paths and cycles, out of which at most m are cycles.

IDEA 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.

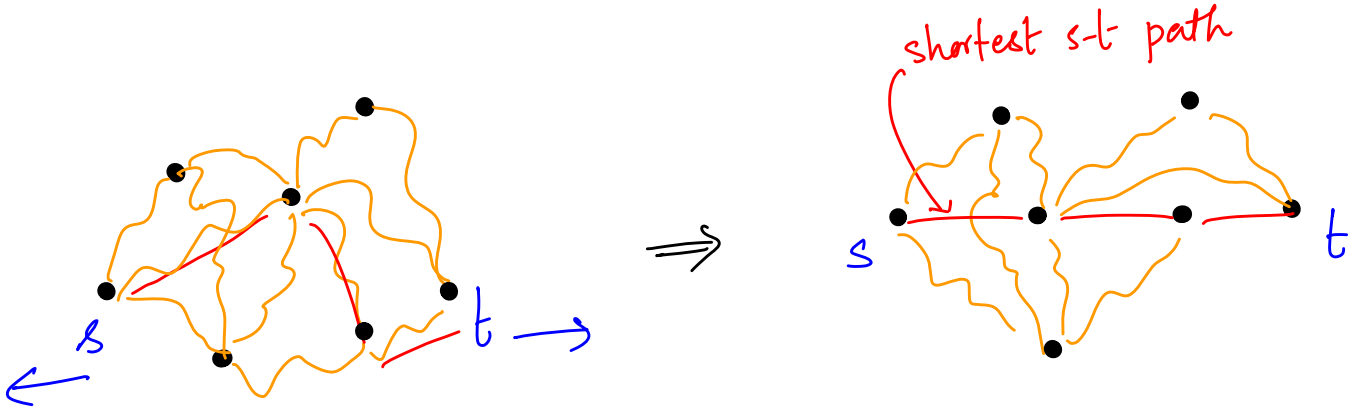
We first consider the case where all $c_{ij} \geq 0$. This case will be easier than the general case that allows $c_{ij} < 0$. As it turns out, we would use similar ideas to solve both cases.

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)

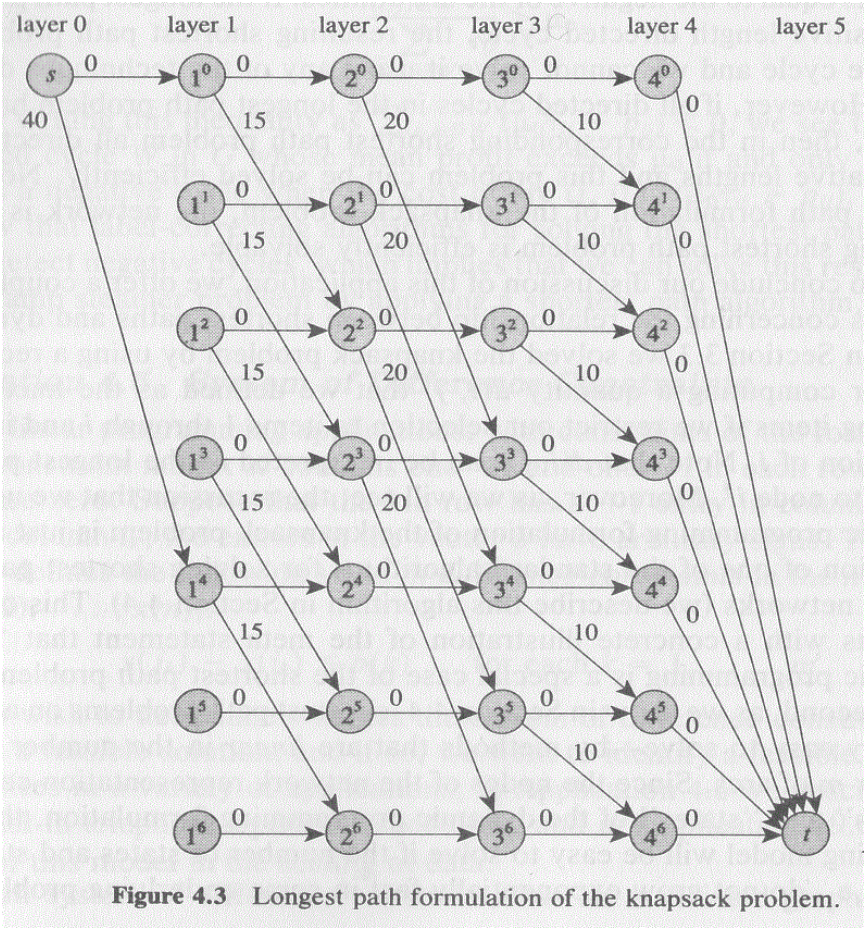
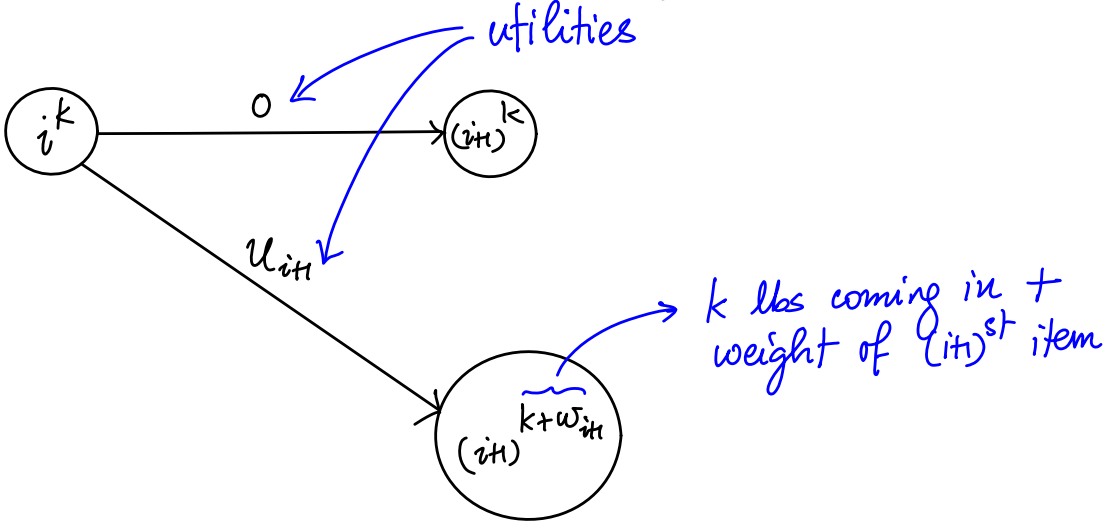


Figure 4.3 Longest path formulation of the knapsack problem.

The figure illustrates a longest path formulation of this problem.

10.8

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.

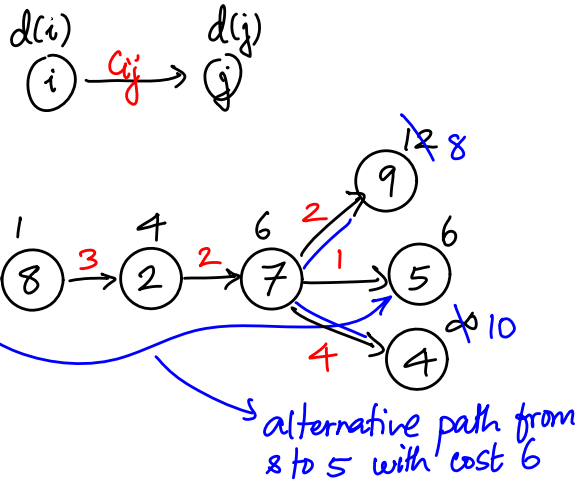
The knapsack problem has many applications, including in cryptography, multiprocessor scheduling, fair division, etc. The model illustrated here (with layers corresponding to each item and levels corresponding to the total weight) is a typical instance of dynamic programming. Hence, under mild assumptions, 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) ∈ 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 outarcs of node i.