网络建模与优化

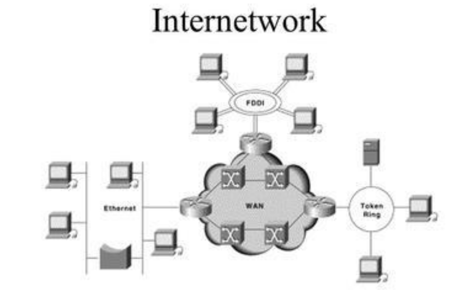
Network modelling and optimization

授课：王晓蕾

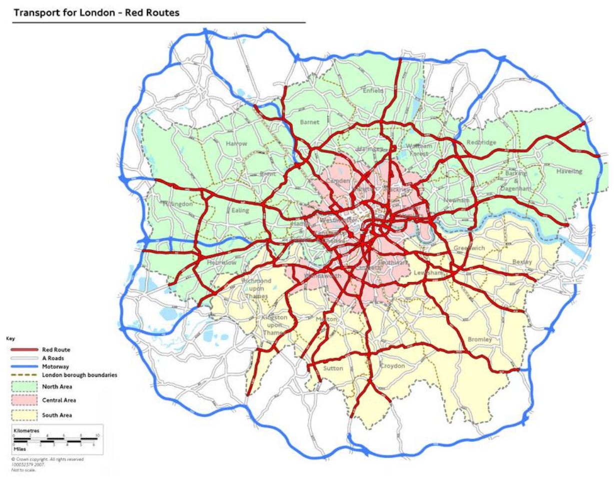
Email：xiaoleiwang@tongji.edu.cn

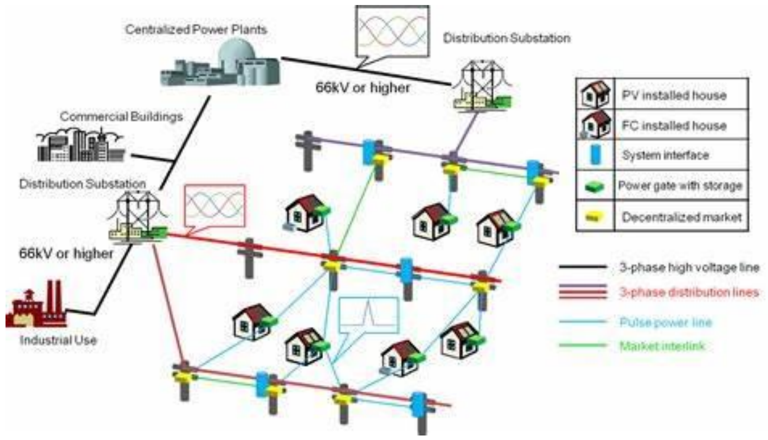
Office: 同济大厦A座1419

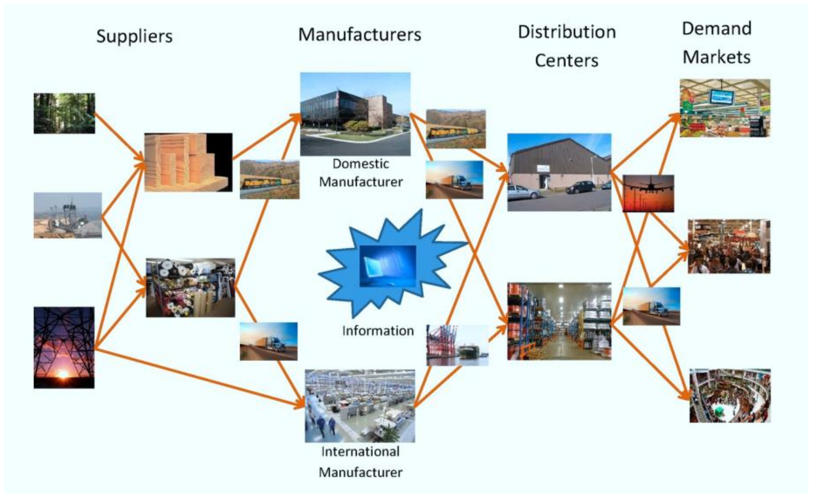
网络无处不在



- An *internetwork* is a collection of individual networks, connected by intermediate networking devices, that functions as a single large network.


Transport for London - Red Routes
Key
Red Route
A Roads
Motorway
London borough boundaries
North Area
Central Area
South Area
Kilometres
Miles
© Crown copyright. All rights reserved 100032379 2007.
Not to scale.





课程内容

第一章 初识网络 （Introduction to network）

第二章 网络中的若干优化问题

最短路径问题（Shortest path problem)

最大流问题 （Maximum flow problem)

最小费用流问题 （Minimum cost flow problem)

第三章 交通流分配问题（Traffic assignment problem）

用户均衡 (User equilibrium)

贝尔曼转化（Beckmann’s transformation）

交通流分配算法（Traffic assignment algorithms）

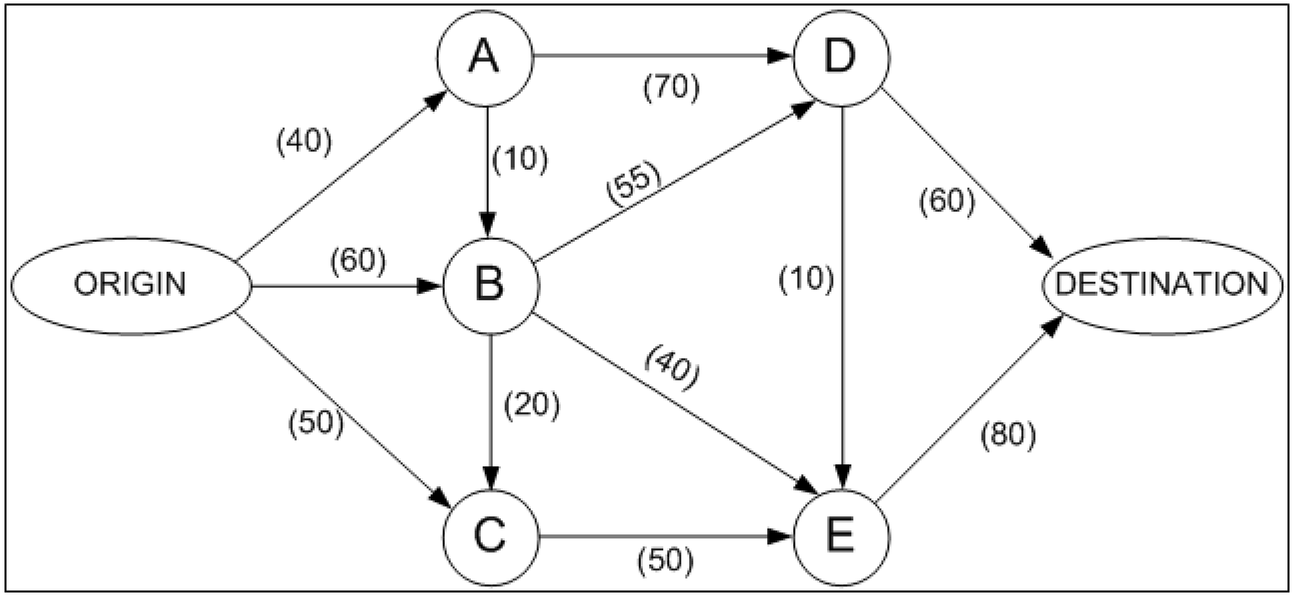
最短路径问题

- 给定每个点之间的距离，寻找点到点的最短距离

终
起
1 推荐
2 方案
3 方案
曹家渡
百富商务楼
中山公园
SOHO东海
东方华发大厦
沪纺大厦
长征医院
上海博物馆
华山医院高架直升机场
复兴公园
上海市皮肤病医院
延安西路
新华路
岳阳
绍兴公园
鲁班路
天亿大厦
肇嘉浜路

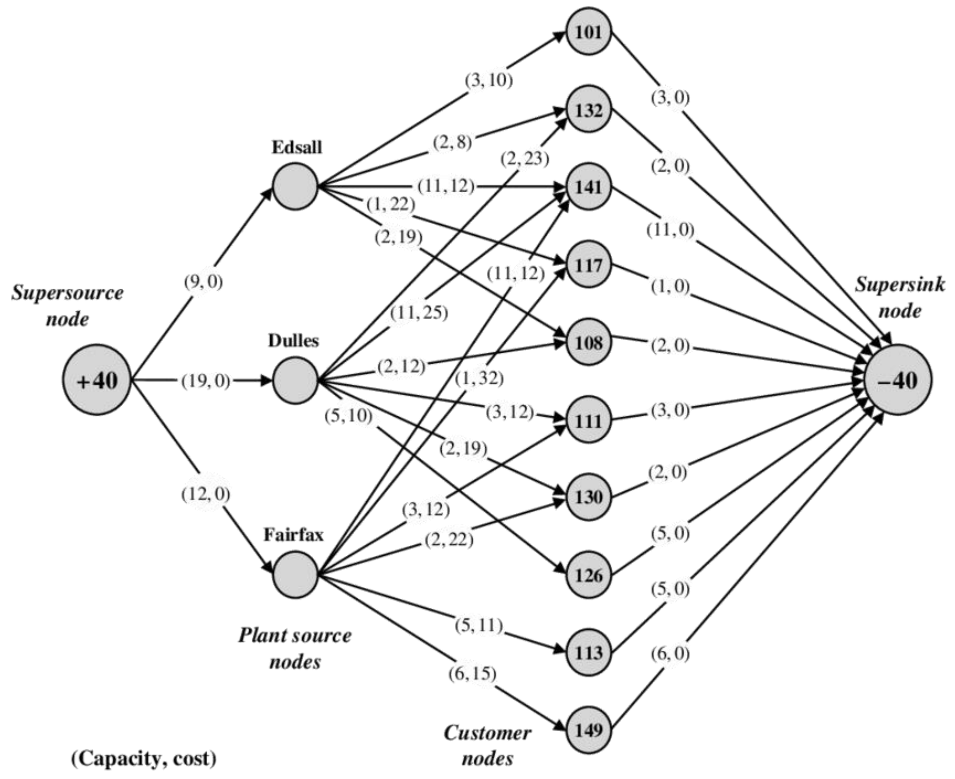
最大流问题

- 给定每个点之间的最大运输能力，寻找点到点的最大运力



最小费用流问题

- 给定供给、需求和每个点之间的运输成本和运力，找到成本最小的供给输送方案以满足所有需求。



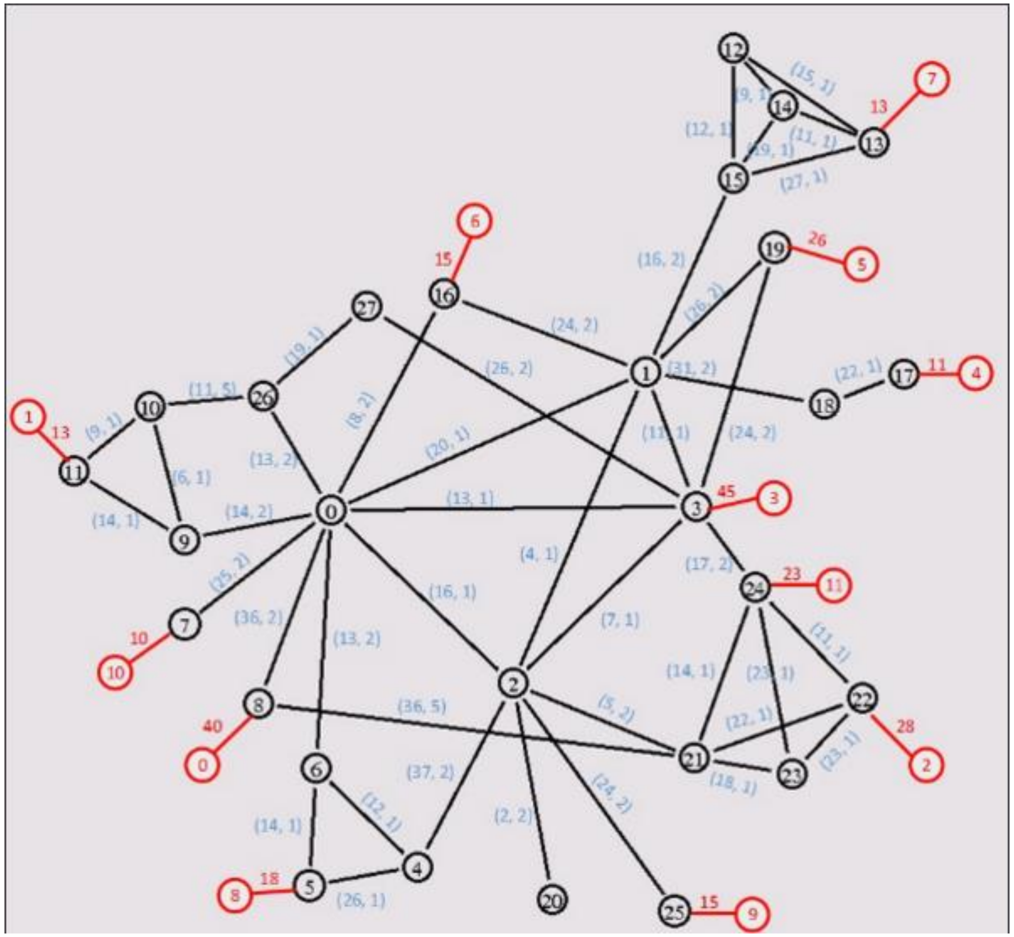
2016华为软件精英挑战赛初赛题目

给定一个带权重的有向图G=(V,E)，V为顶点集，E为有向边集，每一条有向边均有一个权重。对于给定的顶点s、t，以及V的子集V'，寻找从s到t的不成环有向路径P，使得P经过V'中所有的顶点(对经过V'中节点的顺序不做要求)。

若不存在这样的有向路径P，则输出无解，程序运行时间越短，则视为结果越优；若存在这样的有向路径P，则输出所得到的路径，路径的权重越小，则视为结果越优，在输出路径权重一样的前提下，程序运行时间越短，则视为结果越优。

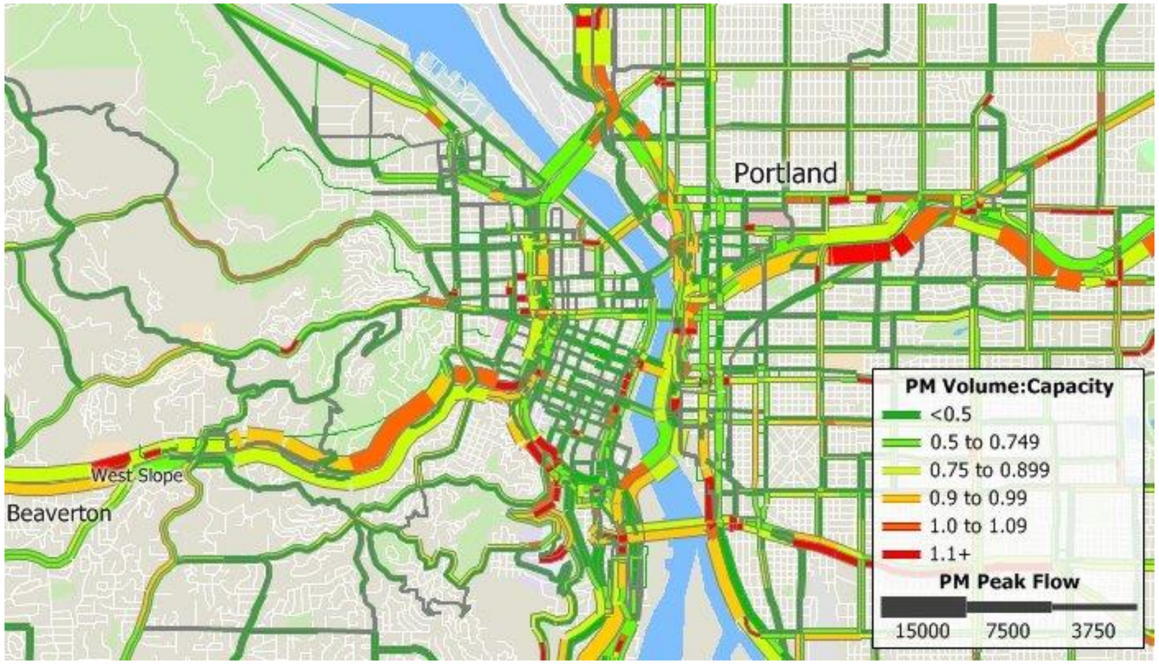
2017华为软件精英挑战赛初赛题目

在给定结构的G省电信网络中，为了视频内容快速低成本的传送到每个住户小区，需要在这个给定网络结构中选择一些网络节点附近放置视频内容存储服务器。需要解决的问题是：在满足所有的住户小区视频播放需求的基本前提下，如何选择视频内容存储服务器放置位置，使得成本最小。



交通流分配问题

- 在一个给定的路网中，存在着大量的独立决策的个体。每个用户都希望以最小的成本从自己的起点到达终点，而每条边上的通行时间是流量的函数，则在用户均衡状态下路网中会形成怎样的流量分配？



课程目标

通过对网络结构下经典交通和物流问题的启发式教学，培养学生数学建模、问题转化和优化求解的能力，为学生进一步从事相关的理论研究或投入相关的实际工作提供必要的思维训练和理论基础。

考核方式

课堂表现 40%

课堂测试 20%

课程项目 40%

参考书

AHUJA, Ravindra K. **Network flows: theory, algorithms, and applications.** Pearson Education, 2017.

SHEFFI, Yosef. **Urban transportation networks.** Prentice-Hall, Englewood Cliffs, NJ, 1985.

第一章 初识网络

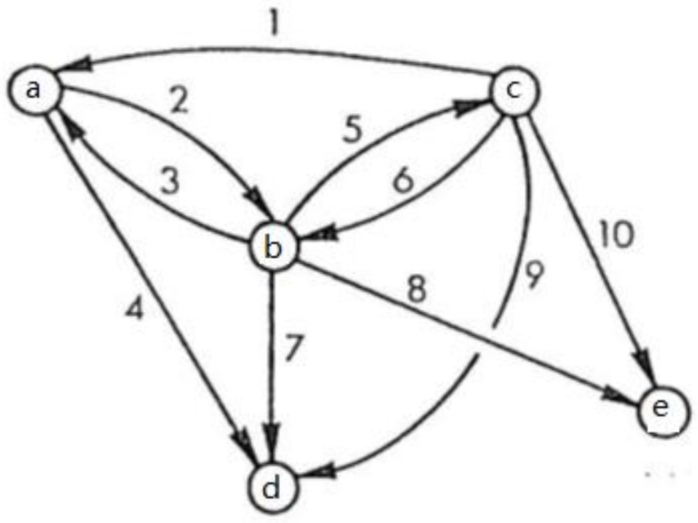
1. 网络中的基本概念
2. 网络算法基础
    a) 网络搜索
    b) 拓扑排序
3. 网络的存储
4. 算法复杂度

1. 网络中的基本概念

Network

GRAPH: A **graph** *G* consists of a set *N* of **nodes** (**vertices**) and a set *A* of **arcs** (**edges** or **links**). Nodes represent potential sites and customers. Arcs represent transportation segments, and they may be directed or undirected.

NETWORK: A **network** is a graph with additional information about arcs (capacity, distance, cost, etc.), c_{ij}, d_{ij}



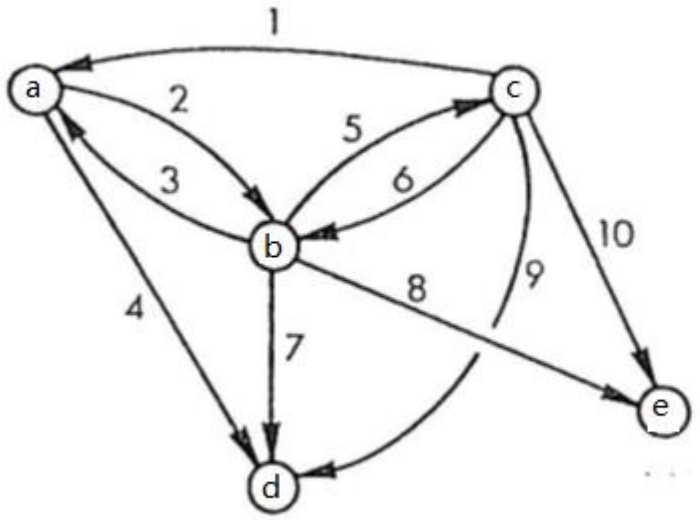
Directed network of links and nodes

Network

Tails and **Heads**: A directed arc (i,j) has two endpoints i and j. We refer to node i as the **tail** of arc (i,j) and j as the **head** of (i,j). We say that the arc (i,j) emanates from node i and terminates at node j. An arc (i,j) is incident to nodes i and j. The arc (i,j) is an outgoing arc of node i and an incoming arc of node j.

Degrees: The **indegree** of a node is the number of incoming arcs of that nodes and its **outdegree** is the number of outgoing arcs. The **degree** of a node is the sum of its indegree and outdegree.

Degree of node b=6



Network

Adjacency List: The **arc adjacency list** A(i) of a node i is the set of arcs emanating from that node, that is, A(i)={(i,j)∈A:j ∈N}. The **node adjacency list** A(i) is the set of nodes adjacent to that node, that is A(i)={j ∈N: (i,j)∈A}.

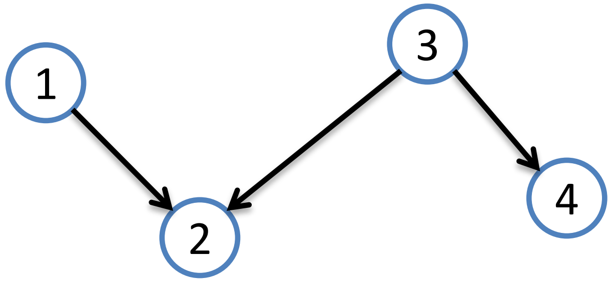
|A(i)|=outdegree of node i

Subgraph: A graph G'=(N',A') is a subgraph of G=(N,A) if $N' \in N$ and $A' \in A$. We say that G' is the subgraph of G induced by N' if A' contains each arc of A with both endpoints in N'. The graph G' is a spanning subgraph of G if N'=N and A'⊆A.

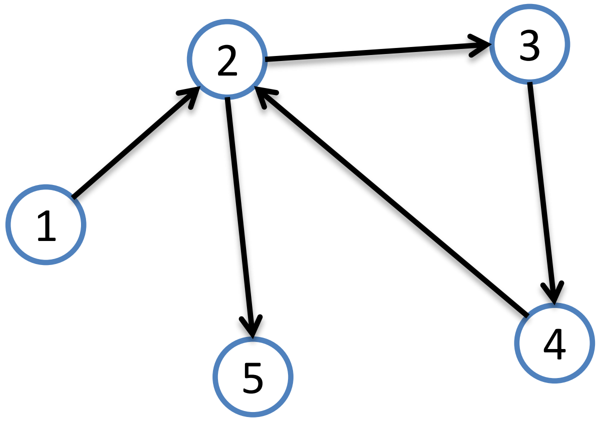
Network

Walk: A walk in a directed graph is a subgraph of G consisting of a sequence of nodes and arcs $i_1 - a_1 - i_2 - \cdots - i_{r-1} - a_{r-1} - i_r$ satisfying the property that for all $1 \leq k \leq r-1$, either $a_k = (i_k, i_{k+1}) \in A$ or $a_k = (i_{k+1}, i_k) \in A$

Directed Walk: A directed walk is an 'oriented' version of a walk in the sense that for any two consecutive nodes i_k and i_{k+1} on the walk, $(i_k, i_{k+1}) \in A$.



1-2-3-4 is a walk, but not a directed walk.

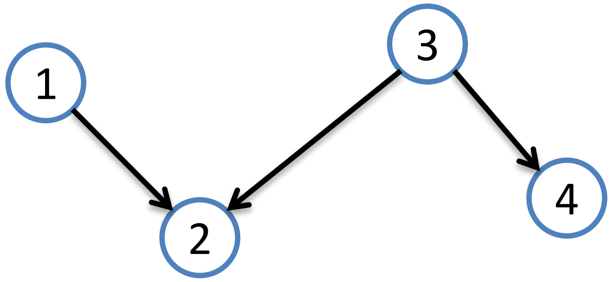


1-2-3-4-5 is a directed walk.

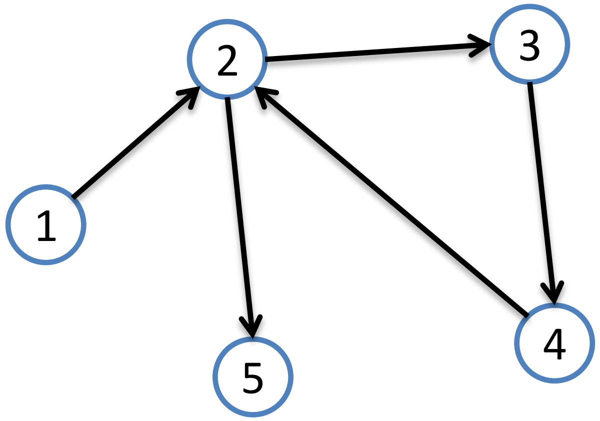
Network

Path: A path is a walk without any repetition of nodes.

Directed path: A **directed path** is a directed walk without any repetition of nodes. We can store a path (or directed path) easily within a computer by defining a **predecessor** index pred(j) for every node j in the path.



1-2-3-4 is a path, but not a directed path.



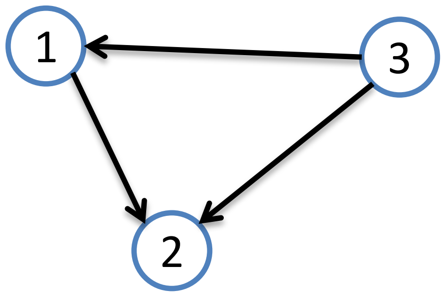
1-2-3-4-5 is not a path. (node 2 is repeatedly visited)

Network

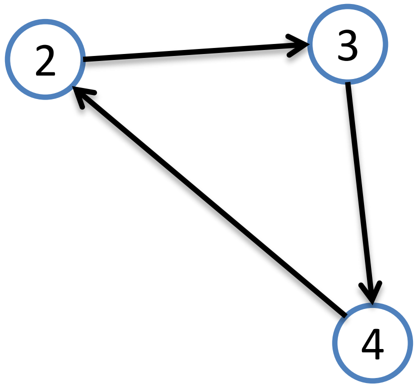
Cycle: A **cycle** is a path $i_1-, \ldots -i_r$ together with the arc (i_1, i_r) or (i_r, i_1).

Directed cycle: A **directed cycle** is a directed path $i_1-, \ldots -i_r$ together with the arc (i_r, i_1).

Acyclic Graph: A graph is a **acyclic** if it contains no directed cycle.



1-2-3 is a cycle, but not a directed cycle

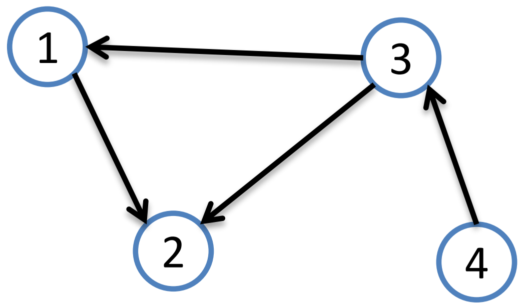


2-3-4 is a directed cycle.

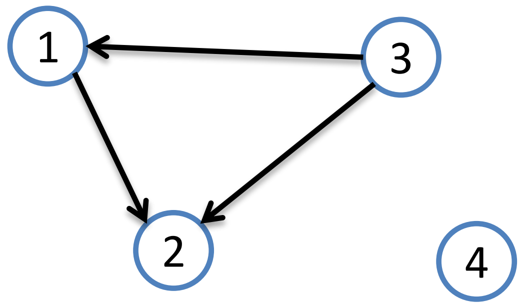
Network

Connectivity: Two nodes i and j are connected if the graph contains at least one path from node i to node j. A graph is connected if every pair of nodes is connected; otherwise the graph is disconnected.

Strong connectivity: A connected graph is strongly connected if it contains at least one directed path from every node to every other node.



Connected graph

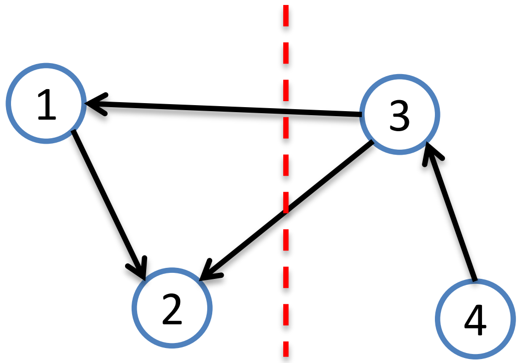


Disconnected graph

Network

Cut: A **cut** is a partition of the node set N into two parts, S and $\bar{S} = N - S$. Each cut defines a set of arcs consisting of those arcs that have one endpoint in S and another endpoint in $\bar{S}$.

s-t Cut: An **s-t cut** is defined with respect to two distinguished nodes s and t, and is a cut $[S, \bar{S}]$ satisfying the property that $s \in S$ and $t \in \bar{S}$.



If s=1, t=4, this is a s-t cut
If s=3, t=4, this is not a s-t cut

Network

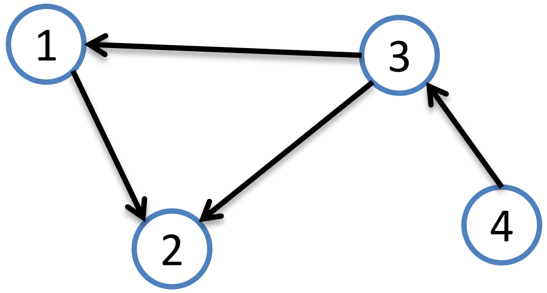
Tree: A **tree** is a connected graph that contains no cycle.

A tree on n nodes contains exactly   arcs.

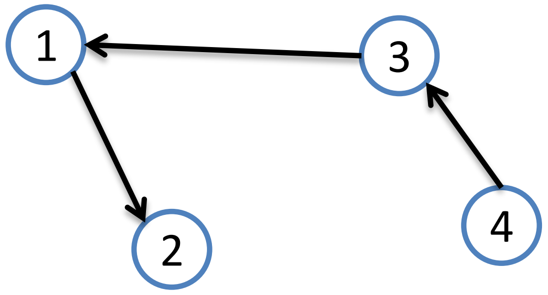
A tree has at least   leaf nodes (i.e., nodes with degree 1).

Every two nodes of a tree are connected by a unique path.

Subtree: A connected subgraph of a tree is a **subtree**.



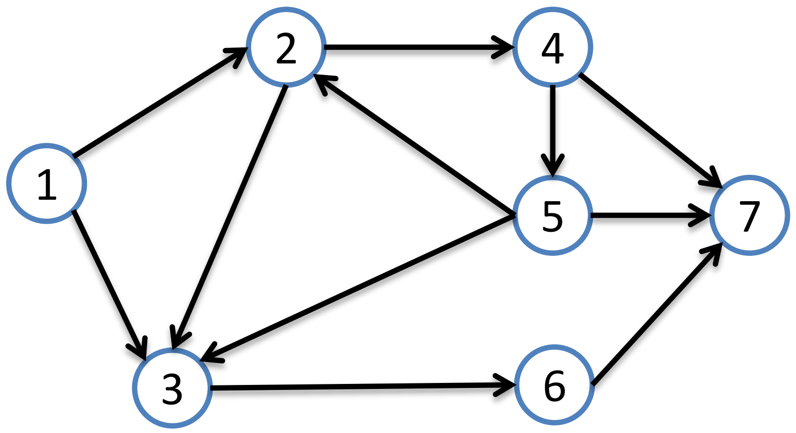
Not a tree



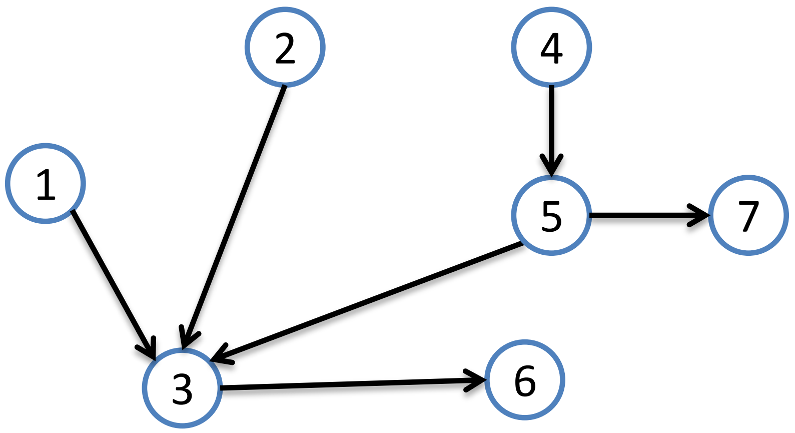
A tree

Network

Spanning tree: A tree T is a **spanning tree** of G if T is a spanning subgraph of G. Every spanning tree of a connected n-node graph G has (n-1) arcs. We refer to the arcs belonging to a **spanning tree** T as *tree arcs* and arcs not belonging to T as *nontree arcs*.



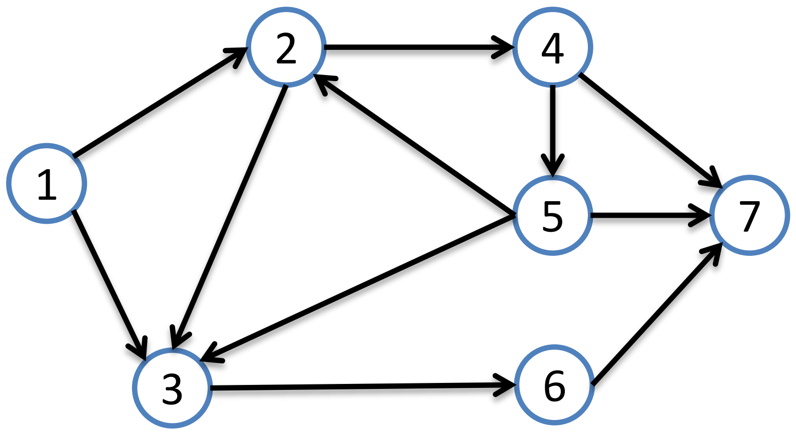
Graph G



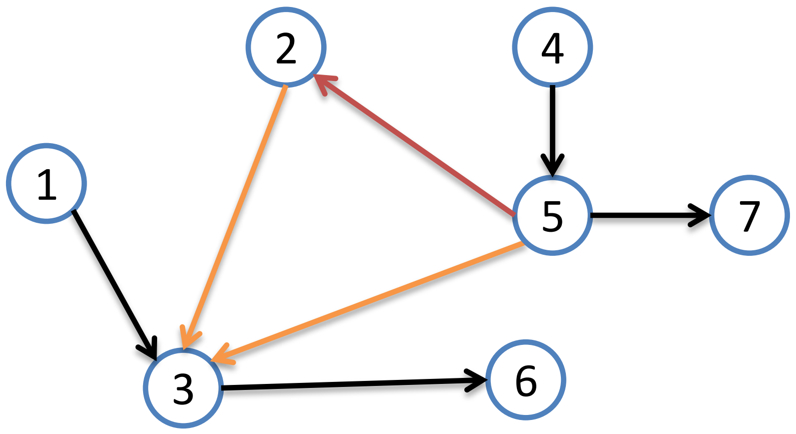
A spanning tree of G

Network

Fundamental cycles: Let T be a spanning tree of the graph G. The addition of any nontree arc to the spanning tree T creates exactly one cycle. We refer to any such cycle as a fundamental cycle of G with respect to the tree T. Since the network contains m-n+1 nontree arcs, it has m-n+1 fundamental cycles. If we delete any arc in a fundamental cycle, we again obtain a spanning tree.



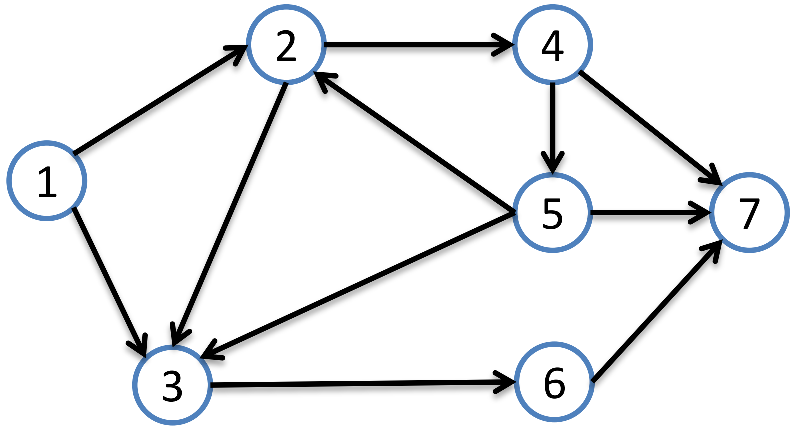
Graph G



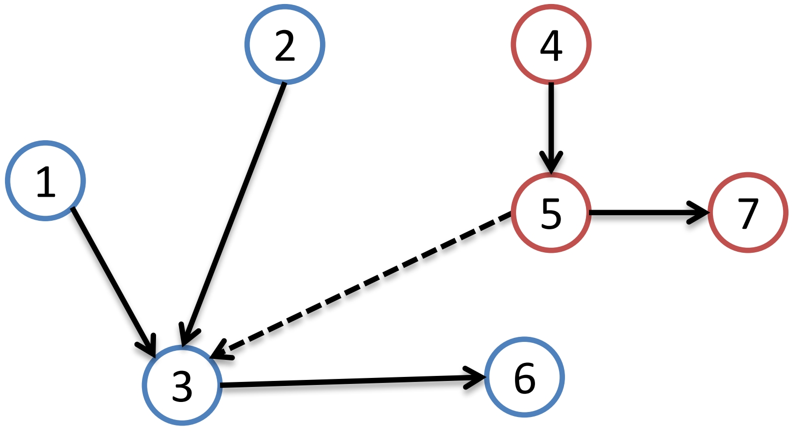
A fundamental cycle

Network

Fundamental cuts: Let T be a spanning tree of the graph G. The deletion of any tree arc of the spanning tree T produces a disconnected graph containing two subtrees T1 and T2. Arcs whose endpoints belong to the different subtrees constitute a cut. We refer to any such cut as a **fundamental cut** of G with respect to the tree T. Since a spanning tree contains n-1 arcs, the network has n-1 fundamental cuts with respect to any tree.



Graph G

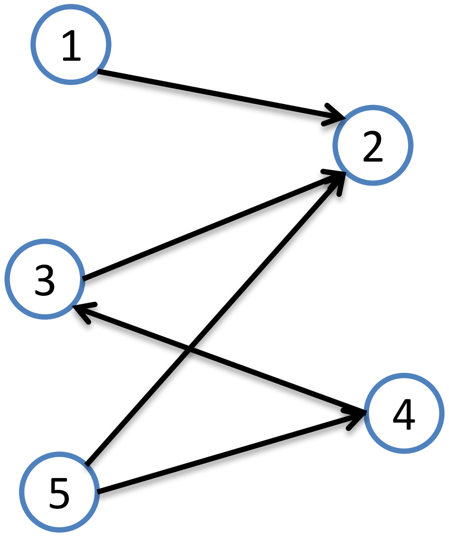


A fundamental cut

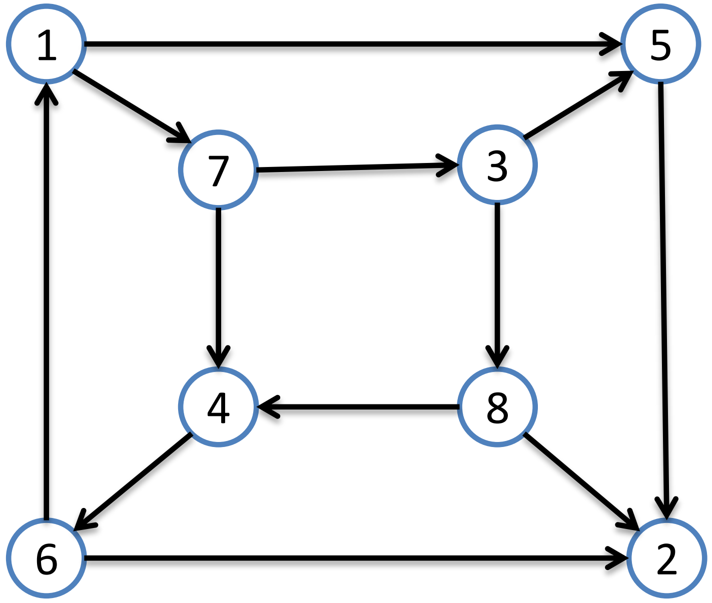
Network

Bipartite Graph: A graph G=(N,A) is a bipartite graph if we can partition its node set into two subsets N_1 and N_2 so that for each arc (*i*, *j*) in A either 1) $i \in N_1$ *and* $j \in N_2$ or 2) j $\in N_1$ *and* $i \in N_2$

A graph G is a bipartite graph if and only if every cycle in G contains an even number of arcs.



A bipartite graph



A bipartite graph?

- **Definitions for undirected networks.** The definitions for directed networks easily translate into those for undirected networks. An undirected arc (i,j) has two end points, i and j, but **its tail and head nodes are undefined**. If the network contains the arc (i,j), node i is adjacent to node j, and node j is adjacent to node i. The arc adjacency list (as well as the node adjacency list) is defined similarly expect that **arc (i,j) appears in A(i) as well as A(j). The degree of a node is the number of nodes adjacent to node i**. Each of the graph theoretic concepts we have defined so far –walks, paths, cycles, cuts and trees– has essentially the same definition for undirected networks except that **we do not distinguish between a path and a directed path, a cycle and directed cycle and so on.**

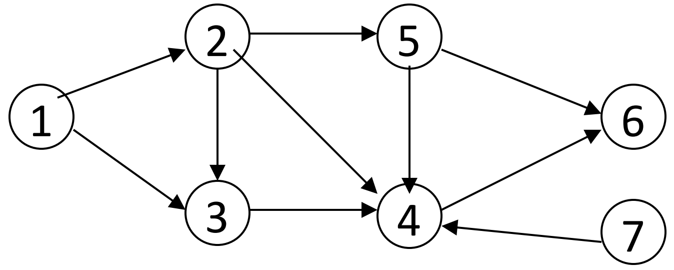
2. 网络算法基础

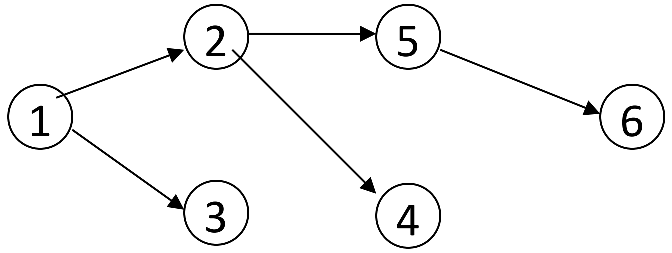
a） 网络搜索
b） 拓扑排序

Network searching

- Given a network *G*=(*N*, *A*) and a source node *s* in *N*, we want to find all nodes that are reachable from *s*
  - A node *t* is reachable from *s* if there is a directed path from node *s* to *t*.
- Basic idea
  - We can start to check each arc starting from *s* to find all nodes that are directly reachable from *s*
    - We call these nodes *marked*
  - Similarly, we can check all arcs that start from each marked node in order to find more nodes that are indirectly reachable from *s*
    - We have more nodes *marked* (meaning reachable from *s*)
  - We do this until no more nodes can be *marked*

Breadth-first Search (source node *s*=1)





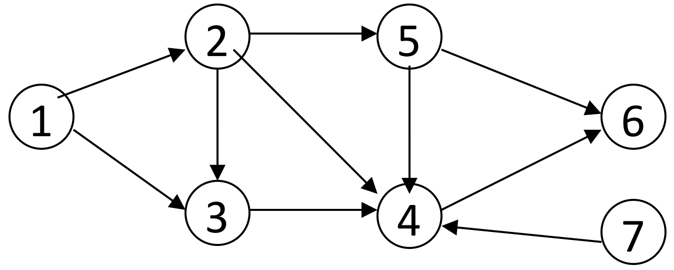
- Iteration 1: from node 1, nodes 2 and 3 are marked
- Iteration 2: from node 2, nodes 5 and 4 are marked
  - Note: node 3 is also reachable from node 2, but node 3 has already been marked before, so we do not market it again
- Iteration 3: from node 3, no new nodes are marked
- Iteration 4: from node 5, node 6 is marked
- Iteration 5: from node 4, no new nodes are marked
- Iteration 6: from node 6, no new nodes are marked
- Now, all marked nodes are checked. Algorithm stops.
  - Node 7 is not reachable from node 1

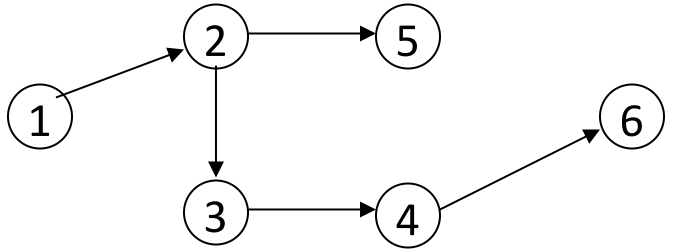
Breadth-first Search: Complexity

- In the algorithm, at each marked node, we check all arcs that start from the node
  - This is called breadth-first search
- The algorithm takes at most *n* iterations
  - Each iteration visits one node
- A rough analysis
  - In each iteration, we check all arcs starting the node, which takes at most $O(n)$ time
  - So the time complexity is in $O(n^2)$
- A more careful analysis
  - Let the number of arcs starting from node *j* be $|A(j)|$
    - Recall that A(j) is called adjacency list of node j, i.e., the set of arcs starting from node j
  - Then each iteration has $|A(j)|$ operations
  - The total operations over all iterations is

    $|A(1)|+|A(2)|+\ldots+|A(n)| \leq m$, total number of arcs
  - Time complexity is $O(m)$

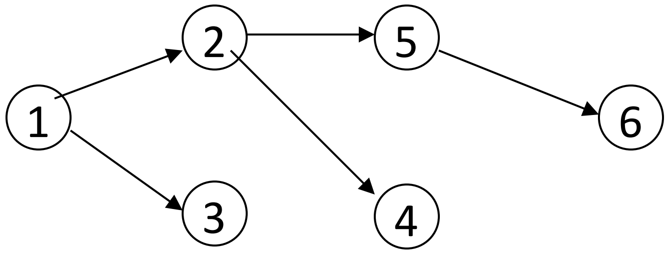
Depth-first Search



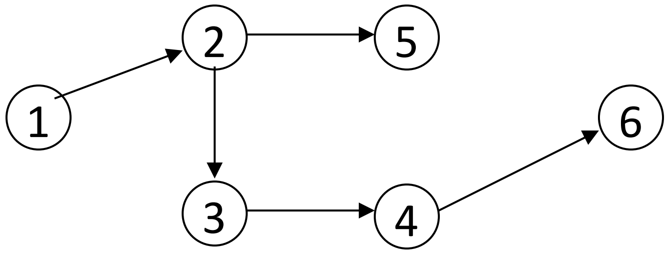


- At each marked node, we only check one arc
  - We always want to go to further nodes if possible: depth-first search
- Iteration 1: node 2 is reachable from node 1
- Iteration 2: node 3 is reachable from node 2
- Iteration 3: node 4 is reachable from node 3
- Iteration 4: node 6 is reachable from node 4
- Iteration 5: back from node 6 to node 4, then back to node 3, node 2
- Finally, node 5 is reachable from node 2
- Time complexity for the depth-first search algorithm is still $O(m)$

Search Trees



Breadth-first search tree



Depth-first search tree

Reverse searching

Adjust the previous searching algorithm with three slight changes:

1) Start from node t;
2) When examining a node, scan its incoming arcs instead of its outgoing arcs;
3) Make node *i* if for an arc (i,j), *i* is unmarked and *j* is marked.

Detecting Strong Connectivity

- How to detect whether a network is strongly connected?
  - A network is **strongly connected** if there exists a directed path from every node to every other node
- Algorithm 1
  - starting from every node, do a search of the network
  - Time complexity: $O(nm)$
- Algorithm 2
  - Step 1: Randomly select a node *s* as the source, and do a search of the network – *to see if every node is reachable from s*
  - Step 2: Change the direction of all arcs in the network, and do a search of the network from node *s again – to see if s is reachable from every node*
  - Time complexity: $O(m)$
  - Why is this algorithm correct?

3. 网络算法基础

a） 网络搜索

b） 拓扑排序

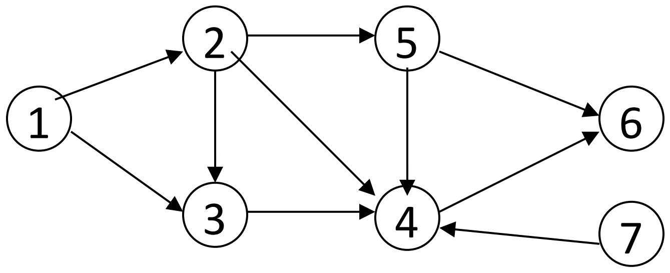
Topological Ordering

- For a network G=(N,A), a topological ordering of nodes satisfies: for every arc $(i, j) \in A$, order(i)<order(j).

Can we find a topological ordering for any network?

A network is acyclic if and only if it possesses a topological ordering of its nodes.

Topological Ordering



Time complexity: O(m)

- Step 1. Select any node with zero indegree, label it 1, and delete it and all the arcs emanating from it.
- Step 2. in the remaining subgraph, select any node of zero indegree, give it a label of 2, and then delete it and all arcs emanating from it.
- Repeat the process until no node has a zero indegree. At this point, if the remaining subgraph contains some nodes and arcs, then the network contains a directed cycle. Otherwise the network is acyclic and the label we have assigned is a topological ordering.

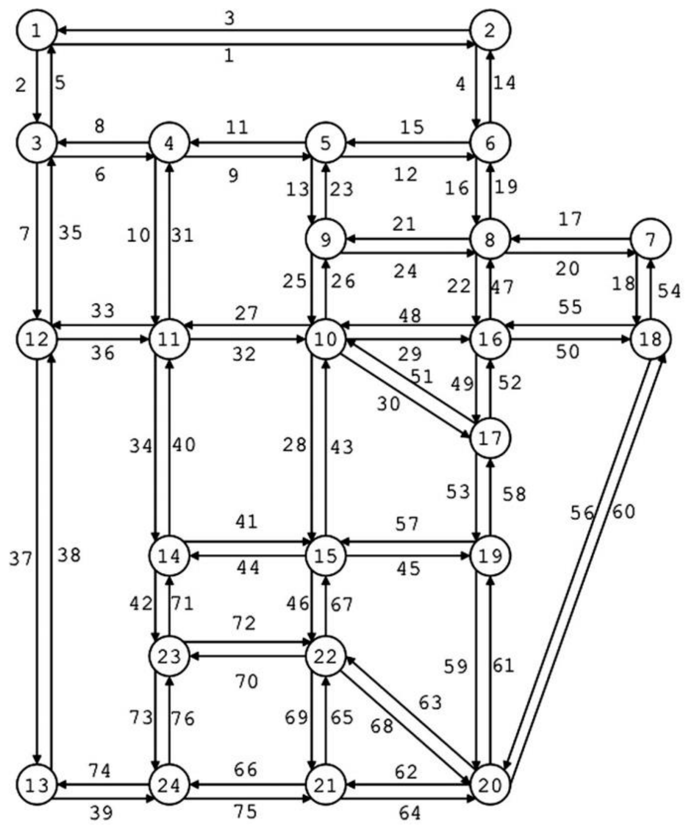
3. 网络的存储

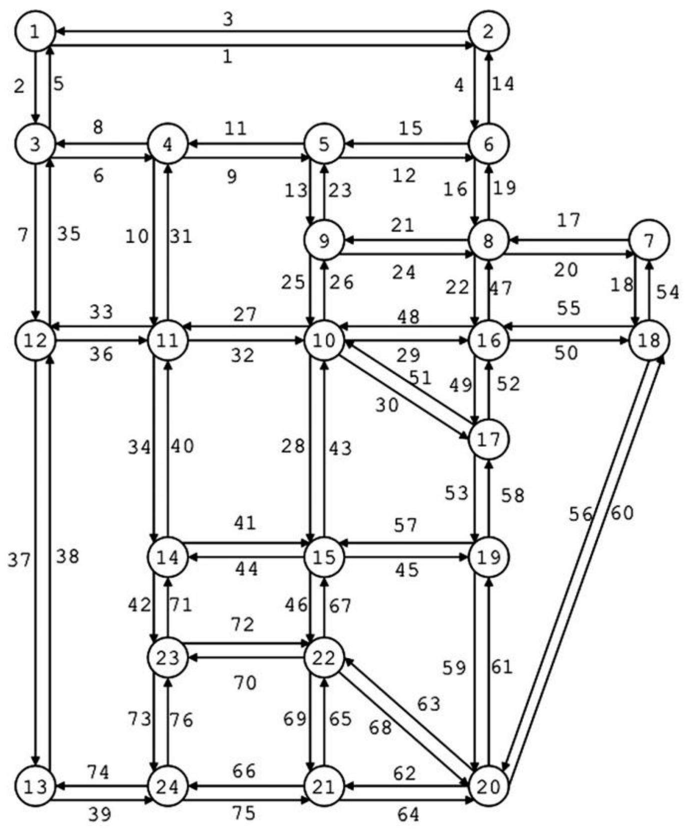
网络的存储

一个网络问题的算法好坏不仅仅取决于算法本身，还取决于在计算机内如何存储网络信息。一个聪明的存储方法和数据结构往往能够减少算法的运行时间。

为了表示一个网络结构，我们需要存储哪些信息？

1. 网络的拓扑结构，即节点和边的关联关系
2. 与点和边相关的数据信息，如成本，容量，需求等





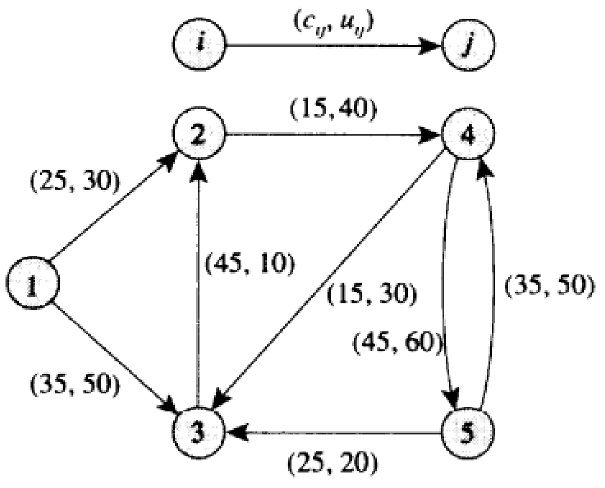
init_node	term_node	capacity	length	free_flow_time	b	power	speed	toll	link_type
1	2	25900.20064	6	6	0.15	4	0	0	1
1	3	23403.47319	4	4	0.15	4	0	0	1
2	1	25900.20064	6	6	0.15	4	0	0	1
2	6	4958.180928	5	5	0.15	4	0	0	1
3	1	23403.47319	4	4	0.15	4	0	0	1
3	4	17110.52372	4	4	0.15	4	0	0	1
3	12	23403.47319	4	4	0.15	4	0	0	1
4	3	17110.52372	4	4	0.15	4	0	0	1
4	5	17782.7941	2	2	0.15	4	0	0	1
4	11	4908.82673	6	6	0.15	4	0	0	1
5	4	17782.7941	2	2	0.15	4	0	0	1
5	6	4947.995469	4	4	0.15	4	0	0	1
5	9	10000 5	5	0.15	4	0	0	1	;
6	2	4958.180928	5	5	0.15	4	0	0	1
6	5	4947.995469	4	4	0.15	4	0	0	1
6	8	4898.587646	2	2	0.15	4	0	0	1
7	8	7841.81131	3	3	0.15	4	0	0	1
7	18	23403.47319	2	2	0.15	4	0	0	1
8	6	4898.587646	2	2	0.15	4	0	0	1
8	7	7841.81131	3	3	0.15	4	0	0	1
8	9	5050.193156	10	10	0.15	4	0	0	1
8	16	5045.822583	5	5	0.15	4	0	0	1
9	5	10000 5	5	0.15	4	0	0	1	;
9	8	5050.193156	10	10	0.15	4	0	0	1
9	10	13915.78842	3	3	0.15	4	0	0	1
10	9	13915.78842	3	3	0.15	4	0	0	1
10	11	10000 5	5	0.15	4	0	0	1	;
10	15	13512.00155	6	6	0.15	4	0	0	1
10	16	4854.917717	4	4	0.15	4	0	0	1
10	17	4993.510694	8	8	0.15	4	0	0	1
11	4	4908.82673	6	6	0.15	4	0	0	1
11	10	10000 5	5	0.15	4	0	0	1	;
11	12	4908.82673	6	6	0.15	4	0	0	1
11	14	4876.508287	4	4	0.15	4	0	0	1
12	3	23403.47319	4	4	0.15	4	0	0	1
12	11	4908.82673	6	6	0.15	4	0	0	1

主要的网络存储方式

1. 关联矩阵表示法
2. 邻接矩阵表示法
3. 邻接表表示法
4. 前向星形表示法
5. 后向星形表示法

关联矩阵表示法

在关联矩阵中，每行对应于图的一个节点，每列对应于图的一条弧。如果一个节点是一条弧的起点，则关联矩阵中对应的元素为1；如果一个节点是一条弧的终点，则关联矩阵中对应的元素为 -1；如果一个节点与一条弧不关联，则关联矩阵中对应的元素为0。



$$
\begin{array}{c|cccccccc}
 & (1,2) & (1,3) & (2,4) & (3,2) & (4,3) & (4,5) & (5,3) & (5,4) \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
3 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & 0 \\
4 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & -1 \\
5 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1
\end{array}
$$

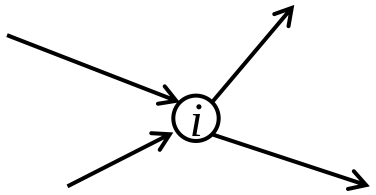
c_{ij}	25	35	15	45	15	45	25	35
u_{ij}	30	50	40	10	30	60	20	50

每条弧和点上的附加信息怎么存储？

关联矩阵表示法

优点：

1. 简单、直接
2. 它能够用于直接表示一些网络问题的约束，具有一些特殊的理论性质



最小费用流问题的一般表达式：

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

subject to

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

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

向量表达式：

$$\text{Minimize } cx$$

subject to

$$\mathcal{N}x = b,$$

$$l \le x \le u.$$

关联矩阵表示法

缺点：

每一列只有两个非零元素（-1，1），在m*n的矩阵中，只有2m个非零元素，浪费大量存储空间。

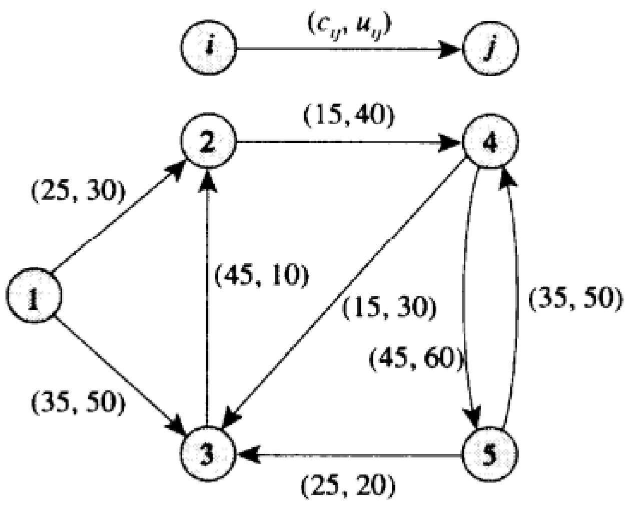
m：边的个数
n：点的个数

$$\begin{array}{c|cccccccc}
 & (1,2) & (1,3) & (2,4) & (3,2) & (4,3) & (4,5) & (5,3) & (5,4) \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
3 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & 0 \\
4 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & -1 \\
5 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1
\end{array}$$

邻接矩阵表示法

在点-点邻接矩阵中，每行对应于图的一个节点，每列也对应于图的一个节点。如果在图中存在一条从i点至j点的弧，则邻接矩阵中对应的元素为1；否则为0。

如要存储弧上的其他信息，则需要重新构造一个n*n的矩阵。



$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
5 & 0 & 0 & 1 & 1 & 0
\end{array}
$$

邻接矩阵表示法

缺点：

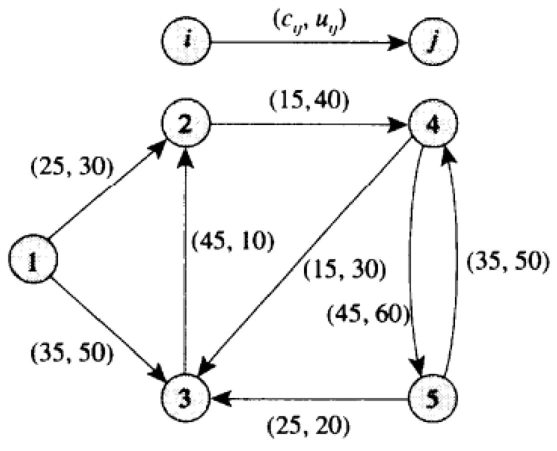
1. 邻接矩阵只有对于非常紧凑的网络才具有较好的存储空间使用效率。

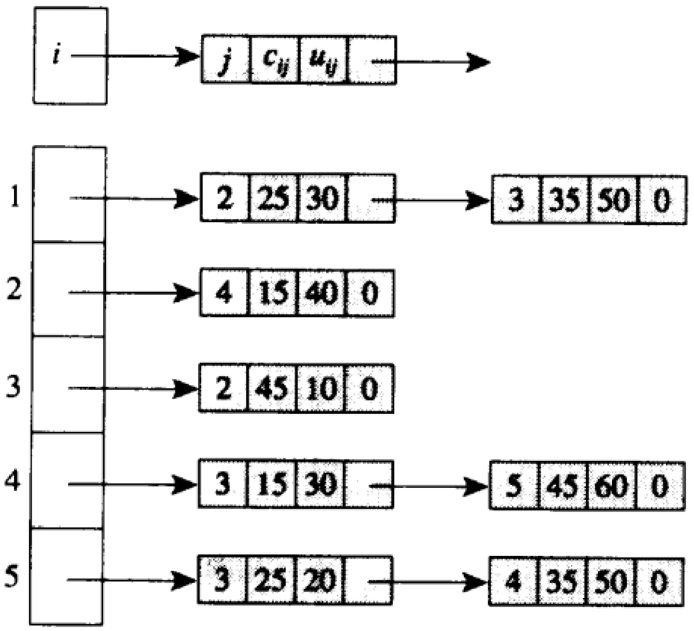
优点：

1. 简单、直接
2. 对于点和点邻接关系的刻画使得很多网络算法的执行非常简便。

邻接表表示法

- 邻接表表示法就是对网络中的每个节点，用一个单向链表列出从该节点出发的所有弧，链表中每个单元对应于一条出弧。而整个网络的邻接表，就是网络中的所有节点的邻接表的集合。为了记录弧上的信息，链表中每个单元除列出弧的另一个端点外，还可以包含弧上的权等数据域。





邻接表表示法

优点：

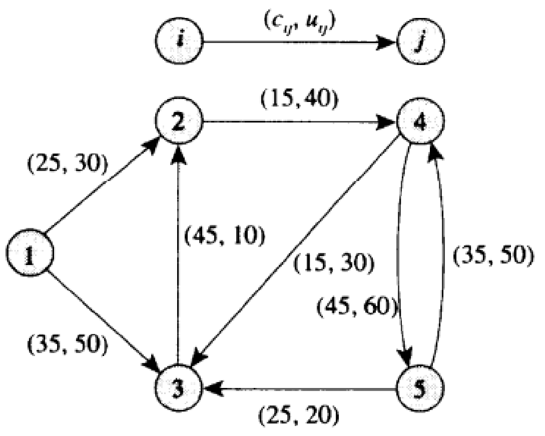
1. 节约存储空间；
2. 能够快速找到每个点的所有后继节点，使得很多网络算法的执行非常简便；
3. 当网络结构发生变化时能够很方便地调整。

缺点：

仅适用于有指针的编程语言。

前向星形表示法

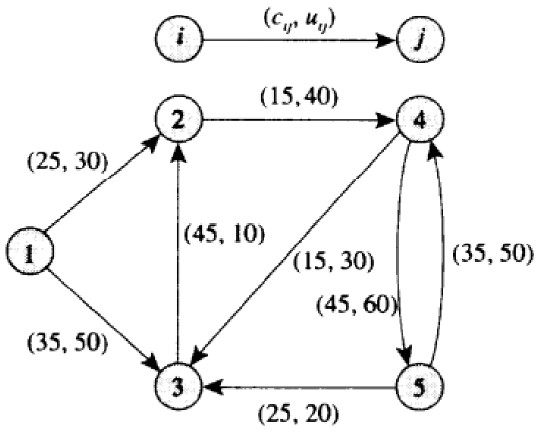
- 前向星形表示法（Forward star representation）的思想与邻接表表示法的思想有一定的相似之处。对每个节点，它也是记录从该节点出发的所有弧，但它不是采用单向链表而是采用一个单一的数组表示。



	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

前向星形表示法

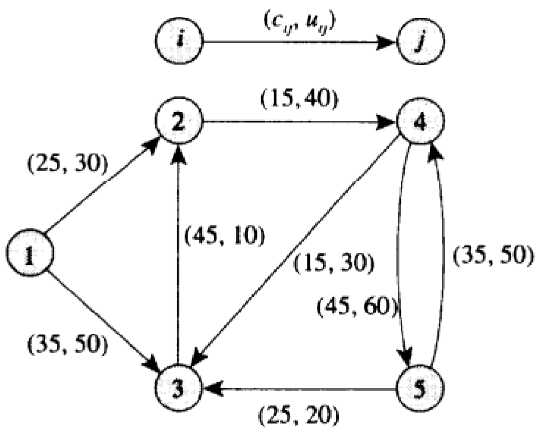
- 在该数组中首先存放从节点1出发的所有弧，然后接着存放从节点2出发的所有弧，依此类推，最后存放从节点 出发的所有孤。对每条弧，要依次存放其起点、终点、权的数值等有关信息。

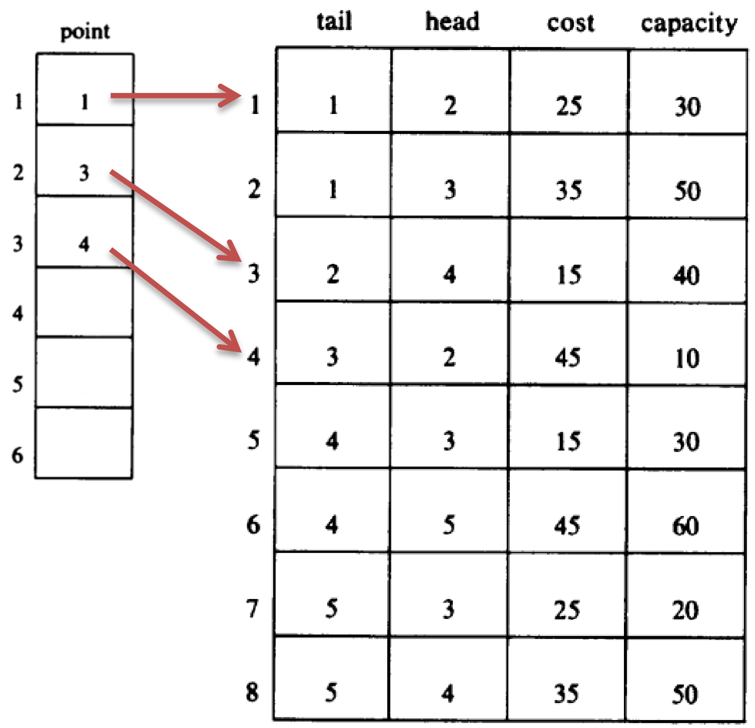


	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

前向星形表示法

- 为了能够快速检索从每个节点出发的所有弧，我们一般还用一个数组记录每个节点出发的弧的起始地址（即弧的编号）。



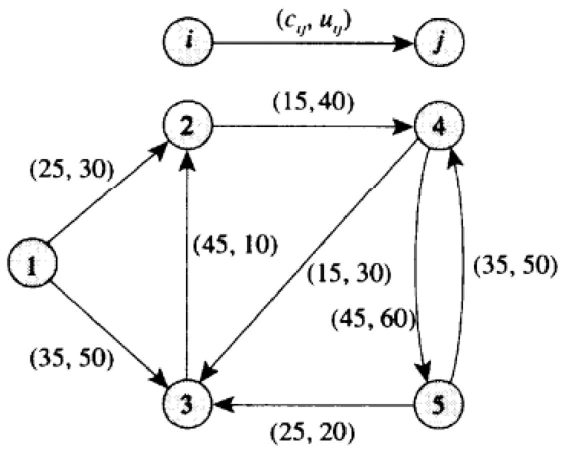


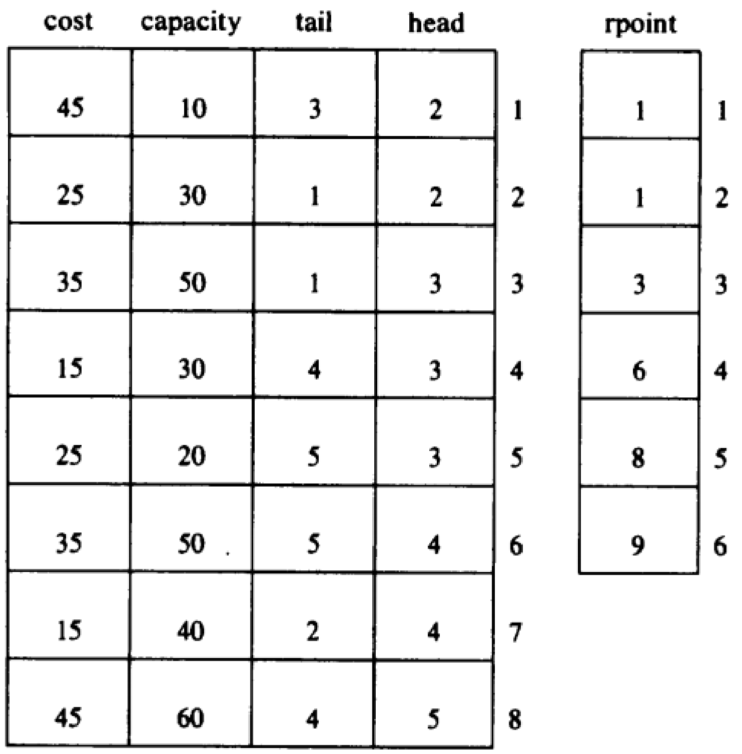
	point
1	1
2	3
3	4
4	
5	
6	

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

后向星形表示法

- 后向星形表示法（Reverse star representation）与前向星形表示法类似。但在后向星形表示法下，能很方便地找到到达任意节点的所有弧。





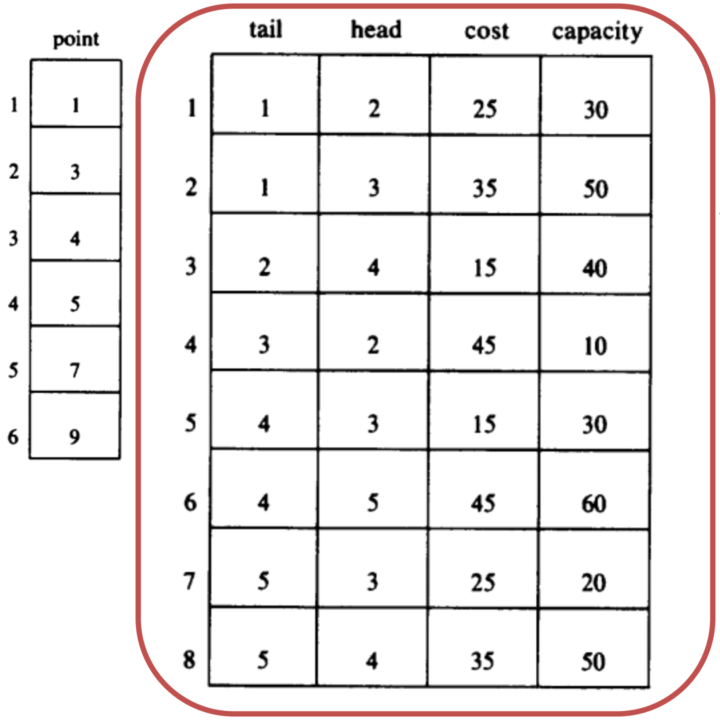
cost	capacity	tail	head	
45	10	3	2	1
25	30	1	2	2
35	50	1	3	3
15	30	4	3	4
25	20	5	3	5
35	50	5	4	6
15	40	2	4	7
45	60	4	5	8

rpoint	
1	1
1	2
3	3
6	4
8	5
9	6

双向星形表示法

- 双向星形表示法（Forward and reverse star representation）是前向和后向星形表示法的结合。通过双向星形表示法能够高效地利用存储的弧信息来实现快速地前向和后向的搜索。

前向星形表示法

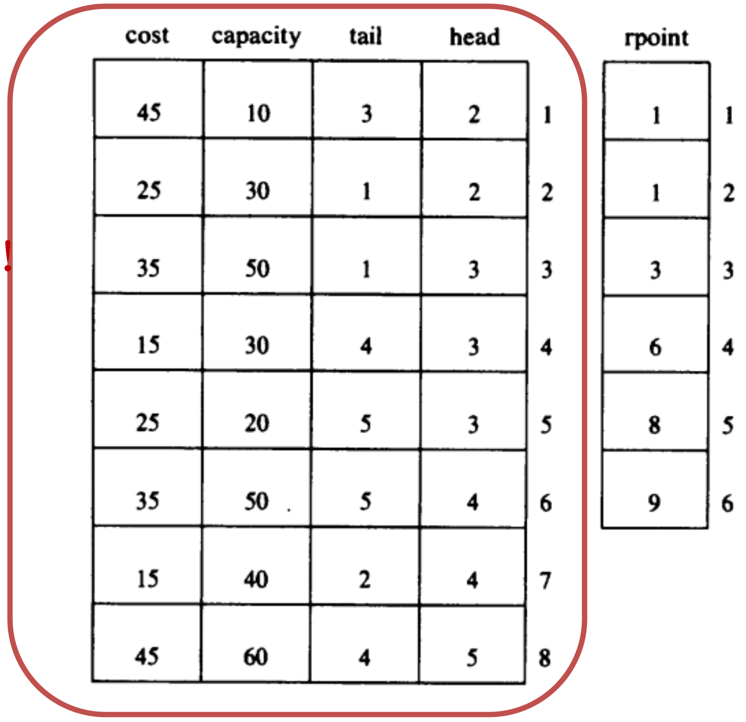


	point
1	1
2	3
3	4
4	5
5	7
6	9

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

重复信息存储！

后向星形表示法



cost	capacity	tail	head	
45	10	3	2	1
25	30	1	2	2
35	50	1	3	3
15	30	4	3	4
25	20	5	3	5
35	50	5	4	6
15	40	2	4	7
45	60	4	5	8

rpoint	
1	1
1	2
3	3
6	4
8	5
9	6

双向星形表示法

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

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

cost	capacity	tail	head	
45	10	3	2	1
25	30	1	2	2
35	50	1	3	3
15	30	4	3	4
25	20	5	3	5
35	50	5	4	6
15	40	2	4	7
45	60	4	5	8

rpoint	
1	1
1	2
3	3
6	4
8	5
9	6

在后向列表中的第一条弧对应前向表中的第4条

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

	tail	head	cost	capacity
1	1	2	25	30
2	1	3	35	50
3	2	4	15	40
4	3	2	45	10
5	4	3	15	30
6	4	5	45	60
7	5	3	25	20
8	5	4	35	50

trace	
4	1
1	2
2	3
5	4
7	5
8	6
3	7
6	8

rpoint	
1	1
1	2
3	3
6	4
8	5
9	6

星形表示法

优点：

1. 节约存储空间
2. 能够快速实现前向和后向搜索

缺点：

当网络结构发生变化时，星形表示法的调整非常耗时(O(m))。

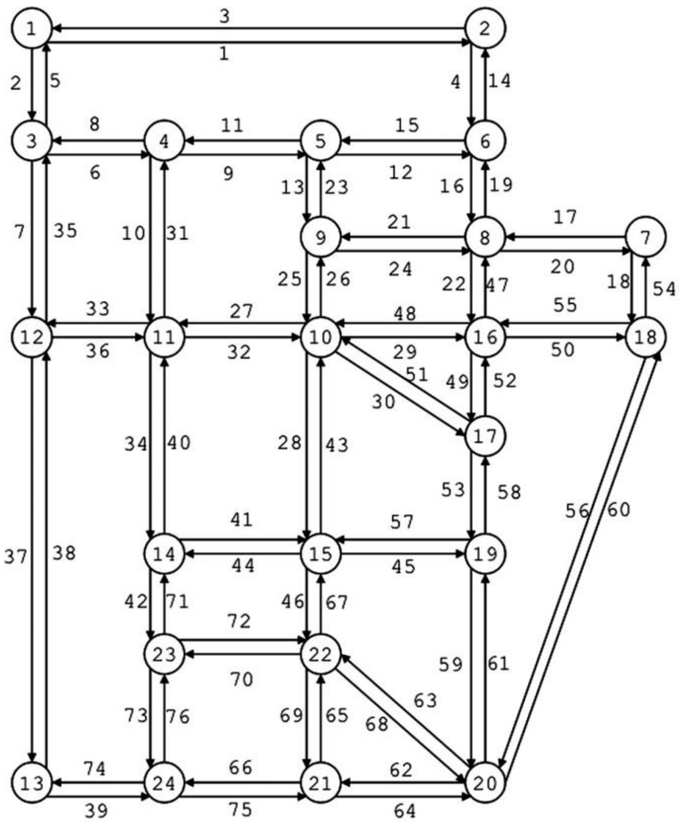
几种网络存储方式的比较

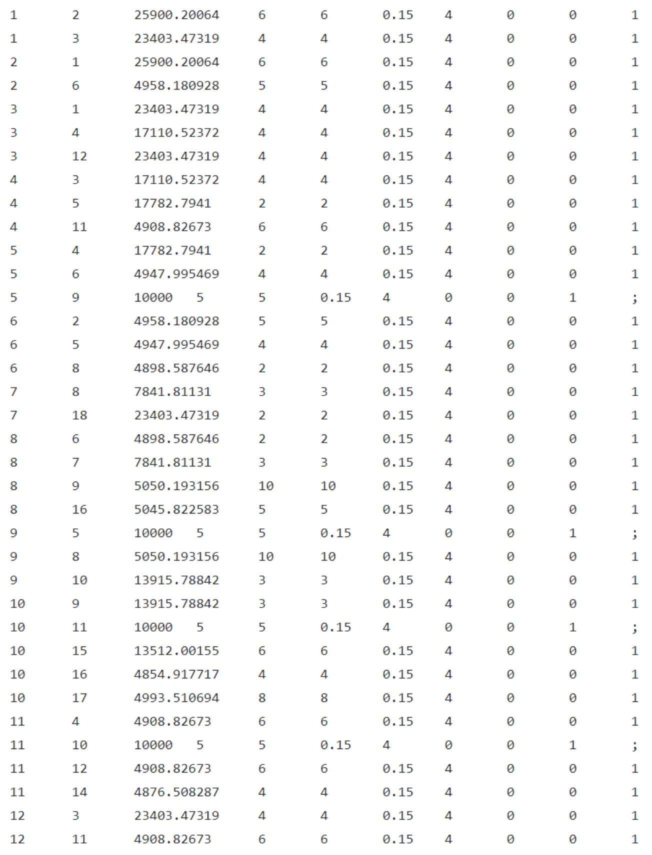
存储方式	存储空间占用	特征
关联矩阵表示法	mn	空间使用效率低 可操作性低 但具有重要理论意义
邻接矩阵表示法	kn^2, k 为常数	适用于紧凑网络 易于网络算法的执行
邻接表表示法	$k_1n + k_2m$, k_1和k_2为常数	空间使用效率高 易于网络算法的执行 在网络变化时易于调整 只适用于有指针的编程语言
星形表示法	$k_3n + k_4m$, k_3和k_4为常数	空间使用效率高 易于网络算法的执行

Homework

- 编程实现下面Sioux Falls network的双向星形存储，编程语言不限，初始网络信息：

https://github.com/bstabler/TransportationNetworks/blob/master/SiouxFalls/SiouxFalls_net.tntp





```
1     2     25900.20064   6     6     0.15  4     0     0     1
1     3     23403.47319   4     4     0.15  4     0     0     1
2     1     25900.20064   6     6     0.15  4     0     0     1
2     6     4958.180928   5     5     0.15  4     0     0     1
3     1     23403.47319   4     4     0.15  4     0     0     1
3     4     17110.52372   4     4     0.15  4     0     0     1
3     12    23403.47319   4     4     0.15  4     0     0     1
4     3     17110.52372   4     4     0.15  4     0     0     1
4     5     17782.7941    2     2     0.15  4     0     0     1
4     11    4908.82673    6     6     0.15  4     0     0     1
5     4     17782.7941    2     2     0.15  4     0     0     1
5     6     4947.995469   4     4     0.15  4     0     0     1
5     9     10000   5     5     0.15  4     0     0     1     ;
6     2     4958.180928   5     5     0.15  4     0     0     1
6     5     4947.995469   4     4     0.15  4     0     0     1
6     8     4898.587646   2     2     0.15  4     0     0     1
7     8     7841.81131    3     3     0.15  4     0     0     1
7     18    23403.47319   2     2     0.15  4     0     0     1
8     6     4898.587646   2     2     0.15  4     0     0     1
8     7     7841.81131    3     3     0.15  4     0     0     1
8     9     5050.193156   10    10    0.15  4     0     0     1
8     16    5045.822583   5     5     0.15  4     0     0     1
9     5     10000   5     5     0.15  4     0     0     1     ;
9     8     5050.193156   10    10    0.15  4     0     0     1
9     10    13915.78842   3     3     0.15  4     0     0     1
10    9     13915.78842   3     3     0.15  4     0     0     1
10    11    10000   5     5     0.15  4     0     0     1     ;
10    15    13512.00155   6     6     0.15  4     0     0     1
10    16    4854.917717   4     4     0.15  4     0     0     1
10    17    4993.510694   8     8     0.15  4     0     0     1
11    4     4908.82673    6     6     0.15  4     0     0     1
11    10    10000   5     5     0.15  4     0     0     1     ;
11    12    4908.82673    6     6     0.15  4     0     0     1
11    14    4876.508287   4     4     0.15  4     0     0     1
12    3     23403.47319   4     4     0.15  4     0     0     1
12    11    4908.82673    6     6     0.15  4     0     0     1
```

4. 算法复杂度

How to Evaluate an Algorithm?

- What is an algorithm?
  - A set of instructions of basic operations to solve a problem
  - Basic operations: assignment, arithmetic (+,-,*,/), logic (to compare)
  - For example, simplex method is an algorithm for solving LP

- What does a good algorithm mean?
  - Time efficiency
  - Space (or memory) efficiency

Defining time efficiency

- First Attempt:
  - An algorithm is efficient if, when implemented, it runs quickly on real input instances.
  - Is it a good definition?
    - Platform-dependent, instance-dependent
- We want to analyze the running time as a function of problem size, or more accurately, problem input size.
  - In an LP max{**cx** | **Ax**=**b**}, the problem size is measured by
    - The dimension of **A**, **b** and **c**
  - In network flow problems, a problem size is naturally measured by
    - Number of nodes: denoted by n
    - Number of arcs: denoted by m

Example 1

- Given an array of numbers $A=\{a_1,a_2,\ldots,a_n\}$ and another number x, determine whether x is in A.
- Algorithm:
  - Step 1: Let $k=1$
  - Step 2: If $a_k=x$, report that x is in A and stop.
  - Step 3: Let $k = k+1$. If $k \leq n$, goto step 2.
  - Step 4: Report that x is not in A and stop.
- In the worst case, the algorithm has to take n iterations
  - In each iteration, we need two comparisons, one addition, one assignment
- But the algorithm may stop earlier, thus taking less than n iterations.

Time Complexity Measurements

- Average-case analysis
  - Obtain bound on running time of algorithm on random input as a function of input size.
  - To have an accurate result, we need to know the distribution of the data in the problem input
  - Hard (or impossible) to accurately model real instances by random distributions.
  - Algorithm tuned for a certain distribution may perform poorly on other inputs.

Time Complexity Measurements

- Empirical analysis
  - We implement the algorithm on a computer, use the computer to solve many different instances of the problem
  - Then we know what is the actual average time it takes
- Worst-case analysis
  - Obtain bound on largest possible running time of algorithm on input size.
  - This is the case we usually take as the default measurement

Worst-case Polynomial-time

- Formal definition of efficiency
  - An algorithm is efficient if its running time is polynomial of its input size in the worst case.

Why Polynomial?

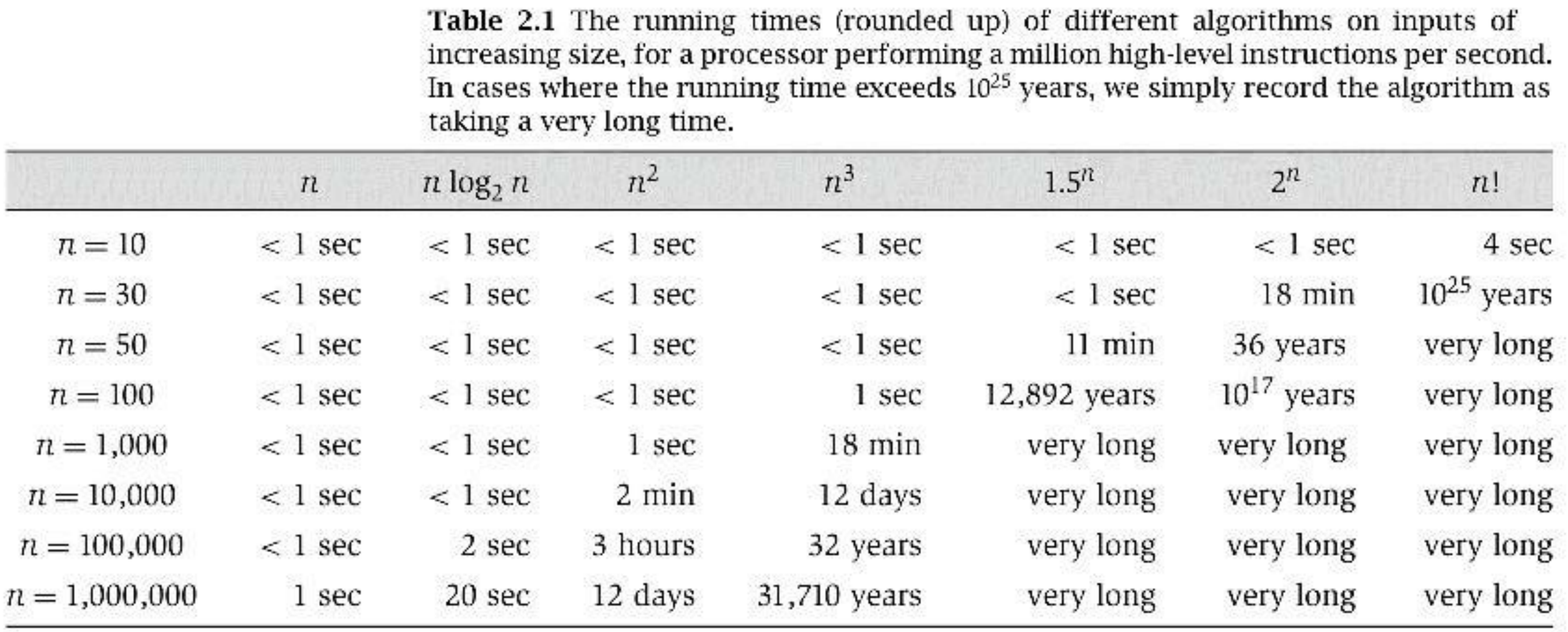


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Exceptions

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

Asymptotic Order

- Asymptotic Upper Bounds: O()
- Asymptotic Lower Bounds: Ω()
- Asymptotic Tight Bounds: θ()

The Big *O* Notation

- T(n): the worst-case running time of a certain algorithm on an input size n.
- T(n)=$O(f(n))$ if for some constants c and n_0, the time taken by the algorithm (on all problem instances) is at most $cf(n)$ for all $n \geq n_0$
  - $T(n)=100n+10$ is in $O(n)$
    - When c=101, n_0=10, we have $g(n)=100n+10 \leq 101n$ for $n \geq n_0$
  - $T(n)=n^2+2n$ is in $O(n^2)$
    - When c=2 and n_0=2, we have $g(n)=n^2+2n \leq 2n^2$ for $n \geq n_0$

- The big *O* notation is about an **upper bound** of the time complexity
- When using the Big-*O* notation, we always ignore some constant factors as well as terms of lower orders
  - $O(n)$ may actually be $2n$, even $100n$
  - $O(n^3)$ may actually be $0.001n^3+100n^2$

Big Ω Notation

- An algorithm is said to run in Ω(*f*(*n*)) time if for some constants *c* and n_0 and all $n \geq n_0$, the algorithm takes at least *cf*(*n*) time on some problem instance.
  - This gives a **lower bound** on the worst case.

θ Notation

- An algorithm is said to be **θ** (*f*(*n*)) if the algorithm is both *O*(*f*(*n*)) and Ω(*f*(*n*)).
- Polynomials. $a_0 + a_1n + \ldots + a_dn^d$ is **Θ**(n^d)

Example 2

- Given a sorted array of numbers $A=\{a_1 \leq a_2 \leq \ldots \leq a_n\}$ and another number x, determine whether x is in A.
- We already have an algorithm in $O(n)$ time
- A better algorithm (known as binary search):
  - Step 1: Let $k=n/2$, $s=1$, $t=n$
  - Step 2: If $a_k=x$, report that x is in A and stop.
  - Step 3: If $s>t$, report that x is not in A and stop.
  - Step 4: If $a_k<x$, let $s=k+1$, $k = (s+t)/2$. Go to step 2.
  - Step 5: Let $t=k-1$, $k = (s+t)/2$. Go to step 2.
- In the worst case, the algorithm needs to take no more than $1+\log_2 n$ iterations
  - The worst-case time complexity is in $O(\log_2 n)$.

P,NP,NP-Complete, and NP-hard

NP-completeness

Initiative

- Suppose that your boss asks you to develop an algorithm for a complex design problem. Despite weeks of sincere efforts you do not succeed in developing an efficient algorithm for solving this problem. How should you report to your boss?

- ***A***. *I can’t find an efficient algorithm, I guess I am just too dumb.*
- ***B.*** *I can’t find an efficient algorithm because no such algorithm is possible.*
- ***C.*** *I can’t find an efficient algorithm, but neither can these famous people.*

Which one is the most smart?

NP-completeness

Initiative

- NP-complete problems （NP完备问题）are a broad class of "**computationally equivalent**" problems, which include thousands of problems and possesses the remarkable property that **each problem in this class can be transformed to every other problem in polynomial time**; as a consequence, each problem is "just as hard" as every other problem.

Cook, S. A. (1971, May). The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (pp. 151-158). ACM.

P,NP,NP-complete,NP-hard

Class P

A recognition problem P1 belongs to class P if some polynomial-time algorithm solves problem P1.

Class NP（non-deterministic polynomial）

A decision problem P1 belongs to class NP if for every yes instance I of P1, there is a short (i.e., polynomial length) verification that the instance I is a yes instance.

Every problem in the class P must also be in class NP.

P,NP,NP-complete,NP-hard

Class NP-complete

NP-complete problems are the "hardest" in NP: if any NP-complete problem is p-time solvable, then all problems in NP are p-time solvable.

Class NP-hard (at least as hard as any NP problem)

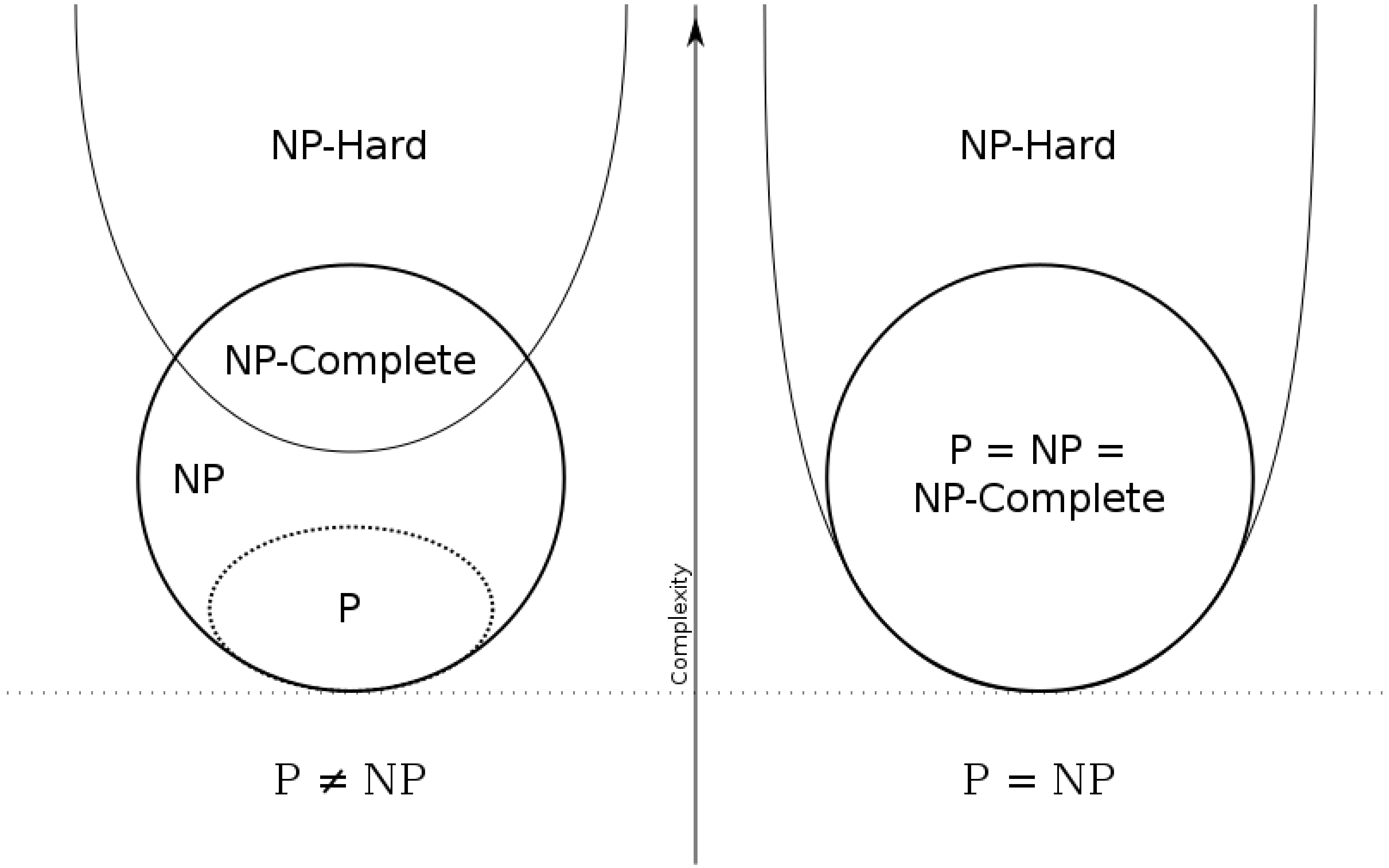
A decison problem P1 is said to be NP-hard if all other problem in the class NP polynomially reduce to P1.

Class NP-hard is broader than the class NP-complete because it includes the class NP as well as problems that are not in class NP.

NP-completeness

- A Problem P1 **polynomially reduces** to P2 （**P1多项式归约到P2**）if some polynomial-time algorithm that solves P1 uses the algorithm for solving P2 at unit cost. (所以，如果存在多项式算法能求解P2，那么一定存在多项式算法能够求解P1；但如果存在多项式算法能求解P1，未必存在多项式算法能够求解P2）

P,NP,NP-complete,NP-hard


NP-Hard
NP-Complete
NP
P
P ≠ NP
Complexity
NP-Hard
P = NP =
NP-Complete
P = NP