Network Flow Algorithms

Prerequisite

- Shortest Path

The Problem

Given: A direct connected graph with integer weighted arcs, along with a source node and a sink node.

Each arc weight corresponds to the ``capacity'' of that arc. A flow through the graph is constructed by assigning an integer amount of ``flow'' to send through each edge such that:

- The flow through each arc is no greater than the arc's capacity.
- For each node other than the source and sink, total flow in is the same as total flow out.

Maximize the total of the weights of the out-arcs of the source minus the weights of the in-arcs (or the total of the weights of the in-arcs of the sink minus the weights of the out-arcs).

Example

Given: The layout of a collection of water pipes, along with the capacity of each pipe. Water in these pipes must flow downhill, so within each pipe, water can only flow in one direction.

Calculate the amount of water that can flow from a given start (the water-purification plant) to a given end (your farm).

The Algorithm

The algorithm (greedily) builds the network flow by iteratively adding flow from the source to the sink.

Start with every arc having weight equal to the beginning weight (The arc weights will correspond to the amount of capacity still unused in that arc).

Given the current graph, find a path from the source to the sink across arcs that all have non-zero weight in the current graph. Calculate the maximum flow across this path, call it PathCap.

For each arc along the path, reduce the capacity of that arc by PathCap. In addition, add the reverse arc (the arc between the same two nodes, but in the opposite direction) with capacity equal to PathCap (if the reverse arc already exists, just increase its capacity).

Continue to add paths until none exist.

This is guaranteed to terminate because you add at least one unit of flow each time (since the weights are always integers), and the flow is strictly monotonically increasing. The use of an added reverse arc is equivalent to reducing the flow along that path.

If you are interested in a more detailed analysis of this algorithm, consult Sedgewick.

Here is pseudocode for the algorithm:

```
 if (source = sink)
   totalflow = Infinity
   DONE

 totalflow = 0

 while (True)
# find path with highest capacity from
 # source to sink
# uses a modified djikstra's algorithm
   for all nodes i
     prevnode(i) = nil
     flow(i) = 0
     visited(i) = False
   flow(source) = infinity

   while (True)
     maxflow = 0
     maxloc = nil
     # find the unvisited node with
       # the highest capacity to it
     for all nodes i
       if (flow(i) > maxflow AND
                               not visited(i))
```

```
                    maxflow = flow(i)
                    maxloc = i
            if (maxloc = nil)
                break inner while loop
            if (maxloc = sink)
                break inner while loop
           visited(maxloc) = true
            # update its neighbors
            for all neighbors i of maxloc
                if (flow(i) < min(maxflow,
                                      capacity(maxloc,i)))
                    prevnode(i) = maxloc
                    flow(i) = min(maxflow,
                                  capacity(maxloc,i))

        if (maxloc = nil)                  # no path
            break outer while loop

        pathcapacity = flow(sink)
        totalflow = totalflow + pathcapacity

  # add that flow to the network,
  # update capacity appropriately
        curnode = sink
              # for each arc, prevnode(curnode),
              # curnode on path:
        while (curnode != source)
            nextnode = prevnode(curnode)
            capacity(nextnode,curnode) =
                  capacity(nextnode,curnode) -
                                                  pathcapacity
            capacity(curnode,nextnode) =
                  capacity(curnode,nextnode) +
                                                  pathcapacity
            curnode = nextnode
```

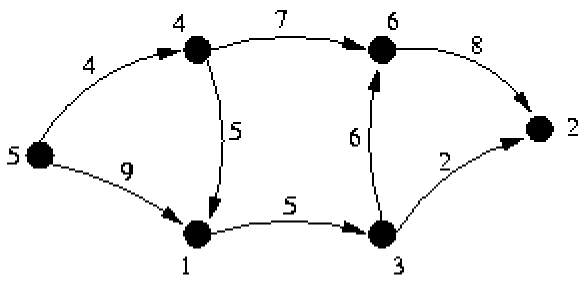
Running time of this formulation is O(F M), where F is the maximum flow and M is the number of arcs. You will generally perform much better, as the algorithm adds as much flow as possible every time.

To determine the flow across each arc, compare the starting capacity with the final capacity. If the final capacity is less, the difference is the amount of flow traversing that arc.

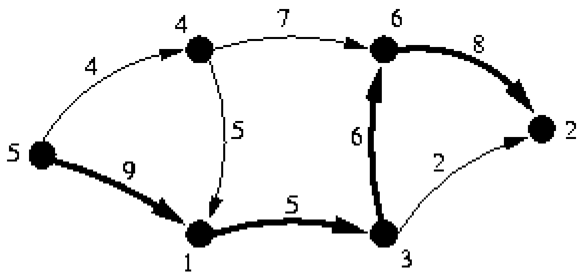
This algorithm may create `eddies,' where there is a loop which does not contribute to the flow itself.

Execution Example

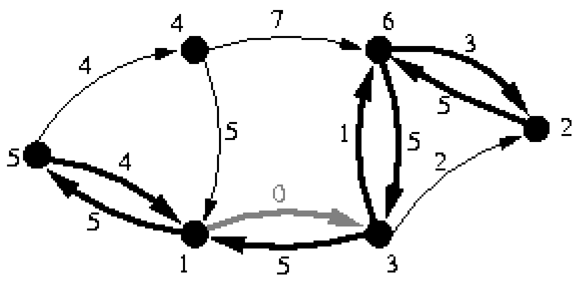
Consider the following network, where the source is node 5, and the sink is node 2.



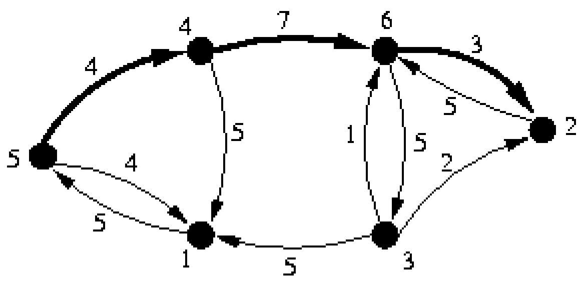
The path with the highest capacity is {5,1,3,6,2}.



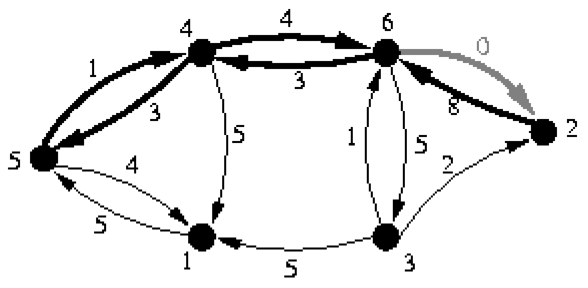
The bottleneck arc on this path is 1->3, which has a capacity of 5. Thus, reduce all arcs on the path by 5, and add 5 to the capacity of the reverse arcs (creating the arcs, if necessary). This gives the following graph:



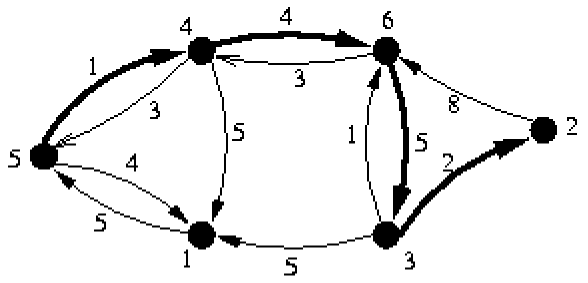
In the new graph, the path with highest capacity is {5,4,6,2}.



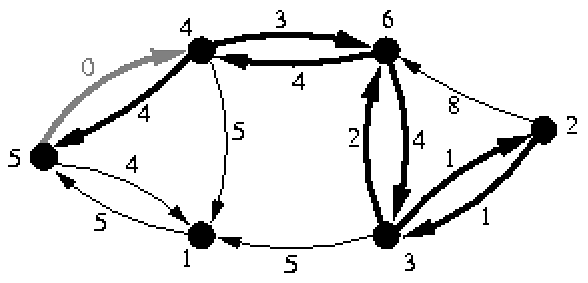
The capacity of this path is 3, so once again, reduce the forward arcs by 3, and increase the reverse arcs by 3.



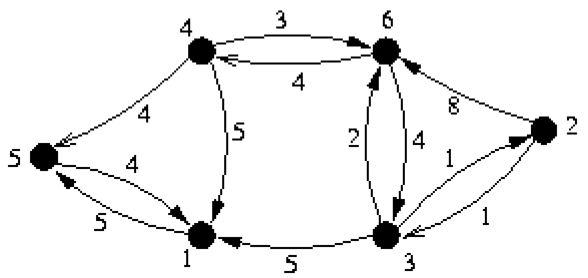
Now the network's maximum capacity path is {5,4,6,3,2}



This flow has only a capacity of 1, as the arc from 5 to 4 has capacity 1. Once again, update the forward and backwards arcs appropriately.



The resulting graph has no paths from the source to the sink. The only nodes reachable from the source node 5 are node 5 itself and node 1.



The algorithm added three flows, the first with capacity 5, the second with capacity 3, and the last with capacity 1. Thus, the maximum flow through the network from node 5 to node 2 is 9.

Extensions

Network flow problems are very extensible, mostly by playing with the graph.

To extend to the case of undirected graphs, simple expand the edge as two arcs in opposite directions.

If you want to limit the amount of traffic through any node, split each node into two nodes, an in-node and and out-node. Put all the in-arcs into the in-node, and all of the out-arcs out of the out-node and place an arc from the in-node to the out-node with capacity equal to the capacity of the node.

If you have multiple sources and sinks, create a `virtual source' and `virtual sink' with arcs from the virtual source to each of the sources and arcs from each of the sinks to the virtual sink. Make each of the added arcs have infinite capacity.

If you have arcs with real-valued weights, then this algorithm is no longer guaranteed to terminate, although it will asymptotically approach the maximum.

Alternative Problems

Network flow can also be used to solve other types of problems that aren't so obvious

Maximum Matching

Given a two sets of objects (call them A and B), where you want to `match' as many individual A objects with individual B objects as possible, subject to the constraint that only certain pairs are possible (object A1 can be matched with object B3, but not object B1 or B2). This is called the `maximum matching' problem.

To reformulate this as network flow, create a source and add an arc with capacity 1 from this source to each A object. Create a sink with an arc from each B object to it with capacity 1. In addition, if object Ai and Bk may be matched together, add an arc from Ai to Bk with capacity 1. Now run the algorithm and determine which arcs between A objects and B objects are used.

Minimum Cut

Given a weight undirected graph, what is the set of edges with minimum total weight such that it separates two given nodes.

The minimum total weight is exactly the flow between those two nodes.

To determine the path, try removing each edge in increasing weight order, and seeing if it reduces the network flow (if it does, it should reduce the flow by the capacity of that edge. The first one which does is a member of the minimum cut, iterate on the graph without that edge.

This can be extended to node cuts by the same trick as nodes with limited capacity. Directed graphs work using the same trick. However, it can not solve the problem of finding a so-called `best match,' where each pairing has a `goodness' value, and you want to create the matching which has the highest total `goodness.'

Example Problems

If the problems talks about maximizing the movement or flow of something from one location to another, it is almost assuredly maximum flow. If it talks about trying to separate two items minimally, it is probably minimum cut. If it talks about maximizes the pairing of any sort of thing (person, object, money, whatever), it is probably maximum matching.

Virus Flow

You have a computer network, with individual machines connected together by wires. Data may flow either direction on the wire. Unfortunately, a machine on your network has caught a virus, so you need to separate this machine from your central server to stop the spread of this virus. Given the cost of shutting down the network connection between each pair of machines, calculate the minimum amount of money which must be spent to separate the contaminated machine from your server.

This is exactly the min cut problem.

Lumberjack Scheduling

Different types of trees require different techniques to be employed by lumberjacks for them to harvest the tree properly. Regardless of the tree or lumberjack, harvest a tree requires 30 minutes. Given a collection of lumberjacks, and the types of trees that each one is able to correctly cut down, and a collection of trees, calculate the maximum number of trees which may be cut down in the next half hour.

Each lumberjack can be paired with each tree of a type that he/she is able to properly harvest. Thus, the problem can be solved using the maximum matching algorithm.

Telecowmunication (USACO Championship 1996)

Given a group of computers in the field, along with the wires running between the computers, what is the minimum number of machines which may crash before two given machines are the network are unable to communicate? Assume that the two given machines will not crash.

This is equivalent to the minimum node cut problem. The two given machines can be arbitrarily labeled the source and sink. The wires are bidirectional. Split each node into an in-node and an out-node, so that we limit the flow through any given machine to 1. Now, the maximum flow across this network is equivalent to the minimum node cut.

To actually determine the cut, iterative remove the nodes until you find one which lowers the capacity of the network.

Science Fair Judging

A science fair has N categories, and M judges. Each judge is willing to judge some subset of the categories, and each category needs some number of judges. Each judge is only able to judge one category at a given science fair. How many judges can you assign subject to these constraints?

This is very similar to the maximum matching problem, except that each category can handle possibly more than one judge. The easiest way to do this is to increase the capacity of the arcs from categories to the sink to be the number of judges required.

Oil Pipe Planning

Given the layout (the capacity of each pipe, and how the pipes are connected together) of the pipelines in Alaska, and the location of each of the intersections, you wish to increase the maximum flow between Juneau and Fairbanks, but you have enough money to only add one pipe of capacity X. Moreover, the pipe can only be 10 miles long. Between which two intersections should this pipe be added to increase the flow the most?

To solve this problem, for each pair of intersections within 10 miles of each other, calculate the increase in the flow between Juneau and Fairbanks if you add a pipe between the intersections. Each of these sub-problems is exactly maximum flow.

Network Flow

网络流

译 By Thunder

问题预备

最短路

问题

给出一个有向连通带权图，包含源点和汇点。

每一条弧的权表示那条弧的容量。一个图的流通过分配整数量给每条边来达到

通过每条弧的流不大于这条弧的容量

除了源点和汇点的每个点，流入量等于流出量

使源点的流出总量减去流入总量（或汇点的流入总量减去流出总量)最大

例

给出水管的设计和每条水管的容量。水管里的水必须从上往下流，所以每条水管里，水只能向一个方向流。

计算能从头（自来水厂）流到尾（您的农场）的水量。

算法

算法（贪心）通过迭代增加从源到汇的流来建立网络流。

从每条弧的权等于初始的权开始（弧的权符合那条弧里还未使用的容量）。

给出当前的图，找到从源到汇的一条路径，这条路径上的弧都是非 0 权。计算这条路径的最大流，叫它路径容量。//增广路径 路径容量 = 瓶颈容量

对于每一条路径上的弧，给弧的权减去路径容量。另外，给路径中的反向弧（在相同两点之间的弧，不过方向相反）加上等于和路径容量相等的权（只要反向弧存在，就增加它的权）。//调整增广路径

继续调整路径直到没有增广路径存在。

这能保证停止，因为您加上每次至少一单位流（权总是整数），并且流严格单调递增。增加反向弧的权相当于减少路径上的流量。

如果您对算法的细节分析感兴趣，请向 Sedgewick 请教(谁啊...).

这是算法的伪代码

```
IF (源点 = 汇点)
    总流 = 无限大
    结束

总流 = 0

WHILE (TRUE)
# 找到从源到汇的最大的容量的路径
# 用修改过的 Dijkstra 算法 (在 Chapter 4 的 ditches 就要用 Bellman-Ford 了)
  FOR 所有的顶点 i
    前驱(i) = nil
    flow(i) = 0
    visited(i) = FALSE
  flow(源点) = 无限大

  WHILE (TRUE)
    最大流 = 0
    最大流节点 = nil
    #  找到最大容量的未访问节点
    FOR 每个顶点 i
      IF ((flow(i)>最大流) AND (NOT visited(i)))
        最大流 = flow(i)
        最大流节点 = i
    IF (最大流节点 = Nil)
```

```
          退出内层 While 循环
      IF (最大流节点 = 汇点)
          退出内层 While 循环
      visited(最大流节点) = TRUE
        #   调整相邻节点
      FOR 最大流节点的所有相邻节点 i
        IF (flow(i)<min(最大流, 最大流节点到 i 的容量))
            前驱(i) = 最大流节点
            flow(i) = min(最大流, 最大流节点到 i 的容量))
    IF (最大流节点 = Nil)    #   没有路径
      退出外层 While 循环

    路径容量 = flow(汇点)
    总流 = 总流 + 路径容量

# 把那个流加到网络里 适当调整容量
    当前节点 = 汇点
#   对于增广路径上每条弧, 前驱(当前节点), 当前节点
    WHILE (当前节点不为源点)
      下一个节点 = 前驱(当前节点)
      下一个节点到当前节点的容量减去路径容量
      当前节点到下一个节点的容量加上路径容量
      当前节点 = 下个节点
```

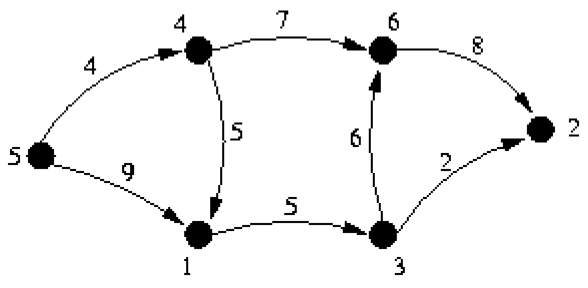
这个方法的运行时间为 O(FM)，F 是最大流 M 是弧的数目. 运行时间通常更短, 因为算法每次总是尽可能大的提升流。

为了确定每条弧上的流，比较开始的容量和最后的容量。如果最后的容量变少了，区别就在于通过这条弧的流。

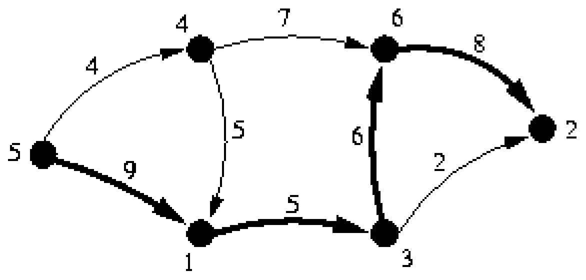
当有回路的时候，这个算法可能产生死循环。

实例

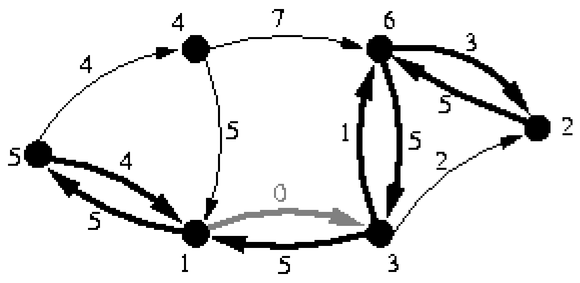
考虑下面那个网络，源点为 5，汇点为 2



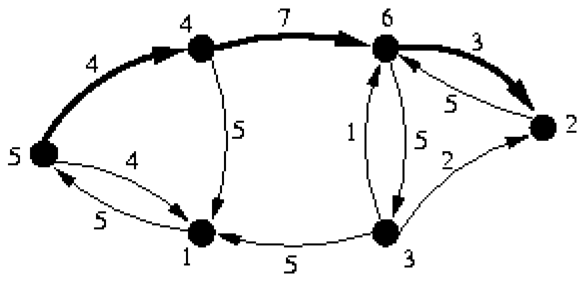
最大增广路径为(5,1,3,6,2)



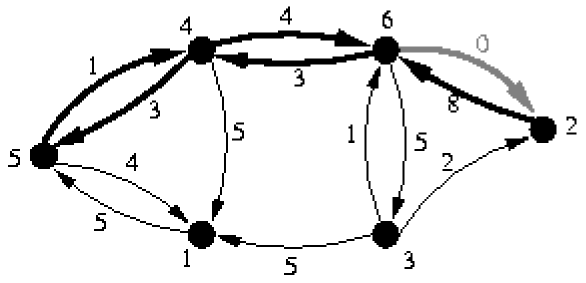
瓶颈弧是 1->3，容量为 5。因此，把路径上所有的弧减去 5，把反向弧上加上 5（如果需要，添加弧）。这就是得到的图：



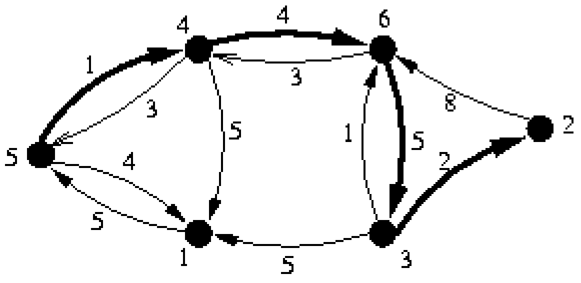
在新的图里，最大增广路径为(5,4,6,2)



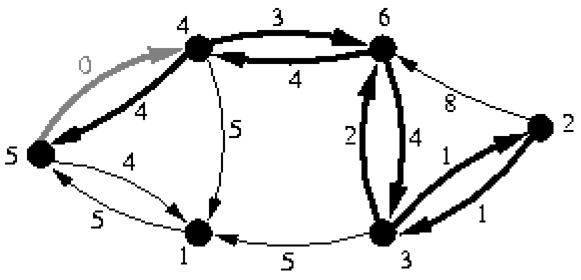
瓶颈容量为 3，再来一次。



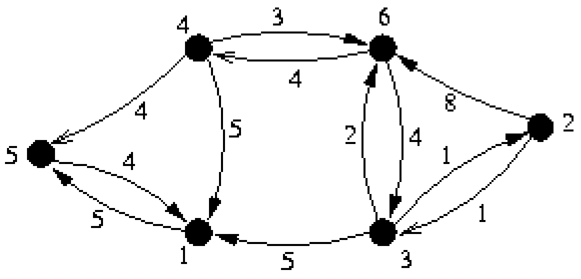
现在网络的最大容量路径为(5,4,6,3,2)



瓶颈为 1，再来一次



结果图从源到汇就没有增广路径了。唯一可以到达的源点的就是它自己和 1 号节点



算法加了三次流，第一次是 5，第二次是 3，最后一次是 1。因此，最大流是 9。

扩展

网络流问题是很广泛的，通常和图结合在一起。

扩展到无向图，只需要简单的把无向边扩展为两条相反方向的弧就可以了。

如果您想要限制任意一个节点的通过量，把一个节点拆成两个，一个流入节点，一个流出节点，把流入的弧连到流入节点里，流出弧连到流出节点里，在流入节点和流出节点之间加上一条弧，容量为限制。

如果您有多源点汇点，建立一个虚拟的源点和虚拟的汇点，虚拟的源点射出弧到每个源点，汇点射出弧到虚拟汇点。容量无限。

如果您有实数型的权，这个问题不再保证能出解，尽管答案能逐渐逼近。

可选问题

网络流能用来解决其他的一些不明显的问题。

最大匹配

给出两个对象集合（叫他们 A 和 B），你想要把 A 和 B 中的元素尽可能多的匹配起来，约束条件是只可以两两配对。这就是最大匹配问题。

把这个问题用网络流的形式表现出来，建立一个源点，射出容量为 1 的弧给 A 的元素，建立一个汇点，从 B 中的元素射出容量为 1 的弧到此汇点。另外，如果 A 中某个元素可 以和 B 中某个元素匹配，从 A 的这个元素向 B 的那个元素射出一条容量为 1 的弧。用网络流算法确定哪些 A 元素到 B 元素的弧被用到。

最小割

给出一个带权无向图，最小割就是一个权和最小的，能够把两个给出点分开的边的集合。

最小的权和就是两个节点之间的流。

要确定路径，按照权递增的顺序尝试删除每条边，看它是否减少了网络流(如果减少了，减少量应为边的容量)。第一个这样的边就是最小割的元素，去掉这条边继续在图上操作。

用相同的技巧，把节点用容量限制，这能够扩展到求节点的割。有向图也是一样。然而，它不能够解决所谓的“最佳匹配”问题，每一对都有一个“佳值”，而且您想要得到最高“佳值”和的匹配。

例题

如果问题讨论取流或者其他从一个地方到另一个地方东西的移动的最优值，就可以几乎确定是最大流。如果它讨论的是将两个物料项目以最小代价分隔开的问题，就可能是最小割。如果它讨论的任何一类东西的最大配对，就可能是最大匹配。

病毒流

您有一个通过网线连接的计算机网络。数据可能从任何一个方向流经网线。不幸的是，一台网络里的机器染上病毒，您需要从中心服务器隔离这台机器来防止病毒传染。给出关闭一对机器连接的代价，计算控制病毒传染到服务器的最小的代价。

这就是最小割问题。

伐木工人的计划

不同种类的树需要雇佣拥有不同的技巧的伐木工人来正确的砍树。不管哪种树或伐木工人，砍一棵树需要 30 分钟。给出伐木工人的信息，和哪种树需要哪位才能正确的砍倒，以及树的信息，计算在半小时内砍倒的最多的树的数目。对于每一个伐木工人，都有一种对应的他（她）能砍倒的树。所以，这个问题能用最大匹配算法来解决。

牛的电话联系（USACO 锦标赛 1996）

给出特定区域里的一组电脑，和电脑之间连着的网线，要想阻止给定两台电脑的联系 最少需要关掉多少网络中的电脑。假定两台给出的电脑没有关掉。

这相当于最少节点割的问题。两台给出的电脑标上源点和汇点。电缆是双向的。把每个点拆成流入点和流出点，这样我们就能够用 1 限制任何一台电脑。现在，网络里的最大流就相当于节点最小割了。

为了具体确定割，反复删除节点，直到您找到一个能降低网络流量的节点。

科学展览审定

一个科学展览由 N 个学科，和 M 个裁判。每个裁判愿意审定某些学科，每个学科需要一些裁判。每个裁判只能够审定在给出的科学展览中的一个学科。您在这些条件下总共能分配给多少裁判任务？

这是一个很类似最大匹配的问题，除了每个学科需要可能多于一个裁判。解决这个问题的最简单的方法是把从科目到汇点的弧的容量赋予需要裁判数的值。

石油管道设计

给出阿拉斯加管道线的设计（每条管道的容量，和管道如何连接），以及每个交点的位置，您想提高朱诺和费尔班柯斯之间最大流量，但您的钱只够添加一条容量为 X 的管道。此外，管道只能 10 英里长。这条管道添加在哪两个交点能最大程度的提高流量？

要解决这个问题，对于距离 10 英里以内的每一对交点，添加一条管道，再计算从朱诺到费尔班柯斯的流量增加值。每一个子问题都是最大流。