Algorithm Tutorials

Maximum Flow

By _efer_
TopCoder Member

Introduction

This article covers a problem that often arises in real life situations and, as expected, in programming contests, with Top Coder being no exception. It is addressed mostly to coders who are not familiar with the subject, but it may prove useful to the more experienced as well. Lots of papers have been written, and there are many algorithms known to solve this problem. While they are not the fastest, the algorithms presented here have the advantage of being simple and efficient, and because of this they are usually preferred during a contest setup. The reader is advised to read the article on graph theory first, as the concepts presented there are needed to understand those presented here.

The Standard Maximum Flow Problem

So, what are we being asked for in a max-flow problem? The simplest form that the statement could take would be something along the lines of: "A list of pipes is given, with different flow-capacities. These pipes are connected at their endpoints. What is the maximum amount of water that you can route from a given starting point to a given ending point?" or equivalently "A company owns a factory located in city X where products are manufactured that need to be transported to the distribution center in city Y. You are given the one-way roads that connect pairs of cities in the country, and the maximum number of trucks that can drive along each road. What is the maximum number of trucks that the company can send to the distribution center?"

A first observation is that it makes no sense to send a truck to any other city than Y, so every truck that enters a city other than Y must also leave it. A second thing to notice is that, because of this, the number of trucks leaving X is equal to the number of trucks arriving in Y.

Rephrasing the statement in terms of graph theory, we are given a network - a directed graph, in which every edge has a certain capacity *c* associated with it, a starting vertex (the source, X in the example above), and an ending vertex (the sink). We are asked to associate another value *f* satisfying $f \leq c$ for each edge such that for every vertex other than the source and the sink, the sum of the values associated to the edges that enter it must equal the sum of the values associated to the edges that leave it. We will call *f* the flow along that edge. Furthermore, we are asked to maximize the sum of the values associated to the arcs leaving the source, which is the total flow in the network.

The image below shows the optimal solution to an instance of this problem, each edge being labeled with the values f/c associated to it.

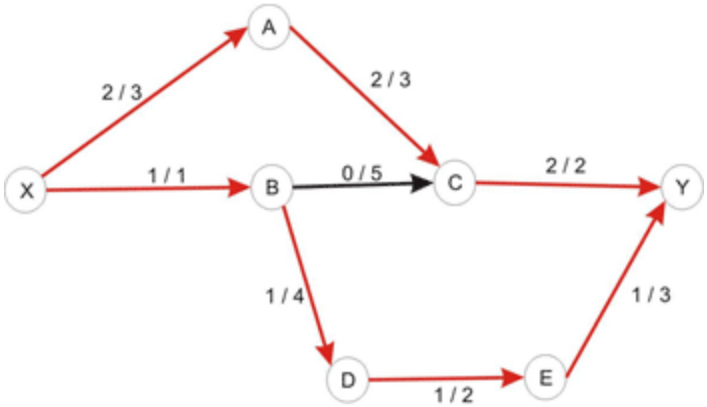


Figure 1a - Maximum Flow in a network

How to Solve It

Now how do we actually solve the problem? First, let us define two basic concepts for understanding flow networks: residual networks and augmenting paths. Consider an arbitrary flow in a network. The residual network has the same vertices as the original network, and one or two edges for each edge in the original. More specifically, if the flow along the edge x-y is less than the capacity there is a forward edge x-y with a capacity equal to the difference between the capacity and the flow (this is called the residual capacity), and if the flow is positive there is a backward edge y-x with a capacity equal to the flow on x-y. An augmenting path is simply a path from the source to the sink in the residual network, whose purpose is to increase the flow in the original one. It is important to understand that the edges in this path can point the "wrong way" according to the original network. The path capacity of a path is the minimum capacity of an edge along that path. Let's take the following example:

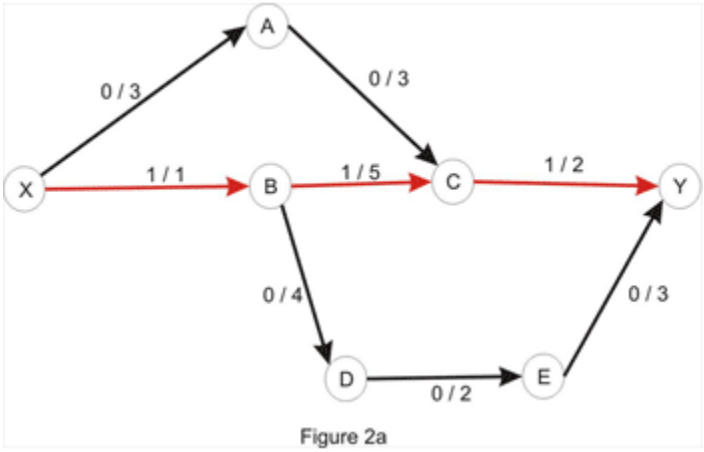


Figure 2a

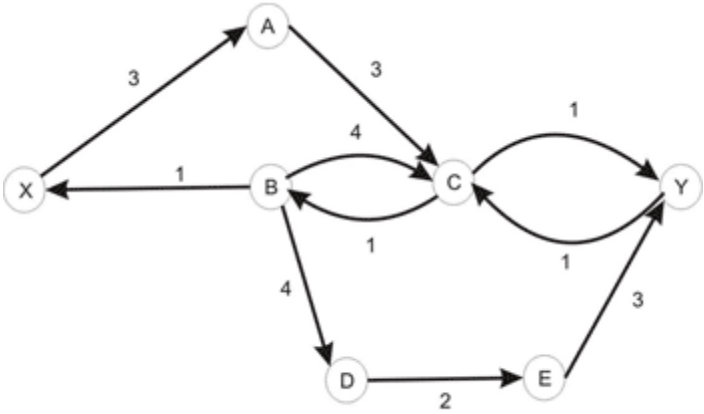


Figure 2b - The residual network of the network in 2a

By considering the path X_A_C_Y, we can increase the flow by 1 - the edges X_A and A_C have capacity of 3, as in the original network, but the edge C_Y has capacity 1, and we take the minimum of these values to get the path capacity. Increasing the flow along this path with 1 yields the flow below:

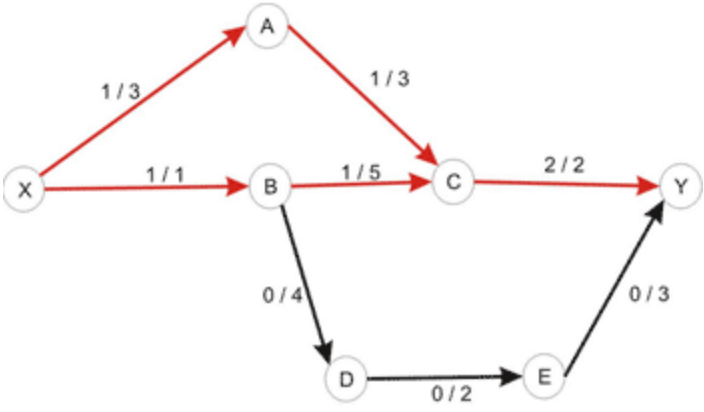


Figure 3a

The value of the current flow is now 2, and as shown in Figure 1, we could do better. So, let's try to increase the flow. Clearly, there is no point in considering the directed paths X_A_C_Y or X_B_D_E_Y as the edges C_Y and X_B, respectively, are filled to capacity. As a matter of fact, there is no directed path in the network shown above, due to the edges mentioned above being filled to capacity. At this point, the question that naturally comes to mind is: is it possible to increase the flow in this case? And the answer is yes, it is. Let's take a look at the residual network:

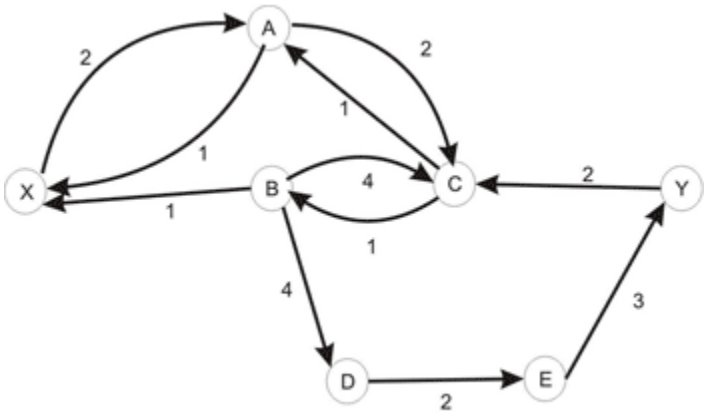


Figure 3b - The residual network of the network in 3a

Let's consider the only path from X to Y here: X_A_C_B_D_E_Y. Note that this is not a path in the directed graph, because C_B is walked in the opposite way. We'll use this path in order to increase the total flow in the original network. We'll "push" flow on each of the edges, except for C_B which we will use in order to "cancel" flow on B_C. The amount by which this operation can be performed is limited by the capacities of all edges along the path (as shown in Figure 3b). Once again we take the minimum, to conclude that this path also has capacity 1. Updating the path in the way described here yields the flow shown in Figure 1a. We are left with the following residual network where a path between the source and the sink doesn't exist:

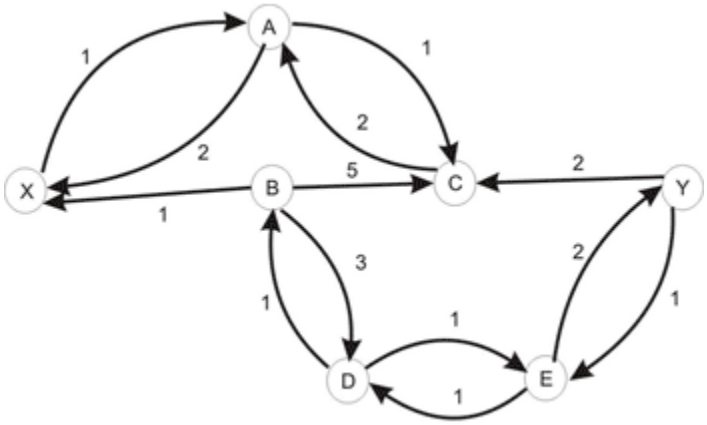


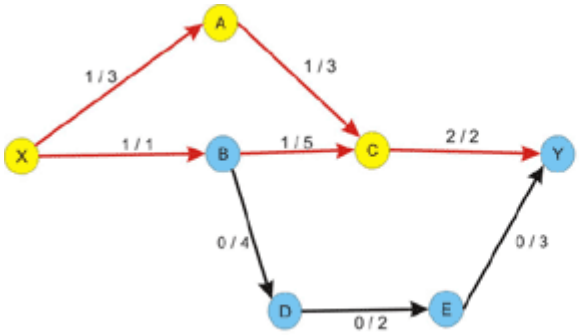
Figure 1b - The residual network of the network in 1a

This example suggests the following algorithm: start with no flow everywhere and increase the total flow in the network while there is an augmenting path from the source to the sink with no full forward edges or empty backward edges - a path in the residual network. The algorithm (known as the Ford-Fulkerson method) is guaranteed to terminate: due to the capacities and flows of the edges being integers and the path-capacity being positive, at each step we get a new flow that is closer to the maximum. As a side note, the algorithm isn't guaranteed to even terminate if the capacities are irrationals.

What about the correctness of this algorithm? It is obvious that in a network in which a maximum flow has been found there is no augmenting path, otherwise we would be able to increase the maximum value of the flow, contradicting our initial assumption. If the converse of this affirmation is true, so that when there is no augmenting path, the value of the flow has reached its maximum, we can breathe a sigh of relief, our algo is correct and computes the maximum flow in a network. This is known as the max-flow min-cut theorem and we shall justify its correctness in a few moments.

A cut in a flow network is simply a partition of the vertices in two sets, let's call them A and B, in such a way that the source vertex is in A

and the sink is in B. The capacity of a cut is the sum of the capacities of the edges that go from a vertex in A to a vertex in B. The flow of the cut is the difference of the flows that go from A to B (the sum of the flows along the edges that have the starting point in A and the ending point in B), respectively from B to A, which is exactly the value of the flow in the network, due to the entering flow equals leaving flow - property, which is true for every vertex other than the source and the sink.

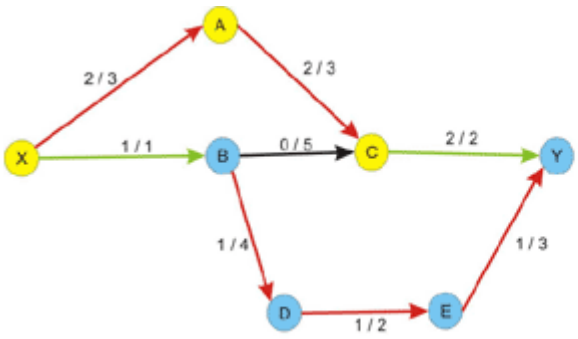


The yellow vertices are in set A and those in blue are in set B.
The capacity of the cut is the sum of the capacities of the edges XB and CY: 1 + 2 = 3.
The flow of the cut is the sum of the flow along the edges XB and CY minus the flow on BC: 1 + 2 - 1 = 2, and is equal to the value of the flow.

Figure 4 - A cut in a network

Notice that the flow of the cut is less or equal to the capacity of the cut due to the constraint of the flow being less or equal to the capacity of every edge. This implies that the maximum flow is less or equal to every cut of the network. This is where the max-flow min-cut theorem comes in and states that the value of the maximum flow through the network is exactly the value of the minimum cut of the network. Let's give an intuitive argument for this fact. We will assume that we are in the situation in which no augmenting path in the network has been found. Let's color in yellow, like in the figure above, every vertex that is reachable by a path that starts from the source and consists of non-full forward edges and of non-empty backward edges. Clearly the sink will be colored in blue, since there is no augmenting path from the source to the sink. Now take every edge that has a yellow starting point and a blue ending point. This edge will have the flow equal to the capacity, otherwise we could have added this edge to the path we had at that point and color the ending point in yellow. Note that if we remove these edges there will be no directed path from the source to the sink in the graph. Now consider every edge that has a blue starting point and a yellow ending point. The flow on this edge must be 0 since otherwise we could have added this edge as a backward edge on the current path and color the starting point in yellow. Thus, the value of the flow must equal the value of the cut, and since every flow is less or equal to every cut, this must be a maximum flow, and the cut is a minimum cut as well.

In fact, we have solved another problem that at first glance would appear to have nothing to do with maximum flow in a network, ie. given a weighted directed graph, remove a minimum-weighted set of edges in such a way that a given node is unreachable from another given node. The result is, according to the max-flow min-cut theorem, the maximum flow in the graph, with capacities being the weights given. We are also able to find this set of edges in the way described above: we take every edge with the starting point marked as reachable in the last traversal of the graph and with an unmarked ending point. This edge is a member of the minimum cut.



The minimum cut of this network is the sum of the capacities of XB and CY and equals 3, which is also the value of the maximum flow.

Figure 5 - A minimum cut in a network

Augmenting-Path Algorithms

The neat part of the Ford-Fulkerson algorithm described above is that it gets the correct result no matter how we solve (correctly!!) the sub-problem of finding an augmenting path. However, every new path may increase the flow by only 1, hence the number of iterations of the algorithm could be very large if we carelessly choose the augmenting path algorithm to use. The function *max_flow* will look like this, regardless of the actual method we use for finding augmenting paths:

```
int max_flow()
result = 0
  while (true)
// the function find_path returns the path capacity of the
    augmenting path found
    path_capacity = find_path()
    // no augmenting path found
    if (d = 0) exit while
    else result += path_capacity
  end while
  return result
```

To keep it simple, we will use a 2-dimensional array for storing the capacities of the residual network that we are left with after each step in the algorithm. Initially the residual network is just the original network. We will not store the flows along the edges explicitly, but it's easy to figure out how to find them upon the termination of the algorithm: for each edge x-y in the original network the flow is given by the capacity of the backward edge y-x in the residual network. Be careful though; if the reversed arc y-x also exists in the original network, this will fail, and it is recommended that the initial capacity of each arc be stored somewhere, and then the flow along the edge is the difference between the initial and the residual capacity.

We now require an implementation for the function *find_path*. The first approach that comes to mind is to use a depth-first search (DFS), as it probably is the easiest to implement. Unfortunately, its performance is very poor on some networks, and normally is less preferred to the ones discussed next.

The next best thing in the matter of simplicity is a breadth-first search (BFS). Recall that this search usually yields the shortest path in an un-weighted graph. Indeed, this also applies here to get the shortest augmenting path from the source to the sink. In the following pseudocode we will basically: find a shortest path from the source to the sink and compute the minimum capacity of an edge (that could be a forward or a backward edge) along the path - the path capacity. Then, for each edge along the path we reduce its capacity and increase the capacity of the reversed edge with the path capacity.

```
int bfs()
  queue Q
  push source to Q
  mark source as visited
  keep an array from with the semnification: from[x] is the
previous vertex visited in the shortest path from the source to x;
initialize from with -1 (or any other sentinel value)
  while Q is not empty
    where = pop from Q
    for each vertex next adjacent to where
      if next is not visited and capacity[where][next] > 0
        push next to Q
        mark next as visited
        from[next] = where
        if next = sink
          exit while loop
    end for
  end while
  // we compute the path capacity
  where = sink, path_cap = infinity
  while from[where] > -1
    prev = from[where] // the previous vertex
    path_cap = min(path_cap, capacity[prev][where])
    where = prev
  end while
  // we update the residual network; if no path is found the while
loop will not be entered
  where = sink
  while from[where] > -1
```

```
    prev = from[where]
    capacity[prev][where] -= path_capacity
        capacity[where][prev] += path_capacity
    where = prev
  end while
  // if no path is found, path_cap is infinity
  if path_cap = infinity
    return 0
  else return path_cap
```

As we can see, this is pretty easy to implement. As for its performance, it is guaranteed that this takes at most $N * M/2$ steps, where N is the number of vertices and M is the number of edges in the network. This number may seem very large, but it is over-estimated for most networks. For example, in the network we considered 3 augmenting paths are needed which is significantly less than the upper bound of 28. Due to the $O(M)$ running time of BFS (implemented with adjacency lists) the worst-case running time of the shortest-augmenting path max-flow algorithm is $O(N * M^2)$, but usually the algorithm performs much better than this.

Next we will consider an approach that uses a priority-first search (PFS), that is very similar to the Dijkstra heap method explained here. In this method the augmenting path with a maximum path capacity is preferred. Intuitively this would lead to a faster algorithm, since at each step we increase the flow with the maximum possible amount. However, things are not always so, and the BFS implementation has better running times on some networks. We assign as a priority to each vertex the minimum capacity of a path (in the residual network) from the source to that vertex. We process vertices in a greedy manner, as in Dijkstra's algorithm, in decreasing order of priorities. When we get to the sink, we are done, since a path with a maximum capacity is found. We would like to implement this with a data structure that allows us to efficiently find the vertex with the highest priority and increase the priority of a vertex (when a new better path is found) - this suggests the use of a heap which has a space complexity proportional to the number of vertices. In TopCoder matches we may find it faster and easier to implement this with a priority queue or some other data structure that approximates one, even though the space required might grow to being proportional with the number of edges. This is how the following pseudocode is implemented. We also define a structure `node` that has the members `vertex` and `priority` with the above significance. Another field `from` is needed to store the previous vertex on the path.

```
int pfs()
  priority queue PQ
  push node(source, infinity, -1) to PQ
  keep the array from as in bfs()
  // if no augmenting path is found, path_cap will remain 0
  path_cap = 0
  while PQ is not empty
    node aux = pop from PQ
    where = aux.vertex, cost = aux.priority
    if we already visited where continue
    from[where] = aux.from
    if where = sink
      path_cap = cost
      exit while loop
    mark where as visited
    for each vertex next adjacent to where
      if capacity[where][next] > 0
        new_cost = min(cost, capacity[where][next])
        push node(next, new_cost, where) to PQ
    end for
  end while
  // update the residual network
  where = sink
  while from[where] > -1
    prev = from[where]
    capacity[prev][where] -= path_cap
    capacity[where][prev] += path_cap
    where = prev
```

```
    end while
    return path_cap
```

The analysis of its performance is pretty complicated, but it may prove worthwhile to remember that with PFS at most $2M1gU$ steps are required, where U is the maximum capacity of an edge in the network. As with BFS, this number is a lot larger than the actual number of steps for most networks. Combine this with the $O(M\ 1\mathrm{g}\ M)$ complexity of the search to get the worst-case running time of this algorithm.

Now that we know what these methods are all about, which of them do we choose when we are confronted with a max-flow problem? The PFS approach seems to have a better worst-case performance, but in practice their performance is pretty much the same. So, the method that one is more familiar with may prove more adequate. Personally, I prefer the shortest-path method, as I find it easier to implement during a contest and less error prone.

...continue to Section 2

Section 2

Max-Flow/Min-Cut Related Problems

How to recognize max-flow problems? Often they are hard to detect and usually boil down to maximizing the movement of something from a location to another. We need to look at the constraints when we think we have a working solution based on maximum flow - they should suggest at least an $O(N^3)$ approach. If the number of locations is large, another algorithm (such as dynamic programming or greedy), is more appropriate.

The problem description might suggest multiple sources and/or sinks. For example, in the sample statement in the beginning of this article, the company might own more than one factory and multiple distribution centers. How can we deal with this? We should try to convert this to a network that has a unique source and sink. In order to accomplish this we will add two "dummy" vertices to our original network - we will refer to them as super-source and super-sink. In addition to this we will add an edge from the super-source to every ordinary source (a factory). As we don't have restrictions on the number of trucks that each factory can send, we should assign to each edge an infinite capacity. Note that if we had such restrictions, we should have assigned to each edge a capacity equal to the number of trucks each factory could send. Likewise, we add an edge from every ordinary sink (distribution centers) to the super-sink with infinite capacity. A maximum flow in this new-built network is the solution to the problem - the sources now become ordinary vertices, and they are subject to the entering-flow equals leaving-flow property. You may want to keep this in your bag of tricks, as it may prove useful to most problems.

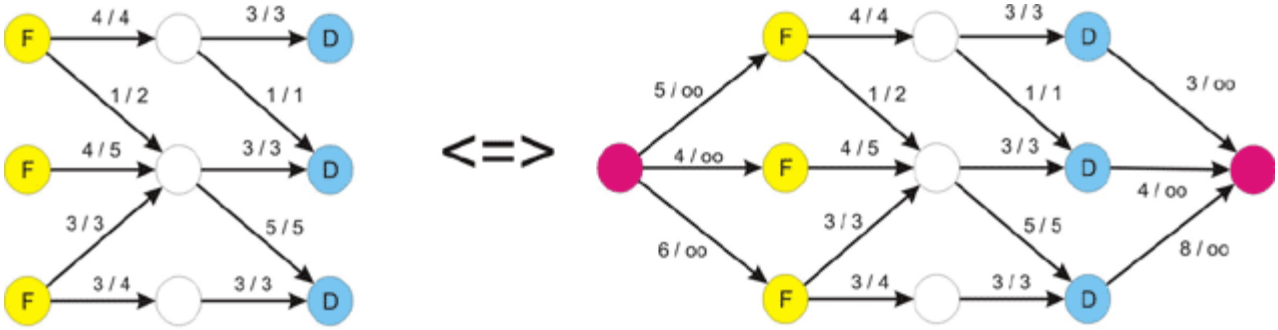


Figure 6 - Reduction of a multiple-source / multiple-sink max-flow problem

What if we are also given the maximum number of trucks that can drive through each of the cities in the country (other than the cities where the factory and the distribution center are located)? In other words we have to deal with vertex-capacities too. Intuitively, we should be able to reduce this to maximum-flow, but we must find a way to take the capacities from vertices and put them back on edges, where they belong. Another nice trick comes into play. We will build a network that has two times more vertices than the initial one. For each vertex we will have two nodes: an in-vertex and an out-vertex, and we will direct each edge x-y from the out-vertex of x to the in-vertex of y. We can assign them the capacities from the problem statement. Additionally we can add an edge for each vertex from the in to the out-vertex. The capacity this edge will be assigned is obviously the vertex-capacity. Now we just run max-flow on this network and compute the result.

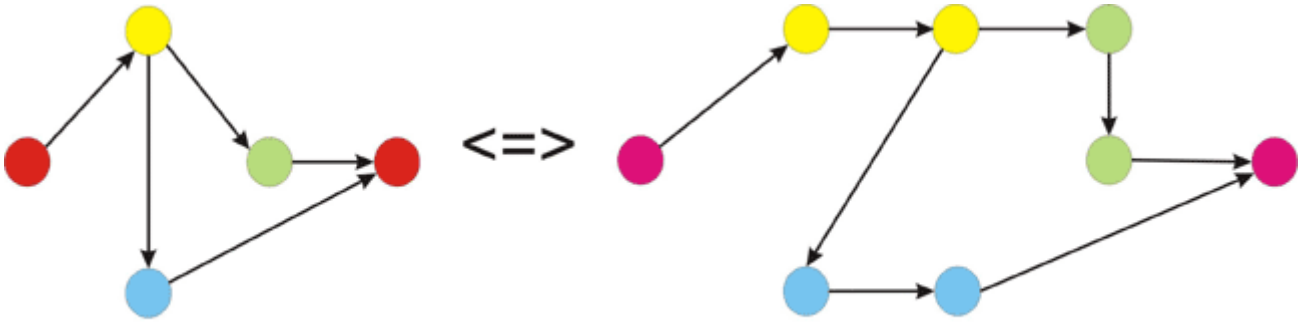


Figure 7 - Eliminating vertex-capacities

Maximum flow problems may appear out of nowhere. Let's take this problem for instance: "You are given the in and out degrees of the vertices of a directed graph. Your task is to find the edges (assuming that no edge can appear more than once)." First, notice that we can perform this simple test at the beginning. We can compute the number M of edges by summing the out-degrees or the in-degrees of the vertices. If these numbers are not equal, clearly there is no graph that could be built. This doesn't solve our problem, though. There are some greedy approaches that come to mind, but none of them work. We will combine the tricks discussed above to give a max-flow algorithm that solves this problem. First, build a network that has 2 (in/out) vertices for each initial vertex. Now draw an edge from every out vertex to every in vertex. Next, add a super-source and draw an edge from it to every out-vertex. Add a super-sink and draw an edge from every in vertex to it. We now need some capacities for this to be a flow network. It should be pretty obvious what the intent with this approach is, so we will assign the following capacities: for each edge drawn from the super-source we assign a capacity equal to the out-degree of the vertex it points to. As there may be only one arc from a vertex to another, we assign a 1 capacity to each of the edges that go from the outs to the ins. As you can guess, the capacities of the edges that enter the super-sink will be equal to the in-degrees of the vertices. If the maximum flow in this network equals M - the number of edges, we have a solution, and for each edge between the out and in vertices that has a flow along it (which is maximum 1, as the capacity is 1) we can draw an edge between corresponding vertices in our graph. Note that both x-y and y-x edges may appear in the solution. This is very similar to the maximum matching in a bipartite graph that we will discuss later. An example is given below where the out-degrees are (2, 1, 1, 1) and the in-degrees (1, 2, 1, 1).

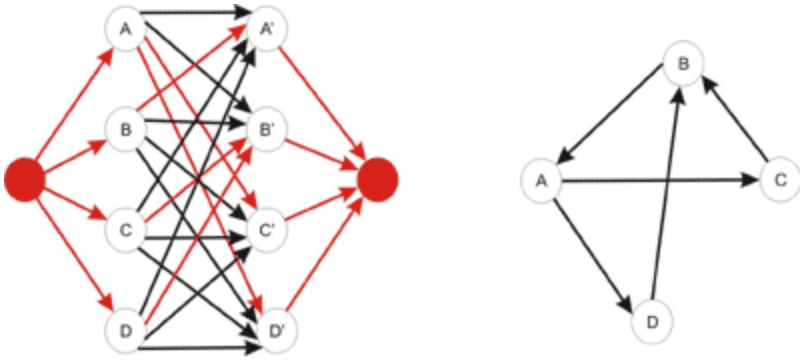


Figure 8

Some other problems may ask to separate two locations minimally. Some of these problems usually can be reduced to minimum-cut in a network. Two examples will be discussed here, but first let's take the standard min-cut problem and make it sound more like a TopCoder problem. We learned earlier how to find the value of the min-cut and how to find an arbitrary min-cut. In addition to this we will now like to have a minimum-cut with the minimum number of edges. An idea would be to try to modify the original network in such a way that the minimum cut here is the minimum cut with the minimum edges in the original one. Notice what happens if we multiply each edge capacity with a constant T. Clearly, the value of the maximum flow is multiplied by T, thus the value of the minimum cut is T times bigger than the original. A minimum cut in the original network is a minimum cut in the modified one as well. Now suppose we add 1 to the capacity of each edge. Is a minimum cut in the original network a minimum cut in this one? The answer is no, as we can see in Figure 8 shown below, if we take T = 2.

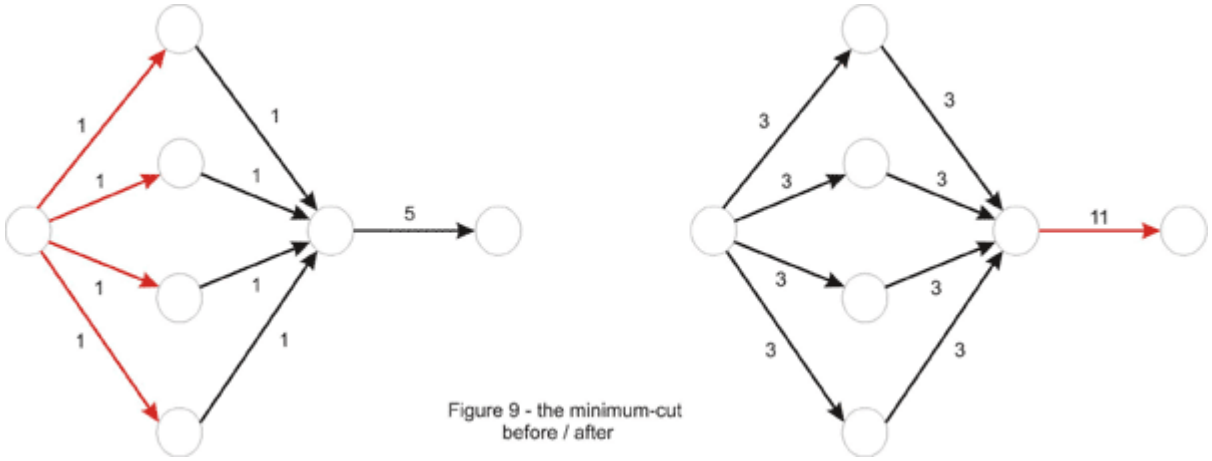


Figure 9 - the minimum-cut before / after

Why did this happen? Take an arbitrary cut. The value of the cut will be T times the original value of the cut, plus the number of edges in it. Thus, a non-minimum cut in the first place could become minimum if it contains just a few edges. This is because the constant might not have been chosen properly in the beginning, as is the case in the example above. We can fix this by choosing T large enough to neutralize the difference in the number of edges between cuts in the network. In the above example T = 4 would be enough, but to generalize, we take T = 10, one more than the number of edges in the original network, and one more than the number of edges that could possibly be in a minimum-cut. It is now true that a minimum-cut in the new network is minimum in the original network as well. However the converse is not true, and it is to our advantage. Notice how the difference between minimum cuts is now made by the number of edges in the cut. So we just find the min-cut in this new network to solve the problem correctly.

Let's illustrate some more the min-cut pattern: "An undirected graph is given. What is the minimum number of edges that should be removed in order to disconnect the graph?" In other words the problem asks us to remove some edges in order for two nodes to be separated. This should ring a bell - a minimum cut approach might work. So far we have only seen maximum flow in directed graphs, but now we are facing an undirected one. This should not be a very big problem though, as we can direct the graph by replacing every (undirected) edge x-y with two arcs: x-y and y-x. In this case the value of the min-cut is the number of edges in it, so we assign a 1 capacity to each of them. We are not asked to separate two given vertices, but rather to disconnect optimally any two vertices, so we must take every pair of vertices and treat them as the source and the sink and keep the best one from these minimum-cuts. An improvement can be made, however. Take one vertex, let's say vertex numbered 1. Because the graph should be disconnected, there must be another vertex unreachable from it. So it suffices to treat vertex 1 as the source and iterate through every other vertex and treat it as the sink.

What if instead of edges we now have to remove a minimum number of vertices to disconnect the graph? Now we are asked for a different min-cut, composed of vertices. We must somehow convert the vertices to edges though. Recall the problem above where we converted vertex-capacities to edge-capacities. The same trick works here. First "un-direct" the graph as in the previous example. Next double the number of vertices and deal edges the same way: an edge x-y is directed from the out-x vertex to in-y. Then convert the vertex to an edge by adding a 1-capacity arc from the in-vertex to the out-vertex. Now for each two vertices we must solve the sub-problem of minimally separating them. So, just like before take each pair of vertices and treat the out-vertex of one of them as the source and the in-vertex of the other one as the sink (this is because the only arc leaving the in-vertex is the one that goes to the out-vertex) and take the lowest value of the maximum flow. This time we can't improve in the quadratic number of steps needed, because the first vertex may be in an optimum solution and by always considering it as the source we lose such a case.

Maximum Bipartite Matching

This is one of the most important applications of maximum flow, and a lot of problems can be reduced to it. A matching in a graph is a set of edges such that no vertex is touched by more than one edge. Obviously, a matching with a maximum cardinality is a maximum matching. For a general graph, this is a hard problem to deal with.

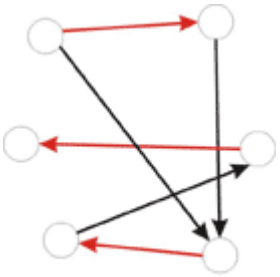


Figure 10

Let's direct our attention towards the case where the graph is bipartite - its vertices can be split into two sets such that there is no edge connecting vertices from the same set. In this case, it may sound like this: "Each of your employees can handle a given set of jobs. Assign a job to as many of them as you can."

A bipartite graph can be built in this case: the first set consists of your employees while the second one contains the jobs to be done. There is an edge from an employee to each of the jobs he could be assigned. An example is given below:

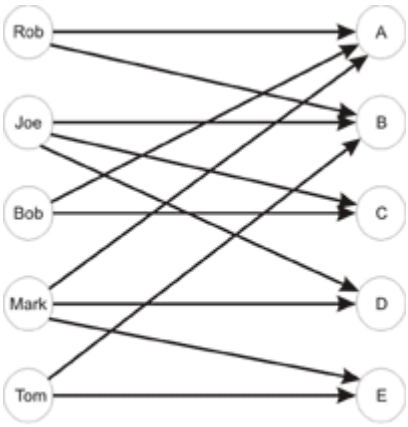


Figure 11

So, Joe can do jobs B, C and D while Mark wouldn't mind being assigned jobs A, D or E. This is a happy case in which each of your employees is assigned a job:

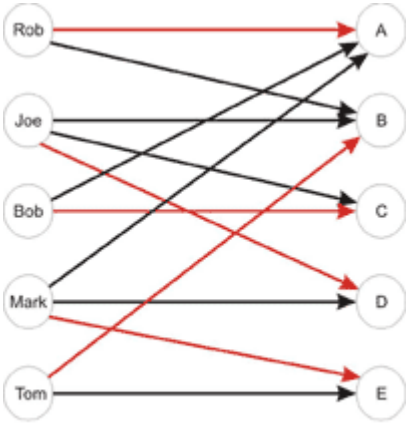


Figure 12

In order to solve the problem we first need to build a flow network. Just as we did in the multiple-source multiple-sink problem we will add two "dummy" vertices: a super-source and a super-sink, and we will draw an edge from the super-source to each of the vertices in set A (employees in the example above) and from each vertex in set B to the super-sink. In the end, each unit of flow will be equivalent to a match between an employee and a job, so each edge will be assigned a capacity of 1. If we would have assigned a capacity larger than 1 to an edge from the super-source, we could have assigned more than one job to an employee. Likewise, if we would have assigned a

capacity larger than 1 to an edge going to the super-sink, we could have assigned the same job to more than one employee. The maximum flow in this network will give us the cardinality of the maximum matching. It is easy to find out whether a vertex in set B is matched with a vertex x in set A as well. We look at each edge connecting x to a vertex in set B, and if the flow is positive along one of them, there exists a match. As for the running time, the number of augmenting paths is limited by $\min(|A|,|B|)$, where by $|X|$ is denoted the cardinality of set X, making the running time $O(N \cdot M)$, where N is the number of vertices, and M the number of edges in the graph.

An implementation point of view is in place. We could implement the maximum bipartite matching just like in the pseudocode given earlier. Usually though, we might want to consider the particularities of the problem before getting to the implementation part, as they can save time or space. In this case, we could drop the 2-dimensional array that stored the residual network and replace it with two one-dimensional arrays: one of them stores the match in set B (or a sentinel value if it doesn't exist) for each element of set A, while the other is the other way around. Also, notice that each augmenting path has capacity 1, as it contributes with just a unit of flow. Each element of set A can be the first (well, the second, after the super-source) in an augmenting path at most once, so we can just iterate through each of them and try to find a match in set B. If an augmenting path exists, we follow it. This might lead to de-matching other elements along the way, but because we are following an augmenting path, no element will eventually remain unmatched in the process.

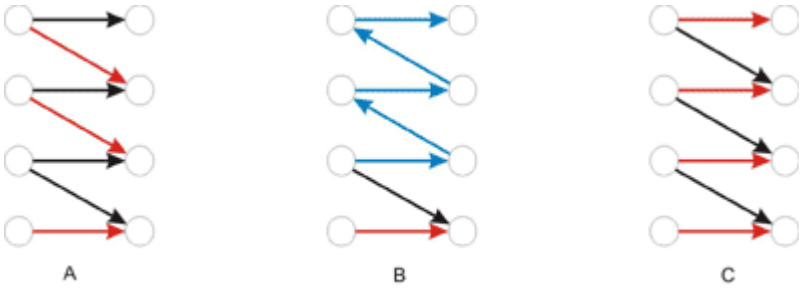


Figure 13 A) before B) an augmenting path C) after

Now let's solve some TopCoder problems!

RookAttack

Problem Statement

This problem asks us to place a maximum number of rooks on a *rows x cols* chessboard with some squares cut out. The idea behind this might be a little hard to spot, but once this is done, we get into a standard maximum bipartite-matching problem.

Notice that at most one rook can be placed on each row or column. In other words, each row corresponds at most to one column where a rook can be placed. This suggests a bipartite matching where set A is composed of elements corresponding to every row of the board, while set B consists of the columns. For each row add edges to every column if the corresponding square is not cut out of the board. Now we can just run maximum bipartite-matching in this network and compute the result. Since there are at most *rows * cols* edges, the time complexity of the algorithm is: $O(rows^2 * cols)$

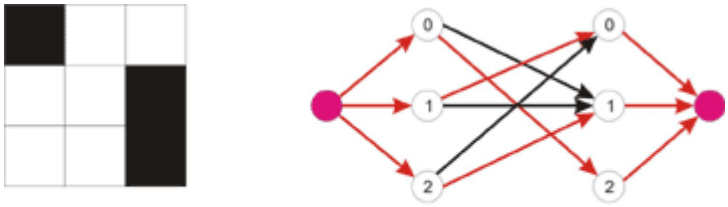


Figure 14 - Example 3 and the corresponding network

In the C++ code below BFS is used for finding an augmenting-path:

```
class RookAttack
 {     // a list of the non-empty squares for each row
   vector lst[300];
```

```
   // in this arrays we keep matches found to every row and column
   int row_match[300], col_match[300];
   // we search for an augmenting path starting with row source
   bool find_match(int source) {
     // from[x] = the row-vertex that precedes x in the path
     int from[300], where, match;
     memset(from, -1, sizeof(from));
     from[source] = source;
     deque q;
     q.push_back(source);
     bool found_path = false;
     while (!found_path && !q.empty()) {
       // where = current row-vertex we are in
       where = q.front(); q.pop_front();
       // we take every uncut square in the current row
       for (int i = 0; i < lst[where].size(); ++ i) {
         match = lst[where][i];
         // next = the row matched with column match
         int next = col_match[match];
         if (where != next) {
           // no row matched with column match thus
we found an augmenting path
           if (next == -1) {
             found_path = true;
             break;
           }
           // a check whether we already visited
the row-vertex next
           if (from[next] == -1) {
             q.push_back(next);
             from[next] = where;
           }
         }
       }
     }
     if (!found_path)
       return false;
     while (from[where] != where) {
       // we de-match where from its current match (aux)
and match it with match
       int aux = row_match[where];
       row_match[where] = match;
       col_match[match] = where;
       where = from[where];
       match = aux;
     }
     // at this point where = source
     row_match[where] = match;
     col_match[match] = where;
     return true;
   }

   public:
   int howMany(int rows, int cols, vector  cutouts)
   {  // build lst from cutouts; column j should appear in
row's i list if square (i, j) is present on the board

      int ret = 0;
      memset(row_match, -1, sizeof(row_match));
      memset(col_match, -1, sizeof(col_match));
      // we try to find a match for each row
      for (int i = 0; i < rows; ++ i)
        ret += find_match(i);
```

```
    return ret;
  }
};
```

Let's take a look at the DFS version, too. We can implement the *find_match* function like this: for each non-empty square in the current row try to match the row with its corresponding column and call *find_match* recursively to attempt to find a new match for the current match (if the current match exists - if not, an augmenting path is found) of this column. If one is found, we can perform the desired match. Note that to make this run in time we must not visit the same column (or row) twice. Notice the C++ code below is extremely short:

```
bool find_match(int where) {
  // the previous column was not matched
  if (where == -1)
    return true;
  for (int i = 0; i < lst[where].size(); ++ i) {
    int match = lst[where][i];
    if (visited[match] == false) {
      visited[match] = true;
      if (find_match(col_match[match])) {
        col_match[match] = where;
        return true;
      }
    }
  }
  return false;
}
```

This runs in time because the number of augmenting paths is the same for both versions. The only difference is that BFS finds the shortest augmenting-path while DFS finds a longer one. As implementation speed is an important factor in TopCoder matches, in this case it would be a good deal to use the slower, but easier DFS version.

The following version of the problem is left as an exercise for the reader: to try and place as many rooks as possible on the board in such a way that the number of rooks on each row is equal to the number of rooks on each column (it is allowed for two rooks to attack each other).

Graduation

Problem Statement

In this problem we are given a set of requirements, each stating that a number of classes should be taken from a given set of classes. Each class may be taken once and fulfills a single requirement. Actually, the last condition is what makes the problem harder, and excludes the idea of a greedy algorithm. We are also given a set of classes already taken. If it weren't for this, to ensure the minimality of the return, the size of the returned string would have been (if a solution existed) the sum of the number of classes for each requirement. Now as many classes as possible must be used from this set.

At first glance, this would have been a typical bipartite-matching problem if every requirement had been fulfilled by taking just a single class. Set A would have consisted of the classes available (all characters with ASCII code in the range 33-126, except for the numeric characters '0'-'9'), while the set of requirements would have played the role of set B. This can be taken care of easily. Each requirement will contribute to set B with a number of elements equal to the number of classes that must be taken in order to fulfill it - in other words, split each requirement into several requirements. At this point, a bipartite-matching algorithm can be used, but care should be allotted to the order in which we iterate through the set of classes and match a class with a requirement.

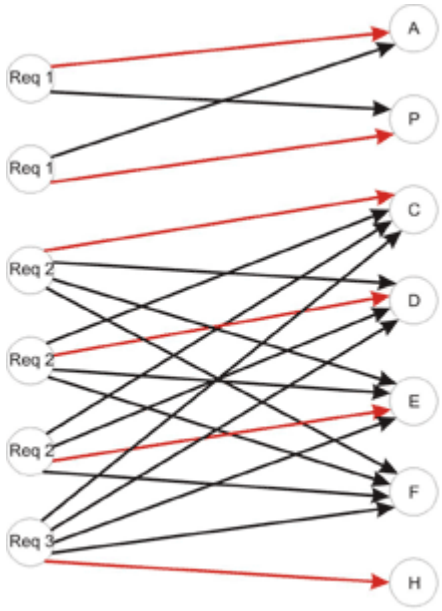


Figure 15 - Example 4 from the problem statement

It is important to understand that any order to iterate through set A can be considered when solving the standard bipartite-matching problem. For example, it doesn't matter what element from set A we choose to be the first one to be matched. Consider the solution found by the algorithm containing this element x from A, matched with an element y from B. Also, we should consider any optimal solution. Clearly, in the optimal, y must be matched with an element z from A, otherwise we can add the pair x-y to the matching, contradicting the fact that the solution is optimal. Then, we can just exchange z with x to come with a solution of the same cardinality, which completes the proof.

That being said, to gain as much as possible from the classes already taken we first must match each of these with a requirement. If, after completing this step, all requirements are fulfilled, we just need to return the empty string, as there is no need for taking more classes. Now we have to deal with the requirement that the return must be the first in lexicographic order. It should be obvious now that the other classes must be considered in increasing order. If a match is found for a class, that class is added to the return value. In the end, if not every requirement is fulfilled, we don't have a solution. The implementation is left as an exercise for the reader.

As a final note, it is possible to speed things up a bit. To achieve this, we will drop the idea of splitting each requirement. Instead we will modify the capacities of the edges connecting those with the super-sink. They will now be equal to the number of classes to be taken for each requirement. Then we can just go on with the same approach as above.

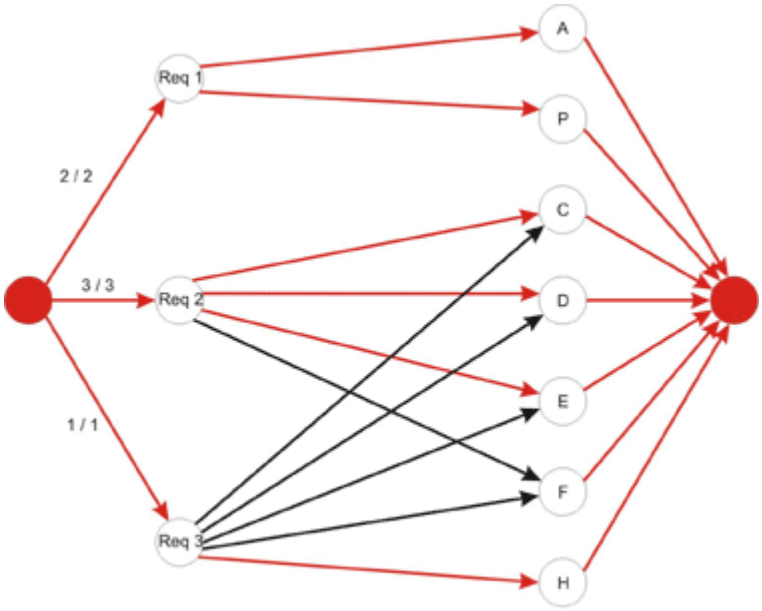


Figure 16 - Example 4 from the problem statement

Parking

Problem Statement

In this problem we have to match each of the cars with a parking spot. Additionally the time it takes for all cars to find a parking spot must be minimized. Once again we build a bipartite graph: set A is the set that consists of the cars and set B contains the parking spots. Each edge connecting elements from different sets has as the cost (and not the capacity!) the time required for the car to get to the parking spot. If the spot is unreachable, we can assign it an infinite cost (or remove it). These costs are determined by running breadth-first search.

For solving it, assume that the expected result is less than or equal to a constant D. Then, there exists a matching in which each edge connecting a car and a parking spot has the cost less than or equal to D. Thus, removing edges with cost greater than D will have no effect on the solution. This suggests a binary search on D, removing all edges with cost greater than D, and then performing a maximum bipartite-matching algorithm. If a matching exists in which every car can drive to a parking spot, we can decrease D otherwise we must increase it.

However, there is a faster and more elegant solution using a priority-first search. Instead of keeping D fixed as above, we could try to successively increase D whenever we find that it is too low. We will start with D = 0. Then we iterate through each of the cars and try to find an augmenting path in which no edge has a cost larger than D. If none exists, we increase D until one path exists. Obviously, we will increase it with the smallest possible amount. In order to achieve this, we will search for the augmenting path with the smallest cost - the cost of the path is the maximum cost of an edge on that path. This can be done with a priority-first search similar to the PFS augmenting-path algorithm presented in the first section of the article. C++ code follows:

```
struct node {
  int where, cost, from;
  node(int _where, int _cost, int _from): where(_where),
cost(_cost), from(_from) {};
};
bool operator < (node a, node b) {
  return a.cost > b.cost;
}

int minTime(vector  park)
{
  // build a cost matrix cost[i][j] = cost of getting from car i to
```

```
parking spot j, by doing a BFS
  // vertices 0, 1, ..., N - 1 will represent the cars, and
vertices N, N + 1, ..., N + M - 1 will represent
//the parking spots; N + M will be the super-sink
  int D = 0, sink = N + M;
  int car_match[105], park_match[105];
  memset(car_match, -1, sizeof(car_match));
  memset(park_match, -1, sizeof(park_match));

  for (int source = 0; source < N; ++ source) {
    bool visited[210];
    memset(visited, false, sizeof(visited));
    int from[210];
    memset(from, -1, sizeof(from));
    priority_queue pq;
    pq.push(node(source, 0, -1));
    while (!pq.empty()) {
      int cst = pq.top().cost, where = pq.top().where,
_from = pq.top().from;
      pq.pop();
      if (visited[where]) continue;
      visited[where] = true;
      from[where] = _from;
      // if where is a car try all M parking spots
      if (where < N) {
        for (int i = 0; i < M; ++ i) {
          // if the edge doesn't exist or this car
is already matched with this parking spot
          if (cost[where][i] == infinity ||
car_match[where] == i) continue;
          int ncst = max(cst, cost[where][i]);
          // the i-th parking spot is N + i
          pq.push(node(N + i, ncst, where));
        }
      }
      else {
        // if this parking spot is unmatched we found
the augmenting path with minimum cost
        if (park_match[where - N] == -1) {
          from[sink] = where;
          // if D needs to be increased, increase it
          D = max(D, cst);
          break;
        }
        // otherwise we follow the backward edge
        int next = park_match[where - N];
        int ncst = max(cst, cost[next][where]);
        pq.push(node(next, ncst, where));
      }
    }

    int where = from[sink];
    // if no augmenting path is found we have no solution
    if (where == -1)
      return -1;
    // follow the augmenting path
    while (from[where] > -1) {
      int prev = from[where];
      // if where is a parking spot the edge (prev, where)
is a forward edge and the match must be updated
      if (where >= N) {
        car_match[prev] = where;
        park_match[where - N] = prev;
```

```
      }
      where = prev;
    }
  }

  return D;
}
```

Here are some problems to practice:
PlayingCubes - for this one ignore the low constraints and try to find a max-flow algorithm
DataFilter - be warned this is the hard problem from the TCCC 2004 Finals and is tough indeed!
Some other problems from http://acm.uva.es/p/: 563, 753, 820, 10122, 10330, 10511, 10735.

For any questions or comments please use the Forums.