Bipartite matching

A *bipartit*e (or *bicolourable*) graph is a graph G=(V,E) in which it is possible to split the set of vertices, V, into two non-empty sets – L and R – such that all of the edges in E have one endpoint in L and the other one in R. A *matchin*g in any graph is any subset, S, of the edges chosen in such a way that no two edges in S share an endpoint. A *maximu*m *matchin*g is a matching of maximum size. The problem of finding a maximum matching in a given graph is NP-hard in general, but if the graph is bipartite, there is a neat polynomial time solution. First of all, why do we care about bipartite matching? Here is a classic problem. There are m job applicants and n job openings. Each applicant has a subset of the job openings she is interested in. Conversely, each job opening can only accept one applicant out of some subset of the applicants. In other words, there are certain allowed or "compatible" pairings of applicants to jobs. Find an assignment of jobs to applicants in such a way all the pairings are allowed and as many applicants as possible get jobs.

We can model this with a bipartite graph – on the left side we have the applicants and on the right side we have the jobs. Draw an edge between applicant u and job v if they are compatible. The answer that we are looking for is the maximum matching.

One way to solve the problem is to reduce it to an instance of Maximum Flow. Draw the bipartite graph with applicants on the left and jobs on the right. Give each edge a capacity of 1 – having a flow of 1 between applicant u and job v will correspond to matching applicant u to job v. Now we need to ensure that no applicant gets matched to more than 1 job. For that, add a vertex, s, on the far left and connect it to each applicant by an edge of capacity 1. We also want each job to only be filled by at most one applicant. To enforce that, add a vertex, t, on the far right and connect each job opening to t by an edge of capacity 1. Finally, find the maximum flow from s to t. The amount of flow we can push is exactly the number of original edges that will be used to connect an applicant to a job. Furthermore, no applicant or job vertex will be used more than once. To get the optimal matching, read it off from the resulting flow network.

The implementation of this algorithm is straightforward once we have the code for MaxFlow. However, since each edge has capacity 1, we can simplify the code a lot. Here is an implementation that uses DFS to find augmenting paths in the Ford-Fulkerson algorithm and represents the graph and the flow network implicitly.

Example **1: Maximu**m **Bipartite Matchin**g

```
#include <string.h>

#define M 128
#define N 128

bool graph[M][N];
bool seen[N];
int matchL[M], matchR[N];
int n, m;

bool bpm( int u )
{
   for( int v = 0; v < n; v++ ) if( graph[u][v] )
   {
      if( seen[v] ) continue;
      seen[v] = true;

      if( matchR[v] < 0 || bpm( matchR[v] ) )
```

```
      {
         matchL[u] = v;
         matchR[v] = u;
         return true;
      }
   }
   return false;
}

int main()
{
   // Read input and populate graph[][]
   // Set m, n

   memset( matchL, -1, sizeof( matchL ) );
   memset( matchR, -1, sizeof( matchR ) );
   int cnt = 0;
   for( int i = 0; i < m; i++ )
   {
      memset( seen, 0, sizeof( seen ) );
      if( bpm( i ) ) cnt++;
   }

   // cnt contains the number of happy pigeons
   // matchL[i] contains the hole of pigeon i or -1 if pigeon i is unhappy
   // matchR[j] contains the pigeon in hole j or -1 if hole j is empty

   return 0;
}
```

Most of the work is done by the bpm(u) function that tries to match the left vertex u to something on the right side. It does that by trying all right vertices v and assigning u to v if either v is unassigned, or if v's match on the left side can be reassigned to some other vertex on the right that is larger than v. If you try running this algorithm on an example, you will see that this is simply another way of implementing DFS. Each call to bpm(u) finds an augmenting path starting at u. The path is allowed to use left-to-right edges that are still unused, as well as right-to-left edges that are used (by undoing a unit of flow). bpm(u) returns true if a path was found and false otherwise. Each augmenting path adds one more edge to the matching, and these are counted by the 'cnt' variable in main(). The seen[] array is the usual DFS seen [] array. matchL[] and matchR[] store the matching; a -1 means "vertex is not matched (yet)".

Can you determine the worst-case running time of this algorithm? (Typing the link below in the browser and reading it there would be cheating.) Note that the bipartite matching problem is symmetric. We can flip the left and right sides of the graph and assign jobs to applicants instead of applicants to jobs – it's the same thing. Using this fact, can you improve the worst-case running time? How could you do that with the fewest changes to the code?

This code, with a lot more comments can be found here:
http://www.shygypsy.com/tools/bpm.cpp

References

Cormen, Thomas H., et al. Introduction to Algorithms. , 2nd ed. Cambridge: The MIT press, 2002.
Frank & Igor's notes (of course!)