Code Jam 2008 - Round 3

Analysis: No Cheating

Programming contests have certainly changed over the years. Nowadays, dynamic programming is a trivial matter to many, and bipartite matching is no longer a secret idea. More than half of the contestants solved the small datasets of this problem, and 77 passed the large tests.

Small dataset (Test set 1 - Visible)

The small tests can be solved using dynamic programming. The main idea is to do this row by row.

The naive dynamic programming has states (R1, r), where r is the row number and R1 is the possible configurations of the current row. In order to compute the value of (R1, r), (the maximum number of students who can be put in the first r rows, with the r-th row identical to R1), one can loop through all of the possible values of R2 and use the value of (R2, r-1).

A more sophisticated approach also goes row by row, but in each row, we do it by squares. The states are (S, r, c), where (r, c) is the current position, and S is the bitmap for up to n positions before (r, c). These are all the squares to the left on the same row, and all the squares from the (c-1)-st to the last on the last row. In this way, each value of (S, r, c) can be computed in constant time.

One may estimate the number of states to be $O(M\ N\ 2^N)$. But actually the number of states is much smaller. It is not hard to see, with the restriction that no two horizontally adjacent seats can be both occupied, that the number of states S is $O(F_N)$, the N-th Fibonacci number. This means that the dynamic programming solution can even handle a classroom of size about 35 by 35.

Large dataset (Test set 2 - Hidden)

Well, but universities do have very large classrooms. The large dataset has tests with 80 rows and 80 columns.

The key idea is to see through the puzzle and notice that this is a standard graph problem. Let each square be a node. What the problem requires is to pick a set of nodes, such that some pairs of nodes cannot be picked together. Let us draw an edge between each such pair of nodes. This is the well-known problem of finding a maximum independent set of the graph. While the general independent set problem is NP-hard, we do know that the problem is tractable on bipartite graphs. A-ha! The graph is indeed bipartite, because all of the edges are between a square in odd-numbered columns and squares in even-numbered columns.

We can easily find the maximum matching on bipartite graphs. Let's see how a maximum matching can help us solve the problem. Let's assume that the size of the matching is *X*, and the number of nodes in the graph is *N*. Notice that the the maximum independent set cannot contain more than *N* - *X* nodes, because that would mean you will have two nodes that share an edge of the maximum matching in this set.

In general, we have the following relations between the important graph parameters. For a graph G = (V, E), let α(G) be the size of the maximum independent set, m(G) be the size of a maximum matching, μ(G) be the size of a minimum vertex cover,

- The complement of an independent set is a vertex cover, and vice versa. So α(G) + μ(G) = |V|.
- To cover all the edges, we need one vertex from any edge of a matching. So μ(G) ≥ m(G).
- If G is bipartite, then μ(G) = m(G). Equivalently, α(G) + m(G) = |V|.

The last fact can be proved directly, or as a consequence of Hall's theorem, or as an application of the powerful max-flow-min-cut theorem. Below we provide a constructive proof that suits to our problem.

Proof

We will prove that *N* - *X* is smaller than or equal to the size of the maximum independent set. Let's have a maximum matching *M* and a set *S* which contains all the nodes that are not in *M*. Now let's add to *S* all the nodes from *M* which are on the left side of the bipartite graph. *S* is of size *N* - *X*. If *S* is not an independent set, then it has a node *u* from *M* and another one *v* that doesn't belong to *M*, such that (*u*, *v*) is an edge. To fix this problem we remove *u* from *S* and add node *u'* to *S* where (*u*, *u'*) is an edge from the matching. If this operation caused a problem with another node in the matching, then we can remove that node and add it's match, and so on. This repeated procedure is guaranteed to finish since we only delete nodes from the left side of the graph and add nodes from the right side. There will be no problem caused by the nodes on the right side of the graph and nodes outside the matching because that would mean that we can find an alternating path, and the maximum matching is not the largest possible.

We have proved that in any bipartite graph, the size of the maximum independent set equals the size of the graph minus the size of the maximum matching.

Code

We believe that many contestants have the standard bipartite matching algorithm prepared. You may download many nice samples from the scoreboard. Below we show a sample solution which keeps the bipartite graph only in mind, and actually runs the matching algorithm on the board itself. From the judges:

```
int C, N, M;
string bd[100];

int nbx[100][100], nby[100][100], v[100][100];
int T=0;

bool dfs(int a, int b) {
  if (a<0) return true;
  if(v[a][b]==T) return false;
  v[a][b]=T;
  for (int i=a-1;i<=a+1;i++)
    for (int j=b-1;j<=b+1;j+=2)
      if (i>=0 && i<M && j>=0 && j<N && bd[i][j]=='.') {
        if (dfs(nbx[i][j], nby[i][j])) {
          nbx[i][j]=a; nby[i][j]=b;
          nbx[a][b]=i; nby[a][b]=j;
          return true;
        }
      }
  return false;
}

int play() {
  memset(nbx,-1,sizeof(nbx));
  memset(nby,-1,sizeof(nby));
  memset(v, -1, sizeof(v)); T=-1;
  int rst=0;
  for(int i=0;i<M;i++) for(int j=0;j<N;j++) {
    if (bd[i][j]=='.') {
      rst++;
      if (j%2) {
        T++;
        if (dfs(i,j)) rst--;
      }
    }
  }
  return rst;
}

int main() {
  cin>>C;
  for (int i=1; i<=C; i++) {
    cin>>M>>N;
    for (int r=0;r<M;r++) cin>>bd[r];
    cout<<"Case #"<<i<<": "<<play()<<endl;
  }
  return 0;
}
```

More Information

Maximum independent set - Bipartite Matching