USACO JAN15 Problem 'grass' Analysis

by Mark Gordon

This problem asks us to find the largest cycle in a directed graph going through node 1 where one of the edges may be traversed backwards. Suppose the optimal solution follows the edge v->u backwards, instead going from u to v. Then it must be the case that node 1 can reach node u and node v can reach node 1.

This immediately suggests an algorithm; loop over all edges v->u that satisfy this property and quickly compute the number of nodes that can be reached along a path from v to 1 to u. Doing this directly on the graph could appear a bit difficult given that Bessie may need to return to the same node several times to visit the maximum number of distinct nodes. However, after finding the Strongly Connected Components of the graph this problem becomes tractable. Tarjan's algorithm, among others, is one algorithm that can be used to do this in $O(V+E)$ time.

Once we compute the Strongly Connected Components (SCCs) of the graph we can create a new Directed Acyclic Graph (DAG) where each node is weighted by the number of nodes in the corresponding strongly connected component. Now we can use dynamic programming to compute the maximum path weight of any path going from an SCC ending in the SCC containing node 1. Because the graph is acyclic this recurrence relation is well defined. We can do the same thing to find the maximum path weight from the SCC containing node 1 to any other SCC.

After calculating each of these values we simply need to iterate over all edges in the SCC graph and find the one that maximizes the sum of these two paths (making sure to only count the nodes in the SCC containing 1 once). Additionally, we should make sure to handle the edge cases where the SCC containing 1 cannot reach any other SCCs or cannot be reach by any other SCCs or both.

Here's my solution that implements these ideas using Tarjan's algorithm for SCC computation.

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include <cstring>
#include <cstdio>

#define MAXN 100010

using namespace std;

/* Given a graph (represented by the adjacency list E) and a labelling of nodes
 * into components, returns a new graph (as an adjacency list) where there is an
 * edge between components if there was an edge that crossed the components in
 * the original graph.  Useful after computing SCCs to obtain the SCC DAG.
 * ~O(V + E) */
vector<vector<int> > collapse(const vector<vector<int> >& E,
                              const vector<int>& L) {
  int N = E.size();
  vector<vector<int> > R(*max_element(L.begin(), L.end()) + 1, vector<int>());
  for(int i = 0; i < N; i++) {
    int x = L[i];
    for(int j = 0; j < E[i].size(); j++) {
      int y = L[E[i][j]];
      if(x != y) R[x].push_back(y);
    }
  }
  for(int i = 0; i < R.size(); i++) {
    sort(R[i].begin(), R[i].end());
    R[i].resize(unique(R[i].begin(), R[i].end()) - R[i].begin());
  }
  return R;
}

int scc_nidx;
int scc_idx[MAXN];
int scc_lnk[MAXN];

int scc_lbl;
int scc_sp;
int scc_stk[MAXN];

/* Helper function used by scc implementing Tarjan's SCC algorithm */
void scc_dfs(int u, const vector<vector<int> >& E, vector<int>& L) {
  int idx = scc_idx[u] = scc_nidx++;
  int& lnk = scc_lnk[u];
  lnk = idx;

  scc_stk[scc_sp++] = u;
  for(int i = 0; i < E[u].size(); i++) {
    int v = E[u][i];
    if(scc_idx[v] == -1) {
      scc_dfs(v, E, L);
      lnk = min(lnk, scc_lnk[v]);
    } else {
      lnk = min(lnk, scc_idx[v]);
    }
  }

  if(idx == lnk) {
    for(int y = -1; y != u; ) {
      y = scc_stk[--scc_sp];
      scc_idx[y] = E.size();
      L[y] = scc_lbl;
    }
    scc_lbl++;
  }
}

/* Return a labelling of the M SCCs.
 * The ith vertex is part of the L[i]-th SCC.  O(V + E) */
vector<int> scc(const vector<vector<int> >& E) {
  int N = E.size();
  vector<int> L(N, -1);

  memset(scc_idx, -1, sizeof(int) * N);
  scc_nidx = scc_lbl = scc_sp = 0;
  for(int i = 0; i < N; i++) {
    if(L[i] == -1) {
      scc_dfs(i, E, L);
    }
  }
  return L;
}

int memo[2][100010];

/* Computes the maximum weight path from s to t.
 * Doesn't include the weight of node t. Returns -2 if s cannot reach t. */
int solve(int cid, const vector<int>& W,
          const vector<vector<int> >& E, int s, int t) {
  if (s == t)
    return 0;

  int& ref = memo[cid][s];
  if (ref != -1)
    return ref;

  ref = -2;
  for (int i = 0; i < E[s].size(); i++) {
    int res = solve(cid, W, E, E[s][i], t);
    if (res >= 0)
      ref = max(ref, W[s] + res);
  }
  return ref;
}

int main() {
  int N, M;
  cin >> N >> M;

  vector<vector<int> > E(N);
  for (int i = 0; i < M; i++) {
    int u, v;
    cin >> u >> v;
    u--; v--;
    E[u].push_back(v);
  }

  /* Compute the SCC graph and its reverse */
  vector<int> L = scc(E);
  E = collapse(E, L);
  int st = L[0];

  vector<vector<int> > RE(E.size());
  for (int i = 0; i < E.size(); i++)
    for (int j = 0; j < E[i].size(); j++)
      RE[E[i][j]].push_back(i);

  /* Compute the weight of each node in the SCC graph. */
  vector<int> W(E.size(), 0);
  for (int i = 0; i < N; i++)
    W[L[i]]++;

  /* Try each edge in the SCC graph. */
  int result = W[st];
  memset(memo, -1, sizeof(memo));
  for (int i = 0; i < E.size(); i++) {
    /* Compute the max path weight from node i to st. */
    int r1 = solve(0, W, E, i, st);
    if (r1 < 0)
      continue;
    for (int j = 0; j < E[i].size(); j++) {
      /* Compute the max path weight from node st to E[i][j]. */
      int r2 = solve(1, W, RE, E[i][j], st);
      if (r2 >= 0)
        result = max(result, W[st] + r1 + r2);
    }
  }

  cout << result << endl;
}
```