Pontificia Universidad Católica del Perú - FCI

XieXieLucas Notebook - Froz/Phibrain/Ands

November 9, 2017

Contents

1 Bipartite Matching

```
// O(V*E) maximum bipartite matching
int p[MAX];          // Parent of right-node v in the matching
int vis[MAX]; // Whether left-node u has been visited
vi adj[MAX];  // Standard adjacency list

int match(int u) {
	if (vis[u]) return 0;
	vis[u] = 1;
	REP(i,0,adj[u].size()){
		int v = adj[u][i];
		if (p[v] == -1 || match(p[v])) {
			p[v] = u;
			return 1;
		}
	}
   return 0;
}
```

```
int main(){
	// build adj here with n left nodes
    // and V total nodes
	int n = 1000000;
    int V = 2000000;
	int maxMatch = 0;
	REP(i,0,V) p[i] = -1;
	REP(u,0,n){
		REP(i,0,n) vis[i] = 0;
		maxMatch += match(u);
	}
	printf("Found %d matchings\n", maxMatch)
}
```

2 Dinic Flow

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source and sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain actual flow values, look at edges with capacity > 0
//       (zero capacity edges are residual edges).
```

```
typedef long long LL;

struct Edge {
  int u, v;
  LL cap, flow;
  Edge() {}
  Edge(int u, int v, LL cap): u(u), v(v), cap(cap), flow(0) {}
};

struct Dinic {
  int N;
  vector<Edge> E;
  vector<vector<int>> g;
  vector<int> d, pt;

  Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

  void AddEdge(int u, int v, LL cap) {
    if (u != v) {
      E.emplace_back(Edge(u, v, cap));
      g[u].emplace_back(E.size() - 1);
      E.emplace_back(Edge(v, u, 0));
      g[v].emplace_back(E.size() - 1);
    }
  }

  bool BFS(int S, int T) {
    queue<int> q({S});
    fill(d.begin(), d.end(), N + 1);
    d[S] = 0;
    while(!q.empty()) {
      int u = q.front(); q.pop();
      if (u == T) break;
      for (int k: g[u]) {
        Edge &e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
          d[e.v] = d[e.u] + 1;
          q.emplace(e.v);
        }
      }
    }
    return d[T] != N + 1;
  }

  LL DFS(int u, int T, LL flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int &i = pt[u]; i < g[u].size(); ++i) {
      Edge &e = E[g[u][i]];
      Edge &oe = E[g[u][i]^1];
      if (d[e.v] == d[e.u] + 1) {
        LL amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (LL pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }

  LL MaxFlow(int S, int T) {
    LL total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (LL flow = DFS(S, T))
        total += flow;
    }
    return total;
  }
};
```

3 Edmonds Blossom

```
// Maximum general matching (not necessarily bipartite)
// Make sure to set N in main()
// Claimed O(N^4) running time

int N; // the number of vertices in the graph
typedef vector<int> vi;
typedef vector< vector<int> > vvi;
vi match;
vi vis;

void couple(int n, int m) { match[n]=m; match[m]=n; }
```

```
// True if augmenting path or a blossom (if blossom is non-empty).
// the dfs returns true from the moment the stem of the flower is
// reached and thus the base of the blossom is an unmatched node.
// blossom should be empty when dfs is called and
// contains the nodes of the blossom when a blossom is found.
bool dfs(int n, vvi &conn, vi &blossom) {
  vis[n]=0;
  REP(i, 0, N) if(conn[n][i]) {
    if(vis[i]==-1) {
      vis[i]=1;
      if(match[i]==-1 || dfs(match[i], conn, blossom)) {
                  couple(n,i);
                  return true;
          }
    }
    if(vis[i]==0 || SZ(blossom)) { // found flower
      blossom.pb(i); blossom.pb(n);
      if(n==blossom[0]) { match[n]=-1; return true; }
      return false;
    }
  }
  return false;
}

// search for an augmenting path.
// if a blossom is found build a new graph (newconn) where the
// (free) blossom is shrunken to a single node and recurse.
// if a augmenting path is found it has already been augmented
// except if the augmented path ended on the shrunken blossom.
// in this case the matching should be updated along the
// appropriate direction of the blossom.
bool augment(vvi &conn) {
      REP(m, 0, N) if(match[m]==-1) {
            vi blossom;
            vis=vi(N,-1);
            if(!dfs(m, conn, blossom)) continue;
            if(SZ(blossom)==0) return true; // augmenting path found

// blossom is found so build shrunken graph
            int base=blossom[0], S=SZ(blossom);
            vvi newconn=conn;
            REP(i, 1, S-1) REP(j, 0, N)
                  newconn[base][j]=newconn[j][base]|=conn[blossom[i]][
                     j];
            REP(i, 1, S-1) REP(j, 0, N)
                  newconn[blossom[i]][j]=newconn[j][blossom[i]]=0;
            newconn[base][base]=0; // is now the new graph
            if(!augment(newconn)) return false;
            int n=match[base];

// if n!=-1 the augmenting path ended on this blossom
    if(n!=-1) REP(i, 0, S) if(conn[blossom[i]][n]) {
      couple(blossom[i], n);
      if(i&1) for(int j=i+1; j<S; j+=2)
                  couple(blossom[j],blossom[j+1]);
      else for(int j=0; j<i; j+=2)
                  couple(blossom[j],blossom[j+1]);
      break;
    }
    return true;
  }
  return false;
}

// conn is the NxN adjacency matrix
// returns the number of edges in a max matching.
int edmonds(vvi &conn) {
  int res=0;
  match=vi(N,-1);
  while(augment(conn)) res++;
  return res;
}

/****************************************************/
set<pair<int,int> > used;
int main(){
  int n;
  cin >> n;
  N = n;
  vvi conn;
  vi tmp;
  tmp.assign(n,0);
  REP(i, 0, n) conn.push_back(tmp);
  int u, v;
  while(cin >> u >> v){
    u--; v--;
    if(u > v) swap(u,v);
    if(used.count(make_pair(u,v))) continue;
    used.insert(make_pair(u,v));
    conn[u][v] = conn[v][u] = 1;
```

```
  }
  int res = edmonds(conn);
  cout<<res*2<<endl;
  REP(i, 0, n) {
    if(match[i] > i){
      cout<<i+1<<" "<<match[i] + 1<<endl;
    }
  }
  return 0;
}
```

4 Min Cost Max Flow

```
const int MAXN = 5010;

const ll INF = 1e15;
struct edge { int dest;ll origcap, cap; ll cost; int rev; };

struct MinCostMaxFlow {

    vector<edge> adj[MAXN];
    ll dis[MAXN], cost;
    int source, target, iter;
    ll cap;
    edge* pre[MAXN];
    int queued[MAXN];
    MinCostMaxFlow (){}
    void AddEdge(int from, int to, ll cap, ll cost) {
        adj[from].push_back(edge {to, cap, cap, cost, (int)adj[to].size()})
            ;
        adj[to].push_back(edge {from,0, 0, -cost, (int)adj[from].size()
             - 1});
    }

    bool spfa() {
        REP(i,0,MAXN) queued[i] = 0;
        fill(dis, dis + MAXN, INF);
        queue<int> q;
        pre[source] = pre[target] = 0;
        dis[source] = 0;
        q.emplace(source);
        queued[source] = 1;
        while (!q.empty()) {
            int x = q.front();

            ll d = dis[x];
            q.pop();
            queued[x] = 0;
            for (auto& e : adj[x]) {
                int y = e.dest;
                ll w = d + e.cost;
                if (e.cap < 1 || dis[y] <= w) continue;
                dis[y] = w;
                pre[y] = &e;
                if(!queued[y]){
                        q.push(y);
                        queued[y] = 1;
                }
            }
        }
        edge* e = pre[target];

        if (!e) return 0;
        while (e) {
            edge& rev = adj[e->dest][e->rev];
            e->cap -= cap;
            rev.cap += cap;
            cost += cap * e->cost;
            e = pre[rev.dest];
        }
        return 1;
    }

    pair<ll,ll> GetMaxFlow(int S, int T) {
        cap = 1, source = S, target = T, cost = 0;
        while(spfa()) {}
        ll totflow = 0;
        for(auto e: adj[source]){
            totflow += (e.origcap - e.cap);
        }
        return make_pair(totflow, cost);
    }
};
```

5 Push Relabel Max Flow

```
// Fast O(|V|^3) flow, works for n ~ 5000 with no problem
// Actual flow values in edges with cap > 0 (0 cap = residual)

typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel {
  int N;
  vector<vector<Edge> > G;
  vector<LL> excess;
  vector<int> dist, active, count;
  queue<int> Q;

  PushRelabel(int N) :
      N(N), G(N), excess(N), dist(N), active(N), count(2*N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

      void Enqueue(int v) {
              if (!active[v] && excess[v] > 0) {
                      active[v] = true; Q.push(v);
              }
      }

  void Push(Edge &e) {
    int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
    if (dist[e.from] <= dist[e.to] || amt == 0) return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
  }

  void Gap(int k) {
    for (int v = 0; v < N; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--;
      dist[v] = max(dist[v], N+1);
      count[dist[v]]++;
      Enqueue(v);
    }
  }

  void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2*N;
    for (int i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
        dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    Enqueue(v);
  }

      void Discharge(int v) {
      for (int i = 0; excess[v] > 0 && i < G[v].size(); i++)
              Push(G[v][i]);
      if (excess[v] > 0) {
                      if (count[dist[v]] == 1) Gap(dist[v]);
                      else Relabel(v);
              }
      }

  LL GetMaxFlow(int s, int t) {
    count[0] = N-1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
      excess[s] += G[s][i].cap;
      Push(G[s][i]);
    }

    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
      active[v] = false;
      Discharge(v);
```

```
    }

    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
  }
};
```