Team Note of PetrSU QA

Compiled on November 26, 2018

Contents

1 Graph

1.1 Dinic

Description: Almost linear in practice. $\mathcal{O}(m\sqrt{n})$ on unit network.
Time Complexity: $\mathcal{O}(n^2m)$

```
const int MAXE = 1e5, MAXV = 1e5;
const ll INF_FLOW = INF;

int edgeTo[MAXE], nextEdge[MAXE], E, edgeCap[MAXE], edgeCost[MAXE];
int firstEdge[MAXV], firstEdgeTmp[MAXV], S, T;
int myQueue[MAXN], qHead, qTail, vertexLevel[MAXV];

void addEdge(int from, int to, ll cap, ll cs) {
  edgeTo[E] = to,  nextEdge[E] = firstEdge[from];
  edgeCap[E] = cap, edgeCost[E] = cs, firstEdge[from] = E++;
  edgeTo[E] = from, nextEdge[E] = firstEdge[to];
  edgeCap[E] = 0, edgeCost[E] = -cs, firstEdge[to] = E++;
}

void init() { E = 0; fill(firstEdge, firstEdge + MAXV, -1); }

bool buildLevelGraph() {
  qTail = qHead = 0;
  fill(vertexLevel, vertexLevel + MAXV, MAXV + 1);
  myQueue[qHead++] = S, vertexLevel[S] = 0;
  while(qTail != qHead) {
    int v = myQueue[qTail++];
    for(int id = firstEdge[v]; id != -1; id = nextEdge[id]) {
      int to = edgeTo[id];
      if(edgeCap[id] && vertexLevel[to] > vertexLevel[v] + 1) {
        vertexLevel[to] = vertexLevel[v] + 1, myQueue[qHead++] = to;
      }
    }
  }
  return vertexLevel[T] != MAXV + 1;
}

ll getBlockingFlow(int v, ll curFlow) {
  if(v == T || !curFlow) return curFlow;
  for(int &id = firstEdgeTmp[v]; id != -1; id = nextEdge[id]) {
```

```
    int to = edgeTo[id];
    if(vertexLevel[to] != vertexLevel[v] + 1 || !edgeCap[id])
    continue;
    ll newFlow = getBlockingFlow(to, min(edgeCap[id], curFlow));
    if(newFlow) {
      edgeCap[id] -= newFlow, edgeCap[id ^ 1] += newFlow;
      return newFlow;
    }
  }
  return 0;
}
ll maxFlow() {
  ll flow = 0, add = 0;
  while(buildLevelGraph()) {
    copy(firstEdge, firstEdge + MAXV, firstEdgeTmp);
    while((add = getBlockingFlow(S, INF_FLOW)))
      flow += add;
  }
  return flow;
}
```

1.2 Mincost

Description: Complexity is strange but in practice works nice.
Time Complexity: $\mathcal{O}(\textit{something big, never reached in ACM tasks})$

```
const int MAXN = 4e5, INF = 1e9;
int gg[111][111], fl[111];
int n, m, S, T, E;
int head[MAXN], to[MAXN], cap[MAXN], nxt[MAXN], cost[MAXN];
int was[MAXN], dd[MAXN], pp[MAXN], qh, qt, qq[MAXN];

void addEdge(int a, int b, int cp, int cs) {
    to[E] = b, cap[E] = cp, cost[E] = cs;
    nxt[E] = head[a], head[a] = E++;
    to[E] = a, cap[E] = 0, cost[E] = -cs;
    nxt[E] = head[b], head[b] = E++;
}

bool SPFA() {
    fill(was, was + MAXN, 0);
    fill(dd, dd + MAXN, INF);
    was[S] = 1, dd[S] = 0, qh = qt = 0, qq[qt++] = S;
    while(qh != qt) {
        int v = qq[qh++];
        if(qh == MAXN) qh = 0;
        was[v] = 0;
        for(int id = head[v]; id != -1; id = nxt[id]) {
            int nv = to[id];
            if(cap[id] > 0 && dd[nv] > dd[v] + cost[id]) {
                dd[nv] = dd[v] + cost[id];
                if(!was[nv]) {
                    was[nv] = 1, qq[qt++] = nv;
                    if(qt == MAXN) qt = 0;
                }
                pp[nv] = id;
            }
        }
    }
    return dd[T] != INF;
}

pair < int, int > mincost() {
    int flow = 0, cost_flow = 0;
    while(SPFA()) {
        int add = INF, add_cost = 0;
        for(int i = T; i != S; i = to[pp[i] ^ 1]) {
            add_cost += cost[pp[i]];
            add = min(add, cap[pp[i]]);
        }
        flow += add;
        cost_flow += add * add_cost;
        for(int i = T; i != S; i = to[pp[i] ^ 1]) {
            cap[pp[i]] -= add;
            cap[pp[i] ^ 1] += add;
        }
    }
    return { flow, cost_flow };
}
```

1.3 Bridges and cut points

Description: Works with multi edges but adds extra $\log n$.
Time Complexity: $\mathcal{O}(n \log n)$

```
void dfs(int v, int p = -1) {
  used[v] = 1;
  tin[v] = fup[v] = timer++;
  int ch = 0;
  for(auto to : gg[v]) {
    if(to == p) continue;
    if(used[to]) fup[v] = min(fup[v],tin[to]);
    else {
      dfs(to, v);
      fup[v] = min(fup[v], fup[to]);
      if(tin[v] < fup[to] && cnt[{min(to, v), max(to, v)}] < 2)
        bridges.pb(id[{min(to, v), max(to, v)}]);
      if(p != -1 && tin[v] <= fup[to]) cutPoints.insert(v);
      ch++;
    }
  }
  if(p == -1 && ch > 1) cutPoints.insert(v);
}
```

1.4 LCA with binary lifting

Description: Need to rind dfs and precalc binary liftings.
Time Complexity: $\mathcal{O}(\log n)$

```
int lca(int u, int v) {
  if(depth[u] > depth[v]) swap(u, v);
  for(int i = LOG2MAXN - 1; i >= 0; i--)
    if(depth[v] - depth[u] >= (1 << i))
      v = dp[i][v];
  if(u == v) return u;
  for(int i = LOG2MAXN - 1; i >= 0; i--)
    if(dp[i][v] != dp[i][u]) {
      u = dp[i][u];
      v = dp[i][v];
    }
  return dp[0][u];
}
```

1.5 Kuhn with greedy heuristic

Description: Supposed to run faster than usual Kuhn.
Time Complexity: $\mathcal{O}(n^3)$

```
bool try_kuhn(int v) {
  used[v] = 1;
  for(auto to : gg[v])
    if(mt[to] == -1 || (mt[to] != -1 && !used[mt[to]] &&
    try_kuhn(mt[to]))) {
      mt[to] = v;
      rmt[v] = to;
      return 1;
    }
  return 0;
}

void solve() {
  memset(mt, -1, sizeof(mt));
  memset(rmt, -1, sizeof(rmt));
  while(1) {
    bool any = 0;
    memset(used, 0, sizeof(used));
    for(int i = 0; i < n; i++)
      if(rmt[i] == -1)
        any |= try_kuhn(i);
    if(!any) break;
  }
  vpii ans;
  for(int i = 0; i < n; i++) if(rmt[i] != -1) ans.pb({i + 1, rmt[i]
  + 1});
  cout << sz(ans) << endl;
  for(auto x : ans) cout << x.fi << ' ' << x.se << endl;
}
```

2 Data Structure

2.1 Polynomial hashes

Description: Almost unbreakable.
Time Complexity: $\mathcal{O}(n), \mathcal{O}(1)$

```
// deg[] = {1, P, P^2, P^3, ...}
// h[] = {0, s[0], s[0]*P + s[1], s[0]*P^2 + s[1]*P + s[2], ...}
const int MOD = (int)(1e9 + 7);
int h[MAXN], p[MAXN], P = max(239, (int)rnd());

void gen_hash(string s) {
  h[0] = 0, p[0] = 1;
  int n = sz(s);
  for(int i = 0; i < n; i++) {
    h[i + 1] = (h[i] * 1LL * P + s[i]) % MOD;
    p[i + 1] = (p[i] * 1LL * P) % MOD;
  }
}

int get_hash(int l, int r) {
  return (h[r + 1] - (h[l] * 1LL * p[r - l + 1]) % MOD + MOD) % MOD;
}
```

2.2 Fenwick

Description: Considered to work in constant time in practice.
Time Complexity: $\mathcal{O}(\log n), \mathcal{O}(\log n)$

```
// structure for maintaining Fenwick Tree
struct BIT {
  vll dataMul, dataAdd;
  int size, mxlog;

  // initialize BIT and calculate mxlog for binsearch
  void init(int nn) {
    mxlog = 0;
    while((1 << mxlog) < nn) mxlog++;
    size = (1 << mxlog) + 1;
    dataMul.resize(size);
    dataAdd.resize(size);
  }

  // add linear function (mul * x + add) to [pos]
  void add(int pos, ll mul, ll add) {
    for(; pos < size; pos |= (pos + 1)) {
       dataMul[pos] += mul;
       dataAdd[pos] += add;
    }
  }

  // calculate sum on [0..pos]
  ll query(int pos) {
    ll mul = 0, add = 0, start = pos;
    for(; pos >= 0; pos = (pos & (pos + 1)) - 1) {
      mul += dataMul[pos];
      add += dataAdd[pos];
    }
    return mul * start + add;
  }

  // add val to all elements on [l..r]
  void add_range(int l, int r, ll val) {
    add(l, val, (l - 1) * -val);
    add(r, -val, r * val);
  }

  // calculate sum on [l..r]
  ll sum(int l, int r) {
    return query(r) - query(l - 1);
  }

  // find k-th order statictic, almost binsearch
  int kth(ll k) {
    int res = 0;
   for(int i = mxlog; i >= 0; i--) {
     if(dataAdd[res + (1 << i) - 1] < k) {
       k -= dataAdd[res + (1 << i) - 1];
       res += ((1 << i));
     }
   }
   return res;
  }

} bit;
```

2.3 Sparse table

Description: Not to fuck up.
Time Complexity: $\mathcal{O}(n \log n), \mathcal{O}(1)$

```
int st[LOG2MAXN][MAXN], lg[MAXN];
int n;

void initST() {
  for(int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
  for(int i = 1; i < LOG2MAXN; i++)
    for(int j = 0; j < n - (1 << (i - 1)); j++)
      st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
}

int queryST(int l, int r) {
  int curlg = lg[r - l + 1];
  return min(st[curlg][l], st[curlg][r - (1 << (curlg)) + 1]);
}
```

3 Math

3.1 Linear inverse modulo prime

Description: Suprisingly laconic.
Time Complexity: $\mathcal{O}(p)$

```
inverse[1] = 1;
for (int i = 2; i < p; i++)
  inverse[i] = (p - (p / i) * inverse[p % i] % p) % p;
```

3.2 FFT

Description: You never know, you never know...
Time Complexity: $\mathcal{O}(n \log n)$

```
struct cd {
  double real, imag;

  cd() {}
  cd(double _real, double _imag) : real(_real), imag(_imag) {}
  void operator/= (const int k) { real /= k, imag /= k; }
  cd operator* (const cd & a) { return cd((real * a.real - imag *
  a.imag), (real * a.imag + imag * a.real)); }
  cd operator- (const cd & a) { return cd((real - a.real), (imag -
  a.imag)); }
  cd operator+ (const cd & a) { return cd((real + a.real), (imag +
  a.imag)); }
};

const int LOG = 20;
const int N = 1 << LOG;
cd A[N], B[N], C[N], F[2][N], w[N];
int rev[N];

void initFFT() {
  double alp;
  for(int i = 0; i < N; i++) {
    alp = (2 * PI * i) / N;
    w[i] = cd(cos(alp), sin(alp));
  }
  int k = 0;
  for(int mask = 1; mask < N; mask++) {
    if(mask == (1 << (k + 1))) k++;
    rev[mask] = rev[mask ^ (1 << k)] ^ (1 << (LOG - 1 - k));
  }
}

void FFT(cd * A, int k) {
  int L = 1 << k;
  for(int mask = 0; mask < L; mask++) F[0][rev[mask] >> (LOG - k)] =
  A[mask];
  int t = 0, nt = 1;
  for(int lvl = 0; lvl < k; lvl++) {
    int len = 1 << lvl;
    for(int st = 0; st < L; st += (len << 1)) {
      for(int i = 0; i < len; i++) {
        cd add = F[t][st + len + i] * w[(i << (LOG - 1 - lvl))];
        F[nt][st + i] = F[t][st + i] + add;
        F[nt][st + len + i] = F[t][st + i] - add;
```

```
      }
    }
    swap(t, nt);
  }
  for(int i = 0; i < L; i++) A[i] = F[t][i];
}

void invFFT(cd * A, int k) {
  FFT(A, k);
  for(int i = 0; i < (1 << k); i++) A[i] /= (1 << k);
  reverse(A + 1, A + (1 << k));
}

void input() {
  int n;
  cin >> n;
  int k = 0;
  while((1 << k) < 2 * n + 1) k++;
  for(int i = 0; i < n + 1; i++) cin >> A[n - i].real;
  for(int i = 0; i < n + 1; i++) cin >> B[n - i].real;
  FFT(A, k);
  FFT(B, k);
  for(int i = 0; i < (1 << k); i++) C[i] = A[i] * B[i];
  invFFT(C, k);
  for(int i = 2 * n; i >= 0; i--) cout << (ll)(round(C[i].real)) <<
  ' ';
  cout << "\n";
}
```

3.3 Gauss

Description: Solves system of linear equations.
Time Complexity: $\mathcal{O}(n^3)$

```
// returns the number of solutions
int gauss() {
  // n - number of equations
  // m - number of variables
  int m = n;
  memset(where, -1, sizeof(where));
  for(int row = 0, col = 0; col < m && row < n; col++) {
    int sel = row;
    //[PARTIAL PIVOTING]
    for(int i = row; i < n; i++)
      if(abs(aa[i][col]) > abs(aa[sel][col]))
        sel = i;
    // if no pivot - skip this line
    // means that variable #col is independent
    if(abs(aa[sel][col]) < EPS) continue;
    // else swap two lines
    for(int i = col; i <= m; i++) swap(aa[sel][i], aa[row][i]);
    where[col] = row;
    //[/PARTIAL PIVOTING]
    //[KILL EVERYBODY IN THIS COL]
    for(int i = 0; i < n; i++)
      if(i != row) {
        double c = aa[i][col] / aa[row][col];
        for(int j= col; j <= m; j++)
          aa[i][j] -= aa[row][j] * c;
      }
    //[/KILL EVERYBODY IN THIS COL]
    row++;
  }
  // count ans
  for(int i = 0; i < m; i++)
    if(where[i] != -1)
      ans[i] = aa[where[i]][n] / aa[where[i]][i];
  // check if any solution exist
  for(int i = 0; i < n; i++) {
    double sum = 0;
    for(int j = 0; j < m; j++)
      sum += ans[j] * aa[i][j];
    if(abs(sum - aa[i][m]) > EPS) return 0;
  }
  // check if we have independent variables
  for(int i = 0; i < m; i++) if(where[i] == -1) return INF;
  return 1;
}
```