Gungnir's Standard Code Library

Shanghai Jiao Tong University

Dated: July 25, 2016

Contents

Chapter 1 Computational Geometry

1.1 2D

1.1.1 Basis

```
typedef double DB;
const DB eps = 1e-8;

inline int sign(DB x) {
  return x < -eps ? -1 : ( x > eps ? 1 : 0 );
}
inline DB msqrt(DB x) {
  return sign(x) > 0 ? sqrt(x) : 0;
}

struct Point {
  DB x, y;
  inline Point(): x(0), y(0) {}
  inline Point(DB x, DB y): x(x), y(y) {}
  inline Point operator+(const Point &rhs) const {
    return Point(x + rhs.x, y + rhs.y);
  }
  inline Point operator-(const Point &rhs) const {
    return Point(x - rhs.x, y - rhs.y);
  }
  inline Point operator*(DB k) const {
    return Point(x * k, y * k);
  }
  inline Point operator/(DB k) const {
    assert(sign(k));
    return Point(x / k, y / k);
  }
};

inline DB dot(const P& a, const P& b) {
  return a.x * b.x + a.y * b.y;
}

inline DB det(const P& a, const P& b) {
  return a.x * b.y - a.y * b.x;
}
```

Chapter 2 Graph Theory

2.1 Basis

```
struct Graph {  // Remember to call .init()!
    int e, nxt[M], v[M], adj[N], n;
    bool base;
    inline void init(bool _base, int _n = 0) {
        assert(n < N);
        n = _n; base = _base;
        e = 0; memset(adj + base, -1, sizeof(*adj) * n);
    }
    inline int new_node() {
        adj[n + base] = -1;
        assert(n + base + 1 < N);
        return n++ + base;
    }
    inline void ins(int u0, int v0) {  // directional
        assert(u0 < n + base && v0 < n + base);
        v[e] = v0; nxt[e] = adj[u0]; adj[u0] = e++;
        assert(e < M);
    }
    inline void bi_ins(int u0, int v0) {  // bi-directional
        ins(u0, v0); ins(v0, u0);
    }
};
```

2.2 Double Connected Graph (vertex)

`dcc.forest` is a set of connected tree whose vertices are chequered with cut-vertex and DCC.

```
const bool DCC_VERTEX = 0, DCC_EDGE = 1;
struct DCC {  // N = N0 + M0. Remember to call init(&raw_graph).
  Graph *g, forest; // g is raw graph ptr.
  int dfn[N], DFN, low[N];
  int stack[N], top;
  int expand_to[N];   // Where edge i is expanded to in expaned graph.
  // Vertex i expaned to i.
  int compress_to[N];  // Where vertex i is compressed to.
  bool vertex_type[N], cut[N], compress_cut[N], branch[M];
  //std::vector<int> DCC_component[N]; // Cut vertex belongs to none.
  inline void init(Graph *raw_graph) {
    g = raw_graph;
  }
  void DFS(int u, int pe) {
    dfn[u] = low[u] = ++DFN; cut[u] = false;
    if (!~g->adj[u]) {
      cut[u] = 1;
      compress_to[u] = forest.new_node();
      compress_cut[compress_to[u]] = 1;
    }
    for (int e = g->adj[u]; ~e; e = g->nxt[e]) {
      int v = g->v[e];
      if ((e ^ pe) > 1 && dfn[v] > 0 && dfn[v] < dfn[u]) {
        stack[top++] = e;
        low[u] = std::min(low[u], dfn[v]);
      }
      else if (!dfn[v]) {
        stack[top++] = e; branch[e] = 1;
        DFS(v, e);
        low[u] = std::min(low[v], low[u]);
        if (low[v] >= dfn[u]) {
          if (!cut[u]) {
            cut[u] = 1;
            compress_to[u] = forest.new_node();
            compress_cut[compress_to[u]] = 1;
          }
          int cc = forest.new_node();
          forest.bi_ins(compress_to[u], cc);
          compress_cut[cc] = 0;
          //DCC_component[cc].clear();
          do {
            int cur_e = stack[--top];
            compress_to[expand_to[cur_e]] = cc;
            if (branch[cur_e]) {
              int v = g->v[cur_e];
              if (cut[v])
                forest.bi_ins(cc, compress_to[v]);
              else {
                //DCC_component[cc].push_back(v);
                compress_to[v] = cc;
              }
            }
          } while (stack[top] != e);
        }
      }
    }
  }
  void solve() {
    forest.init(g->base);
    int n = g->n;
    for (int i = 0; i < g->e; i++) {
      expand_to[i] = g->new_node();
      branch[i] = 0;
    }
    memset(dfn + g->base, 0, sizeof(*dfn) * n); DFN = 0;
    for (int i = 0; i < n; i++)
      if (!dfn[i + g->base]) {
        top = 0;
        DFS(i + g->base, -1);
      }
  }
} dcc;

dcc.init(&raw_graph);
dcc.solve();
// Do something with dcc.forest ...
```