ACM ICPC Team Reference 2010 World Finals

Team Anuncie Aqui
Universidade Federal de Sergipe

Contents

1. Configuration files and scripts

1.1. .emacs. Hash: 7b957da3d3526ffc2b050f1e572cd34d

```
(global-font-lock-mode t)
(setq transient-mark-mode t)
(require 'font-lock)
(require␣'paren)
(global-set-key [f5] 'cxx-compile)
(set-input-mode␣nil␣nil␣1)
(fset␣'yes-or-no-p 'y-or-n-p)

(require␣'cc-mode)
(defun cxx-compile()
  (interactive)
  (progn
    (save-buffer)
    (compile (concat "g++␣-g␣-O2␣-o␣" (substring buffer-file-name 0 -4)
           buffer-file-name))
  )
)

(add-hook 'c++-mode-hook␣'(lambda () (c-set-style "stroustrup")))
```

1.2. **Hash generator.** Hash: bc6ecca6940a0b9f7ac157cdb76c9e95

```
#!/bin/sh
sed ':a;N;$!ba;s/[ \n\t]//g' | md5sum
```

2. Graph algorithms

2.1. **Dijkstra's algorithm.** Hash: 43f08cddaedf61cf177a2d6ab79f6ec5

```
int dist[MAXV], last_edge[MAXV], d_visited[MAXV];
int prev_edge[MAXE], weight[MAXE], adj[MAXE];
int nedges;
priority_queue<pair<int, int> > d_q;

void d_init() {
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
}

void d_aresta(int v, int w, int eweight) {
   prev_edge[nedges] = last_edge[v];
   weight[nedges] = eweight;
   adj[nedges] = w;
   last_edge[v] = nedges++;
}
```

```
void dijkstra(int s, int num_nodes = MAXV) {
   memset(dist, 0x3f, sizeof dist);
   memset(d_visited, 0, sizeof d_visited);
   d_q.push(make_pair(dist[s] = 0, s));

   while(!d_q.empty()) {
      int v = d_q.top().second; d_q.pop();
      if(d_visited[v]) continue; d_visited[v] = true;

      for(int i = last_edge[v]; i != -1; i = prev_edge[i]) {
         int w = adj[i], new_dist = dist[v] + weight[i];
         if(new_dist < dist[w])
            d_q.push(make_pair(-(dist[w] = new_dist), w));
      }
   }
}
```

2.2. **Gabow's algorithm.** Hash: e238cc09d9e95923de0373606443f296

```
int prev_edge[MAXE], v[MAXE], w[MAXE], last_edge[MAXV];
int type[MAXV], label[MAXV], first[MAXV], mate[MAXV], nedges;
bool g_flag[MAXV], g_souter[MAXV];

void g_init() {
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
}

void g_aresta(int a, int b) {
   prev_edge[nedges] = last_edge[a];
   v[nedges] = a;
   w[nedges] = b;
   last_edge[a] = nedges++;
   prev_edge[nedges] = last_edge[b];
   v[nedges] = b;
   w[nedges] = a;
   last_edge[b] = nedges++;
}

void g_label(int v, int join, int edge, queue<int>& outer) {
   if(v == join) return;
   if(label[v] == -1) outer.push(v);

   label[v] = edge;
   type[v] = 1;
   first[v] = join;

   g_label(first[label[mate[v]]], join, edge, outer);
```

```
}

void g_augment(int _v, int _w) {
   int t = mate[_v];
   mate[_v] = _w;

   if(mate[t] != _v) return;
   if(label[_v] == -1) return;

   if(type[_v] == 0) {
      mate[t] = label[_v];
      g_augment(label[_v], t);
   }
   else if(type[_v] == 1) {
      g_augment(v[label[_v]], w[label[_v]]);
      g_augment(w[label[_v]], v[label[_v]]);
   }
}

int gabow(int n) {
   memset(mate, -1, sizeof mate);
   memset(first, -1, sizeof first);

   int u = 0, ret = 0;
   for(int z = 0; z < n; z++) {
      if(mate[z] != -1) continue;

      memset(label, -1, sizeof label);
      memset(type, -1, sizeof type);
      memset(g_souter, 0, sizeof g_souter);

      label[z] = -1; type[z] = 0;

      queue<int> outer;
      outer.push(z);

      bool done = false;
      while(!outer.empty()) {
         int x = outer.front(); outer.pop();

         if(g_souter[x]) continue;
         g_souter[x] = true;

         for(int i = last_edge[x]; i != -1; i = prev_edge[i]) {
            if(mate[w[i]] == -1 && w[i] != z) {
               mate[w[i]] = x;
               g_augment(x, w[i]);
               ret++;

               done = true;
               break;
            }

            if(type[w[i]] == -1) {
               int v = mate[w[i]];
               if(type[v] == -1) {
                  type[v] = 0;
                  label[v] = x;
                  outer.push(v);

                  first[v] = w[i];
               }
               continue;
            }

            int r = first[x], s = first[w[i]];
            if(r == s) continue;

            memset(g_flag, 0, sizeof g_flag);
            g_flag[r] = g_flag[s] = true;

            while(true) {
               if(s != -1) swap(r, s);
               r = first[label[mate[r]]];
               if(g_flag[r]) break; g_flag[r] = true;
            }

            g_label(first[x], r, i, outer);
            g_label(first[w[i]], r, i, outer);

            for(int c = 0; c < n; c++)
               if(type[c] != -1 && first[c] != -1 && type[first[c]] != -1)
                  first[c] = r;
         }
         if(done) break;
      }
   }
   return ret;
}
```

2.3. **Dinic's algorithm.** Hash: fe9ff338463acc9e482635277b7343df

```
int last_edge[MAXV], cur_edge[MAXV], dist[MAXV], visited[MAXV];
int prev_edge[MAXE], cap[MAXE], flow[MAXE], adj[MAXE];
int nedges;

void d_init() {
   nedges = 0;
   memset(last_edge, -1, sizeof last_edge);
}

void d_aresta(int v, int w, int capacity, bool r = true) {
   prev_edge[nedges] = last_edge[v];
   cap[nedges] = capacity;
   adj[nedges] = w;
   flow[nedges] = 0;
   last_edge[v] = nedges++;

   if(r) d_aresta(w, v, 0, false);
}

bool d_auxflow(int source, int sink) {
   queue<int> q;
   q.push(source);

   memset(dist, -1, sizeof dist);
   dist[source] = 0;
   memcpy(cur_edge, last_edge, sizeof last_edge);

   while(!q.empty()) {
      int v = q.front(); q.pop();
      for(int i = last_edge[v]; i != -1; i = prev_edge[i]) {
         if(cap[i] - flow[i] == 0) continue;

         if(dist[adj[i]] == -1) {
            dist[adj[i]] = dist[v] + 1;
            q.push(adj[i]);

            if(adj[i] == sink) return true;
         }
      }
   }

   return false;
}

inline int rev(int i) { return i ^ 1; }

int d_augmenting(int v, int sink, int c) {
   if(v == sink) return c;
   if(visited[v]) return 0;
   visited[v] = true;

   for(int& i = cur_edge[v]; i != -1; i = prev_edge[i]) {
      if(cap[i] - flow[i] == 0 || dist[adj[i]] != dist[v] + 1)
         continue;

      int val;
      if(val = d_augmenting(adj[i], sink, min(c, cap[i] - flow[i]))) {
         flow[i] += val;
         flow[rev(i)] -= val;
         return val;
      }
   }

   return 0;
}

int dinic(int source, int sink) {
   int ret = 0;
   while(d_auxflow(source, sink)) {
      int flow;
      memset(visited, 0, sizeof visited);
      while(flow = d_augmenting(source, sink, 0x3f3f3f3f))
         ret += flow;
   }

   return ret;
}
```

3.1. **Fractions.** Hash: a700805eec8c0f64c93c38da7239d2cc

```
#include <algorithm>
#include <iostream>

struct frac {
   long long num, den;

   frac() : num(0), den(1) { };
   frac(long long num, long long den) { set_val(num, den); }
   frac(long long num) : num(num), den(1) { };

   void set_val(long long _num, long long _den) {
      num = _num/std::__gcd(_num, _den);
      den = _den/std::__gcd(_num, _den);
      if(den < 0) { num *= -1; den *= -1; }
   }

   void operator*=(frac f) { set_val(num * f.num, den * f.den); }
   void operator+=(frac f) { set_val(num * f.den + f.num * den, den * f.den); }
   void operator-=(frac f) { set_val(num * f.den - f.num * den, den * f.den); }
   void operator/=(frac f) { set_val(num * f.den, den * f.num); }
};

bool operator<(frac a, frac b) {
   if((a.den < 0) ^ (b.den < 0)) return a.num * b.den > b.num * a.den;
   return a.num * b.den < b.num * a.den;
}

std::ostream& operator<<(std::ostream& o, const frac f) {
   o << f.num << "/" << f.den;
   return o;
}

bool operator==(frac a, frac b) { return a.num * b.den == b.num * a.den; }
bool operator!=(frac a, frac b) { return !(a == b); }
bool operator<=(frac a, frac b) { return (a == b) || (a < b); }
bool operator>=(frac a, frac b) { return !(a < b); }
bool operator>(frac a, frac b) { return !(a <= b); }
frac operator/(frac a, frac b) { frac ret = a; ret /= b; return ret; }
frac operator*(frac a, frac b) { frac ret = a; ret *= b; return ret; }
frac operator+(frac a, frac b) { frac ret = a; ret += b; return ret; }
frac operator-(frac a, frac b) { frac ret = a; ret -= b; return ret; }
frac operator-(frac f) { return 0 - f; }
```

3.2. **Chinese remainder theorem.** Hash: 1e69e6802de5d9eb002fea6e59ac9402

```
struct t {
   long long a, b; int g;
   t(long long a, long long b, int g) : a(a), b(b), g(g) { }
   t swap() { return t(b, a, g); }
};

t egcd(int p, int q) {
   if(q == 0) return t(1, 0, p);

   t t2 = egcd(q, p % q);
   t2.a -= t2.b * (p/q);
   return t2.swap();
}

int crt(int a, int p, int b, int q) {
   t t2 = egcd(p, q); t2.a %= p*q; t2.b %= p*q;
   assert(t2.g == 1);
   int ret = ((b * t2.a)%(p*q) * p + (a * t2.b)%(p*q) * q) % (p*q);
   return ret >= 0 ? ret : ret + p*q;
}
```

4. Geometry

4.1. **Point class.** Hash: fdb0b5a1db66a3b070e495d64092ebd8

```
typedef double TYPE;
const TYPE EPS = 1e-9;

inline int sgn(TYPE a) { return a > EPS ? 1 : (a < -EPS ? -1 : 0); }
```

```
inline int cmp(TYPE a, TYPE b) { return sgn(b - a); }

struct pt {
   TYPE x, y;
   pt(TYPE x, TYPE y = 0) : x(x), y(y) { }

   TYPE operator||(pt p) { return x*p.x + y*p.y; }
   TYPE operator%(pt p) { return x*p.y - y*p.x; }
   pt operator~() { return pt(x, -y); }
   pt operator+(pt p) { return pt(x + p.x, y + p.y); }
   pt operator-(pt p) { return pt(x - p.x, y - p.y); }
   pt operator*(pt p) { return pt(x*p.x - y*p.y, x*p.y + y*p.x); }
   pt operator/(TYPE t) { return pt(x/t, y/t); }
   pt operator/(pt p) { return (*this * ~p)/(p||p); }
};

TYPE norm(pt a) { return a||a; }
TYPE abs(pt a) { return sqrt(a||a); }
TYPE ccw(pt a, pt b, pt c) { return cmp((a-c)%(b-c)); }
double arg(pt a) { return atan2(a.y, a.x); }
pt f_polar(TYPE mod, double ang) { return pt(mod * cos(ang), mod * sin(ang)); }

ostream& operator<<(ostream& o, pt p) {
   return o << "(" << p.x << "," << p.y << ")";
}
```

5. Data structures

5.1. **Fractions.** Hash: 55270191425b80c1e23745937a2abbb6

```
typedef int TYPE;

class treap {
public:
   treap *left, *right;
   int priority, sons;
   TYPE value;

   treap(TYPE value) : left(NULL), right(NULL), value(value), sons(0) {
      priority = rand();
   }

   ~treap() {
      if(left) delete left;
      if(right) delete right;
   }
};

treap* find(treap* t, TYPE val) {
   if(!t) return NULL;
   if(val == t->value) return t;

   if(val < t->value) return find(t->left, val);
   if(val > t->value) return find(t->right, val);
}

void rotate_to_right(treap* &t) {
   treap* n = t->left;
   t->left = n->right;
   n->right = t;
   t = n;
}

void rotate_to_left(treap* &t) {
   treap* n = t->right;
   t->right = n->left;
   n->left = t;
   t = n;
}

void fix_augment(treap* t) {
   if(!t) return;
   t->sons = (t->left ? t->left->sons + 1 : 0) +
      (t->right ? t->right->sons + 1 : 0);
}

void insert(treap* &t, TYPE val) {
   if(!t)
      t = new treap(val);
   else
      insert(val <= t->value ? t->left : t->right, val);

   if(t->left && t->left->priority > t->priority)
      rotate_to_right(t);
```

```
   else if(t->right && t->right->priority > t->priority)
      rotate_to_left(t);

   fix_augment(t->left); fix_augment(t->right); fix_augment(t);
}

inline int p(treap* t) {
   return t ? t->priority : -1;
}

void erase(treap* &t, TYPE val) {
   if(!t) return;
   if(t->value != val)
      erase(val < t->value ? t->left : t->right, val);
   else {
      if(!t->left && !t->right)
         delete t, t = NULL;
      else {
         p(t->left) < p(t->right) ? rotate_to_left(t) : rotate_to_right(t);
         erase(t, val);
      }
   }

   fix_augment(t->left); fix_augment(t->right); fix_augment(t);
}
```