Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

11. November 2014

Inhaltsverzeichnis

1 Datenstrukturen

1.1 Union-Find

```
1  vector<int> parent, rank2; //manche compiler verbieten Variable mit Namen rank
2
3  int findSet(int n) { //Pfadkompression
4    if (parent[n] != n) parent[n] = findSet(parent[n]);
5    return parent[n];
6  }
7
8  void linkSets(int a, int b) { //union by rank
9    if (rank2[a] < rank2[b]) parent[a] = b;
10   else if (rank2[b] < rank2[a]) parent[b] = a;
11   else {
12     parent[a] = b;
13     rank2[b]++;
14   }
15 }
16
17 void unionSets(int a, int b) {
18   if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
19 }
```

1.2 Segmentbaum

```
1  int a[MAX_N], m[4 * MAX_N];
2
3  int query(int x, int y, int k = 0, int X = 0, int Y = MAX_N - 1) {
4    if (x <= X && Y <= y) return m[k];
5    if (y < X || Y < x) return -1000000000; //ein "neutrales" Element
6    int M = (X + Y) / 2;
7    return max(query(x, y, 2 * k + 1, X, M), query(x, y, 2 * k + 2, M + 1, Y));
8  }
9
10 void update(int i, int v, int k = 0, int X = 0, int Y = MAX_N - 1) {
11   if (i < X || Y < i) return;
12   if (X == Y) {
13     m[k] = v;
14     a[i] = v;
15     return;
16   }
17   int M = (X + Y) / 2;
18   update(i, v, 2 * k + 1, X, M);
19   update(i, v, 2 * k + 2, M + 1, Y);
20   m[k] = max(m[2 * k + 1], m[2 * k + 2]);
21 }
22
23 void init(int k = 0, int X = 0, int Y = MAX_N - 1) {
24   if (X == Y) {
25     m[k] = a[X];
26     return;
27   }
28   int M = (X + Y) / 2;
29   init(2 * k + 1, X, M);
30   init(2 * k + 2, M + 1, Y);
31   m[k] = max(m[2 * k + 1], m[2 * k + 2]);
32 }
```

2 Graphen

2.1 Strongly Connected Components (Tarjans-Algorithmus)

```
1  int counter, sccCounter, n; //n == number of vertices
2  vector<bool> visited, inStack;
3  vector< vector<int> > adjlist;
4  vector<int> d, low, sccs;
5  stack<int> s;
6
```

```
 7  void visit(int v) {
 8    visited[v] = true;
 9    d[v] = counter;
10    low[v] = counter;
11    counter++;
12    inStack[v] = true;
13    s.push(v);
14
15    for (int i = 0; i < (int)adjlist[v].size(); i++) {
16      int u = adjlist[v][i];
17      if (!visited[u]) {
18        visit(u);
19        low[v] = min(low[v], low[u]);
20      } else if (inStack[u]) {
21        low[v] = min(low[v], low[u]);
22      }
23    }
24
25    if (d[v] == low[v]) {
26      int u;
27      do {
28        u = s.top();
29        s.pop();
30        inStack[u] = false;
31        sccs[u] = sccCounter;
32      } while(u != v);
33      sccCounter++;
34    }
35  }
36
37  void scc() {
38    //read adjlist
39
40    visited.clear(); visited.assign(n, false);
41    d.clear(); d.resize(n);
42    low.clear(); low.resize(n);
43    inStack.clear(); inStack.assign(n, false);
44    sccs.clear(); sccs.resize(n);
45
46    counter = 0;
47    sccCounter = 0;
48    for (i = 0; i < n; i++) {
49      if (!visited[i]) {
50        visit(i);
51      }
52    }
53    //sccs has the component for each vertex
54  }
```

2.2   Max-Flow (EDMONDS-KARP-Algorithmus)

```
 1  int s, t, f; //source, target, single flow
 2  int res[MAX_V][MAX_V]; //adj-matrix
 3  vector< vector<int> > adjList;
 4  int p[MAX_V]; //bfs spanning tree
 5
 6  void augment(int v, int minEdge) {
 7    if (v == s) { f = minEdge; return; }
 8    else if (p[v] != -1) {
 9      augment(p[v], min(minEdge, res[p[v]][v]));
10      res[p[v]][v] -= f; res[v][p[v]] += f;
11  }}
12
13  int maxFlow() { //first inititalize res, adjList, s and t
14    int mf = 0;
15    while (true) {
16      f = 0;
17      bitset<MAX_V> vis; vis[s] = true;
18      queue<int> q; q.push(s);
19      memset(p, -1, sizeof(p));
20      while (!q.empty()) { //BFS
```

```
21        int u = q.front(); q.pop();
22        if (u == t) break;
23        for (int j = 0; j < (int)adjList[u].size(); j++) {
24          int v = adjList[u][j];
25          if (res[u][v] > 0 && !vis[v]) {
26            vis[v] = true; q.push(v); p[v] = u;
27      }}}
28
29      augment(t, INF); //add found path to max flow
30      if (f == 0) break;
31      mf += f;
32    }
33    return mf;
34 }
```

3   Geometrie

3.1   Closest Pair

```
 1 double squaredDist(point a, point b) {
 2   return (a.first-b.first) * (a.first-b.first) + (a.second-b.second) * (a.second-b.second);
 3 }
 4
 5 bool compY(point a, point b) {
 6   if (a.second == b.second) return a.first < b.first;
 7   return a.second < b.second;
 8 }
 9
10 double shortestDist(vector<point> &points) {
11   //check that points.size() > 1 and that ALL POINTS ARE DIFFERENT
12   set<point, bool(*)(point, point)> status(compY);
13   sort(points.begin(), points.end());
14   double opt = 1e30, sqrtOpt = 1e15;
15   auto left = points.begin(), right = points.begin();
16   status.insert(*right); right++;
17
18   while (right != points.end()) {
19     if (fabs(left->first - right->first) >= sqrtOpt) {
20       status.erase(*(left++));
21     } else {
22       auto lower = status.lower_bound(point(-1e20, right->second - sqrtOpt));
23       auto upper = status.upper_bound(point(-1e20, right->second + sqrtOpt));
24       while (lower != upper) {
25         double cand = squaredDist(*right, *lower);
26         if (cand < opt) {
27           opt = cand;
28           sqrtOpt = sqrt(opt);
29         }
30         ++lower;
31       }
32       status.insert(*(right++));
33     }
34   }
35   return sqrtOpt;
36 }
```

3.2   Formeln - `std::complex`

```
 1 //komplexe Zahlen als Darstellung fuer Punkte
 2 typedef pt complex<double>;
 3 //Winkel zwischen Punkt und x-Achse in [0, 2 * PI), Winkel zwischen a und b
 4 double angle = arg (a), angle_a_b = arg (a - b);
 5 //Punkt rotiert um Winkel theta
 6 pt a_rotated = a * exp (pt (0, theta));
 7 //Mittelpunkt des Dreiecks abc
 8 pt centroid = (a + b + c) / 3;
 9 //Skalarprodukt
10 double dot(pt a, pt b) {
11   return real(conj(a) * b);
12 }
```

```cpp
13  //Kreuzprodukt, 0, falls kollinear
14  double cross(pt a, pt b) {
15    return imag(conj(a) * b);
16  }
17  //wenn Eckpunkte bekannt
18  double areaOfTriangle(pt a, pt b, pt c) {
19    return abs(cross(b - a, c - a)) / 2.0;
20  }
21  //wenn Seitenlaengen bekannt
22  double areaOfTriangle(double a, double b, double c) {
23    double s = (a + b + c) / 2;
24    return sqrt(s * (s-a) * (s-b) * (s-c));
25  }
26  // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 aehnlich?
27  // Erste Zeile testet Aehnlichkeit mit gleicher Orientierung,
28  // zweite Zeile testst Aehnlichkeit mit unterschiedlicher Orientierung
29  bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
30    return (
31      (b2 - a2) * (c1 - a1) == (b1 - a1) * (c2 - a2) ||
32      (b2 - a2) * (conj (c1) - conj (a1)) == (conj (b1) - conj (a1)) * (c2 - a2)
33    );
34  }
35  //Linksknick von a->b nach a->c
36  double ccw(pt a, pt b, pt c) {
37    return cross(b - a, c - a); //<0 => falls Rechtsknick, 0 => kollinear, >0 => Linksknick
38  }
39  //Streckenschnitt, Strecken a-b und c-d
40  bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
41    if (ccw(a, b, c) == 0 && ccw(a, b, d) == 0) { //kollinear
42      double dist = abs(a - b);
43      return (abs(a - c) <= dist && abs(b - c) <= dist) || (abs(a - d) <= dist && abs(b - d) <= dist);
44    }
45    return ccw(a, b, c) * ccw(a, b, d) <= 0 && ccw(c, d, a) * ccw(c, d, b) <= 0;
46  }
47  //Entfernung von p zu a-b
48  double distToLine(pt a, pt b, pt p) {
49    return abs(cross(p - a, b - a)) / abs(b - a);
50  }
51  //liegt p auf a-b
52  bool pointOnLine(pt a, pt b, pt p) {
53    return abs(distToLine(a, b, p)) < EPSILON;
54  }
55  //testet, ob d in der gleichen Ebene liegt wie a, b, und c
56  bool isCoplanar(pt a, pt b, pt c, pt d) {
57    return (b - a) * (c - a) * (d - a) == 0;
58  }
```

4   Mathe

4.1   Binomialkoeffizienten

```cpp
1  ll calc_binom(ll N, ll K) {
2    ll r = 1, d;
3    if (K > N) return 0;
4    for (d = 1; d <= K; d++) {
5      r *= N--;
6      r /= d;
7    }
8    return r;
9  }
```

5   Sonstiges

5.1   2-SAT

1. Bedingungen in 2-CNF formulieren.

2. Implikationsgraph bauen, $(a \lor b)$ wird zu $\neg a \Rightarrow b$ und $\neg b \Rightarrow a$.

3. Finde die starken Zusammenhangskomponenten.

4. Genau dann lösbar, wenn keine Variable mit ihrer Negation in einer SCC liegt.