Team notebook

April 8, 2016

Contents

1 dataStructure

1.1 disjoint$_s et_u nion - find$

```
#include <bits/stdc++.h>
#define N 1000
using namespace std;
/*
  when is used Weighted-union with path compression
  it take O(log*(N)) for each union find operation.
  Where N is the numbre of elements in the set.
  in real world log*(N) reaches at most up 5
*/
int p[N];
int rank[N];
// initialize each node as your self root
void initialize(int n) {
  for (int i = 0; i < n; i++) {
    p[i] = i;
    rank[i] = 1;
  }
}

// find_set root of an element and path compression
int find_set(int elem) {
  return ((p[elem] == elem)? elem : p[elem] = find_set(p[elem]));
}

/* find_set iterative
  int find_set(int elem) {
  while (p[elem] != elem) {
    p[elem] = p[p[elem]];
    elem = p[elem];
  }
}
*/

void join_sets(int a, int b) {
  int pa = find_set(a);
  int pb = find_set(b);
```

```
  if (pa != pb) {
    p[pa] = pb;
    rank[pb] += rank[pa];
  } else {
    return;
  }
}

/* Weighted-join_sets(rank)
void join_sets(int a, int b) {
  int root_a = find_set(a);
  int root_b = find_set(b);
  if (rank[root_a] > rank[root_b]) {
    p[root_b] = root_a;
  } else {
    p[root_a] = root_b;
  }
  if (rank[root_a] == rank[root_b]) {
    rank[root_b] = rank[root_a] + 1;
  }
}
*/

// Are u and v in the same connected component?
bool same_component(int u, int v) {
  return find_set(u) == find_set(v);
}

void show_dsu(int nodes) {
  for (int i = 0; i < nodes; i++) {
    cout << find_set(i) << " ";
  }
  cout << endl;
}

int main() {
  int nodes, edges, u, w, q;
  cin >> nodes >> edges;
  initialize(nodes);
  while (edges--) {
    cin >> u >> w;
    join_sets(u, w);
  }
  // queries
  while ((cin >> u >> w) && (u != -1 || w != -1)) {
    cout << ((same_component(u, w))? "same :)" : "not same :(") << endl;
  }
  show_dsu(nodes);
  return 0;
}
```

2 graphs

2.1 bfs

```
#include <bits/stdc++.h>
/*
APPLICATIONS OF BFS
http://www.geeksforgeeks.org/applications-of-breadth-first-traversal/
*/
using namespace std;

class Graph {
  int nvertices;
  list<int> *adj;

public:
  Graph(int v);
  void add_edge(int v, int w, bool directed);
  void bfs(int s);
};

Graph::Graph(int v) {
  this->nvertices = v;
  adj = new list<int>[v];
}

void Graph::add_edge(int v, int w, bool directed) {
  adj[v].push_back(w);
  if (!directed)
    add_edge(w, v, true);
}

void Graph::bfs(int s) {
  vector<bool> visited(nvertices, false);
  queue<int> q;
```

```
  q.push(s);
  visited[s] = true;
  list<int>::iterator it;

  while (!q.empty()) {
    s = q.front();
    cout << s << " ";
    q.pop();
    for (it = adj[s].begin(); it != adj[s].end(); ++it) {
      if (!visited[*it]) {
        visited[*it] = true;
        q.push(*it);
      }
    }
  }
}

int main() {
    int v, e, x, y;
    cout << "Number of vertices: ";
    cin >> v;
    cout << "Number of Edges: ";
    cin >> e;
    Graph g(v);
    for (int i = 0; i < e; ++i) {
      cin >> x >> y;
      g.add_edge(x, y, true);
    }

    cout << "Following is Breadth First Traversal (starting from vertex
        2) \n";
    g.bfs(2);

    return 0;
}
```

2.2 bfs*levels*

```
#include <bits/stdc++.h>

using namespace std;

class Graph {
  int nvertices;
  list<int> *adj;
  vector<bool> visited;
  int *level;
public:
  Graph(int v);
  void add_edge(int v, int w, bool directed);
  void bfs(int s);
  void show_levels();
};

Graph::Graph(int v) {
  this->nvertices = v;
  adj = new list<int>[v];
  level = new int[v];
  visited.assign(v, false);
}

void Graph::add_edge(int v, int w, bool directed) {
  adj[v].push_back(w);
  if (!directed)
    add_edge(w, v, true);
}

void Graph::bfs(int s) {
  queue<int> q;
  list<int>::iterator i;

  q.push(s);
  level[s] = 0;
  visited[s] = true;

  while (!q.empty()) {
    int cur = q.front();
    q.pop();

    for (i = adj[cur].begin(); i != adj[cur].end(); i++) {
      if (!visited[*i]) {
        level[*i] = level[cur] + 1;
        q.push(*i);
        visited[*i] = true;
      }
    }
  }
}
```

```
void Graph::show_levels() {
  for (int i = 0; i < nvertices; i++) {
    cout << "Nodo " << i << " con nivel " << level[i] << endl;
  }
}


int main() {
  int v, e, x, y;
  cout << "Number of vertices: ";
  cin >> v;
  cout << "Number of Edges: ";
  cin >> e;
  Graph g(v);
  for (int i = 0; i < e; ++i) {
    cin >> x >> y;
    g.add_edge(x, y, true);
  }

  g.bfs(0);
  g.show_levels();

  return 0;
}
```

2.3 bfs$_{p}arents$

```
#include <bits/stdc++.h>

using namespace std;

class Graph {
  int vertices;
  list<int> *adj;
  int *parent;

public:
  Graph(int v);
  void addEdge(int x, int y);
  void bfs(int start, int end);
  void route(int s);
};
```

```
Graph::Graph(int v) {
  vertices = v;
  adj = new list<int>[v];
  parent = new int[v];
}

void Graph::addEdge(int x, int y) {
  adj[x].push_back(y);
}

void Graph::route(int s) {
  stack<int> st;
  st.push(s);
  while (s != -1) {
    s = parent[s];
    st.push(s);
  }
  int n = st.size()-1;
  st.pop();
  for (int i = 0; i < n; ++i) {
    cout << st.top() << " ";
    st.pop();
  }
}

void Graph::bfs(int start, int end) {
  vector<bool> visited(vertices, false);
  queue<int> q;

  q.push(start);
  visited[start];
  parent[start] = -1;
  list<int>::iterator i;

  while (!q.empty()) {
    start = q.front();
    q.pop();
    for (i = adj[start].begin(); i != adj[start].end(); i++) {
      if (*i == end){
        parent[*i] = start;
        visited[*i] = true;
        route(start);
        break;
      }
```

```
      if(!visited[*i]){
        q.push(*i);
        parent[*i] = start;
        visited[*i] = true;
      }
    }
  }
}

int main() {
  int v, e, x, y, start, end;
  cin >> v >> e;
  Graph g(v);
  for (int i = 0; i < e; ++i) {
    cin >> x >> y;
    g.addEdge(x, y);
  }
  cin >> start >> end;
  g.bfs(start, end);
  return 0;
}
```

2.4 conected$_c$omponents$_d$fs

```
#include <bits/stdc++.h>

using namespace std;

class Graph {
  list<int> *adj;
  int vertices;
  void init_visited();
public:
  bool *visited;
  Graph(int vertices);
  void add_edge(int v, int w);
  void dfs(int s);
};

Graph::Graph(int vertices) {
  this->vertices = vertices;
  adj = new list<int>[vertices];
  visited = new bool[vertices];
  init_visited();
}

void Graph::init_visited() {
  for (int i = 0; i < vertices; i++) {
    visited[i] = false;
  }
}

void Graph::add_edge(int v, int w) {
  adj[v].push_back(w);
  adj[w].push_back(v);
}

void Graph::dfs(int s) {
  visited[s] = true;
  list<int>::iterator it;
  for (it = adj[s].begin(); it != adj[s].end(); it++) {
    if (!visited[*it]) {
      dfs(*it);
    }
  }
}

int main() {
  int nodes, edges, x, y, components = 0;
  cin >> nodes >> edges;
  Graph g(nodes);
  while (edges--) {
    cin >> x >> y;
    g.add_edge(x, y);
  }

  for (int i = 0; i < nodes; i++) {
    if (!g.visited[i]) {
      g.dfs(i);
      components++;
    }
  }

  cout << "Number of connected components: " << components << endl;
  return 0;
}
```

2.5 dfs

```
#include <bits/stdc++.h>

using namespace std;
/*
  APPLICATIONS OF DFS
  http://www.geeksforgeeks.org/applications-of-depth-first-search/
*/
class Graph {
  int vertices;
  list<int> *adj;
  void dfs_util(int s, bool visited[]);
public:
  Graph(int v);
  void add_edge(int v, int w);
  void dfs(int s);
};

Graph::Graph(int v) {
  vertices = v;
  adj = new list<int>[v];
}

void Graph::add_edge(int v, int w) {
  adj[v].push_back(w);
}

void Graph::dfs_util(int s, bool visited[]) {
  visited[s] = true;
  cout << s << " ";
  list<int>::iterator i;
  for (i = adj[s].begin(); i != adj[s].end(); ++i) {
    if (!visited[*i]) {
      dfs_util(*i, visited);
    }
  }
}

void Graph::dfs(int s) {
  bool *visited = new bool[vertices];
  for (int i = 0; i < vertices; ++i)
    visited[i] = false;

  dfs_util(s, visited);
}

int main() {
  int v, e, x, y;
  cout << "Number of vertices: ";
  cin >> v;
  cout << "Number of Edges: ";
  cin >> e;
  Graph g(v);
  for (int i = 0; i < e; ++i) {
    cin >> x >> y;
    g.add_edge(x, y);
  }

  cout << "Following is Depth First Traversal (starting from vertex 2)
      \n";
  g.dfs(2);
  return 0;
}
```

2.6 dfs$_{w}ith_{t}imes_{t}oposort$

```
#include <bits/stdc++.h>
#define N 1000
using namespace std;

class Graph {
  int vertices;
  list<int> *adj;
  vector<int> time_init,
              time_fin,
              state;
public:
  Graph(int vertices) {
    this->vertices = vertices;
    adj = new list<int> [vertices];
    time_init.assign(vertices, -1);
    time_fin.assign(vertices, -1);
    state.assign(vertices, 0);
  }
};

int main() {
```

```
  int nodes, edges, u, w;
  cin >> nodes >> edges;
  cout << adj.size() << endl;

  while (edges--) {
    cin >> u >> w;
  }
  return 0
}
```

2.7 dijkstra

```
/*
* Tested with http://www.spoj.com/problems/EZDIJKST/
*/
#include <bits/stdc++.h>
#define D(x) cout << #x " = " << x << endl
#define INF INT_MAX

using namespace std;

struct Edge {
  int to, dist;
  Edge() {}
  Edge(int t,int d): to(t), dist(d) {}
  bool operator < (const Edge &e) const {
    return dist > e.dist;
  }
};

int dijkstra(int s, int t,vector <vector<Edge> > &adj, vector<int> &d,
    vector<int> &p) {
  priority_queue<Edge> pq;
  d[s] = 0;
  pq.push(Edge(s, 0));
  while (!pq.empty()) {
    int cur = pq.top().to;
    int dist = pq.top().dist;
    pq.pop();
    if (cur == t) {
      if (dist > d[cur]) continue;
      // cout << "Path: " << endl;
      // while (cur != -1) {
      //   cout << cur << " ";
      //   cur = p[cur];
      // }
      // cout << endl;
      return dist;
    }
    for (int i = 0; i < adj[cur].size(); i++) {
      int to = adj[cur][i].to;
      int weight_extra = adj[cur][i].dist;
      if (dist + weight_extra < d[to]) {
        d[to] = dist + weight_extra;
        p[to] = cur;
        pq.push(Edge(to, d[to]));
      }
    }
  }
  return INF;
}

vector<long long> dijkstra_set(vector<vector<edge>> &g, int s,
    vector<int> &p) {
  set<edge> Q;
  vector<long long> d(g.size(), inf);
  p.assign(g.size(), -1);

  d[s] = 0;
  Q.insert(edge(s, 0));
  while (!Q.empty()) {
    int cur = Q.begin()-> to;
    long long dist = Q.begin()-> w;
    Q.erase(Q.begin());
    if (dist > d[cur]) continue;
    for (auto &e : g[cur]) {
      if (d[e.to] > d[cur] + e.w) { // relax
        if (Q.count(edge(e.to, d[e.to]))) {
          Q.erase(edge(e.to, d[e.to]));
        }
        d[e.to] = d[cur] + e.w;
        p[e.to] = cur;
        Q.insert(edge(e.to, d[e.to]));
      }
    }
  }

  return d;
```

```
}

int main() {
  int nodes, edges, tc, s, t, a, b, w;
  cin >> tc;
  while (tc--) {
    cin >> nodes >> edges;

    vector <vector<Edge> > adj(nodes, vector<Edge>());
    vector<int> d(nodes, INF);
    vector<int> p(nodes, -1);

    for (int i = 0; i < edges; i++) {
      cin >> a >> b >> w;
      adj[a-1].push_back(Edge(b-1, w));
    }

    cin >> s >> t;
    int ans = dijkstra(s-1, t-1, adj, d, p);

    if (ans == INF) {
      cout << "NO" << endl;
    } else {
      cout << ans << endl;
    }
  }

  return 0;
}
```

3 miscellaneous

3.1 $\text{set}_b its$

```
#include <bits/stdc++.h>

using namespace std;

// First method. complexity (-)(logn) (theta log n)
int count_set_bits(unsigned int x) {
  int count = 0;
  while (x) {
    count += x & 1;
    x >>= 1;
  }
  return count;
}
// Brian Kernighans Algorithm O(logn)
int set_bits(unsigned int x) {
  int count = 0;
  while (x) {
    x &= (x - 1);
    count++;
  }
  return count;
}
/*
  You are given two numbers A and B. Write a program to count
  number of bits needed to be flipped to convert A to B.
*/
int main() {
  unsigned int a, b;
  cin >> a >> b;
  int a_xor_b = a ^ b;
  cout << "A needs: " << set_bits(a_xor_b) << endl;
  return 0;
}
```

4 $\text{mod}_p ow$

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
const ll mod = 2147483647;

ll power(ll base, ll expo) {
  ll ans = 1;
  while (expo) {
    if (expo & 1) {
      ans *= base;
      ans %= mod;
    }
    expo >>= 1;
```

```
    base = base * base;
    base %= mod;
  }
  return ans;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  ll expo, base;
  cin >> base >> expo;
  cout << power(base, expo) << endl;
  return 0;
}
```

5 strings

5.1 KMP

```
int KMP_matcher(string &pattern, string &text) {
       int m = pattern.size();
       vector<int> border(m);
       border[0] = 0;
       for (int i = 1; i < m; ++i) {
              border[i] = border[i-1];
              while(border[i] > 0 && pattern[i] != pattern[border[i]]) {
                     border[i] = border[border[i] - 1];
              }
              if(pattern[i] == pattern[border[i]])
     border[i]++;
       }

       int n = text.size();
       int ans = 0;
       int seen = 0;
       for (int j = 0; j < n; ++j) {
              while(seen > 0 && text[j] != pattern[seen]) {
                     seen = border[seen - 1];
              }
              if (text[j] == pattern[seen])
     seen++;
              if (seen == m) {
                     ans++;
                     seen = border[m - 1];
              }
       }

       return ans; // number of occurrences
}

int main()
{
       int cases;
       string pattern, text;
       cin>>cases;
       for (int caso = 0; caso < cases; ++caso) {
              int ans = 0;
              cin >> text >> pattern;
   int ans = KMP_matcher(pattern, text);
              cout << "Case " << caso+1 << ": " << ans << endl;
       }
       return 0;
}
```

5.2 matchingAutomata

```
#include <bits/stdc++.h>
#define D(x) cout << #x " = " << (x) << endl

using namespace std;

const string alpha = "abc";

bool match(vector<vector<int> > &P, string &T) {
  int state = 0;
  for (int i = 0; i < T.size(); ++i) {
    state = P[state][T[i] - 'a'];
    D(state);
    D(T[i] - 'a');
    if (state == P.size() - 1) {
      return true;
    }
  }
  return false;
}
```

```
vector<vector<int> > build_automata(string &s) {
  vector<vector<int> > g(s.size() + 1, vector<int> (alpha.size()));

  for (int i = 0; i < s.size(); ++i) {
      string cur = s.substr(0, i);
      for (int j = 0; j < alpha.size(); ++j) {
        string next = cur;
        next.push_back(alpha[j]);
        int best = 0;
        for (int k = 1; k <= next.size(); ++k) {
          string suffix = next.substr(next.size() - k, k);
          string preffix = s.substr(0, k);
          if (suffix == preffix) {
            best = k;
          }
        }
        g[i][j] = best;
      }
  }
  return g;
}

int main() {
  string pattern, text;
  while (cin >> pattern >> text) {
    cout << "p " << pattern << endl;
    cout << "t " << text << endl;
    vector<vector<int> > automata = build_automata(pattern);
    cout << "Automata" << endl;
    for (int i = 0; i < automata.size(); i++) {
      for (int j = 0; j < automata[i].size(); j++) {
        cout << automata[i][j] << " ";
      }
      cout << endl;
    }
    cout << ((match(automata, text))? "Match" : "Not match :(") << endl;
  }
  return 0;
}
```

5.3 useful$_s$*trings*

```
#include <bits/stdc++.h>
using namespace std;

string delete_spaces(string &str) {
  str.erase(remove(str.begin(), str.end(), ' '), str.end());
  return str;
}

string to_str(int a) {
  std::stringstream ss;
  ss << a;
  return ss.str();
}

int to_int(string s) {
  istringstream buffer(s);
  int value;
  buffer >> value;;
  return value;
}

void getline_stuff() {
  // Input: size1 type1 size2 type2 ...
  string line;
  getline(cin, line);
  stringstream ll(line);

  while (ll >> size >> type) {
    cout << size << type;
  }
}

int main() {
  string s;
  getline(cin, s);
  cout << "Before: " << s << endl;
  cout << "After: " << delete_spaces(s) << endl;
  return 0;
}
```