# Nocow Library (additional2)

ヘクト

November 21, 2017

## 1 Aho-Corasick

```
// Description: 複数文字列パターンマッチングオートマトン
// TimeComplexity: O(∑|S|)
// Verifyed: Todo

const int sigma = 26;
inline int convert(char& arg) {
  // 1-indexed
  return arg - 'a' + 1;
}

vector<vector<int>> fg; // 0 faliure otherwise goto
vector<vector<int>> ac;

auto set_union(const vi &a, const vi &b) {
  vector<int> res;
  set_union(begin(a), end(a), begin(b), end(b), back_inserter(res));
  return res;
}

void add_state(){
  fg.push_back(vector<int>(1 + sigma, 0));
  ac.push_back(vector<int>());
}

int build(vector<string> &pattern) {
  fg.clear();
  ac.clear();

  const int root = 1;
  rep(loop, 2) add_state();
  fg[root][0] = root; // root failure

  // Trie
  for (auto &s : pattern) {
    int now = root;
    for (auto &c : s) {
      int i = convert(c);

      if (fg[now][i] == 0) {
        fg[now][i] = int(fg.size());
        add_state();
      }
      now = fg[now][i];
    }
    ac[now].push_back(i);
  }

  // Aho-corasick
  queue<int> q;
  for (int i = 1; i <= sigma; ++i) {
    if (fg[root][i]) {
      fg[fg[root][i]][0] = root;
      int nxt = fg[root][i];
      q.push(nxt);
    } else
      fg[root][i] = root;
  }

  // abc と遷移した時に bc も検知できるようにしている.
  while (!q.empty()) {
    int now = q.front(); q.pop();
    for (int i = 1; i <= sigma; ++i) {
      if (fg[now][i]) {
        int nxt = fg[now][0];
        while (!fg[nxt][i]) nxt = fg[nxt][0];
        fg[fg[now][i]][0] = fg[nxt][i];
        ac[fg[now][i]] = set_union(ac[fg[now][i]], ac[fg[nxt][i]]);
        q.push(fg[now][i]);
      }
```

```
    }
  }
  return root;
}

vector<int> match(int root, string &s, vector<string> &pattern) {
  int now = root;
  vector<int> res(pattern.size(), 0);
  for (auto &c : s) {
    int i = convert(c);
    while (!fg[now][i]) now = fg[now][0];
    now = fg[now][i];
    for (auto &j : ac[now]) res[j]++;
  }
  return res;
}
```

## 2 Heavy-Light-Decomposition

```
template <int V> class HLD {
    int par[V], depth[V], heavy[V];
    int head[V], vid[V], inv[V];

    int dfs(const G& graph, int v, int p) {
        int sz = 1, smax = 0;
        for (auto &e : graph[v]) {
            if (e.to == p) continue;
            par[e.to] = v, depth[e.to] = depth[v] + 1;

            int sub_sz = dfs(e.to, v);
            sz += sub_sz;

            if (smax < sub_sz) {
                smax = sub_sz, heavy[v] = e.to;
            }
        }
        return sz;
    }

    void init(const G& graph) {
        const int n = graph.size();
        fill_n(heavy, n, -1);
        dfs(graph, 0, -1);

        int id = 0;
        rep(h, n) {
            if (par[h] != -1 and heavy[par[h]] == h) continue;
            for (int v = h; v != -1; v = heavy[v]) {
                inv[id] = v, vid[v] = id++, head[v] = h;
            }
        }
    }

    // 頂点属性の for_each
    void for_each_vertex(int u, int v, auto &f) {
        if (vid[u] > vid[v]) swap(u, v);
        f(max(vid[head[v]], vid[u]), vid[v] + 1);
        if (head[u] != head[v]) for_each_vertex(u, par[head[v]], f);
    }

    // 頂点属性の for_each （有向 f の 3 番目の引数には順方向なら 0、逆方向なら 1 が渡される
    void for_each_vertex_directed(int u, int v, auto &f) {
        if (vid[u] > vid[v]) {
            f(max(vid[head[u]], vid[v]), vid[u] + 1, 1);
            if (head[u] != head[v]) for_each_vertex_directed(parent[head[u]], v, f);
        } else {
            f(max(vid[head[v]], vid[u]), vid[v] + 1, 0);
            if (head[u] != head[v]) for_each_vertex_directed(u, parent[head[v]], f);
        }
    }

    // 辺属性の for_each
    void for_each_edge(int u, int v, auto &f) {
        if (vid[u] > vid[v]) swap(u, v);
        if (head[u] != head[v]) {
            f(vid[head[v]], vid[v]);
            for_each_edge(u, parent[head[v]], f);
        } else {
            if (u != v) f(vid[u] + 1, vid[v]);
        }
    }

    // 辺属性の for_each （有向 f の 3 番目の引数には順方向なら 0、逆方向なら 1 が渡される
    void for_each_edge_directed(int u, int v, auto &f) {
        if (vid[u] > vid[v]) {
            if (head[u] != head[v]) {
                f(vid[head[u]], vid[u], 1);
                for_each_edge(par[head[u]], v, f);
            } else {
                if (u != v) f(vid[v] + 1, vid[u], 1);
```

```
            }
        } else {
            if (head[u] != head[v]) {
                f(vid[head[v]], vid[v], 0);
                for_each_edge_directed(u, par[head[v]], f);
            } else {
                if (u != v) f(vid[u] + 1, vid[v], 0);
            }
        }
    }

    // 頂点 u の d 個上の頂点を求める（存在しないなら 0 を返す）
    int ancestor(int u, int d) {
        while (1) {
            if (depth[head[u]] <= depth[u] - d) break;
            d -= depth[u] - depth[head[u]] + 1;
            if (head[u] == 0) return 0;
            u = parent[head[u]];
        }
        return inv[vid[u] - d];
    }

    // 頂点 u と頂点 v の LCA を求める
    int lca(int u, int v) {
        if (vid[u] > vid[v]) swap(u, v);
        if (head[u] == head[v]) return u;
        return lca(u, parent[head[v]]);
    }

    // 頂点 u と頂点 v の距離を求める
    int dist(int u, int v) { return depth[u] + depth[v] - 2 * depth[lca(u, v)];}
};
```

---

## 3　線形連立方程式のメモ

$\mathbf{Ax} = \mathbf{b}$ を $\mathbf{x}$ について解くとき，$\mathbf{A} \in \mathbb{R}^{n \times m}, \mathbf{b} \in \mathbb{R}^n$

- 決定系 $\mathrm{rank}(\mathbf{A}) = n = m$　解が唯一
- 優決定系 $\mathrm{rank}(\mathbf{A}) = m < n$ 解がなし
- 劣決定系 $\mathrm{rank}(\mathbf{A}) = n < m$ 解が複数
- ランク落ち $\mathrm{rank}(\mathbf{A}) < \min(n, m)$ 線形従属になっている方程式を取り除くと、上の 3 つに収束