# EIO
## Graafialgoritmid

Oliver Nisumaa

March 30, 2019

## 1 Effektiivne programmeerimistehnika C++ga

Veel mitte eriti kogenud programeerijad implementeerivad tihti graafide ja nendega seonduvaid algoritme ebaoptimaalsetel viisidel saamata aru, et võiks saada palju paremini. Järgnevad näited ei ole kindlasti ainuõiged viisid graafidega seonduva implmenteerimiseks, kuid kui olete seni kasutand halvemaid implementatsioone, on soovitav mõned neist(võib-olla muudetud kujul) kasutusele võtta.

### 1.1 Graafi esitus mälus(*adjancency list*)

Naabritele võib viidata nii indeksi kui ka pointeriga. Pointeril on see eelis, et siis ei ole vaja teada, mis massiivis vaadeldav graaf üldse asub. Samas pointerid võtavad rohkem mälu ja nede õppekõver on väga järsk ja pikk.

```
struct vertex {
  vector< int > adj;
};

vector< vertex > graph;

int main() {
  int n, m;
  cin >> n >> m;
  graph.resize(n);
  for (int i = 0; i < m; ++i) {
    int a, b;
    cin >> a >> b;  // 0-indexed
    graph[a].adj.push_back(b);
    // only for undirected graphs
    graph[b].adj.push_back(a);
  }
  graph.resize(graph.size() + 10);  // This
  ↪  could break adj-links if using
  ↪  pointers
}
```

```
struct vertex {
  vector< vertex * > adj;
};

int main() {
  int n, m;
  cin >> n >> m;
  vector< vertex > graph(n);
  for (int i = 0; i < m; ++i) {
    int a, b;
    cin >> a >> b;  // 0-indexed
    graph[a].adj.push_back(&graph[b]);
    // only for undirected graphs
    graph[b].adj.push_back(&graph[a]);
  }
}
```

### 1.2 Kordsed servad

Kui suunamata graaf sisaldab kordseid servi, siis on tihti vaja salvestada koos serva sihttippuga ka serva indeks.

```
struct vertex {
  vector< pair< int, int > > adj;  // vertex_idx, edge_idx
};

int main() {
  int n, m;
  cin >> n >> m;
  graph.resize(n);
  for (int i = 0; i < m; ++i) {
    int a, b;
    cin >> a >> b;  // 0-indexed
    graph[a].adj.emplace_back(b, i);
    graph[b].adj.emplace_back(a, i);
  }
}
```

### 1.3 DFS üldises graafis

DFS tasub peaegu alati implementeerida rekursiivselt.

```
struct vertex {
  vector< int > adj;
  bool vis = false;
  void dfs();
};

vector< vertex > graph;

void vertex::dfs() {
  if (!vis) {
    vis = true;  // always first
    // PREORDER calculation
    for (int nxt : adj) {
      graph[nxt].dfs();
    }
    // POSTORDER calculation
  }
}
```

```
struct vertex {
  vector< int > adj;
  bool vis = false;
  void dfs(vector< vertex > &graph) {
    if (!vis) {
      vis = true;  // always first
      // PREORDER calculation
      for (int nxt : adj) {
        graph[nxt].dfs(graph);
      }
      // POSTORDER calculation
    }
  }
};
```

```
struct vertex {
  vector< vertex * > adj;
  bool vis = false;
  void dfs(){
    if (!vis) {
      vis = true;  // always first
      // PREORDER calculation
      for (vertex *nxt : adj) {
        nxt->dfs();
      }
      // POSTORDER calculation
    }
  }
};
```

### 1.4 Kordsed servad

```
struct vertex {
  vector< int > adj;
  bool vis = false;
  void dfs();
};

vector< vertex > graph;

void vertex::dfs() {
  if (!vis) {
    vis = true;  // always first
    // PREORDER calculation
    for (int nxt : adj) {
      graph[nxt].dfs();
    }
    // POSTORDER calculation
  }
}
```

### 1.5 DFS puu peal

Puu peal piisab juba läbitud tippudesse uuesti sattumise vältimiseks kontrollimisest, et me tuldud teed tagasi ei läheks.

```
struct vertex {
  vector< int > adj;
  int idx;
  void dfs(int from);
};

vector< vertex > graph;

void vertex::dfs(int from) {
  // PREORDER calculation
  for (int nxt : adj) {
    if (nxt != from) {
      graph[nxt].dfs(idx);
    }
  }
  // POSTORDER calculation
}
int main() {
  int n, m;
  cin >> n >> m;
  vector< vertex > graph(n);
  for (int i = 0; i < n; ++i) {
    graph[i].idx = i;
  }
  ...
  graph[0].dfs(-1);
}
```

```
struct vertex {
  vector< vertex * > adj;
  void dfs(vertex *from) {
    // PREORDER calculation
    for (vertex *nxt : adj) {
      if (nxt != from) {
        nxt->dfs(this);
      }
    }
    // POSTORDER calculation
  }
};
...
graph[0].dfs(NULL);
```

## 2 tsükli avastamine

Lisame igale tippule kaks *bool* tüüpi muutujat: "Kas me oleme sellest tippust DFS-i teinud?"(vis eelnevates koodides), "Kas tipp asub hetkel DFS ahelal(*call stack*-il)?". Kui jõuame DFS-iga tippu, mis juba asub DFS ahelal, siis oleme leidnud tsükli. Suunamata sidusas graafis piisab, kui alustada DFS-i suvalisest tippust. Ülejäänud graafides tuleb proovida alustada igast tipust(Miks?). Miks see algoritm leiab tsükli alati üles?

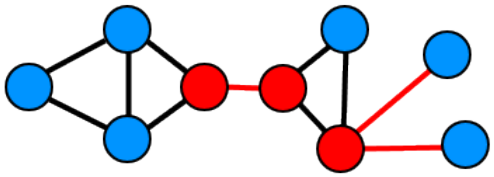
## 3 topoloogiline sorteerimine

Suunatud graafi jaoks on tippude lineaarne jäjestus topoloogiline sorteering parajasti siis, kui ei leidu ühtegi serva, mis oleks selle järjestusse järgi suuremast tipust väiksemasse.

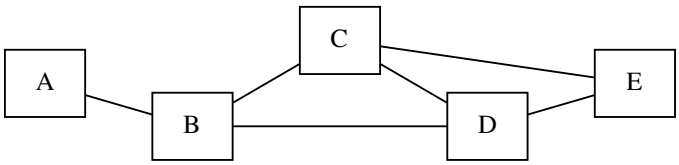
## 4 Sillad ja artikulatsiooni punktid

Need on defineeritud sidusa suunamata graafi jaoks. Sild on serv, pärast mille eemaldamist ei ole graaf sidus. Artikulatsiooni punkt on tipp, pärast mille eemaldamist ei ole graaf sidus.

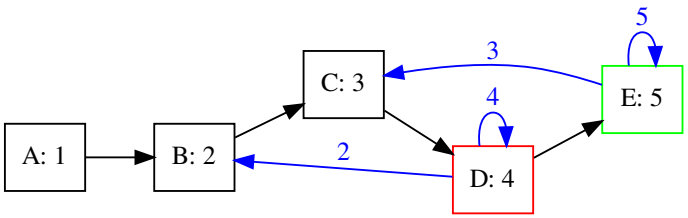
Järgnevas näites on Artikulatsiooni punktid punased tippud ja sillad punased servad.



Lisame igasse tippu kaks arvu - mitmendana me sellese tippu DFS käigus jõudsime(*preoder* indeks), mis on vähim *preoder* indeks, mida me alampuus DFSi tehes nägime. Vaatleme nende väärtuste arvutamist järgnevas graafis.




Mustad nooled ja numbrid tähistavad DFSi (*preoder*)järjekorda. Sinised nooled tähistavad, milliseid *preoder* indekseid me tipu $D$ alampuus $\{D, E\}$ nägime.




Kontrollimiseks, kas mingi serv on sild, piisab kui vaadata serva otspunktides olevaid arvude paare(4 arvu). Mis on täpne tingimus? Miks see leiab kõik sillad? Miks see ühtegi muud serva ei leia?

Kontrollimiseks, kas mingi tipp on artikulatsiooni punkt, piisab kui vaadata eraldi kõigi sealt väljuvate servade otspunktides olevaid arvude paare(iga väljuva serva jaoks 4 arvu). Mis on täpne tingimus? Miks see leiab kõik artikulatsiooni punktid? Miks see ühtegi muud punkti ei leia?

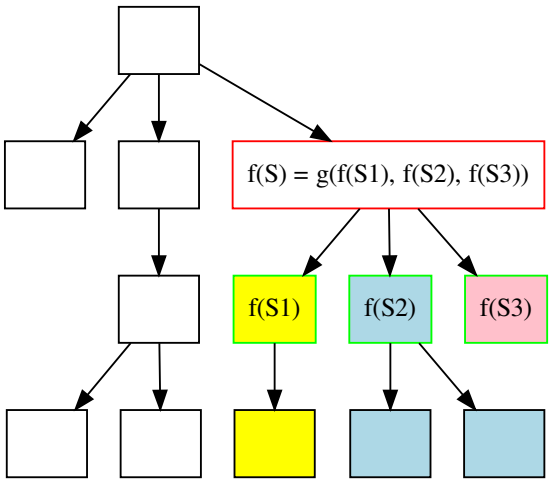
## 5 Hulkade ühendamine

Oletame, et on vaja DSU stiilis töödelda käske kujul ühenda kaks suvalist hulka. Erinevalt DSU-st ei küsita, mis hulgas mingi element asub, vaid on reaalselt vaja teada, mis elementides mingi hulk koosneb. Kui meile piisab hulgate esitamisest lingitud loendina, siis saame kaks hulka ühendada konstantse ajaga.

Vaatleme juhte, kus meile sellest ei piisa. Kui teha seda naiivselt ja lihtsalt tõsta elemendid iga ühendamise käigus kokku võib kõikide elementide ühendamine võtta $\mathcal{O}(n^2)$ aega. Aga teeme kavalamalt: Kahe hulga ühendamisel liikutame konstantses ajas suurema alamhulga uue hulga kohale ja siis lisame sinna lineaarses ajas elemendid väiksemast hulgast. Mis on selle keerukus? Mis saab siis, kui meil on vaja hoida elemente sorteeritult ja esitame hulki *set*-ina?

## 6 Kompsik

Oletame, et me tahame kogu puu või kõigi selle alampuude jaoks mingisugust väärtust arvutada. Üks variant selle tegemiseks on leida mingi väärtus, mida me oskame effektiivselt arvutada iga alampuu jaoks kasutades väärtusi selle alampuu lastel. Järgneval joonisel arvutatatkse väärtus alampuu jaoks, mis on juuritud punases tippu, kasutades funktsiooni $g$ ja funktsiooni väärtusi selle lastel(rohelised).




Keerulisemates olukordades ei saa "väärtust" alampuu jaoks taandada konstansele suurusele ja selle jaoks tuleb kasutada näiteks massivi, lingitud loendit, kahendpuud(*set/map/priority_ queue*) või räsitabelit, mõnikord koguni mitut erinevat struktuuri.

Näiteks oletame, et meil on juuritud puus osad tippud sinist ja teised punast värvi. Saame iga serva värvida kas punast või sinist värvi. Teatud värvi serva mööda oskame sama värvi punkte ülespoole liigutada. Kui viime punane ja sinine punkti kokku, siin need annhileeruvad. Kui punktidel on lisaks värvile veel muid omadus, siis ei saa hoida lihtsalt ühte arvu, kui palju meil alampuus ühte või teist värvi punkte on, vaid peame neid hoidma mingis struktuuris.