Netafræði

Keppnisforritunarlegur inngangur

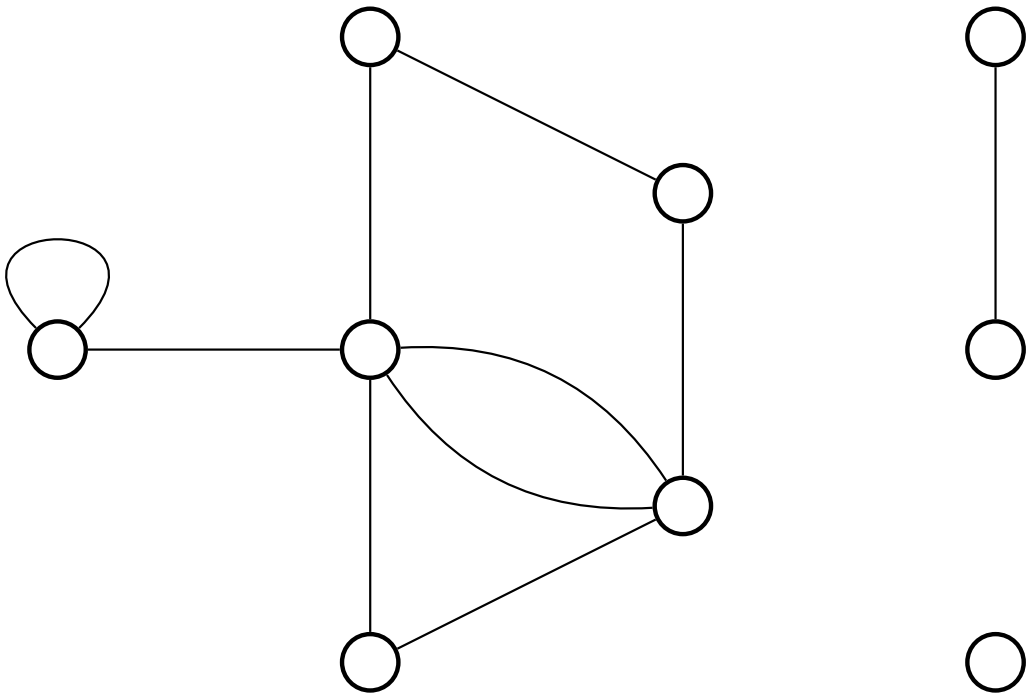
Atli Fannar Franklín

28. febrúar 2022

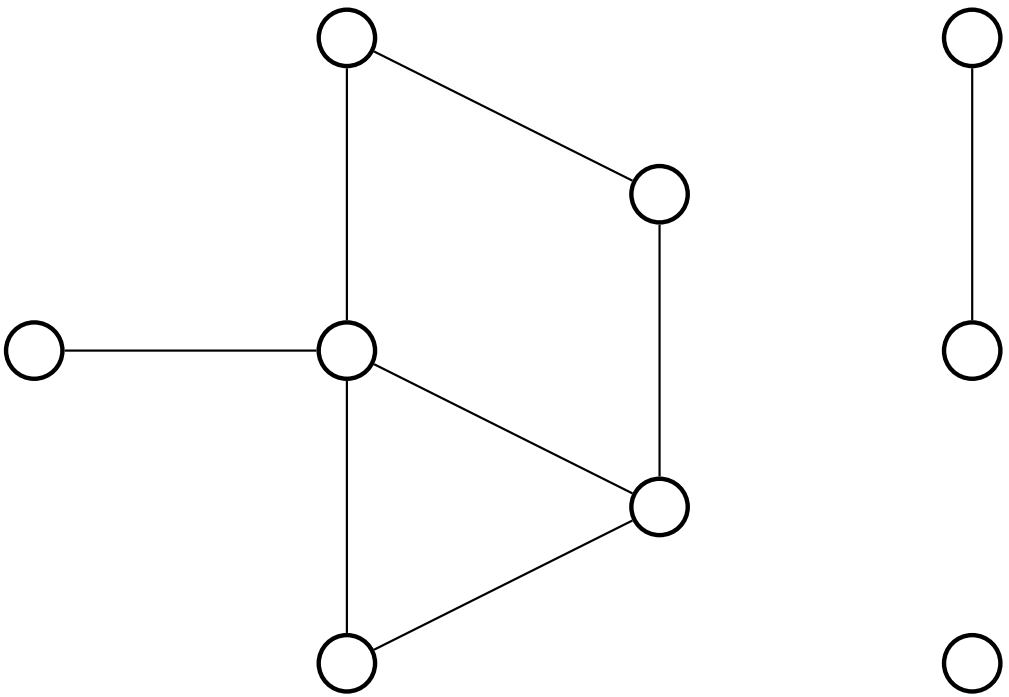
Hvað er net?

- Þetta er mengi af *hnútum* sem tengdir eru með *leggjum* sem liggja frá einhverjum hnút í einhvern hnút.
- Formlega séð er net þrennd (V, E, ϵ) þar sem V er mengi hnúta, E er mengi leggja og ϵ er vörpun úr E í $\mathcal{P}(V)$ þ.a. $|\epsilon(v)| = 1$ eða 2 sem úthlutar hverjum legg *endapunktum* sínum.
- Ef hnútar hafa legg á milli sín köllum við þá nágranna. Ef tveir leggir hafa sameiginlegan endapunkt köllum við þá aðlæga. Einnig segjum að endapunktur sé aðlægur legg sínum. Leggur með aðeins einn endapunkt kallast snara. Net þar sem engir tveir leggir hafa sömu endapunkta og hefur engar snörur kallast *einfalt*.
- Langoftast vinnum við með einföld net.

Dæmi um (ekki einfalt) net



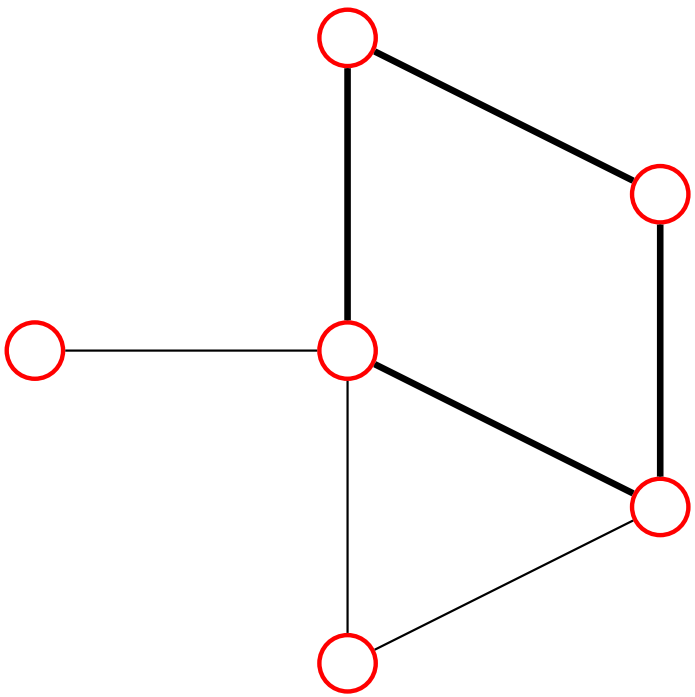
Dæmi um (einfalt) net

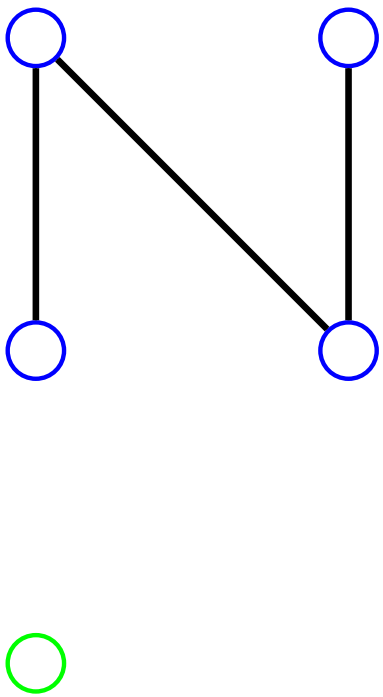


Vegir og rásir

- Runa leggja þar sem hver leggur hefur annan endapunkt sameiginlegan með þeim sem kom á undan og hinn sameiginlegan með þann sem kemur á eftir kallast *vegur*. Ef vegurinn byrjar og endar á sama stað köllum við hann *rás*. Ef við endurtökum enga hnúta köllum við veginn einfaldan. Rás er einföld ef við endurtökum enga leggi.
- Við segjum að tveir hnútar séu tengdir ef til er vegur milli þeirra. *Samhengisþáttur* í neti er þá óstækkanlegt hlutmengi hnúta sem eru allir innbyrðis tengdir.
- Skilgreina má *tré* á hundrað vegu, en hér skulum við bara skilgreina það sem samanhangandi net án rása. Net án rása (sem er þá ekki nauðsynlega samanhangandi) er þá kallað *skógur*.

Dæmi um samhengisþætti, veg og rás

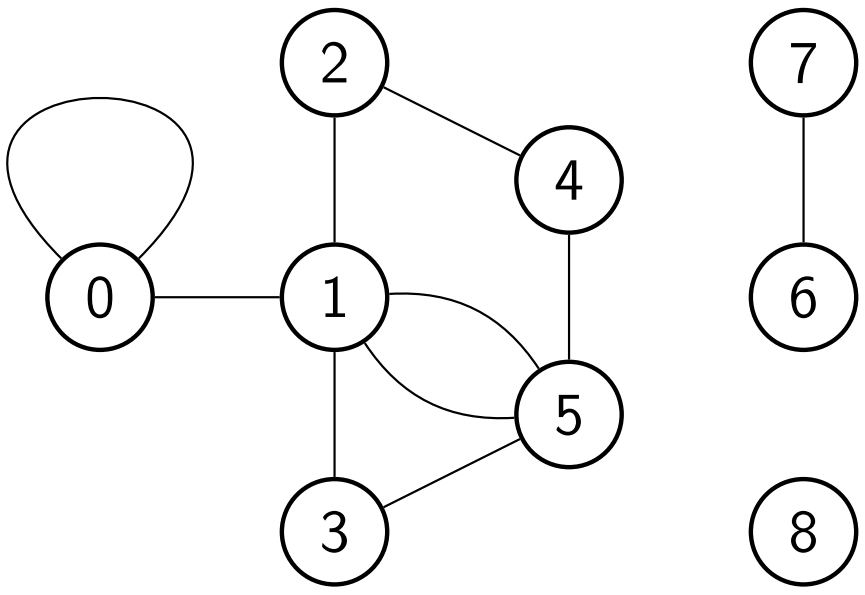




Framsetning í tölvu

- Við höfum þrjár leiðir til að tákna almenn net í tölvu (til eru einhverjar exótískari útgáfur eins og link-cut tree og heavy-light decomposition en við látum þessar þrjár duga).
- Við höfum þá nágrannaframsetningu, fylkjaframsetningu og listaframsetningu fyrir almenn net. Notum nágrannaframsetninguna langmest. Sjáum nú þessar framsetningar fyrir netið sem við sáum áðan.

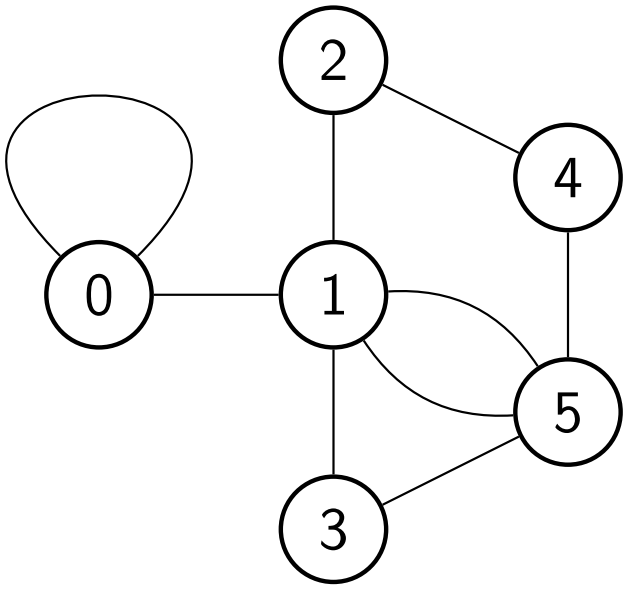
Nágrannaframsetning (adjacency list)

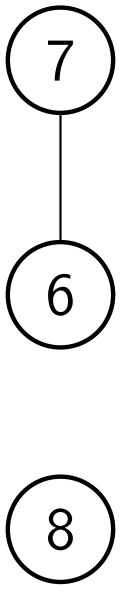



```
G = [
      [0, 1],
      [0, 2, 3, 5, 5],
      [1, 4],
      [1, 5],
      [2, 5],
      [1, 1, 3, 4],
      [7],
      [6],
      []
]
```

- `G` er þá fylki/vigur vigra og `G[i]` er vigur með nágranna `i`.

Fylkjaframsetning (adjacency matrix)

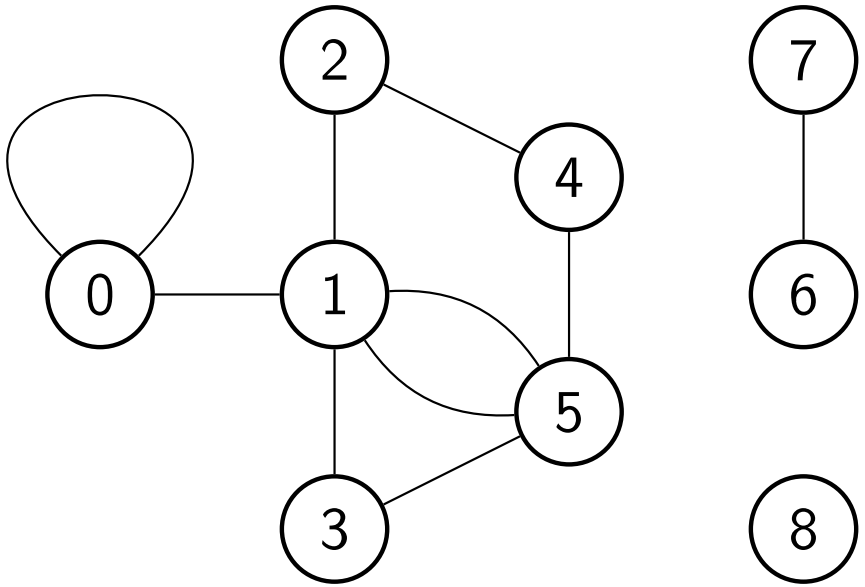





```
G = [
    [1, 1, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 1, 1, 0, 2, 0, 0, 0],
    [0, 1, 0, 0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0],
    [0, 2, 0, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

- `G` er þá tvívítt fylki og `G[i][j]` er fjöldi leggja milli `i` og `j`.

Listaframsetning (edge list)



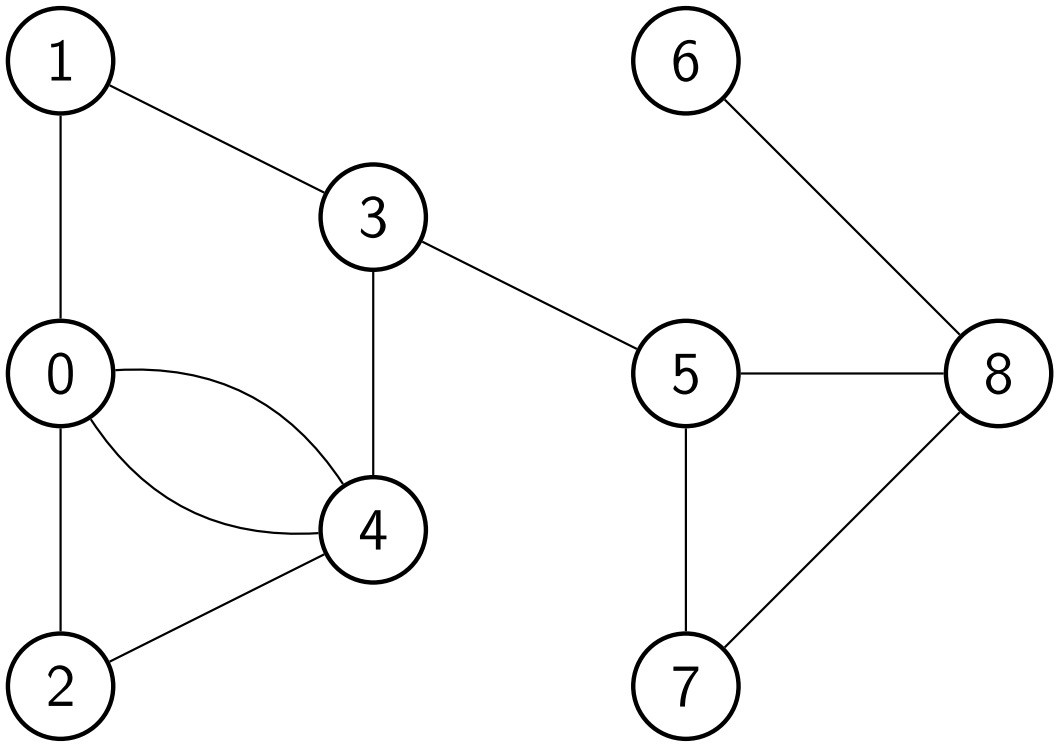

```
G = [
      [0, 0],
      [0, 1],
      [1, 2],
      [1, 3],
      [1, 5],
      [1, 5],
      [2, 4],
      [3, 5],
      [4, 5],
      [6, 7]
]
```

- `G` er þá vigur/fylki para og hvert par `[i, j]` segir að til sé leggur milli `i` og `j` í netinu.

Netaleitir

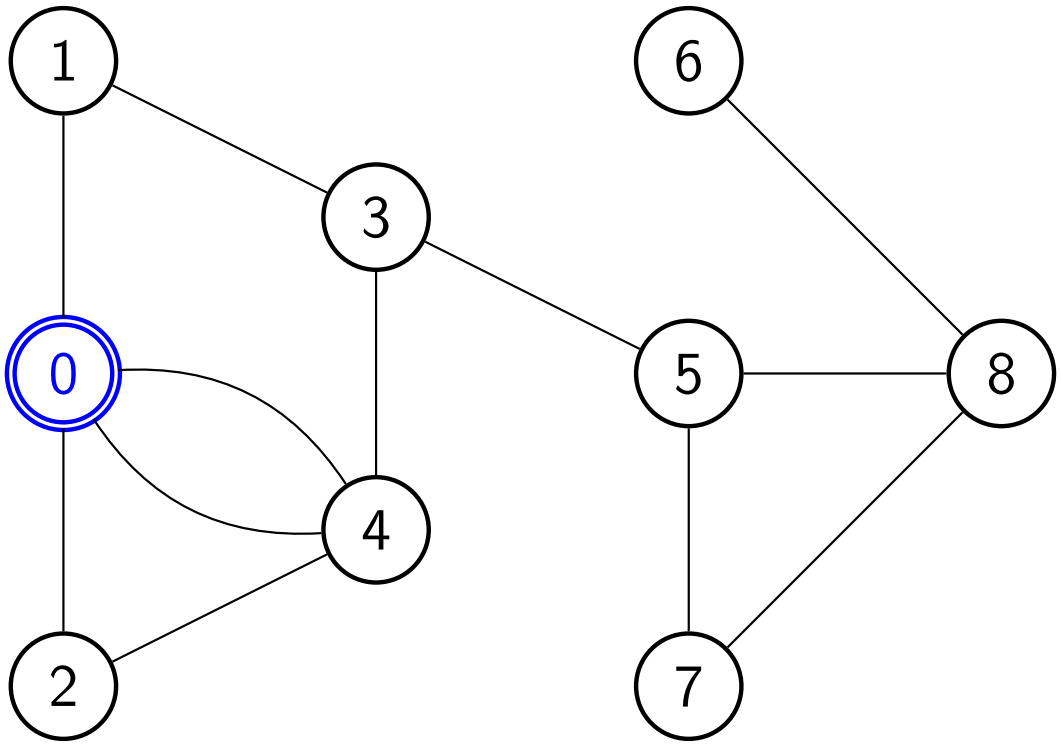
- Mörg dæmi fela það í sér að leita að einhverju í neti eða að ítra í gegnum hnúta nets. Hvernig má gera þetta?
- Fyrir svona net (sjáum meira í næstu viku fyrir aðrar tegundir af netum) þá höfum við tvær leitaraðferðir. Við köllum þær breiddarleit (Breadth-first search, BFS) og dýptarleit (Depth-first search, DFS).
- Bæði fela það í sér að byrja bara einhverstaðar og fara svo endurtekið í alla nágrannana nema þá sem maður er búinn með. Munurinn felst í því í hvaða röð þetta er gert. Í BFS hendum við nágrönnunum í biðröð en í DFS hendum við þeim á hlaða.
- Tímaflækja reikniritanna beggja er $\mathcal{O}(V + E)$ þar sem V er fjöldi hnúta og E er fjöldi leggja.
- Skoðum nú dæmi um hvernig BFS og DFS fara með ólíkum hætti í gegnum sama netið.

BFS



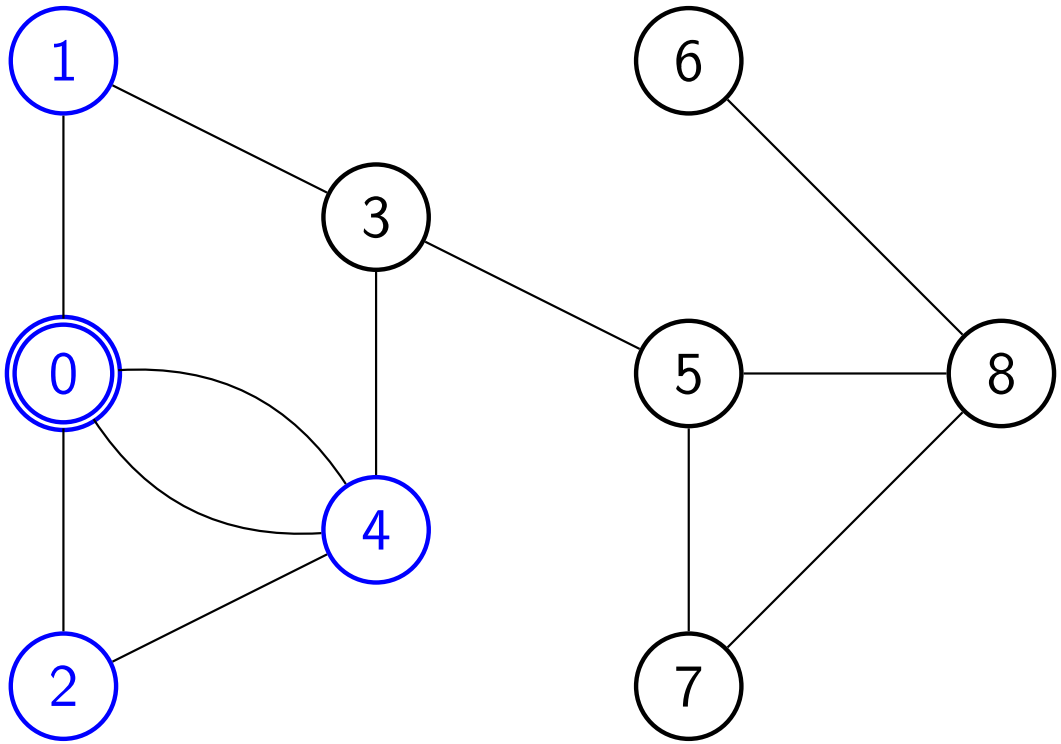

```
q = [
  0
]
```

BFS



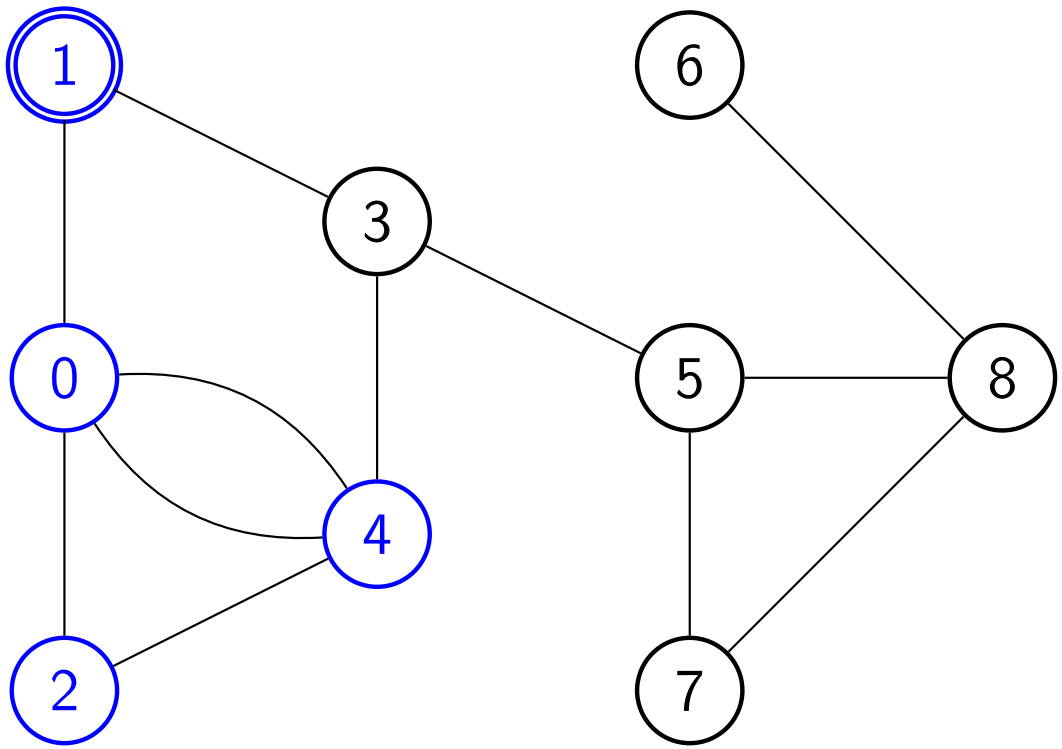

```
q = [
]
```

BFS



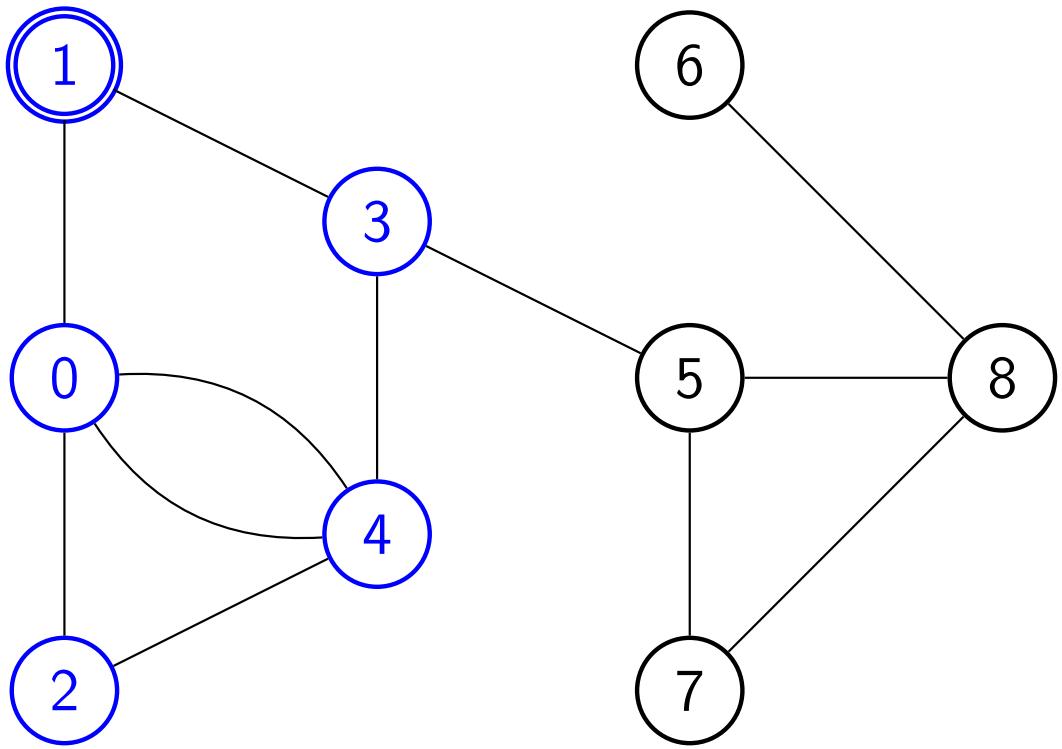

```
q = [
 1,
 2,
 4
]
```

BFS



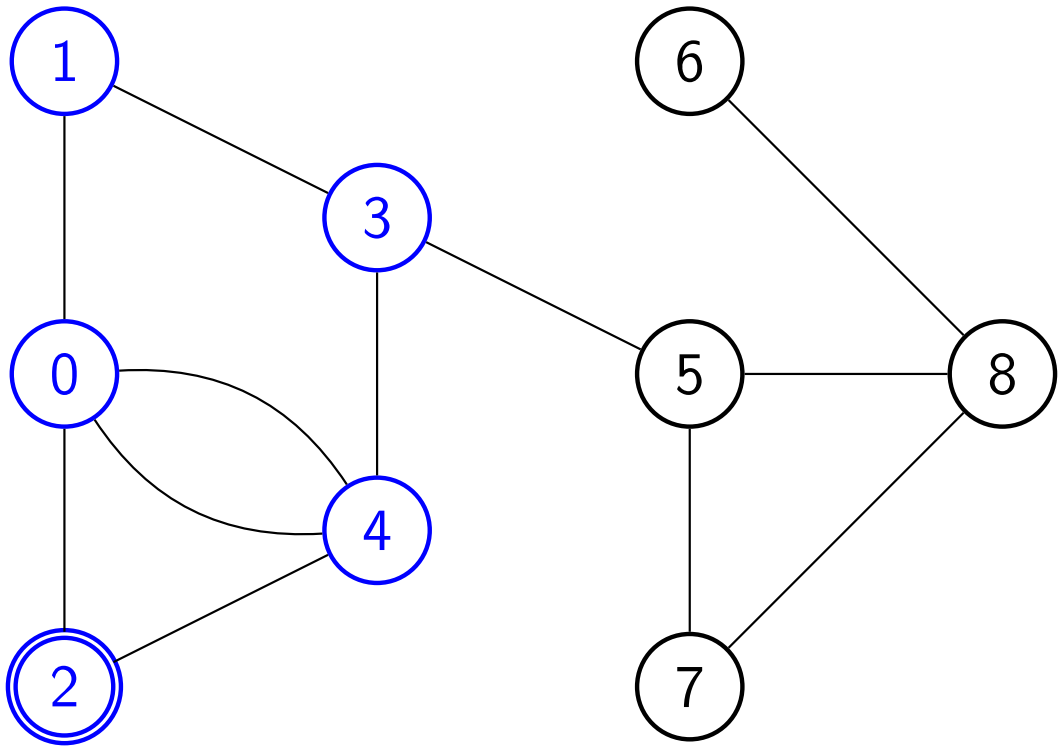

```
q = [
 2,
 4
]
```

BFS



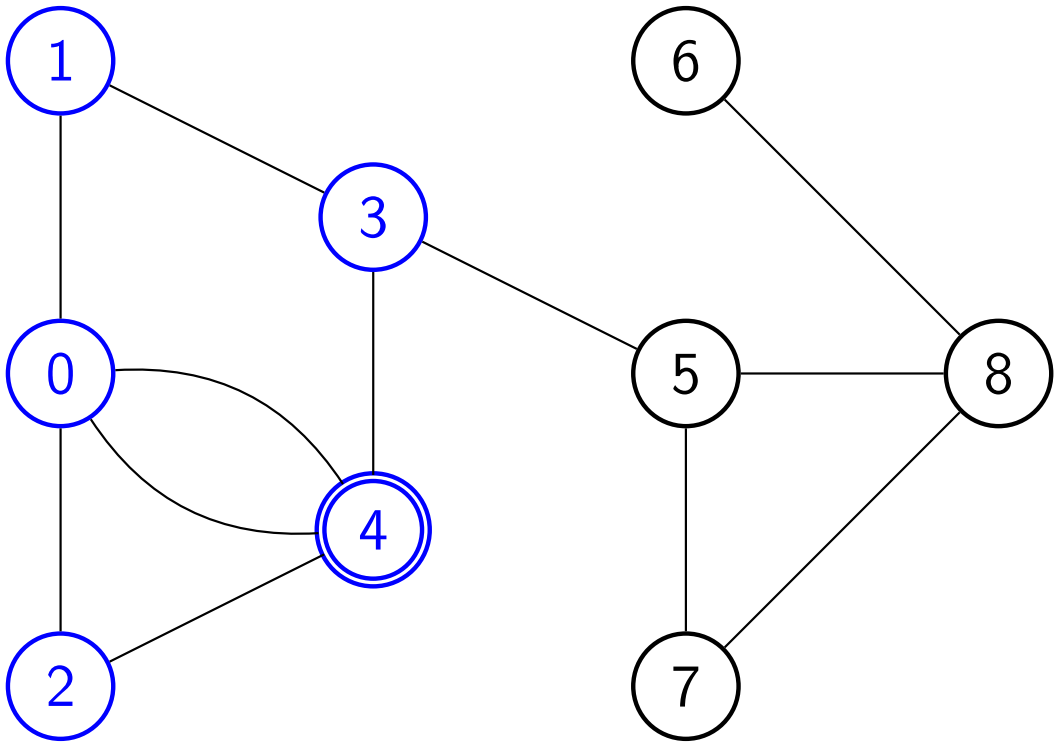

```
q = [
 2,
 4,
 3
]
```

BFS



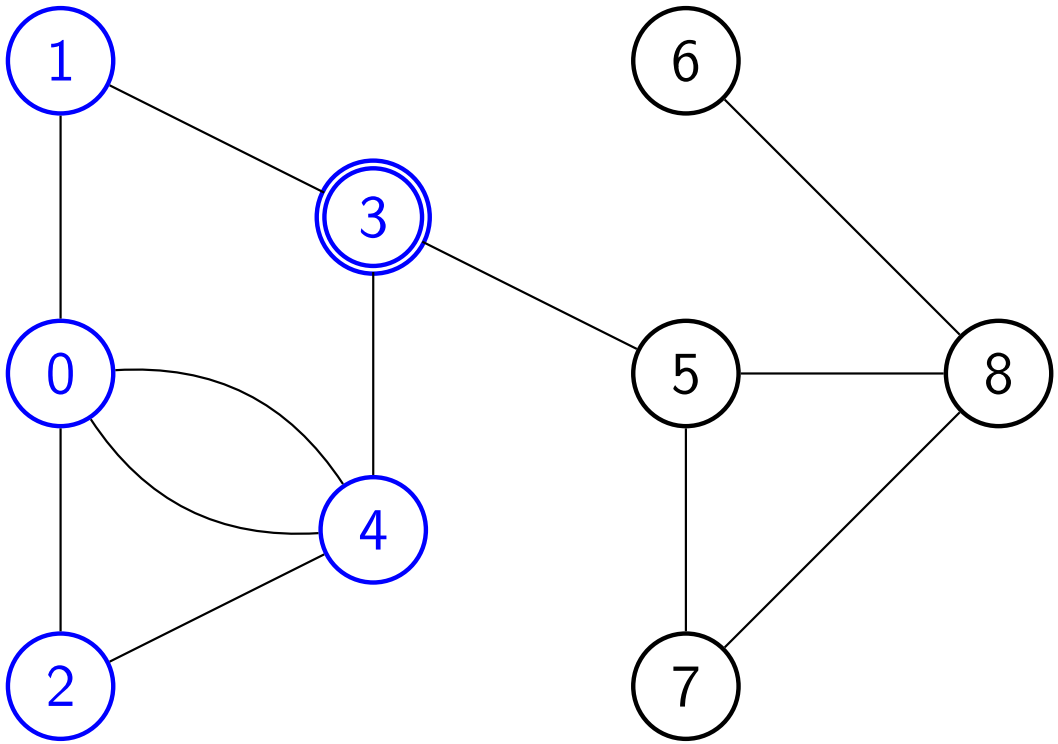

```
q = [
 4,
 3
]
```

BFS



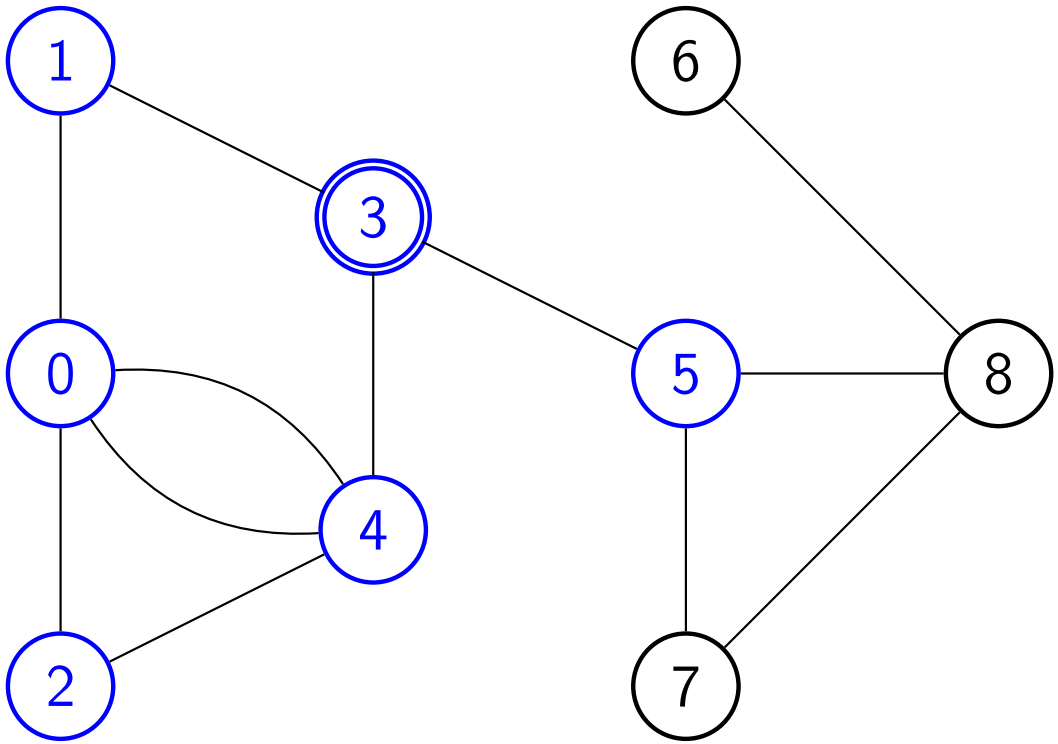

```
q = [
 3
]
```

BFS



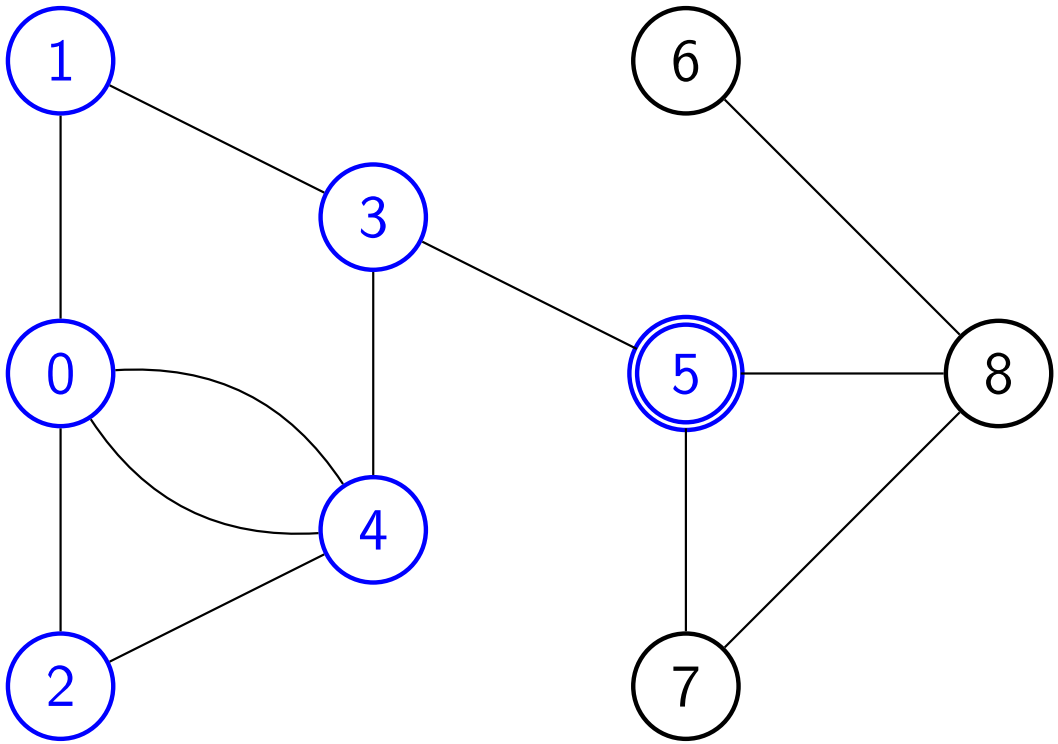

```
q = [
]
```

BFS



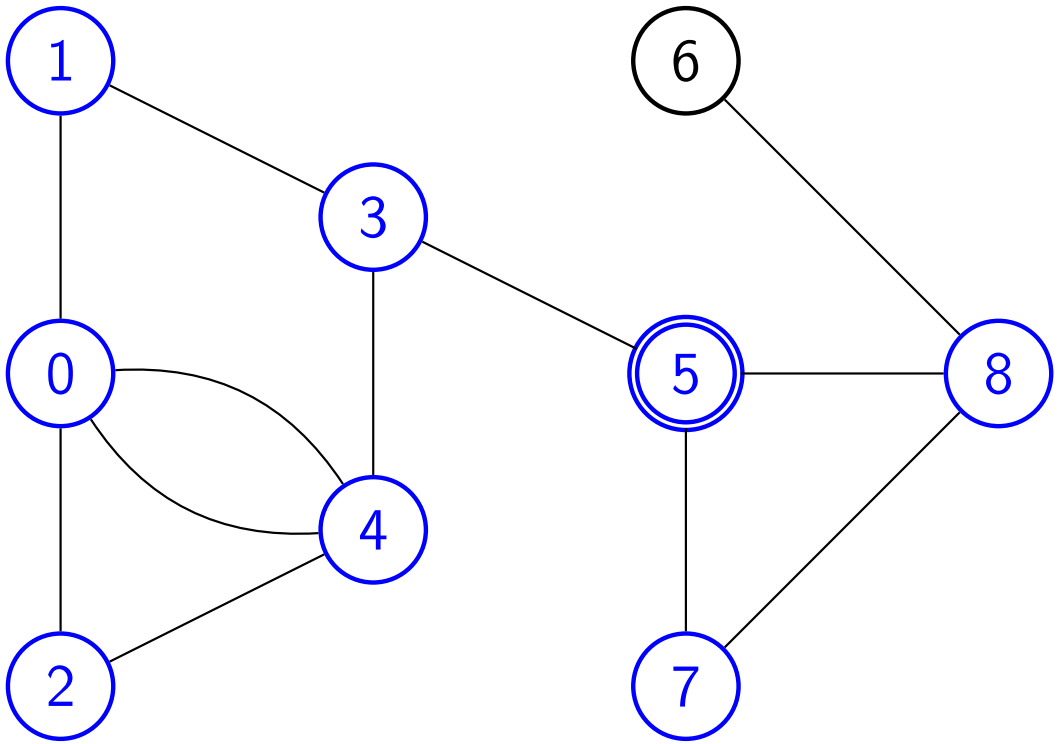

```
q = [
 5
]
```

BFS



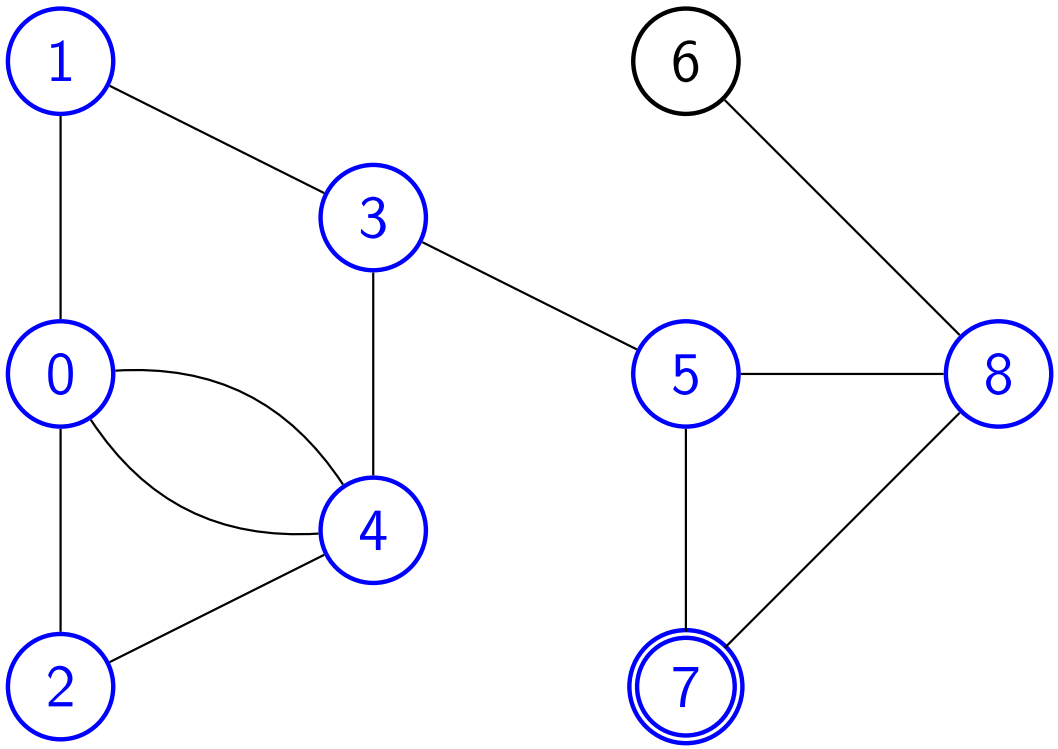

```
q = [
]
```

BFS



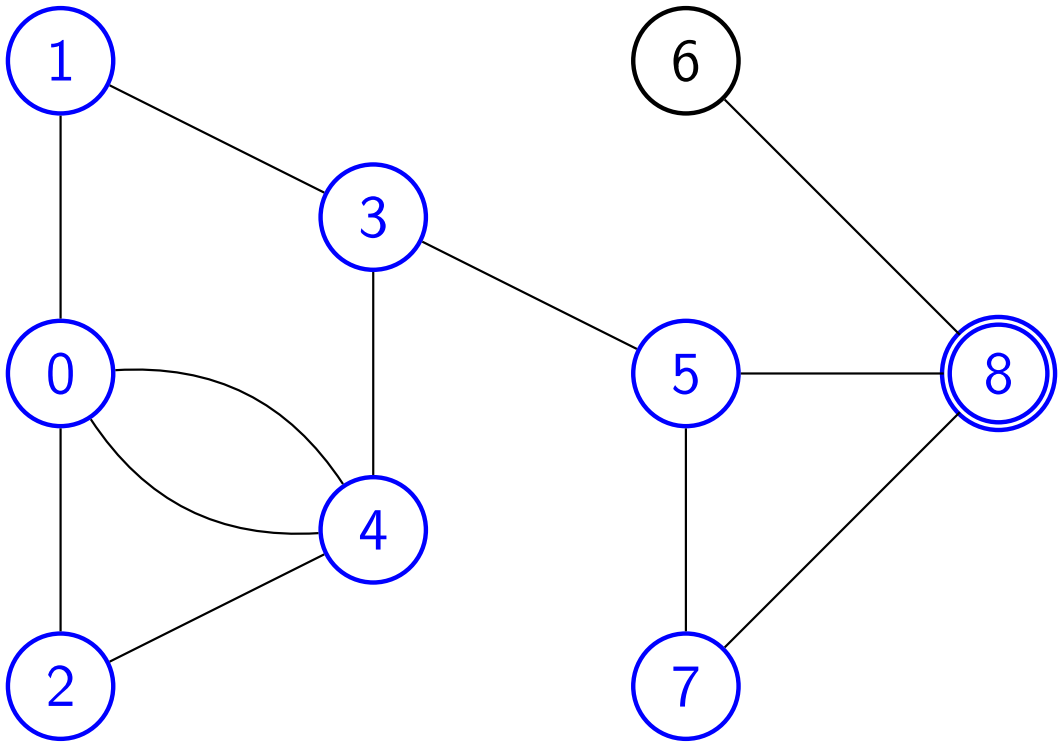

```
q = [
 7,
 8
]
```

BFS



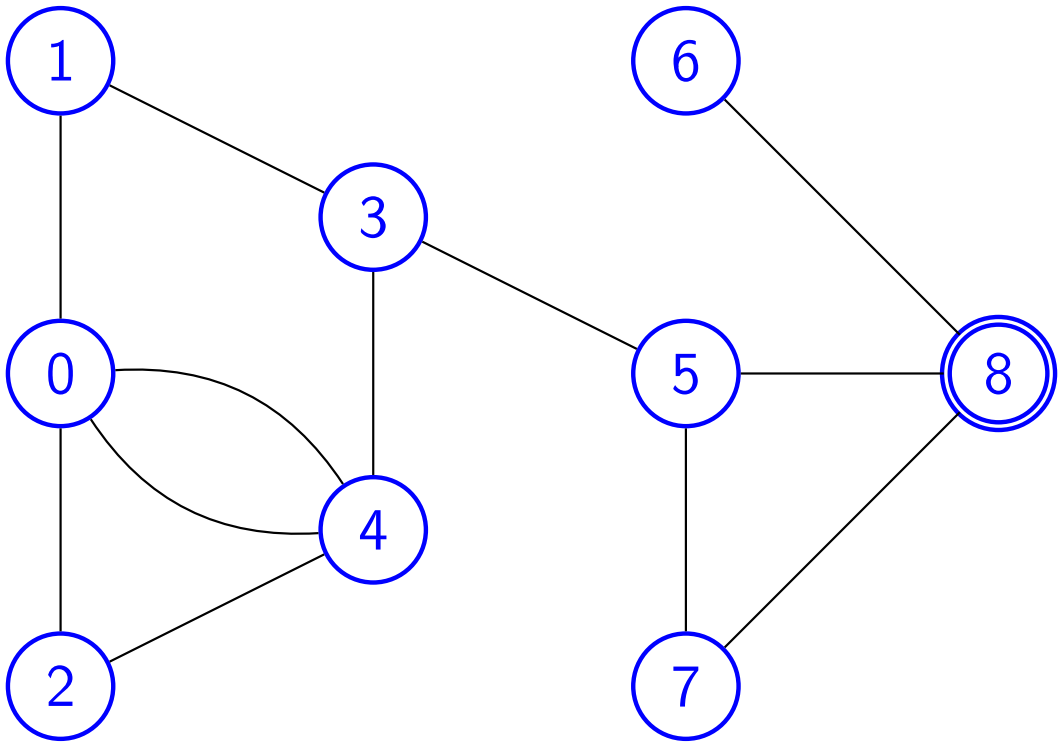

```
q = [
 8
]
```

BFS



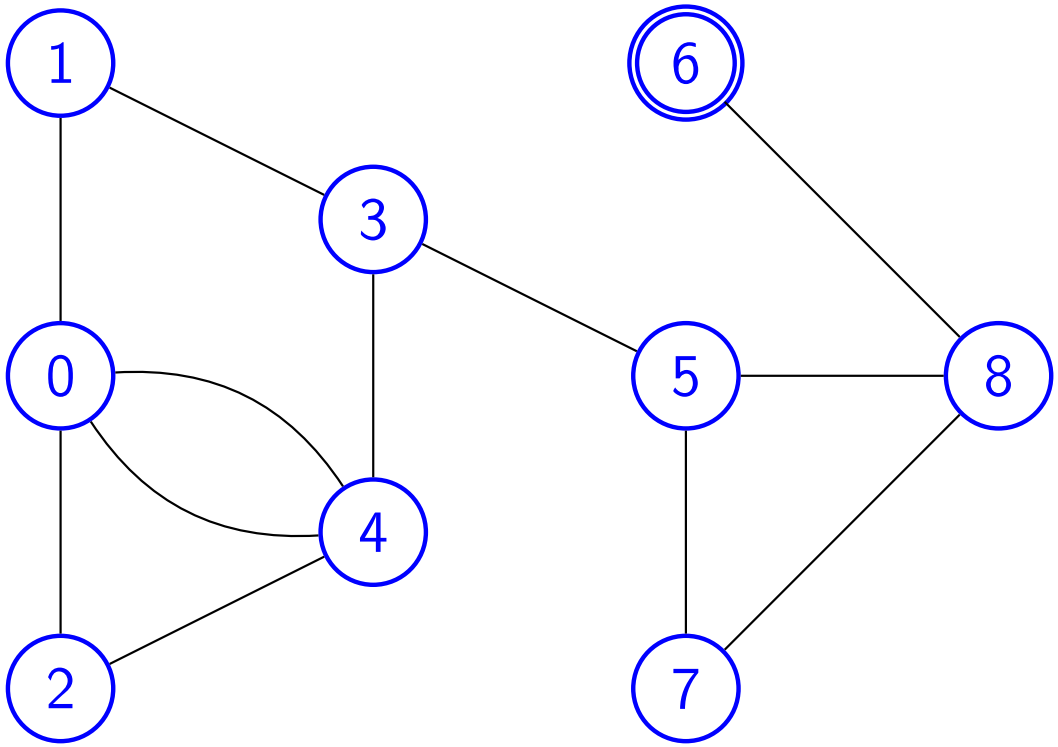

```
q = [
]
```

BFS



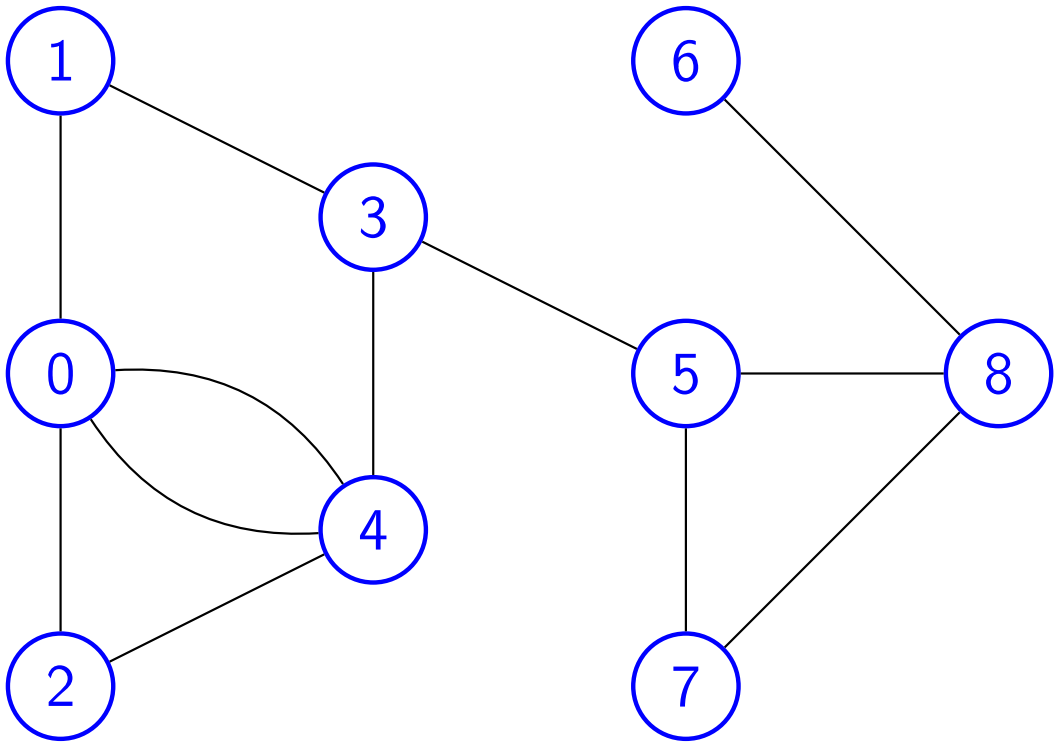

```
q = [
 6
]
```

BFS




```
q = [
]
```

BFS

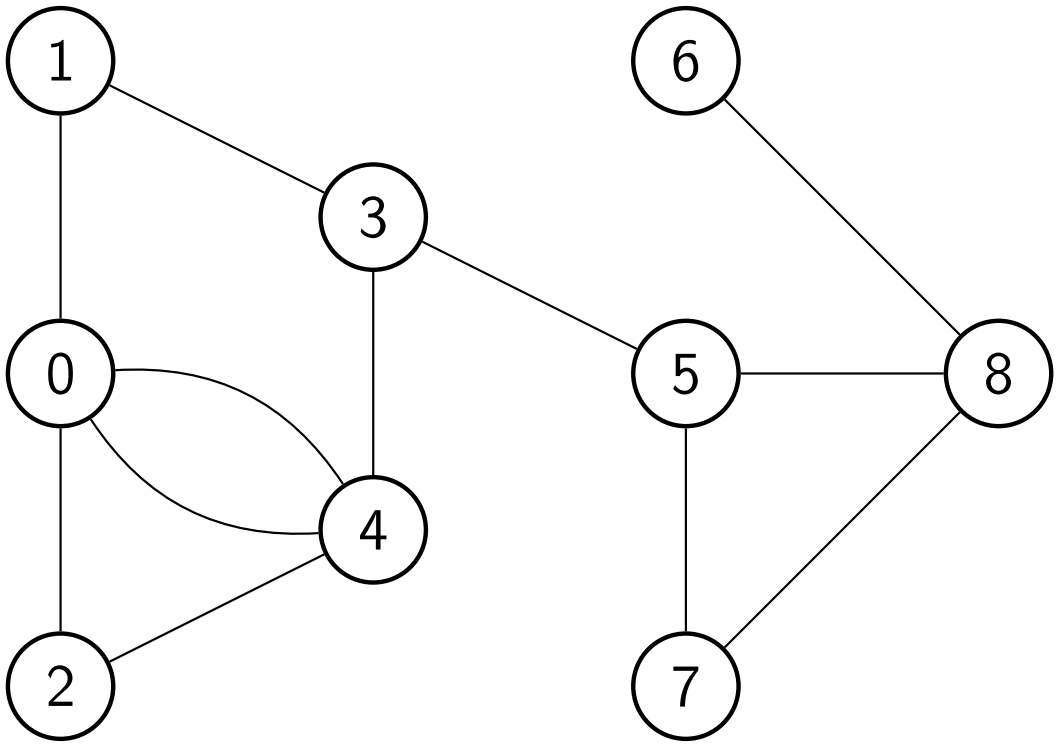



```
q = [
]
```

BFS útfærsla með nágrannalista

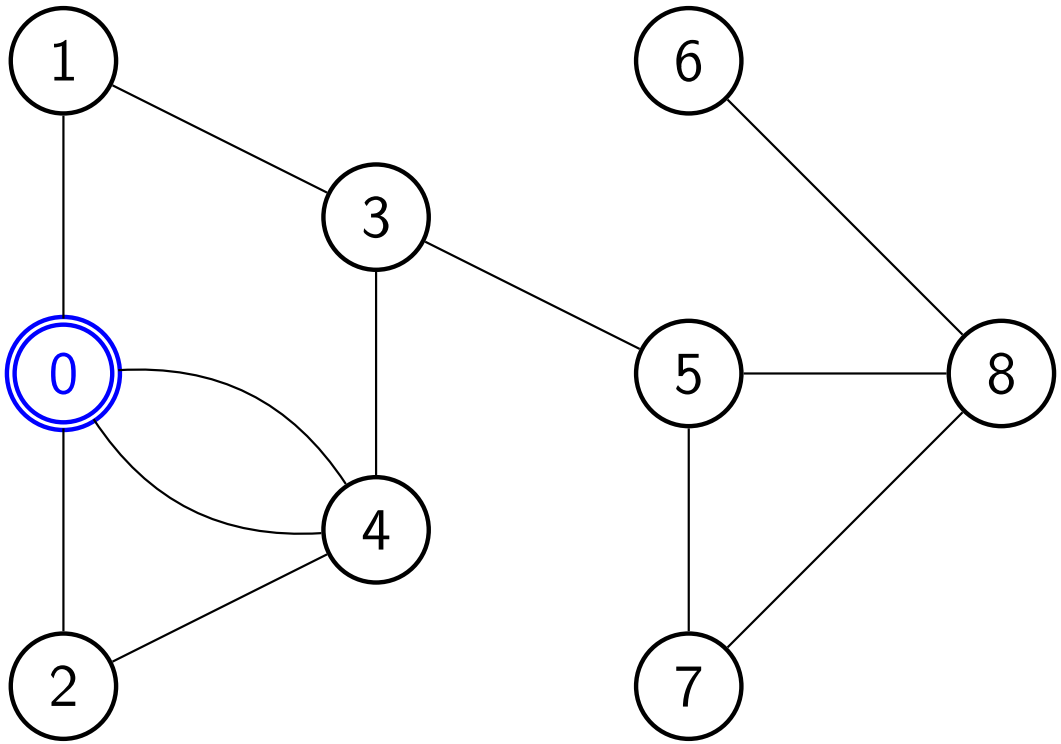
```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;

int main() {
    int n, m, a, b;
    cout << "Gefðu upp fjölda hnúta og leggja\n";
    cin >> n >> m;
    cout << "Gefðu upp m leggi (0-index)\n";
    vvi g(n, vi());
    for(int i = 0; i < m; ++i) {
        cin >> a >> b;
        g[a].push_back(b), g[b].push_back(a);
    }
    queue<int> q; vector<bool> d(n, false);
    q.push(0); d[0] = true;
    cout << "Í samhengisþætti 0 fundust:\n";
    while(!q.empty()) {
        int cur = q.front(); q.pop();
        cout << cur << '\n';
        for(int x : g[cur]) {
            if(d[x]) continue;
            d[x] = true; q.push(x); }}}
```

DFS



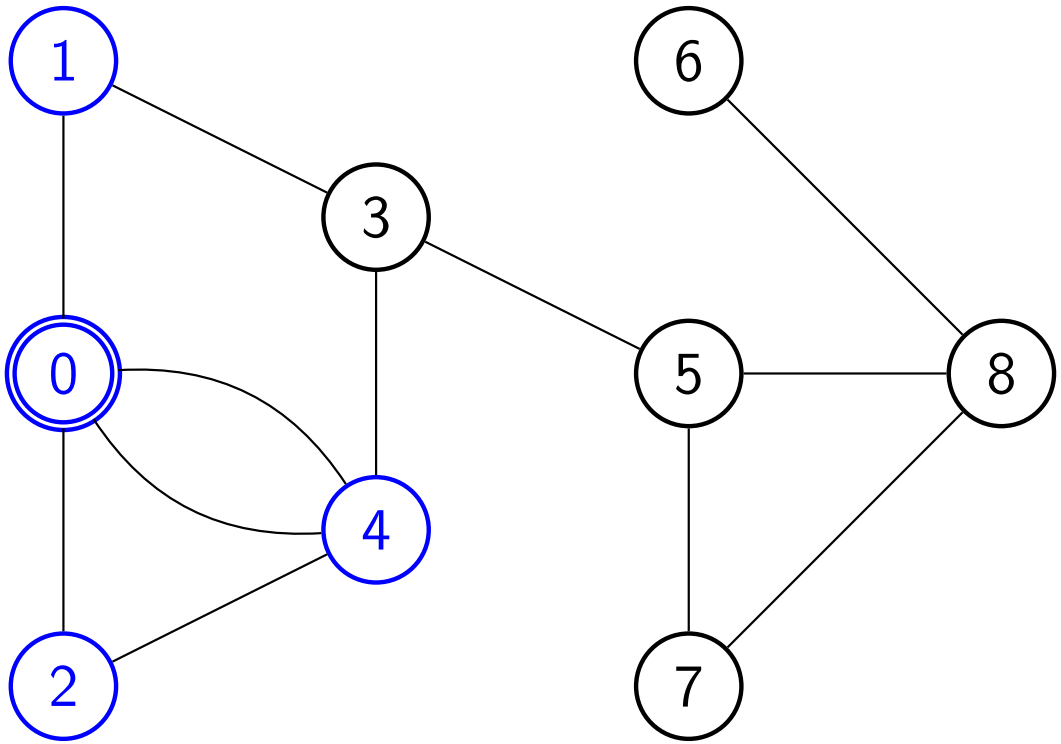

```
s = [
  0
]
```

DFS



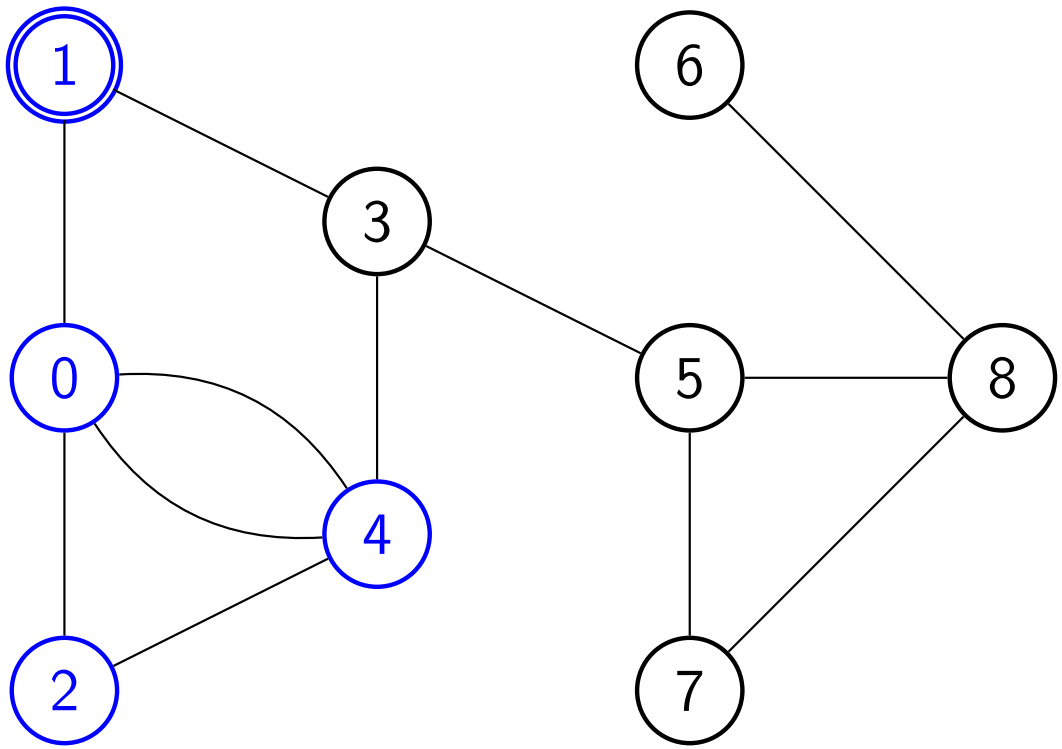

```
s = [
]
```

DFS



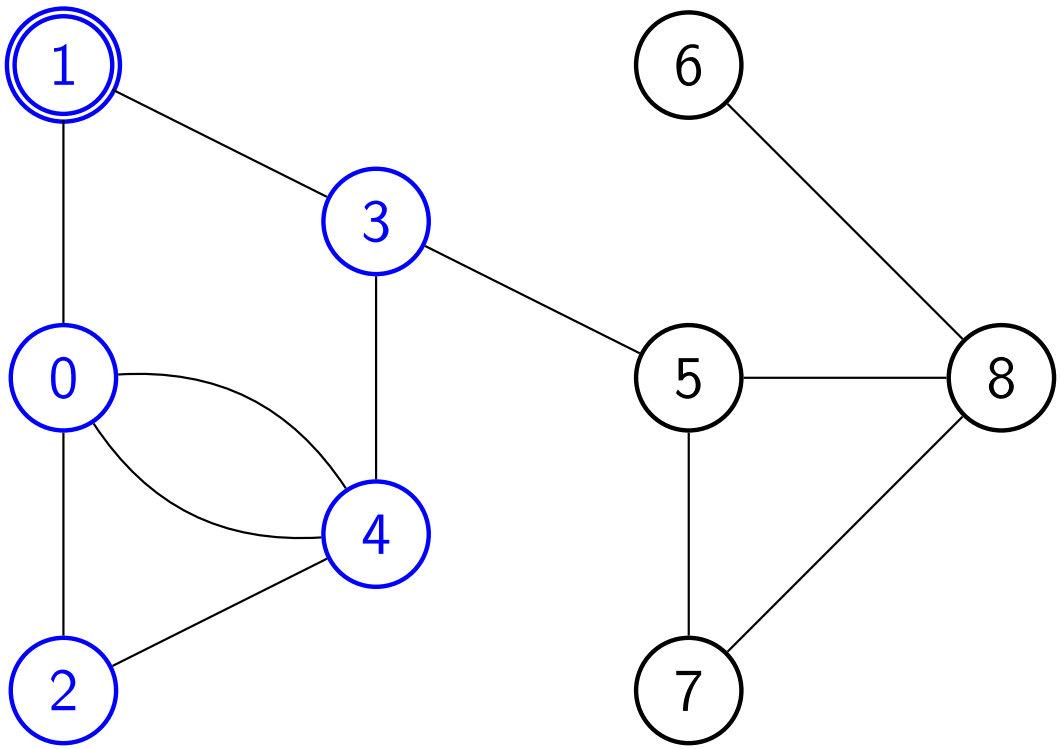

```
s = [
 1,
 2,
 4
]
```

DFS



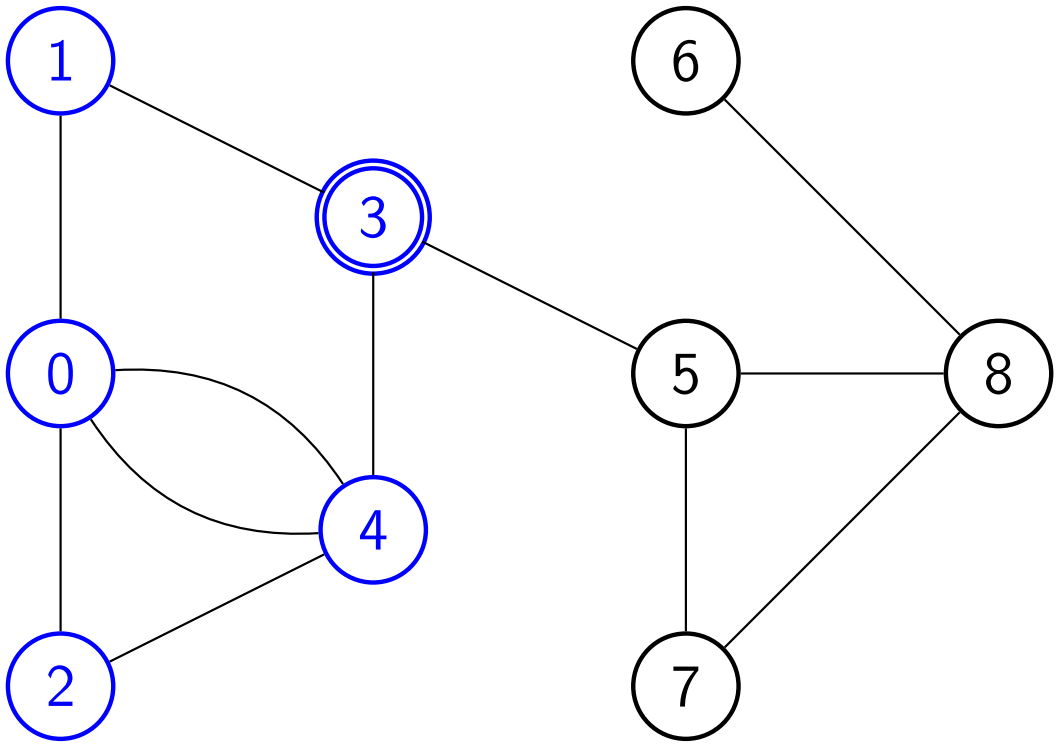

```
s = [
 2,
 4
]
```

DFS



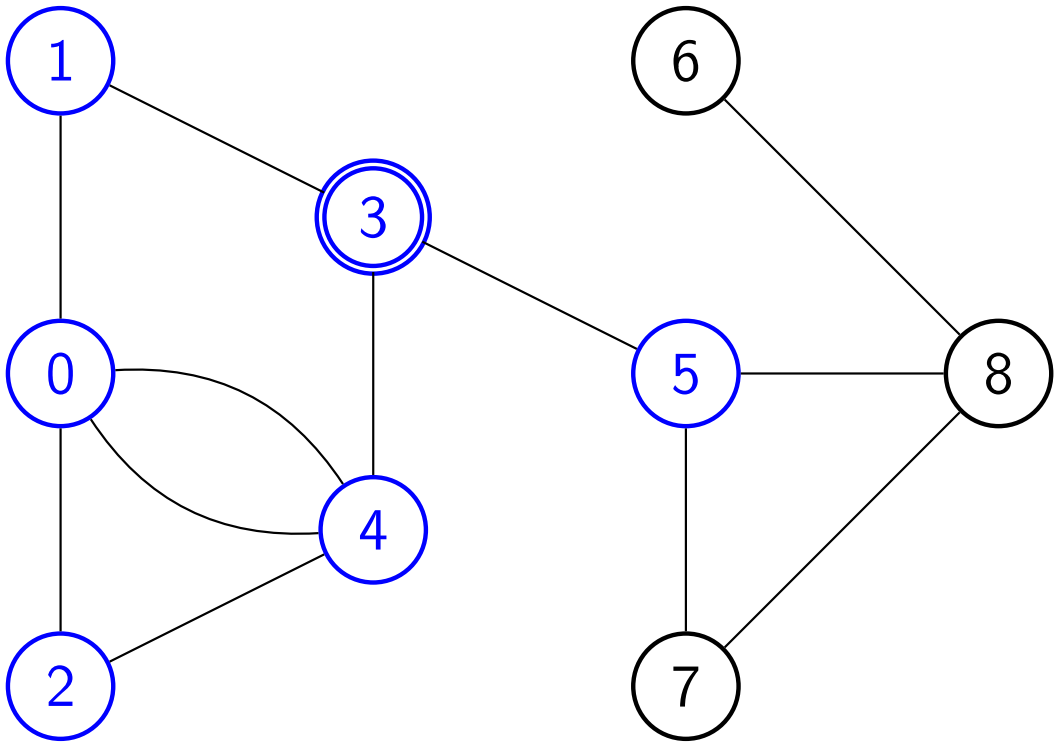

```
s = [
 3,
 2,
 4
]
```

DFS



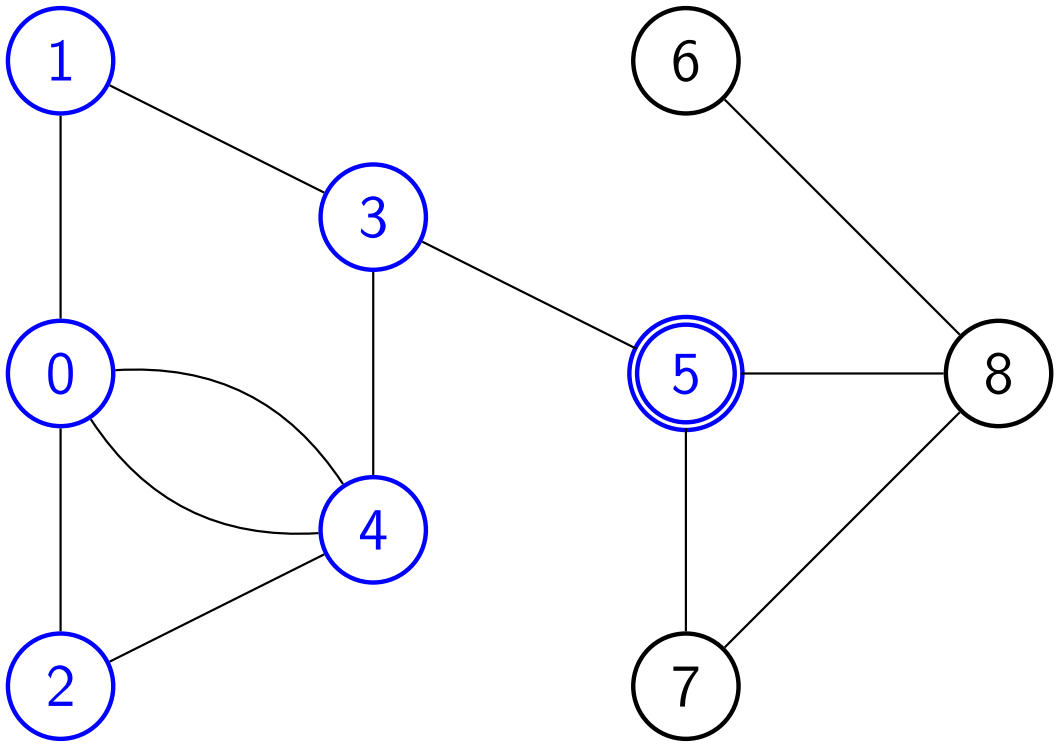

```
s = [
 2,
 4
]
```

DFS



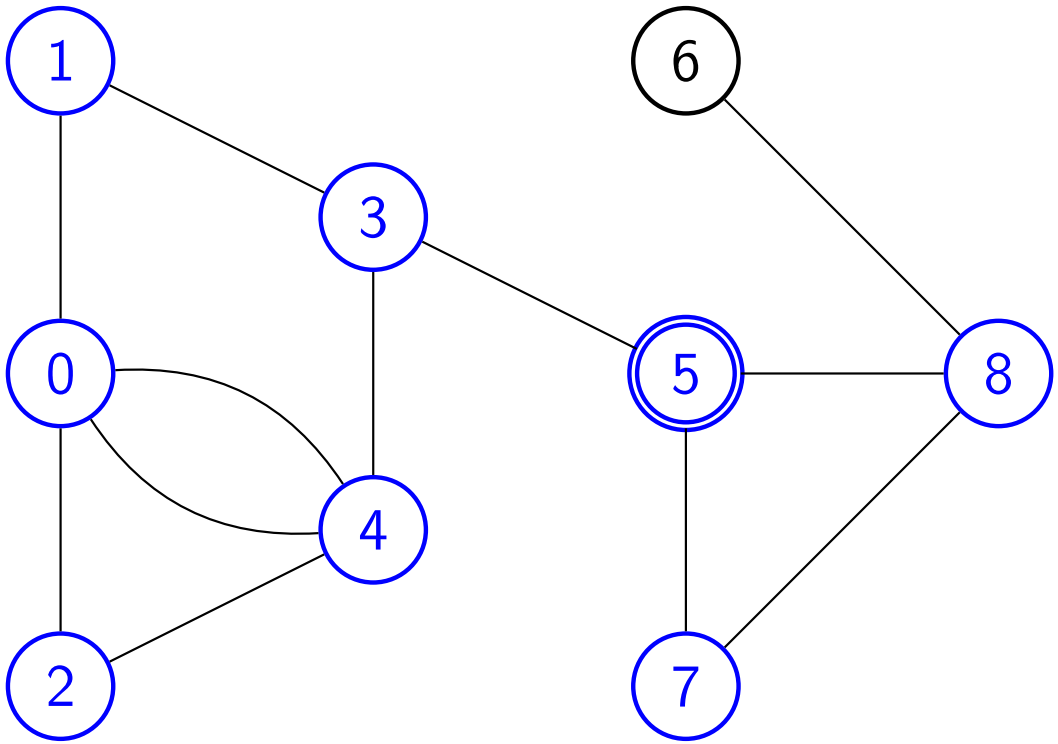

```
s = [
 5,
 2,
 4
]
```

DFS



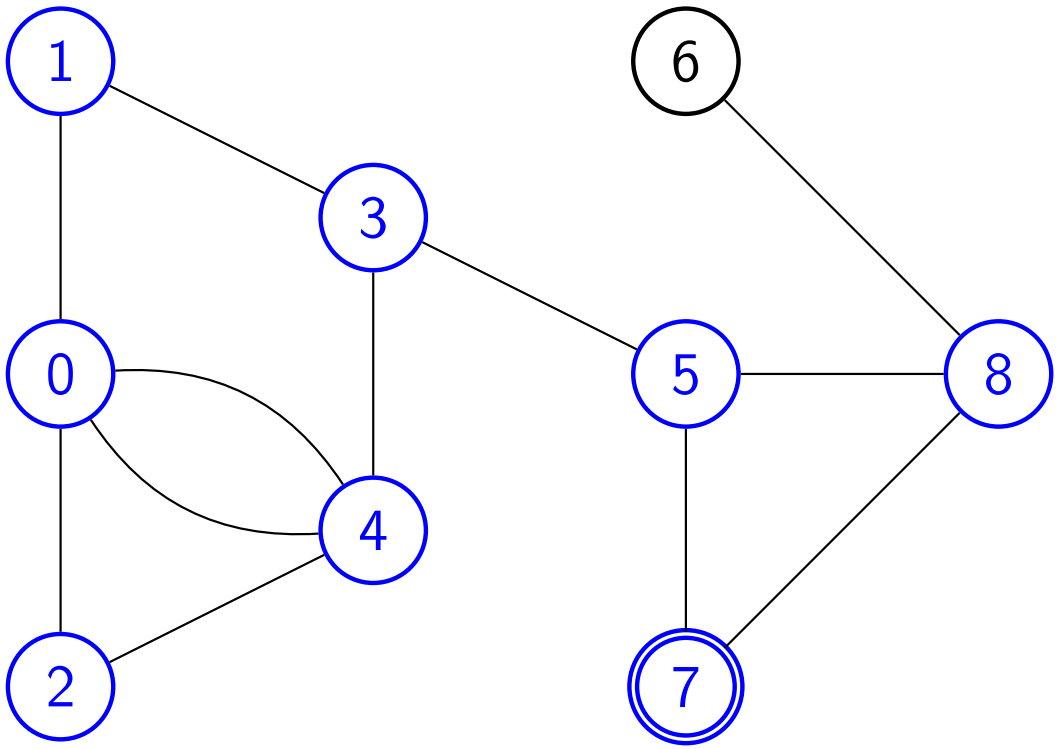

```
s = [
 2,
 4
]
```

DFS



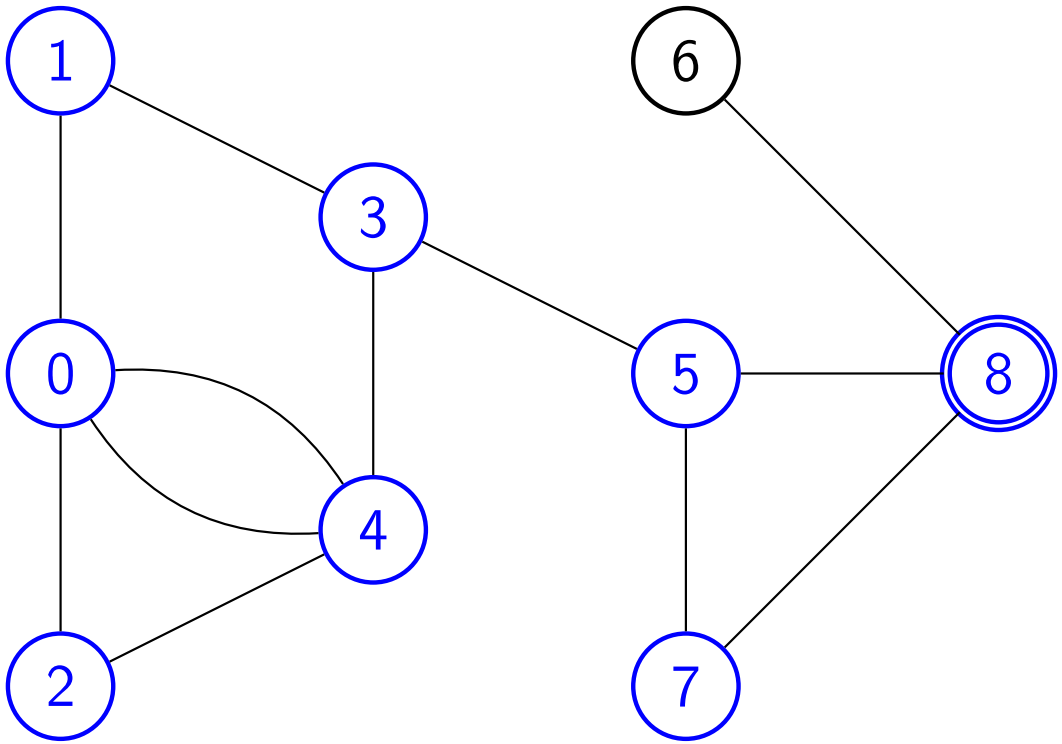

```
s = [
  7,
  8,
  2,
  4
]
```

DFS



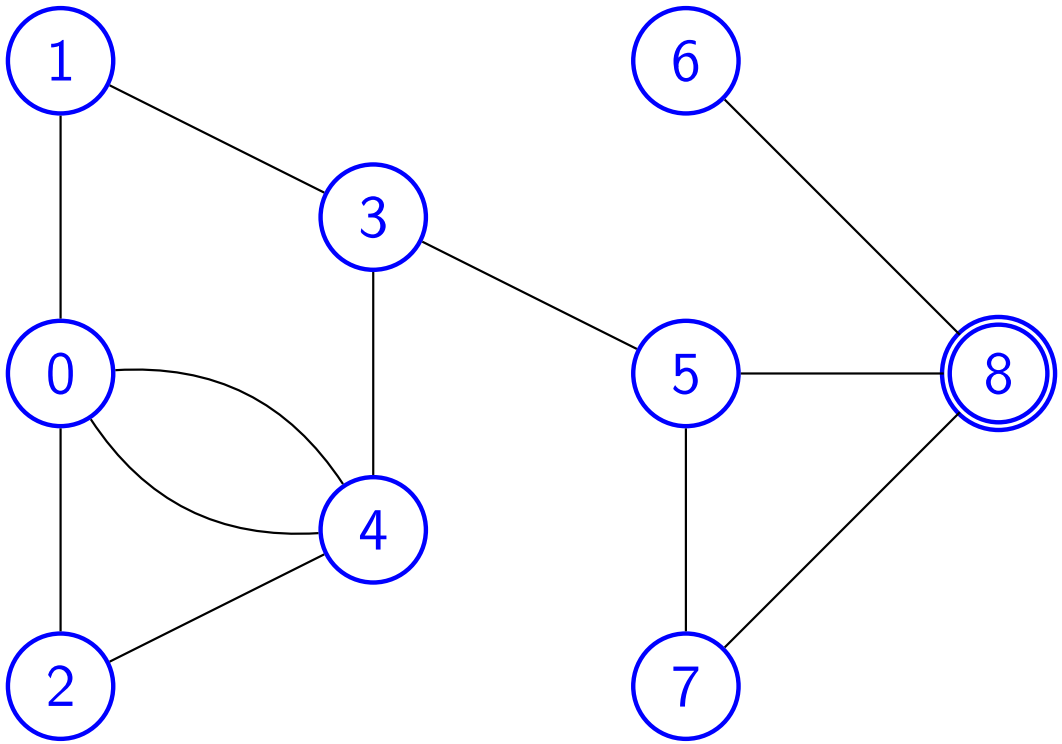

```
s = [
 8,
 2,
 4
]
```

DFS



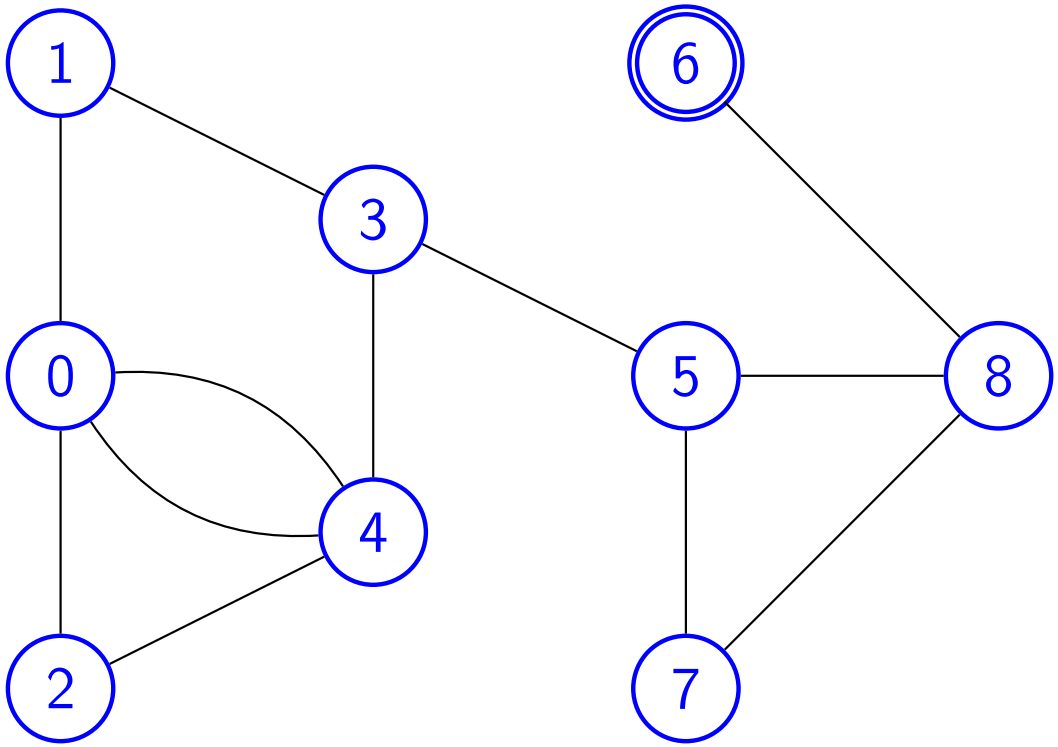

```
s = [
 2,
 4
]
```

DFS



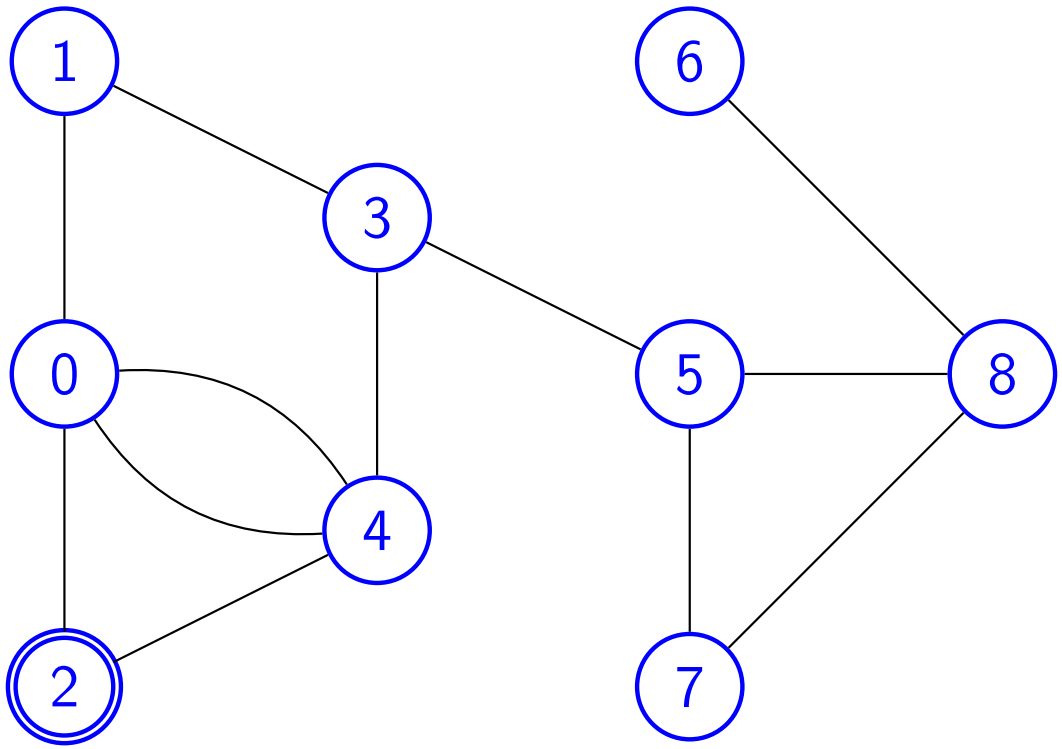

```
s = [
 6,
 2,
 4
]
```

DFS



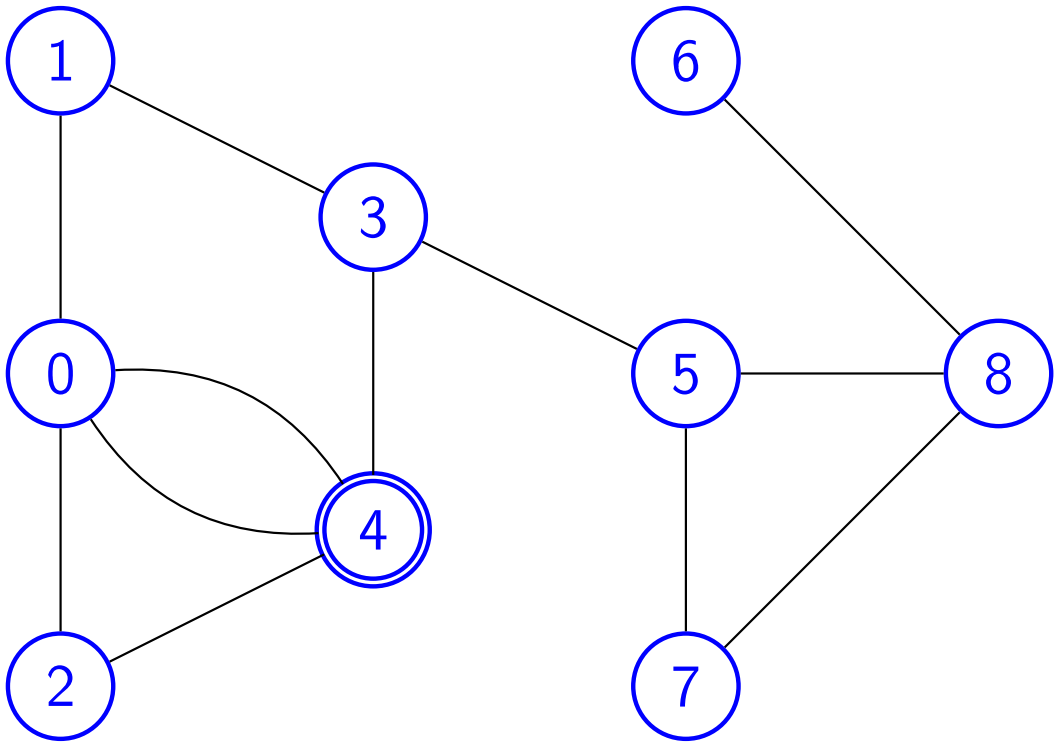

```
s = [
 2,
 4
]
```

DFS



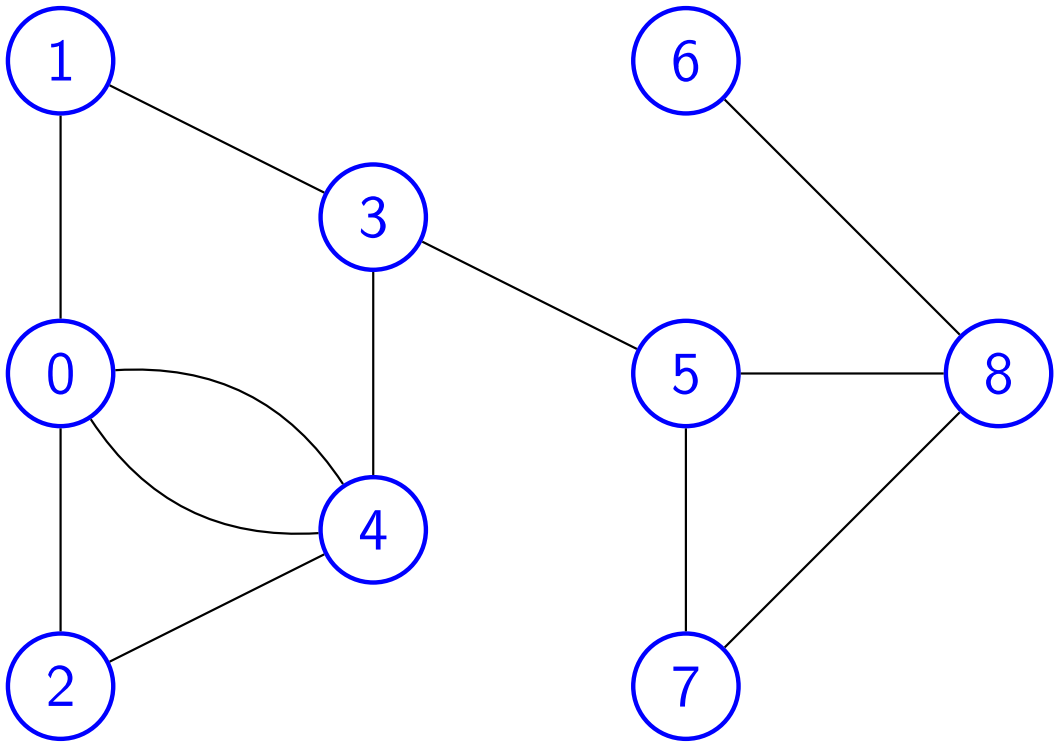

```
s = [
  4
]
```

DFS




```
s = [
]
```

DFS




```
s = [
]
```

DFS útfærsla með nágrannalista

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;

int main() {
    int n, m, a, b;
    cout << "Gefðu upp fjölda hnúta og leggja\n";
    cin >> n >> m;
    cout << "Gefðu upp m leggi (0-index)\n";
    vvi g(n, vi());
    for(int i = 0; i < m; ++i) {
        cin >> a >> b;
        g[a].push_back(b), g[b].push_back(a);
    }
    stack<int> q; vector<bool> d(n, false);
    q.push(0); d[0] = true;
    cout << "Í samhengisþætti 0 fundust:\n";
    while(!q.empty()) {
        int cur = q.top(); q.pop();
        cout << cur << '\n';
        for(int x : g[cur]) {
            if(d[x]) continue;
            d[x] = true; q.push(x); }}}
```

BFS vs. DFS

- Spurning er þá mögulega, hvenær á að nota hvert?
- Ef BFS er notað verða hnútar skoðaðir í fjarlægðarröð, þ.e. auðveldlega má bæta smá kóða í BFS til að fá hvað þarf að ferðast um marga leggi til að komast í aðra hnúta frá upphafspunkti.
- Kosturinn við DFS er ekki jafn augljós. En endurkvæmnu eiginleikar DFS munu nýtast mjög í reikniritum þegar áfram líður, aðstæður þar sem BFS dugar ekki.

BFS útfærsla (+ fjarlægð)

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;
vi distances(vvi &g, int s) {
    vector<int> d(n, -1), q;
    q.push(s); d[s] = 0;
    while(!q.empty()) {
        int cur = q.front(); q.pop();
        for(int x : g[cur]) {
            if(d[x] != -1) continue;
            d[x] = d[cur] + 1; q.push(x); }}}
int main() {
    int n, m, a, b;
    cout << "Gefðu upp fjölda hnúta og leggja\n";
    cin >> n >> m;
    cout << "Gefðu upp m leggi (0-index)\n";
    vvi g(n, vi());
    for(int i = 0; i < m; ++i) {
        cin >> a >> b;
        g[a].push_back(b); g[b].push_back(a);
    }
    for(int x : distances(g, 0)) cout << x << ' ';
    cout << endl; }
```

Samhengisþættir

- Við sjáum að þetta fer bara í gegnum einn samhengisþátt netsins.
- Ef netið er ekki samanhangandi (s.s. hefur fleiri en einn samhengisþátt) þá þarf að halda utan um samhengisþættina og leita í hverjum fyrir sig.
- Gera má það með union-find með því að join-a öllum aðlægum hnútum og gefur þá union-find-ið í hvaða samhengisþætti hver hnútur er.
- Sjáum meira um þetta þegar við tölum um tengihnúta og brýr, en fyrst stutt innskot um tvíhlutanet.

Litun á neti

- Þegar talað er um litun á neti er átt við að lita hnúta netsins þannig að engir nágrannar hafi sama lit.
- Að finna lágmarksfjölda lita í neti er NP-vandamál svo við munum ekki skoða það hér. Jafnvel bara að skoða hvort lita megi net með 3 litum er NP-vandamál.
- Hvort lita megi net með einum lit er ekki spennandi því það er hægt þ.þ.a.a. netið hafi enga leggi. Því er tilvikið fyrir tvo liti það sem við munum skoða. Hvenær má lita net með tveimur litum?
- Þegar þetta er hægt má skipta hnútunum í tvo hópa svo allir leggirnir liggji milli hópa en ekki milli tveggja hnúta í sama hópnum.
- Net þar sem þetta er hægt kallast tvíhlutanet (bipartite graph).

Tvíhlutanet

- Tvíhlutanet hafa marga þægilega eiginleika. Þið getið til dæmis reynt að sannfæra ykkur um að tvíhlutanet hafi engar rásir af odda lengd.
- En hvernig ákvörðum við hvort net sé tvíhlutanet?
- Við getum reynt að lita það í tveimur litum gráðugt og séð hvort við litum okkur út í horn!
- Litum upphafspunkt bláan, nágranna hans rauða, nágranna þeirra bláa o.s.frv. og sjáum hvort það gangi upp. Göngum s.s. í gegnum netið með BFS og litum jafnóðum. Gera þarf þetta fyrir hvern samhengisþátt og er netið tvíhlutanet ef sérhver samhengisþáttur þess er það.
- Sjáum hér útfærslu fyrir einn samhengisþátt (eða samanhangandi net). Hvernig ætti að breyta þessu svo þetta virki fyrir ósamanhangandi net?

Tvíhlutatékk (fyrir einn samhengisþátt)

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;

bool bipartite(vvi& g) {
    queue<int> q;
    q.push(0);
    vi color(g.size(), -1);
    color[0] = 0;
    while(!q.empty()) {
        int cur = q.front();
        q.pop();
        for(int x : g[cur]) {
            if(color[x] == -1) {
                color[x] = 1 - color[cur];
                q.push(x);
            } else if(color[x] == color[cur]) {
                return false;
            }
        }
    }
    return true;
}
```

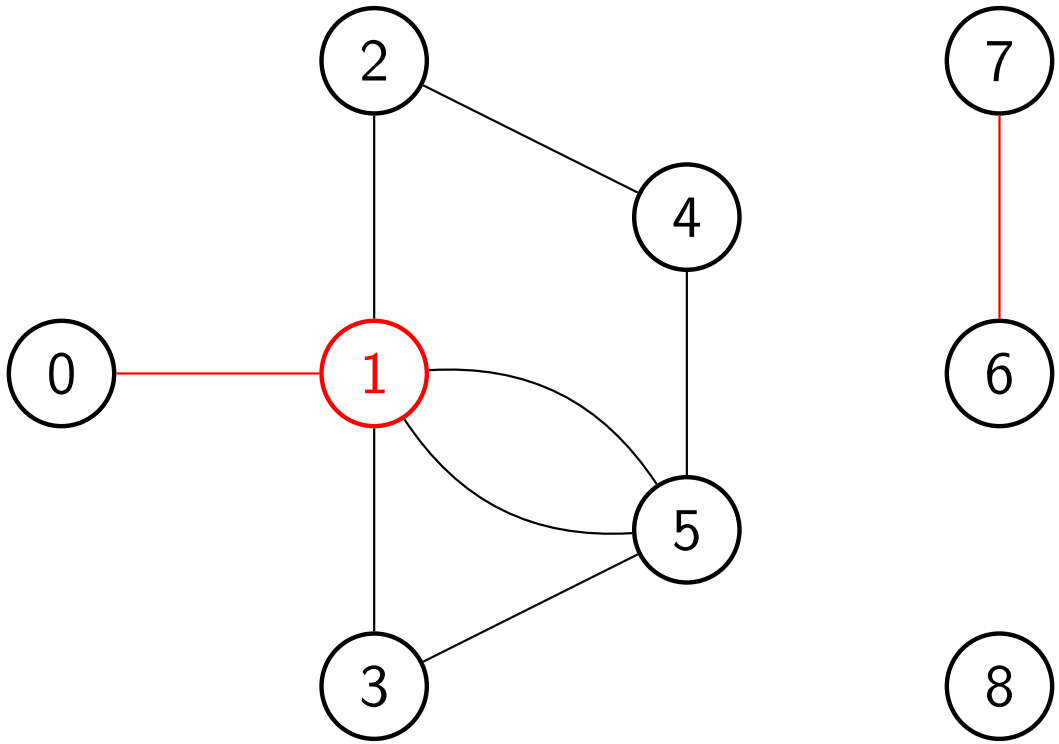
Tvíhlutatékk

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;

bool bipartite(vvi& g) {
    queue<int> q;
    vi color(g.size(), -1);
    for(int i = 0; i < g.size(); ++i) {
        if(color[i] != -1) continue;
        q.push(i);
        color[i] = 0;
        while(!q.empty()) {
            int cur = q.front();
            q.pop();
            for(int x : g[cur]) {
                if(color[x] == -1) {
                    color[x] = 1 - color[cur];
                    q.push(x);
                } else if(color[x] == color[cur]) {
                    return false;
                } } } }
    return true;
}
```

Tengihnútar og brýr

- Tengihnútar (cut points) eru hnútar sem hafa þann eiginleika að ef þeir eru fjarlægðir fjölgar samhengisþáttum netsins. Brýr (bridges) eru leggir með sama eiginleika. Þeir eru því hnútar/leggir sem eru nauðsynlegir til að halda netinu samanhangandi.
- Ef maður horfir á mynd af neti er yfirleitt frekar auðvelt að 'sjá út' hvaða hnútar eru tengihnútar og hvaða leggir eru brýr. Skoðum dæmi um þetta.

Dæmi um tengihnúta og brýr



Að finna tengihnúta og brýr

- En hvernig má finna þetta með tölvu?

Að finna tengihnúta og brýr

- En hvernig má finna þetta með tölvu?
- Kemur í ljós að finna má þetta allt með því að fara í gegnum netið með einu DFS!
- Þegar við löbbum í gegn með DFS geymum við fyrir hvern hnút tvær tölur, oftast kallaðar `low` og `num`.
- `num` geymir hvað það tók okkur mörg skref að komast í þann hnút með DFS leitinni og `low` segir hvert er lægsta `num` gildi sem er hægt að komast í úr þessum hnút án þess að endurtaka leggi sem við notuðum í DFS leitinni sjálfri.
- Ef við förum í hnút v á eftir hnút u og við endum með að `low` fyrir v sé stærra en `num` fyrir u þá er leggurinn frá u til v brú því við komumst bara úr v í u með þeim eina legg.
- Ef við förum í hnút v á eftir hnút u og við endum með að `low` fyrir v sé stærra en eða jafnt `num` fyrir u þá er u tengihnútur því þá er enginn önnur leið til baka úr v sem fer ekki um u.

Að reikna `num` og `low`

- Í byrjun förum við bara í gegnum netið með DFS og setjum `num` útfrá því. Við látum í byrjun bara `low` vera jafnt `num` í hverjum hnút.
- Svo ef við rekumst á legg sem við notuðum ekki í DFS-leitinni til að komast í hnútinn sem við erum stödd í uppfærum við `low` fyrir alla hnútana sem komast í þann legg.
- Þetta virkar fyrir alla hnútana nema fyrir þann sem við byrjum í, en við tékkum bara á honum sérstaklega.
- Skoðum nú útfærslu á þessu.

Tengihnúta- og brúarleit útfærð

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int,int> ii;
typedef vector<ii> vii;
vi low, num;
int curnum;

void dfs(vvi &g, vi &cp, vii &bri, int u, int p) {
    low[u] = num[u] = curnum++;
    int cnt = 0; bool found = false;
    for(int v : g[u]) {
        if(num[v] == -1) {
            dfs(g, cp, bri, v, u);
            low[u] = min(low[u], low[v]);
            cnt++; found |= low[v] >= num[u];
            if(low[v] > num[u]) bri.push_back(ii(u, v));
        } else if(p != v) low[u] = min(low[u], num[v]); }
  if(found && (p != -1 || cnt > 1)) cp.push_back(u); }

pair<vi,vii> cutpointsbridges(vvi &g) {
    vi cp; vii bri;
    num = vi(g.size(), -1);
    low.resize(g.size());
    curnum = 0;
    for(int i = 0; i < g.size(); ++i) {
        if(num[i] == -1) {
            dfs(g, cp, bri, i, -1);
        }
    }
    return pair<vi,vii>(cp, bri); }
```

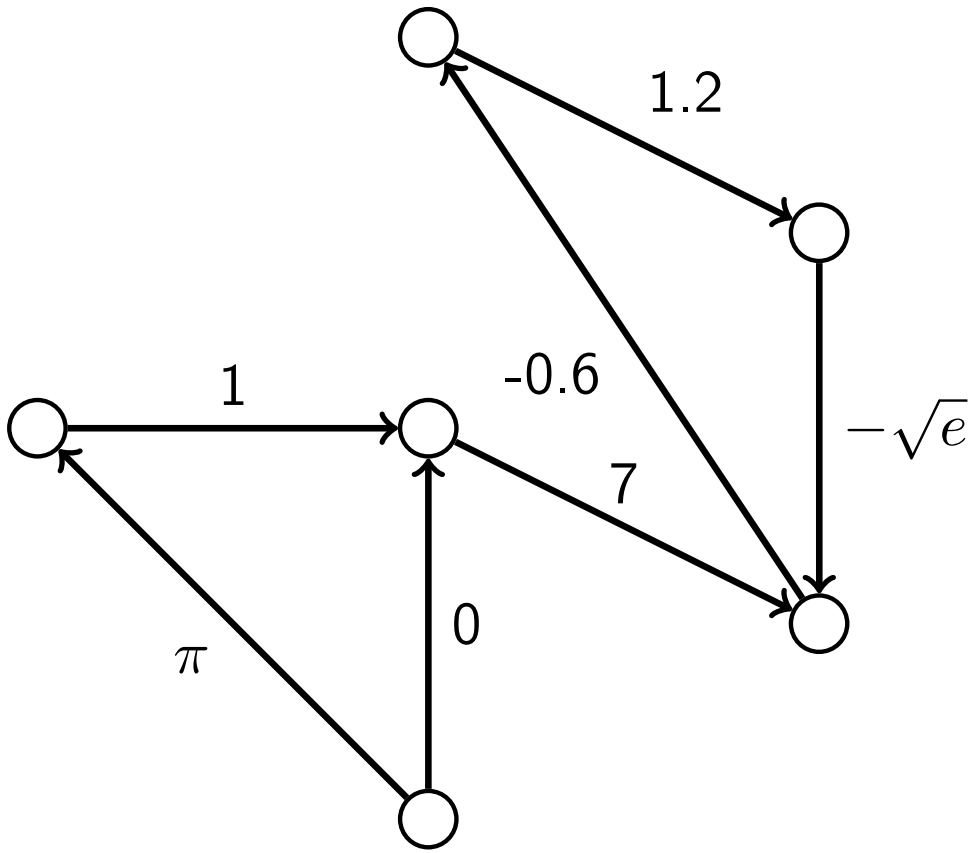
DFS fyrir- og eftirskilyrði

- Aðferðin sem við beittum hér til að finna tengihnúta og brýr er nýting á fyrirbæri sem kallast DFS fyrir- og eftirskilyðri.
- Þar sem DFS notar hlaða og endurkvæmni með forritunarmálum er útfærð með hlaða má framkvæma DFS með því að kalla á leit endurkvæmt í öllum nágrönnum í staðinn fyrir að nota sinn eiginn hlaða.
- Þá er hægt að framkvæma reikninga þegar komið er inn í hnútinn (fyrirskilyrði) og eftir að búið er að afgreiða alla nágranna hans og verið er að bakka til baka (eftirskilyrði).
- Þetta er mjög öflug aðferð og er vert að velta þessu aðeins fyrir sér.
- ATH: BFS virkar ekki til þess að gera svona útreikninga.

Hvað er stefnt eða vigtað net?

- Stefnt net er net þar sem leggirnir hafa stefnu, þ.e.a.s. þeir liggja frá einum hnút til annars en ekki til baka. Þetta getur táknað einstefnur eða aðra ferðamáta sem virka bara í eina átt, hlutir að flæða í eina átt, hver sigrar hvern og margt fleira.
- Vigtað net er net þar sem leggirnir hafa þyngd (eða lengd) sem er einhver rauntala. Þetta getur táknað lengd vegarins sem svarar til leggjarins, kostnað við að smíða legginn, um hversu mikið annar leikmaður vann hinn og margt fleira.
- Net geta verið óstefnd og óvigtuð, óstefnd og vigtuð, stefnd og óvigtuð eða stefnd og vigtuð. Í öllum þessum tilvikum geta þau verið einföld eða ekki.

Dæmi um stefnt vigtað net



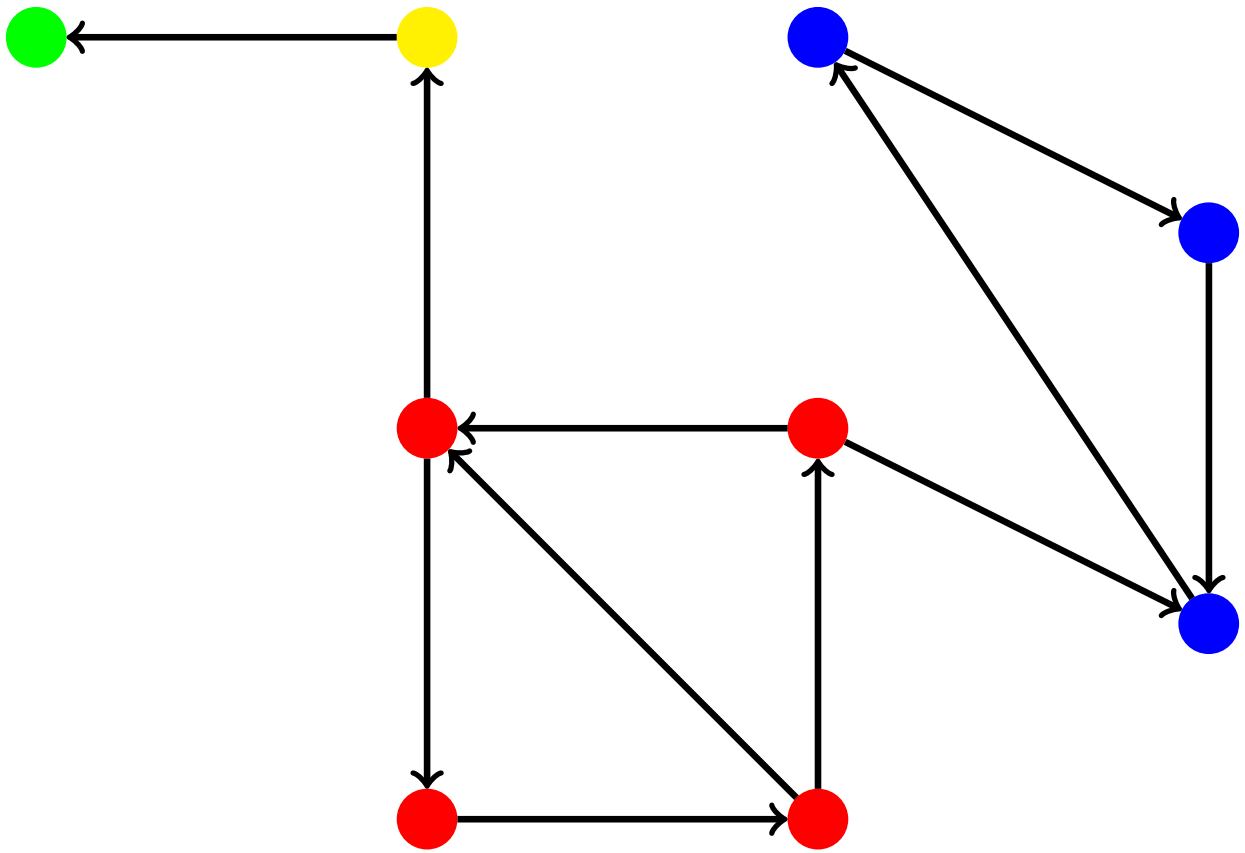
Framsetning

- Nágrannaframseting er geymd með sama hætti í stefndu neti nema við geymum bara að a sé nágranni b ef örin liggur í þá átt en ekki öfugt. Ef netið er vigtað er hægt að geyma tvenndir (n, w) í listanum þar sem n er nágrannin og w er vigt örvarinnar þangað. Notum þetta áfram mest.
- Í fylkjaframsetningu getum við bara geymt einföld vigtuð net eða óvigtuð net. Ef þau eru vigtuð látum við töluna í $A[i][j]$ vera vigt leggjarins í stað fjölda leggja. Við getum þá ekki heldur haft vigtir sem eru 0 nema við látum ∞ tákna enga leggi. Ef netið er stefnt setjum við bara $A[i][j]$ sem legginn frá i til j en $A[j][i]$ sem legginn frá j til i.
- Í listaframsetningu breytist lítið. Ef netið er stefnt táknar þá (a, b) legg frá a til b en ekki öfugt. Ef netið er vigtað notum við þrenndir (a, b, w) í stað tvennda og geymum þá vigtina w aftast.

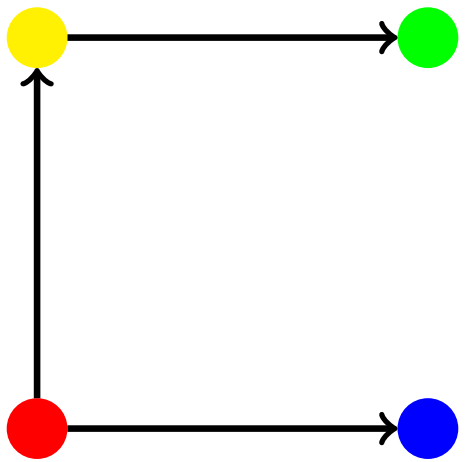
Hvað getum við gert nýtt á stefndu neti?

- Við getum skoðað samhengisþætti í nýju ljósi. Við getum talað um stranglega samanhangandi þætti (SCC) sem eru hlutar stefnds nets þar sem við komumst milli sérhverra tveggja hnúta þess.
- Með þessu getum við líka látið alla slíka samhengisþætti skreppa saman í einn hnút og köllum við það herpingu nets. Við það að gera þetta fæst svokallað órásað stefnt net (DAG), sem þýðir einfaldlega að ekki sé hægt að fara í hring í netinu ef ferðast er meðfram örvunum.
- Skulum sjá dæmi um stranglega samanhangandi þætti, herpingu og órásað stefnt net.

Stranglega samanhangandi þættir



Herping



Ákvörðun stranglegra samanhangandi þátta

- Oft er nytsamlegt að ákvarða stranglega samanhangandi þætti nets. En hvernig má gera það?
- Við getum breytt reikniritinu úr síðasta fyrirlestri sem finnur brýr og tengihnúta til þess að gera þetta.
- Þetta vinnur útfrá því að ef leggur er brú er engin leið að komast til baka skv. skilgr. brúar, svo endahnútar brúarinnar verða að vera í ólíkum stranglega samanhangandi þáttum.
- Þetta til viðbótar við það að skoða hvenær við komumst til baka í fyrri hnút gefur okkur reiknirit Tarjans til að finna stranglega samanhangandi þætti.
- Útfærslan í bókinni skilar lista af listum þar sem hver listi er einn stranglega samanhangandi þáttur. Í minni útfærslu skila ég union-find tilviki í staðinn. Útfærslan dregur einkenni sín af útfærslu Bjarka Ágústs Guðmundssonar.

Reiknirit Tarjans

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;

struct union_find {
  vi p; union_find(int n) : p(n, -1) { }
  int find(int x) {
      return p[x] < 0 ? x : p[x] = find(p[x]); }
  bool united(int x, int y) {
      return find(x) == find(y); }
  void unite(int x, int y) {
    int xp = find(x), yp = find(y);
    if (xp == yp) return;
    if (p[xp] > p[yp]) swap(xp,yp);
    p[xp] += p[yp], p[yp] = xp;
    return; }
int size(int x) { return -p[find(x)]; } };

vi ord;
vector<bool> done;

void dfs(vvi& g, int v) {
    done[v] = true;
    for(int x : g[v]) if(!done[x]) dfs(g, x);
    ord.push_back(v);
}
```

```
pair<union_find, vi> tarjan(vvi& g) {
    ord.clear();
    union_find uf(g.size());
    vi dag; vvi gr(g.size());
    for(int i = 0; i < g.size(); ++i) {
        for(int x : g[i]) {
            gr[x].push_back(i);
        }
    }
    done.resize(g.size());
    for(int i = 0; i < g.size(); ++i)
        done[i] = false;
    for(int i = 0; i < g.size(); ++i)
        if(!done[i]) dfs(gr, i);
    for(int i = 0; i < g.size(); ++i)
        done[i] = false;
    stack<int> s;
    for(int i = g.size() - 1; i >= 0; --i) {
        if(done[ord[i]]) continue;
        s.push(ord[i]); dag.push_back(ord[i]);
        while(!s.empty()) {
            int t = s.top();
            done[t] = true;
            s.pop();
            uf.unite(t, ord[i]);
            for(int x : g[t]) {
                if(!done[x]) s.push(x);
            }
        }
    }
    reverse(dag.begin(), dag.end());
    return make_pair(uf, dag);
}
```

Prófun á Tarjan

```
int main() {
    int n, m, a, b;
    cin >> n >> m;
    vvi g(n, vi());
    for(int i = 0; i < m; ++i) {
        cin >> a >> b;
        g[a].push_back(b);
    }
    auto t = tarjan(g);
    for(int x : t.first.p) {
        cout << x << ' ';
    }
    cout << endl;
    for(int x : t.second) {
        cout << x << ' ';
    }
    cout << endl;
}
```

```
// Inntak
9 11
1 0
3 1
2 3
2 1
0 2
1 7
7 8
3 4
4 5
5 6
6 4
// Úttak
2 2 -4 2 5 -3 5 -1 -1
0 4 7 8
```

DAG?

- Tarjan keyrir almennt í $\mathcal{O}(E+V)$, í minni útfærslu er það í raun $\mathcal{O}(E+V\alpha(V))$ út af union-find en það er í öllum raunhæfum tilfellum varla neinn munur þar á.
- En hvað er þetta dag sem ég skila líka? Eins og kom fram áðan er DAG órásað stefnt net. Það merkilega við slík net er að raða má hnútunum í röð (ekki endilega ótvírætt ákvarðaða) þ.a. ef til er leggur frá a til b verður b að vera á eftir a í röðinni.
- Þetta kallast grannröðun (topological sort). Reikniritið hér á undan gefur okkur sem sagt grannröðun á herpingu netsins frítt með!
- En hvað ef við viljum grannröðun á neti sem þegar er órásað? Það myndi virka að keyra reikniritið að ofan því þá yrðu allir hnútar sinn eiginn stranglega samanhangandi þáttur. Það má gera það með smærra forriti sem fjarlægir endurtekið hnúta með innstig 0 og lætur þá vera stærsta. Ef þetta fjarlægir ekki alla hnúta er netið ekki DAG. Sjáum útfærslu á þessu.

Reiknirit Kahns

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vvi;

// amb er hvort það sé ótvírætt röð
// skilar lengd minna en g.size() ef ekki dag
vi topsort(vvi &g, bool &amb) {
    vi deg(g.size()), res;
    for(vi v : g) for(int x : v) deg[x]++;
    queue<int> q;
    for(int i = 0; i < g.size(); ++i) if(deg[i] == 0) q.push(i);
    amb = false;
    while(q.size() > 0) {
        if(q.size() > 1) amb = true;
        int cur = q.front(); q.pop();
        res.push_back(cur);
        for(int x : g[cur]) if(--deg[x] == 0) q.push(x);
    }
    return res;
}
```

Spannandi tré

- Það sem við getum gert núna er að skoða spurninguna 'Hver er ódýrasta leiðin til að tengja alla hnúta netsins saman?'
- Þá veljum við eitthvert hlutmengi leggja sem mynda tré. Af hverju tré? Nú, ef við værum með rás mætti sleppa einhverjum legg í rásinni, sem væri ódýrara.
- Svona fyrirbæri köllum við spannandi tré í netinu, og er þá spurningin hvernig við getum ákvarðað lágmarksþyngd á spannandi tré nets.

Reiknirit Kruskals

- Þó svo það sé kannski ekki augljóst í fyrstu kemur í ljós að til sé mjög einfalt gráðugt reiknirit til að gera þetta.
- Við bætum alltaf við ódýrasta leggnum sem myndar ekki rás!
- Æfing: sanna að þetta virki.
- Við getum þá haldið utan um hvaða leggir hafa veg á milli sín með því að nota union-find. Þá röðum við bara öllum leggjunum í vaxandi þyngdarröð og bætum við þar til allt er orðið tengt og skilum þyngdinni (eða trénu eftir tilvikum). Athugum að net þarf að vera tengt til að slíkt tré sé til. Annars þarf að gera þetta í hverjum samhengisþætti fyrir sig.
- Þetta keyrir í $\mathcal{O}(E \log(E))$ (α hverfur því það er minna en $\log$).

Kruskal útfærsla sem skilar þyngd

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef pair<double,ii> dii;
typedef pair<double,int> di;
typedef vector<di> vdi;
typedef vector<vdi> vvdi;
// sama unionfind og síðast

double kruskal(vvdi& g) {
    priority_queue<dii> s; union_find uf(g.size());
    double res = 0;
    for(int i = 0; i < g.size(); ++i)
        for(auto x : g[i]) s.push(dii(-x.first, ii(x.second, i)));
    while(uf.size(0) < g.size()) {
        auto x = s.top(); s.pop();
        if(uf.united(x.second.first, x.second.second)) continue;
        res -= x.first; uf.unite(x.second.first, x.second.second);
    }
    return res; }

int main() {
    int n, m, a, b;
    double w;
    cin >> n >> m;
    vvdi g(n, vdi());
    for(int i = 0; i < m; ++i) {
        cin >> a >> b >> w;
        g[a].push_back(di(w, b));
        g[b].push_back(di(w, a));
    } cout << kruskal(g) << endl; }
```

Hríslur

- Svipað má gera yfir stefnd net.
- Þá skoðar maður svokallaðar spannandi hríslur.
- Það að skoða spannandi hríslur er töluvert flóknara og kemur mun sjaldnar fyrir í keppnisforritun, svo við sleppum því.

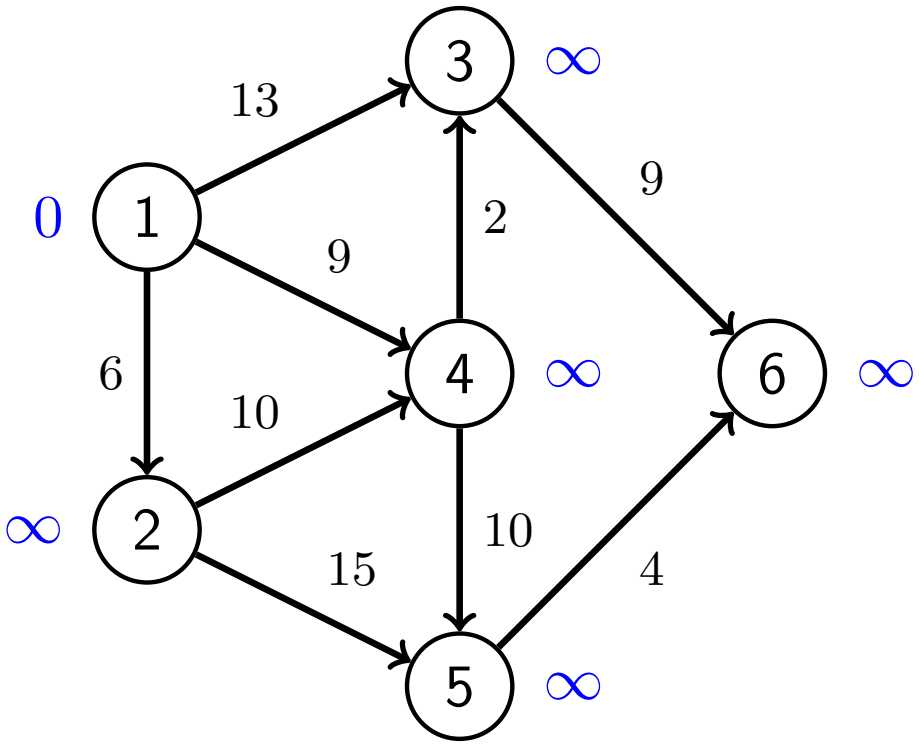
Leitarreiknirit

- Það sem við munum skoða í restinni á þessum fyrirlestri eru ýmsar leiðir til að finna bestu leið frá a til b í stefndu og vigtuðu neti.
- Bendum fyrst á eitt stórt vandamál. Ef til er rás með neikvæða þyngd í netinu er hægt að fara eftir henni aftur og aftur til að fá eins lítinn kostnað og maður vill. Því munum við nú fyrst gera ráð fyrir að við höfum net með ekki neikvæðum vigtum.
- Hvernig getum við þá fundið stystu leið frá a til b?
- Með einu frægasta reikniriti tölvunarfræðinnar, reikniriti Dijkstra!
- Einnig þekkt sem óvinur allra Vim-notenda eða reikniritið sem enginn er sammála um hvernig á að bera fram, ekki einu sinni Hollendingar sjálfir.

Dijkstra

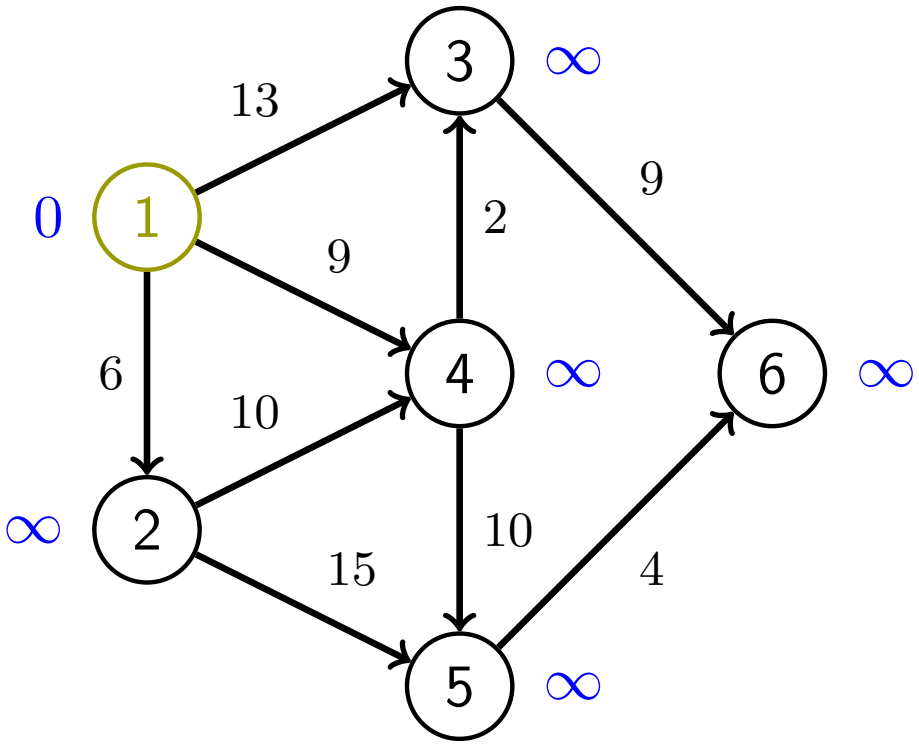
- Dijkstra finnur stystu leið í alla hnúta frá einum hnút v. Það byrjar á því að segja að þessi fjarlægð sé ∞ fyrir alla hnúta og að hnúturinn sem við komum úr sé enginn.
- Við geymum svo hnútana sem við eigum eftir að skoða í forgangsbiðröð þar sem forgangurinn er stutt í þá.
- Svo meðan biðröðin er ekki tóm tökum við efsta og bætum við nágrönnum þess sem við erum ekki búin að skoða en uppfærum líka fjarlægðina á nágrönnunum ef við erum búin að finna styttri leið. Við uppfærum þá um leið úr hvaða hnút við vorum að koma.

Dijkstra in action



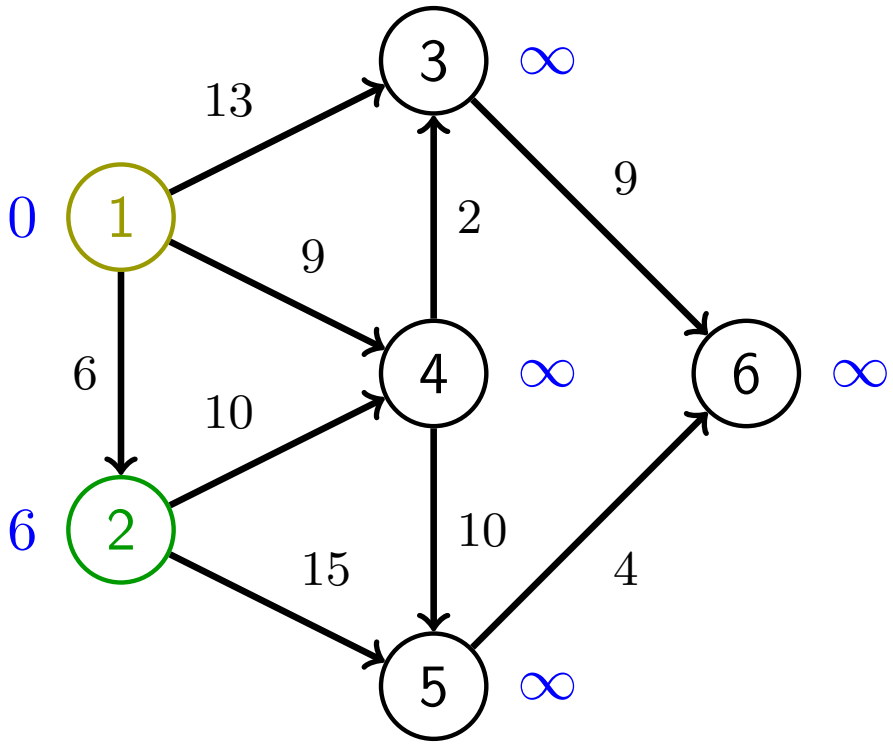

```
pq = [
  (0, 0)
]
```

Dijkstra in action



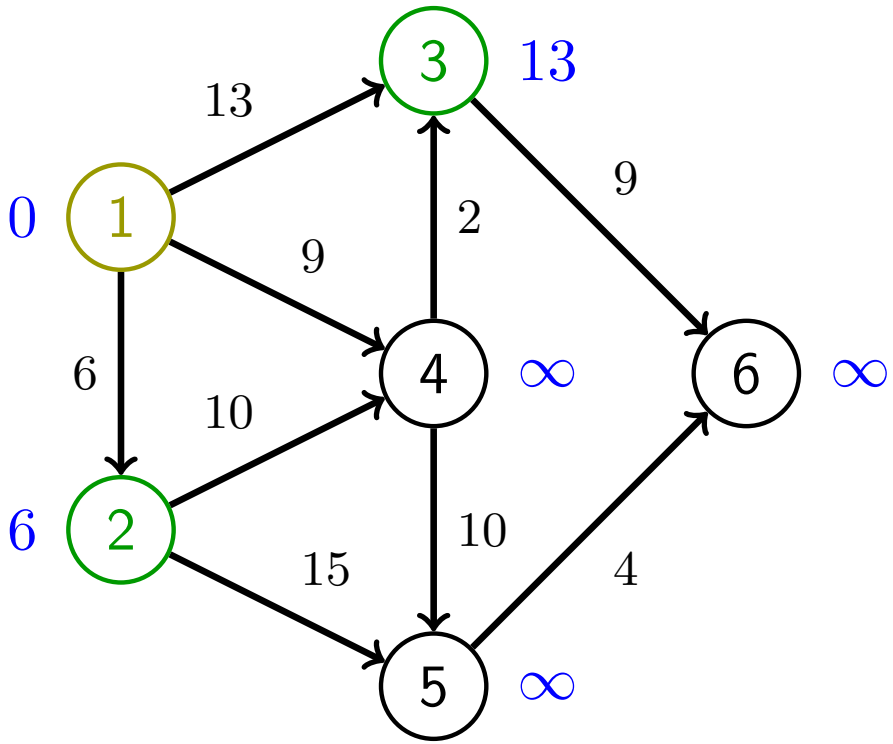

```
pq = [
]
```

Dijkstra in action



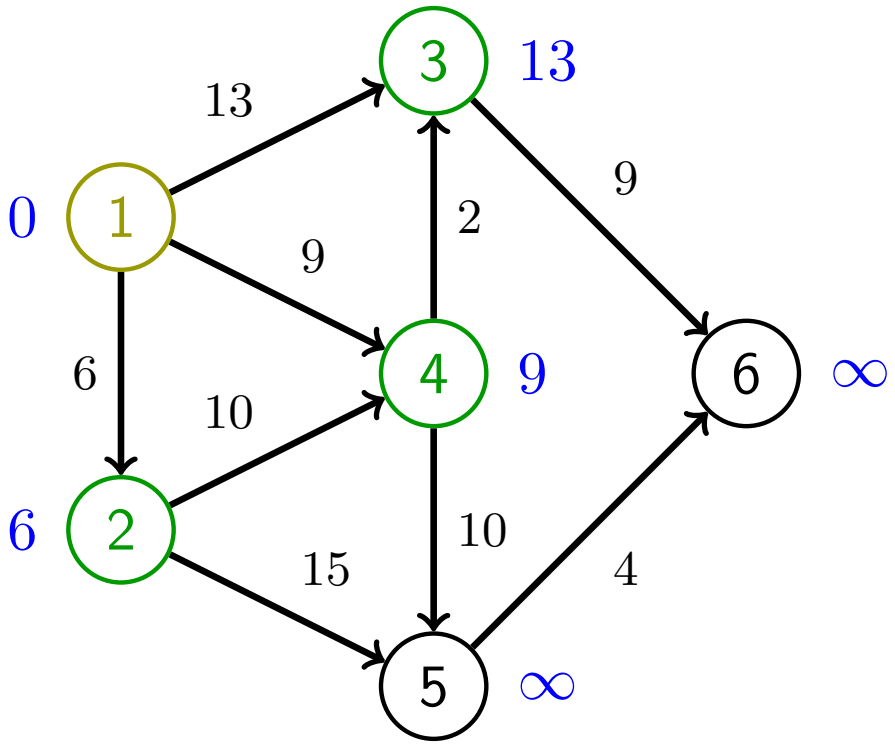

```
pq = [
 (6, 2)
]
```

Dijkstra in action



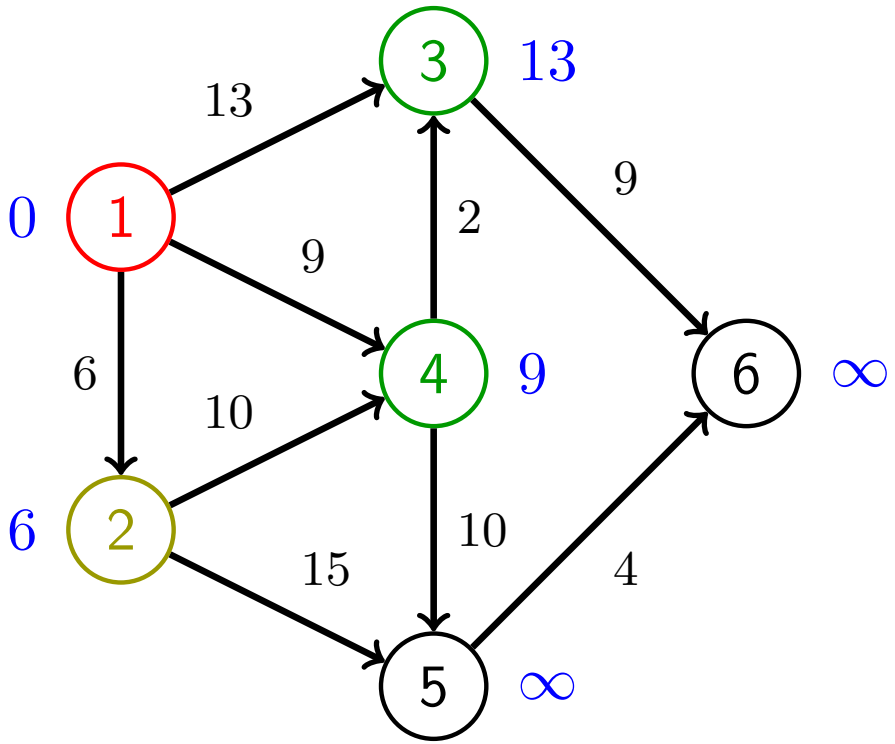

```
pq = [
  (6, 2),
  (13, 3)
]
```

Dijkstra in action



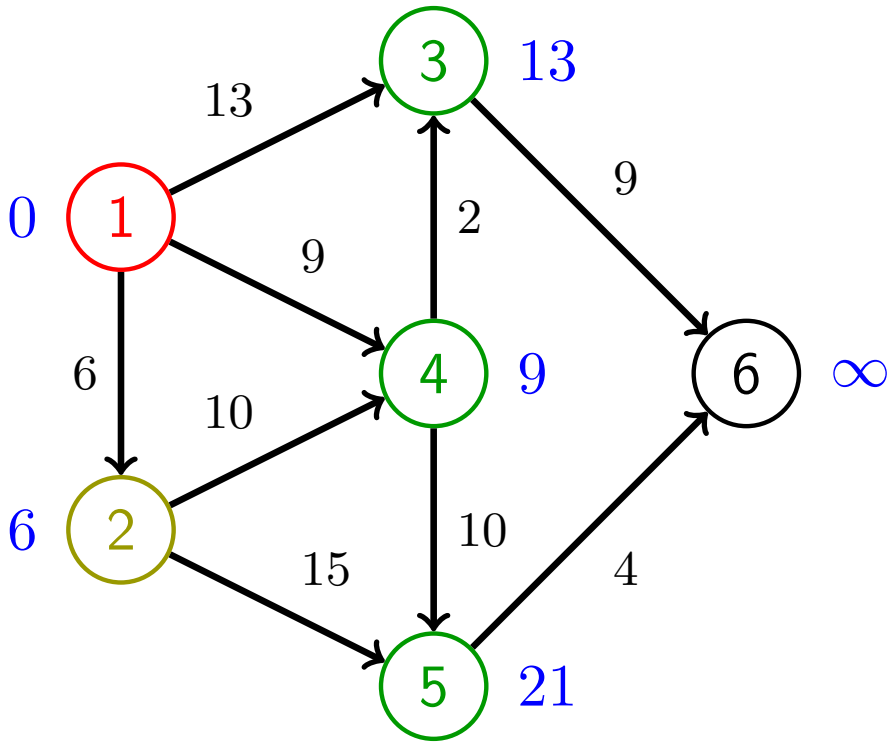

```
pq = [
  (6, 2),
  (9, 4),
  (13, 3)
]
```

Dijkstra in action



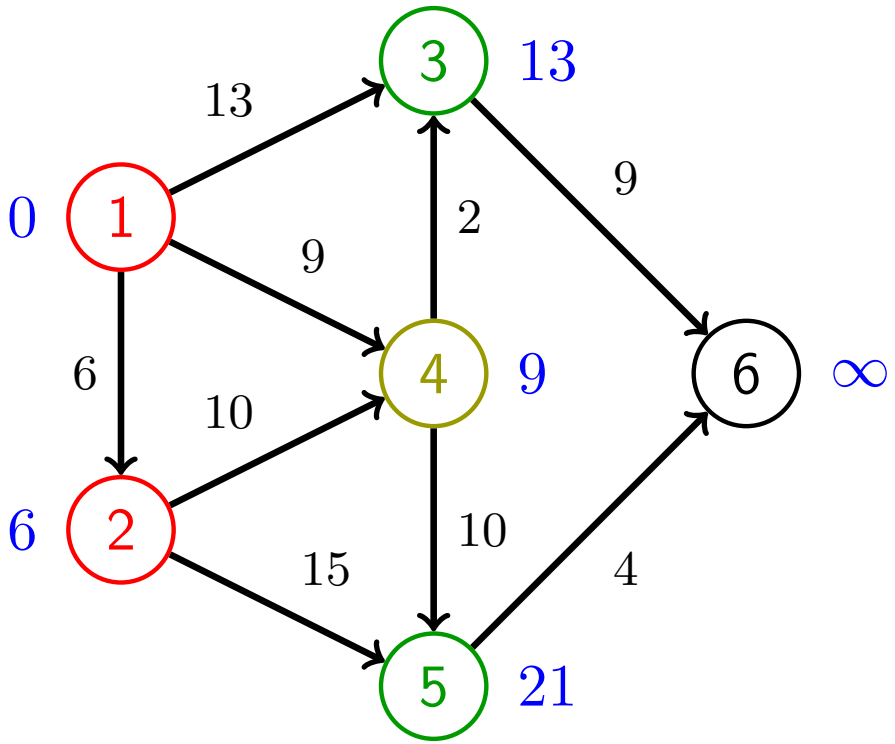

```
pq = [
  (9, 4),
  (13, 3)
]
```

Dijkstra in action



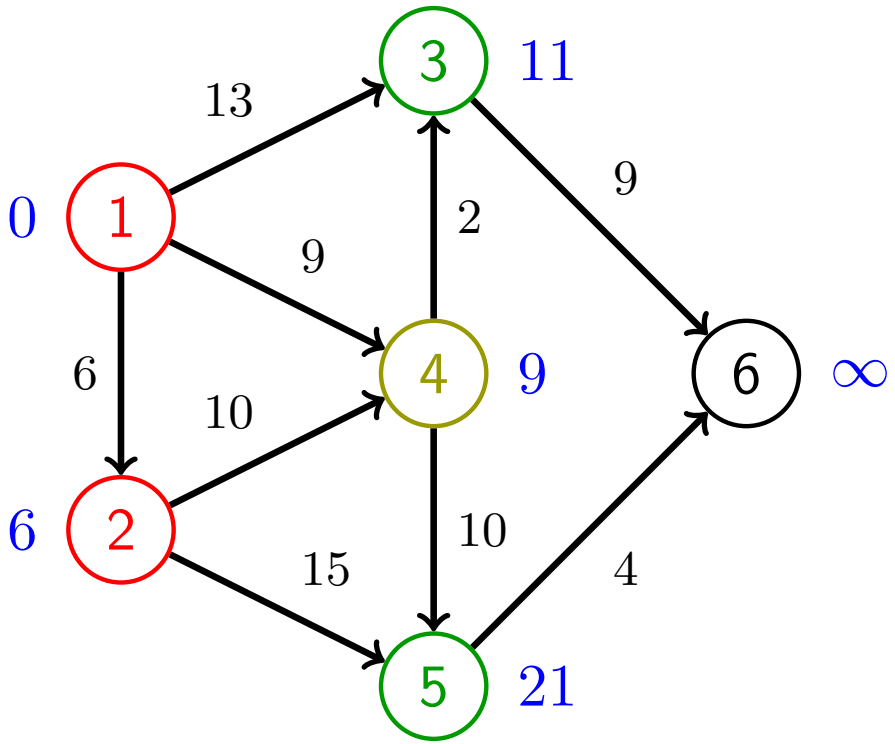

```
pq = [
  (9, 4),
  (13, 3),
  (21, 5)
]
```

Dijkstra in action



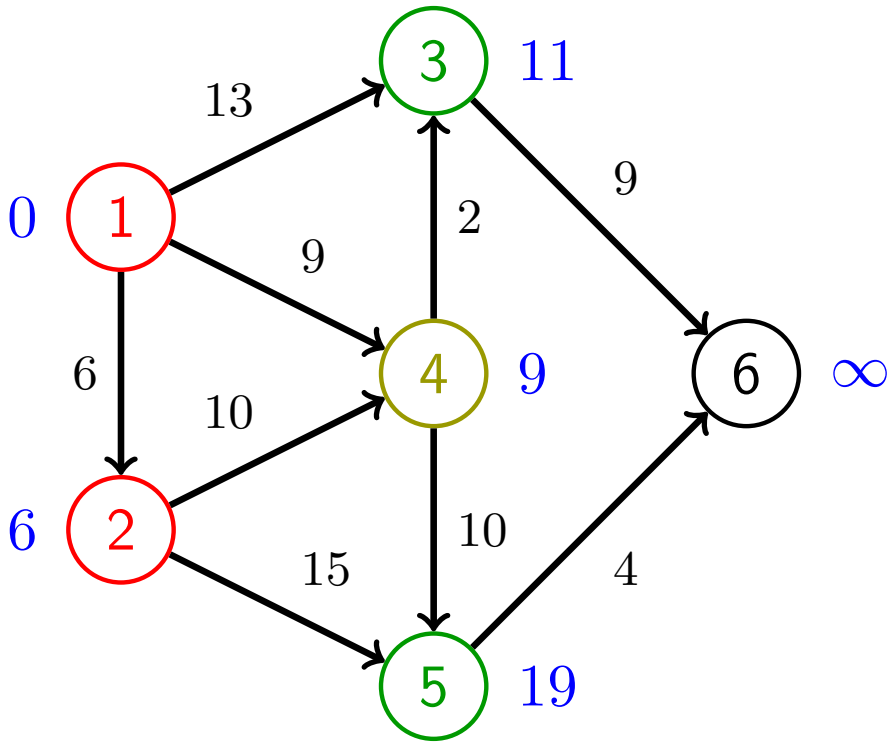

```
pq = [
  (13, 3),
  (21, 5)
]
```

Dijkstra in action



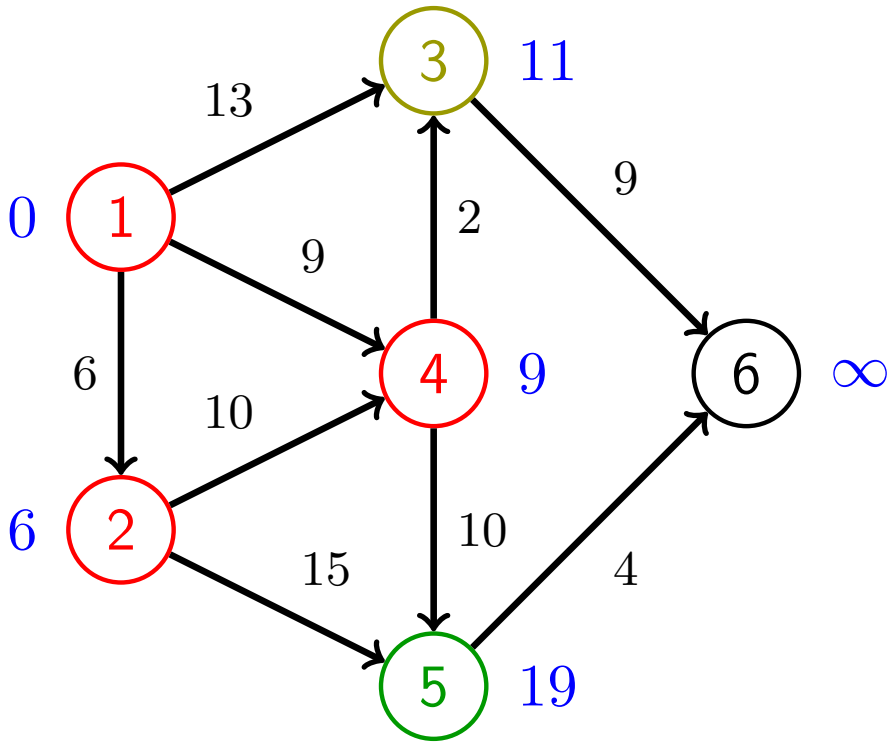

```
pq = [
  (11, 3),
  (21, 5)
]
```

Dijkstra in action



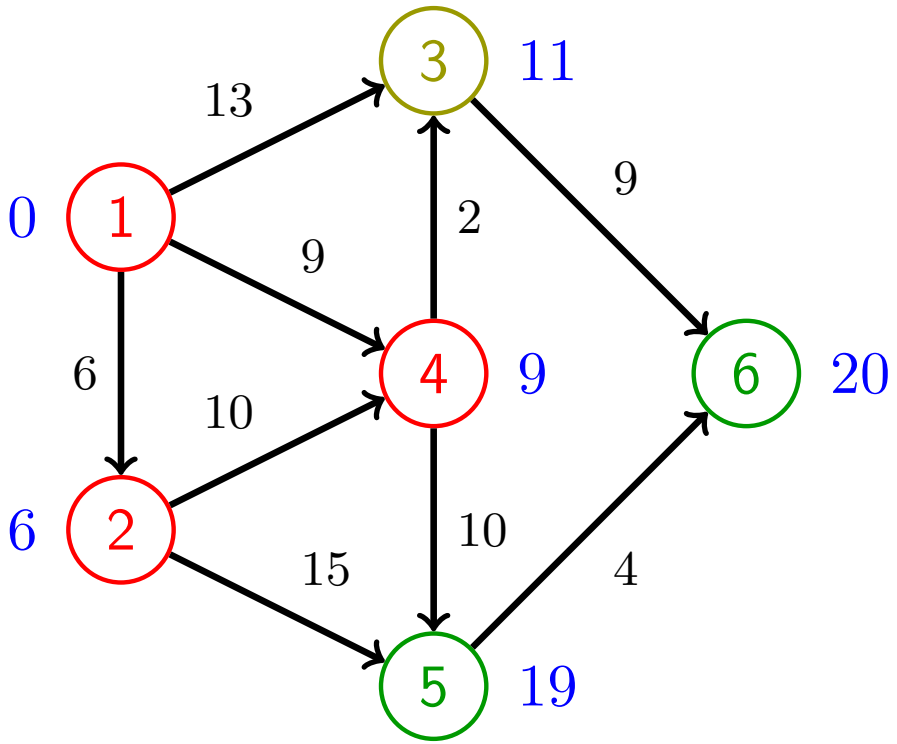

```
pq = [
  (11, 3),
  (19, 5)
]
```

Dijkstra in action



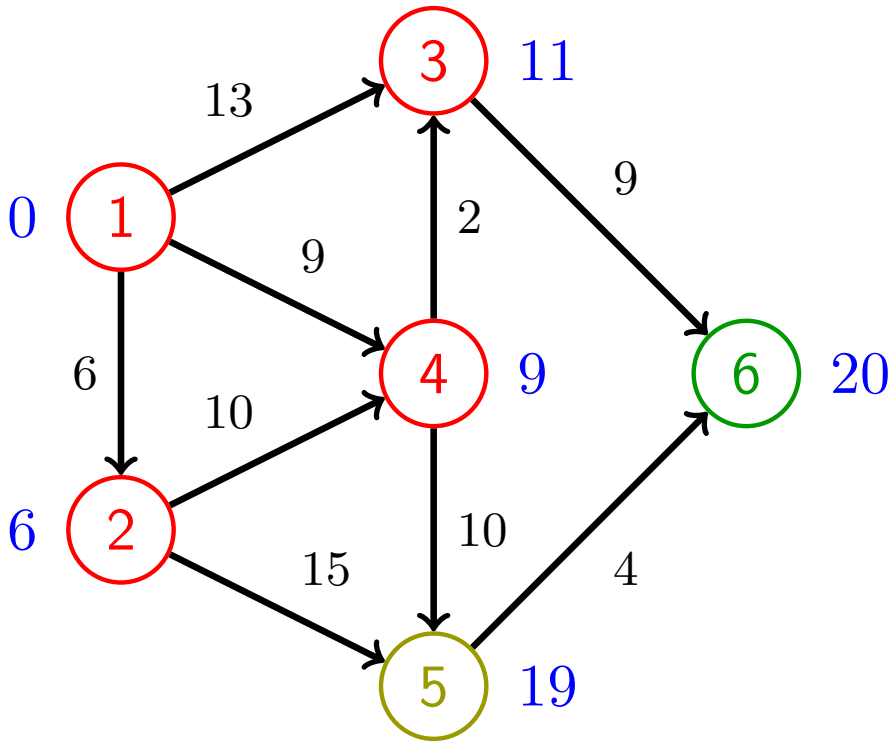

```
pq = [
 (19, 5)
]
```

Dijkstra in action



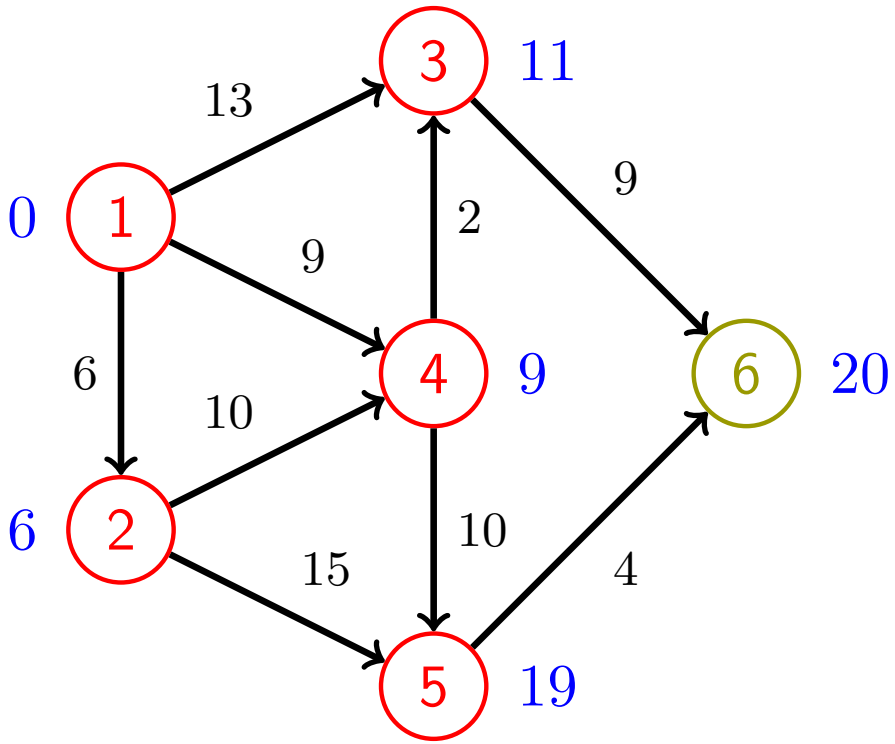

```
pq = [
  (19, 5),
  (20, 6)
]
```

Dijkstra in action



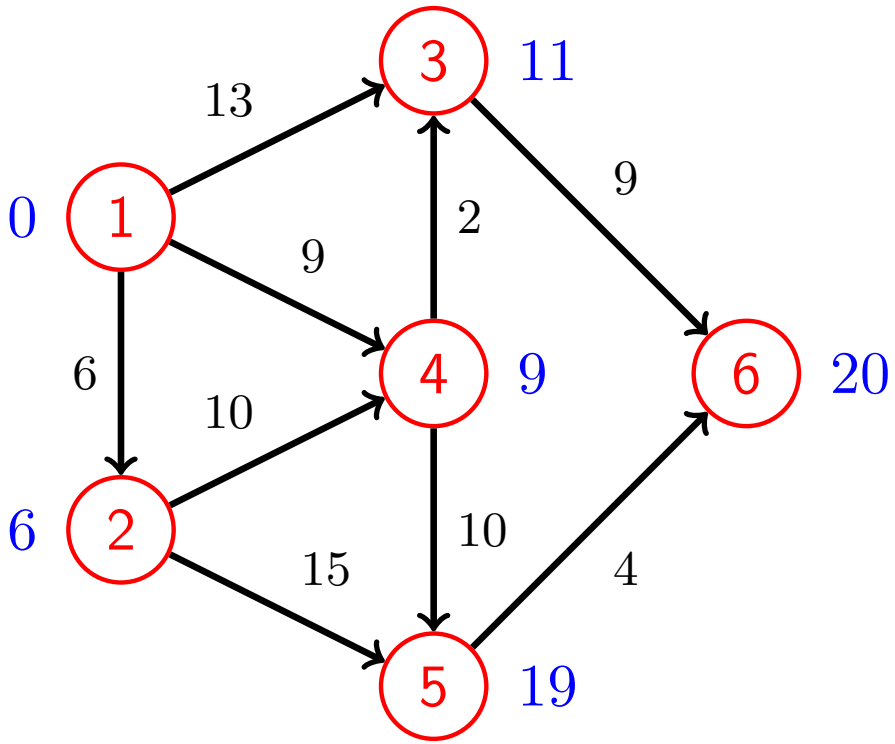

```
pq = [
  (20, 6)
]
```

Dijkstra in action




```
pq = [
]
```

Dijkstra in action




```
pq = [
]
```

Dijkstra útfærsla

- Eins og við gerðum þetta hér að ofan þá lækkuðum við gildin á vigtum í forgangsbiðröðinni.
- C++ STL priority queue (og Java hliðstæðan) bjóða ekki upp á þetta annað en sumar útfærslur.
- Því sleppum við því að lækka það og bætum bara inn öðru stykki með lægri vigt. Ef við rekumst svo á staki í biðröðinni með hærri vigt en er í raun sleppum við því bara.
- Með því að geyma hvaðan við komum í hvern hnút getum við svo líka fengið leiðina sjálfa.
- Þetta allt saman gefur okkur $\mathcal{O}(E + V \log(V))$ keyrslutíma.
- Athugavert er að benda á að ef maður gefur Dijkstra-útfærslu fall sem gefur því hugmynd um hversu nálægt það er lokapunktinum, svokallað 'heuristic function', fæst það sem kallast A* reikniritið sem er notað víða um heim.

Dijkstra útfærsla

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef pair<int,int> ii;
typedef vector<ii> vii;
typedef vector<vii> vvii;

vi dijkstra(vvii& g, int s) {
    vi dist(g.size(), INT_MAX);
    priority_queue<ii> pq;
    dist[s] = 0; pq.push(ii(0, s));
    while(!pq.empty()) {
        auto t = pq.top();
        pq.pop();
        if(-t.first != dist[t.second]) continue;
        for(ii x : g[t.second]) {
            if(dist[t.second] + x.first < dist[x.second]) {
                dist[x.second] = dist[t.second] + x.first;
                pq.push(ii(-dist[x.second], x.second));
            }
        }
    }
    return dist;
}
```

Bellman-Ford

- En hvað ef það er rás með neikvæða vigt í netinu?
- Þá kemur Bellman-Ford reikniritið til bjargar!
- Það er meira að segja töluvert einfaldara reiknirit. Við upphafsstillum allar lengdir eins og í Dijkstra. Svo ítrum við í gegnum alla leggi og styttum vegalengdina milli endapunkta þess ef það er betra. Við gerum þetta jafnoft og við höfum marga hnúta.
- Æfing: sanna að þetta virki alltaf. Góð leið til að hugsa um það er að stysta leið frá a til b fer mest í gegnum jafn marga leggi og við höfum hnúta, svo hún hlýtur að finnast eftir það margar styttingar (ekki formleg sönnun).
- Þetta gefur okkur $\mathcal{O}(VE)$ tímaflækju, töluvert verri en Dijkstra, en það er lítið við því að gera. Þessi útfærsla setur vigtina sem mjög litla tölu ef hægt er að fara í neikvæða rás á leiðinni (þyngdin ætti þá að vera $-\infty$).

Bellman-Ford útfærsla

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef pair<int,int> ii;
typedef vector<ii> vii;
typedef vector<vii> vvii;
typedef long long ll;

vi bellmanford(vvii& g, ll a) {
        vi dist(g.size(), INT_MAX); dist[a] = 0;
          for(int i = 0; i < g.size() - 1; ++i) {
                  for(int j = 0; j < g.size(); ++j) {
                          if(dist[j] == INT_MAX) continue;
                          for(ii x : g[j]) dist[x.second] = min(dist[x.second], dist[j] + x.first);
                  }
          }
          while(true) {
                  bool done = true;
                    for(int i = 0; i < g.size(); ++i) {
                      if(dist[i] == INT_MAX) continue;
                      for(ii x : g[i]) {
                                if(dist[x.second] == -INT_MAX) continue;
                                if(dist[i] + x.first < dist[x.second]) {
                                 dist[x.second] = -INT_MAX;
                                  done = false;
                                  }
                          }
                   }
                 if(done) break;
        }
        return dist;
}
```

Floyd-Warshall

- Þetta reiknirit er ekki ósvipað Bellman-Ford, en ætlun þess er að reikna út stystu fjarlægð milli sérhverra tveggja punkta.
- Svipað og Bellman-Ford notar það kvika bestun. Það setur upp töflu $S[i][j][k]$ sem er stysta fjarlægð frá i til j með því að fara bara í gegnum hnútana $1, 2, \ldots, k$.
- Ef við viljum reikna út $S[i][j][k]$ tökum við eftir því að annað hvort fer stysti vegurinn frá i til j gegnum k eða ekki. Ef ekki er þetta jafnt $S[i][j][k-1]$. Ef svo er þá fer vegurinn bara einu sinni gegnum k svo svarið verður $S[i][k][k-1] + S[k][j][k-1]$ því við ferðumst fyrst frá i í k með hnútunum $1, \ldots, k-1$ og svo frá k í j með hnútunum $1, \ldots, k-1$.
- Með það í huga að við þurfum að reikna grunntilvikið $S[i][j][0]$ sérstaklega þá getum við útfært þetta með þrefaldri for-lykkju sem gefur $\mathcal{O}(V^3)$ keyrslutíma. Við þurfum líka að passa neikvæðar rásir, en tékka má á þeim með því að skoða hvort $S[i][i][n]$ sé einhverntímann neikvætt (æfing!)

Floyd-Warshall útfærsla

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef vector<ll> vl;
typedef vector<vl> vvl;
typedef pair<ll,ll> lll;
typedef vector<lll> vll;
typedef vector<vll> vvll;
#define rep(i,a,b) for(auto i=(a); i != (b); ++i)

vvl floydwarshall(vvll& g) {
  ll n = g.size();
  vvl dp(n, vl(n, LLONG_MAX / 4));
  rep(i, 0, n) dp[i][i] = 0;
  rep(i, 0, n) for(lll x : g[i])
    dp[i][x.second] = min(x.first, dp[i][x.second]);
  rep(k, 0, n) rep(i, 0, n) rep(j, 0, n) {
    if(dp[i][k] == LLONG_MAX / 4 ||
      dp[k][j] == LLONG_MAX / 4) continue;
    dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]); }
  rep(k, 0, n) rep(i, 0, n) rep(j, 0, n) {
    if(dp[i][k] == LLONG_MAX / 4 || dp[k][j] == LLONG_MAX / 4 ||
      dp[i][j] == LLONG_MAX / 4) continue;
    if(dp[i][k] + dp[k][j] < dp[i][j]) dp[i][j] = -(LLONG_MAX / 4); }
  return dp; }
```

Flæðadót

- Fyrir áhugasama mæli ég eindregið með að lesa kaflann um flæði í bókinni eða tala við okkur.
- Dæmi um flæði verða merkt sérstaklega sem ítarefni en mörg erfið keppnisforritunardæmi byggja á því að leysa flæðisdæmi (annað hvort beint eða með því að nota það til að leysa spyrðingar).
- Reiknirit sem mætti þá kynna sér eru Edmond-Karp, Dinic, Hopcroft-Karp o.fl.