Netafræði II

Stefnd og vigtuð net

Atli Fannar Franklín

24. febrúar 2019

Efnisyfirlit

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.

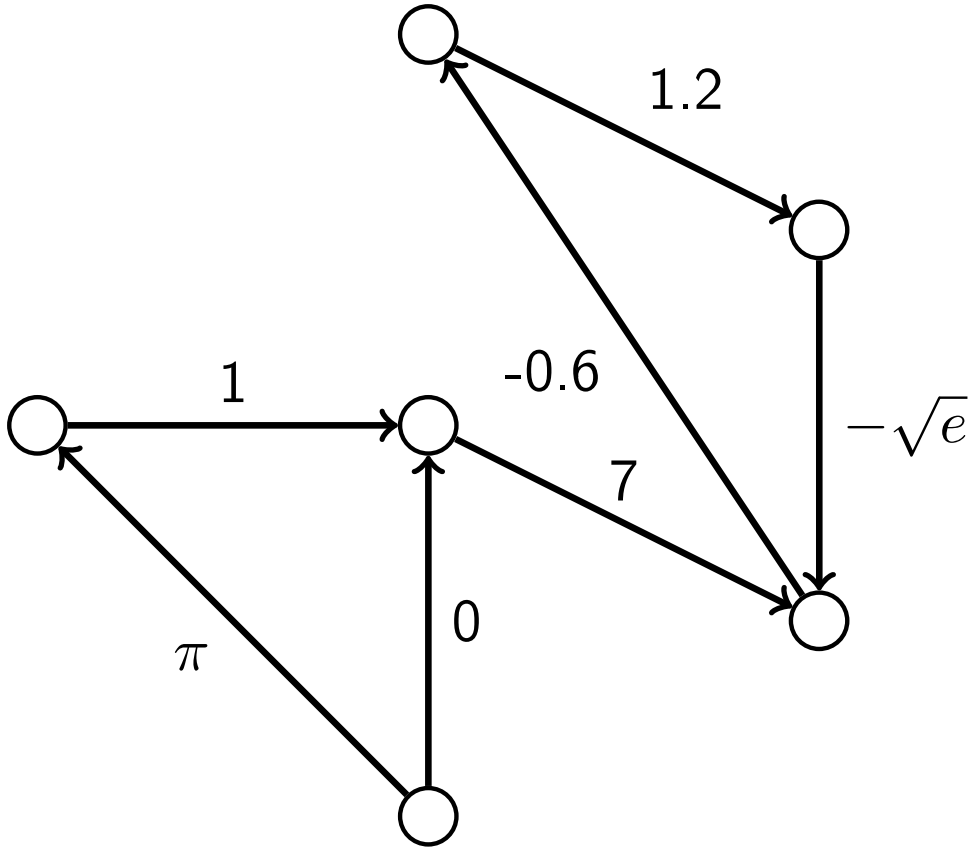
Hvað er stefnt eða vigtað net?

- 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.

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 geyumum 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.

Framsetning

- Nágrannaframseting er geymd með sama hætti í stefndu neti nema við geyumum 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.

Framsetning

- Nágrannaframseting er geymd með sama hætti í stefndu neti nema við geyumum 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.

Efnisyfirlit

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.

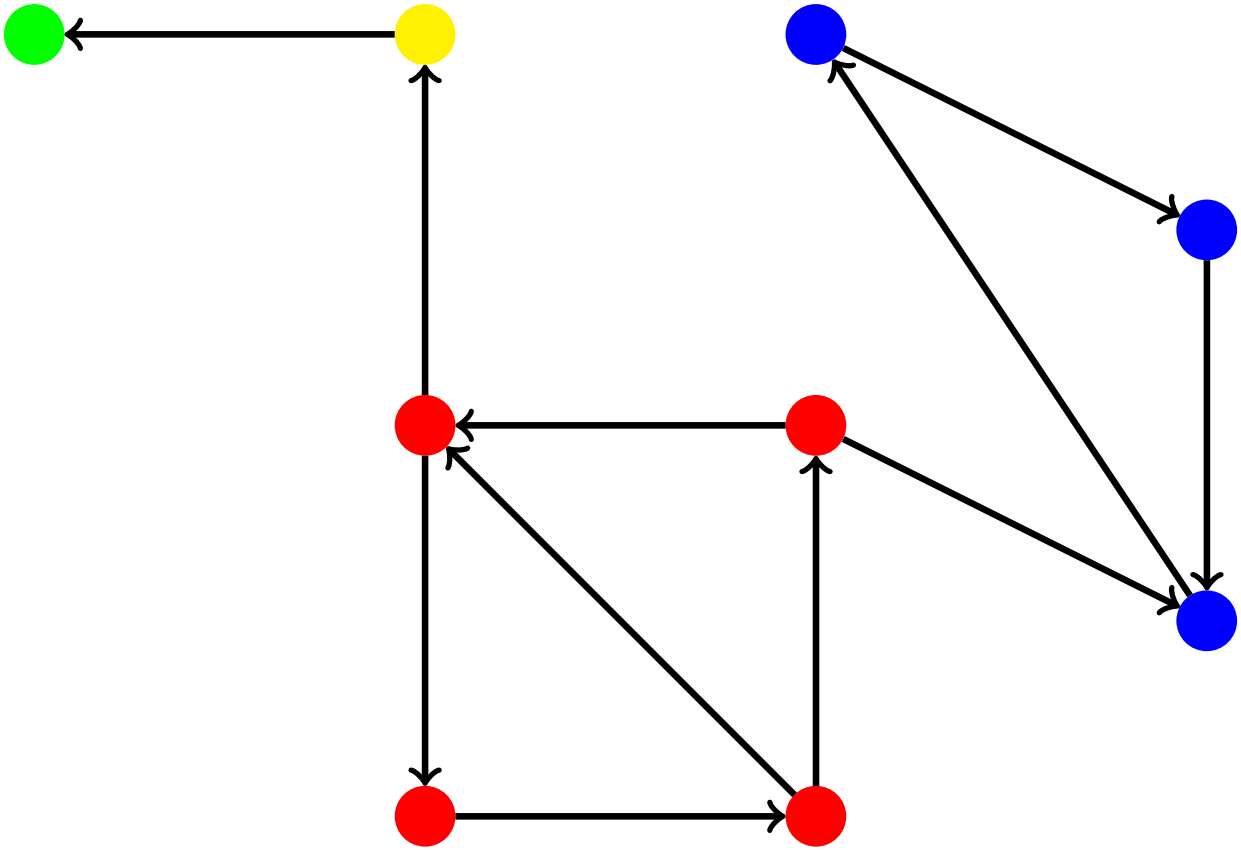
Hvað getum við gert nýtt á stefndu neti?

- 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.

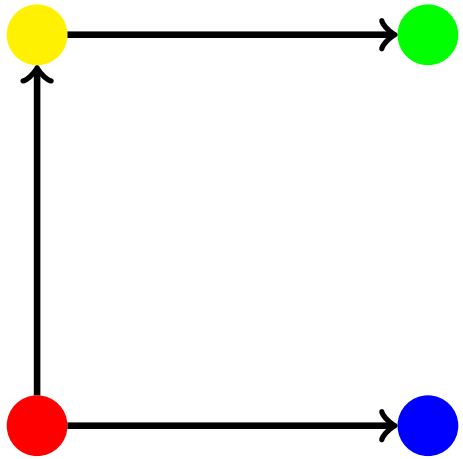
Hvað getum við gert nýtt á stefndu neti?

- 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?

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.

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.

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.

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.

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.

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!

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 gerir í raun það sama og að Tarjan, bara með minna bókhaldi. En einnig er hægt að nota reiknirit Kahn's (sjá bók). En þar sem öll þessi reiknirit hafa sömu tímaflækju ($\mathcal{O}(V+E)$), þá látum við þetta duga.

Efnisyfirlit

Spannandi tré

- Fyrsta sem manni dytti í hug væri líklegast að skoða ódýrustu leiðina milli staða. En í ljós kemur að það sé lítill munur á að skoða þetta á stefndu og óstefndu neti ef það er vigtað, svo skoðum það á eftir.

Spannandi tré

- Fyrsta sem manni dytti í hug væri líklegast að skoða ódýrustu leiðina milli staða. En í ljós kemur að það sé lítill munur á að skoða þetta á stefndu og óstefndu neti ef það er vigtað, svo skoðum það á eftir.
- Það sem við getum gert núna er að skoða spurninguna 'Hver er ódýrasta leiðin til að tengja alla hnúta netsins saman?'

Spannandi tré

- Fyrsta sem manni dytti í hug væri líklegast að skoða ódýrustu leiðina milli staða. En í ljós kemur að það sé lítill munur á að skoða þetta á stefndu og óstefndu neti ef það er vigtað, svo skoðum það á eftir.
- 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.

Spannandi tré

- Fyrsta sem manni dytti í hug væri líklegast að skoða ódýrustu leiðina milli staða. En í ljós kemur að það sé lítill munur á að skoða þetta á stefndu og óstefndu neti ef það er vigtað, svo skoðum það á eftir.
- 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. Er einhver með gisk?

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. Er einhver með gisk?
- Við bætum alltaf við ódýrasta leggnum sem myndar ekki rás!

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. Er einhver með gisk?
- Við bætum alltaf við ódýrasta leggnum sem myndar ekki rás!
- Æfing: sanna að þetta virki.

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. Er einhver með gisk?
- 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.

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. Er einhver með gisk?
- 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.

Hríslur

- Svipað má gera yfir stefnd net.
- Þá skoðar maður svokallaðar spannandi hríslur.

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.

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.
- Gott að læra það einhvern tímann samt, eins og við komumst að síðastliðinn nóvember.

Efnisyfirlit

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.

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.

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?

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!

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.

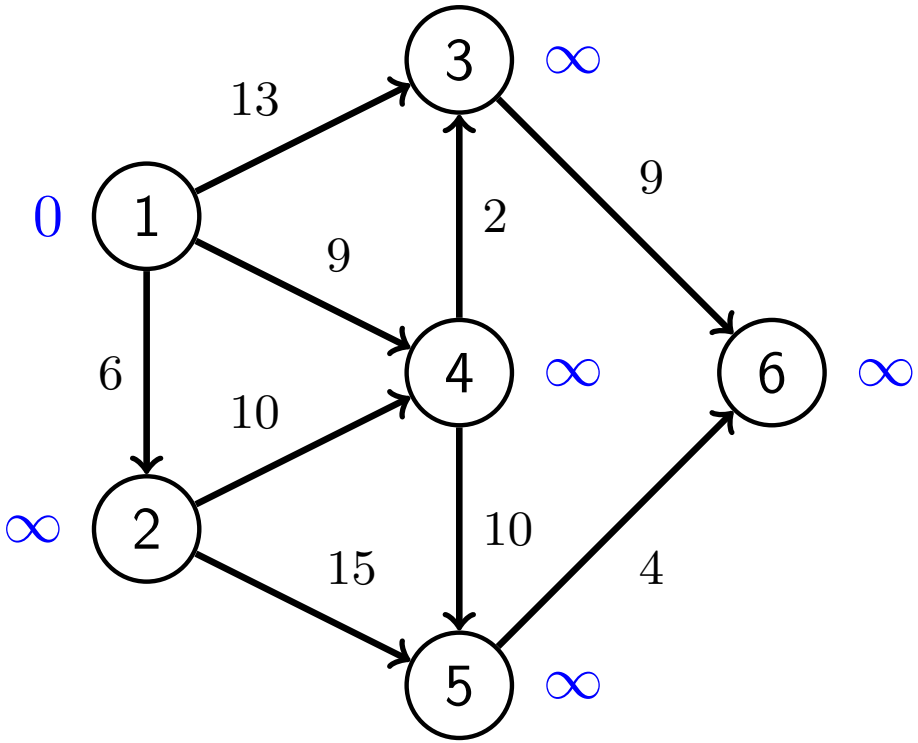
Dijkstra

- 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.

Dijkstra

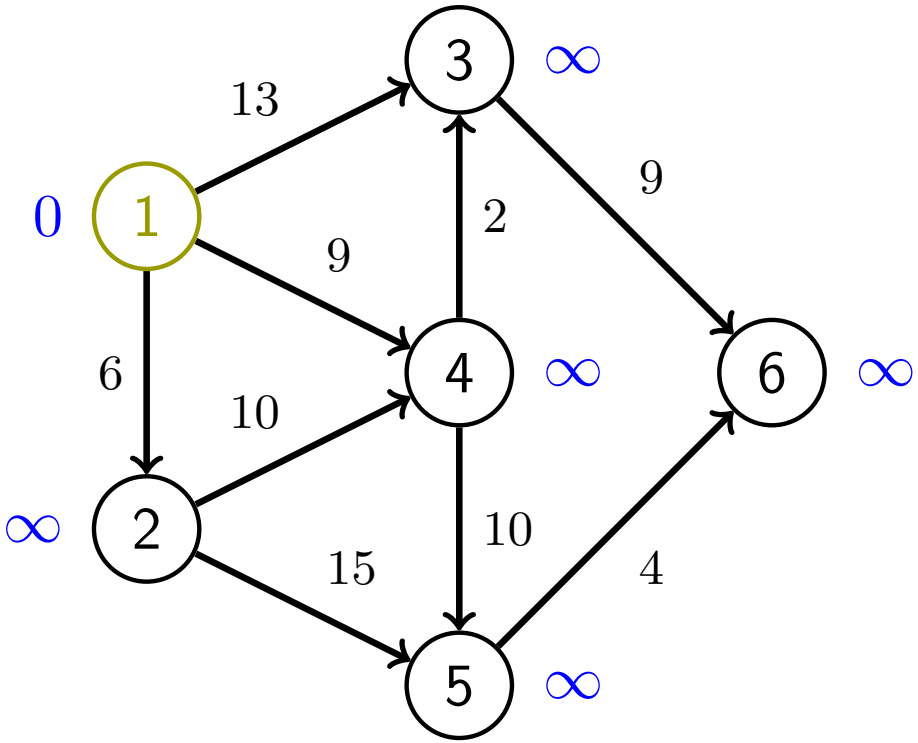
- Dijkstra finnur stystu leið í alla hnúta frá einum hnút v. Það byrjar á því að segja að þessi fjarlægð sé ∞ fyrir alla hnúta og að hnúturinn sem við komum úr sé enginn.
- Við geymum svo hnútana sem við eigum eftir að skoða í forgangsbiðröð þar sem forgangurinn er stutt í þá.
- Svo meðan biðröðin er 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



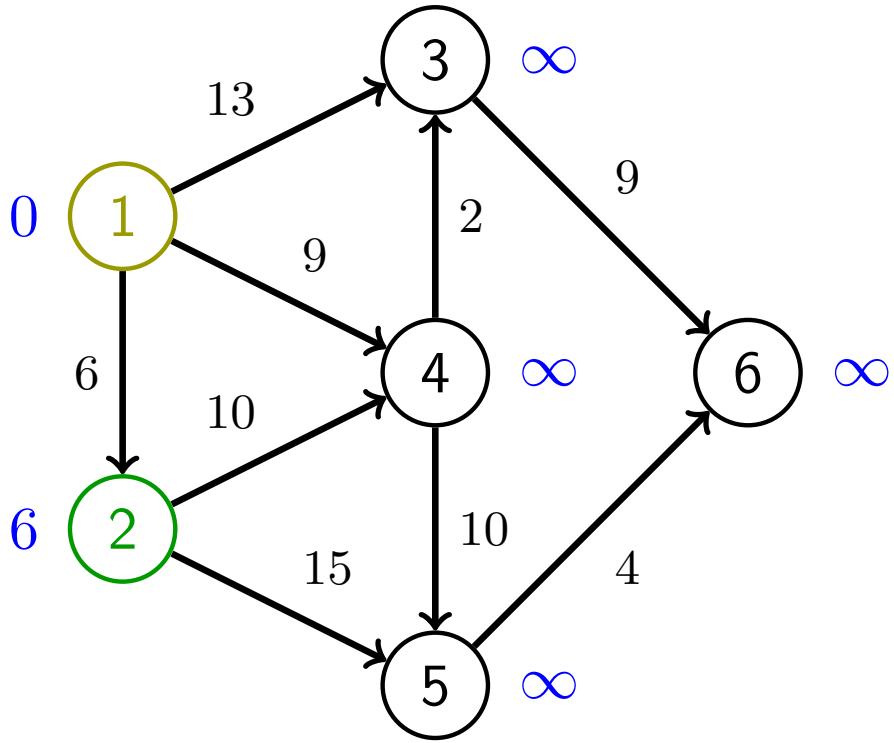

```
pq = [
  (0, 0)
]
```

Dijkstra in action



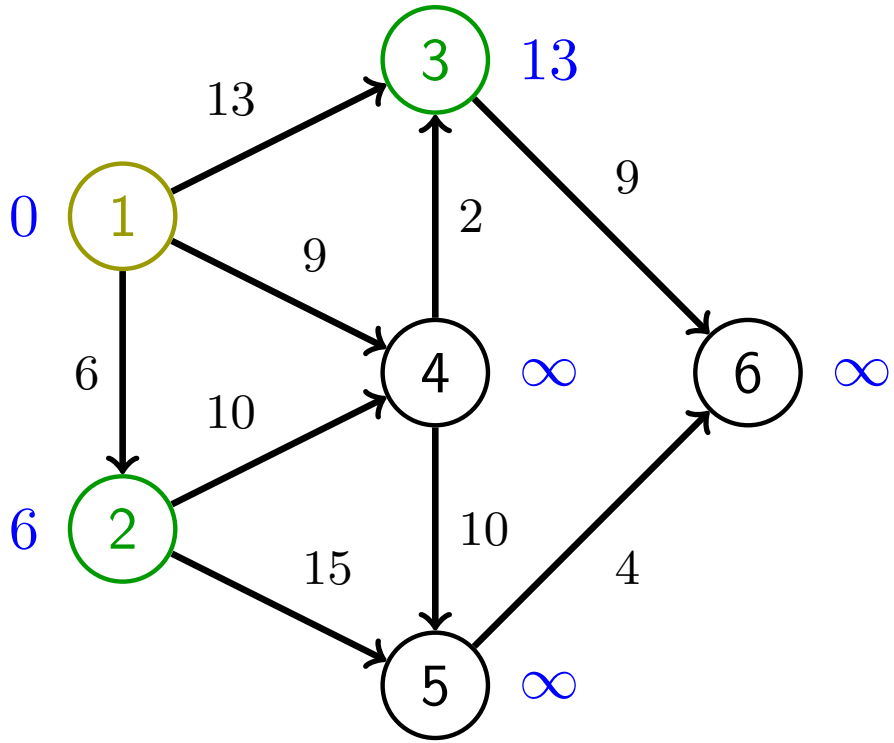

```
pq = [
]
```

Dijkstra in action



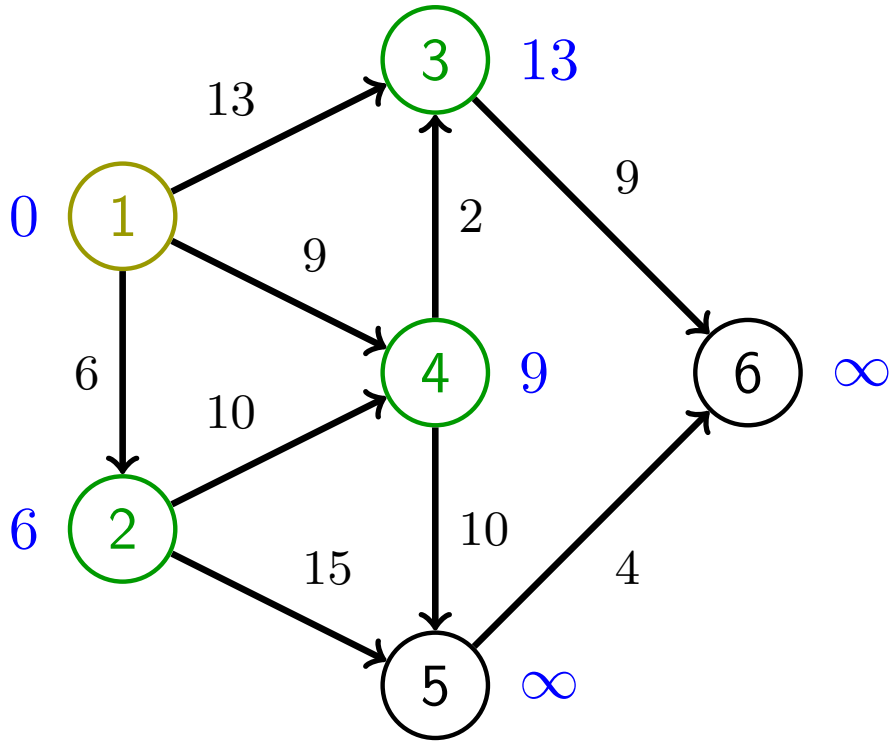

```
pq = [
  (6, 2)
]
```

Dijkstra in action



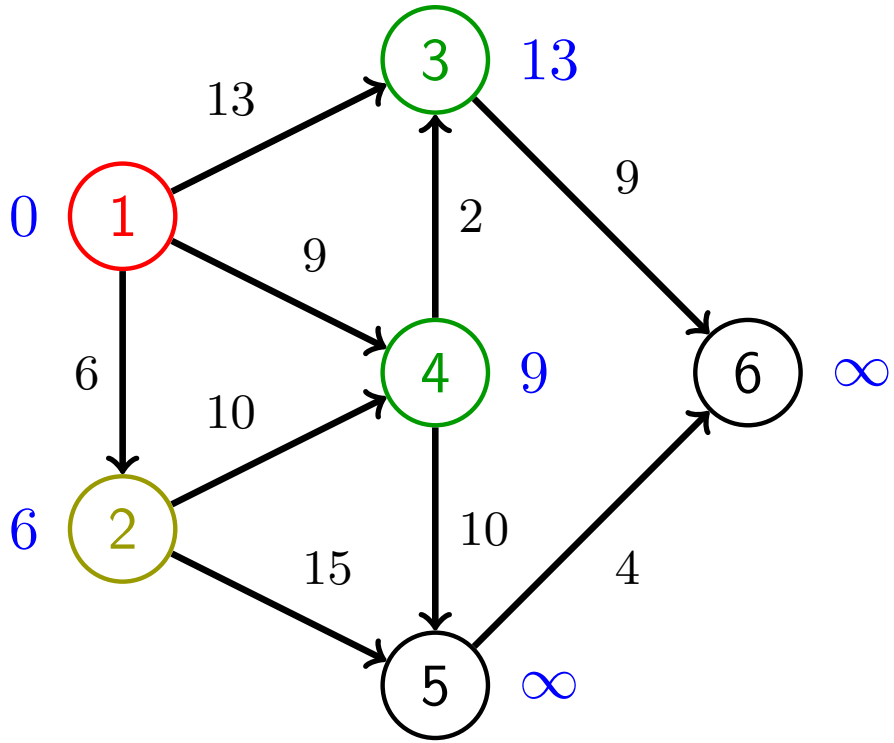

```
pq = [
  (6, 2),
  (13, 3)
]
```

Dijkstra in action



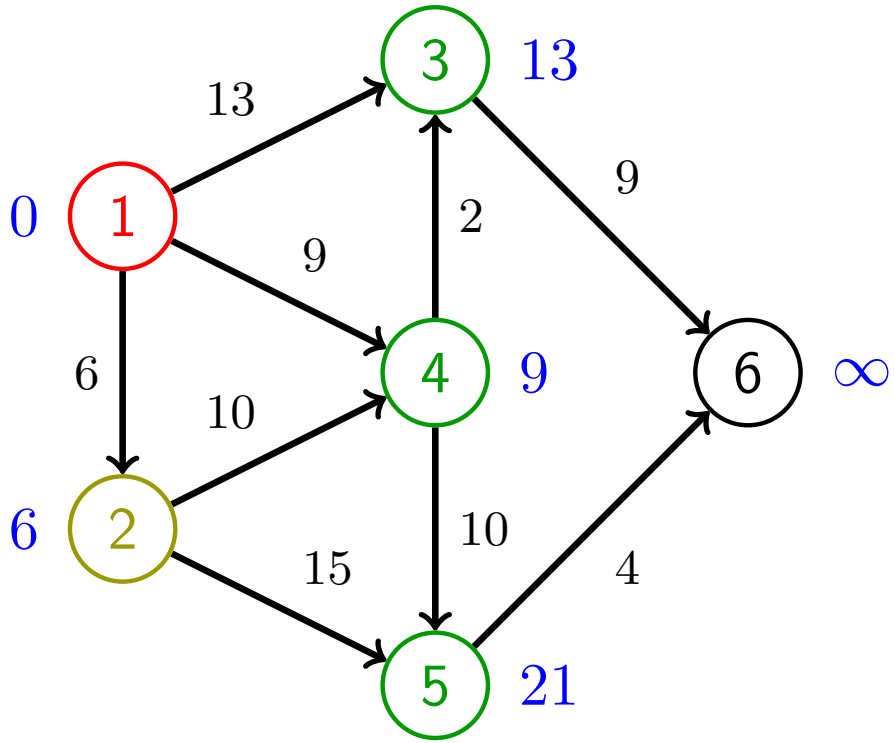

```
pq = [
  (6, 2),
  (9, 4),
  (13, 3)
]
```

Dijkstra in action



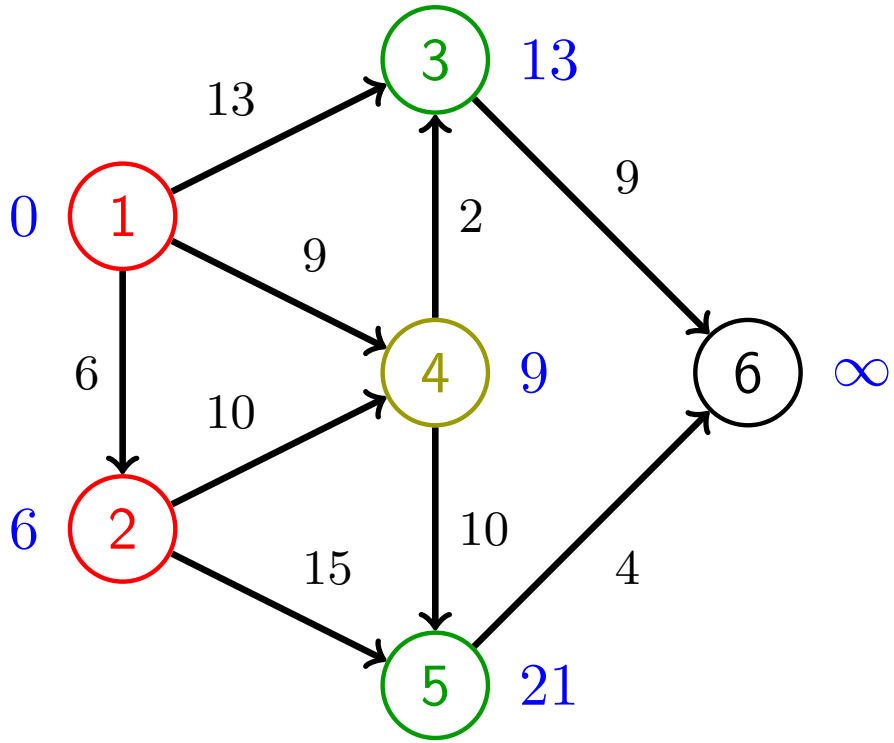

```
pq = [
  (9, 4),
  (13, 3)
]
```

Dijkstra in action



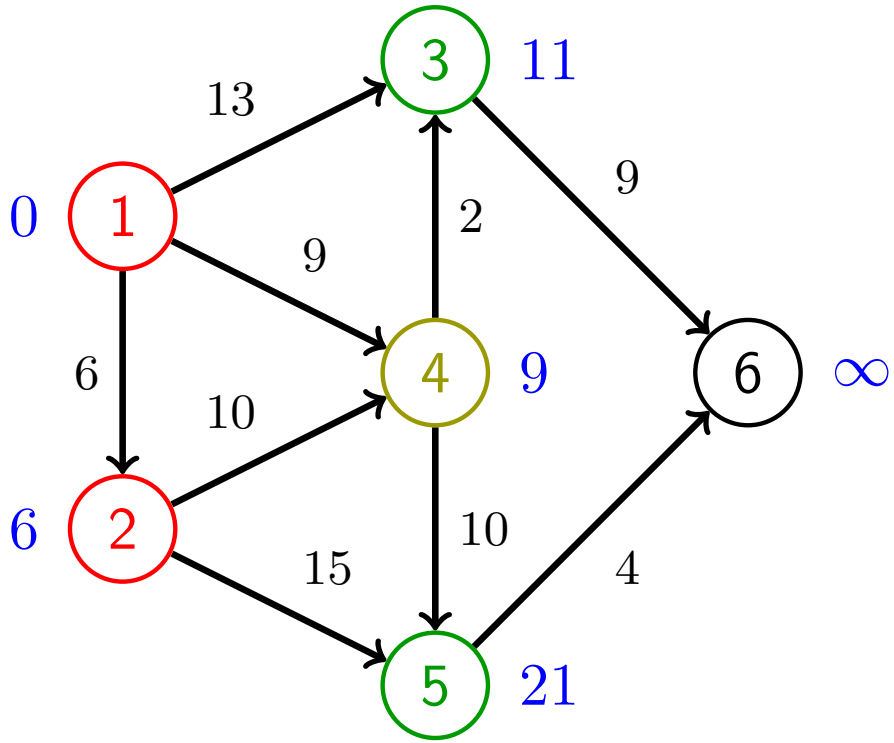

```
pq = [
  (9, 4),
  (13, 3),
  (21, 5)
]
```

Dijkstra in action



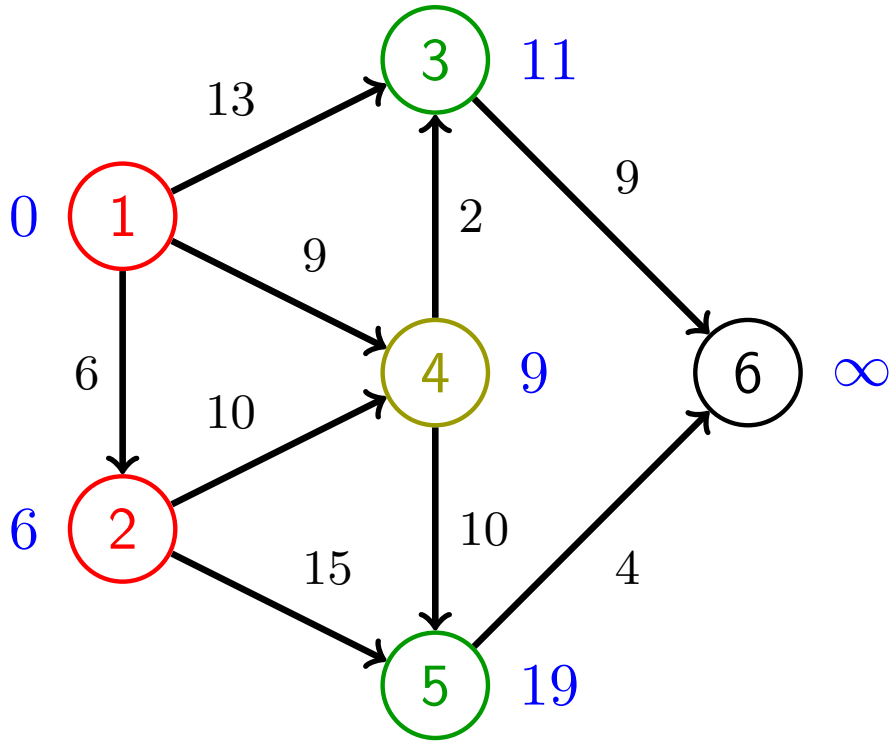

```
pq = [
  (13, 3),
  (21, 5)
]
```

Dijkstra in action



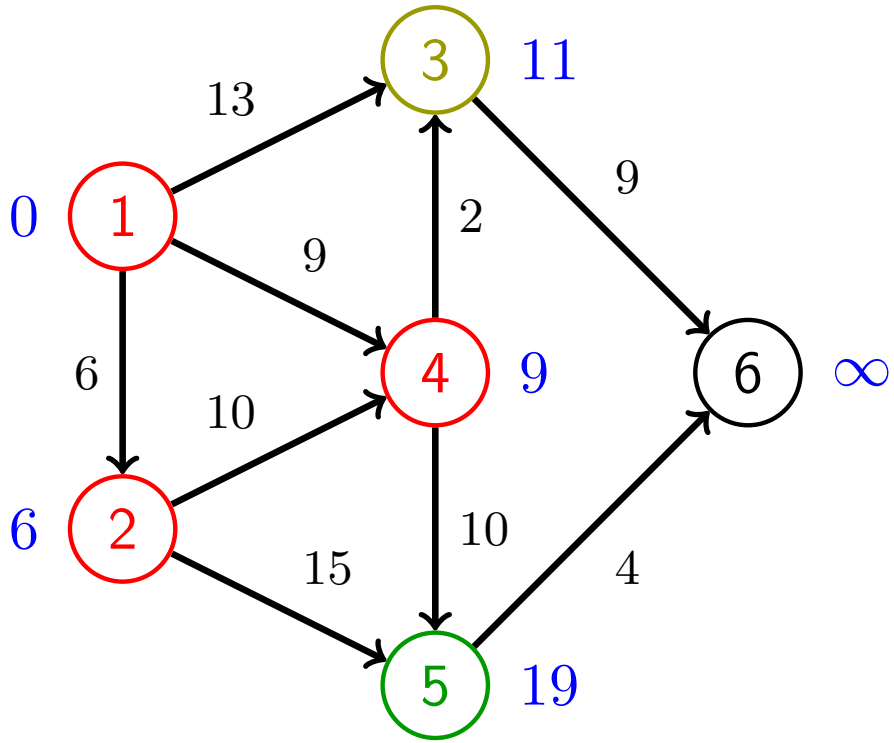

```
pq = [
  (11, 3),
  (21, 5)
]
```

Dijkstra in action



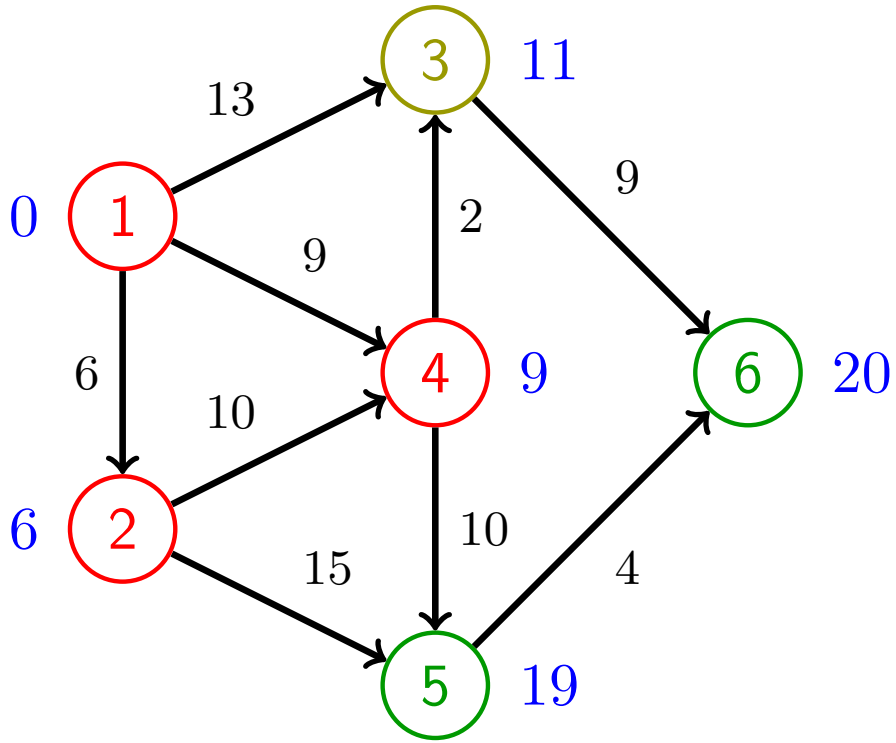

```
pq = [
  (11, 3),
  (19, 5)
]
```

Dijkstra in action



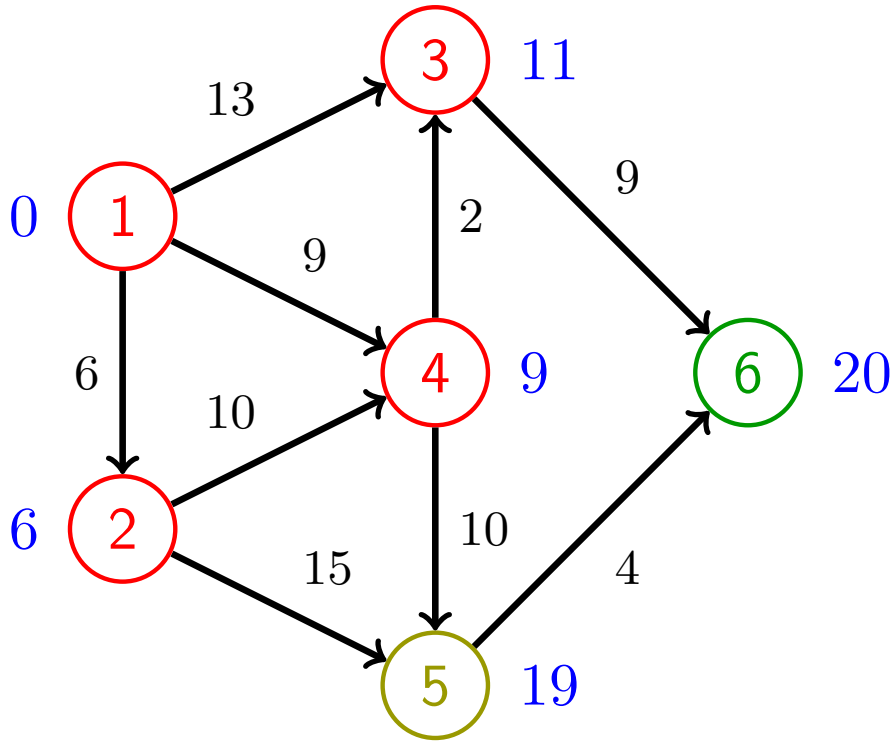

```
pq = [
  (19, 5)
]
```

Dijkstra in action



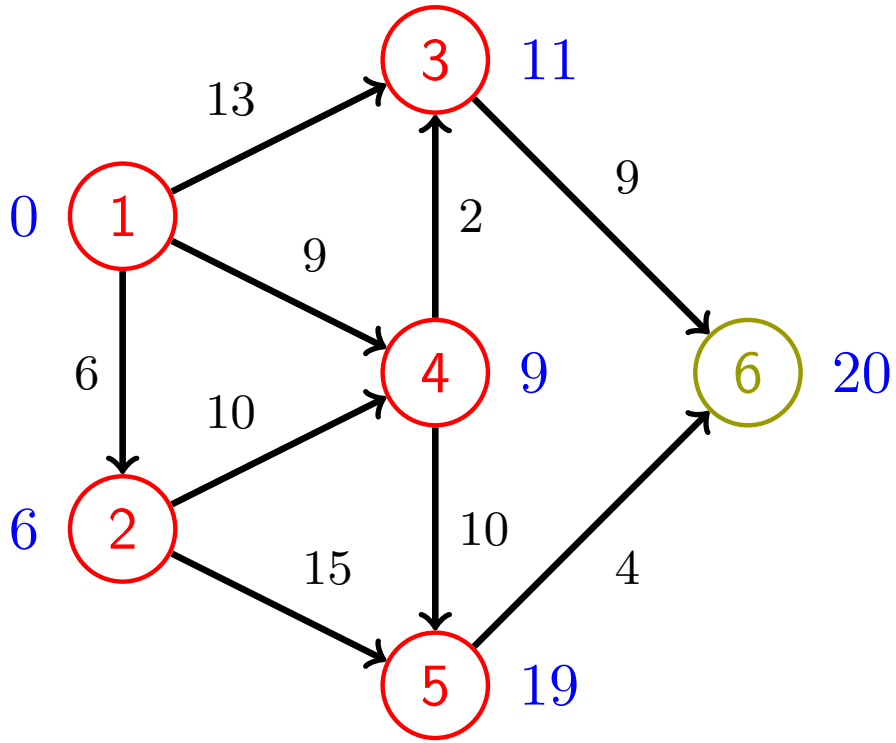

```
pq = [
  (19, 5),
  (20, 6)
]
```

Dijkstra in action



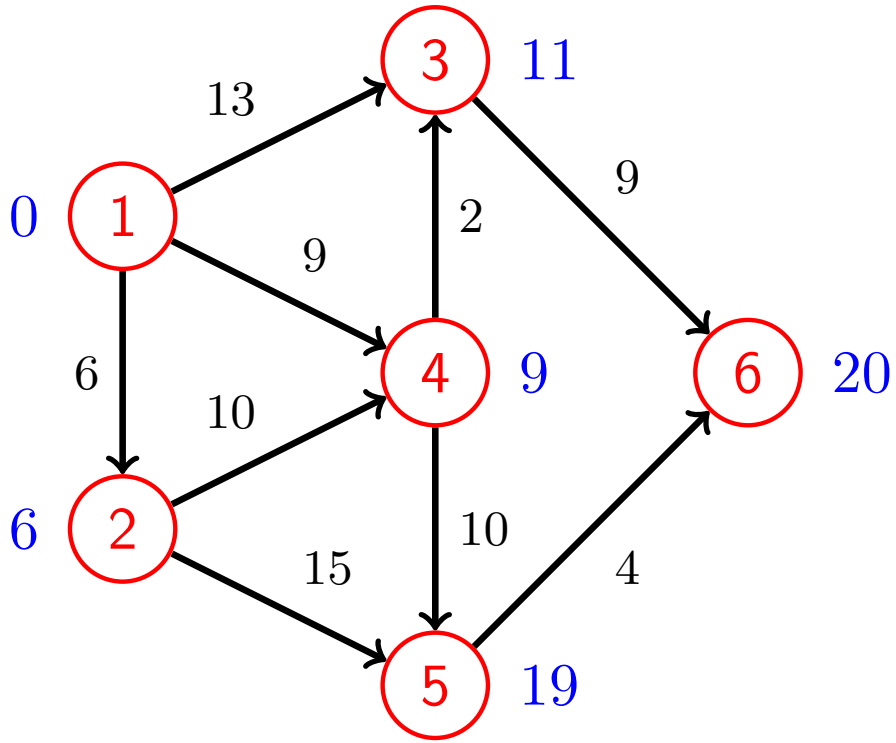

```
pq = [
  (20, 6)
]
```

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.

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.

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.

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.

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.

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

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef pair<double,int> di;
typedef vector<di> vdi;
typedef vector<vdi> vvdi;
typedef vector<double> vd;

pair<vi, double> dijkstra(vvdi& g, int a, int b) {
    vd dist(g.size(), DBL_MAX);
    vi prv(g.size(), -1);
    priority_queue<di> pq;
    dist[a] = 0; pq.push(di(0, a));
    while(!pq.empty()) {
        auto t = pq.top();
        pq.pop();
        if(-t.first != dist[t.second]) continue;
        for(di x : g[t.second]) {
            if(dist[t.second] + x.first < dist[x.second]) {
                dist[x.second] = dist[t.second] + x.first;
                prv[x.second] = t.second;
                pq.push(di(-dist[x.second], x.second));
            }
        }
    }
    if(prv[b] == -1) return make_pair(vi(), DBL_MAX);
    vi path(1, b);
    while(path.back() != a) path.push_back(prv[path.back()]);
    reverse(path.begin(), path.end());
    return make_pair(path, dist[b]);
}
```

Prófun á Dijkstra

```
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));
    }
    auto res = dijkstra(g, 0, n - 1);
    cout << res.second << endl;
    for(int x : res.first) cout << x << ' ';
    cout << endl;
}
```

```
// Inntak
6 9
0 1 6
0 2 13
0 3 9
1 3 10
1 4 15
2 5 9
3 2 2
3 4 10
4 5 4
// Úttak
20
0 3 2 5
```

Bellman-Ford

- En hvað ef það er rás með neikvæða vigt í netinu?

Bellman-Ford

- En hvað ef það er rás með neikvæða vigt í netinu?
- Þá kemur Bellman-Ford reiknirítið til bjargar!

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 stuttum vegalengdina milli endapunkta þess ef það er betra. Við gerum þetta jafnoft og við höfum marga hnúta.

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 stuttum 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).

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 stuttum 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. Sjáum nú útfærslu sem skilar fjarlægð í alla aðra hnúta með heiltölufjarlægðum til tilbreytingar.

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;

vi bellmanford(vvii& g, int a, bool& negcyc) {
    negcyc = false;
    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);
            }
        }
    }
    for(int i = 0; i < g.size(); ++i) {
        for(ii x : g[i]) {
            if(dist[i] + x.first < dist[x.second]) negcyc = true;
        }
    }
    return dist;
}
```

Floyd-Warshall

- Þetta reiknirit er ekki ósvipað Bellman-Ford, en ætlun þess er að reikna út stystu fjarlægð milli sérhverra tveggja punkta.

Floyd-Warshall

- Þetta reiknirit er ekki ósvipað Bellman-Ford, en ætlun þess er að reikna út stystu fjarlægð milli sérhverra tveggja punkta.
- Það sem það er gerir öðruvísi hins vegar er að það notar 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 $1, 2, \ldots, k$.

Floyd-Warshall

- Þetta reiknirit er ekki ósvipað Bellman-Ford, en ætlun þess er að reikna út stystu fjarlægð milli sérhverra tveggja punkta.
- Það sem það er gerir öðruvísi hins vegar er að það notar 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 $1, \ldots, k-1$.

Floyd-Warshall

- Þetta reiknirit er ekki ósvipað Bellman-Ford, en ætlun þess er að reikna út stystu fjarlægð milli sérhverra tveggja punkta.
- Það sem það er gerir öðruvísi hins vegar er að það notar 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

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

vvi floydwarshall(vvii& g) {
    int n = g.size();
    // sækjum aldrei k-gildi nema einum til baka svo við
    // látum tvívítt duga og yfirskrifum bara jafnóðum
    vvi dp(n, vi(n, INT_MAX));
    for(int i = 0; i < n; ++i) {
        dp[i][i] = 0;
    }
    for(int i = 0; i < n; ++i) {
        for(ii x : g[i]) {
            dp[i][x.second] = x.first;
        }
    }
    for(int k = 0; k < n; ++k) {
        for(int i = 0; i < n; ++i) {
            for(int j = 0; j < n; ++j) {
                if(dp[i][k] == INT_MAX || dp[k][j] == INT_MAX) continue;
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
            }
        }
    }
    return dp;
}
```

Efnisyfirlit

Flæðadót

- Fyrir áhugasama mæli ég eindregið með að lesa kaflann um flæði í bókinni eða tala við okkur.

Flæðadót

- Fyrir áhugasama mæli ég eindregið með að lesa kaflann um flæði í bókinni eða tala við okkur.
- Dæmi um flæði verða ekki sett fyrir en mörg erfið keppnisforritunardæmi byggja á því að leysa flæðisdæmi (annað hvort beint eða með því að nota það til að leysa spyrðingar).

Flæðadót

- Fyrir áhugasama mæli ég eindregið með að lesa kaflann um flæði í bókinni eða tala við okkur.
- Dæmi um flæði verða ekki sett fyrir 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 mæti þá kynna sér eru Edmond-Karp, Dinic, Hopcroft-Karp o.fl.