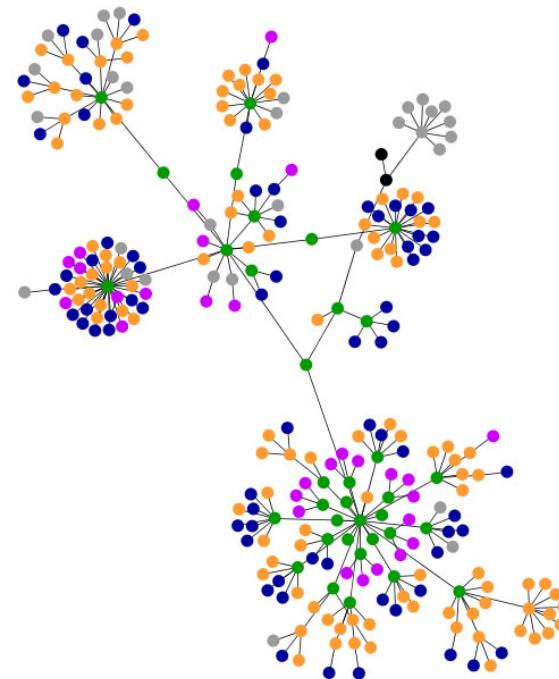


13 Graafit

Seuraavaksi tutustutaan tietorakenteeseen, jonka muodostavat pisteet ja niiden välille muodostetut yhteydet – graafiin.

Keskitymme myös tyypillisimpiin tapoihin etsiä tietoa graafista eli graafialgoritmeihin.

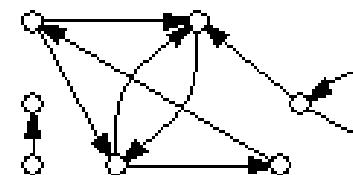
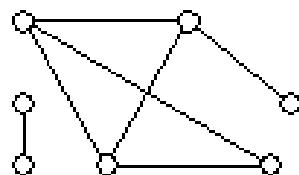


Kuva: Flickr, Rachel D

13.1 Graafien esittäminen tietokoneessa

Graafi on ohjelmistotekniikassa keskeinen rakenne, joka koostuu solmuista (*vertex*, *node*) ja niitä yhdistävistä kaarista (*edge*, *arc*).

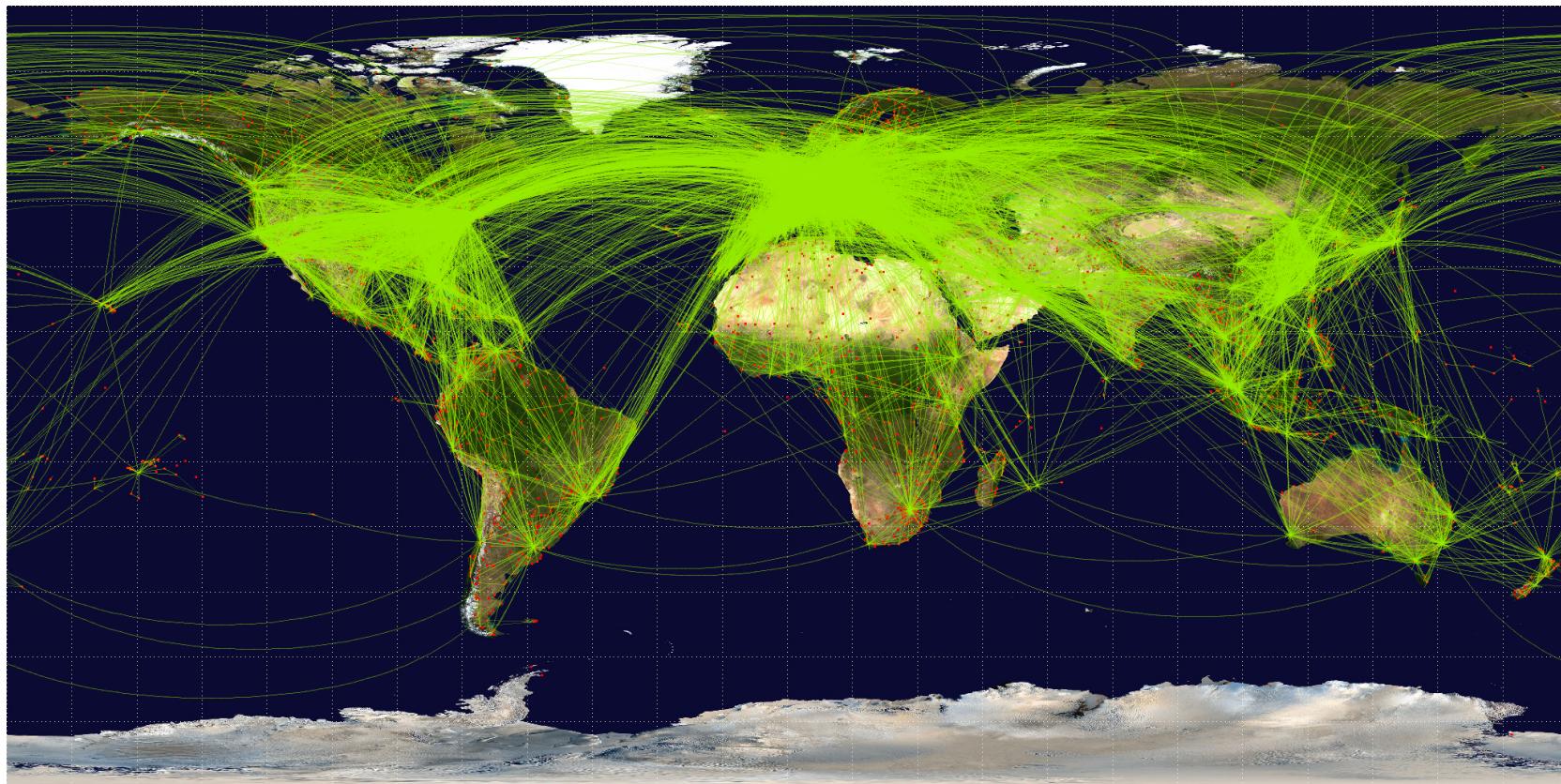
Graafi voi olla suuntaamatonta (*undirected*) tai suunnattua (*directed*).



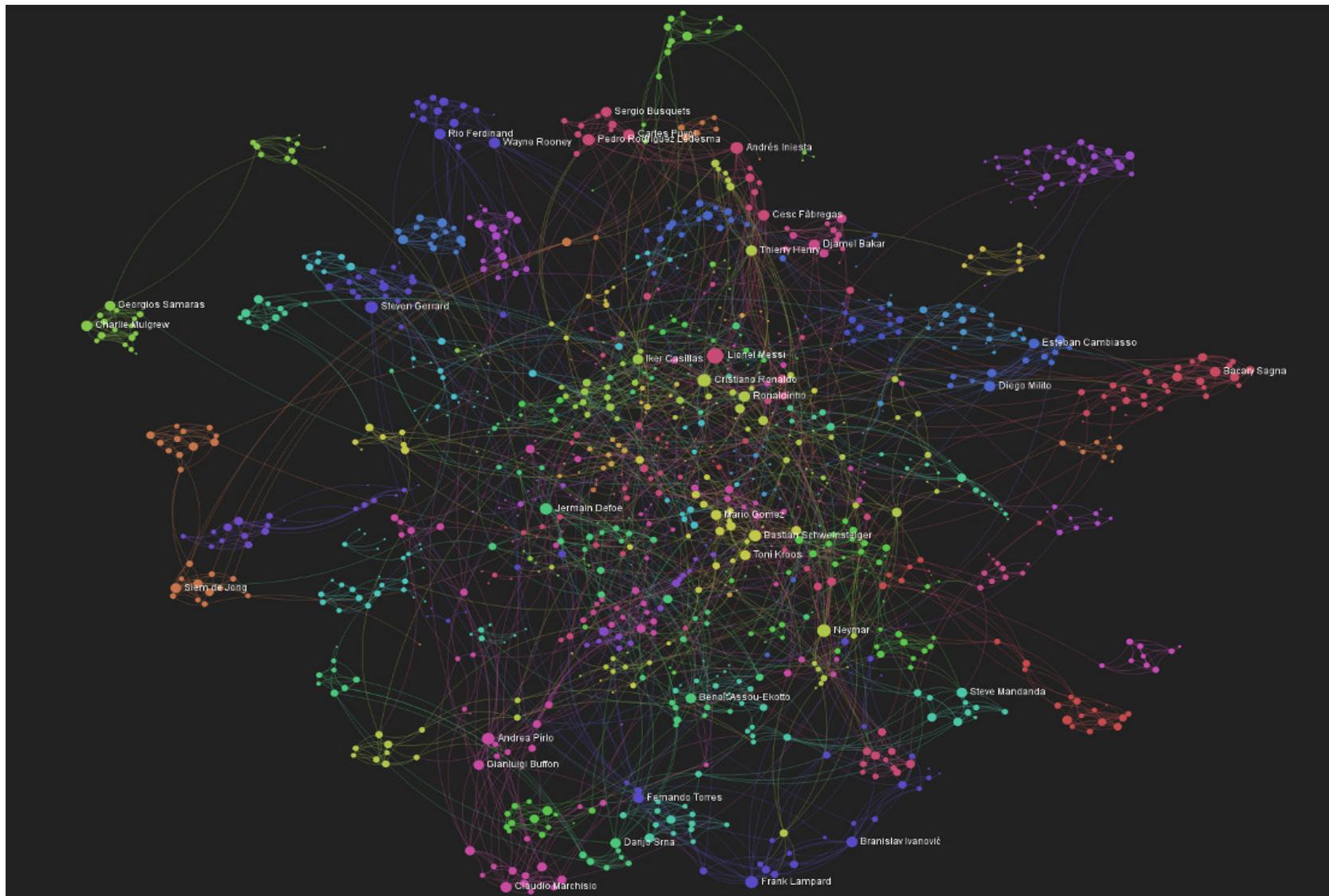


Kuva 18: Kuva: Getty Images

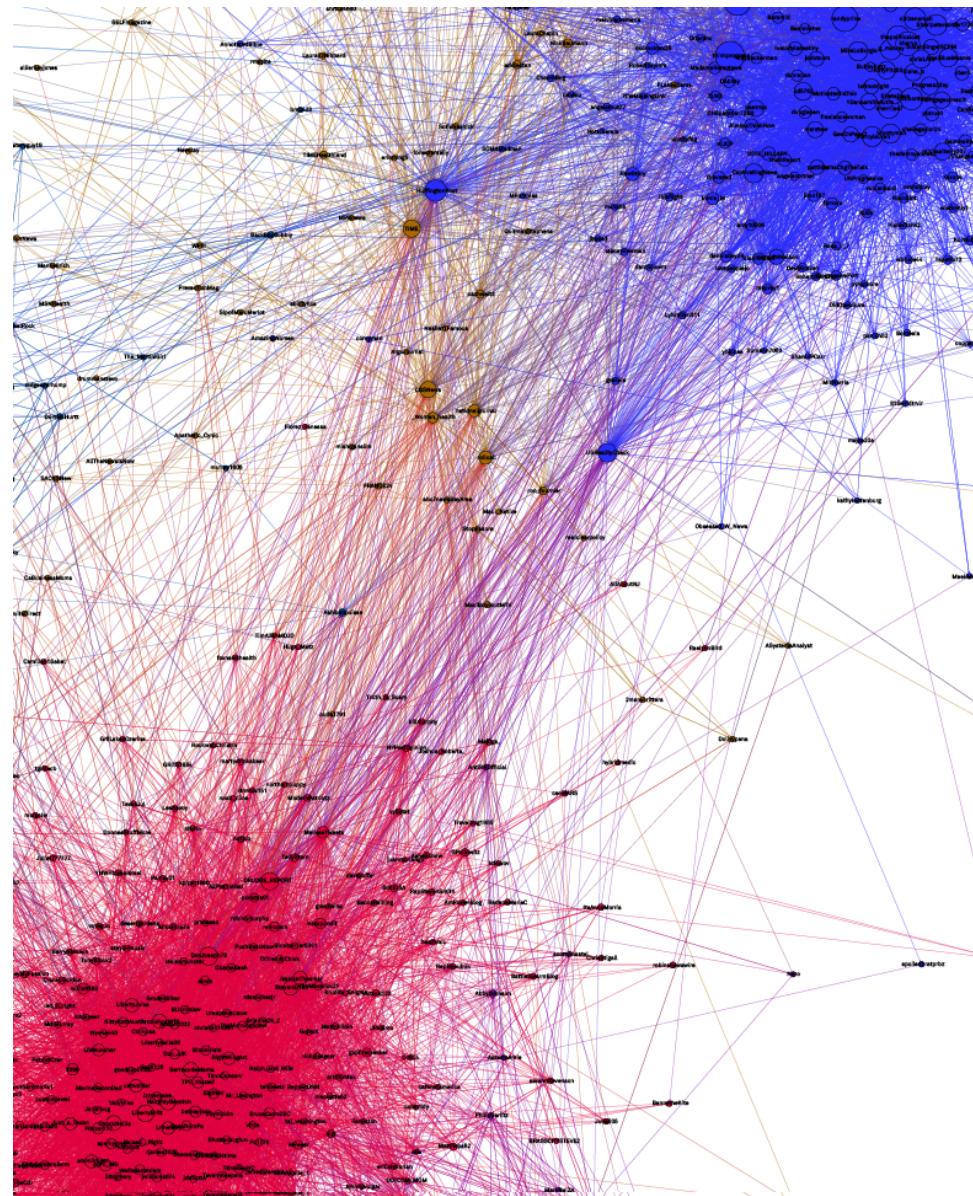
- kaaviokuvat voidaan usein ajatella graafeiksi
- asioiden välisiä suhteita voi usein esittää graafeina
- monet tehtävät voidaan palauttaa graafitehtäviksi



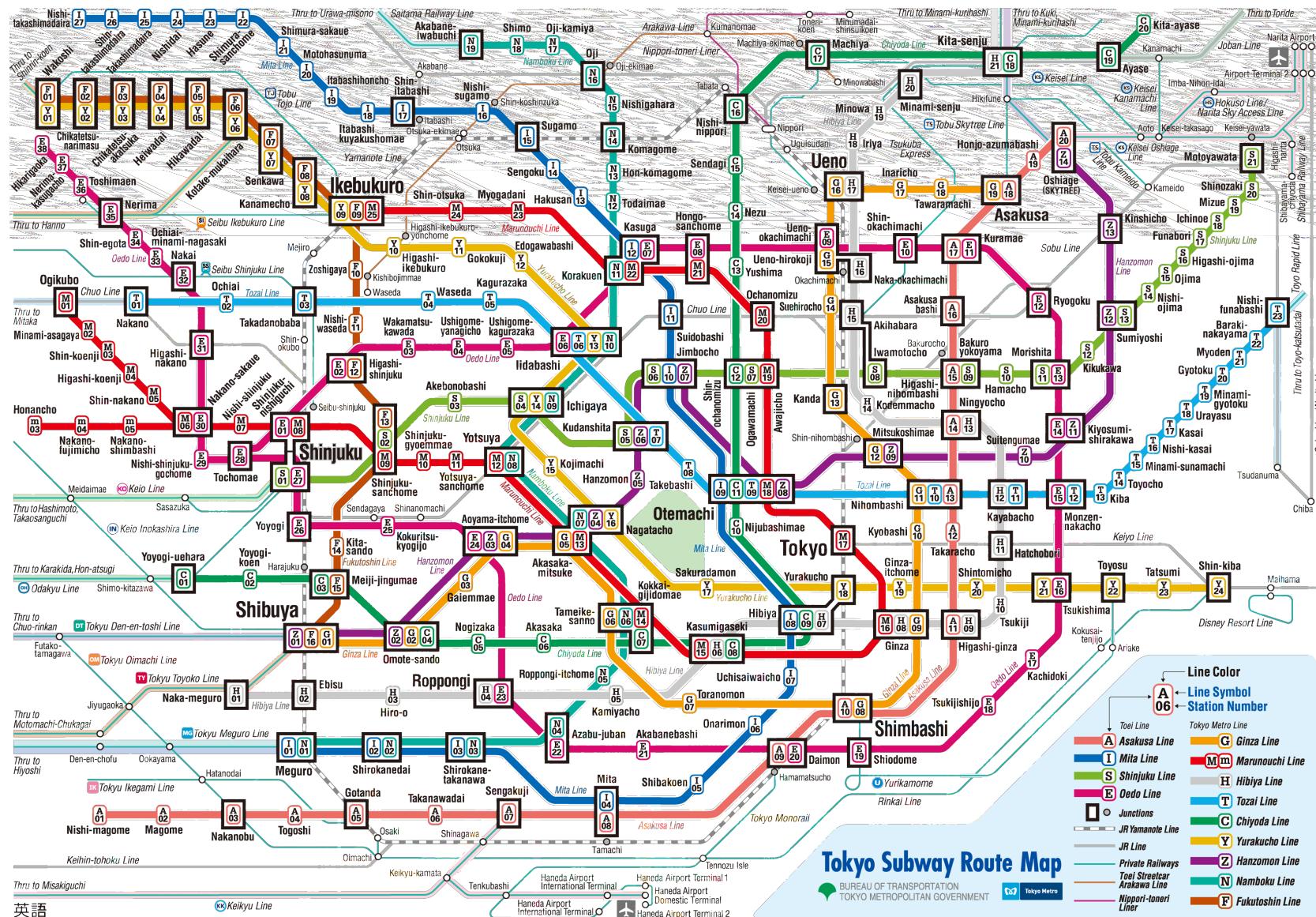
Kuva 19: Kuva: Wikimedia Commons, User:Jpatokal



Kuva 20: Kuva: Flickr, yaph, <http://exploringdata.github.io/vis/footballers-search-relations/>



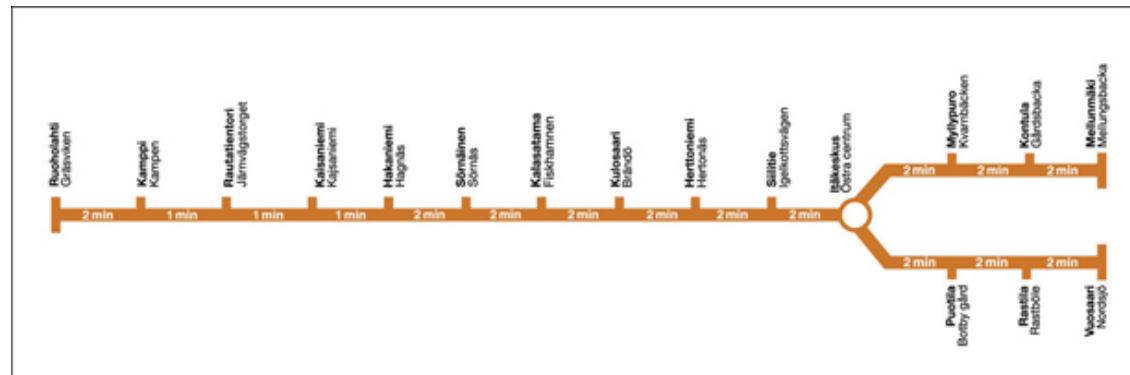
Kuva 21: Kuva: Flickr, Andy Lamb



Kuva 22: Kuva: Tokyo Metro

Matematiikassa graafi G ilmoitetaan usein parina $G = (V, E)$.

- V = solmujen joukko
- E = kaarien joukko
- tällöin samojen solmujen välillä saa yleensä olla vain yksi kaari molempien suuntiin
 - aina tämä ei kuitenkaan käytännön sovelluksessa riitä
 - esimerkiksi junia-aikatauluja esittävässä graafissa kaupunkien välillä on yleensä useampia vuoroja
 - täällästa graafia kutsutaan *monigraafiksi (multigraph)*



Kuva: HSL

- jos solmujen välillä sallitaan vain yksi kaari suuntaansa $\Rightarrow E \subseteq V^2$
 - suunnatulla graafilla $|E|$ voi vaihdella välillä $0, \dots, |V|^2$
 - laskettaessa graafialgoritmien suoritusaikoja oletamme tämän

Graafialgoritmin suorituskyky ilmoitetaan yleensä sekä $|V|$:n että $|E|$:n funktiona

- helpouden vuoksi jätämme itseisarvomerkit pois kertaluokkamerkintöjen sisällä ts. $O(VE) = O(|V| \cdot |E|)$

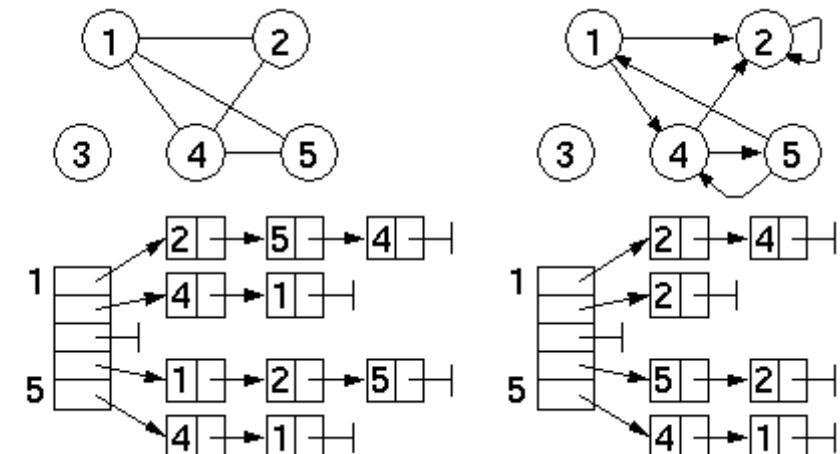
Graafin esittämiseen tietokoneessa on kaksi perusmenetelmää *kytkentälista* (*adjacency list*) ja *kytkentämatriisi* (*adjacency matrix*).

Näistä kahdesta kytkentälistaesitys on yleisempi, ja täällä kurssilla keskitytään siihen.

- Solmut on talletettu johonkin tietorakenteeseen (valittu tietorakenne riippuu siitä, mitä oheisdataa solmut sisältävät, miten niitä pitää pystyä hakemaan, lisäämään jne.)
- Yksinkertaisimillaan jokaisessa solmussa on tietorakenne, jossa on tallessa mihin solmuihin tästä solmusta on kaari.
 - Valittu tietorakenne riippuu siitä, paljonko kaaria arvellaan olevan, lisätäänkö/poistetaanko niitä jatkuvasti, täytyykö tiettyä kaarta pystyä hakemaan nopeasti jne.)
 - Tieto kohdesolmosta voidaan tallettaa osoittimena, solmun indeksinä (jos solmut indeksoitavassa tietorakenteessa) tms.
 - Solmujen järjestyksellä kytkentälistassa ei yleensä ole väliä

- Graafin kytkentälistojen yhteiskoko on
 - $|E|$, jos graafi on suunnattu, $2 \cdot |E|$, jos graafi on suuntaamaton

\Rightarrow kytkentälistaesityksen muistin kulutus on $O(\max(V, E)) = O(V + E)$
- Tiedon ”onko kaarta solmesta v solmuun u ” haku edellyttää yhden kytkentälistan läpi seläamista mikä vie hitaimmillaan aikaa $\Theta(V)$ (ellei solmesta lähteviä kaaria talleteta johonkin tietorakenteeseen, josta niistä pystytään nopeasti hakemaan kohdekaaren perusteella)
- Jos kaari on painotettu tai siihen liittyy oheisdataa, täytyy kaaresta tallettaa kohdesolmun lisäksi myös oheisdata (esim. struct, jossa kohdesolmu ja oheisdata)
- Joskus on tarpeen tallettaa myös tieto solmuun *tulevista* kaarista samaan tapaan (esim. solmujen ja kaarien poistamisen helpottamiseksi)



Kytkentämatriisiesityksen avulla edelliseen kysymykseen pystytään vastaamaan helposti.

- kytkentämatriisi on $|V| \times |V|$ -matriisi A , jonka alkio a_{ij} on
 - 0, jos solmesta i ei ole kaarta solmuun j
 - 1, jos solmesta i on kaari solmuun j
- edellisen esimerkin graafien kytkentämatriisit ovat

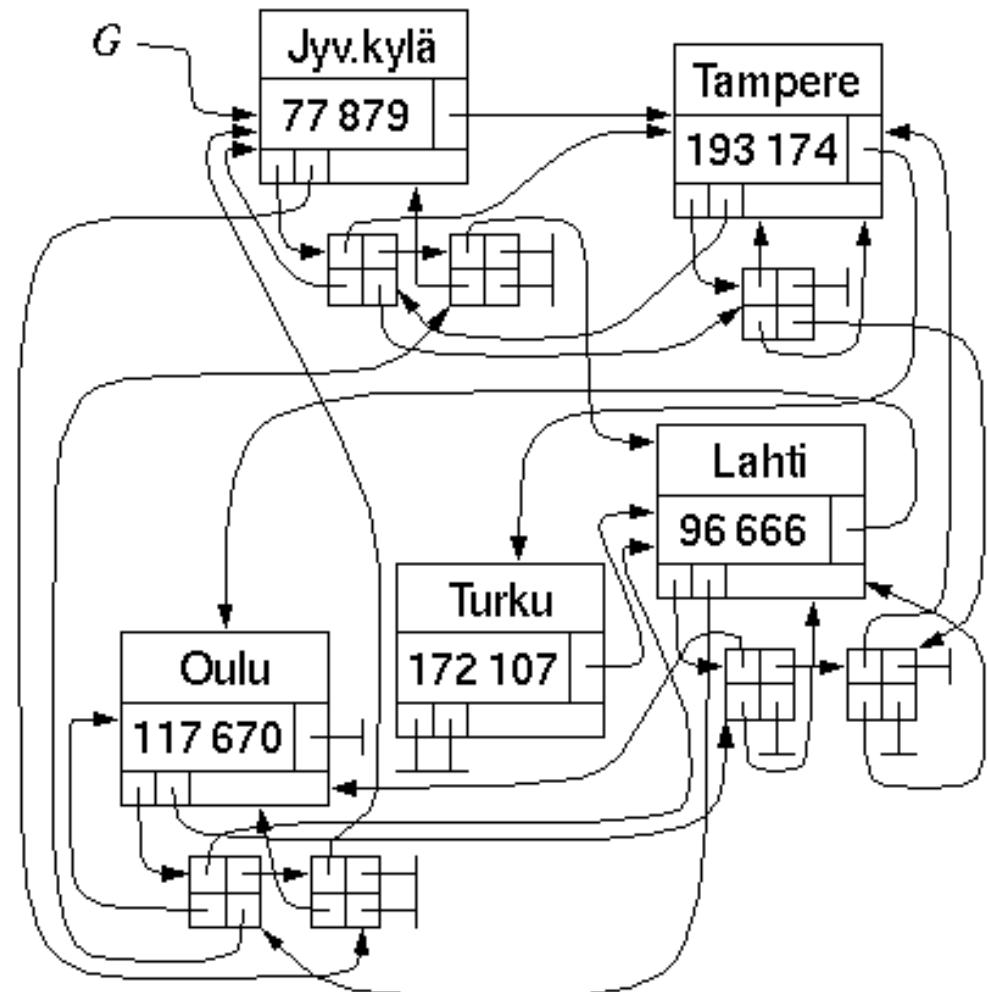
	1	2	3	4	5		1	2	3	4	5	
1	0	1	0	1	1		1	0	1	0	1	0
2	1	0	0	1	0		2	0	1	0	0	0
3	0	0	0	0	0		3	0	0	0	0	0
4	1	1	0	0	1		4	0	1	0	0	1
5	1	0	0	1	0		5	1	0	0	1	0

- muistin kulutus on aina $\Theta(V^2)$
 - jokainen alkio tarvitsee vain bitin muistia, joten useita alkioita voidaan tallettaa yhteen sanaan \Rightarrow vakiokerroin saadaan aika pieneksi
- kytkentämatriisiesitystä kannattaa käyttää lähinnä vain hyvin tiheiden graafien yhteydessä.

Tarkastellaan tarkemmin kytkentälistaesitystavan toteutusta:

- käytännön sovelluksissa solmuun kertyy usein monenlaista tehtävän tai algoritmin vaatimaa tietoa
 - nimi
 - bitti, joka kertoo, onko solmussa käyty
 - osoitin, joka kertoo, mistä solmusta tähän solmuun viimeksi tuliin
 - ...
- ⇒ solmusta kannattaa tehdä itsenäinen alkio, jossa on tarpeelliset kentät
- yleensä sama pätee myös kaariin

- pääperiaate:
 - talletta jokainen asia yhteen kertaan
 - ota käyttöön osoittimet, joilla voit helposti kulkea haluamiisi suuntiin



13.2 Yleistä graafialgoritmeista

Käsitteitä:

- askel = siirtyminen solmusta toiseen yhtä kaarta pitkin
 - suunnatussa graafissa askel on otettava kaaren suuntaan
- solmun v_2 etäisyys (*distance*) solmusta v_1 on lyhimmän v_1 :stä v_2 :een vievän polun pituus.
 - jokaisen solmun etäisyys itsestään on 0
 - merkitään $\delta(v_1, v_2)$
 - suunnatussa graafissa on mahdollista (ja tavallista), että $\delta(v_1, v_2) \neq \delta(v_2, v_1)$
 - jos v_1 :stä ei ole polkua v_2 :een, niin $\delta(v_1, v_2) = \infty$

Algoritmien ymmärtämisen helpottamiseksi annamme usein solmuille värit.

- valkoinen = solmua ei ole vielä löydetty
- harmaa = solmu on löydetty, mutta ei loppuun käsitelty
- musta = solmu on löydetty ja loppuun käsitelty
- solmun väri muuttuu järjestyksessä valkoinen → harmaa → musta
- värikoodaus on lähinnä ajattelun apuväline, eikä sitä välttämättä tarvitse toteuttaa täysin, yleensä riittää tietää, onko solmu löydetty vai ei.
 - usein tämäkin informaatio on pääteltävissä nopeasti muista kentistä

Monet graafialgoritmit perustuvat graafin tai sen osan läpikäyntiin tietyssä järjestyksessä.

- perusläpikäytijärjestyksiä on kaksi, **leveyteen ensin** -haku (*BFS*) ja **syvyyteen ensin** -haku (*DFS*).
- läpikäynnillä tarkoitetaan algoritmia, jossa
 - käydään kerran kerran graafin tai sen osan jokaisessa solmussa
 - kuljetaan kerran kerran graafin tai sen osan jokainen kaari

Hakualgoritmit käyttävät lähtökohtana joitain annettua graafin solmua, *lähtösolmua* (*source*) ja etsivät kaikki ne solmut, joihin pääsee lähtösolmusta nollalla tai useammalla askelleella.

13.3 Leveyteen ensin -haku (breadth-first)

Leveyteen ensin -hakua voi käyttää esimerkiksi

- kaikkien solmujen etäisyyden määrittämiseen lähtösolmusta
- (yhdellä) lyhimmän polun löytämiseen lähtösolmusta jokaiseen solmuun

Leveyteen ensin -haun nimi tulee siitä, että se tutkii tutkitun ja tuntemattoman graafin osan välistä rajapintaa koko ajan koko sen leveydeltä.

Solmujen kentät:

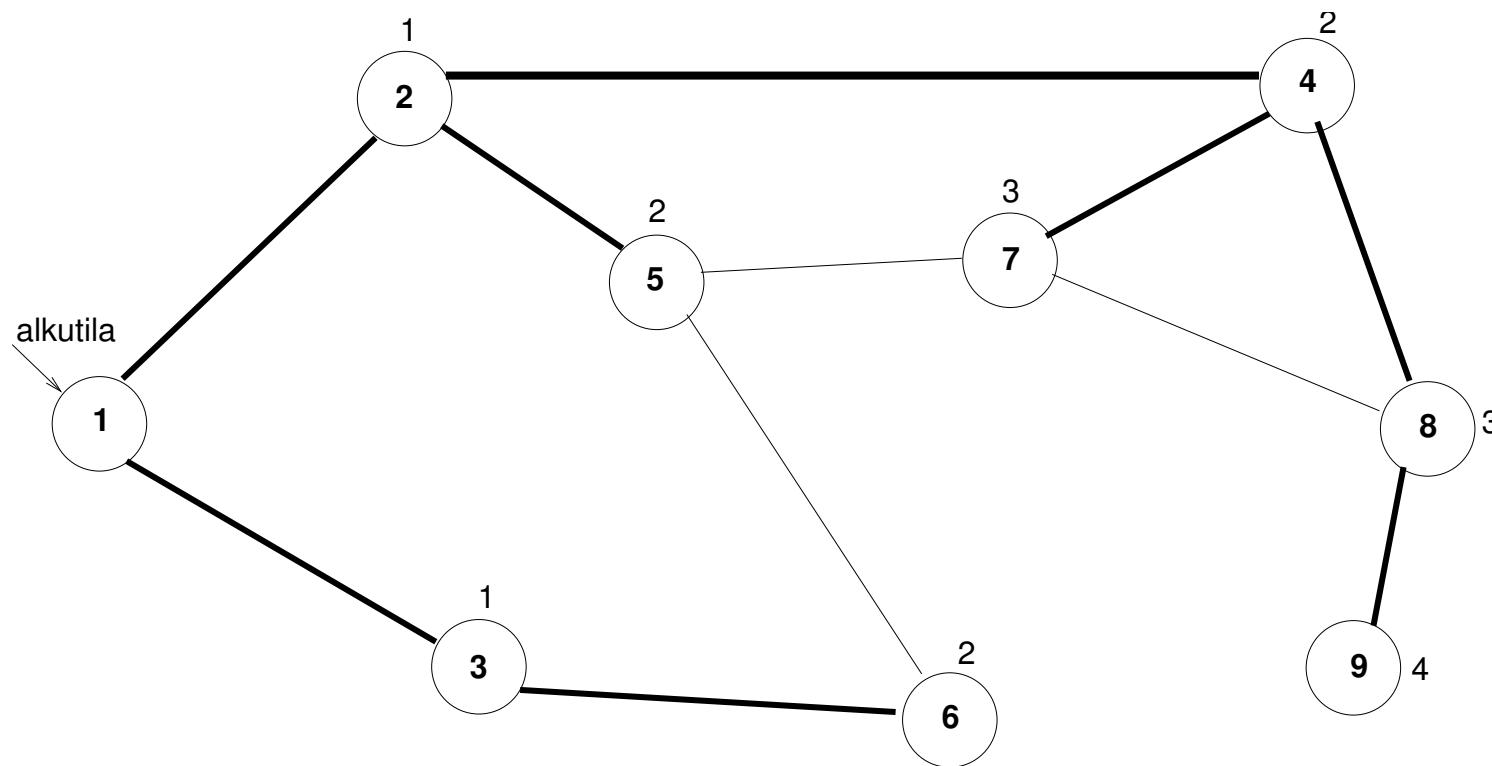
- $v \rightarrow d$ = jos solmu v on löydetty niin sen etäisyys s :stä, muutoin ∞
- $v \rightarrow \pi$ = osoitin solmuun, josta haku ensi kerran tuli v :hen, löytämättömille solmuille NIL
- $v \rightarrow colour$ = solmun v väri
- $v \rightarrow Adj$ = solmun v naapurisolmujen joukko

Algoritmin käyttämä tietorakenne Q on jono (noudattaa FIFO-jonokuria).

$\text{BFS}(s)$	(algoritmi saa parametrinaan aloitussolmun s)
1 \triangleright alussa kaikkien solmujen kentät ovat arvoiltaan $\text{colour} = \text{WHITE}$, $d = \infty$, $\pi = \text{NIL}$	(merkitään alkutila löydetyksi)
2 $s \rightarrow \text{colour} := \text{GRAY}$	(etäisyys alkutilasta alkutilaan on 0)
3 $s \rightarrow d := 0$	(työnnetään alkutila jonoon)
4 $\text{PUSH}(Q, s)$	(toistetaan niin kauan kun tiloja riittää)
5 while $Q \neq \emptyset$ do	(vedetään jonosta seuraava tila)
6 $u := \text{POP}(Q)$	(käydään u :n naapurit läpi)
7 for each $v \in u \rightarrow \text{Adj}$ do	(jos solmua ei ole vielä löydetty ...)
8 if $v \rightarrow \text{colour} = \text{WHITE}$ then	(... merkitään se löydetyksi)
9 $v \rightarrow \text{colour} := \text{GRAY}$	(kasvatetaan etäisyyttä yhdellä)
10 $v \rightarrow d := u \rightarrow d + 1$	(tilaan v tuliin tilan u kautta)
11 $v \rightarrow \pi := u$	(työnnetään tila jonoon odottamaan käsitellyä)
12 $\text{PUSH}(Q, v)$	(merkitään tila u käsitellyksi)
13 $u \rightarrow \text{colour} := \text{BLACK}$	

Kaikkia algoritmin käyttämiä solmun kenttiä ei välttämättä käytännön toteutuksessa tarvita, vaan osan arvoista voi päätellä toisistaan.

Alla olevassa kuvassa graafin solmut on numeroitu siinä järjestyksessä, jossa BFS löytää ne. Solmujen etäisyys alkutilasta on merkitty solmun viereen ja haun kulkureitti tummennettu.



Suoritusaika solmujen (V) ja kaarien (E) määrien avulla ilmaistuna:

- ennen algoritmin kutsumista solmut pitäää alustaa
 - järkevässä ratkaisussa tämä on tehtävissä ajassa $O(V)$
- rivillä 7 algoritmi selaa solmun lähtökaaret
 - onnistuu käyttämällämme kytkentälistaesityksellä solmun kaarien määrään nähdien lineaarisessa ajassa
- kukin jono-operaatio vie vakiomäärän aikaa
- while-silmukan kierrosten määrä
 - vain valkoisia solmuja laitetaan jonoon
 - samalla solmun väri muuttuu harmaaksi
 - ⇒ kukin solmu voi mennä jonoon korkeintaan kerran
- ⇒ while-silmukka pyörähtää siis korkeintaan $O(V)$ määrään kertoja

- for-silmukan kierrosten määrä
 - algoritmi kulkee jokaisen kaaren korkeintaan kerran molempien suuntiin
- ⇒ for-silmukka käydään läpi yhteenä korkeintaan $O(E)$ kertaa
- ⇒ koko algoritmin suoritusaika on siis $O(V + E)$

Algoritmin lopetettua π -osoittimet määrittelevät puun, joka sisältää löydetyt solmut, ja jonka juurena on lähtösolmu s .

- leveyteen ensin -puu (*breadth-first tree*)
- π -osoittimet määäräävät puun kaaret “takaperin”
 - osoittavat juurta kohti
 - $v \rightarrow \pi = v$:n edeltäjä (*predecessor*) eli isä (*parent*)
- kaikki lähtösolmusta saavutettavissa olevat solmut kuuluvat puuhun
- puun polut ovat mahdollisimman lyhyitä polkuja s :stä löydettyihin solmuihin

Lyhimmän polun tulostaminen

- kun BFS on asettanut π -osoittimet kohdalleen, lyhin polku lähtösolmesta s solmuun v voidaan tulostaa seuraavasti:

`PRINT-PATH(G, s, v)`

```

1   if  $v = s$  then                               (rekursion pohjatapaus)
2       print  $s$ 
3   else if  $v \rightarrow \pi = \text{NIL}$  then      (haku ei ole saavuttanut solmua  $v$  lainkaan)
4       print "ei polkua"
5   else
6       PRINT-PATH( $G, s, v \rightarrow \pi$ )          (rekursiokutsu ...)
7       print  $v$                                 (...jonka jälkeen suoritetaan tulostus)

```

- ei-rekursiivisen version voi tehdä esim.
 - kokoamalla solmujen numerot taulukkoon kulkemalla π -osoittimia pitkin, ja tulostamalla taulukon sisällön takaperin
 - kulkemalla polku kahdesti, ja käänämällä π -osoittimet takaperin kummallakin kertaa (jälkimmäinen käänös ei tarpeen, jos π -osoittimet saa turmella)

13.4 Syvyyteen ensin -haku (depth-first)

Syvyyteen ensin -haku on toinen perusläpikäytijärjestysistä.

Siinä missä leveyteen ensin -haku tutkii koko hakurintamaa sen koko leveydeltä, syvyyteen ensin -haku menee yhtä polkua eteen pään niin kauan kuin se on mahdollista.

- polkuun hyväksytään vain solmuja, joita ei ole aiemmin nähty
- kun algoritmi ei enää pääse eteenpäin, se peruuttaa juuri sen verran kuin on tarpeen uuden etenemisreitin löytämiseksi, ja lähtee sitä pitkin
- algoritmi lopettaa, kun se peruuttaa viimeisen kerran takaisin lähtösolmuun, eikä löydä enää sieltäkään tutkimattomia kaaria

Algoritmi muistuttaa huomattavasti leveyteen ensin -haun pseudokoodia.

Merkittäviä eroja on oikeastaan vain muutama:

- jonon sijasta käsitteilyvuoroaan odottavat tilat talletetaan pinoon
- algoritmi ei löydä lyhimpiä polkuja, vaan ainoastaan jonkin polun
 - tästä syystä esimerkkipseudokoodia on yksinkertaistettu jättämällä π -kentät pois

Algoritmin käyttämä tietorakenne S on pino (noudattaa LIFO-jonokuria).

DFS(s)	(algoritmi saa parametrinaan aloitussolmun s)
1 \triangleright alussa kaikkien (käsittelemättömiin) solmujen värikenttä $colour = \text{WHITE}$	
2 PUSH(S, s)	(työnnetään alkutila pinoon)
3 while $S \neq \emptyset$ do	(jatketaan niin kauan kun pinossa on tavaraa)
4 $u := \text{POP}(S)$	(vedetään pinosta viimeisin sinne lisätty tila)
5 if $u \rightarrow colour = \text{WHITE}$ then	(jos solmua ei ole vielä käsitelty ...)
6 $u \rightarrow colour := \text{GRAY}$	(merkitään tila käsitellyssä olevaksi)
7 PUSH(S, u)	(työnnetään taas pinoon (mustaksi värijäys))
8 for each $v \in u \rightarrow Adj$ do	(käydään $u:n$ naapurit läpi)
9 if $v \rightarrow colour = \text{WHITE}$ then	(jos solmua ei ole vielä käsitelty ...)
10 PUSH(S, v)	(... työnnetään se pinoon odottamaan käsitelyä)
11 else if $v \rightarrow colour = \text{GRAY}$ then	(harmaa solmu! Sykli löytynyt! ...)
12 ?????	(käsittele sykli, jos se kiinnostaa)
13 else	
14 $u \rightarrow colour := \text{BLACK}$	(kaikki lapset käsitelty, solmu on valmis)

Jos halutaan tutkia koko graafi, voidaan kutsua syvyyteen ensin -hakua kertaalleen kaikista vielä tutkimattomista solmuista.

- tällöin solmuja ei väritetä valkoisiksi kutsukertojen välillä

Rivin 5 perään voitaisi lisätä operaatio, joka kaikille graafin alkioille halutaan tehdä. Voidaan esimerkiksi

- tutkia onko tila maalitila, ja lopettaa jos on
- ottaa talteen solmuun liittyvää oheisdataa
- muokata solmuun liittyvää oheisdataa

Suoritusaika voidaan laskea samoin kuin leveyteen ensin -haun yhteydessä:

- ennen algoritmin kutsumista solmut pitää alustaa
 - järkevässä ratkaisussa tämä on tehtävissä ajassa $O(V)$
- rivillä 6 algoritmi selaa solmun lähtökaaret
 - onnistuu käyttämällämme kytkentälistaesityksellä solmun kaarien määrään nähdien lineaarisessa ajassa
- kukin pino-operaatio vie vakiomäärän aikaa
- while-silmukan kierrosten määrä
 - vain valkoisia solmuja laitetaan pinoon
 - samalla solmun väri muuttuu harmaaksi
 - ⇒ kukin solmu voi mennä pinoon korkeintaan kerran
- ⇒ while-silmukka pyörähtää siis korkeintaan $O(V)$ määrän kertoja

- for-silmukan kierrosten määrä
 - algoritmi kulkee jokaisen kaaren korkeintaan kerran molempien suuntiin
- ⇒ for-silmukka käydään läpi yhteenä korkeintaan $O(E)$ kertaa
- ⇒ koko algoritmin suoritusaika on siis $O(V + E)$

DFS on myös mahdollista toteuttaa rekursiivisesti, jolloin algoritmin pinona toimii funktioiden kutsupino.

- Rekursiivinen versio on itse asiassa jonkin verran yksinkertaisempi kuin iteratiivinen versio!
- (Keksitkö, miksi?)

Huom! Ennen algoritmin kutsumista kaikki solmut tulee alustaa valkoisiksi!

DFS(u)

```
1  $u \rightarrow colour := \text{GRAY}$ 
2 for each  $v \in u \rightarrow Adj$  do
3   if  $v \rightarrow colour = \text{WHITE}$  then
4     DFS( $v$ )
5   else if  $v \rightarrow colour = \text{GRAY}$  then
6     ▷ silmukka on löytynyt
7    $u \rightarrow colour := \text{BLACK}$ 
```

(merkitään tila löydetyksi)

(käydään kaikki $u:n$ naapurit läpi)

(jos ei olla vielä käyty $v:ssä\ldots$)

(...jatketaan etsintää rekursiivisesti tilasta v)

(jos on jo käyty, muttei loppuun käsitelty ...)

(...silmukka on löytynyt)

(merkitään tila käsitellyksi)

Suoritusaika:

- rekursiivinen kutsu tehdään ainoastaan valkoisille solmuille
- funktion alussa solmu väritetään harmaaksi
⇒ DFS:ää kutsutaan korkeintaan $O(V)$ kertaa
- kuten aiemmassakin versiossa for-silmukka kiertää korkeintaan kaksi kierrosta kutakin graafin kaarta kohden koko algoritmin suorituksen aikana
⇒ siis for-silmukan kierroksia tulee korkeintaan $O(E)$ kappaletta
- muut operaatioista ovat vakioaikaisia
⇒ koko algoritmin suoritusaika on edelleen $O(V + E)$

Leveyteen ensin -haku vai syvyyteen ensin -haku:

- lyhimmän polun etsimiseen täytyy käyttää leveyteen ensin -hakua
- jos graafin esittämä tilavaruus on hyvin suuri, käyttää leveyteen ensin -haku yleensä huomattavasti enemmän muistia
 - syvyyteen ensin -haun pinon koko pysyy yleensä pienempänä kuin leveyteen ensin -haun jonon koko
 - useissa sovelluksissa esimerkiksi tekoälyn alalla jonon koko estää leveyteen ensin -haun käytön
- mikäli graafin koko on ääretön, ongelmaksi nousee se, ettei syvyyteen ensin -haku välttämättä löydää ikinä maalitilaa, eikä edes lopeta ennen kuin muisti loppuu
 - näin tapahtuu, jos algoritmi lähtee tutkimaan hedelmätöntä äärettömän pitkää haaraa
 - tätä ongelmaa ei kuitenkaan esiinny äärellisten graafien yhteydessä

- syvyyteen ensin -haun avulla voi ratkaista joitakin monimutkaisempia ongelmia, kuten graafin silmukoiden etsintää
 - harmaat solmut muodostavat lähtösolmesta nykyiseen solmuun vievän polun
 - mustista solmuista pääsee vain mustiin ja harmaisiin solmuihin
⇒ jos nykyisestä solmesta pääsee harmaaseen solmuun, niin graafissa on silmukka