Elementære Grafalgoritmer

Algorithms and Datastructures, F25, Lecture 8

Andreas Holck Høeg-Petersen

Department of Computer Science
Aalborg University

April 11, 2025

Opdateringer

- Tak for mange fine afleveringer! De vil blive behandlet. . . i løbet af og efter påsken

Opdateringer

- Tak for mange fine afleveringer! De vil blive behandlet. . . i løbet af og efter påsken
- I dag blev mine nye slides kun 90% færdige - så der er lige et blast from the past i slutningen af forelæsningen

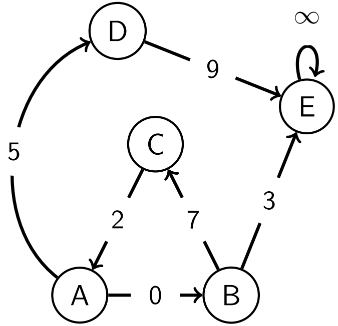
Outline

Outline

Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

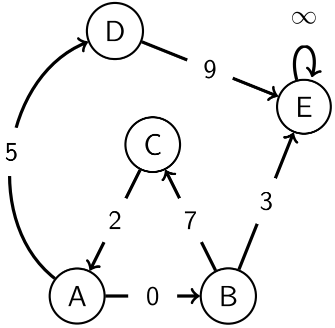


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)

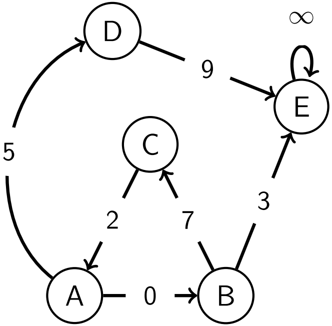


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)

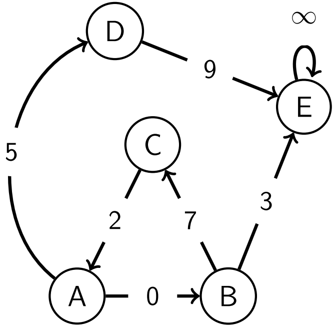


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter

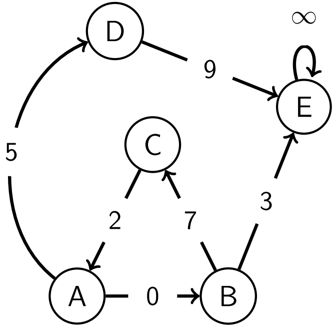


Grafer

Hvad og hvorfor?

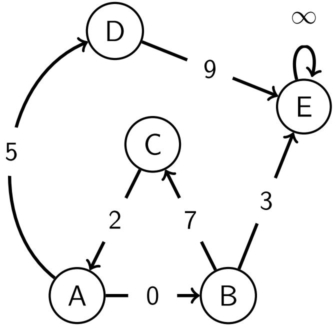
En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med



En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med
  - Dependency trees

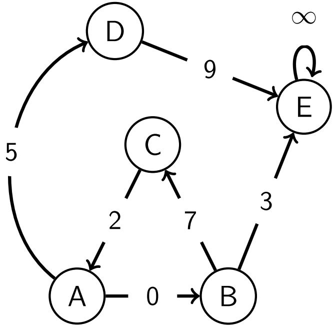


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med
  - Dependency trees
  - Tidsserier

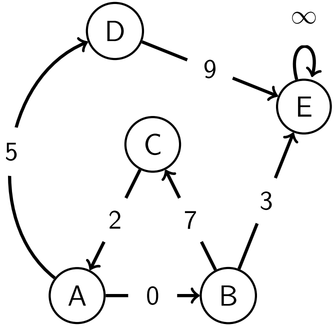


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med
  - Dependency trees
  - Tidsserier
  - Stifinding i netværk eller kort

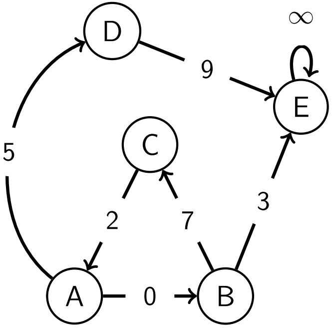


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med
  - Dependency trees
  - Tidsserier
  - Stifinding i netværk eller kort
  - Internettet er en stor graf — det samme er sociale netværk

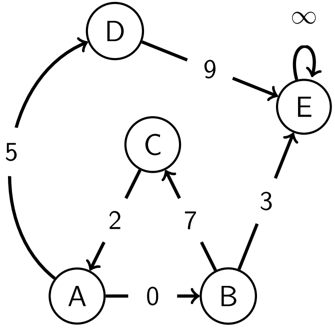


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med
  - Dependency trees
  - Tidsserier
  - Stifinding i netværk eller kort
  - Internettet er en stor graf — det samme er sociale netværk
  - Kan modellerer sandsynlighedsdistributioner (Bayesian networks)

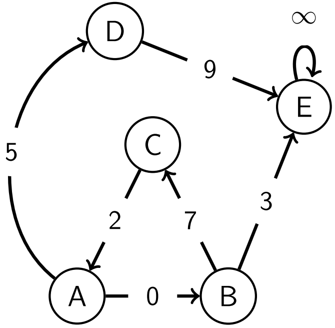


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med
    - Dependency trees
    - Tidsserier
    - Stifinding i netværk eller kort
    - Internettet er en stor graf — det samme er sociale netværk
    - Kan modellerer sandsynlighedsdistributioner (Bayesian networks)
    - Neurale netværk er basically grafer og trænes ved at konstruere en computational graph for gradienterne (backpropagation)

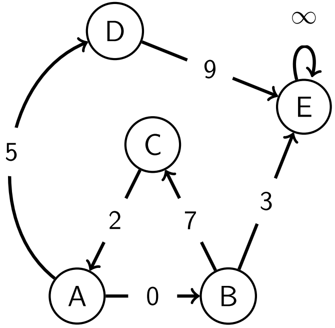


Grafer

Hvad og hvorfor?

En graf er en måde at repræsentere objekter og deres interne relationer.

- Ligesom træer, har grafer knuder (verticies/nodes), der er forbundet af kanter (edges)
- Modsat træer, er der ikke nødvendigvis nogen naturlig rækkefølge at læse grafen i (på?)
- Vi specificerer en graf $G = (V, E)$, hvor V er mængden af knuder og $E \subset V \times V$ er mængden af kanter
- Grafer er en *ekstrem* fleksibel datastruktur til at modellere problemer med
  - Dependency trees
  - Tidsserier
  - Stifinding i netværk eller kort
  - Internettet er en stor graf — det samme er sociale netværk
  - Kan modellerer sandsynlighedsdistributioner (Bayesian networks)
  - Neurale netværk er basically grafer og trænes ved at konstruere en computational graph for gradienterne (backpropagation)
  - ... og meget, meget mere



Grafer

Orienterede og ikke-orienterede

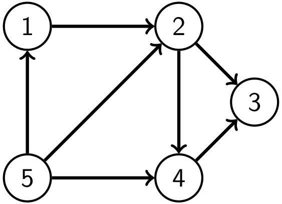
Grafer kommer overordnet set i to varianter:

Grafer

Orienterede og ikke-orienterede

Grafer kommer overordnet set i to varianter:

- I orienterede (directed) grafer har hver kant en retning
- En kant $(u, v) \in E$ er en kant, der går fra knude u til knude v



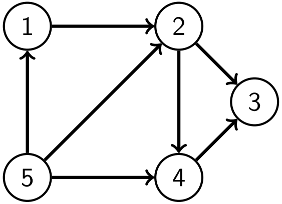
$$V = \{5, 1, 2, 4, 3\}$$
$$E = \{(5,1), (5,2), (5,4), (1,2), (2,3), (2,4), (3,4)\}$$

Grafer

Orienterede og ikke-orienterede

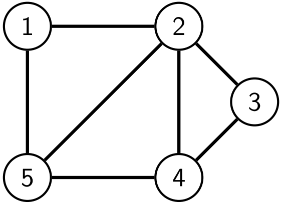
Grafer kommer overordnet set i to varianter:

- I orienterede (directed) grafer har hver kant en retning
- En kant $(u, v) \in E$ er en kant, der går fra knude u til knude v



$$V = \{5, 1, 2, 4, 3\}$$
$$E = \{(5,1), (5,2), (5,4), (1,2), (2,3), (2,4), (3,4)\}$$

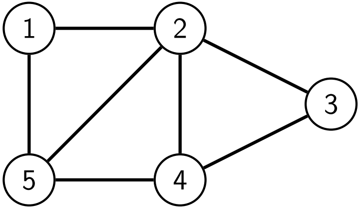
- I ikke-orienterede (undirected) grafer har en kant $u, v \in E$ ikke en retning, men forbinder u og v i begge retninger
- Ikke-orienterede grafer er en special case af orienterede grafer



$$V = \{5, 1, 2, 4, 3\}$$
$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer



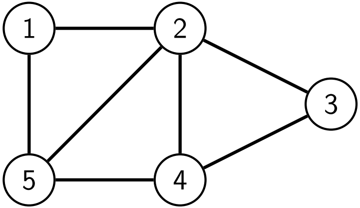
$$V = \{5, 1, 2, 4, 3\}$$

$$E = \{\{5, 1\}, \{5, 2\}, \{5, 4\}, \{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$



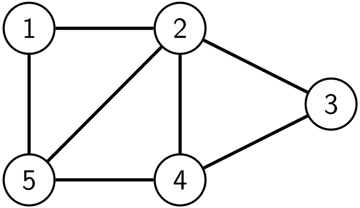
$$V = \{5, 1, 2, 4, 3\}$$

$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$
- Hvor mange kanter ($|E|$) kan der maksimalt være i en graf med $|V|$ knuder?



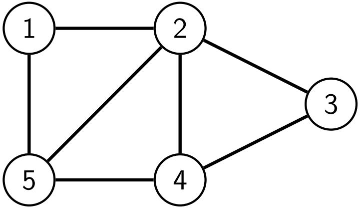
$$V = \{5, 1, 2, 4, 3\}$$

$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$
- Hvor mange kanter ($|E|$) kan der maksimalt være i en graf med $|V|$ knuder?
  - For en ikke-orienteret graf, $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$

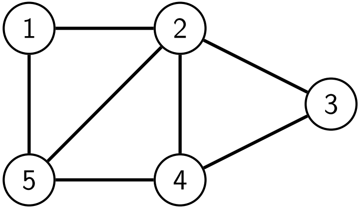


$$V = \{5, 1, 2, 4, 3\}$$
$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$
- Hvor mange kanter ($|E|$) kan der maksimalt være i en graf med $|V|$ knuder?
  - For en ikke-orienteret graf, $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$
  - For en orienteret graf, $|E| = |V|(|V|-1) = O(|V|^2)$



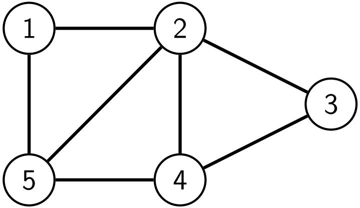
$$V = \{5, 1, 2, 4, 3\}$$

$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$
- Hvor mange kanter ($|E|$) kan der maksimalt være i en graf med $|V|$ knuder?
  - For en ikke-orienteret graf, $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$
  - For en orienteret graf, $|E| = |V|(|V|-1) = O(|V|^2)$
- Hvis $|E|$ er meget mindre end $|V|^2$ siger vi, at grafen er sparse



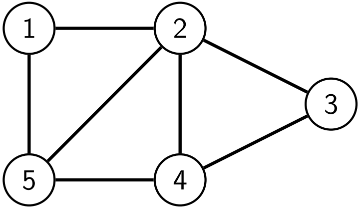
$$V = \{5, 1, 2, 4, 3\}$$

$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$
- Hvor mange kanter ($|E|$) kan der maksimalt være i en graf med $|V|$ knuder?
  - For en ikke-orienteret graf, $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$
  - For en orienteret graf, $|E| = |V|(|V|-1) = O(|V|^2)$
- Hvis $|E|$ er meget mindre end $|V|^2$ siger vi, at grafen er sparse
- Hvis $|E|$ er tæt på $|V|^2$ siger vi, at grafen er dense



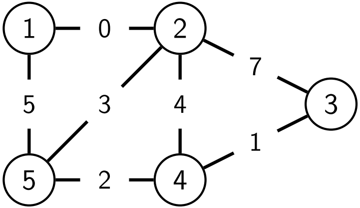
$$V = \{5, 1, 2, 4, 3\}$$

$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$
- Hvor mange kanter ($|E|$) kan der maksimalt være i en graf med $|V|$ knuder?
  - For en ikke-orienteret graf, $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$
  - For en orienteret graf, $|E| = |V|(|V|-1) = O(|V|^2)$
- Hvis $|E|$ er meget mindre end $|V|^2$ siger vi, at grafen er sparse
- Hvis $|E|$ er tæt på $|V|^2$ siger vi, at grafen er dense
- Vi kan også have at gøre med vægtede grafer

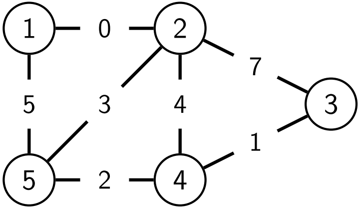


$$V = \{5, 1, 2, 4, 3\}$$
$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Grafer

Flere detaljer

- Når vi analyserer grafer, angiver vi typisk kompleksiteten som en funktion af både $|V|$ og $|E|$
- Hvor mange kanter ($|E|$) kan der maksimalt være i en graf med $|V|$ knuder?
  - For en ikke-orienteret graf, $|E| = \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$
  - For en orienteret graf, $|E| = |V|(|V|-1) = O(|V|^2)$
- Hvis $|E|$ er meget mindre end $|V|^2$ siger vi, at grafen er sparse
- Hvis $|E|$ er tæt på $|V|^2$ siger vi, at grafen er dense
- Vi kan også have at gøre med vægtede grafer
  - Her antager vi en weight function $w : E \to \mathbb{R}$, hvor $w(u, v)$ giver vægten af kanten fra u til v



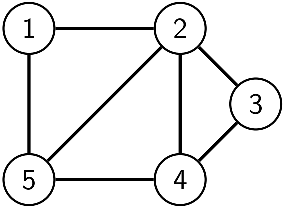
$$V = \{5, 1, 2, 4, 3\}$$

$$E = \{\{5,1\}, \{5,2\}, \{5,4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}\}$$

Repræsentation af grafer

Adjacency matrix

Vi skal se på to måder at repræsentere grafer på:



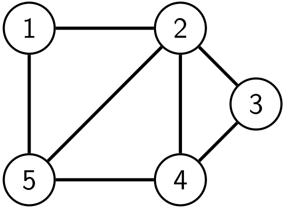
$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

Repræsentation af grafer

Adjacency matrix

Vi skal se på to måder at repræsentere grafer på:

- For en graf $G = (V, E)$ er en adjacency matrix en matrix (duh!) med størrelse $|V|^2$



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

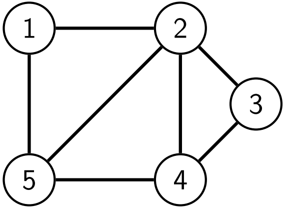
Repræsentation af grafer

Adjacency matrix

Vi skal se på to måder at repræsentere grafer på:

- For en graf $G = (V, E)$ er en adjacency matrix en matrix (duh!) med størrelse $|V|^2$
- Hvert element i matricen $A = (a_{ij})$ defineres således at

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$



$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

Repræsentation af grafer

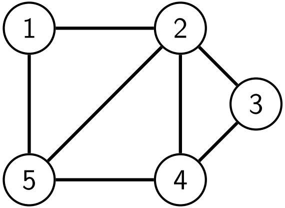
Adjacency matrix

Vi skal se på to måder at repræsentere grafer på:

- For en graf $G = (V, E)$ er en adjacency matrix en matrix (duh!) med størrelse $|V|^2$
- Hvert element i matricen $A = (a_{ij})$ defineres således at

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Nemt når vi skal repræsentere vægtede grafer — så lader vi bare $a_{ij} = w(i,j)$ hvis $(i,j) \in E$



$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \left|\begin{array}{ccccc} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{array}\right| \end{array}$$

Repræsentation af grafer

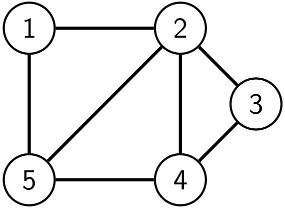
Adjacency matrix

Vi skal se på to måder at repræsentere grafer på:

- For en graf $G = (V, E)$ er en adjacency matrix en matrix (duh!) med størrelse $|V|^2$
- Hvert element i matricen $A = (a_{ij})$ defineres således at

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Nemt når vi skal repræsentere vægtede grafer — så lader vi bare $a_{ij} = w(i,j)$ hvis $(i,j) \in E$
- Hvor meget plads bruger dette?



$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

Repræsentation af grafer

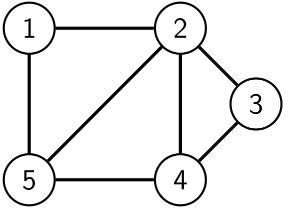
Adjacency matrix

Vi skal se på to måder at repræsentere grafer på:

- For en graf $G = (V, E)$ er en adjacency matrix en matrix (duh!) med størrelse $|V|^2$
- Hvert element i matricen $A = (a_{ij})$ defineres således at

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Nemt når vi skal repræsentere vægtede grafer — så lader vi bare $a_{ij} = w(i,j)$ hvis $(i,j) \in E$
- Hvor meget plads bruger dette? $O(|V|^2)$
- Derfor mest oplagt hvis grafen er dense

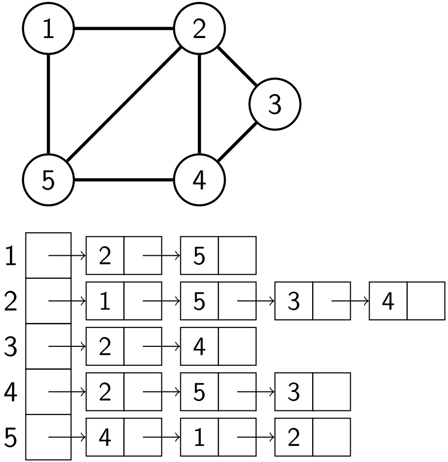


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Repræsentation af grafter

Adjacency lists

Den anden måde er ved brug af adjacency lists:

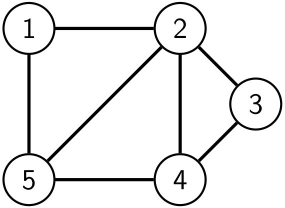


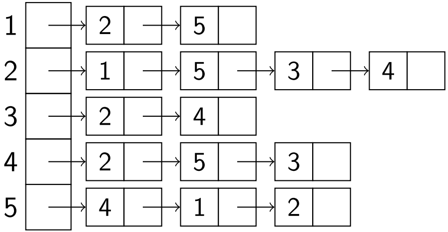
Repræsentation af grafter

Adjacency lists

Den anden måde er ved brug af adjacency lists:

- En adjacency lists for en graf $G = (V, E)$ består af et array (eller en hash-table) $Adj[1 : |V|]$



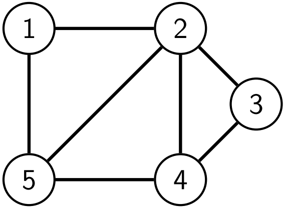


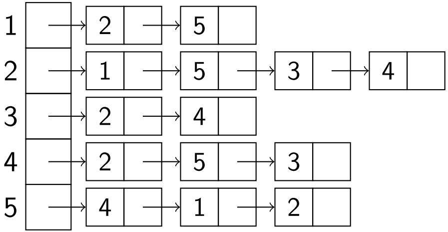
Repræsentation af grafter

Adjacency lists

Den anden måde er ved brug af adjacency lists:

- En adjacency lists for en graf $G = (V, E)$ består af et array (eller en hash-table) $Adj[1 : |V|]$
- For hver knude $u \in V$ har vi en liste $Adj[u]$, som indeholder alle knuder v hvor $\{u, v\} \in E$



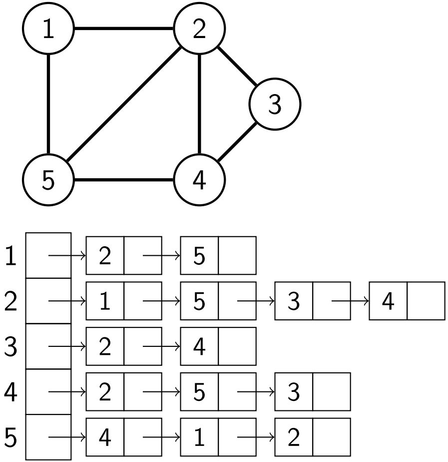


Repræsentation af grafter

Adjacency lists

Den anden måde er ved brug af adjacency lists:

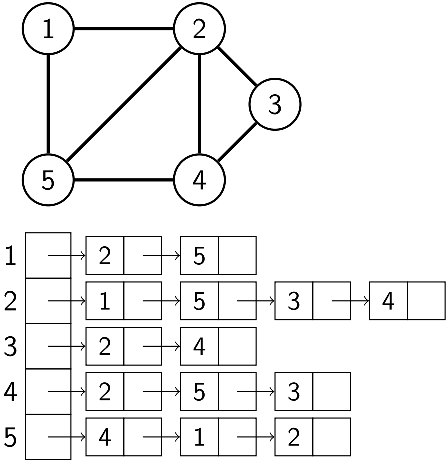
- En adjacency lists for en graf $G = (V, E)$ består af et array (eller en hash-table) $Adj[1 : |V|]$
- For hver knude $u \in V$ har vi en liste $Adj[u]$, som indeholder alle knuder v hvor $\{u, v\} \in E$
- Hvor meget plads bruger dette?



Repræsentation af grafter
Adjacency lists

Den anden måde er ved brug af adjacency lists:

- En adjacency lists for en graf $G = (V, E)$ består af et array (eller en hash-table) $Adj[1 : |V|]$
- For hver knude $u \in V$ har vi en liste $Adj[u]$, som indeholder alle knuder v hvor $\{u, v\} \in E$
- Hvor meget plads bruger dette?
  - $O(|V| + |E|)$

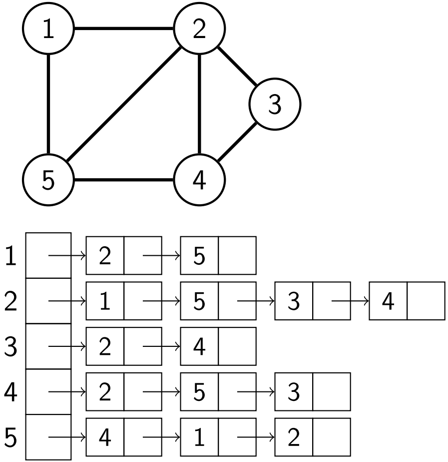


Repræsentation af grafter

Adjacency lists

Den anden måde er ved brug af adjacency lists:

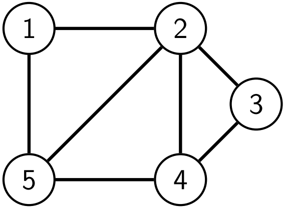
- En adjacency lists for en graf $G = (V, E)$ består af et array (eller en hash-table) $Adj[1 : |V|]$
- For hver knude $u \in V$ har vi en liste $Adj[u]$, som indeholder alle knuder v hvor $\{u, v\} \in E$
- Hvor meget plads bruger dette?
  - $O(|V| + |E|)$
- Bedre eller værre end adjacency matrix?

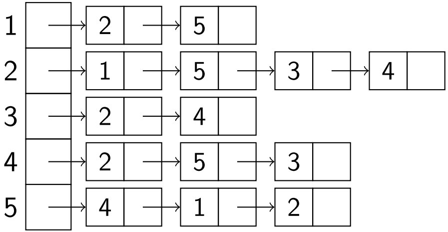


Repræsentation af grafter
Adjacency lists

Den anden måde er ved brug af adjacency lists:

- En adjacency lists for en graf $G = (V, E)$ består af et array (eller en hash-table) $Adj[1 : |V|]$
- For hver knude $u \in V$ har vi en liste $Adj[u]$, som indeholder alle knuder v hvor $\{u, v\} \in E$
- Hvor meget plads bruger dette?
  - $O(|V| + |E|)$
- Bedre eller værre end adjacency matrix?
  - Meget bedre hvis grafen er sparse





Outline

Søgning i grafer

Breadth-first search

Vi ser nu på to fundamentale måder at søge i en graf på.

[1] Gælder kun for ikke-vægtede grafer

Søgning i grafer

Breadth-first search

Vi ser nu på to fundamentale måder at søge i en graf på.

- Breadth-first search (BFS) søger, som navnet indikerer, i 'bredden' først

[1] Gælder kun for ikke-vægtede grafer

Søgning i grafer

Breadth-first search

Vi ser nu på to fundamentale måder at søge i en graf på.

- Breadth-first search (BFS) søger, som navnet indikerer, i 'bredden' først
- Givet en graf $G = (V, E)$ og en start-knude $s \in V$ finder BFS (afstanden på) den korteste sti fra s til alle andre knuder i G (der kan nåes fra s)[1]

[1] Gælder kun for ikke-vægtede grafer

Søgning i grafer

Breadth-first search

Vi ser nu på to fundamentale måder at søge i en graf på.

- Breadth-first search (BFS) søger, som navnet indikerer, i 'bredden' først
- Givet en graf $G = (V, E)$ og en start-knude $s \in V$ finder BFS (afstanden på) den korteste sti fra s til alle andre knuder i G (der kan nåes fra s)[1]
- Undervejs besøges alle knuder i V, der kan nåes fra s, og på hver knude v, som vi kommer til fra u, noterer vi dens predecessor $v.\pi = u$

[1] Gælder kun for ikke-vægtede grafer

Vi ser nu på to fundamentale måder at søge i en graf på.

- Breadth-first search (BFS) søger, som navnet indikerer, i 'bredden' først
- Givet en graf $G = (V, E)$ og en start-knude $s \in V$ finder BFS (afstanden på) den korteste sti fra s til alle andre knuder i G (der kan nåes fra s)[1]
- Undervejs besøges alle knuder i V, der kan nåes fra s, og på hver knude v, som vi kommer til fra u, noterer vi dens predecessor $v.\pi = u$
- Som artefakt får vi en predecessor subgraph af G i form af $G_\pi = (V_\pi, E_\pi)$

[1] Gælder kun for ikke-vægtede grafer

Søgning i grafer

Breadth-first search

Vi ser nu på to fundamentale måder at søge i en graf på.

- Breadth-first search (BFS) søger, som navnet indikerer, i 'bredden' først
- Givet en graf $G = (V, E)$ og en start-knude $s \in V$ finder BFS (afstanden på) den korteste sti fra s til alle andre knuder i G (der kan nåes fra s)[1]
- Undervejs besøges alle knuder i V, der kan nåes fra s, og på hver knude v, som vi kommer til fra u, noterer vi dens predecessor $v.\pi = u$
- Som artefakt får vi en predecessor subgraph af G i form af $G_\pi = (V_\pi, E_\pi)$
    - Hvor

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$
$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

[1] Gælder kun for ikke-vægtede grafer

Søgning i grafer

Breadth-first search

Vi ser nu på to fundamentale måder at søge i en graf på.

- Breadth-first search (BFS) søger, som navnet indikerer, i 'bredden' først
- Givet en graf $G = (V, E)$ og en start-knude $s \in V$ finder BFS (afstanden på) den korteste sti fra s til alle andre knuder i G (der kan nåes fra s)[1]
- Undervejs besøges alle knuder i V, der kan nåes fra s, og på hver knude v, som vi kommer til fra u, noterer vi dens predecessor $v.\pi = u$
- Som artefakt får vi en predecessor subgraph af G i form af $G_\pi = (V_\pi, E_\pi)$
  - Hvor

$$V_\pi = \{v \in V : v.\pi \neq \mathrm{NIL}\} \cup \{s\}$$
$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

  - Denne subgraf kalder vi også et breadth-first tree T med rod i s og hvor alle stier fra s til enhver anden knude $v \in T$ svarer til den korteste sti fra s til v i G

[1] Gælder kun for ikke-vægtede grafer

Søgning i grafer

Breadth-first search

Vi ser nu på to fundamentale måder at søge i en graf på.

- Breadth-first search (BFS) søger, som navnet indikerer, i 'bredden' først
- Givet en graf $G = (V, E)$ og en start-knude $s \in V$ finder BFS (afstanden på) den korteste sti fra s til alle andre knuder i G (der kan nåes fra s)[1]
- Undervejs besøges alle knuder i V, der kan nåes fra s, og på hver knude v, som vi kommer til fra u, noterer vi dens predecessor $v.\pi = u$
- Som artefakt får vi en predecessor subgraph af G i form af $G_\pi = (V_\pi, E_\pi)$
    - Hvor

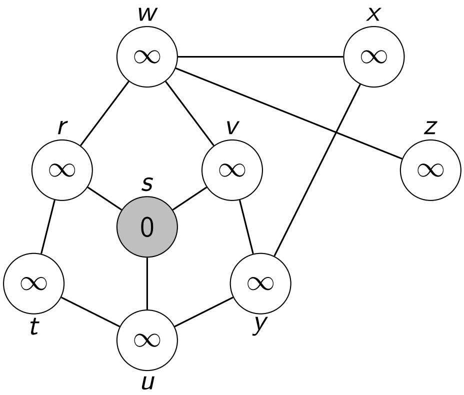
$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$
$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

    - Denne subgraf kalder vi også et breadth-first tree T med rod i s og hvor alle stier fra s til enhver anden knude $v \in T$ svarer til den korteste sti fra s til v i G
- Fungerer både på orienterede og ikke-orienterede grafer

[1] Gælder kun for ikke-vægtede grafer

Breadth-first search

Eksempel

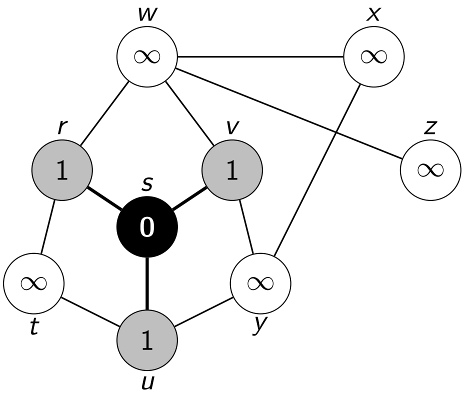


$$Q = \langle s \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s

Breadth-first search

Eksempel

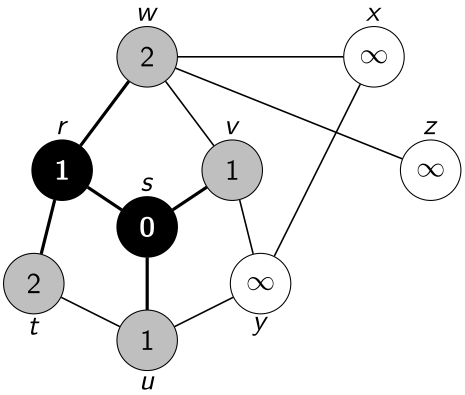


$Q = \langle r, u, v \rangle$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var

Breadth-first search

Eksempel

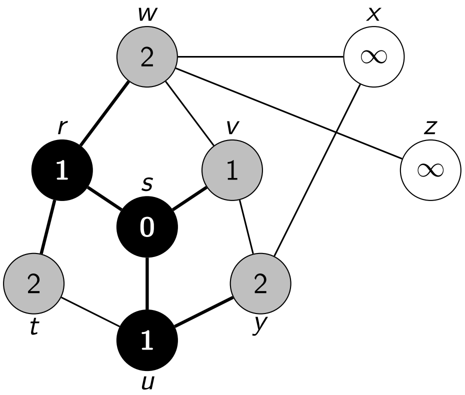


$$Q = \langle u, v, t, w \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s

Breadth-first search

Eksempel

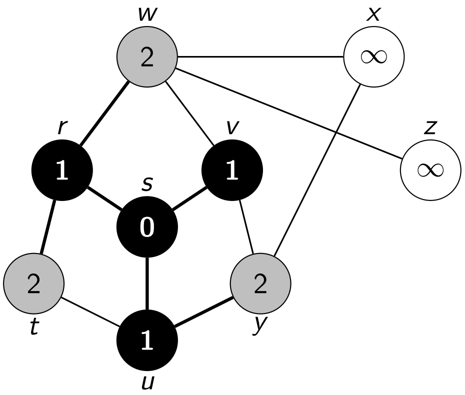


$$Q = \langle v, t, w, y \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s
- Dette fortsætter vi med, og fordi vi bruger en queue tager vi alt i en breadth-first rækkefølge

Breadth-first search

Eksempel

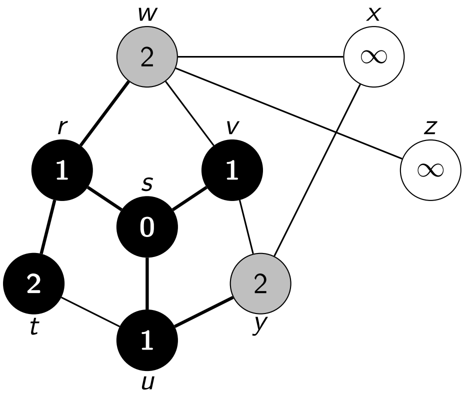


$$Q = \langle t, w, y \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s
- Dette fortsætter vi med, og fordi vi bruger en queue tager vi alt i en breadth-first rækkefølge
- Da vi allerede har besøgt alle v's naboer, så skal vi bare markere den som færdig

Breadth-first search

Eksempel

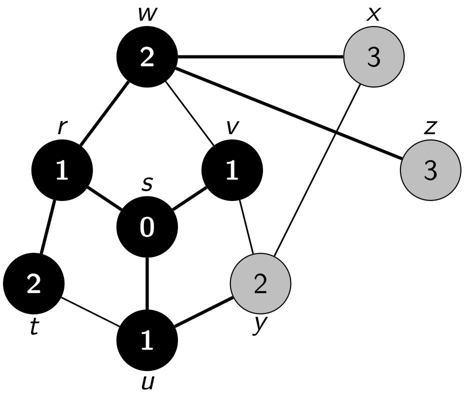


$$Q = \langle w, y \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s
- Dette fortsætter vi med, og fordi vi bruger en queue tager vi alt i en breadth-first rækkefølge
- Da vi allerede har besøgt alle v's naboer, så skal vi bare markere den som færdig

Breadth-first search

Eksempel

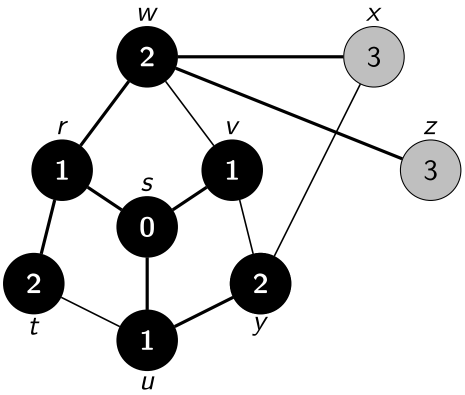


$$Q = \langle y, x, z \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s
- Dette fortsætter vi med, og fordi vi bruger en queue tager vi alt i en breadth-first rækkefølge
- Da vi allerede har besøgt alle v's naboer, så skal vi bare markere den som færdig

Breadth-first search

Eksempel

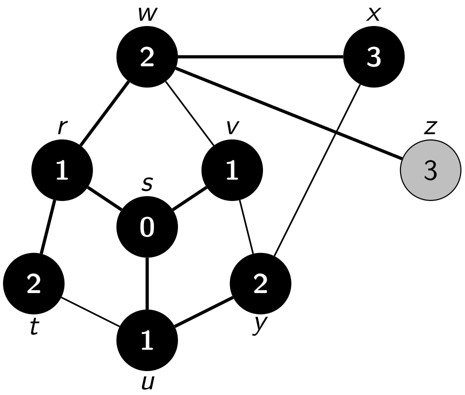


$$Q = \langle x, z \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s
- Dette fortsætter vi med, og fordi vi bruger en queue tager vi alt i en breadth-first rækkefølge
- Da vi allerede har besøgt alle v's naboer, så skal vi bare markere den som færdig

Breadth-first search

Eksempel

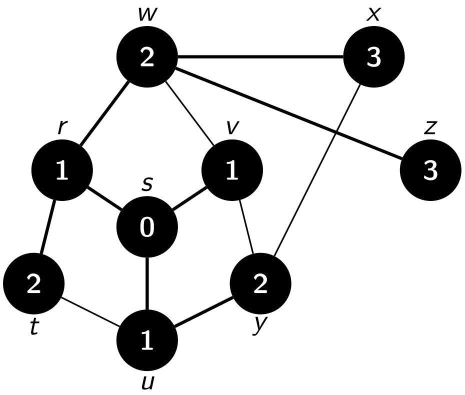


$$Q = \langle z \rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s
- Dette fortsætter vi med, og fordi vi bruger en queue tager vi alt i en breadth-first rækkefølge
- Da vi allerede har besøgt alle v's naboer, så skal vi bare markere den som færdig

Breadth-first search

Eksempel



$$Q = \langle\rangle$$

- Vi starter fra s og antager, at alle andre knuder er uendeligt langt væk fra s
- Så gemmer vi alle naboer til s i en kø Q og noterer, hvor langt væk de var
- Så dequeuer vi det første element i Q (som her er r), tilføjer dets naboer til Q og noterer afstanden til s
- Dette fortsætter vi med, og fordi vi bruger en queue tager vi alt i en breadth-first rækkefølge
- Da vi allerede har besøgt alle v's naboer, så skal vi bare markere den som færdig
- Når alle knuder er færdigbehandlet, udgør de fede kanter hele breadth-first træet T

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Vi annoterer hver knude u med attributterne
  - $u.color \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$
  - $u.d$, afstanden fra s
  - $u.\pi$, u's mor i breadth-first træet

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Vi annoterer hver knude u med attributterne
  - $u.color \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$
  - $u.d$, afstanden fra s
  - $u.\pi$, u's mor i breadth-first træet
- Så sætter vi s til at være grå, angiver dens afstand (til den selv) som 0 og indsætter den som første og hidtil eneste element i Q

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Vi annoterer hver knude u med attributterne
  - $u.color \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$
  - $u.d$, afstanden fra s
  - $u.\pi$, u's mor i breadth-first træet
- Så sætter vi s til at være grå, angiver dens afstand (til den selv) som 0 og indsætter den som første og hidtil eneste element i Q
- Sålænge Q ikke er tom, så dequeuer vi det forreste element u

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Vi annoterer hver knude u med attributterne
  - $u.color \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$
  - $u.d$, afstanden fra s
  - $u.\pi$, u's mor i breadth-first træet
- Så sætter vi s til at være grå, angiver dens afstand (til den selv) som 0 og indsætter den som første og hidtil eneste element i Q
- Sålænge Q ikke er tom, så dequeuer vi det forreste element u
- Herefter løber vi alle u's naboer igennem, og hvis de er hvide, så har vi endnu ikke behandlet dem

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Vi annoterer hver knude u med attributterne
  - $u.color \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$
  - $u.d$, afstanden fra s
  - $u.\pi$, u's mor i breadth-first træet
- Så sætter vi s til at være grå, angiver dens afstand (til den selv) som 0 og indsætter den som første og hidtil eneste element i Q
- Sålænge Q ikke er tom, så dequeuer vi det forreste element u
- Herefter løber vi alle u's naboer igennem, og hvis de er hvide, så har vi endnu ikke behandlet dem
- I så fald maler vi dem grå, sætter afstand og mor og tilføjer dem til Q

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Vi annoterer hver knude u med attributterne
  - $u.color \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$
  - $u.d$, afstanden fra s
  - $u.\pi$, u's mor i breadth-first træet
- Så sætter vi s til at være grå, angiver dens afstand (til den selv) som 0 og indsætter den som første og hidtil eneste element i Q
- Sålænge Q ikke er tom, så dequeuer vi det forreste element u
- Herefter løber vi alle u's naboer igennem, og hvis de er hvide, så har vi endnu ikke behandlet dem
- I så fald maler vi dem grå, sætter afstand og mor og tilføjer dem til Q
- Til sidst maler vi u sort

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Vi annoterer hver knude u med attributterne
  - $u.color \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$
  - $u.d$, afstanden fra s
  - $u.\pi$, u's mor i breadth-first træet
- Så sætter vi s til at være grå, angiver dens afstand (til den selv) som 0 og indsætter den som første og hidtil eneste element i Q
- Sålænge Q ikke er tom, så dequeuer vi det forreste element u
- Herefter løber vi alle u's naboer igennem, og hvis de er hvide, så har vi endnu ikke behandlet dem
- I så fald maler vi dem grå, sætter afstand og mor og tilføjer dem til Q
- Til sidst maler vi u sort

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Hvad er kompleksiteten?

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Hvad er kompleksiteten?
- Initialiseringen løber gennem alle knuder, ergo $O(|V|)$

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Hvad er kompleksiteten?
- Initialiseringen løber gennem alle knuder, ergo $O(|V|)$
- Efter initialisering gøres en knude aldrig hvid igen — ergo, enqueues og dequeues hver knude højst 1 gang, så $O(|V|)$

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Hvad er kompleksiteten?
- Initialiseringen løber gennem alle knuder, ergo $O(|V|)$
- Efter initialisering gøres en knude aldrig hvid igen — ergo, enqueues og dequeues hver knude højst 1 gang, så $O(|V|)$
- Dermed bliver hver adjacency liste også kun scannet 1 gang

Breadth-first search

Pseudo-kode

AALBORG UNIVERSITET

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Hvad er kompleksiteten?
- Initialiseringen løber gennem alle knuder, ergo $O(|V|)$
- Efter initialisering gøres en knude aldrig hvid igen — ergo, enqueues og dequeues hver knude højst 1 gang, så $O(|V|)$
- Dermed bliver hver adjacency liste også kun scannet 1 gang
- Den totale længde af alle adjacency lister er $\Theta(|E|)$

Breadth-first search

Pseudo-kode

BFS(G, s)

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
Enqueue(Q, s)
while Q ≠ ∅
    u = Dequeue(Q)
    for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
    u.color = BLACK
```

- Hvad er kompleksiteten?
- Initialiseringen løber gennem alle knuder, ergo $O(|V|)$
- Efter initialisering gøres en knude aldrig hvid igen — ergo, enqueues og dequeues hver knude højst 1 gang, så $O(|V|)$
- Dermed bliver hver adjacency liste også kun scannet 1 gang
- Den totale længde af alle adjacency lister er $\Theta(|E|)$
- Altså bliver den samlede køretid af BFS $O(|V| + |E|)$

Outline

Exercises

Super fedt! <3

På Moodle! Go! Fungerer det fint?

Outline

Depth-first search

Nu går vi dybere!

Den anden søgemetode hedder depth-first search (DFS).

Depth-first search

Nu går vi dybere!

Den anden søgemetode hedder depth-first search (DFS).

- Her tager vi ikke udgangspunkt i en bestemt start-knude, men i stedet går vi alle knuder i grafen igennem

Depth-first search

Nu går vi dybere!

Den anden søgemetode hedder depth-first search (DFS).

- Her tager vi ikke udgangspunkt i en bestemt start-knude, men i stedet går vi alle knuder i grafen igennem
- I stedet for at udfdorske hele 'niveauer' i grafen først, så følger vi stien fra en knude til dens 'dybeste' efterkommer inden, at vi går videre til naboer

Depth-first search

Nu går vi dybere!

AALBORG UNIVERSITET

Den anden søgemetode hedder depth-first search (DFS).

- Her tager vi ikke udgangspunkt i en bestemt start-knude, men i stedet går vi alle knuder i grafen igennem
- I stedet for at udfdorske hele 'niveauer' i grafen først, så følger vi stien fra en knude til dens 'dybeste' efterkommer inden, at vi går videre til naboer
- I stedet for et 'breadth-first tree', så får vi med DFS en 'depth-first forest' (der kan være flere rødder, som ikke er forbundne)

Depth-first search

Nu går vi dybere!

Den anden søgemetode hedder depth-first search (DFS).

- Her tager vi ikke udgangspunkt i en bestemt start-knude, men i stedet går vi alle knuder i grafen igennem
- I stedet for at udfdorske hele 'niveauer' i grafen først, så følger vi stien fra en knude til dens 'dybeste' efterkommer inden, at vi går videre til naboer
- I stedet for et 'breadth-first tree', så får vi med DFS en 'depth-first forest' (der kan være flere rødder, som ikke er forbundne)
  - Dette er en predecessor subgraph $G_\pi = (V, E_\pi)$ hvor

$$E_\pi = \{(v.\pi, v) : v \in V \text{and } v.\pi \neq \text{NIL}\}$$

Depth-first search

Algoritme

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)

DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Algoritme

- Proceduren DFS(G) initialiserer alle knuder og starter ‘timeren’

DFS(G)

for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-Visit(G, u)

DFS-Visit(G, u)

$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each vertex $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
        $v.\pi = u$
        DFS-Visit(G, v)
$time = time + 1$
$u.f = time$
$u.color = \text{BLACK}$

Depth-first search

Algoritme

- Proceduren DFS(G) initialiserer alle knuder og starter 'timeren'
- Herefter tager den knuderne en af gangen og kalder sub-proceduren DFS-Visit på dem (hvis de stadig er hvide)

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)

DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Algoritme

- Proceduren DFS(G) initialiserer alle knuder og starter 'timeren'
- Herefter tager den knuderne en af gangen og kalder sub-proceduren DFS-Visit på dem (hvis de stadig er hvide)
- I DFS-Visit noterer vi, hvornår vi 'opdagede' u ved at sætte $u.d = time$

DFS(G)

for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-Visit(G, u)

DFS-Visit(G, u)

$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each vertex $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
        $v.\pi = u$
        DFS-Visit(G, v)
$time = time + 1$
$u.f = time$
$u.color = \text{BLACK}$

Depth-first search

Algoritme

- Proceduren DFS(G) initialiserer alle knuder og starter 'timeren'
- Herefter tager den knuderne en af gangen og kalder sub-proceduren DFS-Visit på dem (hvis de stadig er hvide)
- I DFS-Visit noterer vi, hvornår vi 'opdagede' u ved at sætte $u.d = time$
- Herefter tager vi hver nabo v til u, sætter $v.\pi$ og kalder DFS-Visit rekursivt på v (såfremt den er hvid)

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Algoritme

- Proceduren DFS(G) initialiserer alle knuder og starter 'timeren'
- Herefter tager den knuderne en af gangen og kalder sub-proceduren DFS-Visit på dem (hvis de stadig er hvide)
- I DFS-Visit noterer vi, hvornår vi 'opdagede' u ved at sætte $u.d = time$
- Herefter tager vi hver nabo v til u, sætter $v.\pi$ og kalder DFS-Visit rekursivt på v (såfremt den er hvid)
- Til sidst noterer vi på u, hvornår vi færdiggjorde den ($u.f$) og maler den sort

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel

DFS(G)

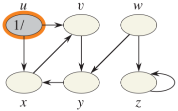
```
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel



DFS(G)

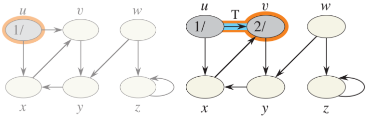
```
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel



DFS(G)

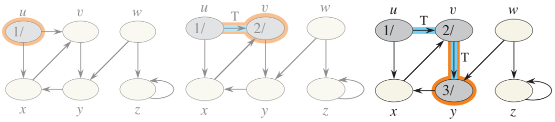
```
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
                v.π = u
                DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

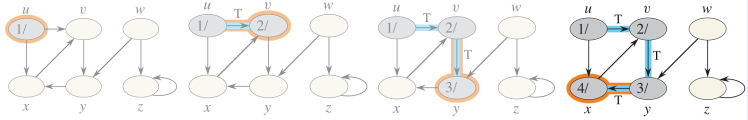
Depth-first search

Eksempel



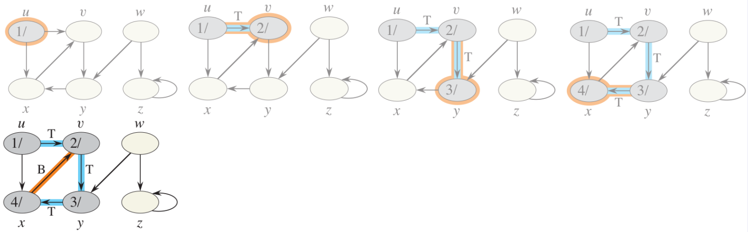

```
DFS(G)
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-Visit(G, u)

DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
                v.π = u
                DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```




```
DFS(G)
  for each vertex u ∈ G.V
      u.color = WHITE
      u.π = NIL
  time = 0
  for each vertex u ∈ G.V
      if u.color == WHITE
          DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY
  for each vertex v ∈ G.Adj[u]
      if v.color == WHITE
          v.π = u
          DFS-Visit(G, v)
  time = time + 1
  u.f = time
  u.color = BLACK
```

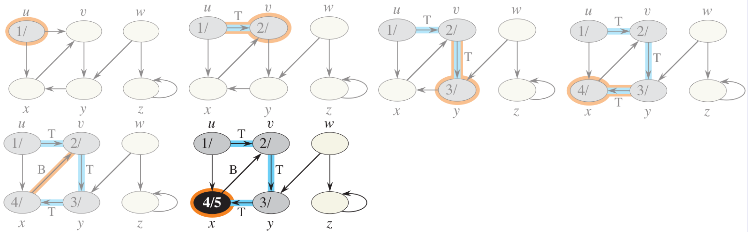



```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel



DFS(G)

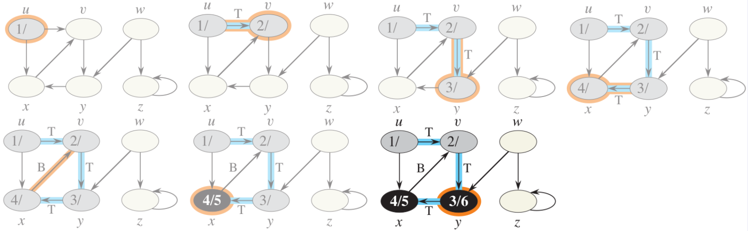
for each vertex $u \in G.V$
  $u.color = \text{WHITE}$
  $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
  if $u.color == \text{WHITE}$
    DFS-Visit(G, u)

DFS-Visit(G, u)

$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each vertex $v \in G.Adj[u]$
  if $v.color == \text{WHITE}$
    $v.\pi = u$
    DFS-Visit(G, v)
$time = time + 1$
$u.f = time$
$u.color = \text{BLACK}$

Depth-first search

Eksempel

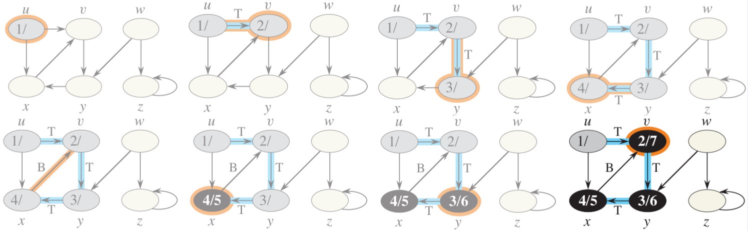



```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel

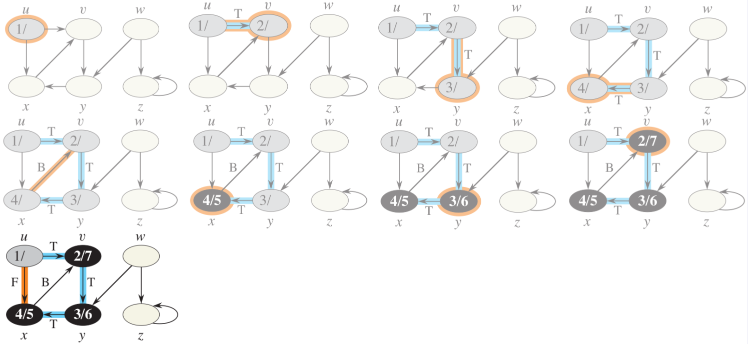


DFS(G)

for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-Visit(G, u)

DFS-Visit(G, u)

$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each vertex $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
        $v.\pi = u$
        DFS-Visit(G, v)
$time = time + 1$
$u.f = time$
$u.color = \text{BLACK}$



DFS(G)

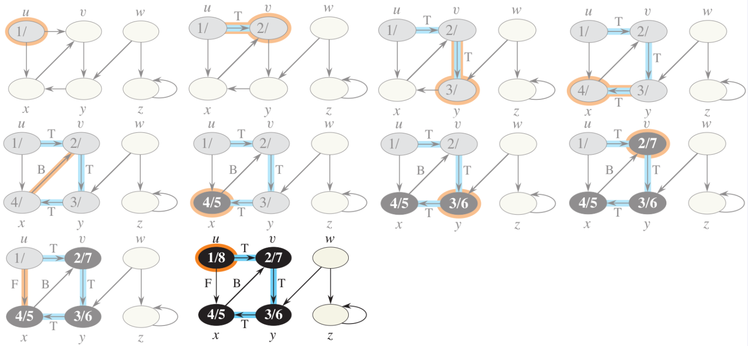
for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-Visit(G, u)

DFS-Visit(G, u)

$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each vertex $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
        $v.\pi = u$
        DFS-Visit(G, v)
$time = time + 1$
$u.f = time$
$u.color = \text{BLACK}$

Depth-first search

Eksempel

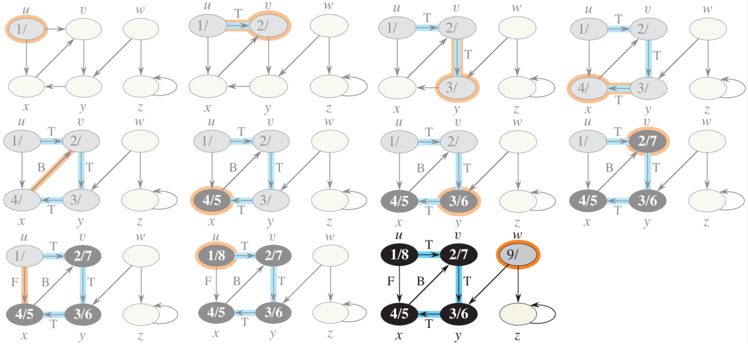



```
DFS(G)
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
                v.π = u
                DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel



DFS(G)

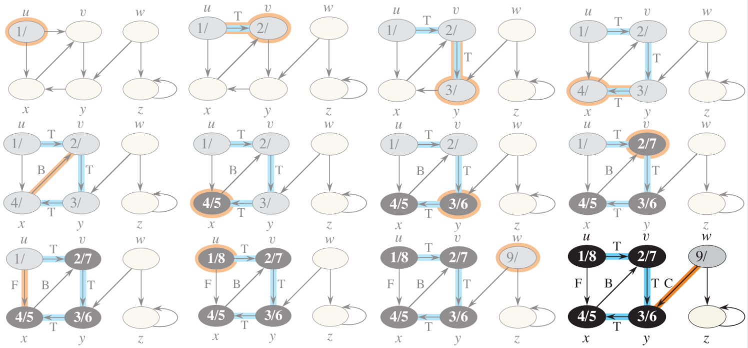
for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-Visit(G, u)

DFS-Visit(G, u)

$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each vertex $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
        $v.\pi = u$
        DFS-Visit(G, v)
$time = time + 1$
$u.f = time$
$u.color = \text{BLACK}$

Depth-first search

Eksempel

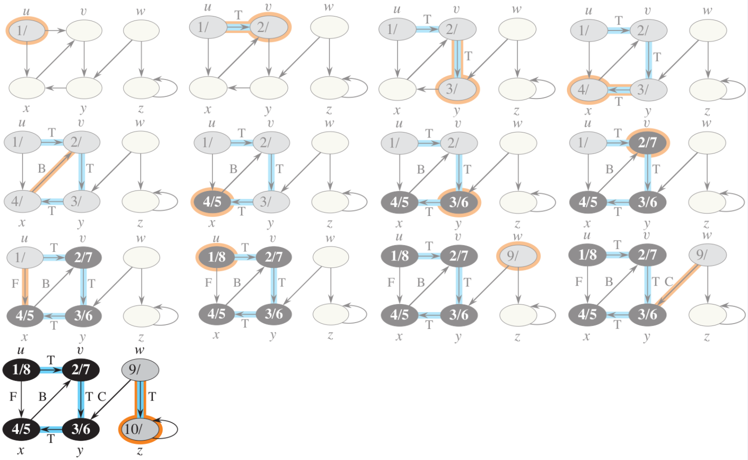



```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel



DFS(G)

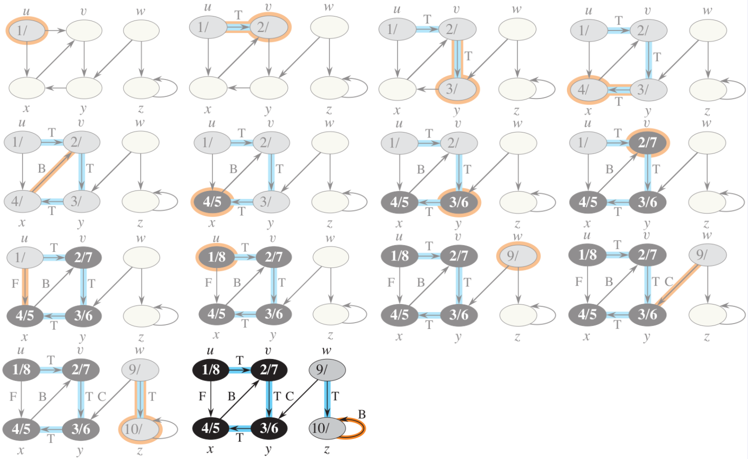
```
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
                v.π = u
                DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel

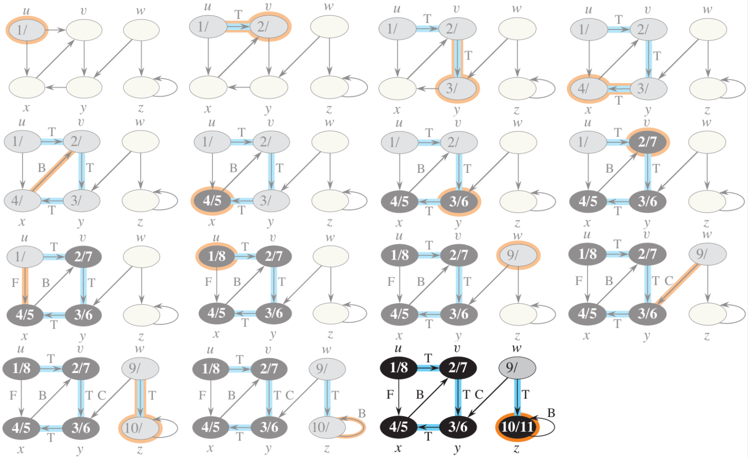



```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel

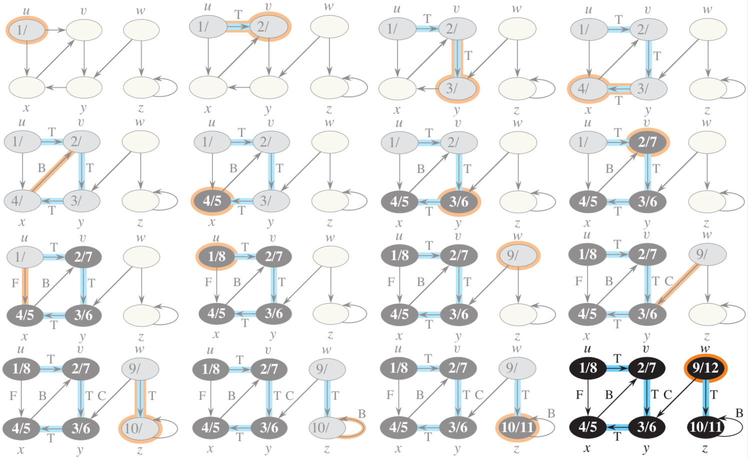


```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)

DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel

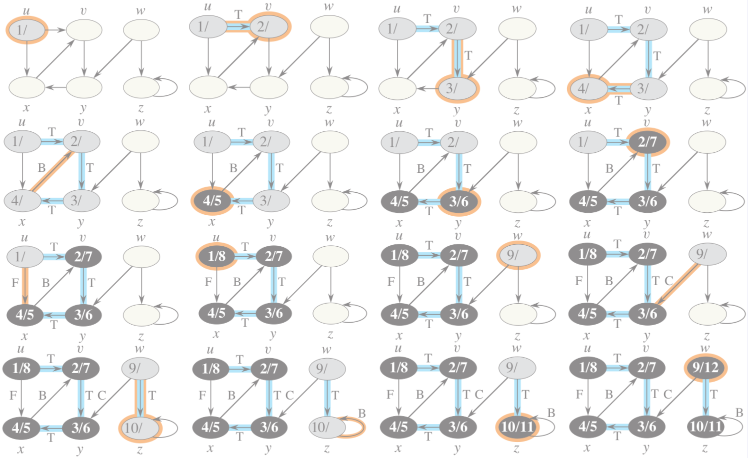


```
DFS(G)
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-Visit(G, u)

DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
                v.π = u
                DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel

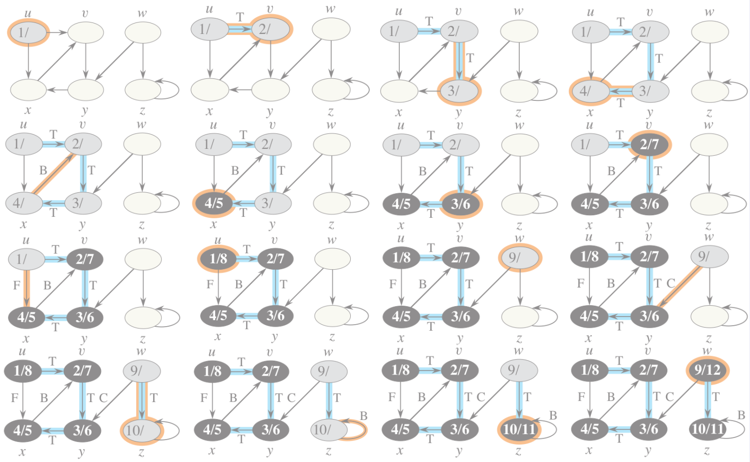



```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel

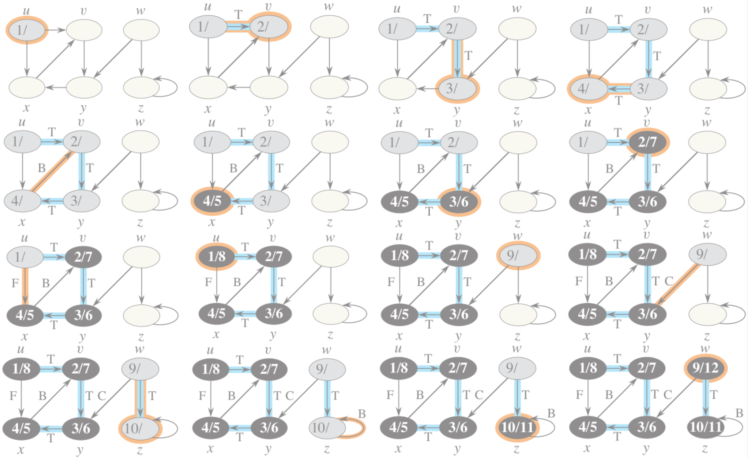


```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)

DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Depth-first search

Eksempel



DFS(G)

for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-Visit(G, u)

DFS-Visit(G, u)

$time = time + 1$
$u.d = time$
$u.color = \text{GRAY}$
for each vertex $v \in G.Adj[u]$
    if $v.color == \text{WHITE}$
        $v.\pi = u$
        DFS-Visit(G, v)
$time = time + 1$
$u.f = time$
$u.color = \text{BLACK}$

Depth-first search

Analyse

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Hvor lang tid tager det?

Depth-first search

Analyse

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-Visit(G, v)
time = time + 1
u.f = time
u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS?

Depth-first search

Analyse

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
1   time = time + 1
2   u.d = time
3   u.color = GRAY
4   for each vertex v ∈ G.Adj[u]
5       if v.color == WHITE
6           v.π = u
7           DFS-Visit(G, v)
8   time = time + 1
9   u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$

Depth-first search

Analyse

DFS(G)

```
1  for each vertex u ∈ G.V
2        u.color = WHITE
3        u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6        if u.color == WHITE
7              DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5        if v.color == WHITE
 6              v.π = u
 7              DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS?

Depth-first search

Analyse

DFS(G)

1 **for** each vertex $u \in G.V$
2 $\quad u.color = \text{WHITE}$
3 $\quad u.\pi = \text{NIL}$
4 $time = 0$
5 **for** each vertex $u \in G.V$
6 $\quad$ **if** $u.color == \text{WHITE}$
7 $\quad\quad$ DFS-Visit(G, u)

DFS-Visit(G, u)

1 $time = time + 1$
2 $u.d = time$
3 $u.color = \text{GRAY}$
4 **for** each vertex $v \in G.Adj[u]$
5 $\quad$ **if** $v.color == \text{WHITE}$
6 $\quad\quad v.\pi = u$
7 $\quad\quad$ DFS-Visit(G, v)
8 $time = time + 1$
9 $u.f = time$
10 $u.color = \text{BLACK}$

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?

Depth-first search

Analyse

DFS(G)

1 **for** each vertex $u \in G.V$
2 $\quad u.color = \text{WHITE}$
3 $\quad u.\pi = \text{NIL}$
4 $time = 0$
5 **for** each vertex $u \in G.V$
6 $\quad$ **if** $u.color == \text{WHITE}$
7 $\qquad$ DFS-Visit(G, u)

DFS-Visit(G, u)

1 $time = time + 1$
2 $u.d = time$
3 $u.color = \text{GRAY}$
4 **for** each vertex $v \in G.Adj[u]$
5 $\quad$ **if** $v.color == \text{WHITE}$
6 $\qquad v.\pi = u$
7 $\qquad$ DFS-Visit(G, v)
8 $time = time + 1$
9 $u.f = time$
10 $u.color = \text{BLACK}$

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit?

Depth-first search

Analyse

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$

Depth-first search

Analyse

```
DFS(G)
1   for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4   time = 0
5   for each vertex u ∈ G.V
6       if u.color == WHITE
7           DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit?

Depth-first search

Analyse

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit? $O(|E|)$ og så noget med DFS-Visit eller hvad?

Depth-first search

Analyse

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit? $O(|E|)$ og så noget med DFS-Visit eller hvad?
- Linie 8-10 er konstante

Depth-first search

Analyse

DFS(G)

1 **for** each vertex $u \in G.V$
2 $\quad u.color = \text{WHITE}$
3 $\quad u.\pi = \text{NIL}$
4 $time = 0$
5 **for** each vertex $u \in G.V$
6 $\quad$ **if** $u.color == \text{WHITE}$
7 $\qquad \text{DFS-Visit}(G, u)$

DFS-Visit(G, u)

1 $time = time + 1$
2 $u.d = time$
3 $u.color = \text{GRAY}$
4 **for** each vertex $v \in G.Adj[u]$
5 $\quad$ **if** $v.color == \text{WHITE}$
6 $\qquad v.\pi = u$
7 $\qquad \text{DFS-Visit}(G, v)$
8 $time = time + 1$
9 $u.f = time$
10 $u.color = \text{BLACK}$

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit? $O(|E|)$ og så noget med DFS-Visit eller hvad?
- Linie 8-10 er konstante
- Så lander vi i noget ala $O(|V| \cdot |E|)$?

Depth-first search

Analyse

DFS(G)

```
1   for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4   time = 0
5   for each vertex u ∈ G.V
6       if u.color == WHITE
7           DFS-Visit(G, u)
```

DFS-Visit(G, u)

```
1   time = time + 1
2   u.d = time
3   u.color = GRAY
4   for each vertex v ∈ G.Adj[u]
5       if v.color == WHITE
6           v.π = u
7           DFS-Visit(G, v)
8   time = time + 1
9   u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit? $O(|E|)$ og så noget med DFS-Visit eller hvad?
- Linie 8-10 er konstante
- Så lander vi i noget ala $O(|V| \cdot |E|)$?
  - Nej! Strukturen hjælper os:

Depth-first search

Analyse

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit? $O(|E|)$ og så noget med DFS-Visit eller hvad?
- Linie 8-10 er konstante
- Så lander vi i noget ala $O(|V| \cdot |E|)$?
  - Nej! Strukturen hjælper os:
  - DFS-Visit kaldes kun 1 gang pr knude, $O(|V|)$

Depth-first search

Analyse

```
DFS(G)
1   for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4   time = 0
5   for each vertex u ∈ G.V
6       if u.color == WHITE
7           DFS-Visit(G, u)

DFS-Visit(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit? $O(|E|)$ og så noget med DFS-Visit eller hvad?
- Linie 8-10 er konstante
- Så lander vi i noget ala $O(|V| \cdot |E|)$?
  - Nej! Strukturen hjælper os:
  - DFS-Visit kaldes kun 1 gang pr knude, $O(|V|)$
  - Og vi følger kun hver kant 1 gang, $O(|E|)$

Depth-first search

Analyse


```
DFS(G)
1   for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4   time = 0
5   for each vertex u ∈ G.V
6       if u.color == WHITE
7           DFS-Visit(G, u)

DFS-Visit(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK
```

Hvor lang tid tager det?

- Linie 1-4 i DFS? $O(|V|)$
- Linie 5-7 i DFS? $O(|V| \cdot O(\text{DFS-Visit}))$
  - Hmmm, så en rekursion?
- Linie 1-3 i DFS-Visit? $\Theta(1)$
- Linie 4-7 i DFS-Visit? $O(|E|)$ og så noget med DFS-Visit eller hvad?
- Linie 8-10 er konstante
- Så lander vi i noget ala $O(|V| \cdot |E|)$?
  - Nej! Strukturen hjælper os:
  - DFS-Visit kaldes kun 1 gang pr knude, $O(|V|)$
  - Og vi følger kun hver kant 1 gang, $O(|E|)$
- Så vi lander på velkendte $O(|V| + |E|)$

Outline

Topologisk sortering

Tilbage til sortering

Nu vender vi lige for en kort stund snuden tilbage til sortering og ser på en ny sorteringsalgoritme: Topologisk sortering

Topologisk sortering

Tilbage til sortering

Nu vender vi lige for en kort stund snuden tilbage til sortering og ser på en ny sorteringsalgoritme: Topologisk sortering

- Algoritme til at sortere knuder i en directed acyclic graph (DAG)

Topologisk sortering

Tilbage til sortering

Nu vender vi lige for en kort stund snuden tilbage til sortering og ser på en ny sorteringsalgoritme: Topologisk sortering

- Algoritme til at sortere knuder i en directed acyclic graph (DAG)
- Rækkefølgen kræver bare, at hvis der er en kant fra u til v i G, så skal u komme før v i output-listen

Topologisk sortering

Tilbage til sortering

Nu vender vi lige for en kort stund snuden tilbage til sortering og ser på en ny sorteringsalgoritme: Topologisk sortering

- Algoritme til at sortere knuder i en directed acyclic graph (DAG)
- Rækkefølgen kræver bare, at hvis der er en kant fra u til v i G, så skal u komme før v i output-listen
- Dette er smart, f.eks. når man skal kompilere software, hvor forskellige sub-moduler er afhængige af hinanden

Topologisk sortering

Pseudo pseudo-kode

Algoritmen?

Topological-Sort(G)

```
call DFS(G) to compute finish times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

Topologisk sortering

Pseudo pseudo-kode

Algoritmen?

Topological-Sort(G)

1. call DFS(G) to compute finish times $v.f$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. **return** the linked list of vertices

- Kompleksitet?

Topologisk sortering

Pseudo pseudo-kode

Algoritmen?

Topological-Sort(G)

1 call DFS(G) to compute finish times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

- Kompleksitet?
- $O(|V| + |E|)$!

Topologisk sortering

Pseudo pseudo-kode

Algoritmen?

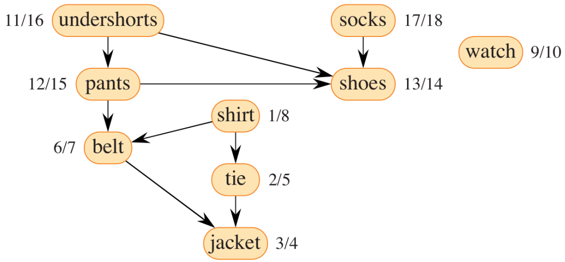
Topological-Sort(G)

1. call DFS(G) to compute finish times $v.f$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. **return** the linked list of vertices

- Kompleksitet?
- $O(|V| + |E|)$!
- Det gør det til den hurtigeste sorteringsalgoritme, som vi har set

Topologisk sortering

Eksempel

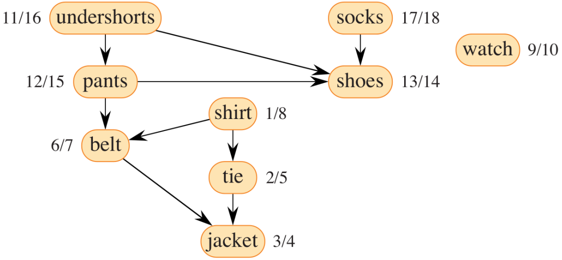


Topological-Sort(G)

1 call DFS(G) to compute
finish times $v.f$ for each vertex x
2 as each vertex is finished, insert
it onto the front of a linked list
3 **return** the linked list of vertices

Topologisk sortering

Eksempel



Topological-Sort(G)

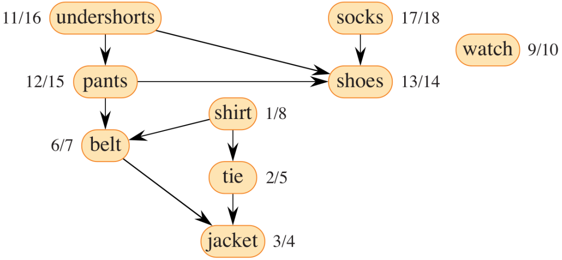
call DFS(G) to compute
finish times $v.f$ for each vertex x
as each vertex is finished, insert
it onto the front of a linked list
return the linked list of vertices

jacket

3/4

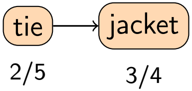
Topologisk sortering

Eksempel



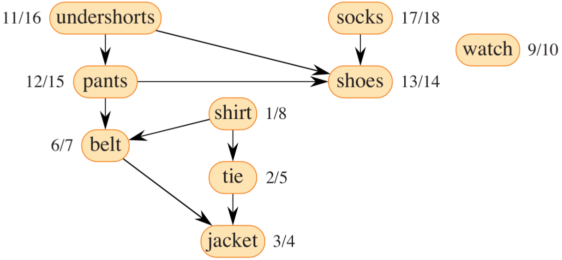
Topological-Sort(G)

```
call DFS(G) to compute
        finish times v.f for each vertex x
as each vertex is finished, insert
        it onto the front of a linked list
return the linked list of vertices
```



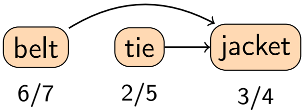
Topologisk sortering

Eksempel

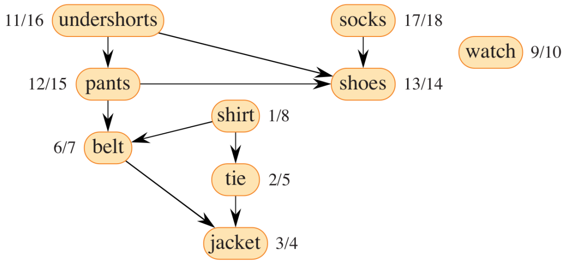


Topological-Sort(G)

call DFS(G) to compute
finish times $v.f$ for each vertex x
as each vertex is finished, insert
it onto the front of a linked list
return the linked list of vertices

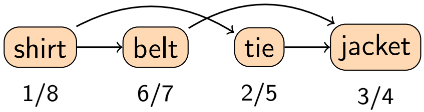


Topologisk sortering
Eksempel



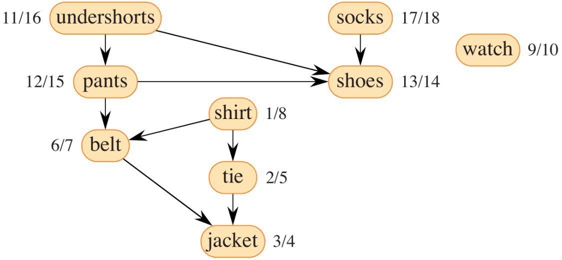
Topological-Sort(G)

1 call DFS(G) to compute
finish times $v.f$ for each vertex x
2 as each vertex is finished, insert
it onto the front of a linked list
3 **return** the linked list of vertices



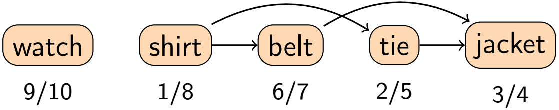
Topologisk sortering

Eksempel



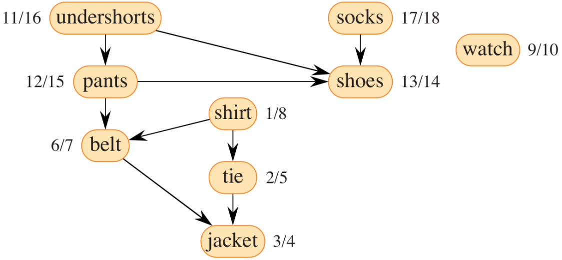
Topological-Sort(G)

1 call DFS(G) to compute finish times $v.f$ for each vertex x
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices



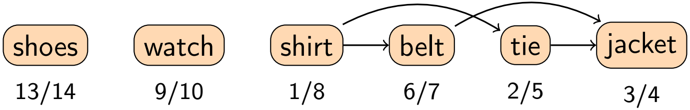
Topologisk sortering

Eksempel



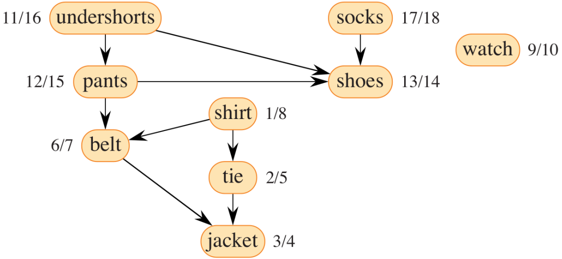
Topological-Sort(G)

call DFS(G) to compute
finish times $v.f$ for each vertex x
as each vertex is finished, insert
it onto the front of a linked list
return the linked list of vertices



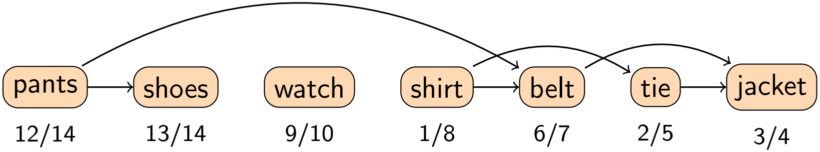
Topologisk sortering

Eksempel



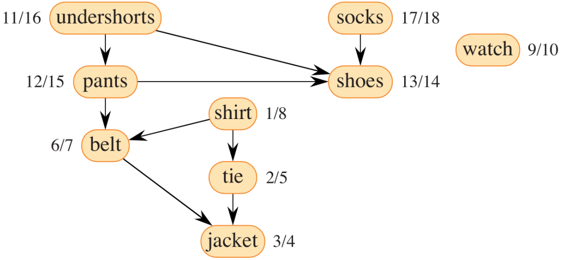
Topological-Sort(G)

1 call DFS(G) to compute
finish times $v.f$ for each vertex x
2 as each vertex is finished, insert
it onto the front of a linked list
3 **return** the linked list of vertices



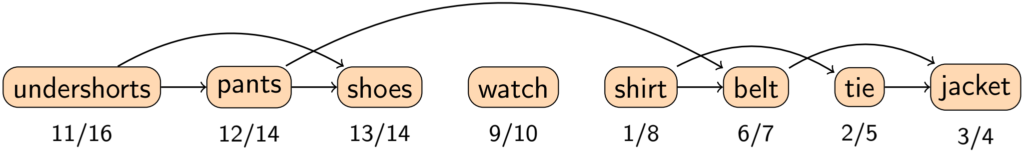
Topologisk sortering

Eksempel



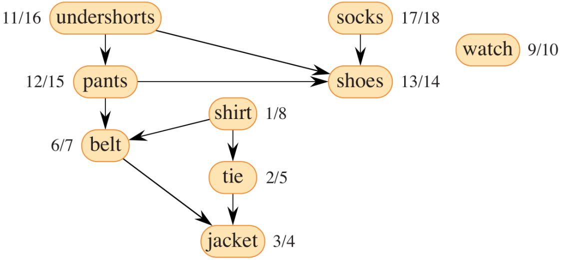
Topological-Sort(G)

call DFS(G) to compute
finish times $v.f$ for each vertex x
as each vertex is finished, insert
it onto the front of a linked list
return the linked list of vertices



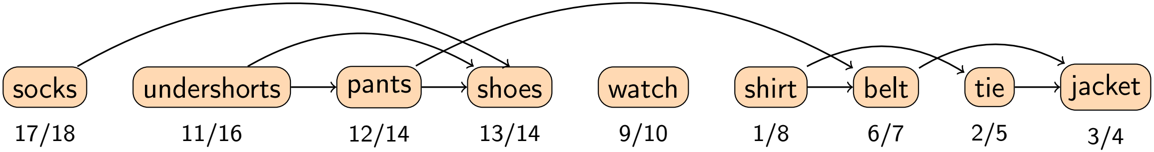
Topologisk sortering

Eksempel



Topological-Sort(G)

1 call DFS(G) to compute
finish times $v.f$ for each vertex x
2 as each vertex is finished, insert
it onto the front of a linked list
3 **return** the linked list of vertices



Tak for i dag!

Flere exercises..

Den bedste måde ikke at snyde sig selv på er lave exercises!