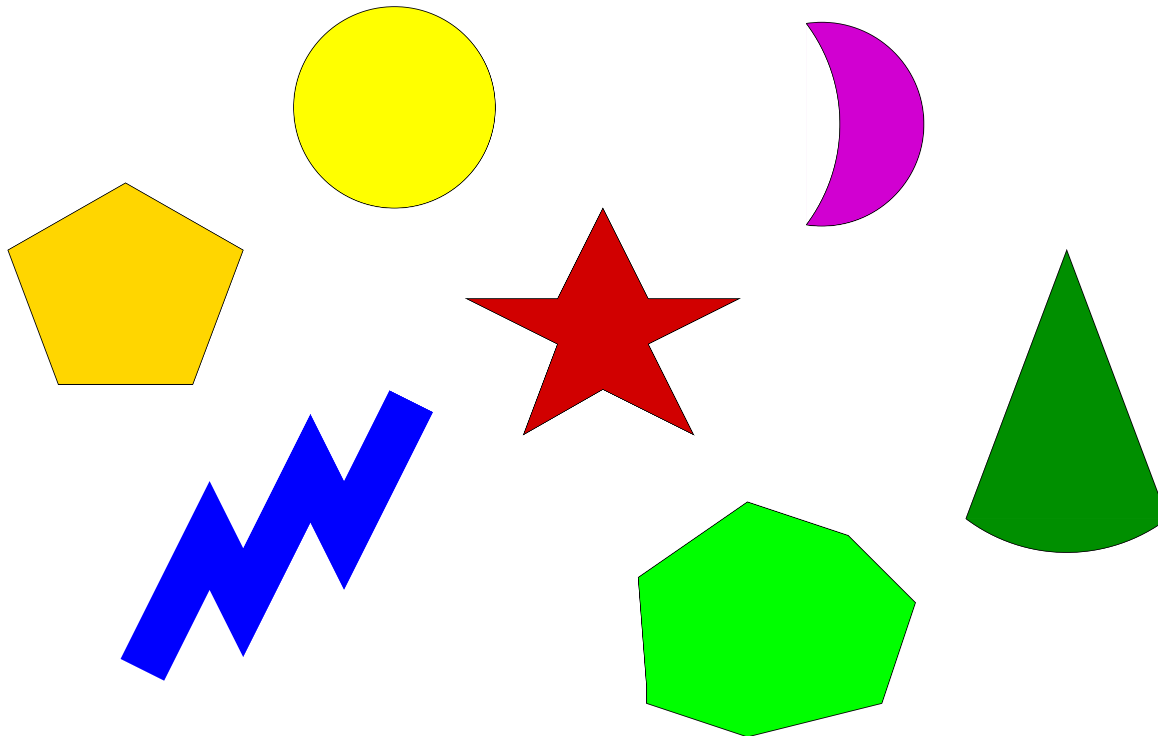
Vijfde college algoritmiek

3 maart 2016

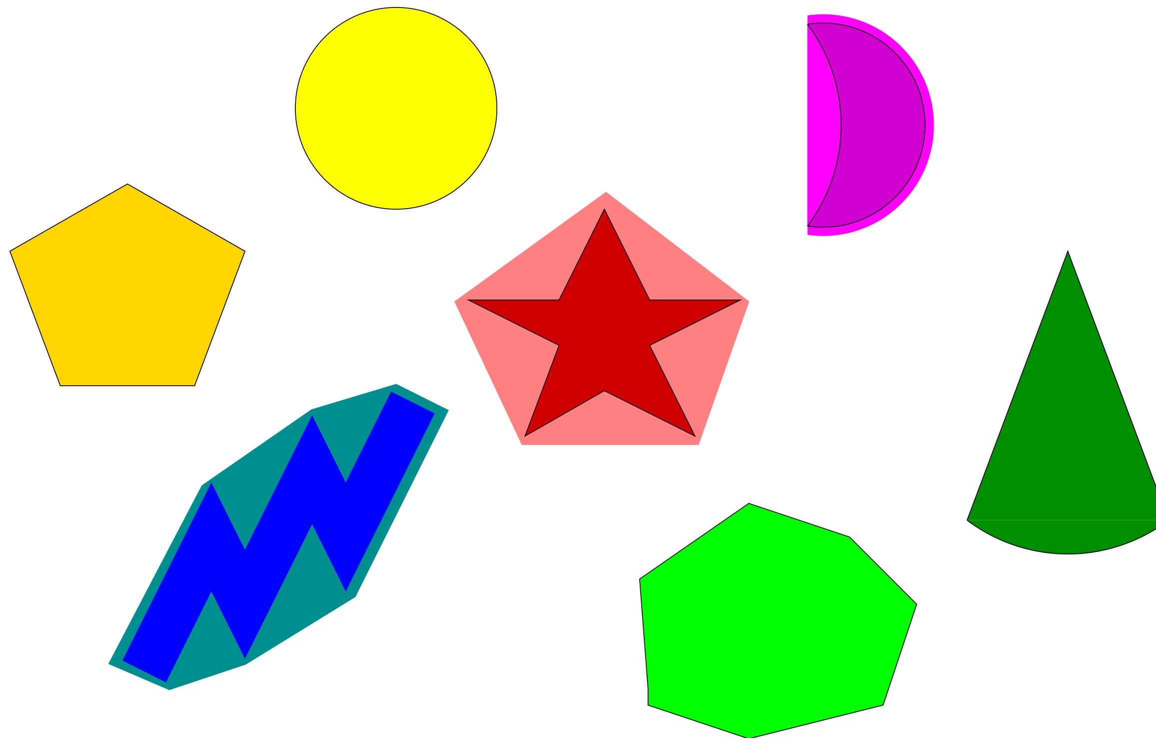
Brute Force, Exhaustive search en Backtracking

Een verzameling punten in het platte vlak heet convex als voor elk tweetal punten uit die verzameling geldt dat het verbindend lijnstuk ook weer in die verzameling ligt.



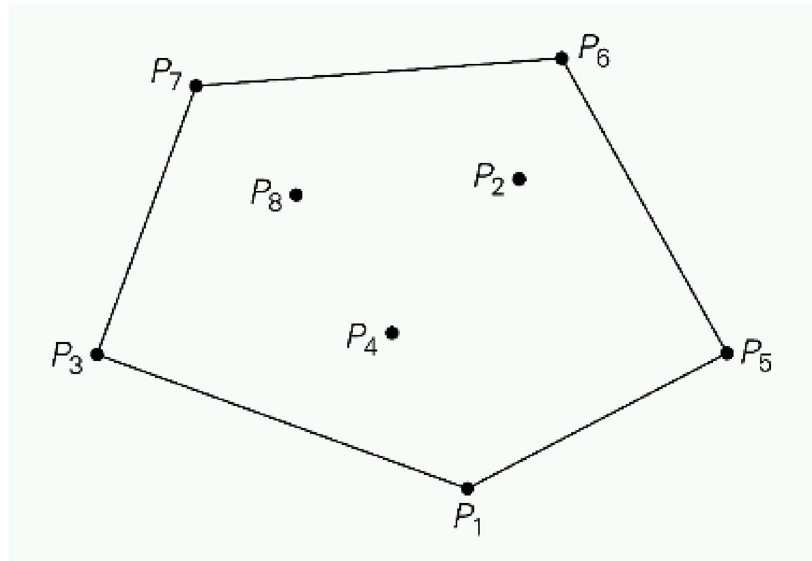
Convexe en niet-convexe vormen

De convex hull (convex omhulsel) van een verzameling S van punten in $\mathbf{R}^2$ is de kleinste convexe verzameling die S bevat.



Convexe omhulsels

Stelling: de convex hull van een verzameling S van $n > 2$ punten in $\mathbf{R}^2$ (niet alle op één lijn) is een convexe veelhoek (polygoon) met als hoekpunten enkele punten uit S.



De convex hull van de verzameling $\{P_1, P_2, \ldots, P_8\}$ is de convexe veelhoek met hoekpunten P_1, P_5, P_6, P_7 en P_3

We baseren ons brute force algoritme op de volgende observatie: een lijnstuk P_iP_j maakt deel uit van de rand van de convex hull van $\{P_1, P_2, \ldots, P_8\}$ d.e.s.d.a. alle andere punten van de verzameling aan een en dezelfde kant van de lijn door P_i en P_j liggen.

Brute force: Ga *voor elk tweetal* punten $P_i = (x_i, y_i)$ en $P_j = (x_j, y_j)$ na of *alle andere punten* aan dezelfde kant van de lijn $(y_j - y_i)x + (x_i - x_j)y = x_iy_j - y_ix_j$ liggen. Zo ja, dan is P_iP_j dus deel van de convex hull.

Complexiteit: $O(n^3)$

Opmerking: het kan veel beter, namelijk $O(n \lg n)$

Brute force:

- **Voordelen**:
  - algemeen toepasbaar
  - eenvoudig
  - levert voor een aantal belangrijke problemen (zoeken, patroonherkenning) een zeer behoorlijk algoritme op

- **Nadelen**:
  - levert meestal geen efficiënt algoritme op
  - soms onacceptabel langzaam

Exhaustive search: brute force benadering voor problemen die te maken hebben met het vinden van een element met een speciale eigenschap binnen een verzameling van bijv. permutaties of deelverzamelingen of toestanden of ...

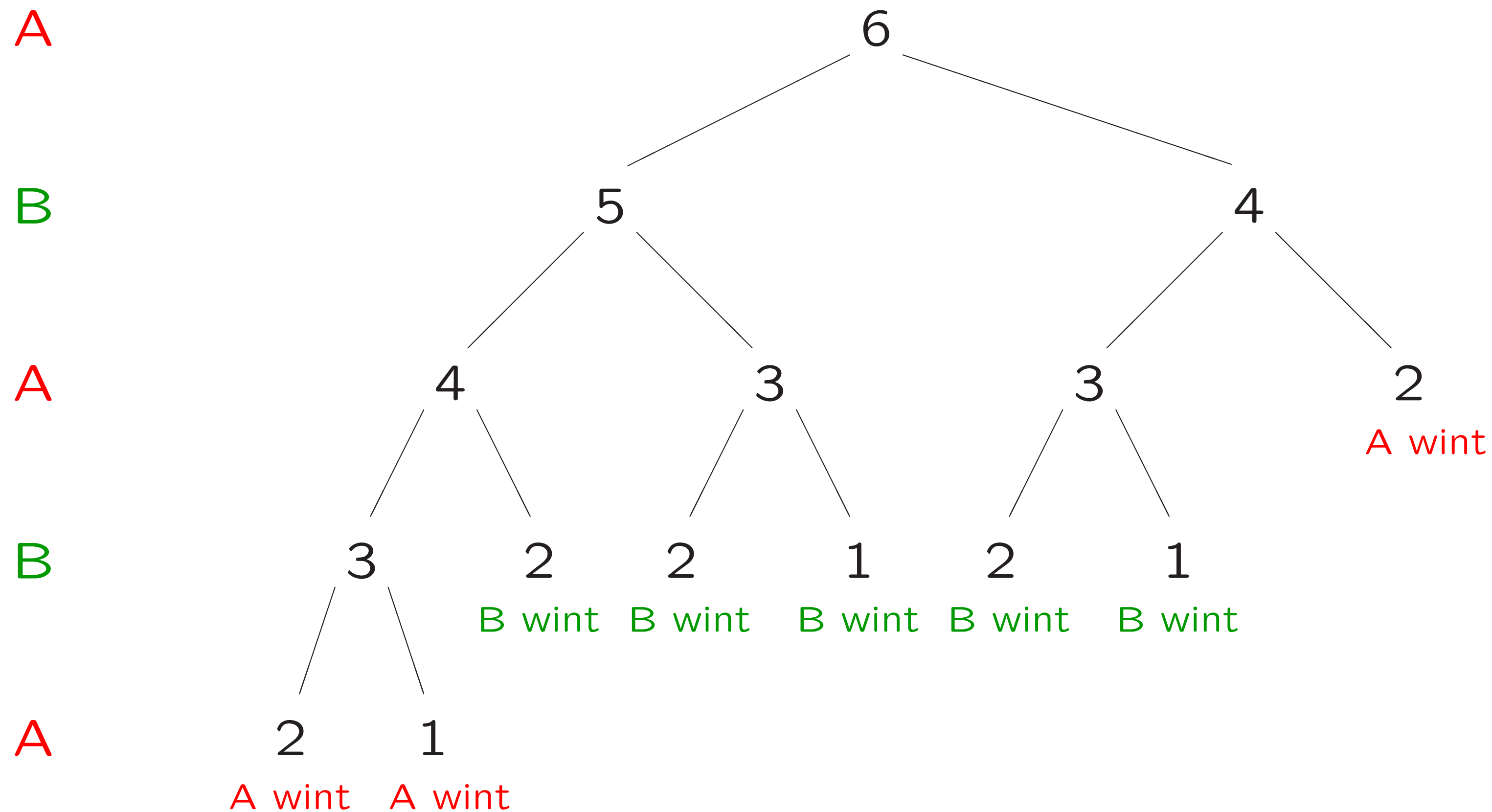
Methode:

. construeer op een systematische manier alle kandidaatoplossingen, bijvoorbeeld alle permutaties van de getallen 1 t/m n

. evalueer elk van deze mogelijke oplossingen

. retourneer een/de kandidaatoplossing met de gevraagde eigenschap (als die bestaat) (*)

(*) soms, zoals bij optimalisatieproblemen, *moet* je daartoe alle kandidaatoplossingen gezien hebben



Exhaustive search: doorloop (*als het ware*) de hele spelboom om te bepalen of een stand winnend is. Alle toestanden/alle spelverlopen worden zo bekeken. Je kunt stoppen zodra je een winnende zet gevonden hebt.

Belangrijke observatie:

> een stand is winnend voor degene die aan de beurt is, dan en slechts dan als ten minste één van zijn directe vervolgstanden niet winnend is voor de tegenstander

$\Longrightarrow$ RECURSIE

Met terugzetten:

```
winnend(stand)::

    if eindstand(stand) then
        //  makkelijk; bijv return false;
    else
        for alle mogelijke zetten i do
            doezet(stand,i);
            if not winnend(stand) then
                undoezet(stand,i);
                return true;
            undoezet(stand,i);
        od
        return false;
    fi
```

Zie ook Programmeermethoden (college over recursie)

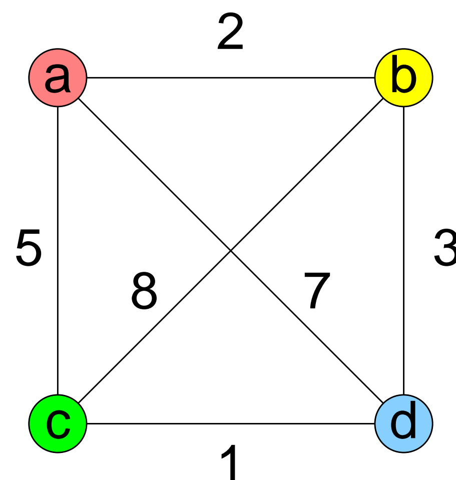
Traveling Salesman Problem (handelsreizigersprobleem)

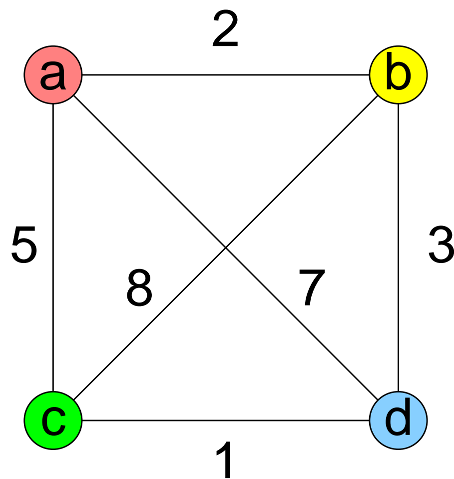
Gegeven n steden waarvan alle onderlinge afstanden bekend zijn.

Gevraagd: de/een kortste route die elke stad precies één keer aandoet, en weer terugkeert in het vertrekpunt.

Ofwel: vind de/een kortste Hamiltonkring in een samenhangende gewogen (volledige) graaf.

Voorbeeld:





Route	**Lengte**
a → b → c → d → a	$2+8+1+7=18$
a → b → d → c → a	$2+3+1+5=11$
a → c → b → d → a	$5+8+3+7=23$
a → c → d → b → a	$5+1+3+2=11$
a → d → b → c → a	$7+3+8+5=23$
a → d → c → b → a	$7+1+8+2=18$

Complexiteit: $\Omega((n-1)!)$,
immers alle $(n-1)!$ mogelijke Hamiltonkringen worden bekeken.

Knapzakprobleem

Gegeven n objecten, met gewicht $w_1, \ldots, w_n$ en waarde $v_1, \ldots, v_n$, en een knapzak met capaciteit W.

Gevraagd: de meest waardevolle deelverzameling der objecten die in de knapzak past (dus met totaalgewicht $\leq W$).

Voorbeeld:

object	gewicht	waarde
1	8	42
2	3	14
3	4	40
4	5	27

knapzakcapaciteit 12

deelverzameling	gewicht	waarde
$\emptyset$	0	0
$\{1\}$	8	42
$\{2\}$	3	14
$\{3\}$	4	40
$\{4\}$	5	27
$\{1,2\}$	11	56
$\{1,3\}$	12	82
$\{1,4\}$	13	te zwaar
$\{2,3\}$	7	54
$\{2,4\}$	8	41
$\{3,4\}$	9	67
$\{1,2,3\}$	15	te zwaar
$\{1,2,4\}$	16	te zwaar
$\{1,3,4\}$	17	te zwaar
$\{2,3,4\}$	12	81
$\{1,2,3,4\}$	20	te zwaar

Complexiteit: $\Omega(2^n)$,
immers alle 2^n deelverzamelingen van n objecten worden bekeken.

Assignmentproblem (toewijzingsprobleem)

Gegeven n personen en n taken (jobs). Persoon i kan taak j doen voor $\texttt{kosten}[i][j]$ euro.

Gevraagd: de/een toewijzing van de personen aan de jobs (één persoon per job en één job per persoon) met minimale kosten.

Voorbeeld:

	job 1	job 2	job 3	job 4
Anna	9	2	7	8
Bob	6	4	3	7
Carla	5	8	1	8
David	7	6	9	4

$n = 4$

	job 1	job 2	job 3	job 4
Anna	9	2	7	8
Bob	6	4	3	7
Carla	5	8	1	8
David	7	6	9	4

$n = 4$

```
1,2,3,4 -> 9+4+1+4 = 18
1,2,4,3 -> 9+4+8+9 = 30
1,3,2,4 -> 9+3+8+4 = 24
1,3,4,2 -> 9+3+8+6 = 26
1,4,2,3 -> 9+7+8+9 = 33
1,4,3,2 -> 9+7+1+6 = 23
2,1,3,4 -> 2+6+1+4 = 13
2,1,4,3 -> 2+6+8+9 = 25
2,3,1,4 -> ..
2,3,4,1 -> ..
2,4,1,3 -> ..
2,4,3,1 -> ..
3,1,2,4 -> ..
3,1,4,2 -> ..
3,2,1,4 -> ..
3,2,4,1 -> ..
3,4,1,2 -> ..
3,4,2,1 -> ..
4,1,2,3 -> ..
4,1,3,2 -> ..
4,2,1,3 -> ..
4,2,3,1 -> ..
4,3,1,2 -> ..
4,3,2,1 -> ..
```

De goedkoopste toewijzing is hier `2,1,3,4`, met kosten 13.

Complexiteit: $\Omega(n!)$,
immers alle $n!$ mogelijke toewijzingen worden bekeken.

Eindconclusie Exhaustive Search

* Exhaustive search algoritmen werken **alleen voor kleine probleeminstanties** in acceptabele tijd
* Voor veel problemen zijn er veel efficiëntere algoritmen bekend (Eulerkring, kortste paden, toewijzingsprobleem)
* Voor andere problemen is exhaustive search (of varianten daarop) in essentie de enig bekende oplossing (handelsreizigersprobleem, knapzakprobleem)

Graafwandelingen

- Bij veel (graaf)problemen is het nodig om alle knopen van de graaf op een systematische manier te bezoeken

- **Graafwandelingen:**

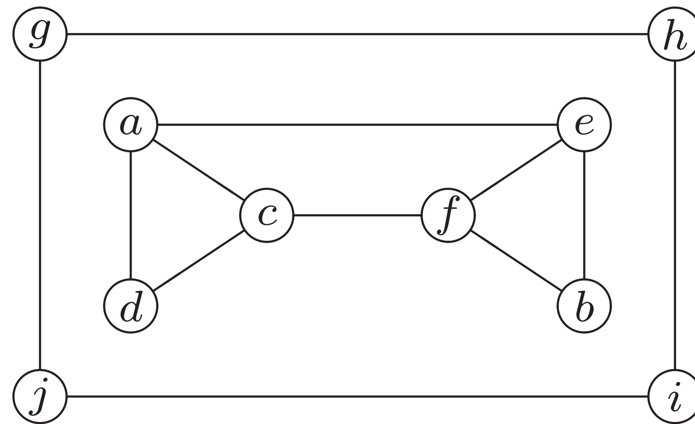
1. Depth-first-search: te vergelijken met WLR-wandeling bij bomen

2. Breadth-first-search: te vergelijken met nivo-orde wandeling bij bomen

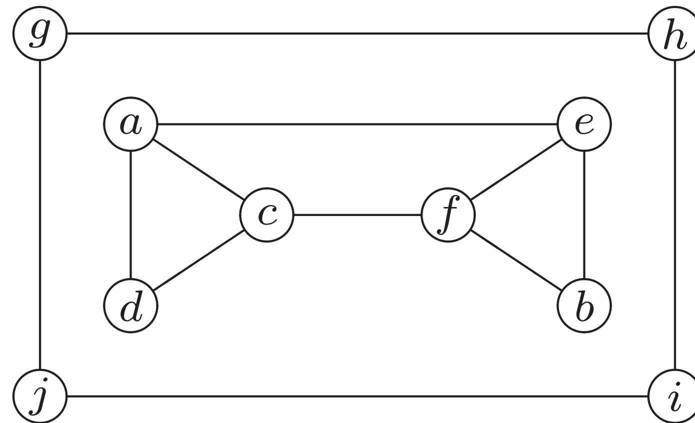
Depth-first-search

- De wandeling begint in een gegeven knoop v van de graaf.
- Vanuit een zojuist bezochte knoop wordt vervolgens steeds een aangrenzende -nog onbezochte- knoop bezocht, en vandaaruit op dezelfde manier verder gelopen tot je niet verder kan.
- In dat geval wordt teruggegaan naar de knoop waar je net vandaan kwam, en wordt een andere aangrenzende knoop daarvan bezocht, en zo verder tot je weer bij v terug bent.
- Aangrenzende knopen kunnen bijvoorbeeld altijd in alfabetische volgorde bezocht worden.
- Een knoop wordt steeds als reeds bezocht gemarkeerd op het moment dat deze voor de eerste keer bekeken wordt.
- Alle knopen die vanuit v bereikbaar zijn worden zo precies één keer bezocht. Voor niet-samenhangende grafen moet bovenstaande telkens herhaald worden vanuit een resterende, nog niet bezochte knoop.
- Depth-first-search kan recursief of met behulp van een stapel worden geïmplementeerd.

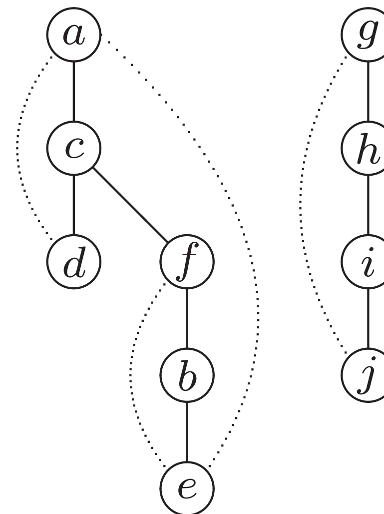
Depth-First Search



Depth-First Search



	$e_{6,2}$	
	$b_{5,3}$	$j_{10,7}$
$d_{3,1}$	$f_{4,4}$	$i_{9,8}$
$c_{2,5}$		$h_{8,9}$
$a_{1,6}$		$g_{7,10}$



```
ALGORITME DFS (G)
    // Implementeert DFS wandeling door gegeven graaf
    // Invoer: Graaf G = (V,E)
    // Uitvoer: Graaf G met zijn knopen genummerd in de volgorde
    //          waarin ze bij DFS wandeling voor het eerst worden ontdekt

{
    for  elke knoop v in V  do
        mark[v] = 0;   // nog niet bezocht
    od
    teller = 0;
    for  elke knoop v in V  do
        if  mark[v] == 0  then
            dfs (v);
        fi
    od
}
```

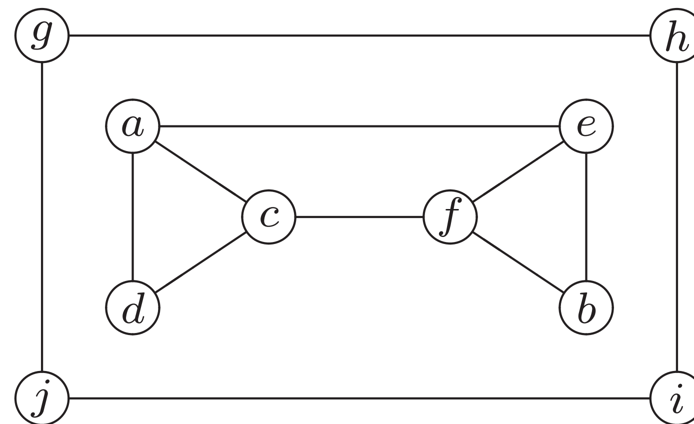
Pseudo-code DFS (vervolg)

```
dfs (v)
    // Bezoekt recursief alle nog onbezochte knopen die via een pad
    // met v zijn verbonden, en nummert deze in de volgorde waarin
    // ze worden ontdekt, met globale variabele ‘teller’

{
    teller ++;
    mark[v] = teller;
    for  elke buurknoop w van v  do
        if  mark[w] == 0  then
            dfs (w);
        fi
    od
}
```

Er is ook niet-recursieve implementatie, met expliciete stapel

Complexiteit Depth-First Search



Met adjacency matrix

Met adjacency list

Breadth-first-search

- De wandeling begint in een gegeven knoop v van de graaf.

- Vanuit een zojuist bezochte knoop worden eerst alle aangrenzende -nog onbezochte- knopen bezocht, dan de daaraan grenzende knopen (voor zover nog niet eerder bezocht), en zo verder totdat alle bereikbare knopen bezocht zijn.

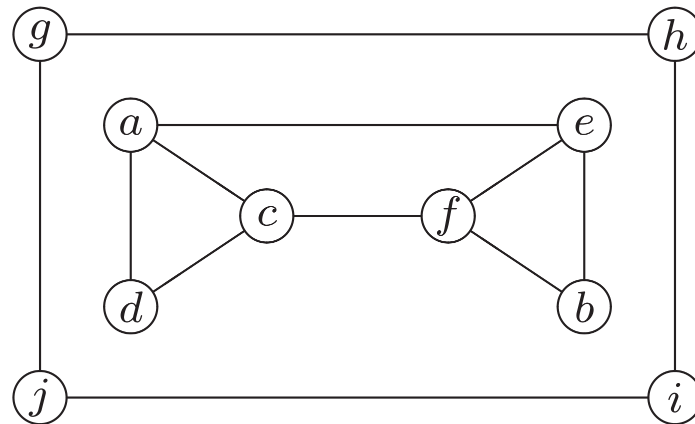
- Knopen worden zo bezocht in volgorde van hun afstand vanaf v.

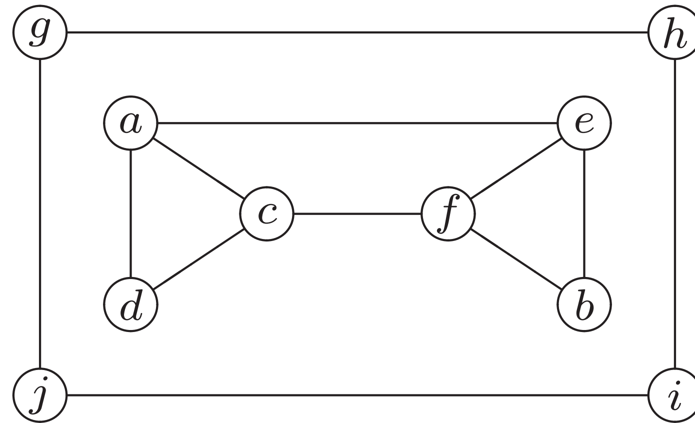
- Aangrenzende knopen kunnen bijvoorbeeld altijd in alfabetische volgorde bezocht worden.

- Bij de implementatie gebruiken we een rij. In de eerste stap wordt v gemarkeerd als bezocht en in de rij gezet. In elke volgende stap wordt de voorste knoop uit de rij gehaald, en worden diens buren gemarkeerd als bezocht en in de rij geplaatst.

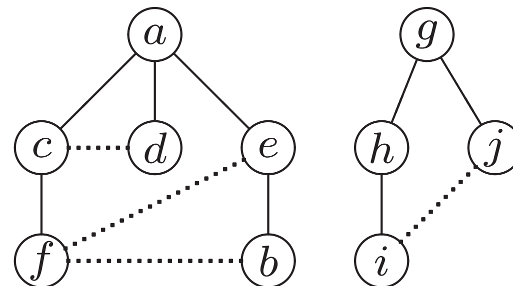
- Alle knopen die vanuit v bereikbaar zijn worden zo precies één keer bezocht. Voor niet-samenhangende grafen moet bovenstaande telkens herhaald worden vanuit een resterende, nog niet bezochte knoop.

Breadth-First Search





a_1 c_2 d_3 e_4 f_5 b_6
g_7 h_8 j_9 i_{10}



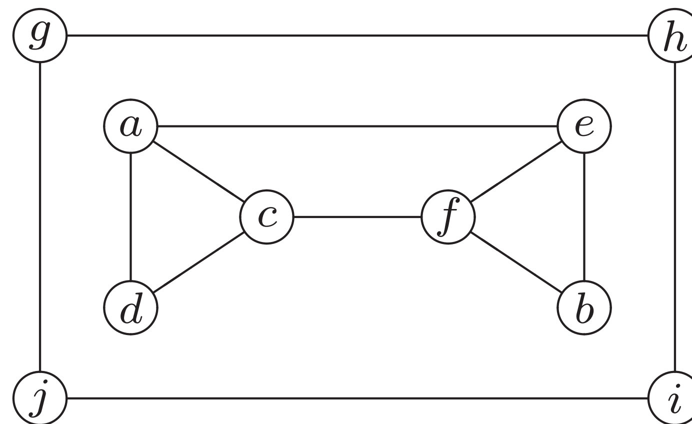
```
ALGORITME BFS (G)
    // Implementeert BFS wandeling door gegeven graaf
    // Invoer: Graaf G = (V,E)
    // Uitvoer: Graaf G met zijn knopen genummerd in de volgorde
    //          waarin ze bij BFS wandeling worden bezocht

{
    for  elke knoop v in V  do
        mark[v] = 0;   // nog niet bezocht
    od
    teller = 0;
    for  elke knoop v in V  do
        if  mark[v] == 0  then
            bfs (v);
        fi
    od
}
```

Pseudo-code BFS (vervolg)

```
bfs (v)
    // Bezoekt alle nog onbezochte knopen die via een pad
    // met v zijn verbonden, en nummert deze in de volgorde waarin
    // ze worden bezocht, met globale variabele ‘teller’
{
    teller ++;
    mark[v] = teller;
    initialiseer queue met v erin;
    while  queue is niet leeg  do
        for  elke buurknoop w van voorste-knoop-in-queue  do
            if  mark[w] == 0  then
                teller ++;
                mark[w] = teller;
                voeg w toe aan queue;  // achteraan
            fi
        od
        verwijder voorste knoop uit queue;
    od
}
```

Complexiteit Breadth-First Search



Met adjacency matrix

Met adjacency list

DFS vs BFS

	DFS	**BFS**
Data structuur	een stapel	een queue
Aantal volgordes knopen	twee volgordes	één volgorde
Soorten takken (onger. grf)	tree en back edges	tree en cross edges
Toepassingen	samenhang,	samenhang
	acycliciteit,	acycliciteit
	‘articulation points’	minimum-tak pad
Complexiteit voor adj. matrix	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert^2)$
Complexiteit voor adj. list	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$

Backtracking

deel 1

Bij veel problemen gaat het erom een element met een speciale eigenschap te vinden binnen een ruimte die exponentieel groeit als functie van de invoergrootte. Dan wordt meestal backtracking gebruikt als goed alternatief voor ES.

Exhaustive search genereert alle kandidaatoplossingen en haalt daar het speciale element tussenuit.

Backtracking

- bouwt kandidaatoplossingen component voor component op,
- kijkt al tijdens de constructie of de deeloplossing nog tot een oplossing kan leiden en
- zo niet, breidt dan de deeloplossing niet verder uit

Op deze manier spaar je soms veel werk uit en kun je dus grotere probleeminstanties oplossen.

Backtracking versus exhaustive search

Exhaustive search bekijkt *alle* volledige kandidaatoplossingen.
Backtracking controleert telkens van deeloplossingen of ze nog aan de eisen/restricties voldoen; zo niet, dan weet je zeker dat alle uitbreidingen van deze oplossing ook niet voldoen, dus die hoef je dan niet meer expliciet te bekijken.

Voorbeeld
Gegeven de rij $A =$ `3, 1, 4, 1, 5, 9, 2, 6, 5, 7`.
Gevraagd: de/een langste *stijgende* deelrij (met volgorde der elementen als in A zelf).

Exhaustive search. Genereer alle 2^{10} deelrijtjes van A en controleer van elk daarvan of hij stijgend is en bepaal wat de langste deelrij is.

Backtracking. Bouw de deelrijtjes stap voor stap op (hoe?) en controleer na elke stap of het deelrijtje nog wel stijgend is. Zo niet, dan hoeft het rijtje niet meer uitgebreid te worden (het kan toch niets worden). Houd de lengte van het deelrijtje bij en vergelijk die met de lengte van de tot dusver gevonden langste deelrij.

Deze methode kan erg veel werk uitsparen. Bijvoorbeeld het deelrijtje `3, 1` is al niet stijgend, dus alle 2^8 deelrijtjes van A die met `3, 1` beginnen zeker ook niet. Deze hoeven bij backtracking dus niet allemaal te worden gegenereerd.

Basisidee backtracking

- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties (en nog wel tot een oplossing kan leiden)

- op elk moment kun je kiezen uit een aantal mogelijke vervolgstappen; maak een keuze en ga langs die weg verder met het opbouwen van de oplossing

- als een keuze op niets uitloopt, herzie je deze keuze en probeer je een andere mogelijkheid

Vergelijk

- het vinden van de uitgang in een doolhof: loop steeds verder en als je bij het zoeken vastloopt, ga terug op je pad om het laatste open alternatief te proberen

- **Lezen/leren bij dit college:**
  Paragraaf 3.4, 3.5, sheets, 12 inl., 12.1

- **Werkcollege** brute force en backtracking

  donderdag 3 maart 2016, 13:45–15:30, in zaal Benoordenhout

- **Opgaven:**

  zie `http://www.liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

- **Volgend college:**

  donderdag 17 maart 2016