# Computationale Intelligentie

Dirk Thierens

# Een voorbeeld: de 8-puzzel

Gegeven zijn een 3x3-raam en 8 genummerde tegels:

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

**begintoestand**

$\longrightarrow$

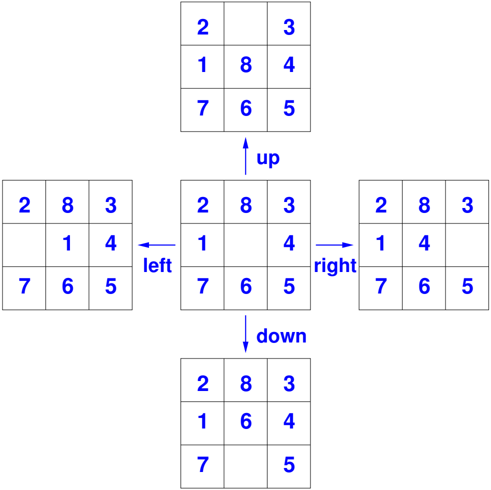
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

**doeltoestand**

- een toestand is een specificatie van de posities van de 8 tegels in het raam;
- het horizontaal of vertikaal verschuiven van een tegel transformeert de ene toestand in de andere;
- het probleem is het vinden van een reeks van toestanden, en bijbehorende operatoren, waarmee de begintoestand in de doeltoestand wordt getransformeerd.

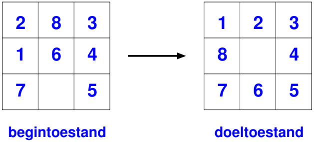
# De operatoren

Voor de 8-puzzel zijn vier operatoren gedefinieerd:

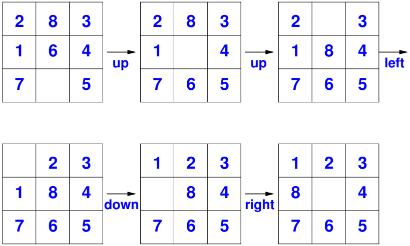


# Een oplossing van de 8-puzzel

Beschouw nogmaals de 8-puzzel met de volgende begin- en doel-toestand:




Een oplossing van de puzzel is een reeks van toestanden, en bijbehorende operatoren, waarmee de begintoestand in de doeltoestand wordt gebracht:



# Een zoekprobleem

### Definitie

Een (instantie van een )zoekprobleem (met paden als oplossingen) is een tupel $P = (T, B, D, O)$ met

- $T$ is de (eindige) verzameling van toestanden van $P$;
- $B \subseteq T$ is de verzameling van begintoestanden van $P$;
- $D \subseteq T$ is de verzameling van doeltoestanden van $P$;
- $O \subseteq \mathcal{P}(T \times T)$ is de verzameling van operatoren van $P$.

# De toestandsruimte

De verzameling $T$ van alle toestanden van een zoekprobleem $P$ heet de toestandsruimte van $P$.

Voorbeeld

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| | 8 | 4 |
| 7 | 6 | 5 |

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 4 | |
| 7 | 6 | 5 |

| | | |
|---|---|---|
| 2 | | 8 |
| 1 | 4 | 3 |
| 7 | 6 | 5 |

...

De toestandsruimte van de 8-puzzel bevat $9! = 362\,880$ toestanden ...
$\square$

# Paden en oplossingen

## Definitie

Zij $P = (T, B, D, O)$ een zoekprobleem. Dan,

- een pad $t$ in $T$ is een eindige reeks toestanden $t = t_1, \ldots, t_n, n \geq 1$, zodanig dat voor elke $k = 1, \ldots, n-1$ er een $o \in O$ is met $(t_k, t_{k+1}) \in o$;
- de lengte van een pad $t$ in $T$ met $n$ toestanden is gelijk aan $n - 1$;
- een oplossing van $P$ is een pad $t = t_1, \ldots, t_n$ in $T$ met $t_1 \in B$ en $t_n \in D$;
- een oplossing $t$ van $P$ is optimaal als $t$ van alle oplossingen van $P$ minimale lengte heeft.

# De zoekruimte

De zoekruimte van een zoekprobleem $P$ is de verzameling van alle toestanden $t_i \in T$ waarvoor een pad $t_1, \ldots, t_i$ met $t_1 \in B$ bestaat.

Voorbeeld

Beschouw een variant van de 8-puzzel met de operatoren left, right en down, met de volgende begin- en doeltoestand:

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

**begintoestand**

$\longrightarrow$

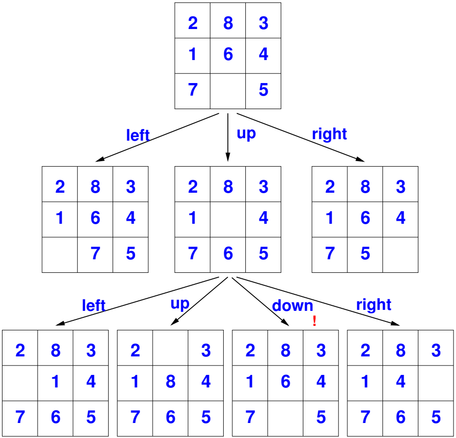
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

**doeltoestand**

De toestandsruimte bevat de doeltoestand, maar de zoekruimte bevat de doeltoestand niet. □

# Een structuur op de zoekruimte

De operatoren van een zoekprobleem definiëren een structuur op de zoekruimte van het probleem:



# De zoekgraaf en de zoekboom

## Definitie

Zij $P = (T, B, D, O)$ een zoekprobleem. De zoekgraaf voor $P$ en de begintoestand $b \in B$ is de gerichte graaf $G_P(b) = (V, A)$ waarin:

- $b \in V$;
- voor elke toestand $t \in V$ geldt dat elke toestand $t'$ waarvoor een $o \in O$ bestaat met $(t, t') \in o$, bevat is in $V$;
- voor alle toestanden $t, t' \in V$, waarvoor een $o \in O$ bestaat met $(t, t') \in o$, geldt dat $t \to t' \in A$.

## Definitie

De zoekboom voor $P$ en $b$ is de boom met wortel $b$, die verkregen is door enumeratie van alle paden in $G_P(b)$ vanuit $b$.

# Zoeken

Het zoeken naar een optimale oplossing van een zoekprobleem $P$ vanuit de begintoestand $b$

$$=$$

het zoeken naar een kortste pad naar een doeltoestand $d$ in de zoekgraaf $G_P(b)$ voor $P$ en $b$.

# Een generiek zoekalgoritme

De essentie van een zoekalgoritme is in pseudocode:

```
procedure search(P,b) returns solution:
  initialise(b,data);
  while not termination-criterion(data) do
    operator ← select-operator(O,data);
    data ← apply(operator,data)
  enddo
endprocedure
```

De procedure `search` wordt aangeroepen met een zoekprobleem $P = (T, B, D, O)$ en een begintoestand $b \in B$:

- de functie `select-operator` selecteert een geschikte operator uit de verzameling $O$;
- de functie `apply` past de geselecteerde operator toe op een toestand uit de data.

# Een overzicht van de onderwerpen

In dit vak zullen de volgende zoekalgoritmen aan bod komen:

- ongeïnformeerd zoeken: breadth-first search, depth-first search, backtracking;
- heuristisch zoeken: best-first search, $A^*$;
- zoeken met kosten: cost-based search, heuristische cost-based search;
- lokaal zoeken: hill climbing, tabu search, simulated annealing;
- zoeken met een tegenstander: minimax, $\alpha$-$\beta$ pruning;
- constraint satisfaction: chronologische backtracking, forward checking, backjumping.

# Vier eigenschappen

Van de algoritmen worden telkens vier eigenschappen bestudeerd:

- volledigheid: vindt het algoritme altijd een pad naar een doeltoestand als de zoekboom van het probleem zo'n pad bevat ?
- optimaliteit: vindt het algoritme altijd een pad naar een doeltoestand van minimale lengte ?
- rekentijd: hoeveel rekentijd gebruikt het algoritme voor het vinden van een pad naar een doeltoestand ?
- geheugenbeslag: hoeveel geheugen gebruikt het algoritme tijdens het zoeken ?

# Computationele Intelligentie

## Ongeïnformeerd zoeken

# Ongeïnformeerd zoeken

- een algoritme voor ongeïnformeerd zoeken doorzoekt de zoekruimte van een probleem op een systematische wijze zonder gebruik te maken van extra kennis van het probleem;
- enkele voorbeelden zijn:
  - (dynamische) breadth-first search;
  - (dynamische) depth-first search;
  - (dynamische) depth-first search met iteratief verdiepen;
  - backtracking.

# Breadth-first search – inleiding

Gegeven is een zoekboom $T$.

- breadth-first search onderzoekt of een gegeven knoop een doeltoestand weergeeft;
- breadth-first search onderzoekt een knoop op diepte $i$ in $T$ pas als alle knopen op diepte $i-1$ zijn onderzocht;
- breadth-first search houdt een front bij van de te onderzoeken knopen in $T$.

Breadth-first search doorzoekt de boom $T$ laag voor laag.

# Breadth-first search

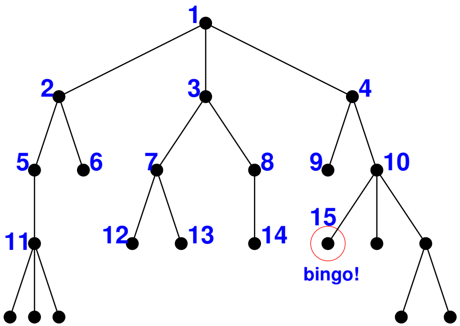
De essentie van breadth-first search is in pseudocode

```
procedure bfs(L) returns t:
  if empty(L) then return nil
  else
    t ← dequeue(L);
    if goal(t) then return t
    else
      L ← enqueue(L,successors(t));
      bfs(L)
endprocedure
```

De procedure `bfs` wordt aangeroepen voor een expliciet gegeven zoekboom $T$ en een queue $L$ met initieel alleen de wortel van $T$, en geeft een toestand (inclusief een pad naar die toestand) terug.

# Een voorbeeld

Veronderstel dat de volgende zoekboom expliciet gegeven is:




Breadth-first search vergelijkt de knopen van de zoekboom met de doeltoestand in de aangegeven volgorde.

# De vier eigenschappen:

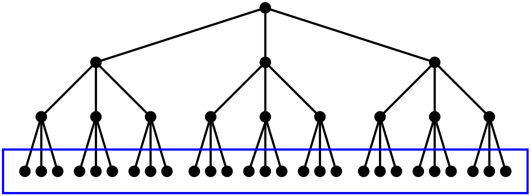
## Volledigheid, optimaliteit, rekentijd, geheugenbeslag.

Als de zoekboom van een zoekprobleem tenminste één doeltoestand bevat, dan geldt:

- breadth-first search vindt altijd een pad naar een doeltoestand;
- breadth-first search vindt zo'n pad naar een doeltoestand met minimale afstand tot de wortel van de boom;
- breadth-first search kost gemiddeld exponentieel veel tijd en ruimte — exponentieel in de diepte van de boom.

# Een ruwe analyse van het geheugenbeslag

Beschouw een zoekboom $T$ met een vertakkingsfactor $b$ en diepte $d$:



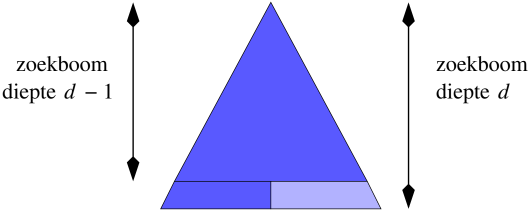
- breadth-first search gebruikt al $b^d$ ruimte voor het opslaan van het front op diepte $d$;
- de benodigde ruimte voor het opslaan van alle knopen is

$$1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

en is dus van de orde $b^d$.

# Een ruwe analyse van de rekentijd

Beschouw een zoekboom met vertakkingsfactor $b$ en diepte $d$, en veronderstel dat de enige doeltoestand zich op diepte $d$ bevindt:




Breadth-first search voert gemiddeld

$$\frac{b^{d+1}-1}{b-1} - \frac{b^d-1}{2} = \frac{b^{d+1}+b^d+b-3}{2\cdot(b-1)}$$

vergelijkingen van een knoop met de doeltoestand uit.

# Een ruwe analyse – vervolg

Met een vertakkingsfactor van 10, een geheugenbeslag van 100 bytes per knoop en een rekentijd van 1000 knopen per seconde, geldt:

| *diepte* | *rekentijd* | *geheugenbeslag* |
|---:|---:|---:|
| 0 | 1 milliseconde | 100 bytes |
| 2 | 0.1 seconde | 11 kilobytes |
| 4 | 11 seconden | 1 megabyte |
| 6 | 18 minuten | 111 megabytes |
| 8 | 31 uren | 11 gigabytes |
| 10 | 128 dagen | 1 terabyte |
| 12 | 35 jaren | 111 terabytes |

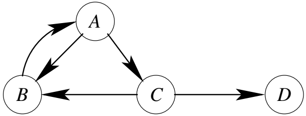
# Dynamische breadth-first search

In een dynamische variant van breadth-first search geldt:

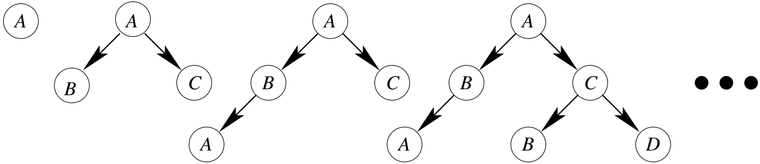
- de zoekboom van een probleem is slechts impliciet gegeven;
- een deel van de zoekboom wordt dynamisch gegenereerd door knopen in een breadth-first volgorde te expanderen;
- een knoop wordt geëxpandeerd door zijn successors te genereren.

# Een voorbeeld

Beschouw een zoekprobleem met de volgende zoekgraaf:



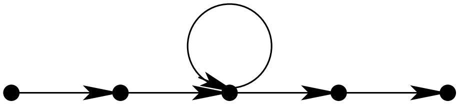

De zoekboom voor het probleem met de begintoestand $A$ wordt met de volgende stappen gegenereerd:



# Cykels

Voor elk dynamisch zoekalgoritme geldt:

- bij het genereren van de zoekboom voor een probleem uit de zoekgraaf kunnen paden met cykels ontstaan:



- een kortste pad naar een doeltoestand bevat nooit een cykel.

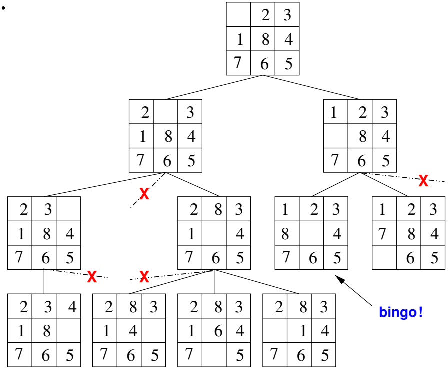
# Herhaalde toestanden

Herhaald voorkomende toestanden kunnen in een dynamische zoekboom in meer of mindere mate voorkomen worden:

- bij het expanderen van een knoop wordt nooit een toestand opgenomen die gelijk is aan de ouder van de knoop;
- bij het expanderen van een knoop wordt nooit een toestand opgenomen die al voorkomt op het pad van de wortel naar de knoop;
- bij het expanderen van een knoop wordt nooit een toestand opgenomen die al eens eerder is gegenereerd.

# Een voorbeeld

Voor de 8-puzzel wordt de volgende zoekboom dynamisch gegenereerd:




Bij het expanderen van een knoop is nooit een toestand opgenomen die gelijk is aan de ouder van de knoop.

# Depth-first search – inleiding

Gegeven is een zoekboom $T$.

- depth-first search onderzoekt of een gegeven knoop een doel-toestand weergeeft;
- depth-first search onderzoekt een knoop op een pad $p$ in $T$ pas als alle knopen op alle paden links van $p$ zijn onderzocht;
- depth-first search houdt een front bij van de te onderzoeken knopen in $T$.

Depth-first search doorzoekt de boom $T$ pad voor pad.

# Depth-first search

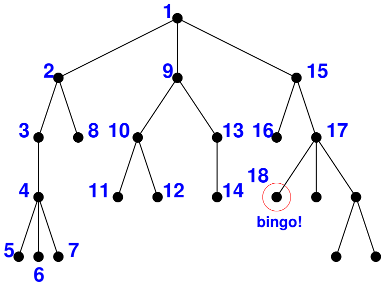
De essentie van depth-first search is in pseudocode

```
procedure dfs(L) returns t:
  if empty(L) then return nil
  else
    t ← pop(L);
    if goal(t) then return t
    else
      L ← push(L,successors(t));
      dfs(L)
endprocedure
```

De procedure `dfs` wordt aangeroepen voor een expliciet gegeven zoekboom $T$ en een stack $L$ met initieel alleen de wortel van $T$, en geeft een toestand (inclusief een pad naar die toestand) terug.

# Een voorbeeld

Veronderstel dat de volgende zoekboom expliciet gegeven is:




Depth-first search vergelijkt de knopen van de zoekboom met de doeltoestand in de aangegeven volgorde.

# De vier eigenschappen:

## Volledigheid, optimaliteit, rekentijd, geheugenbeslag.

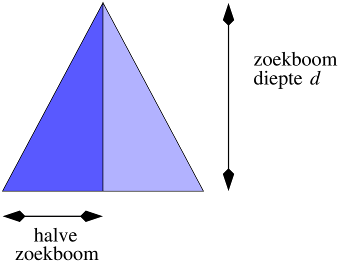
Als de zoekboom van een zoekprobleem eindig is en tenminste één doeltoestand bevat, dan geldt:

- depth-first search vindt altijd een pad naar een doeltoestand;
- de gevonden doeltoestand is niet noodzakelijk een doeltoestand met minimale afstand tot de wortel van de boom;
- depth-first search kost gemiddeld exponentieel veel tijd en ruimte — exponentieel in de diepte van de boom.

Als de zoekboom niet eindig is, dan vindt depth-first search niet noodzakelijk een doeltoestand.

# Een ruwe analyse van de rekentijd

Beschouw een zoekboom met vertakkingsfactor $b$ en diepte $d$, en veronderstel dat de enige doeltoestand zich op diepte $d$ bevindt:




Depth-first search voert gemiddeld

$$\frac{1}{2}\left((d+1)+\left(\frac{b^{d+1}-1}{b-1}\right)\right)=\frac{b^{d+1}+b\cdot d+b-d-2}{2\cdot(b-1)}$$

vergelijkingen van een knoop met de doeltoestand uit.

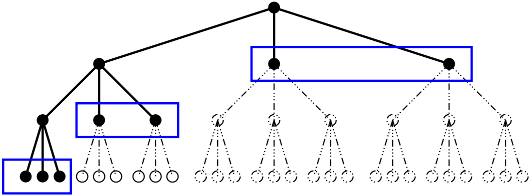
# Dynamische depth-first search

In een dynamische variant van depth-first search geldt:

- de zoekboom van een probleem is slechts impliciet gegeven;
- een deel van de zoekboom wordt dynamisch gegenereerd door knopen in een depth-first volgorde te expanderen;
- een knoop wordt geëxpandeerd door zijn successors te genereren.

# Een ruwe analyse van het geheugenbeslag

Beschouw een zoekboom $T$ met een vertakkingsfactor $b$ en diepte $d$:



Dynamische depth-first search gebruikt maximaal $d \cdot (b-1)+1$ ruimte voor het opslaan van het front.

# Een vergelijking met breadth-first search

Met een vertakkingsfactor van 10 en een geheugenbeslag van 100 bytes per knoop geldt voor de dynamische varianten:

| *diepte* | *geheugenbeslag depth-first* | *geheugenbeslag breadth-first* |
|---:|---:|---:|
| 0 | 100 bytes | 100 bytes |
| 2 | 2 kilobytes | 11 kilobytes |
| 4 | 4 kilobytes | 1 megabyte |
| 6 | 6 kilobytes | 111 megabytes |
| 8 | 8 kilobytes | 11 gigabytes |
| 10 | 10 kilobytes | 1 terabyte |
| 12 | 12 kilobytes | 111 terabytes |

# Een dieptegrens

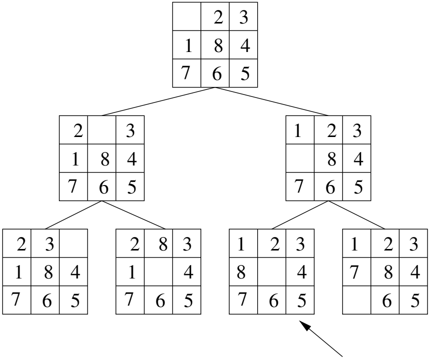
Het basisalgoritme voor depth-first search kan worden uitgebreid met een dieptegrens:

- een dieptegrens specificeert een maximum aan de diepte van te onderzoeken knopen;
- het gebruik van een dieptegrens voorkomt oneindige recursie van depth-first search.

Een concrete dieptegrens wordt meestal gekozen op grond van kennis van het op te lossen probleem.

# Een voorbeeld

Voor de 8-puzzel wordt de volgende zoekboom dynamisch gegenereerd:




De boom is gegenereerd met een dieptegrens van 2; bij het expanderen van een knoop is nooit een toestand opgenomen die gelijk is aan de ouder van de knoop.

# Iteratief verdiepen

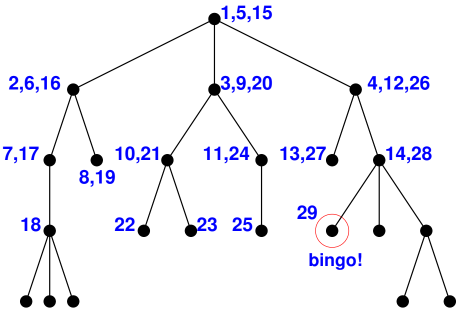
De essentie van iteratief verdiepen is in pseudocode:

```
procedure iterative-deepening(min-depth,step-size) returns t:
  t ← nil;
  all-of-tree ← false;
  depth ← min-depth;
  repeat
    L ← (root(T));
    t, all-of-tree ← dfs(L,depth);
    depth ← depth + step-size;
  until t ≠ nil or all-of-tree;
  return t
endprocedure
```

- de procedure `iterative-deepening` wordt aangeroepen met een minimale dieptegrens en een stapgrootte;
- de functie `dfs` voert een begrensde depth-first search uit op $T$, en retourneert een toestand en of de hele boom is doorlopen.

# Een voorbeeld

Veronderstel dat de volgende zoekboom expliciet gegeven is:




Depth-first search met iteratief verdiepen met minimale dieptegrens 1 en stapgrootte 1 vergelijkt de knopen van de zoekboom met de doeltoestand in de aangegeven volgorde.

# De vier eigenschappen:

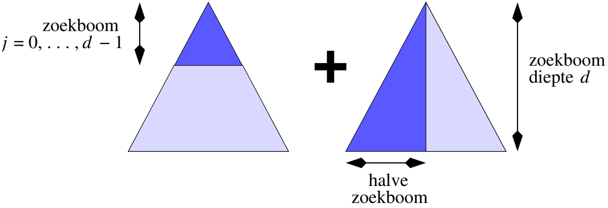
## Volledigheid, optimaliteit, rekentijd, geheugenbeslag.

Als de zoekboom van een zoekprobleem eindig is en tenminste één doeltoestand bevat, dan geldt:

- depth-first search met iteratief verdiepen vindt een doeltoestand;
- depth-first search met een minimale dieptegrens van maximaal 1 en een stapgrootte van 1 vindt een doeltoestand met minimale afstand tot de wortel van de boom;
- depth-first search met iteratief verdiepen kost dan exponentieel veel tijd;
- depth-first search met iteratief verdiepen vergt lineair veel ruimte.

# Een ruwe analyse van de rekentijd

Beschouw een zoekboom met vertakkingsfactor $b$ en diepte $d$, en veronderstel dat de enige doeltoestand in de boom zich op diepte $d$ bevindt:




Depth-first search met iteratief verdiepen voert gemiddeld

$$\left(\sum_{j=0}^{d-1} \frac{b^{j+1}-1}{b-1}\right) + \frac{1}{2}\left(d+1+\frac{b^{d+1}-1}{b-1}\right) =$$

$$= \frac{b^{d+2}+b^{d+1}+b^2d+b^2-4bd-5b+3d+2}{2\cdot(b-1)^2}$$

vergelijkingen van een knoop met de doeltoestand uit.

# Backtracking – inleiding

Backtracking is gerelateerd aan dynamische depth-first search. De overeenkomsten zijn:

- de zoekboom van een zoekprobleem is niet expliciet gegeven;
- een deel van de zoekboom wordt dynamisch gegenereerd door knopen in een depth-first volgorde te expanderen.

De belangrijkste verschillen zijn:

- een knoop wordt meer dan één maal partieel geëxpandeerd;
- een knoop wordt partieel geëxpandeerd door dynamisch eén van zijn successors te genereren.

# Backtracking

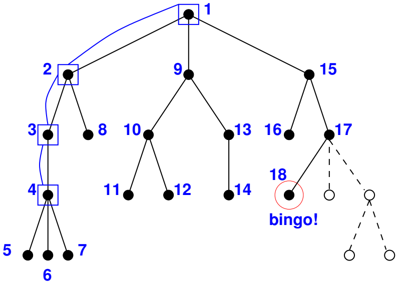
De essentie van backtracking is in pseudocode

```
procedure backtrack(L) returns t:
  if empty(L) then return nil
  else
    t ← first(L);
    if goal(t) then return t
    else
      while there are unexplored successors of t
      and not found do
        t' ← next-successor(t);
        L ← push(L,t');
        backtrack(L)
      endwhile;
      L ← pop(L)
endprocedure
```

De procedure `backtrack` wordt aangeroepen voor een dynamische zoekboom $T$ en een stack $L$ met initieel alleen de wortel van $T$, en retourneert een toestand (inclusief een pad).

# Een voorbeeld

Veronderstel dat de volgende zoekboom gegenereerd wordt:




Backtracking vergelijkt de knopen van de zoekboom met de doeltoestand in dezelfde volgorde als depth-first search.

# De vier eigenschappen:

## Volledigheid, optimaliteit, rekentijd, geheugenbeslag.

Als de zoekboom van een zoekprobleem eindig is en tenminste één doeltoestand bevat, dan geldt:

- backtracking heeft ruwweg dezelfde voor- en nadelen als dynamische depth-first search:
  - volledig;
  - niet optimaal;
- backtracking gebruikt minder ruimte dan dynamische depth-first search;
- backtracking gebruikt ook iets minder tijd.