

# Einführung in die Methoden der Künstlichen Intelligenz

## Suchverfahren / Uninformierte Suche

PD Dr. David Sabel

SoSe 2014

Stand der Folien: 29. Mai 2023

## Motivation für Suchverfahren

### Beispiele:

- Spiele: Suche nach dem besten Zug
- Logik: Suche nach einer Herleitung einer Aussage
- Agenten: Suche nach der optimalen nächsten Aktion
- Planen: Suche nach einer Folge von Aktionen eines intelligenten Agenten.
- Optimierung: Suche eines Maximums einer mehrstelligen Funktion auf den reellen Zahlen

## Repräsentation eines Suchproblems

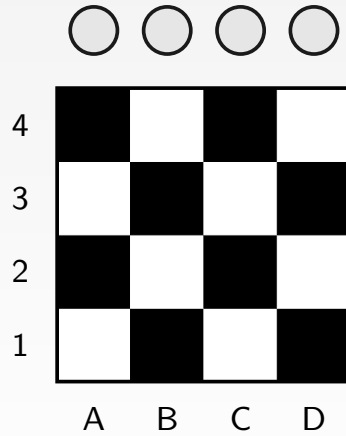
- Anfangssituationen
- Nachfolgerfunktion
- Ein Test auf Zielsituation

Normalerweise nicht gegeben: Der gesamte Suchgraph!

## Beispiele:

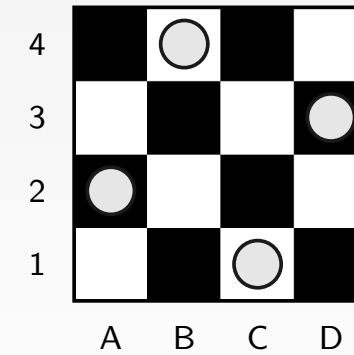
- **Schach**
  - Anfangssituation: Eine Stellung im Spiel
  - Nachfolgerfunktion: Mögliche Züge
  - Zielsituation: Wenn man gewonnen hat
  - Gesucht: Zug der zum Gewinn führt
- **Deduktionssystem**
  - Anfangssituation: Zu beweisende Aussage  $A$ , Menge von Axiomen und bewiesenen Sätzen
  - Nachfolgerfunktion: Deduktionsregeln
  - Zielsituation: Beweis
  - Gesucht: Beweis für  $A$
- **Planen**
  - Anfangssituation: Formale Beschreibung des interessierenden Bereichs z.B. Fahrplan, Start- und Zielort,
  - Nachfolgerfunktion: Zugverbindungen usw.
  - Gesucht: Plan, der Reise ermöglicht

## Beispiel: $n$ Damen



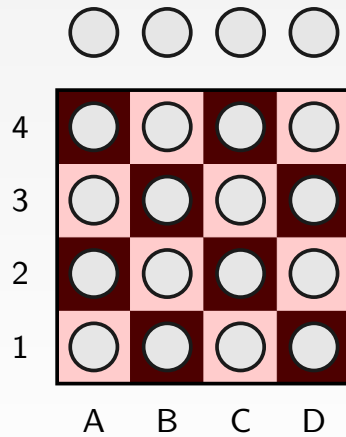
Platziere auf  $n \times n$  Schachbrett  $n$  Damen,  
so dass keine die andere bedroht

## Beispiel: $n$ Damen, mögliche Lösung



Platziere auf  $n \times n$  Schachbrett  $n$  Damen,  
so dass keine die andere bedroht

## Beispiel: $n$ Damen, mögliche Suche



Platziere die Damen zeilenweise nacheinander,  
und backtracke bei Konflikt.

## Beispiel: Missionare und Kannibalen

- 3 Missionare und 3 Kannibalen sind auf einer Seite eines Flusses.
- 1 Boot mit maximal zwei Plätzen
- Bedingung: Auf keiner Uferseite Kannibalen in der Überzahl
- Gesucht: Plan zur Überquerung des Flusses

## Eine Kodierung des Problems

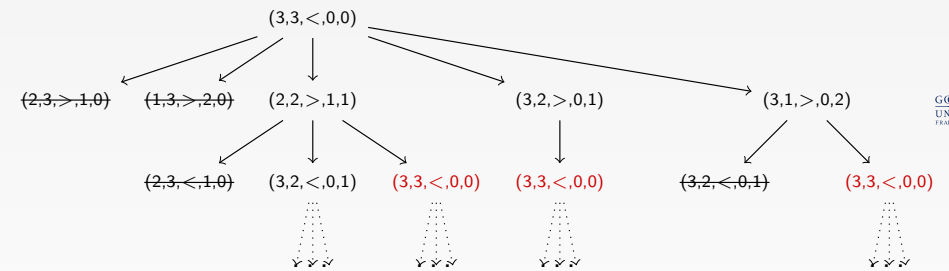
- **Zustand:** (#M, #K, Boot (< oder >), #M, #K)
- **Start:** (3M, 3K, <, 0M, 0K)
- **Ziel:** (0M, 0K, >, 3M, 3K)

Nachfolger des Startzustands:

- (2M, 3K, >, 1M, 0K) **verboten**
- (1M, 3K, >, 2M, 0K) **verboten**
- (2M, 2K, >, 1M, 1K)
- (3M, 2K, >, 0M, 1K)
- (3M, 1K, >, 0M, 2K)



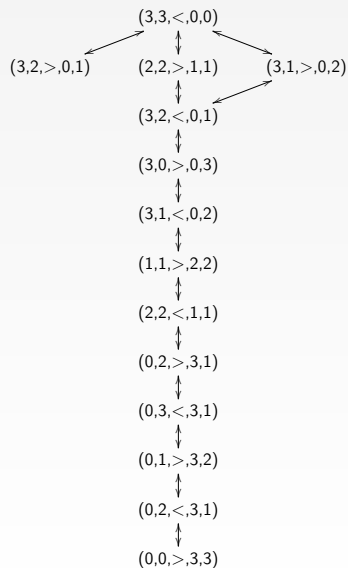
## Suchbaum (Ausschnitt)



Ineffizienzen:

- Doppelte Knoten
  - Schon versuchte Situationen nicht noch einmal
- ⇒ Verwende gerichteten Graphen statt Baum

## Suchgraph



## Modellierung des Problems ist wichtig!

- Nicht Berücksichtigung wichtiger Tatsachen:  
Kannibalen: Ein- und Aussteigen aus dem Boot ist nicht berücksichtigt!
- Sollte man die Zustände während des Bootfahrens berücksichtigen?
- Andere Repräsentation bestimmt die Struktur des Suchraumes.  
Z.B. Modellierung: Missionare und Kannibalen haben eine Identität  
Zustand: Menge der Identitäten auf der Startseite und  $B$  falls das Boot auf der Startseite  
Start:  $\{M_1, M_2, M_3, K_1, K_2, K_3, B\}$   
Ziel:  $\emptyset$  Ist diese Repräsentation besser / schlechter?

# Modellierung (2)

Start  $\{M_1, M_2, M_3, K_1, K_2, K_3, B\}$  hat **21(!) Nachfolger**

- |                                  |                              |
|----------------------------------|------------------------------|
| 1. $\{M_2, M_3, K_1, K_2, K_3\}$ | 12. $\{M_1, K_1, K_2, K_3\}$ |
| 2. $\{M_1, M_3, K_1, K_2, K_3\}$ | 13. $\{M_1, M_3, K_2, K_3\}$ |
| 3. $\{M_1, M_2, K_1, K_2, K_3\}$ | 14. $\{M_1, M_3, K_1, K_3\}$ |
| 4. $\{M_1, M_2, M_3, K_2, K_3\}$ | 15. $\{M_1, M_3, K_1, K_2\}$ |
| 5. $\{M_1, M_2, M_3, K_1, K_3\}$ | 16. $\{M_1, M_2, K_2, K_3\}$ |
| 6. $\{M_1, M_2, M_3, K_1, K_2\}$ | 17. $\{M_1, M_2, K_1, K_3\}$ |
| 7. $\{M_3, K_1, K_2, K_3\}$      | 18. $\{M_1, M_2, K_1, K_2\}$ |
| 8. $\{M_2, K_1, K_2, K_3\}$      | 19. $\{M_1, M_2, M_3, K_3\}$ |
| 9. $\{M_2, M_3, K_2, K_3\}$      | 20. $\{M_1, M_2, M_3, K_2\}$ |
| 10. $\{M_2, M_3, K_1, K_3\}$     | 21. $\{M_1, M_2, M_3, K_1\}$ |
| 11. $\{M_2, M_3, K_1, K_2\}$     |                              |

Davon sind 12 erlaubt

⇒ In dieser Repräsentation ist die Suche sehr viel schwieriger!



# Suchraum, Suchgraph

**Suche nach Lösung** = Suche in einem gerichteten Graphen

**Aber:** Der Graph ist **nicht explizit** gegeben!

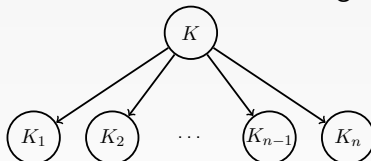
**Suchgraph (Suchraum) gegeben durch:**

- **Knoten:** Situation, Zustände
- **Kanten:** implizit als **Nachfolger-Funktion  $N$**
- **Anfangssituation**
- **Zieltest:** Entscheidbarer Test, ob Knoten Zielknoten



# Eigenschaften

- **Verzweigungsrate des Knotens  $K$**  (branching factor): Anzahl der direkten Nachfolger von  $K$ , also  $|N(K)|$ .



$$N(K) = \{K_1, \dots, K_n\}$$

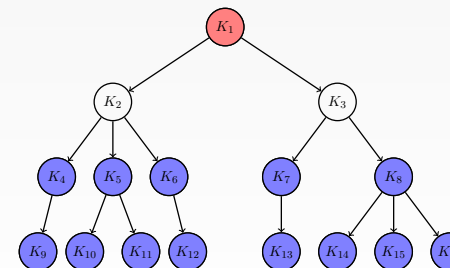
$$|N(K)| = n$$

- **Mittlere Verzweigungsrate des Suchraumes:** Durchschnittliche Verzweigungsrate aller Knoten.

# Eigenschaften(2)

- **Größe des Suchraumes ab Knoten  $K$  in Tiefe  $d$ :** Anzahl Knoten, die von  $K$  aus **in  $d$  Schritten** erreichbar sind  
D.h.  $|\overline{N}^d(K)|$ , wobei

$$\overline{N}^1(M) = \bigcup \{N(L) \mid L \in M\} \text{ und } \overline{N}^i(K) = \overline{N}(\overline{N}^{i-1}(K)).$$



- Größe des Suchraums ab  $K_1$  in Tiefe 2 = 5
- Größe des Suchraums ab  $K_1$  in Tiefe 3 = 8

## Eigenschaften (3)

Eine Suchstrategie ist **vollständig**, wenn sie einen Zielknoten nach endlichen vielen Schritten findet, falls dieser existiert.



## Kombinatorische Explosion

- Üblicherweise: mittlere Verzweigungsrate  $> 1$
- $\Rightarrow$  Suche ist exponentiell in der Tiefe des Suchraums
- das nennt man: **kombinatorische Explosion**
- Die meisten Suchprobleme sind NP-hart (NP-vollständig)



## Blind Search

- Blind Search = Nicht-informierte Suche
- Nur der Suchgraph ist (implizit) gegeben
- keine anderen Informationen (z.B. Heuristik)

Eingabe:

- Menge der initialen Knoten
- Menge der Zielknoten, bzw. eindeutige Festlegung der Eigenschaften der Zielknoten
- Nachfolgerfunktion  $N$

Ausgabe: Pfad zum Zielknoten (falls dieser existiert)

## Nicht-informierte Suche, allg.

### Algorithmus Nicht-informierte Suche

**Datenstrukturen:**  $L$  = Menge von Knoten, markiert Weg dorthin

**Eingabe:** Setze  $L :=$  Menge der initialen Knoten mit leerem Weg

**Algorithmus:**

- 1 Wenn  $L$  leer ist, dann breche ab.
- 2 Wähle einen beliebigen Knoten  $K$  aus  $L$ .
- 3 Wenn  $K$  ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
- 4 Wenn  $K$  kein Zielknoten, dann nehme Menge  $N(K)$  der direkten Nachfolger von  $K$  und verändere  $L$  folgendermaßen:  
 $L := (L \cup N(K)) \setminus \{K\}$  (Wege entsprechend anpassen)  
Mache weiter mit Schritt ??

**Keine Strategie!, nichtdeterministisch**

# Tiefensuche

## Algorithmus Tiefensuche

**Datenstrukturen:**  $L =$  Liste (Stack) von Knoten, markiert Weg dorthin

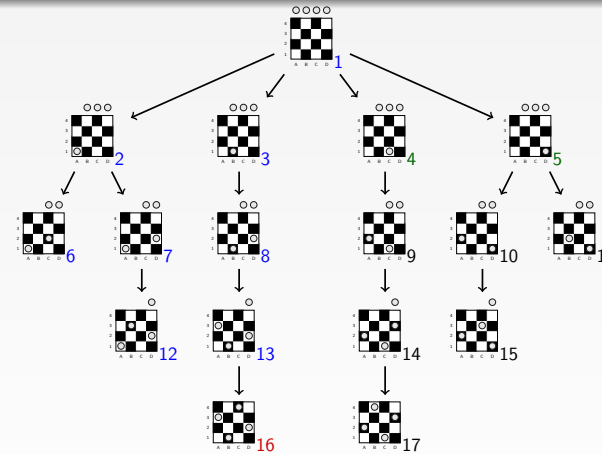
**Eingabe:** Füge die initialen Knoten in die Liste  $L$  ein.

### Algorithmus:

- 1 Wenn  $L$  die leere Liste ist, dann breche ab.
- 2 Wähle ersten Knoten  $K$  aus  $L$ , sei  $R$  die Restliste.
- 3 Wenn  $K$  ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
- 4 Wenn  $K$  kein Zielknoten, dann sei  $N(K)$  die (geordnete) Liste der direkten Nachfolger von  $K$ , mit dem Weg dorthin markiert  
 $L := N(K) ++ R$ . (wobei  $++$  Listen zusammenhängt)  
 Mache weiter mit ??.



# Beispiel Tiefensuche



Am Anfang:

$L := [(1, [])]$  1. Knoten

$K := (1, [])$   $R := []$

$NF(K) = [2, 3, 4, 5]$

$L := [(2, [1]), (3, [1]), (4, [1]), (5, [1])] ++ R = [(2, [1]), (3, [1]), (4, [1]), (5, [1])]$  2.

$K := (2, [1])$   $R := [(3, [1]), (4, [1]), (5, [1])]$

$NF(2) = [6, 7]$

$L := [(6, [1, 2]), (7, [1, 2]), (3, [1]), (4, [1]), (5, [1])]$

# Tiefensuche in Haskell

```
dfs :: (a -> Bool)    -- Zieltest (goal)
    -> (a -> [a])    -- Nachfolgerfunktion (succ)
    -> [a]           -- Startknoten
    -> Maybe (a, [a]) -- Ergebnis: Just (Zielknoten,Pfad)
                        -- oder Nothing

dfs goal succ stack =
  -- Einfuegen der Anfangspfade, dann mit iterieren mit go
  go [(k, [k]) | k <- stack]
  where
    go [] = Nothing -- Alles abgesucht, nichts gefunden
    go ((k,p):r)
      | goal k = Just (k,p) -- Ziel gefunden
      | otherwise = go ([(k',k':p) | k' <- succ k] ++ r)
```

# Eigenschaften der Tiefensuche

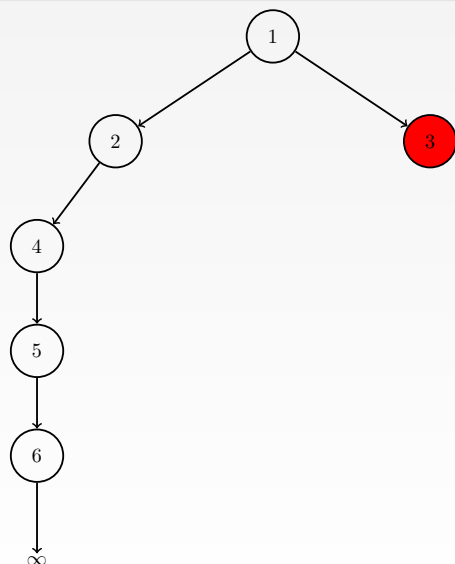
**Komplexität** (worst-case) bei fester Verzweigungsrate  $c > 1$ :

- **Platz:** linear in der Tiefe
- **Zeit:** exponentiell in der Tiefe des Zielknotens

**Vollständigkeit:**

- Nicht vollständig, wenn der Suchgraph unendlich groß ist

## Unvollständigkeit der Tiefensuche



Zielknoten 3 wird  
nie besucht!

## Varianten der Tiefensuche

### Tiefensuche mit Tiefenbeschränkung $k$

- Wenn Tiefe  $k$  überschritten, setze  $NF(K) = \emptyset$
- Findet Zielknoten, die maximal in Tiefe  $k$  liegen

In Haskell:

```
dfsBisTiefe goal succ stack maxdepth =  
  -- wir speichern die Tiefe mit in den Knoten auf dem Stack:  
  go [(k,maxdepth,[k]) | k <- stack]  
  where  
    go [] = Nothing -- Alles abgesucht, nichts gefunden  
    go ((k,i,p):r)  
      | goal k = Just (k,p) -- Ziel gefunden  
      | i > 0 = go ([(k',i-1,k':p) | k' <- succ k] ++ r)  
      | otherwise = go r -- Tiefenschranke erreicht
```

## Varianten der Tiefensuche (2)

### Tiefensuche mit Sharing

- Merke bereits besuchte Knoten, um Knoten nicht doppelt zu besuchen (bei zyklischen Suchgraphen!)
- Speichern der besuchten Knoten: Hashtabelle
- Platz: Anzahl der besuchten Knoten (wegen Speicher für schon untersuchte Knoten)
- Zeit:  $n * \log(n)$  mit  $n =$  Anzahl der untersuchten Knoten.
- Pragmatische Verbesserung: Nur maximal  $l$  viele Knoten speichern (damit der Platz beschränkt ist)

## Tiefensuche mit Sharing

```
dfsSharing goal succ stack =  
  -- Einfuegen der Anfangspfade, dann mit iterieren mit go,  
  -- letztes Argument ist die Merklste  
  go [(k,[k]) | k <- stack] []  
  where  
    go [] mem = Nothing -- Alles abgesucht, nichts gefunden  
    go ((k,p):r) mem  
      | goal k = Just (k,p) -- Ziel gefunden  
      | k 'elem' mem = go r mem -- Knoten schon besucht  
      | otherwise = go ([(k',k':p) | k' <- succ k] ++ r) (k:mem)
```

# Varianten beim Backtracking

## Vorgestelltes Verfahren

- Chronologisches Backtracking

## Varianten

- Dynamic Backtracking
- Dependency-directed Backtracking

Abkürzungen, wenn sichergestellt ist, dass kein Zielknoten übersehen wird.



# Breitensuche

## Algorithmus Breitensuche

**Datenstrukturen:**  $L$  = Menge von Knoten markiert mit Weg

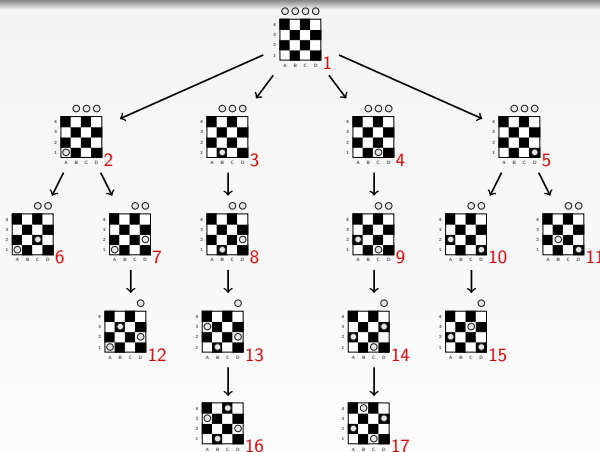
**Eingabe:** Füge die initialen Knoten in die Menge  $L$  ein.

## Algorithmus:

- 1 Wenn  $L$  leer ist, dann breche ab.
- 2 Wenn  $L$  einen Zielknoten  $K$  enthält, dann gebe aus:  $K$  und Weg dorthin.
- 3 Sonst, sei  $N(L)$  Menge aller direkten Nachfolger der Knoten von  $L$ , mit einem Weg dorthin markiert.  
Mache weiter mit Schritt ?? und  $L := N(L)$ .



# Beispiel Breitensuche



Am Anfang:

$L := \{(1, [])\}$  1. Iteration

$L := \{(1, [])\}$

$NF(L) = \{2, 3, 4, 5\}$

$L := \{(2, [1]), (3, [1]), (4, [1]), (5, [1])\}$  2. Iteration

$NF(L) = NF(2) \cup NF(3) \cup NF(4) \cup NF(5) = \{6, 7, 8, 9, 10, 11\}$

# Breitensuche in Haskell

```

bfs goal succ start =
  go [(k,[k]) | k <- start] -- Pfade erzeugen
  where
    go [] = Nothing -- nichts gefunden
    go rs =
      case filter (goal . fst) rs of -- ein Zielknoten enthalten?
        -- Nein, mache weiter mit allen Nachfolgern
        [] -> go [(k',k':p) | (k,p) <- rs, k' <- succ k]
        -- Ja, dann stoppe:
        (r:rs) -> Just r
  
```



## Eigenschaften

**Komplexität** (worst-case) bei fester Verzweigungsrate  $c > 1$

- **Platz:** Anzahl der Knoten in Tiefe  $d$ , d.h.  $O(c^d)$  = exponentiell in der Tiefe  $d$ !
- **Zeit:**  $\underbrace{\text{Anzahl der Knoten in Tiefe } d}_n + \underbrace{(n \log n)}_{\text{Mengenbildung}} = O(c^d(1 + d * \log c))$

### Vollständigkeit

- Die Breitensuche ist vollständig! (bei endlicher Verzweigungsrate)



## Fazit: Tiefen- und Breitensuche

	Tiefensuche	Breitensuche
Zeit	$O(c^d)$	$O(c^d(1 + d \log c))$
Platz	$O(d)$	$O(c^d)$
Vollständig	nein	ja



## Iteratives Vertiefen

- iterative deepening
- Kompromiss a la „Vollständige Tiefensuche“

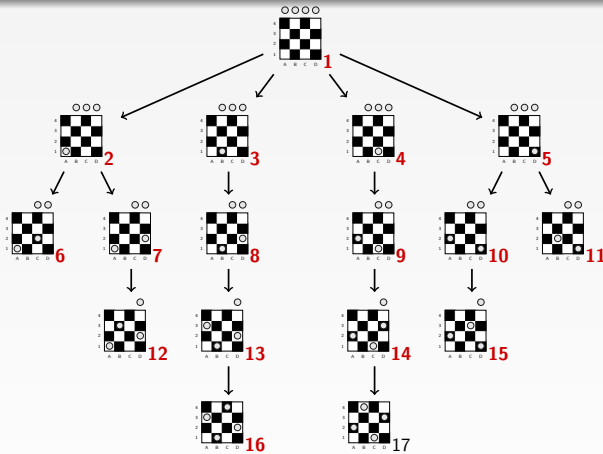
### Pseudo-Algorithmus:

- 1  $k := 0;$
- 2 Tiefensuche mit Tiefenschranke  $k$ ;
- 3 **wenn** Ziel gefunden **dann**  
    breche ab, und gebe Ziel mit Weg aus  
**sonst**
- 4  $k := k + 1;$
- 5 gehe zu ??

## Iteratives Vertiefen in Haskell

```
idfs goal succ stack =
  let -- alle Ergebnisse mit sukzessiver Erhöhung der Tiefenschranke
      alleSuchen = [dfsBisTiefe goal succ stack i | i <- [1..]]
  in
    case filter isJust alleSuchen of
      [] -> Nothing -- Trotzdem nichts gefunden
      (r:rs) -> r -- sonst erstes Ergebnis
```

## Beispiel: Iterative Tiefensuche



Reihenfolge der Knotenbesuche

$k = 1$

1  $k = 2$

1,2,3,4,5  $k = 3$

1,2,6,7,3,8,4,9,5,10,11  $k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11  $k = 5$

1,2,6,7,12,3,8,13,16  $\Rightarrow$  Ziel 41 besuchte Knoten!

## Eigenschaften

**Komplexität** (worst-case) bei mittlerer Verzweigungsrate  $c > 1$

- **Platz:** Linear in der Tiefe
- **Zeit:** ? (gleich)

**Vollständigkeit**

- Die iterative Tiefensuche ist vollständig (bei endlicher Verzweigungsrate)

## Zeitbedarf iteratives Vertiefen (1)

$$\text{Näherung: } \sum_{i=1}^n a^i \approx \frac{a^{n+1}}{a-1}$$

**Tiefensuche mit Tiefenbeschränkung  $k$**

- Alle Knoten besuchen:  $\sum_{i=1}^k c^i$
- Im Mittel besuchte Knoten (Zielknoten in Tiefe  $k$ ):

$$0.5 * \left( \sum_{i=1}^k c^i \right) \approx 0.5 * \left( \frac{c^{k+1}}{c-1} \right)$$

**Iteratives Vertiefen bis Tiefe  $k$** , im Mittel, Zielknoten in Tiefe  $k$ :

$k-1$  Tiefensuchen für Tiefe  $1, \dots, k-1$

+Tiefensuche für Tiefe  $k$  (Hälfte der Knoten)

$$= \sum_{i=1}^{k-1} \frac{c^{i+1}}{c-1} + 0.5 * \left( \frac{c^{k+1}}{c-1} \right)$$

## Zeitbedarf iteratives Vertiefen (2)

$$\text{Näherung: } \sum_{i=1}^n a^i \approx \frac{a^{n+1}}{a-1}$$

$$\begin{aligned} & \sum_{i=1}^{k-1} \frac{c^{i+1}}{c-1} + 0.5 * \left( \frac{c^{k+1}}{c-1} \right) = \frac{\sum_{i=1}^{k-1} c^{i+1}}{c-1} + 0.5 * \left( \frac{c^{k+1}}{c-1} \right) \\ & = \frac{\left( \sum_{i=1}^k c^i \right) - c^k}{c-1} + 0.5 * \left( \frac{c^{k+1}}{c-1} \right) \approx \frac{1}{c-1} \left( \left( \frac{c^{k+1}}{c-1} \right) - c^k \right) + 0.5 * \left( \frac{c^{k+1}}{c-1} \right) \\ & = \left( \frac{c^{k+1}}{(c-1)^2} \right) - \frac{c}{c-1} + 0.5 * \left( \frac{c^{k+1}}{c-1} \right) \approx \left( \frac{c^{k+1}}{(c-1)^2} \right) + 0.5 * \left( \frac{c^{k+1}}{c-1} \right) \end{aligned}$$

## Zeitbedarf iteratives Vertiefen (3)

**Faktor**  $\frac{\text{Iteratives Vertiefen}}{\text{Tiefensuche bis } k}$ :

$$\frac{\frac{c^{k+1}}{(c-1)^2} + 0.5 * \left(\frac{c^{k+1}}{c-1}\right)}{0.5 * \left(\frac{c^{k+1}}{c-1}\right)} = \frac{\frac{c^{k+1}}{(c-1)^2}}{0.5 * \left(\frac{c^{k+1}}{c-1}\right)} + 1 = \frac{2}{c-1} + 1$$

Tabelle der ca.-Werte des Faktors  $d = \frac{2}{c-1} + 1$  ist

$c$	2	3	4	5	...	10
$d$	3	2	1,67	1,5	...	1,22



## Bemerkungen zur iterativen Tiefensuche

Allgemeine Idee dabei:

- Spare Platz, opfere Zeit (speichern vs. neu berechnen)
- Gegensätzlich zum dynamischen Programmieren:  
Dort: Opfere Platz für schnellere Zeit



## Varianten

**Nutze Platz:**

- Speichere Knoten, die bereits expandiert wurden, und betrachte sie nicht neu  
⇒ keine wiederholten Expansionen
- Speichere einen Teil des letzten Suchbaums (z.B. den linken Teil), damit er beim nächsten mal nicht betrachtet werden muss.
- Kombiniere Breitensuche mit Tiefensuche: erst in die Breite, ab dort dann Tiefensuche

## Rückwärtssuche

Idee

- Suche nicht vom Start das Ziel, sondern umgekehrt
- Statt Nachfolgerfunktion benutze Vorgängerfunktion

Lohnenswert?

- Lohnt, wenn die Verzweigungsrate der Vorgängerfunktion kleiner ist, als die der Nachfolgerfunktion
- Problematisch: Finde algorithmische Beschreibung der Vorgängerfunktion

## Bidirektionale Suche

- Suche vorwärts und rückwärts
- Erfordert Vergleich der momentan expandierten Knoten (Schnittmenge)
- Vorteil z.B. bei Breitensuche: Platz: statt  $c^d$  nur  $2 * (c^{d/2})$   
 $\approx$  Doppelt so tief suchen in gleichem Platz (wenn  $\#$ Eingangsknoten  $\approx$   $\#$ Ausgangsknoten)
- Nachteil: Schnittbildung (Zeitintensiv), und Vorgänger- und Nachfolgerfunktion nötig.