

# Grundlagen der Künstlichen Intelligenz

## 9. Klassische Suche: Tiefensuche und iterative Tiefensuche

Malte Helmert

Universität Basel

21. März 2014

# Grundlagen der Künstlichen Intelligenz

21. März 2014 — 9. Klassische Suche: Tiefensuche und iterative Tiefensuche

## 9.1 Tiefensuche

## 9.2 Iterative Tiefensuche

## 9.3 Blinde Suche: Zusammenfassung

# Klassische Suche: Überblick

## Kapitelüberblick klassische Suche:

- ▶ 3.–5. Einführung
- ▶ 6.–9. Basisalgorithmen
  - ▶ 6. Datenstrukturen für Suchalgorithmen
  - ▶ 7. Baumsuche und Graphensuche
  - ▶ 8. Breitensuche und uniforme Kostensuche
  - ▶ 9. **Tiefensuche und iterative Tiefensuche**
- ▶ folgende Kapitel: heuristische Algorithmen

# 9.1 Tiefensuche

## Blinde Suchalgorithmen: Beispiele

### Beispiele für blinde Suchalgorithmen:

- ▶ Breitensuche
- ▶ uniforme Kostensuche
- ▶ **Tiefensuche** (↔ dieses Kapitel)
- ▶ **tiefenbeschränkte Suche** (↔ dieses Kapitel)
- ▶ **iterative Tiefensuche** (↔ dieses Kapitel)

## Tiefensuche

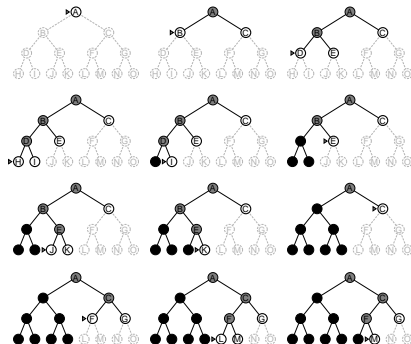
**Tiefensuche** expandiert Knoten **in umgekehrter Erzeugungsreihenfolge** (LIFO).

↔ **tiefster** Knoten zuerst expandiert

↔ z. B. Open-Liste als **Stack** implementiert

## Tiefensuche: Beispiel

**Beispiel:** (Annahme: Knoten in Tiefe 3 haben keine Nachfolger)



## Tiefensuche: einige Eigenschaften

- ▶ fast immer als **Baumsuche** implementiert (wir werden sehen, warum)
- ▶ **nicht vollständig, nicht semi-vollständig, nicht optimal** (Warum?)
- ▶ vollständig für **azyklische** Zustandsräume, z. B. wenn Zustandsraum gerichteter Baum

## Erinnerung: generischer Baumsuchalgorithmus

Erinnerung aus Kapitel 7:

### Generische Baumsuche

```

open := new OpenList
open.insert(make_root_node())
while not open.is_empty():
    n = open.pop()
    if is_goal(n.state):
        return extract_path(n)
    for each  $\langle a, s' \rangle \in \text{succ}(n.state)$ :
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable

```

## Tiefensuche (nicht-rekursive Version)

Tiefensuche (nicht-rekursive Version):

### Tiefensuche (nicht-rekursive Version)

```

open := new Stack
open.push_back(make_root_node())
while not open.is_empty():
    n = open.pop_back()
    if is_goal(n.state):
        return extract_path(n)
    for each  $\langle a, s' \rangle \in \text{succ}(n.state)$ :
        n' := make_node(n, a, s')
        open.push_back(n')
return unsolvable

```

## Nicht-rekursive Tiefensuche: Diskussion

Diskussion:

- ▶ es ist nicht viel falsch mit dem Code (sofern man aufpasst, nicht mehr benötigte Knoten freizugeben, wenn die Programmiersprache keine Garbage-Collection beinhaltet)
- ▶ Tiefensuche als **rekursiver Algorithmus** ist aber einfacher und effizienter
- ↪ Maschinen-Stack als implizite Open-Liste
- ↪ keine Suchknoten-Datenstruktur nötig

## Tiefensuche (rekursive Version)

```

function depth_first_search(s)
if is_goal(s):
    return  $\langle \rangle$ 
for each  $\langle a, s' \rangle \in \text{succ}(s)$ :
    solution := depth_first_search(s')
    if solution  $\neq$  none:
        solution.push_front(a)
    return solution
return none

```

Hauptfunktion:

```

Tiefensuche (rekursive Version)
return depth_first_search(init())

```

## Tiefensuche: Aufwand

### Zeitaufwand:

- ▶ Wenn der Zustandsraum Pfade der Länge  $m$  enthält, kann die Tiefensuche  $O(b^m)$  Knoten erzeugen, selbst wenn sehr kurze Lösungen (z. B. Länge 1) existieren
- ▶ Andererseits: im **besten Fall** können Lösungen der Länge  $\ell$  mit  $O(b\ell)$  erzeugten Knoten gefunden werden. (*Warum?*)
- ▶ verbesserbar auf  $O(\ell)$ , wenn **inkrementelle Nachfolgeberechnung** möglich

### Speicheraufwand:

- ▶ muss nur Knoten **entlang aktuell exploriertem Pfad** speichern ("entlang" = Knoten auf dem Pfad und deren Kinder)
- ↔ Speicheraufwand  $O(bm)$  wenn  $m$  maximale erreichte Suchtiefe
- ▶ dieser niedrige Speicherbedarf ist der Hauptgrund, warum Tiefensuche trotz ihrer Nachteile interessant ist

## 9.2 Iterative Tiefensuche

## Tiefenbeschränkte Suche

### Tiefenbeschränkte Suche:

- ▶ Tiefensuche, die alle Suchknoten in einer gegebenen Tiefe  $n$  **abschneidet** (nicht weiter expandiert)
- ↔ für sich allein nicht sehr nützlich, aber wichtige Zutat in nützlicheren Suchalgorithmen

## Tiefenbeschränkte Suche: Pseudo-Code

```

function depth_limited_search( $s$ ,  $depth\_limit$ ):
  if is_goal( $s$ ):
    return ⟨ ⟩
  if  $depth\_limit > 0$ :
    for each  $\langle a, s' \rangle \in succ(s)$ :
       $solution :=$  depth_limited_search( $s'$ ,  $depth\_limit - 1$ )
      if  $solution \neq none$ :
         $solution.push\_front(a)$ 
      return  $solution$ 
  return none

```

## Iterative Tiefensuche

### Iterative Tiefensuche:

- ▶ **Idee:** führe eine Folge tiefenbeschränkter Suchen mit ansteigenden Tiefenschranken aus
- ▶ klingt verschwenderisch (jede Iteration wiederholt die gesamte vorher geleistete Arbeit), aber tatsächlich ist der Aufwand vertretbar ( $\rightsquigarrow$  Analyse folgt)

### Iterative Tiefensuche

```

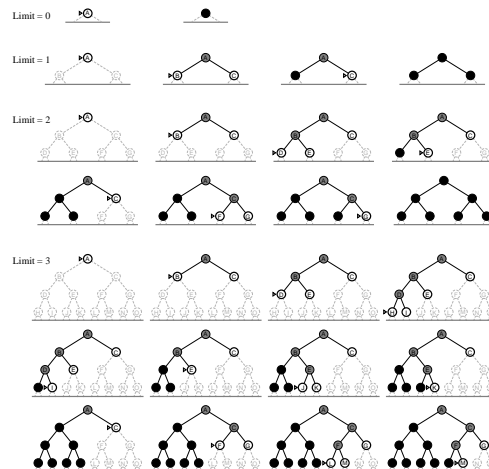
for depth_limit ∈ {0, 1, 2, ...}:
    solution := depth_limited_search(init(), depth_limit)
    if solution ≠ none:
        return solution
  
```

## Iterative Tiefensuche: Eigenschaften

### Kombiniert Vorteile von Breiten- und Tiefensuche:

- ▶ (fast) wie BFS: **semi-vollständig** (allerdings nicht vollständig)
- ▶ wie BFS: **optimal** wenn alle Aktionen dieselben Kosten haben
- ▶ wie DFS: muss nur Knoten entlang eines Pfades speichern  
 $\rightsquigarrow$  Speicheraufwand  $O(bd)$ , wobei  $d$  minimale Lösungslänge
- ▶ Zeitaufwand kaum höher als BFS ( $\rightsquigarrow$  siehe Analyse später)

## Iterative Tiefensuche: Beispiel



## Iterative Tiefensuche: Beispiel für den Aufwand

### Zeitaufwand (erzeugte Knoten):

Breitensuche	$1 + b + b^2 + \dots + b^{d-1} + b^d$
Iterative Tiefensuche	$(d + 1) + db + (d - 1)b^2 + \dots + 2b^{d-1} + 1b^d$

Beispiel:  $b = 10$ ,  $d = 5$

Breitensuche	$1 + 10 + 100 + 1000 + 10000 + 100000$ $= 111111$
Iterative Tiefensuche	$6 + 50 + 400 + 3000 + 20000 + 100000$ $= 123456$

für  $b = 11$ , nur 11% mehr Knoten als mit Breitensuche

## Iterative Tiefensuche: Zeitaufwand

### Satz (Zeitaufwand der iterativen Tiefensuche)

Sei  $b$  der maximale Verzweigungsgrad und  $d$  die minimale Lösungslänge des betrachteten Zustandsraums. Gelte  $b \geq 2$ .

Dann beträgt der **Zeitaufwand** der iterativen Tiefensuche

$$(d + 1) + db + (d - 1)b^2 + (d - 2)b^3 + \dots + 1b^d = O(b^d)$$

und der **Speicheraufwand** beträgt

$$O(bd)$$

## Iterative Tiefensuche: Bewertung

### Iterative Tiefensuche: Bewertung

↔ Iterative Tiefensuche ist oft die Methode der Wahl, wenn

- ▶ **Baumsuche angemessen** (keine Duplikateliminierung nötig) ist
- ▶ und die **Lösungstiefe unbekannt** ist.

## 9.3 Blinde Suche: Zusammenfassung

## Vergleich blinder Suchalgorithmen

### Vollständigkeit, Optimalität, Zeit- und Speicheraufwand

Kriterium	Breiten- suche	uniforme Kostensuche	Tiefen- suche	tiefen- beschr. S.	iterative Tiefensuche
vollständig?	ja*	ja	nein	nein	semi
optimal?	ja**	ja	nein	nein	ja**
Zeit	$O(b^d)$	$O(b^{1+\lceil c^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Speicher	$O(b^d)$	$O(b^{1+\lceil c^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$

$b \geq 2$  Verzweigungsgrad  
 $d$  min. Lösungstiefe  
 $m$  max. Suchtiefe  
 $\ell$  Tiefenschranke  
 $c^*$  optimale Lösungskosten  
 $\epsilon > 0$  min. Aktionskosten

#### Anmerkungen:

\* für BFS-Tree: semi-vollständig  
 \*\* nur mit uniformen Aktionskosten