Grundlagen der Künstlichen Intelligenz

12. Klassische Suche: Tiefensuche und iterative Tiefensuche

Malte Helmert

Universität Basel

16. März 2015

Klassische Suche: Überblick

Kapitelüberblick klassische Suche:

- 5.–7. Grundlagen
- 8.–12. Basisalgorithmen
  - 8. Datenstrukturen für Suchalgorithmen
  - 9. Baumsuche und Graphensuche
  - 10. Breitensuche
  - 11. uniforme Kostensuche
  - 12. Tiefensuche und iterative Tiefensuche
- 13.–20. heuristische Algorithmen

Tiefensuche

Tiefensuche

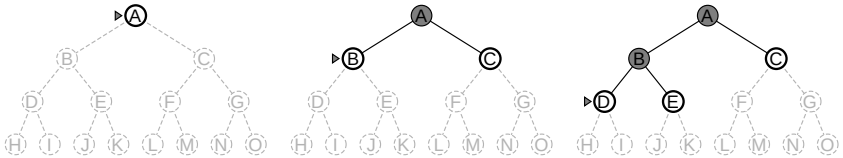
Tiefensuche (depth-first search, DFS) expandiert Knoten in umgekehrter Erzeugungsreihenfolge (LIFO).

$\rightsquigarrow$ tiefster Knoten zuerst expandiert

$\rightsquigarrow$ z. B. Open-Liste als Stack implementiert

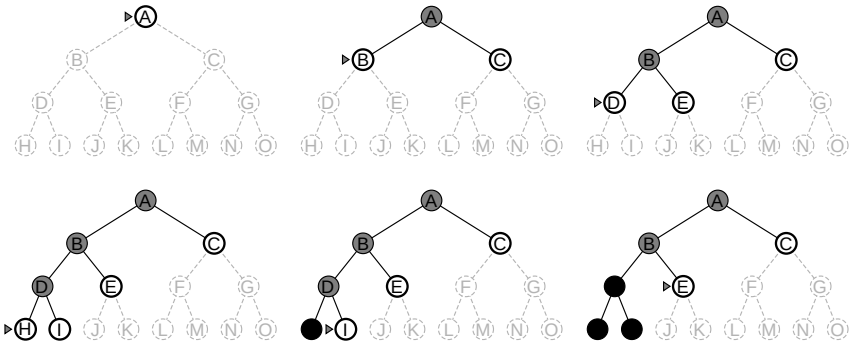
Tiefensuche: Beispiel

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



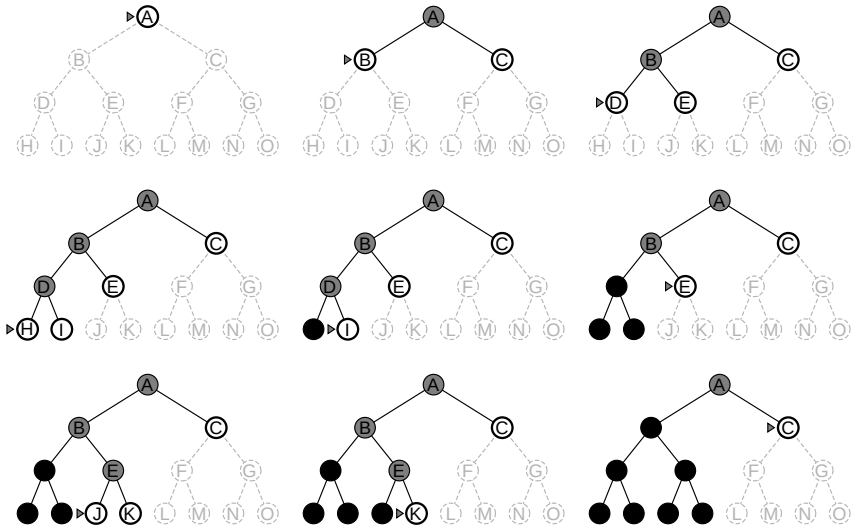
Tiefensuche: Beispiel

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



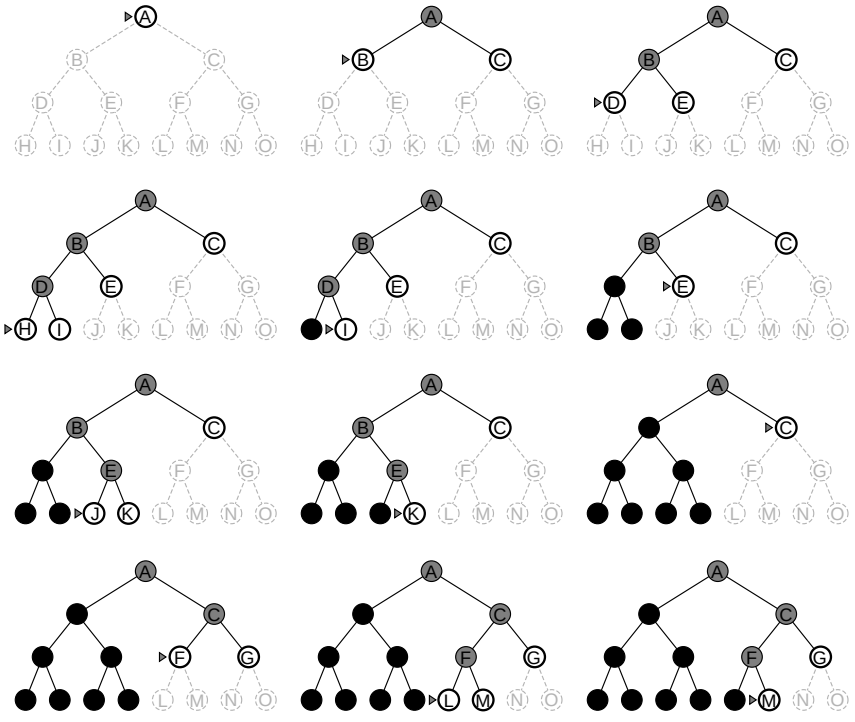
Tiefensuche: Beispiel

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



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 9:

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 ⟨a, s'⟩ ∈ 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 ⟨a, s'⟩ ∈ 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

$\rightsquigarrow$ Maschinen-Stack als implizite Open-Liste

$\rightsquigarrow$ keine Suchknoten-Datenstruktur nötig

Tiefensuche (rekursive Version)

function depth_first_search(s)

```
if is_goal(s):
    return ⟨⟩
for each ⟨a, s'⟩ ∈ succ(s):
    solution := depth_first_search(s')
    if solution ≠ 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 ℓ mit $O(b\ell)$ erzeugten Knoten gefunden werden. (Warum?)
- verbesserbar auf $O(\ell)$, wenn inkrementelle Nachfolgerberechnung möglich

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 ℓ mit $O(b\ell)$ erzeugten Knoten gefunden werden. (Warum?)
- verbesserbar auf $O(\ell)$, wenn inkrementelle Nachfolgerberechnung möglich

Speicheraufwand:

- muss nur Knoten entlang aktuell exploriertem Pfad speichern ("entlang" = Knoten auf dem Pfad und deren Kinder)

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 ℓ mit $O(b\ell)$ erzeugten Knoten gefunden werden. (Warum?)
- verbesserbar auf $O(\ell)$, wenn inkrementelle Nachfolgerberechnung möglich

Speicheraufwand:

- muss nur Knoten entlang aktuell exploriertem Pfad speichern ("entlang" = Knoten auf dem Pfad und deren Kinder)

$\rightsquigarrow$ Speicheraufwand $O(bm)$ wenn m maximale erreichte Suchtiefe

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 ℓ mit $O(b\ell)$ erzeugten Knoten gefunden werden. (Warum?)
- verbesserbar auf $O(\ell)$, wenn inkrementelle Nachfolgerberechnung möglich

Speicheraufwand:

- muss nur Knoten entlang aktuell exploriertem Pfad speichern ("entlang" = Knoten auf dem Pfad und deren Kinder)

$\rightsquigarrow$ Speicheraufwand $O(bm)$ wenn m maximale erreichte Suchtiefe

- dieser niedrige Speicherbedarf ist der Hauptgrund, warum Tiefensuche trotz ihrer Nachteile interessant ist

Iterative Tiefensuche

Tiefenbeschränkte Suche

Tiefenbeschränkte Suche:

- Tiefensuche, die alle Suchknoten in einer gegebenen Tiefe n abschneidet (nicht weiter expandiert)

$\rightsquigarrow$ 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 ⟨a, s'⟩ ∈ succ(s):
        solution := depth_limited_search(s', depth_limit − 1)
        if solution ≠ 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

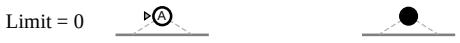
for *depth_limit* $\in \{0, 1, 2, \dots\}$:
    solution := depth_limited_search(init()), *depth_limit*)
    if *solution* $\neq$ **none**:
        return *solution*

Iterative Tiefensuche: Eigenschaften

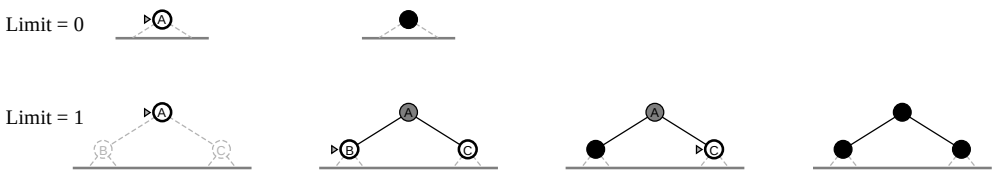
Kombiniert Vorteile von Breiten- und Tiefensuche:

- (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)

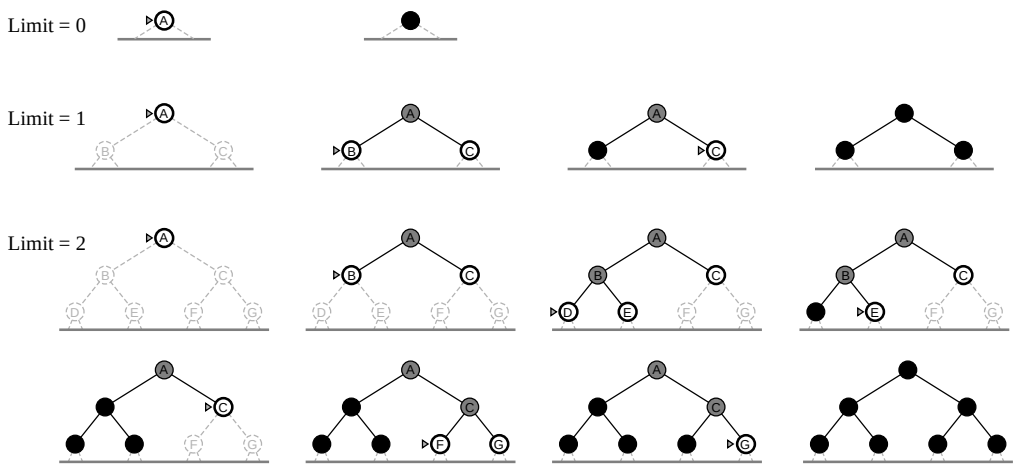
Iterative Tiefensuche: Beispiel



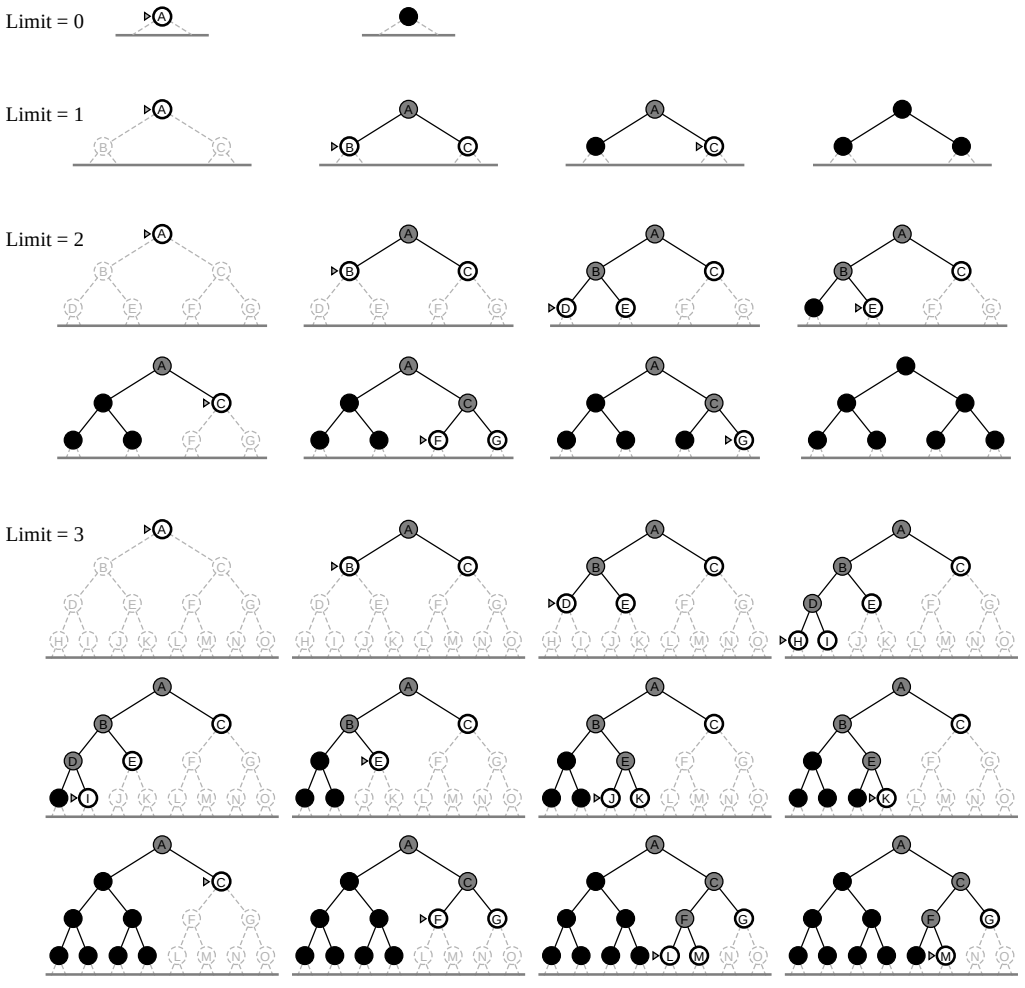
Iterative Tiefensuche: Beispiel



Iterative Tiefensuche: Beispiel



Iterative Tiefensuche: Beispiel



Iterative Tiefensuche: Beispiel für den Aufwand

Zeitaufwand (erzeugte Knoten):

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

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

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

und der Speicheraufwand beträgt

$$O(bd)$$

Iterative Tiefensuche: Bewertung

Iterative Tiefensuche: Bewertung

$\rightsquigarrow$ Iterative Tiefensuche ist oft die Methode der Wahl, wenn

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

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	$O(b^d)$	$O(b^{1+\lfloor c^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Speicher	$O(b^d)$	$O(b^{1+\lfloor c^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$

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

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