Grundlagen der Künstlichen Intelligenz

4. Suchalgorithmen: Grundlagen & blinde Verfahren

Malte Helmert

Universität Basel

11. März 2013

Grundlagen

Suchalgorithmen

Beginne beim Anfangszustand und expandiere in jedem Schritt einen Zustand durch Erzeugen seiner Nachfolger

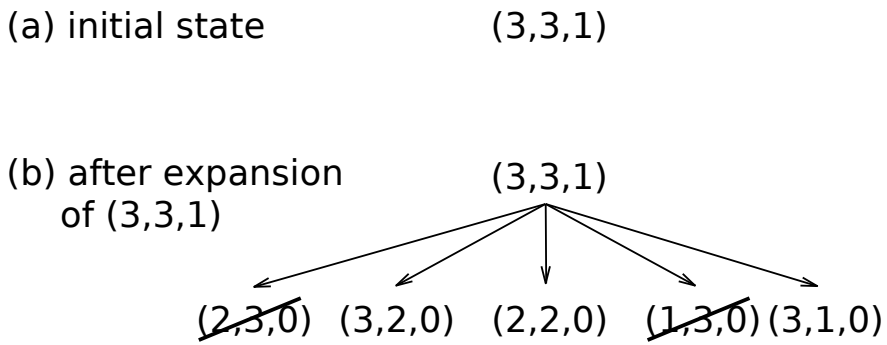
$\rightsquigarrow$ Suchbaum bzw. Suchraum

(a) initial state (3,3,1)

Suchalgorithmen

Beginne beim Anfangszustand und expandiere in jedem Schritt einen Zustand durch Erzeugen seiner Nachfolger

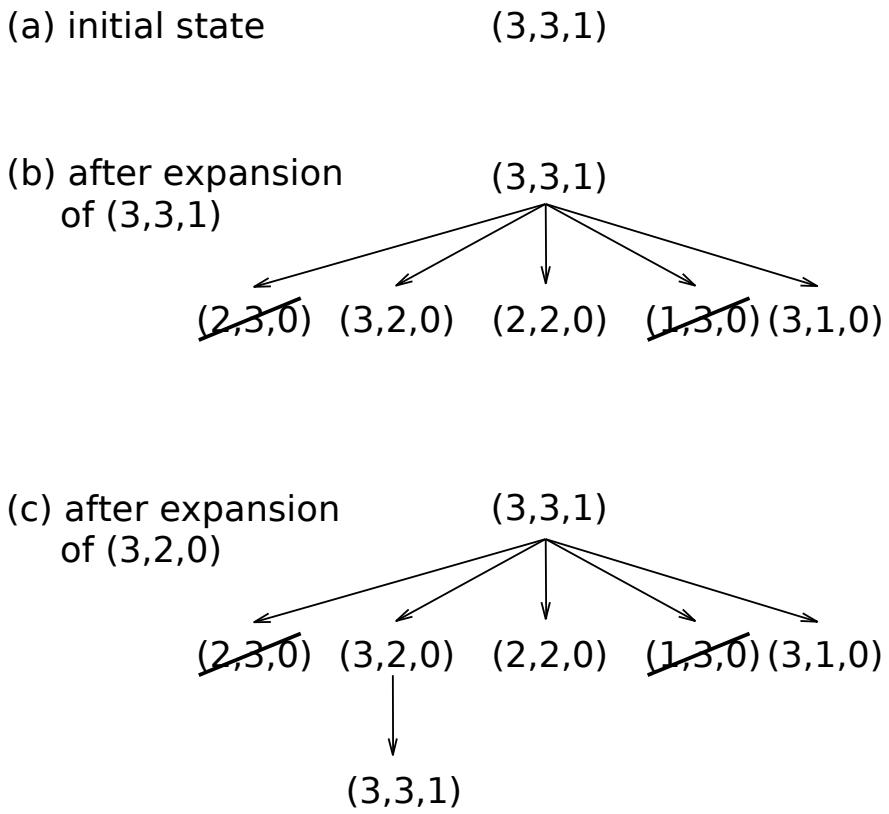
$\rightsquigarrow$ Suchbaum bzw. Suchraum



Suchalgorithmen

Beginne beim Anfangszustand und expandiere in jedem Schritt einen Zustand durch Erzeugen seiner Nachfolger

$\rightsquigarrow$ Suchbaum bzw. Suchraum



Begriffe

- Suchknoten
  repräsentiert in der Suche erreichten Zustand
  und zugehörige Informationen
- Knotenexpansion
  Erzeugen der Nachfolgerknoten eines Knotens
  durch Anwendung aller dort anwendbaren Aktionen
- Open-Liste oder Frontier
  Menge der Knoten, die Kandidaten für die Expansion sind
- Closed-Liste
  Menge der bereits expandierten Knoten
- Suchstrategie
  bestimmt, welcher Knoten als nächstes expandiert wird

Baumsuche vs. Graphensuche

Baumsuche:

- Suchknoten als Baum organisiert (Suchbaum)
- jeder Knoten entspricht einem Pfad vom Anfangszustand
- Duplikate (mehrere Knoten für denselben Zustand) möglich
- Suchbaum kann endlos tief verzweigen („im Kreis laufen“)

Graphensuche:

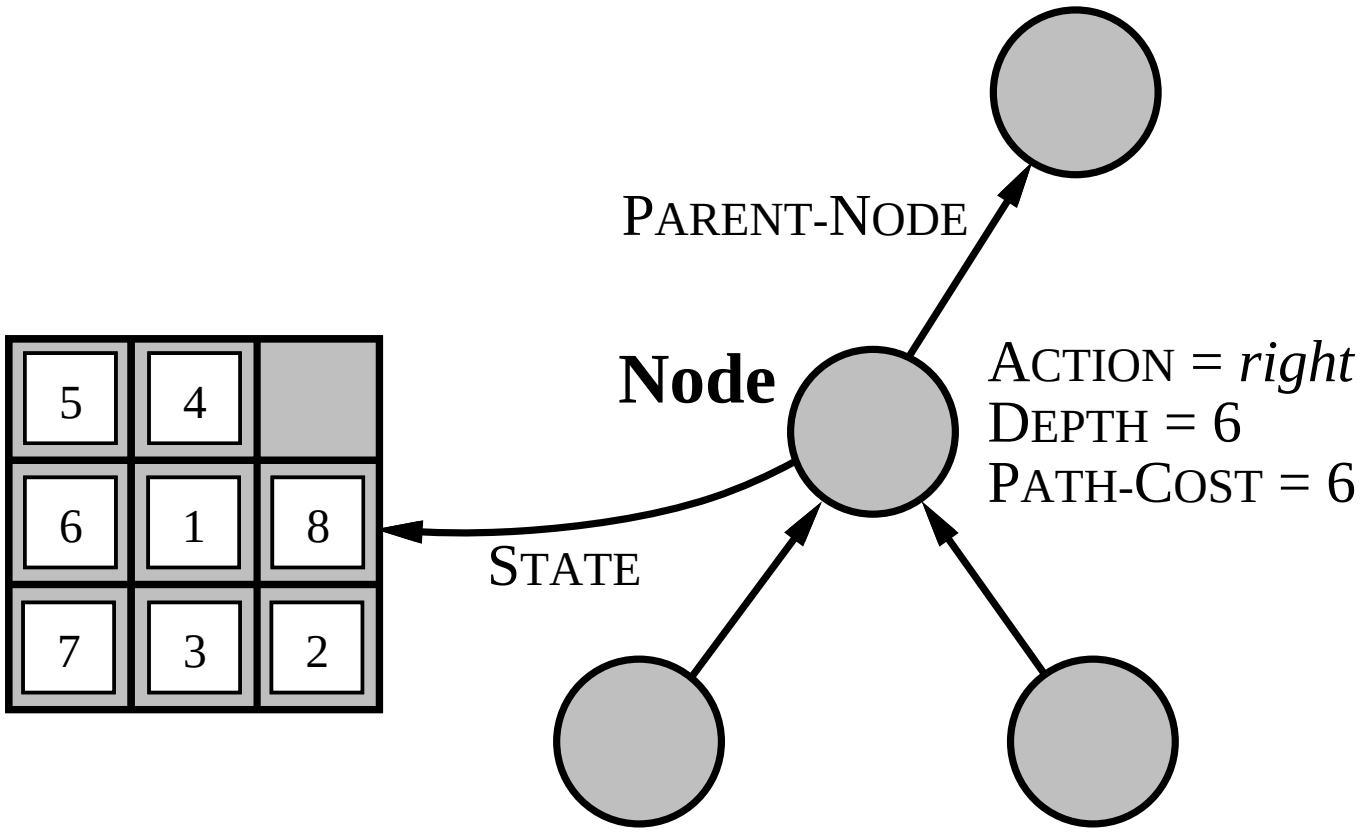
- Suchknoten als gerichteter Graph organisiert (Suchraum)
- Duplikate vermeiden: zu jedem Zustand max. ein Knoten

Implementierung des Suchraums

Datenstruktur für jeden Suchknoten n

n.State:	Zustand, dem der Knoten entspricht
n.Parent:	Suchknoten, der diesen Knoten erzeugt hat
n.Action:	Aktion, durch die wir vom Parent zu n gelangen
n.Path-Cost:	Kosten des Pfades vom Anfangszustand zu n.State, der sich durch Verfolgen der Parent-Verweise ergibt (traditionell $g(n)$ genannt)

Knoten im Suchraum



Operationen für die Open-Liste

Operationen für die Open-Liste

Empty?(*frontier*):	liefert **true** gdw. die Open-Liste leer ist
Pop(*frontier*):	entfernt ein Element aus der Open-Liste und liefert es zurück
Insert(*element*, *frontier*):	fügt ein Element in die Open-Liste ein und liefert die modifizierte Open-Liste zurück

$\rightsquigarrow$ unterschiedliche Implementierungen von Pop/Insert ergeben unterschiedliche Suchstrategien

Allgemeiner Baumsuchalgorithmus

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

- allgemeines Muster für Baumsuchalgorithmen
- konkrete Algorithmen oft wegen Effizienz anders implementiert, aber konzeptuell (= Suchstrategie) identisch

Allgemeiner Graphensuchalgorithmus

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

- allgemeines Muster für Graphensuchalgorithmen
- konkrete Algorithmen oft wegen Effizienz leicht anders implementiert, aber konzeptuell (= Suchstrategie) identisch

Kriterien für Suchalgorithmen

Vollständigkeit: Findet der Algorithmus garantiert eine Lösung, wenn eine existiert? (Semi-Vollständigkeit)
Terminiert er, wenn keine Lösung existiert?

Optimalität: Sind gefundene Lösungen garantiert optimal?

Zeitaufwand: Wie lange benötigt der Algorithmus, um eine Lösung zu finden?
(gemessen in erzeugten Zuständen)

Platzaufwand: Wie viel Speicherplatz benötigt der Algorithmus, um eine Lösung zu finden?
(gemessen in Zuständen)

oft verwendete Grössen:

- b: Verzweigungsgrad (= max. Nachfolgerzahl eines Zustands)
- d: Suchtiefe (Länge des längsten Pfads im Suchraum)

Blinde Suchverfahren

Blinde Suchverfahren

uninformierte oder blinde Suchverfahren

verwenden keine weiteren Informationen über Suchräume ausser den definierenden Bestandteilen*

- Breitensuche, uniforme Kostensuche, Tiefensuche
- tiefenbeschränkte Suche, iterative Tiefensuche
- bidirektionale Suche

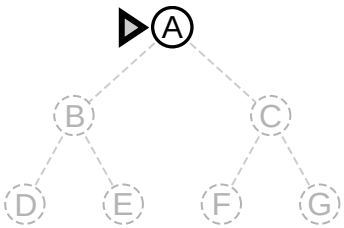
* mit kleiner Ausnahme bei bidirektionaler Suche

Gegensatz: informierte oder heuristische Suchverfahren

Breitensuche

Knoten werden in Reihenfolge ihrer Erzeugung expandiert (FIFO).

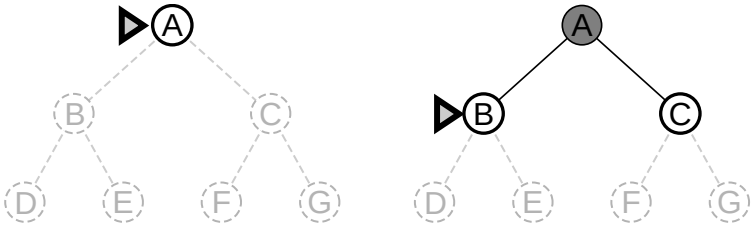
$\rightsquigarrow$ Open-Liste z. B. als verkettete Liste oder deque implementiert



Breitensuche

Knoten werden in Reihenfolge ihrer Erzeugung expandiert (FIFO).

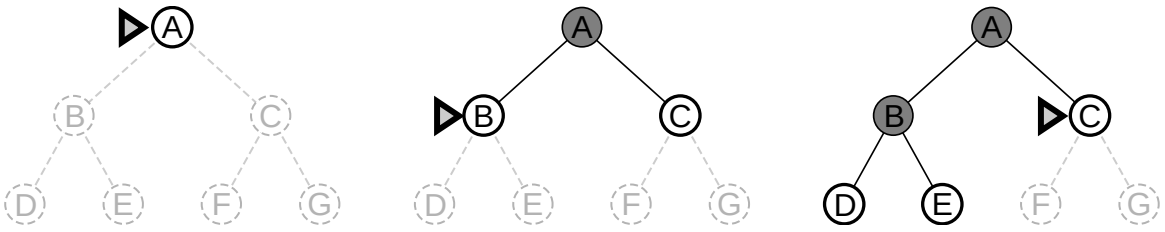
$\rightsquigarrow$ Open-Liste z. B. als verkettete Liste oder deque implementiert



Breitensuche

Knoten werden in Reihenfolge ihrer Erzeugung expandiert (FIFO).

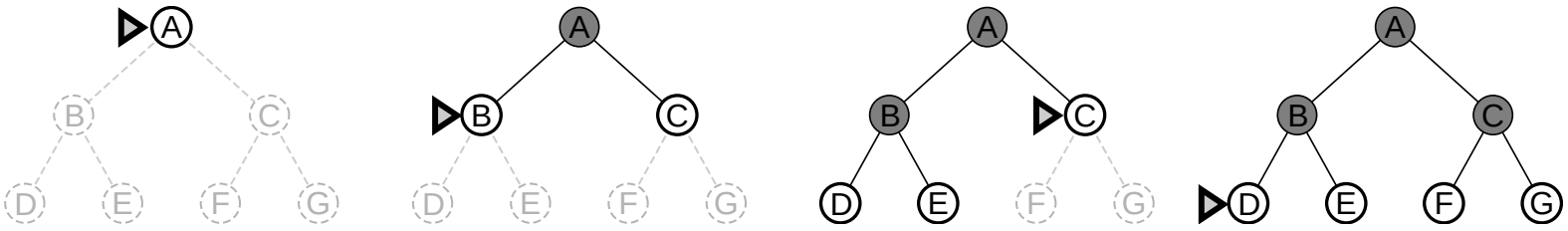
$\rightsquigarrow$ Open-Liste z. B. als verkettete Liste oder deque implementiert



Breitensuche

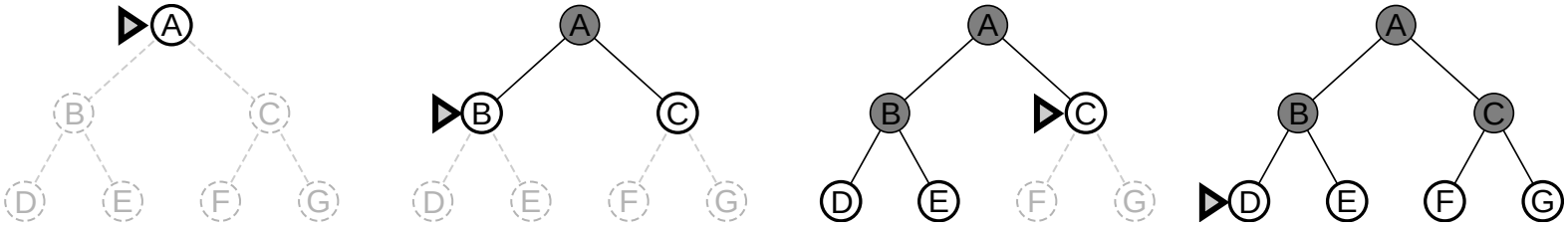
Knoten werden in Reihenfolge ihrer Erzeugung expandiert (FIFO).

$\rightsquigarrow$ Open-Liste z. B. als verkettete Liste oder deque implementiert



Breitensuche

Knoten werden in Reihenfolge ihrer Erzeugung expandiert (FIFO).
$\rightsquigarrow$ Open-Liste z. B. als verkettete Liste oder deque implementiert



- sucht Zustandsraum ebenenweise ab
- findet immer flachsten Zielzustand zuerst
- offensichtlich vollständig
- optimal, wenn alle Aktionen identische Kosten haben

englisch: breadth-first search

Breitensuche: Pseudo-Code

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
frontier ← a FIFO queue with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the shallowest node in *frontier* */
  add *node*.STATE to *explored*
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      if *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)
      frontier ← INSERT(*child*, *frontier*)

Breitensuche: Aufwand

Satz (Zeitaufwand der Breitensuche)

Sei b der (maximale) Verzweigungsgrad und d die minimale Lösungslänge im durchsuchten Zustandsraum. Gelte $b \geq 2$.

Dann ist der Zeitaufwand der Breitensuche

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

Erinnerung: wir messen den Zeitaufwand als Zahl erzeugter Knoten

Es folgt direkt, dass (für $b \geq 2$) auch der Platzaufwand der Breitensuche $O(b^d)$ beträgt. (Warum?)

Breitensuche: Beispiel zum Aufwand

Beispiel: $b = 10$; 100'000 Knoten/Sekunde; 32 Bytes/Knoten:

d	Knoten	Zeit	Speicher
3	1'111	0.01 s	35 KB
5	111'111	1 s	3.4 MB
7	10^7	2 min	339 MB
9	10^9	3 h	33 GB
11	10^{11}	13 Tage	3.2 TB
13	10^{13}	3.5 Jahre	323 TB
15	10^{15}	350 Jahre	32 PB

Breitensuche: Diskussion

Unser Pseudo-Code für Breitensuche ist eine Graphensuche, d. h. sie eliminiert Duplikate.

Warum ist das hier sinnvoller als Baumsuche?

Breitensuche: Diskussion

Unser Pseudo-Code für Breitensuche ist eine Graphensuche, d. h. sie eliminiert Duplikate.

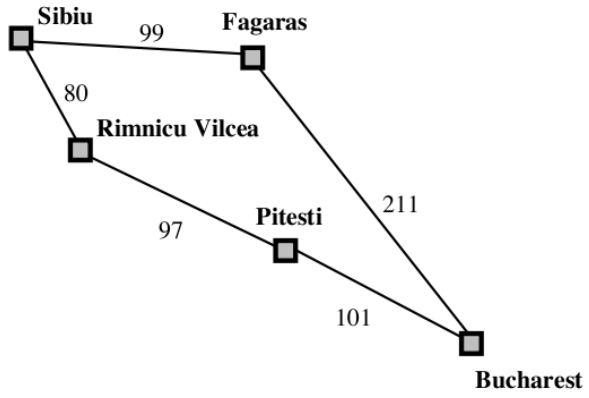
Warum ist das hier sinnvoller als Baumsuche?

Unser Pseudo-Code für Breitensuche führt den Zieltest anderswo aus als der allgemeine Pseudo-Code für Graphensuche.

Warum ist das sinnvoll?

Uniforme Kostensuche

- alle Aktionen gleich teuer $\rightsquigarrow$ Breitensuche optimal
- anderenfalls kann sie suboptimal sein $\rightsquigarrow$ siehe Beispiel



Abhilfe: uniforme Kostensuche (uniform cost search)

- expandiert immer einen Knoten mit minimalen Pfadkosten (n.Path-Cost bzw. $g(n)$)
- Implementierung: Prioritätswarteschlange (Heap) für Open-Liste

Uniforme Kostensuche: Pseudo-Code

Uniforme Kostensuche (mit verzögerter Duplikateliminierung)

```
open := new min-heap ordered by g
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    n = open.pop-min()
    if n.state ∉ closed:
        closed := closed ∪ {n.state}
        if is-goal(n.state):
            return extract-solution(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            if h(s') < ∞:
                n' := make-node(n, a, s')
                open.insert(n')
return unsolvable
```

Uniforme Kostensuche: Eigenschaften

Eigenschaften der uniformen Kostensuche:

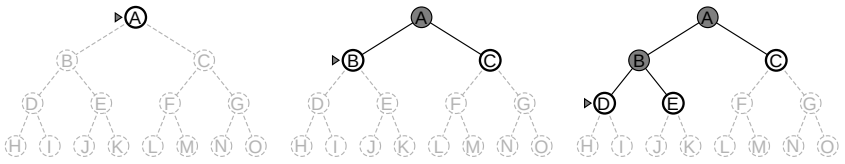
- vollständig und optimal
- identisch mit Dijkstra's Algorithmus (für explizite Graphen)
- Zieltest darf erst beim Expandieren geschehen! (Warum?)
- Zeitaufwand hängt stark von Verteilung der Aktionskosten ab (keine einfachen und genauen Abschätzungen)
  - Sei $\epsilon := \min_{a \in A} cost(a)$. Wenn $\epsilon > 0$, optimale Lösungskosten c^* und Verzweigungsgrad b, dann Aufwand $O(b^{1+\lfloor c^*/\epsilon \rfloor})$. (Warum?)
  - Oft sehr ungenaue obere Schranke.
- Platzaufwand = Zeitaufwand

Tiefensuche

Zuletzt erzeugte Knoten werden zuerst expandiert (LIFO)
$\rightsquigarrow$ tiefste Knoten zuerst (englisch: depth-first search)

- Open-Liste als Stack implementiert

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



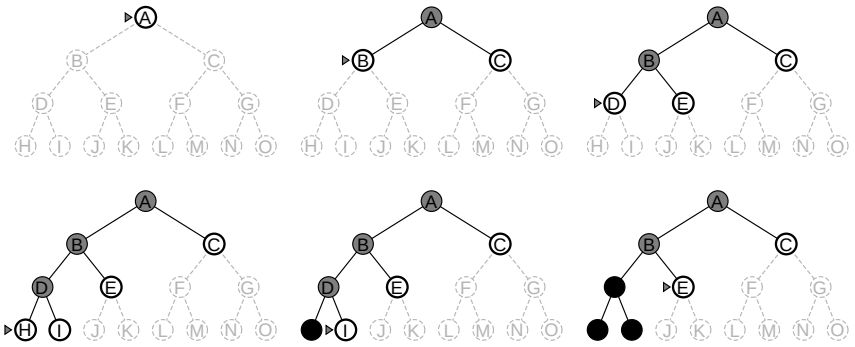
Tiefensuche

Zuletzt erzeugte Knoten werden zuerst expandiert (LIFO)

$\rightsquigarrow$ tiefste Knoten zuerst (englisch: depth-first search)

- Open-Liste als Stack implementiert

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

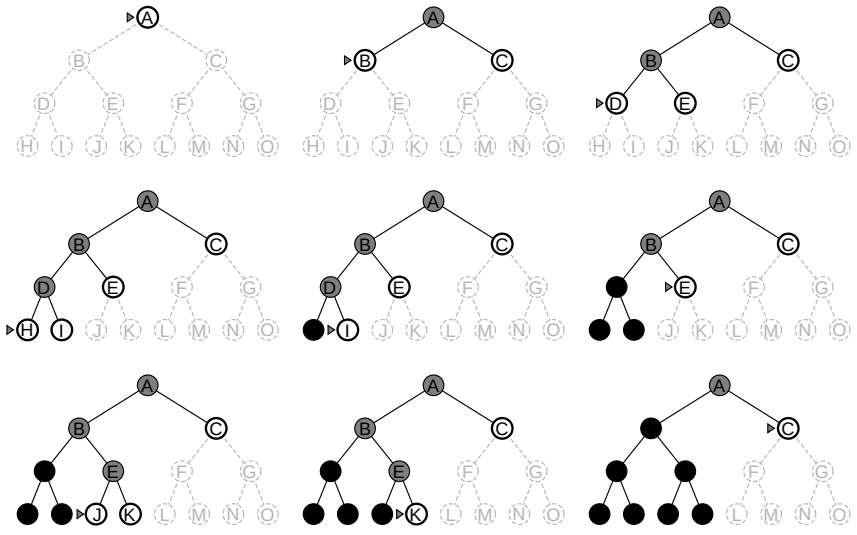


Tiefensuche

Zuletzt erzeugte Knoten werden zuerst expandiert (LIFO)
$\rightsquigarrow$ tiefste Knoten zuerst (englisch: depth-first search)

- Open-Liste als Stack implementiert

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

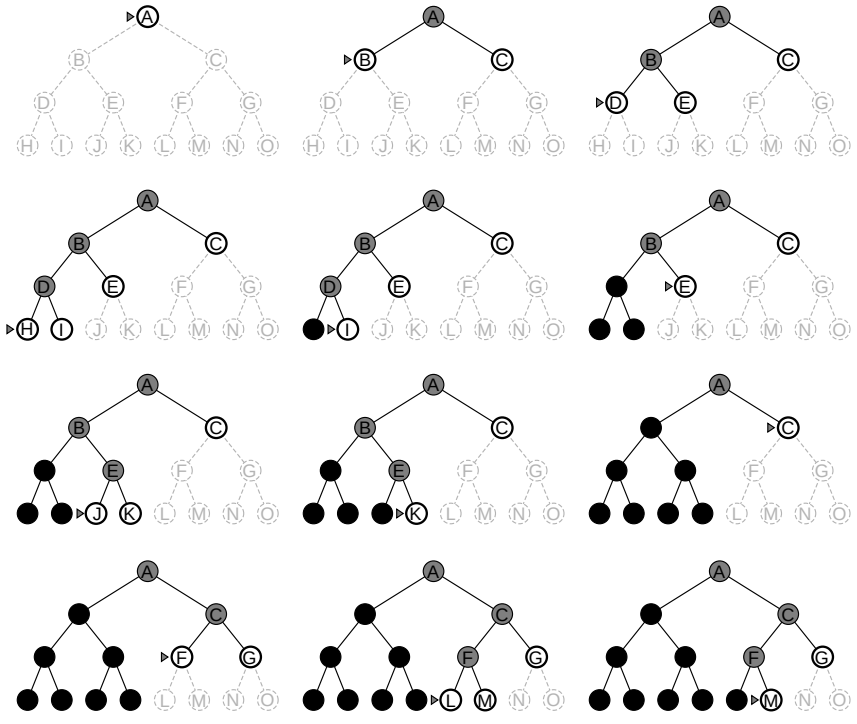


Tiefensuche

Zuletzt erzeugte Knoten werden zuerst expandiert (LIFO)
$\rightsquigarrow$ tiefste Knoten zuerst (englisch: depth-first search)

- Open-Liste als Stack implementiert

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



Tiefensuche: Eigenschaften

- wird fast immer als Baumsuche implementiert (warum, wird später deutlich)
- nicht vollständig oder semi-vollständig, nicht optimal (Warum?)
- vollständig, wenn Zustandsraum azyklisch, z.B. Baum ($\rightsquigarrow$ Constraint-Satisfaction-Probleme)

Tiefensuche: Eigenschaften

- wird fast immer als Baumsuche implementiert
  (warum, wird später deutlich)
- nicht vollständig oder semi-vollständig, nicht optimal
  (Warum?)
- vollständig, wenn Zustandsraum azyklisch, z.B. Baum
  ($\rightsquigarrow$ Constraint-Satisfaction-Probleme)

Implementierung:

- üblich und effizient: Tiefensuche als rekursive Funktion

$\rightsquigarrow$ nutze Stack der Programmiersprache/CPU für die Open-Liste

- Pseudo-Code später ($\rightsquigarrow$ tiefenbeschränkte Suche)

Tiefensuche: Aufwand

Zeitaufwand:

- Wenn im Zustandsraum Pfade der Länge m existieren, kann Tiefensuche $O(b^m)$ viele Knoten erzeugen, selbst wenn sehr kurze Lösungen (z. B. Länge 1) existieren.
- Umgekehrt kann im besten Fall eine Lösung der Länge l mit nur $O(bl)$ erzeugten Knoten gefunden werden.

Tiefensuche: Aufwand

Zeitaufwand:

- Wenn im Zustandsraum Pfade der Länge m existieren, kann Tiefensuche $O(b^m)$ viele Knoten erzeugen, selbst wenn sehr kurze Lösungen (z. B. Länge 1) existieren.
- Umgekehrt kann im besten Fall eine Lösung der Länge l mit nur $O(bl)$ erzeugten Knoten gefunden werden.

Platzaufwand (als Baumsuche!):

- muss nur Knoten entlang des Pfades von der Wurzel zum gerade expandierten Knoten im Speicher halten („entlang“ = Knoten auf diesem Pfad und deren Kinder)

Tiefensuche: Aufwand

Zeitaufwand:

- Wenn im Zustandsraum Pfade der Länge m existieren, kann Tiefensuche $O(b^m)$ viele Knoten erzeugen, selbst wenn sehr kurze Lösungen (z. B. Länge 1) existieren.
- Umgekehrt kann im besten Fall eine Lösung der Länge l mit nur $O(bl)$ erzeugten Knoten gefunden werden.

Platzaufwand (als Baumsuche!):

- muss nur Knoten entlang des Pfades von der Wurzel zum gerade expandierten Knoten im Speicher halten („entlang“ = Knoten auf diesem Pfad und deren Kinder)
- wenn m die maximale Tiefe ist, zu der die Suche vordringt, ist Platzaufwand daher $O(bm)$

Tiefensuche: Aufwand

Zeitaufwand:

- Wenn im Zustandsraum Pfade der Länge m existieren, kann Tiefensuche $O(b^m)$ viele Knoten erzeugen, selbst wenn sehr kurze Lösungen (z. B. Länge 1) existieren.
- Umgekehrt kann im besten Fall eine Lösung der Länge l mit nur $O(bl)$ erzeugten Knoten gefunden werden.

Platzaufwand (als Baumsuche!):

- muss nur Knoten entlang des Pfades von der Wurzel zum gerade expandierten Knoten im Speicher halten („entlang“ = Knoten auf diesem Pfad und deren Kinder)
- wenn m die maximale Tiefe ist, zu der die Suche vordringt, ist Platzaufwand daher $O(bm)$
- dieser niedrige Platzaufwand ist der Grund, warum Tiefensuche trotz aller Nachteile interessant ist

Tiefenbeschränkte Suche

Tiefenbeschränkte Suche (depth-limited search):

- Tiefensuche, bei der Abschneiden des Suchpfads nach Tiefe n erzwungen wird
- das heisst: Knoten führen Tiefe d im Suchbaum mit, und Knoten in Tiefe n werden nicht expandiert

$\rightsquigarrow$ für sich allein nicht sehr nützlich, aber Bestandteil nützlicher Algorithmen

Tiefenbeschränkte Suche: Pseudo-Code

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Iterative Tiefensuche

Iterative Tiefensuche (iterative-deepening search)

- Idee: wiederholte tiefenbeschränkter Suchen mit wachsenden Tiefenschranken
- klingt verschwenderisch (jede Iteration wiederholt die vorher gemachte Arbeit), aber der Aufwand ist tatsächlich vertretbar

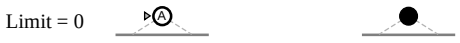
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Iterative Tiefensuche: Eigenschaften

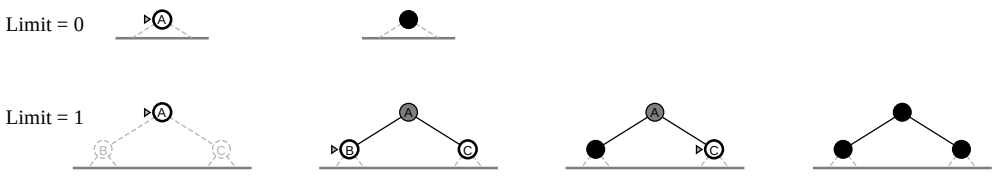
Kombiniert Vorteile von Breiten- und Tiefensuche:

- (fast) wie Breitensuche: semi-vollständig
- wie Breitensuche: bei einheitlichen Kosten optimal
- wie Tiefensuche: Platzbedarf nur entlang eines Pfades
  $\rightsquigarrow$ Platzaufwand $O(bd)$, wobei d minimale Lösungslänge
- Zeitbedarf kaum höher als Breitensuche (siehe später)

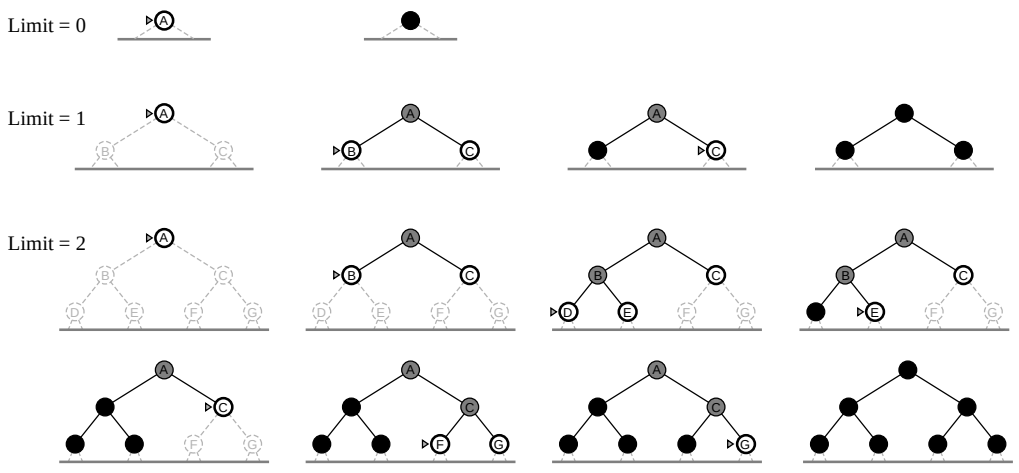
Iterative Tiefensuche: Beispiel



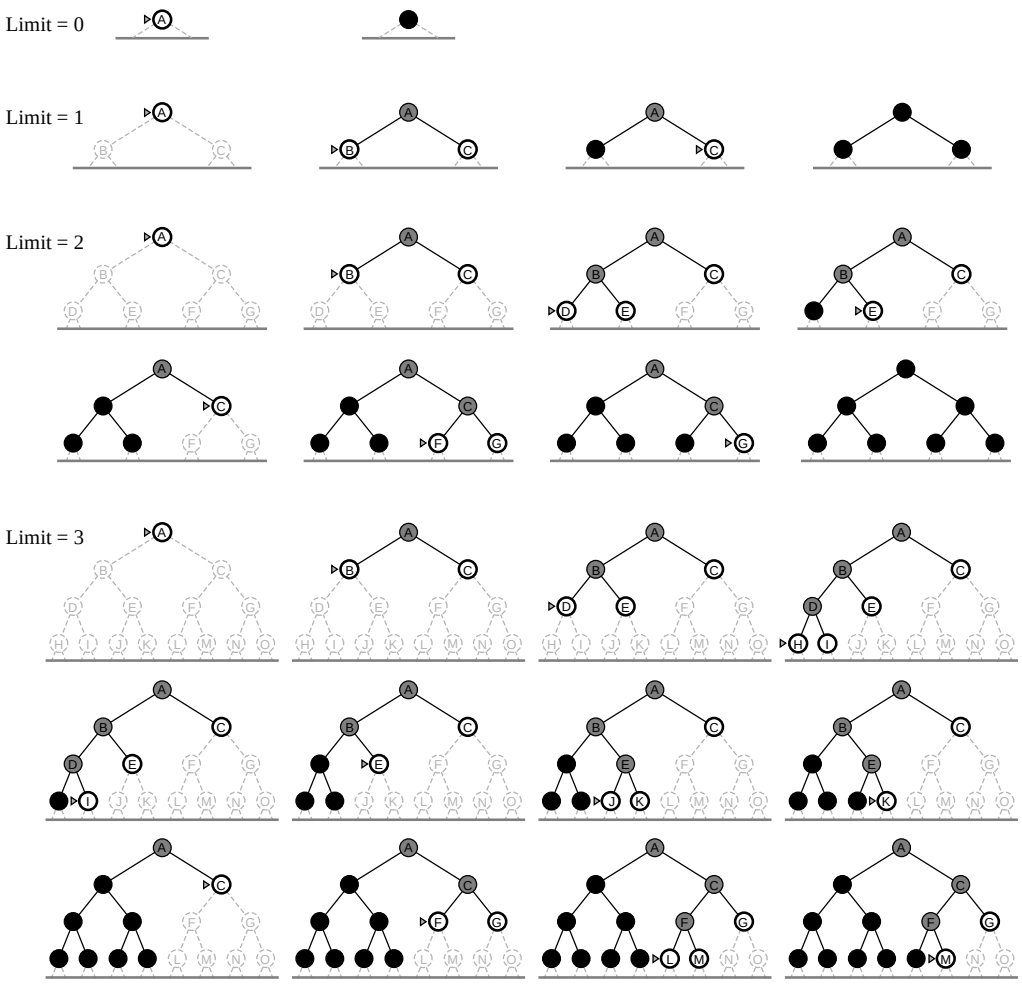
Iterative Tiefensuche: Beispiel



Iterative Tiefensuche: Beispiel



Iterative Tiefensuche: Beispiel



Iterative Tiefensuche: Analyse

Zeitaufwand (erzeugte Zustände):

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

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

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

Für $b = 10$ werden nur 11% mehr Knoten erzeugt als bei Breitensuche.

Iterative Tiefensuche: Analyse (formal)

Satz (Zeitaufwand der iterativen Tiefensuche)

Sei b der (maximale) Verzweigungsgrad und d die minimale Lösungslänge im durchsuchten Zustandsraum. Gelte $b \geq 2$.

Dann ist der Zeitaufwand der iterativen Tiefensuche

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

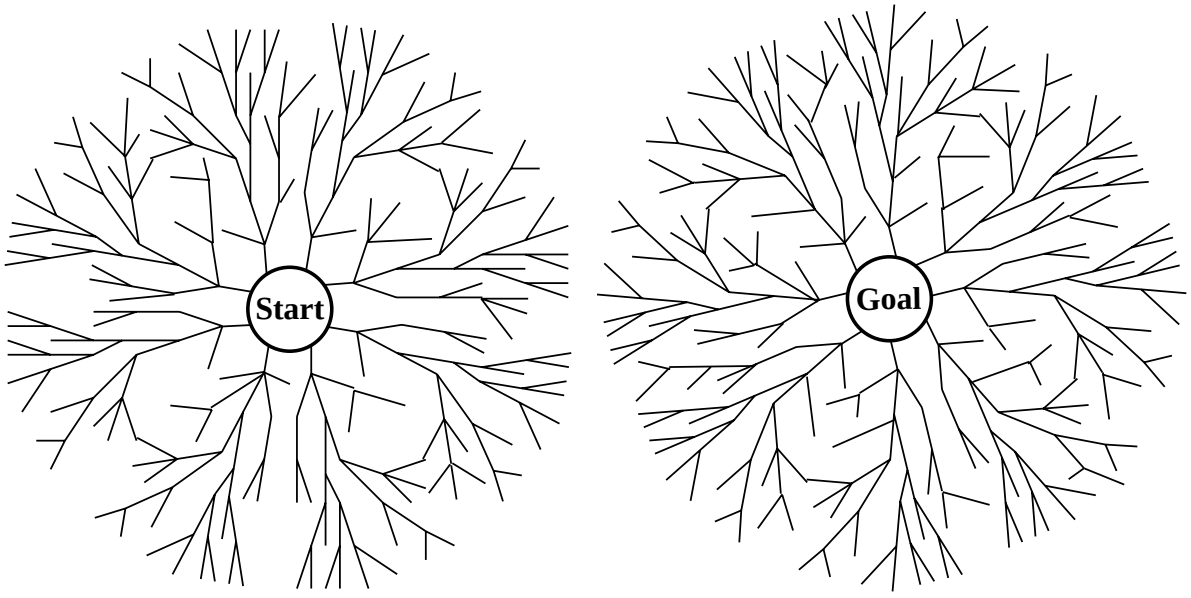
und der Platzaufwand

$$O(bd)$$

Einschätzung

$\rightsquigarrow$ Iterative Tiefensuche ist im Allgemeinen die Methode der Wahl, wenn Baumsuche angemessen und Lösungstiefe unbekannt ist.

Bidirektionale Suche



wenn nur ein Zielzustand und Zustandsraum symmetrisch
$\rightsquigarrow$ Suche in Zeit $O(2 \cdot b^{\lceil d/2 \rceil}) = O(b^{\lceil d/2 \rceil})$ möglich

Beispiel: $b = 10$, $d = 6$ $\rightsquigarrow$ statt 1'111'111 Knoten nur 2'222!

Bidirektionale Suche: Schwierigkeiten

- Aktionen nicht immer symmetrisch,
  daher Berechnung von Vorgängern für Rückwärtssuche schwierig
- in vielen Fällen sehr viele mögliche Zielzustände,
  die schwer charakterisierbar sind
- mindestens eine Suchrichtung muss platzaufwändigen Algorithmus (z. B. Breitensuche) verwenden
- Kombination mit informierten Suchverfahren (nächstes Kapitel) schwierig

Zusammenfassung

Vergleich der Suchverfahren

Vollständigkeit, Optimalität, Zeitaufwand, Platzaufwand

Kriterium	Breiten-Suche	Uniforme Kosten	Tiefen-Suche	tiefen-beschr.	iter. Tiefens.	Bidirek-tional
vollständig?	Ja	Ja	Nein	Nein	Semi	Ja
optimal?	Ja*	Ja	Nein	Nein	Ja*	Ja*,**
Zeit	$O(b^d)$	$O(b^{1+\lfloor c^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Platz	$O(b^d)$	$O(b^{1+\lfloor c^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

$b \geq 2$ Verzweigungsgrad
d minimale Lösungstiefe
m maximale durchsuchte Tiefe
l Tiefenlimit
c^* optimale Lösungskosten
$\epsilon > 0$ minimale Aktionskosten

Anmerkungen:
* nur bei uniformen Aktionskosten
** bei Verwendung von Breitensuchen; Annahmen: nur ein Zielzustand; symmetrisch