Grundlagen der Künstlichen Intelligenz

8. Klassische Suche: Breitensuche und uniforme Kostensuche

Malte Helmert

Universität Basel

17. März 2014

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

Blinde Suche

Blinde Suche

In den folgenden beiden Kapiteln betrachten wir blinde Suchalgorithmen:

blinde Suchalgorithmen

Blinde Suchalgorithmen verwenden keine Informationen über Zustandsräume ausser dem Black-Box-Interface.

Sie werden auch uninformierte Suchalgorithmen genannt.

vergleiche: heuristische Suchalgorithmen (ab Kapitel 10)

Blinde Suchalgorithmen: Beispiele

Beispiele für blinde Suchalgorithmen:

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

Blinde Suchalgorithmen: Beispiele

Beispiele für blinde Suchalgorithmen:

- Breitensuche ($\rightsquigarrow$ dieses Kapitel)
- uniforme Kostensuche ($\rightsquigarrow$ dieses Kapitel)
- Tiefensuche
- tiefenbeschränkte Suche
- iterative Tiefensuche

Blinde Suchalgorithmen: Beispiele

Beispiele für blinde Suchalgorithmen:

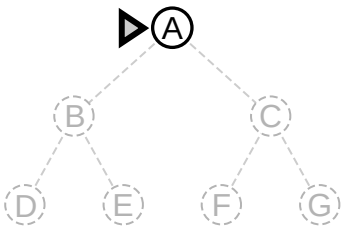
- Breitensuche ($\rightsquigarrow$ dieses Kapitel)
- uniforme Kostensuche ($\rightsquigarrow$ dieses Kapitel)
- Tiefensuche ($\rightsquigarrow$ nächstes Kapitel)
- tiefenbeschränkte Suche ($\rightsquigarrow$ nächstes Kapitel)
- iterative Tiefensuche ($\rightsquigarrow$ nächstes Kapitel)

Breitensuche: Einführung

Breitensuche

Breitensuche expandiert Knoten in Erzeugungsreihenfolge (FIFO).

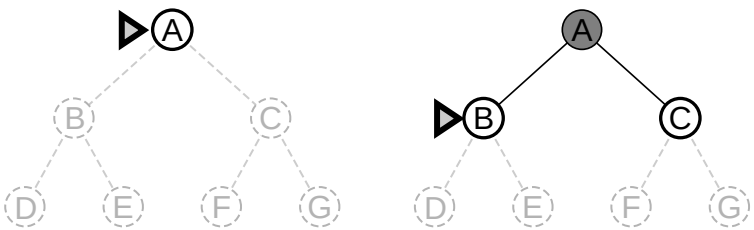
$\rightsquigarrow$ z. B. Open-Liste als verkettete Liste oder Deque



Breitensuche

Breitensuche expandiert Knoten in Erzeugungsreihenfolge (FIFO).

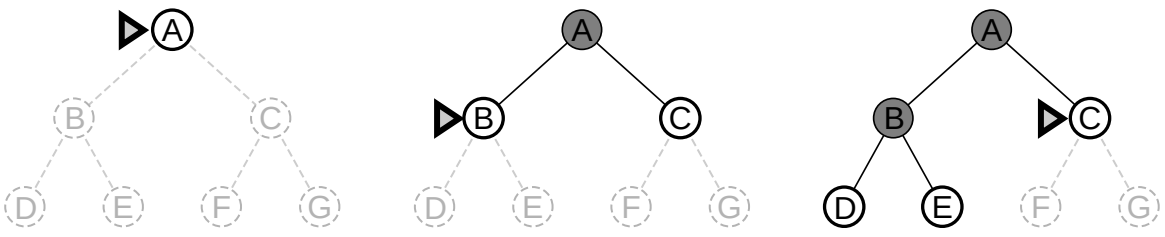
$\rightsquigarrow$ z. B. Open-Liste als verkettete Liste oder Deque



Breitensuche

Breitensuche expandiert Knoten in Erzeugungsreihenfolge (FIFO).

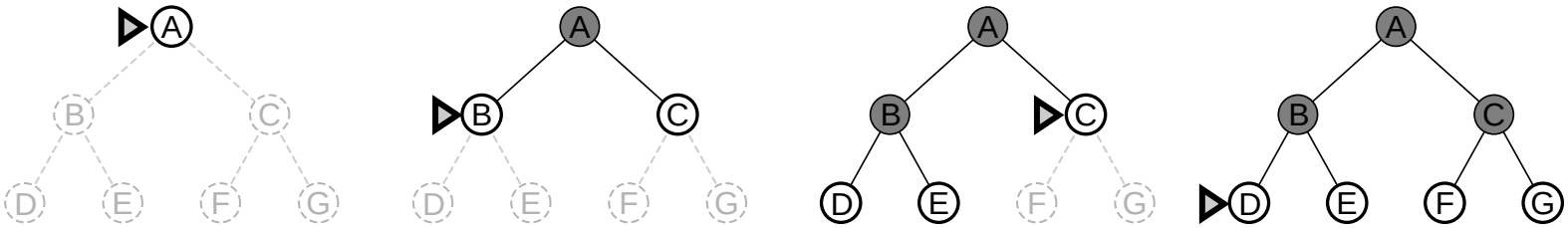
$\rightsquigarrow$ z. B. Open-Liste als verkettete Liste oder Deque



Breitensuche

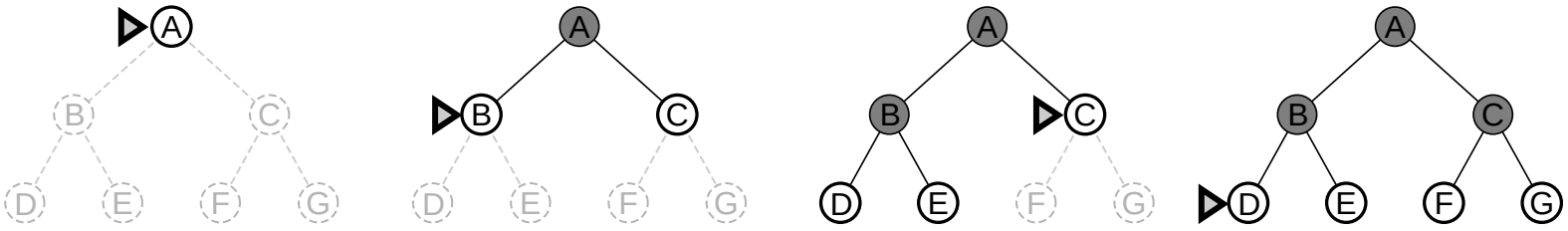
Breitensuche expandiert Knoten in Erzeugungsreihenfolge (FIFO).

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



Breitensuche

Breitensuche expandiert Knoten in Erzeugungsreihenfolge (FIFO).
$\rightsquigarrow$ z. B. Open-Liste als verkettete Liste oder Deque



- durchsucht Zustandsraum ebenenweise
- findet immer flachsten Zielzustand zuerst

Breitensuche: Baumsuche oder Graphensuche?

Breitensuche kann

- ohne Duplikateliminierung (als Baumsuche)
  $\rightsquigarrow$ BFS-Tree
- oder mit Duplikateliminierung (als Graphensuche)
  $\rightsquigarrow$ BFS-Graph

durchgeführt werden.

$\rightsquigarrow$ Wir betrachten beide Varianten.

BFS-Tree

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 ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable
```

BFS-Tree (1. Versuch)

Breitensuche ohne Duplikateliminierung (1. Versuch):

BFS-Tree (1. Versuch)

```
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n = open.pop_front()
    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
```

BFS-Tree (1. Versuch): Diskussion

Das ist schon fast ein brauchbarer Algorithmus,
aber er verschwendet etwas Zeit:

- In einer Breitensuche ist der erste erzeugte Zielknoten auch immer der erster expandierte Zielknoten. (Warum?)
- Daher ist es effizienter, den Zieltest bereits durchzuführen, wenn ein Knoten erzeugt wird
  (nicht erst, wenn er expandiert wird).

$\rightsquigarrow$ Wieviel Zeit spart das?

BFS-Tree (2. Versuch)

Breitensuche ohne Duplikateliminierung (2. Versuch):

BFS-Tree (2. Versuch)

```
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n = open.pop_front()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Tree (2. Versuch): Diskussion

Wo ist der Bug?

BFS-Tree (endgültige Version)

Breitensuche ohne Duplikateliminierung (endgültige Version):

BFS-Tree

```
if is_goal(init()):
    return ⟨⟩
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n = open.pop_front()
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Tree (endgültige Version)

Breitensuche ohne Duplikateliminierung (endgültige Version):

BFS-Tree

```
if is_goal(init()):
    return ⟨⟩
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n = open.pop_front()
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Graph

Erinnerung: generischer Graphensuchalgorithmus

Erinnerung aus Kapitel 7:

Generische Graphensuche

```
open := new OpenList
open.insert(make_root_node())
closed := new ClosedList
while not open.is_empty():
    n = open.pop()
    if closed.lookup(n.state) = none:
        closed.insert(n)
        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
```

Anpassung der generischen Graphensuche für Breitensuche

Anpassung der generischen Graphensuche für Breitensuche:

- analoge Anpassungen zu BFS-Tree
  (Deque als Open-Liste, früher Zieltest)
- da Closed-Liste hier nur zur Duplikaterkennung dient,
  nicht zur Verwaltung von Knoteninformationen,
  reicht eine Mengen-Datenstruktur aus
- aus denselben Gründen, warum frühe Zieltests sinnvoll sind,
  sollten wir Duplikattests gegen die Closed-Liste
  und Updates der Closed-Liste so früh wie möglich vornehmen

BFS-Graph (Breitensuche mit Duplikateliminierung)

BFS-Graph

```
if is_goal(init()):
    return ⟨⟩
open := new Deque
open.push_back(make_root_node())
closed := new HashSet
closed.insert(init())
while not open.is_empty():
    n = open.pop_front()
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        if s' ∉ closed:
            closed.insert(s')
            open.push_back(n')
return unsolvable
```

Eigenschaften der Breitensuche

Eigenschaften der Breitensuche

Eigenschaften der Breitensuche:

- BFS-Tree ist semi-vollständig,
  aber nicht vollständig (Warum?)
- BFS-Graph ist vollständig (Warum?)
- BFS (beide Varianten) ist optimal,
  wenn alle Aktionen dieselben Kosten haben (Warum?),
  aber nicht im allgemeinen Fall (Warum nicht?)
- Aufwand: folgende Folien

Breitensuche: Aufwand

Das folgende Ergebnis gilt für beide BFS-Varianten:

Satz (Zeitaufwand der Breitensuche)

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

Dann beträgt der Zeitaufwand der Breitensuche

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

Erinnerung: wir messen Zeitaufwand in erzeugten Knoten

Es folgt, dass der Speicheraufwand beider BFS-Varianten ebenfalls $O(b^d)$ ist (falls $b \geq 2$). (Warum?)

Breitensuche: Beispiel für den Aufwand

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

d	Knoten	Zeit	Speicher
3	1'111	0.01 s	35 KiB
5	111'111	1 s	3.4 MiB
7	10^7	2 min	339 MiB
9	10^9	3 h	33 GiB
11	10^{11}	13 Tage	3.2 TiB
13	10^{13}	3.5 Jahre	323 TiB
15	10^{15}	350 Jahre	32 PiB

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

Vorteile von BFS-Graph:

- vollständig
- viel (!) effizienter, wenn es viele Duplikate gibt

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

Vorteile von BFS-Graph:

- vollständig
- viel (!) effizienter, wenn es viele Duplikate gibt

Vorteile von BFS-Tree:

- einfacher
- weniger Overhead (Zeit/Platz), wenn wenige/keine Duplikate

BFS-Tree oder BFS-Graph?

Was ist besser, BFS-Tree oder BFS-Graph?

Vorteile von BFS-Graph:

- vollständig
- viel (!) effizienter, wenn es viele Duplikate gibt

Vorteile von BFS-Tree:

- einfacher
- weniger Overhead (Zeit/Platz), wenn wenige/keine Duplikate

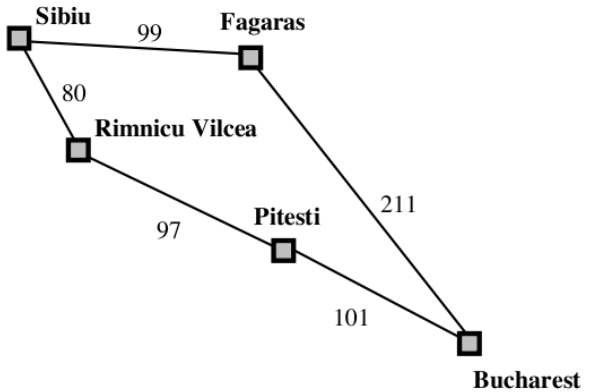
Schlussfolgerung

BFS-Graph ist normalerweise zu bevorzugen,
es sei denn wir wissen, dass es im gegebenen Zustandsraum vernachlässigbar wenige Duplikate gibt.

Uniforme Kostensuche

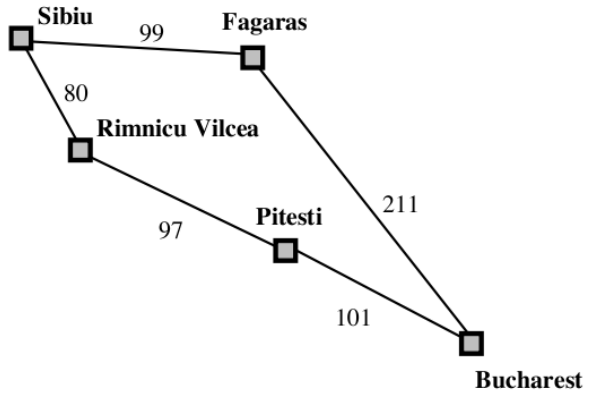
Uniforme Kostensuche

- Breitensuche optimal, wenn alle Aktionskosten gleich
- sonst Optimalität nicht garantiert $\rightsquigarrow$ Beispiel:



Uniforme Kostensuche

- Breitensuche optimal, wenn alle Aktionskosten gleich
- sonst Optimalität nicht garantiert $\rightsquigarrow$ Beispiel:



Abhilfe: uniforme Kostensuche

- expandiere immer Knoten mit minimalen Pfadkosten (n.path_cost alias $g(n)$)
- Implementierung: Prioritätswarteschlange (Min-Heap) für Open-Liste

Erinnerung: generischer Graphensuchalgorithmus

Erinnerung aus Kapitel 7:

Generische Graphensuche

```
open := new OpenList
open.insert(make_root_node())
closed := new ClosedList
while not open.is_empty():
    n = open.pop()
    if closed.lookup(n.state) = none:
        closed.insert(n)
        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
```

Uniforme Kostensuche

Uniforme Kostensuche

```
open := new MinHeap ordered by g
open.insert(make_root_node())
closed := new HashSet
while not open.is_empty():
    n = open.pop_min()
    if n.state ∉ closed:
        closed.insert(n)
        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
```

Uniforme Kostensuche: Diskussion

Anpassung der generischen Graphensuche
für uniforme Kostensuche:

- hier wären frühe Zieltests/frühe Updates der Closed-Liste keine gute Idee. (Warum nicht?)
- wie in BFS-Graph reicht eine Menge für die Closed-Liste aus
- eine Baumsuchvariante ist möglich, aber selten:
  dieselben Nachteile wie BFS-Tree und im allgemeinen
  nicht einmal semi-vollständig (Warum nicht?)

Anmerkung: identisch mit Dijkstras Algorithmus
für kürzeste Pfade in gewichteten Graphen!

Uniforme Kostensuche: Verbesserungen

Mögliche Verbesserungen:

- wenn Aktionskosten kleine Ganzzahlen sind,
  sind Bucket-Heaps oft effizienter
- zusätzliche frühe Duplikatstests für erzeugten Knoten
  können Speicheraufwand reduzieren
  und Laufzeit verbessern oder verschlechtern

Eigenschaften der uniformen Kostensuche (1)

Eigenschaften der uniformen Kostensuche:

- uniforme Kostensuche ist vollständig (Warum?)
- uniforme Kostensuche ist optimal (Warum?)

Eigenschaften der uniformen Kostensuche (2)

Eigenschaften der uniformen Kostensuche:

- Zeitaufwand hängt von Verteilung der Aktionskosten ab (keine einfachen und genauen Schranken).
  - Sei $\varepsilon := \min_{a \in A} cost(a)$ und gelte $\varepsilon > 0$.
  - Seien c^* die optimalen Lösungskosten.
  - Sei b der Verzweigungsgrad und gelte $b \geq 2$.
  - Dann beträgt der Zeitaufwand höchstens $O(b^{\lfloor c^*/\varepsilon \rfloor + 1})$. (Warum?)
  - oft eine sehr schwache obere Schranke
- Speicheraufwand = Zeitaufwand

Zusammenfassung

Zusammenfassung

- blinde Suchverfahren: verwenden keine Informationen ausser Black-Box-Interface des Zustandsraums
- Breitensuche: expandiere Knoten in Erzeugungsreihenfolge
  - durchsucht Zustandsraum ebenenweise
  - als Baumsuche oder als Graphensuche möglich
  - Aufwand $O(b^d)$ bei Verzweigungsgrad b, minimale Lösungslänge d (falls $b \geq 2$)
  - vollständig als Graphensuche; semi-vollständig als Baumsuche
  - optimal bei einheitlichen Aktionskosten
- uniforme Kostensuche: expandiere Knoten in Reihenfolge aufsteigender Pfadkosten
  - üblicherweise als Graphensuche
  - vollständig und optimal