Grundlagen der Künstlichen Intelligenz

3 Problemlösen durch Suche

Problemlösende Agenten, Problemformulierungen, Suchstrategien

Volker Steinhage

Inhalt

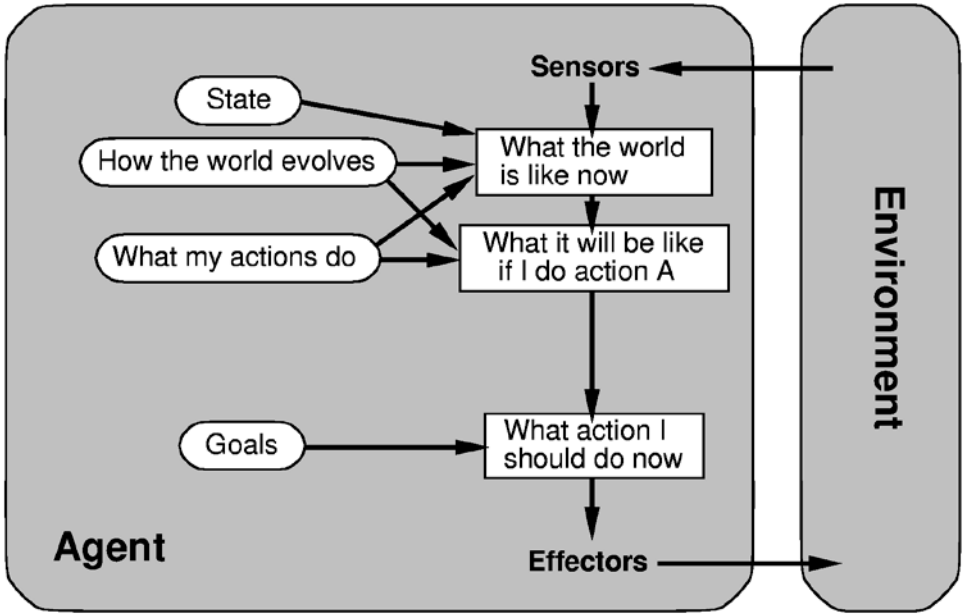
- Problemlösende Agenten
- Problemformulierungen
- Problemtypen
- Beispielprobleme
- Suchstrategien

Problemlösende Agenten

Zunächst Betrachtung zielorientierter Agenten!

- Gegeben: Ein Anfangszustand
- Formuliere: *Ziel* und *Problem*
- Gewünscht: Erreichen eines Zielzustandes *durch Ausführen geeigneter Aktionen*

→ Suche nach geeigneter Aktions-folge und Ausführung dieser Folge



Schwerpunkt dieses Kapitels: Problemformulierung und Suche

→ Agentenentwurf nach den Schema: Formulieren – Suchen – Ausführen

Ein einfacher problemlösender Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Design nach *Formulate-Search-Execute*-Schema:

1) Ziel- und Problemformulierung sowie Suche nach *vollständiger* Aktionssequenz
2) Ausführung der Aktionssequenz – Hinweis: dabei keine Beachtung der Umwelt!
3) Formulierung eines neuen Zieles

Umgebungsanforderung: deterministisch, statisch, diskret (*states*)

Problemformulierung

1) Festlegen des *Weltzustandsraums*

   - *durch Abstraktion*: nur Betrachtung der *relevanten* Aspekte
   - *mit Bestimmung des Problemtyps*: abhängig vom verfügbaren *Wissen* über Weltzustände und Aktionen

2) Festlegen von *Startzuständen*: Weltzustände mit Starteigenschaften

3) Festlegen einer Nachfolgefunktion zur Überführung (Transformation) von Weltzuständen durch geeignete *Operatoren* / *Aktionen*

4) Festlegen eines *Zieltests* zur Überprüfung, ob die Beschreibung eines Zustands einem Zielzustand entspricht

5) Bestimmung von *Pfadkosten*: was kostet die Ausführung einer Aktion und folglich die Ausführung einer Aktionsfolge (s. Folgefolie)

Bemerkung: Die Art der Problemformulierung kann großen Einfluss auf die Schwierigkeit der Lösung haben.

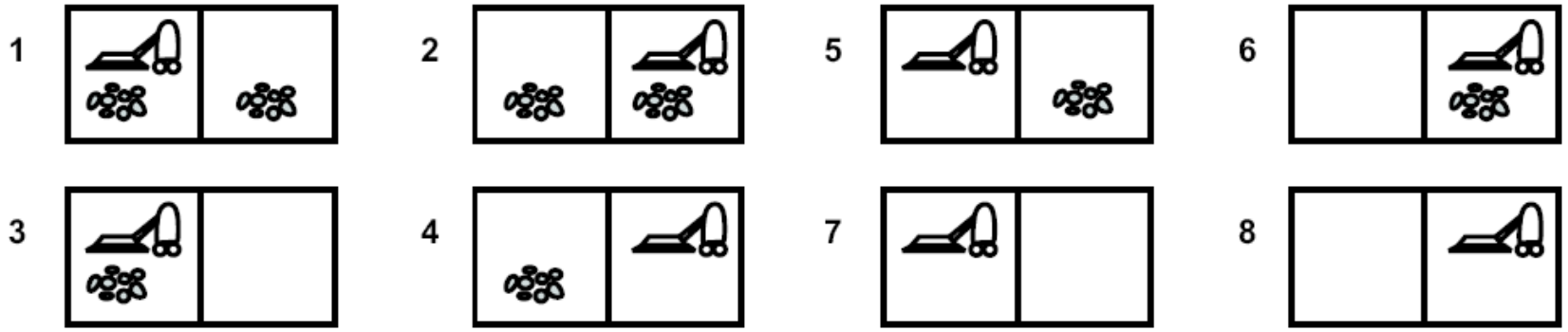
Kosten

- Pfad: Folge von Aktionen, die von einem Zustand zu einem anderen führen
- Pfadkosten: Kostenfunktion *g* über Pfaden
  - entspricht i. A. der Summe der Kosten der einzelnen Aktionen
- Lösung: Pfad von einem Anfangs- zu einem Zielzustand
- Suchkosten: Zeit- und Speicherbedarf der Suche, um eine Lösung zu finden
- Gesamtkosten: Suchkosten + Pfadkosten, also Kosten
  - *für das Suchen* (Suchkosten, *Offline*-Kosten) +
  - *für die Ausführung* (Pfadkosten, *Online*-Kosten)

Formulate-Search-Execute-Schema

Problemformulierung für die abstrakte Staubsaugerwelt

- *Weltzustandsraum:* 2 Agentenpositionen, in beiden Räumen Schmutz oder kein Schmutz $\rightsquigarrow$ $2 \cdot 2^2 = 8$ Weltzustände



- *Startzustände:* jeder beliebige Zustand
- *Aktionen:* links (*L*), rechts(*R*), saugen(*S*)

  (Aktionen *L* im linken Raum, *R* im rechten Raum und *S* in gesaugtem Raum führen wieder zum Ausgangszustand)
- *Zielzustände:* kein Schmutz in beiden Räumen $\rightsquigarrow$ Zustände 7 und 8
- *Pfadkosten:* pro Aktion 1 Kosteneinheit

Beispiel für eine Lösung: $Z_1 \rightarrow_S Z_5 \rightarrow_R Z_6 \rightarrow_S Z_8$ bzw. als Aktionsfolge: *S, R, S*

Problemtypen: Wissen über Zustände und Aktionen

- Einzustandsproblem
  - vollständig beobachtbare Umwelt und deterministische Aktionen

    $\Rightarrow$ Agent weiß immer eindeutig, in welchem Weltzustand er ist und in welchen Zustand er durch jede Aktion kommen wird.

- Mehrzustandsproblem (in Russel/Norvig: sensorloses Problem)
  - nur partiell/vollständig unbeobachtbare Umwelt (eingeschränkte oder keine Sensoren), aber deterministische Aktionen

    $\Rightarrow$ Agent weiß nur, in welcher *Menge von* Weltzuständen er ist.

- Kontingenzproblem
  - partiell beobachtbare Umwelt und nichtdeterministische Aktionen

    $\Rightarrow$ keine eindeutige Aktionsfolge zur Lösung a priori bestimmbar, da Abhängigkeiten bzgl. der tatsächlichen Zustände vorliegen

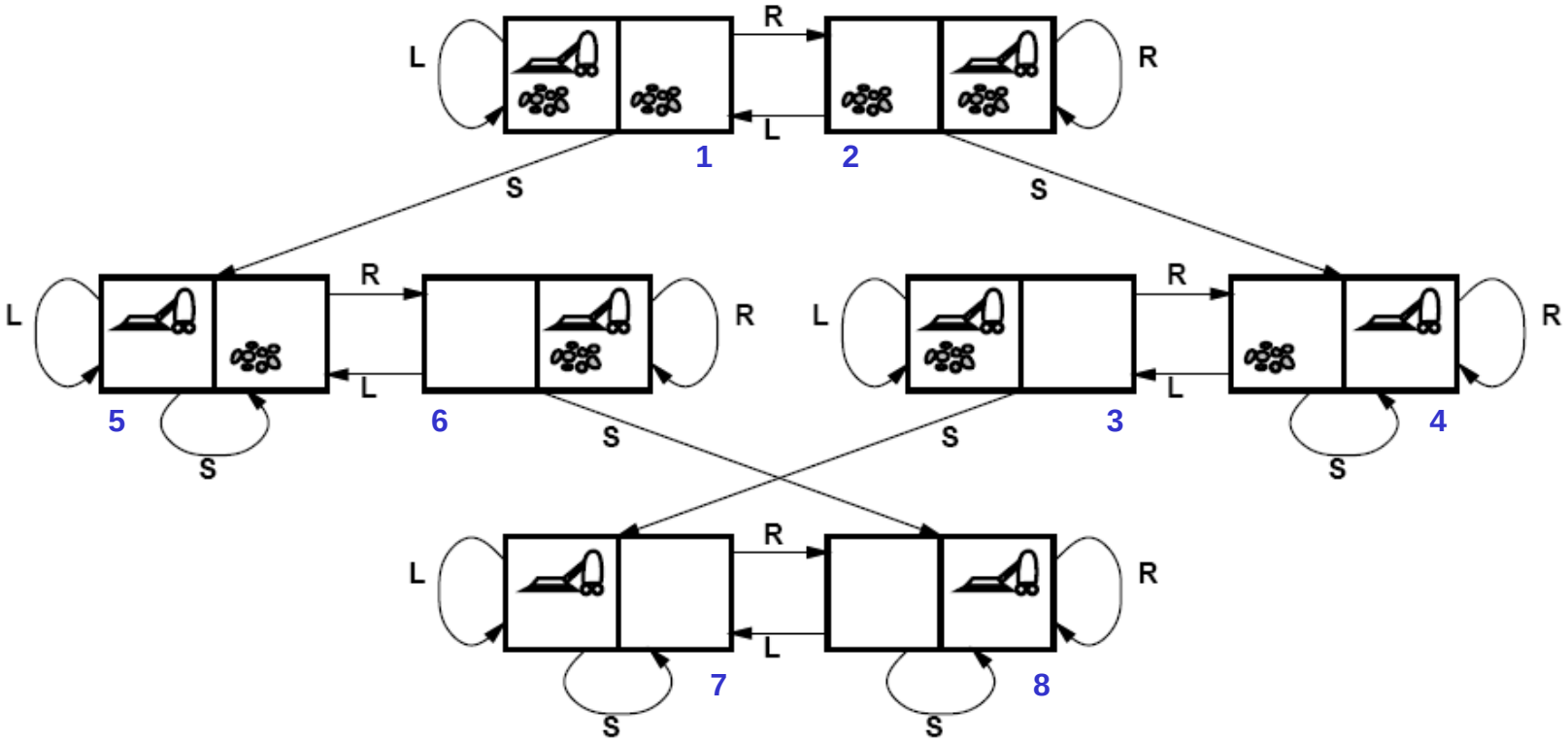
    $\Rightarrow$ erfordert gezielte Beobachtung, um Abhängigkeit aufzulösen.

- Explorationsproblem
  - Umwelt und Effekte der Aktionen sind vollständig oder partiell unbekannt

    $\Rightarrow$ Schwerster Fall! $\rightarrow$ spätere Kapitel

Die Staubsaugerwelt als Einzustandsproblem

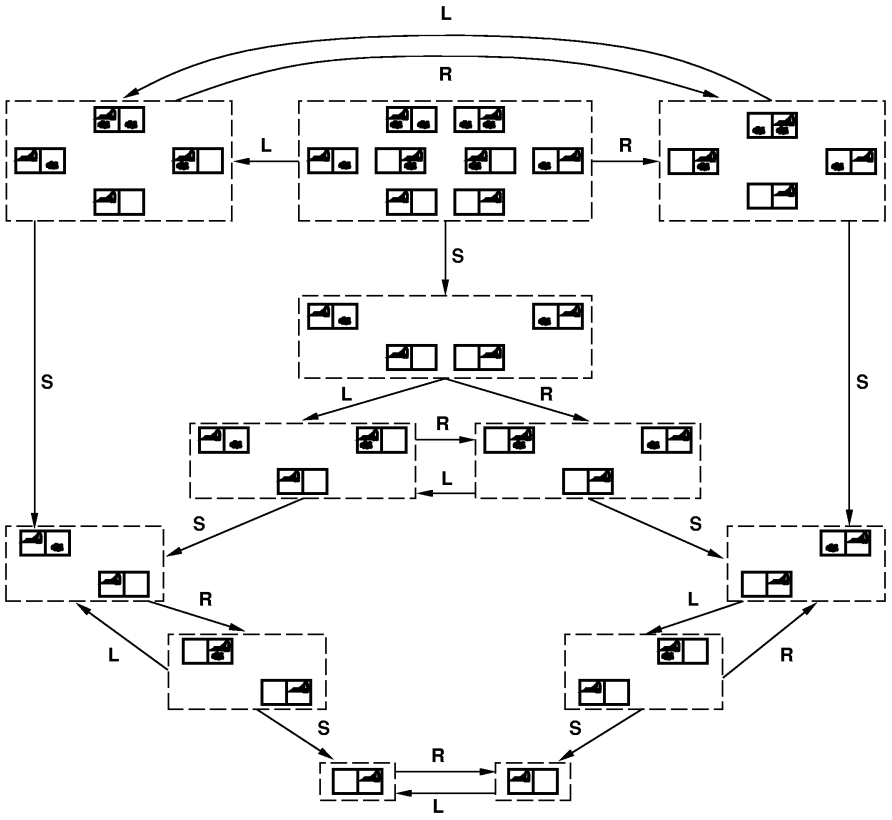
Sind das Wissen um die Welt und die Aktionen *vollständig*, weiß der Staubsauger-Agent immer, wo er ist und ob Schmutz vorliegt. Problemlösen reduziert sich dann auf die Suche nach einem Pfad von einem Anfangszustand zu einem Zielzustand.



Zustände für die Suche: Die Weltzustände 1 – 8.

Die Staubsaugerwelt als Mehrzustandsproblem

Die Aktionen des Staubsaugers sind verlässlich, aber er besitzt keine Sensoren und weiß somit von Beginn an nicht, wo er ist und wo Schmutz ist. Trotzdem kann er das Problem lösen. Zustände sind dann sog. *Glaubenszustände* (*Belief States*):



Ein Glaubenszustand beschreibt die *Menge aller möglichen* physischen Zustände.

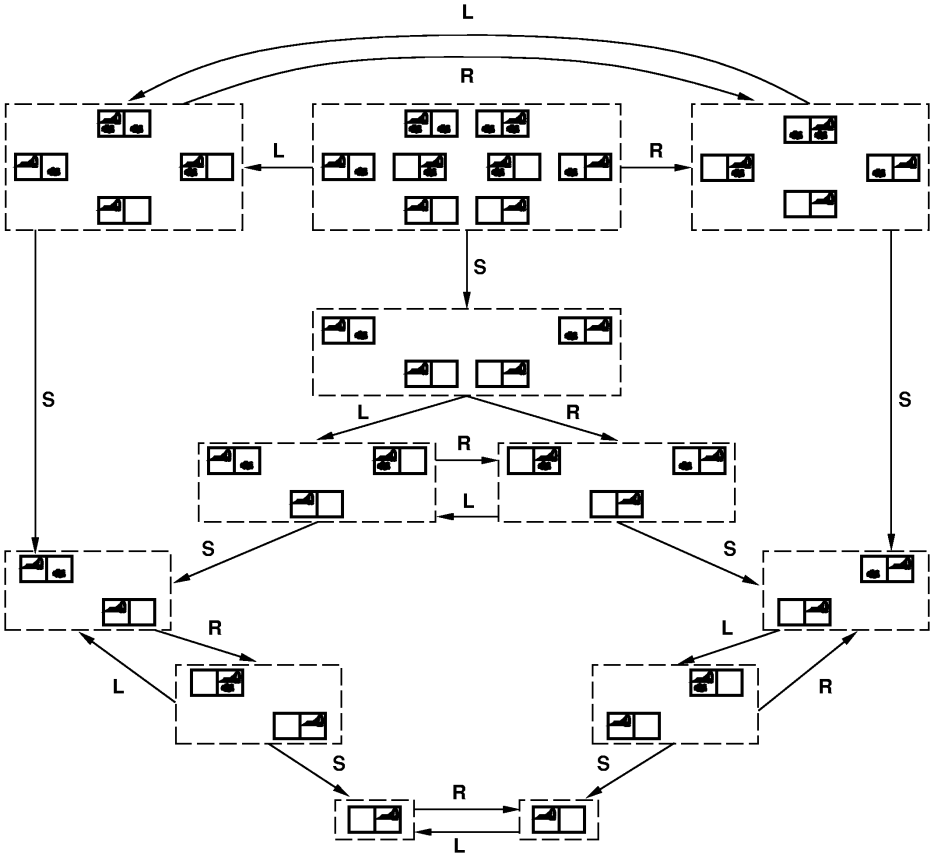
Bei vollständig beobachtbarer Umwelt beschreibt dagegen jeder Glaubenszustand genau einen physischen Zustand.

Zustände für die Suche: hier Zahl der erreichbaren Zustände = 12.

Prinzipiell Worst Case: Potenzmenge der acht Weltzustände: $2^8 = 256$ Zustände.

Die Staubsaugerwelt als Mehrzustandsproblem

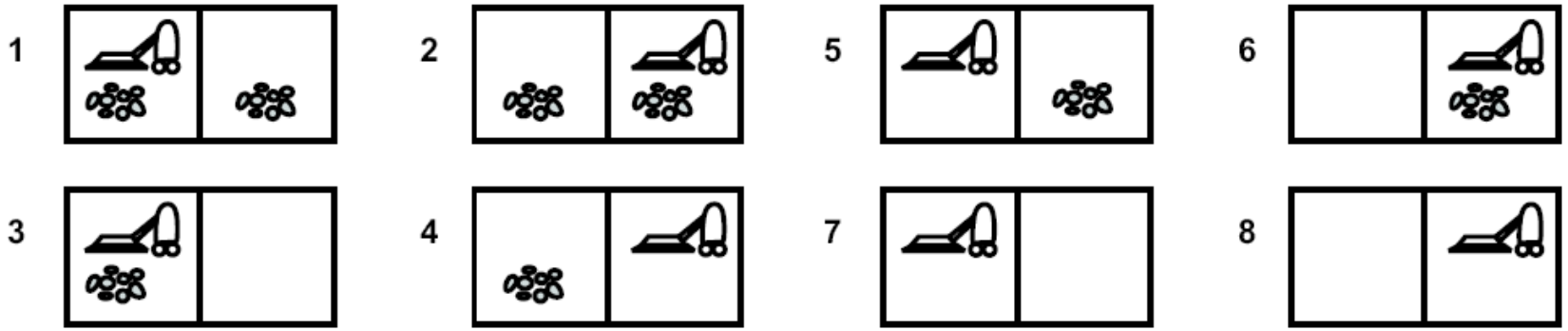
Eine Lösung des Mehrzustandsproblems erzwingt quasi die sukzessive Reduktion der Gesamtmenge aller möglichen Weltzustände über kleinere Potenzmengen letztlich auf solche Potenzmengen, die nur aus phys. Zielzuständen bestehen (hier Zustände 7 und 8).



Lösungen hier z.B. die Aktionssequenzen *R,S,L,S* und *L,S,R,S*.

Die Staubsaugerwelt als Mehrzustandsproblem mit Unsicherheit

- Der Staubsauger kann den Weltzustand nicht erfassen (sensorlos) und die Effekte der Aktionen sind unsicher.
- Bspl.: Die Aktion *Saugen* bewirkt *im Defektfall* den Verlust von Schmutz auf sauberer Fläche. Ansonsten wird vorhandener Schmutz beseitigt.



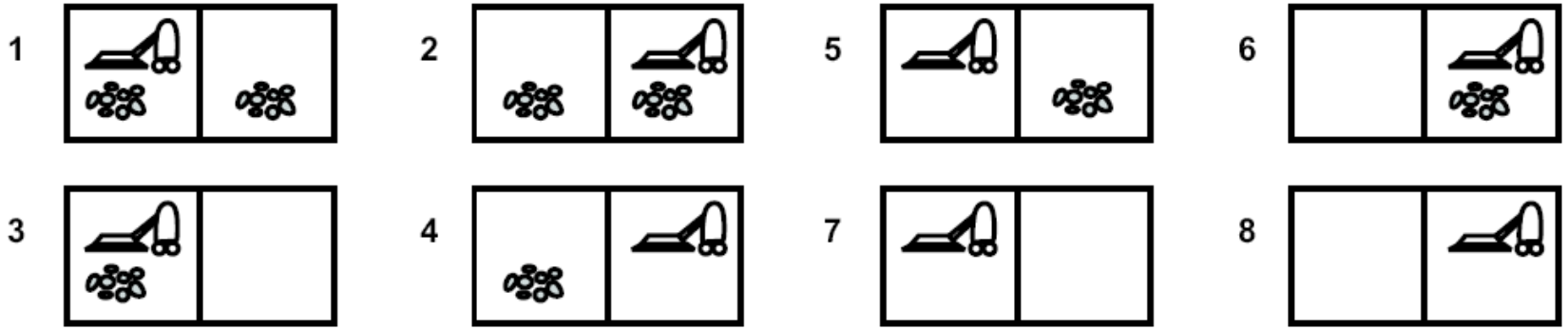
- Bspl.: vom Zustand {4} wird über Aktion *Saugen* der Zustand {2,4} erreicht.
- Vom Startzustand {1,2,3,4,5,6,7,8} führt die Aktion *Saugen* wieder zum selben Zustand {1,2,3,4,5,6,7,8}. Die Menge der mögl. Zustände ist nicht reduzierbar.
- Das Problem ist somit zwar formal gefasst und es existiert auch ein Pfad zu einer Lösung, aber die Umsetzung ist nicht einlösbar, da der Agent einfach nicht wissen kann, was seine Aktionen tatsächlich bewirken.

Die Staubsaugerwelt als Kontingenzproblem

- Das Wissen um die Umwelt und die Aktionen sei *unsicher*, aber der Staubsauger kann seine Welt nach einer Aktion durch Sensoren neu erfassen.

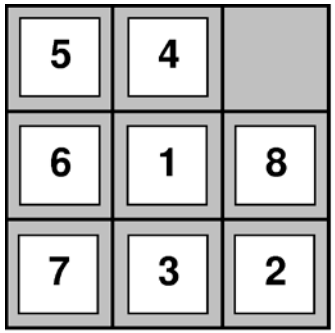
➢ Er muss seine Aktionsfolge *vom tatsächlichen Effekt der Einzelaktionen* abhängig machen. Als Lösung entsteht ein Baum von Aktionsfolgen anstelle eines einzelnen Pfades.

- Bspl.: Die unsichere Aktion *Saugen* bewirke im Defektfall weiterhin den Verlust von Schmutz auf sauberer Fläche. Ansonsten wird vorhandener Schmutz beseitigt.

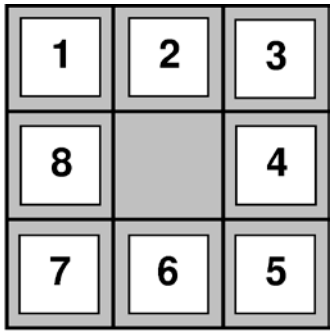


- Vom Zustand {1,3} werde die Aktionsfolge S,R,S erzeugt. Aktion S führt zu {5,7}, Aktion R zu {6,8}. Liegt nach R der Zustand 6 vor, wird durch S der Zielzustand 8 erreicht. Andernfalls wird der Zustand 6 erreicht. Die Aktionsfolge scheitert.
- Die bedingte Aktionsfolge S, R, *if* (R,schmutzig) *then* S wäre die Lösung.

Problemformulierung: Spielproblem *8-er-Puzzle*



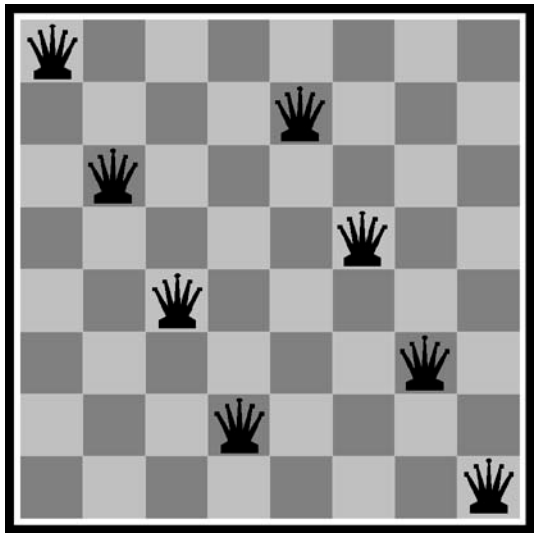
Start State



Goal State

- Zustände:
  - Beschreibung der Lage jedes der 8 Felder und aus Effizienzgründen des leeren Feldes.
- Operatoren:
  - „Verschieben“ des leeren Feldes nach links, rechts, oben und unten.
- Zieltest:
  - Entspricht aktueller Zustand dem rechten Bild?
- Pfadkosten
  - Jeder Schritt kostet 1 Einheit.

Problemformulierung: Spielproblem *8-Damen-Problem* (1)



Bemerkung: Beispiel stellt keine Lösung dar.

- Zieltest:
  - 8 Damen auf dem Brett, keine angreifbar.
- Pfadkosten: 0 ⇝ nur die Lösung interessiert!
- Darstellung 1:
  - Zustände: beliebige Anordnung von 0 bis 8 Damen.
  - Operatoren: setze eine der Damen aufs Brett.
  - ➢ Problem: $64 \cdot 63 \cdot ... \cdot 57 \approx 3 \cdot 10^{14}$ Aktionsfolgen zu untersuchen!

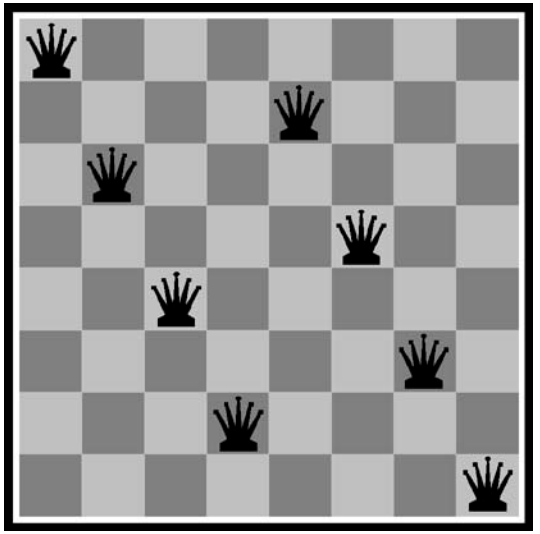
Problemformulierung: Spielproblem *8-Damen-Problem* (2)

- Darstellung 2:
  - Zustände: Anordnung von 0 bis 8 Damen in unangreifbarer Stellung.
  - Operatoren: Setze jede Dame *möglichst links* unangreifbar auf das Brett.

➢ Vorteil: sehr viel weniger Aktionsfolgen für das 8-Damen-Problem: 2057.

➢ Problem: immer noch 10^{52} Folgen für das 100-Damen-Problem (10^{400} in Darstg. 1).

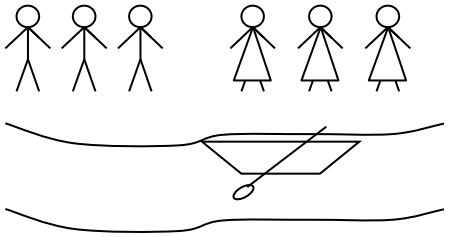
Skalierungs-problem



⇝ noch effizientere Darstellungen und effiziente Verfahren nötig!

Problemformulierung: Spielproblem *Missionare und Kannibalen*

Informelle Problembeschreibung:



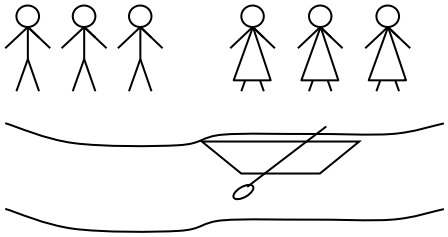
- An einem Fluss haben sich 3 Kannibalen und 3 Missionare getroffen, die alle den Fluss überqueren wollen.
- Es steht ein Boot zur Verfügung, das maximal zwei Personen aufnimmt.
- Es darf keine Situation eintreten, in der an einem Ufer Missionare *und* Kannibalen stehen und die Kannibalen dabei zahlenmäßig überlegen sind.

➢ Finde eine Aktionsfolge, die alle Missionare und Kannibalen wohlbehalten an das andere Ufer bringt.

Formalisierung des *MuK-Problems*

- **Zustände:** Tripel (m,k,b) mit $0 \leq m, k \leq 3$ und $0 \leq b \leq 1$

  für Variablen m, k und b für die Zahlen der Missionare, Kannibalen bzw. Boote am Ausgangsufer.



- **Anfangszustand:** (3,3,1).

- **Aktionen:** Von jedem Zustand aus können mit dem Boot entweder (1) ein Missionar, (2) ein Kannibale, (3) zwei Missionare, (4) zwei Kannibalen oder (5) einer von jeder Sorte übersetzen ⇝ 5 Operatoren

  - Beachte: nicht jeder Zustand ist so erreichbar [z.B. (0,0,1)]

    und einige sind nicht zulässig

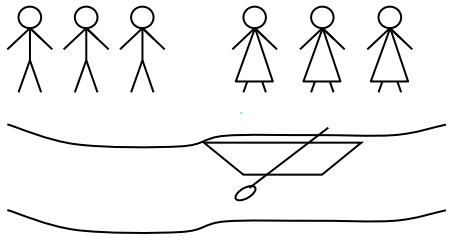
- **Endzustand:** (0,0,0).

- **Pfadkosten:** 1 Einheit pro Flussüberquerung.

Lösung des *MuK-Problems* durch Suche

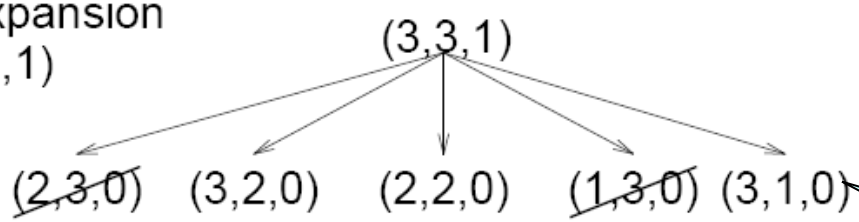
Ausgehend vom ***Anfangszustand*** schrittweise alle ***Folgezustände*** erzeugen

→ ***Suchbaum***:



(a) Anfangszustand (3,3,1)

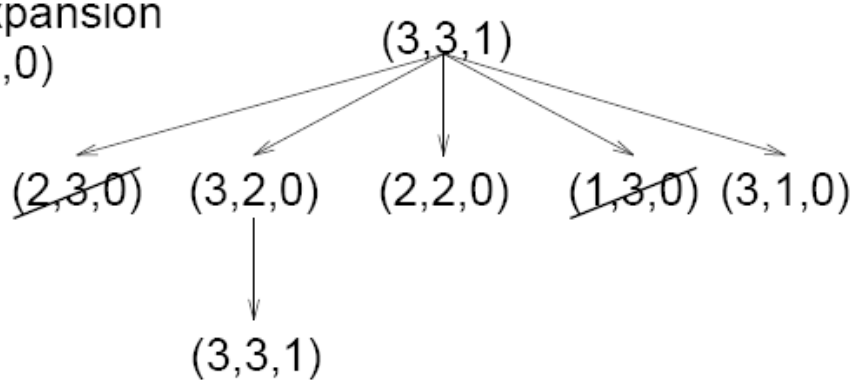
(b) nach Expansion von (3,3,1)



Erste Ableitungs-ebene hat im Suchbaum die ***Tiefe*** *1*

Anfangszustand und die Folge-zustände bilden die Knoten des Suchbaums

(c) nach Expansion von (3,2,0)



Allgemeine Suchprozedur

```
function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

(1)

(2)

Beachte: Zieltest erfogt auf den zur Expansion gewählten Knoten,

nicht auf den gerade erreichten Knoten *(resulting nodes)*!

---

(1) General-Search entspricht Tree-Search in 2. Auflage (s. Kommentar in übernächster Folie).

(2) Expansion = Erzeugen von Folgezuständen

Implementierung des Suchbaums

Knoten sind mehr als Zustände!

Datenstruktur für Knoten im Suchbaum umfasst:

- *State*: korrespondierender Zustand im Zustandsraum
- *Parent-Node*: Vorgängerknoten
- *Operator*: Operator/Aktion, der den aktuellen Knoten erzeugt hat
- *Depth*: Tiefe im Suchbaum = Anzahl der Knotenexpansionen entlang des Pfades vom Ausgangsknoten aus
- *Path-Cost*: Pfadkosten bis zu diesem Knoten

Funktionen zur Knotenexpansion durch eine *Warteschlange* (*Queue*):

- *Make-Queue(Elements)*: Erzeugt eine Queue
- *Empty?(Queue)*: Testet auf Leerheit
- *Remove-Front(Queue)*: Gibt erstes Element zurück
- *Queuing-Fn(Queue, Elements)*: Fügt neue Elemente ein (verschiedene Möglichkeiten)

Allgemeine Suche ... konkretisiert

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

   nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
   loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
   end
```

(1)

- *Queuing Function* implementiert *strategy*
- *nodes* implementiert Warteschlange
- *state(node)* liefert Zustandsbeschreibung vom Knoten *node*
- *EXPAND(node,OPERATORS[problem])* erzeugt alle Nachfolgeknoten von *node* über die zulässigen Operatoren

(1) General-Search entspricht Tree-Search in 2. Auflage, ist aber stimmiger mit der Queue-Terminologie.

Bewertung von Suchstrategien

Kriterien:

- Vollständigkeit: Wird immer eine Lösung gefunden, sofern es eine gibt?
- *Optimalität:* Findet das Verfahren immer die *beste* Lösung? Beste Lösung: minimale Suchtiefe bzw. minimale Pfadkosten
- *Zeitkomplexität:* Wie lange dauert die Suche nach einer Lösung (im schlechtesten Fall) ?
- *Platzkomplexität:* Wie viel Speicher benötigt die Suche (im schlechtesten Fall)?

Grundsätzliche Ansätze für Suchstrategien

Strategien:

- Uninformierte oder blinde Suche: keine problemspezifische Information!
  - Breitensuche, uniforme Kostensuche, Tiefensuche
  - tiefenbeschränkte Suche, iterative Tiefensuche
  - bidirektionale Suche

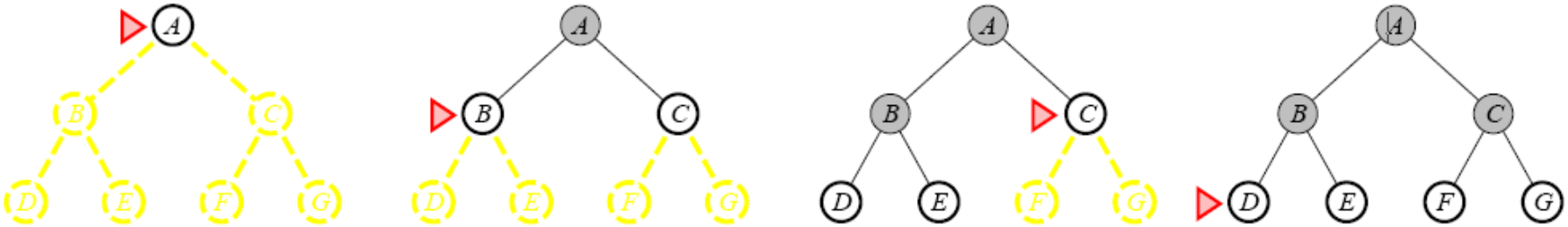
- Informierte oder heuristische Suche
  - ⇝ nächste Vorlesung

Breitensuche (1)

Expandiere Knoten in der Reihenfolge, in der sie erzeugt werden

→ *Queue-Fn = Enqueue-at-end*

→ *FIFO*-Warteschlange (*First-in-First-out)*



- Findet immer die flachste Lösung ⇝ vollständig (bei endlicher *Lösungstiefe,* d.h. Tiefe eines Zielknotens)
- Die Lösung ist optimal, wenn die Pfadkostenfunktion eine nichtfallende Funktion der Knotentiefe ist (z.B. wenn jede Aktion identische, nicht-negative Kosten hat).

Breitensuche (2)

- Allerdings sind die Kosten sehr hoch. Sei b der *maximale Verzweigungs-faktor* und $d > 0$ die Tiefe des kürzesten *Lösungspfads*. Dann werden maximal $O(b^{d+1})$ Knoten erzeugt, d.h.

$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$

Schlechtester Fall: Alle Knoten der Tiefe d – bis auf den Zielknoten – werden expandiert

Beispiel: b = 10, 10.000 Knoten/Sec; 1.000 Bytes/Knoten:

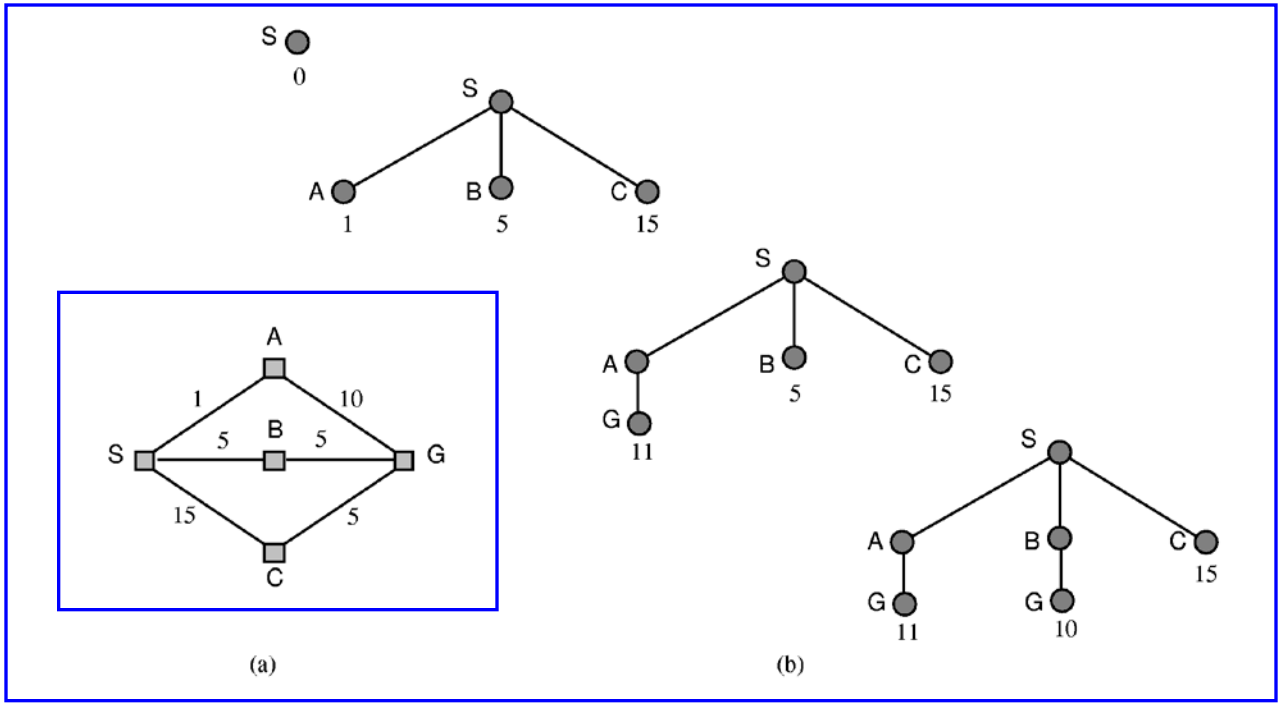
Tiefe	Knoten	Zeit	Speicher
2	1.100	0,11 Sekunden	1 Megabyte
4	111.100	11 Sekunden	100 Megabyte
6	10^7	19 Minuten	10 Gigabyte
8	10^9	31 Stunden	1 Terabyte (10^{12} Byte)
10	10^{11}	129 Tage	100 Terabyte (10^{14} Byte)
12	10^{13}	35 Jahre	10 Petabyte (10^{16} Byte)
14	10^{15}	3.523 Jahre	1 Exabyte (10^{18} Byte)

Uniforme Kostensuche

Abwandlung der Breitensuche:

immer Expansion der Knoten *n* mit den geringsten Pfadkosten *g(n)*.

Bspl.: Weg von *S* nach *G*:



Findet immer die günstigste Lösung, falls $\forall\, n$: $g(succ(n)) \geq g(n)$. Zeit- und Speicherkomplexität: $O(b^{\lceil C^*/\varepsilon \rceil})$ mit Kosten C* für optimalen Pfad und geringsten Aktionskosten ε.

Tiefensuche (1)

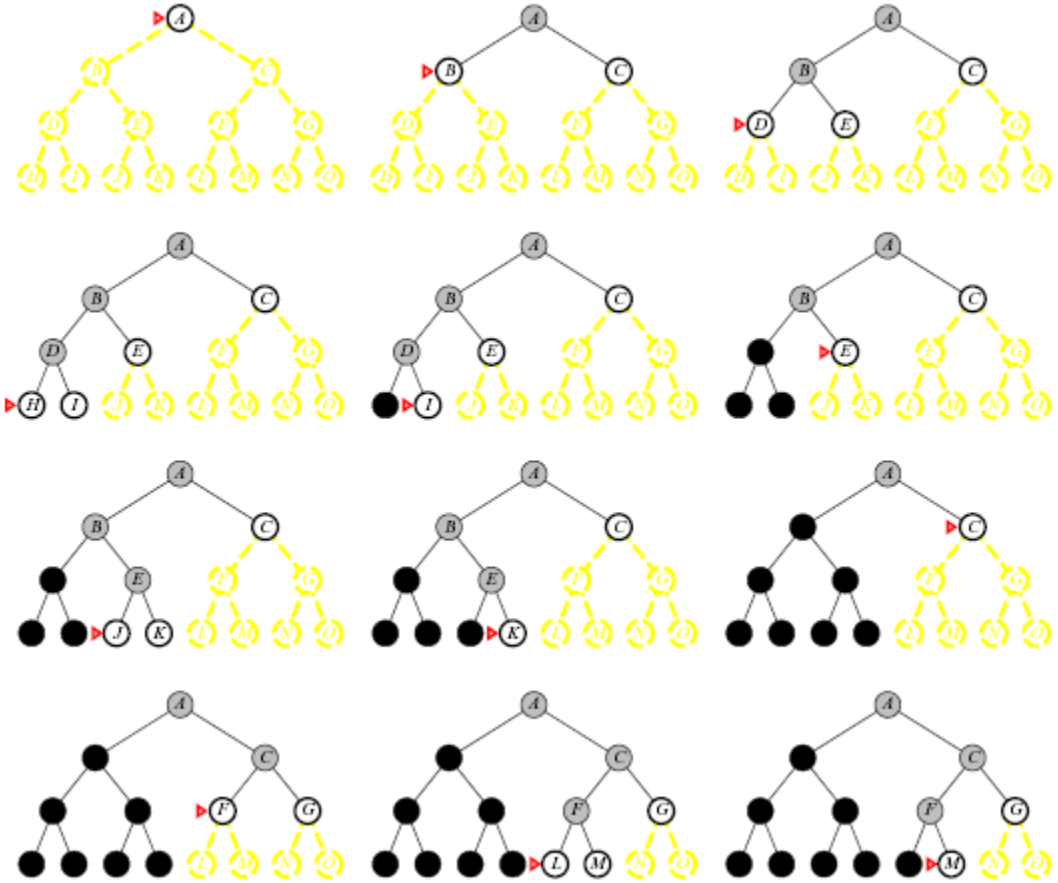
Expandiere immer einen nicht expandierten Knoten mit maximaler Tiefe

→ *Queue-Fn = Enqueue-at-front*

→ *LIFO*-Warteschlange für *Last-in-First-out*

Beispiel für Baum mit max. Suchtiefe *m* = 3:

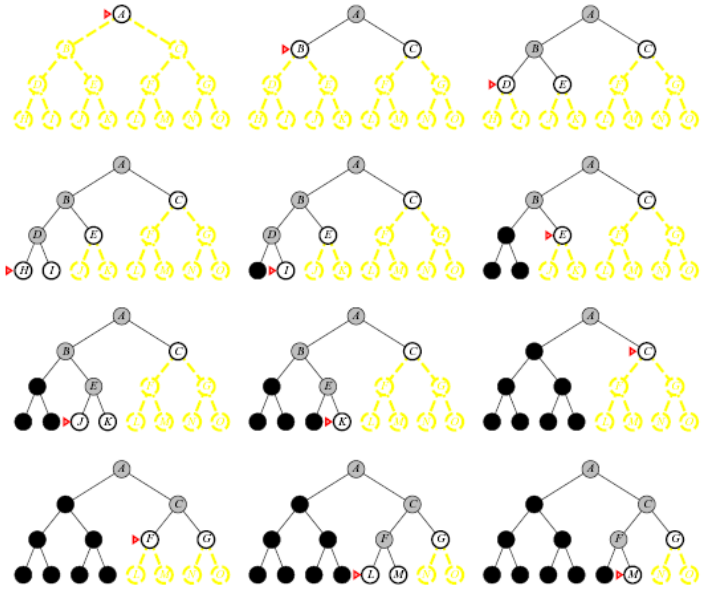
Abgearbeitete Knoten (schwarz) können aus dem Speicher entfernt werden!



Tiefensuche (2)

- Benötigt nur ***Speicher*** für $b \cdot m + 1$ Knoten bei max. Verzweigung *b* und max. *Suchtiefe m.*

  → Speicherkomplexität O(b·m)



- Die sog. Backtracking-Variante generiert immer nur *einen* Nachfolger. Der Vorgänger merkt sich dafür, welche Knoten alternativ als nächste zu erzeugen wären.

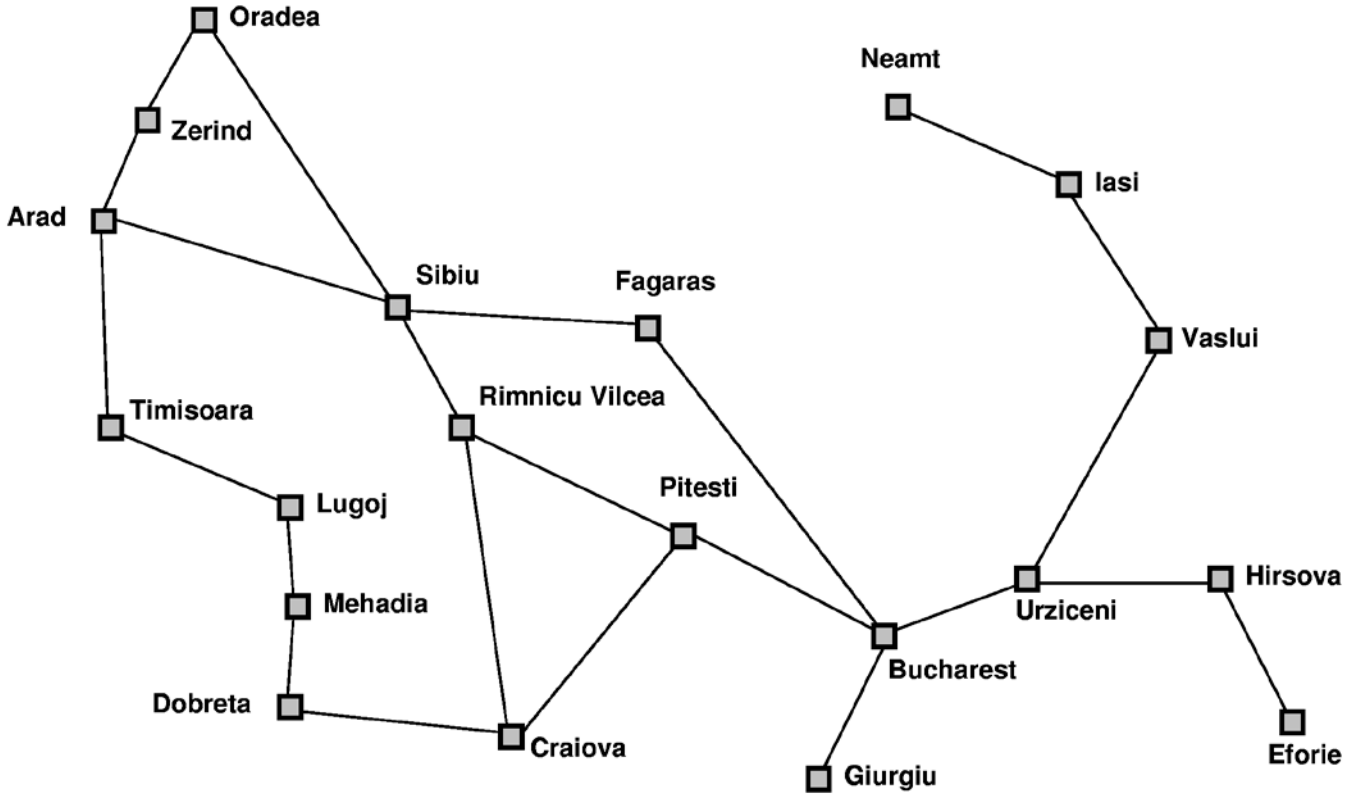
  → *Speicherkomplexität O(m)*

- Im schlechtesten Fall ist *Zeitkomplexität* $O(b^m)$.
- Kann die Lösung bei unendlich tiefen Bäumen verfehlen → unvollständig!
- Kann die optimale Lösung verfehlen → nicht optimal!

Tiefenbeschränkte Suche

Es wird nur bis zu einer vorgegebenen Pfadlänge Tiefensuche durchgeführt.

z.B. *Routenplanung*: bei max. *n* Teilstrecken für alle Städteverbindungen ist Suchtiefe $m > n$ sicher nicht sinnvoll.



Im Bspl. reicht max. Suchtiefe $m = 9$, da alle Städtepaare über max. 9 Teilstrecken verbunden sind ⇝ *Durchmesser* des Problems

Iterative Tiefensuche (1)

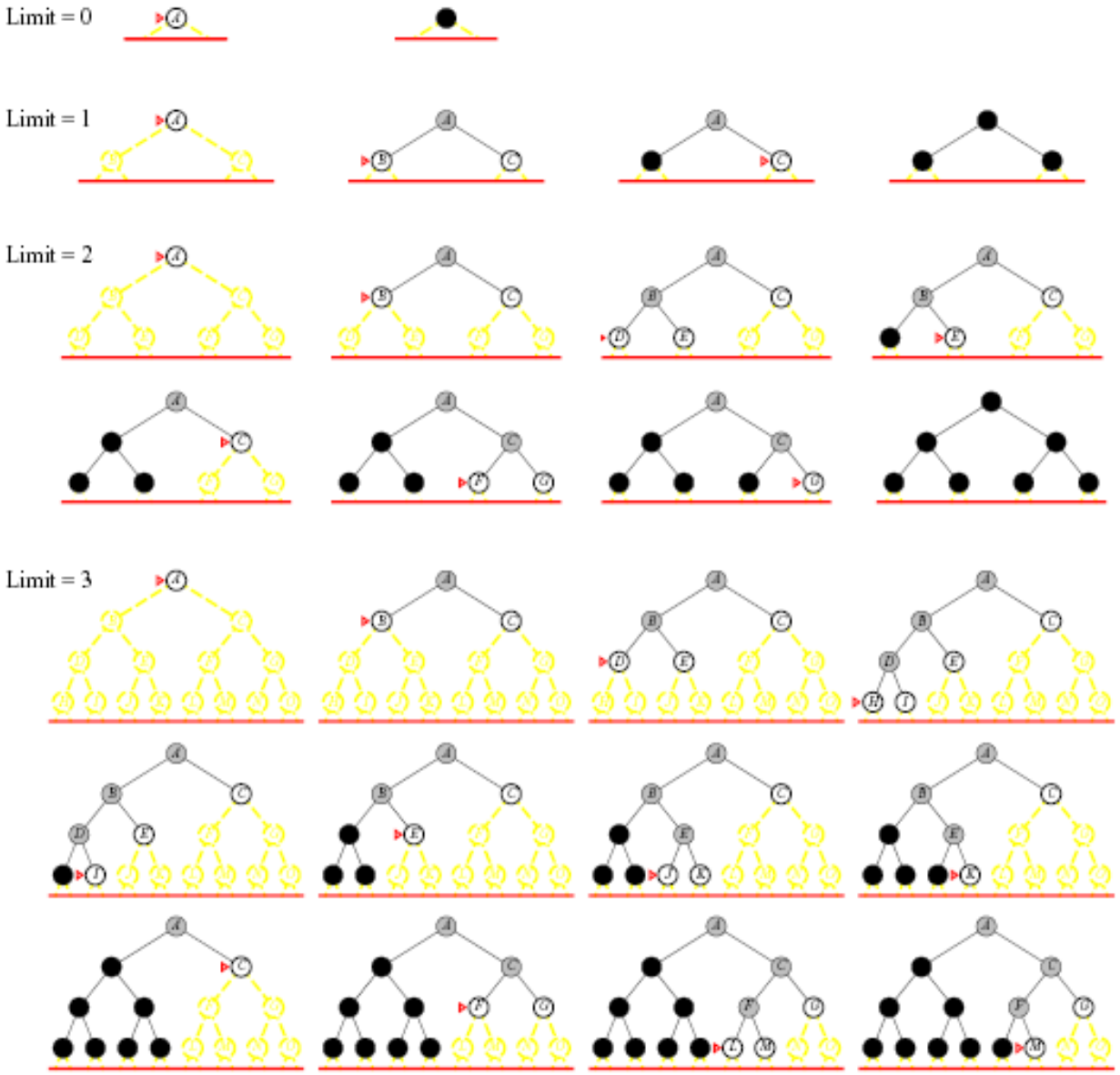
Iterative Tiefensuche

- kombiniert Tiefen- und Breitensuche,
- ist optimal und vollständig wie Breitensuche, braucht aber weniger Speicherplatz.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
    inputs: problem, a problem

    for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH( problem, depth) succeeds then return its result
    end
    return failure
```

Beispiel



Iterative Tiefensuche (2)

Zahl der erzeugten Knoten allgemein:

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

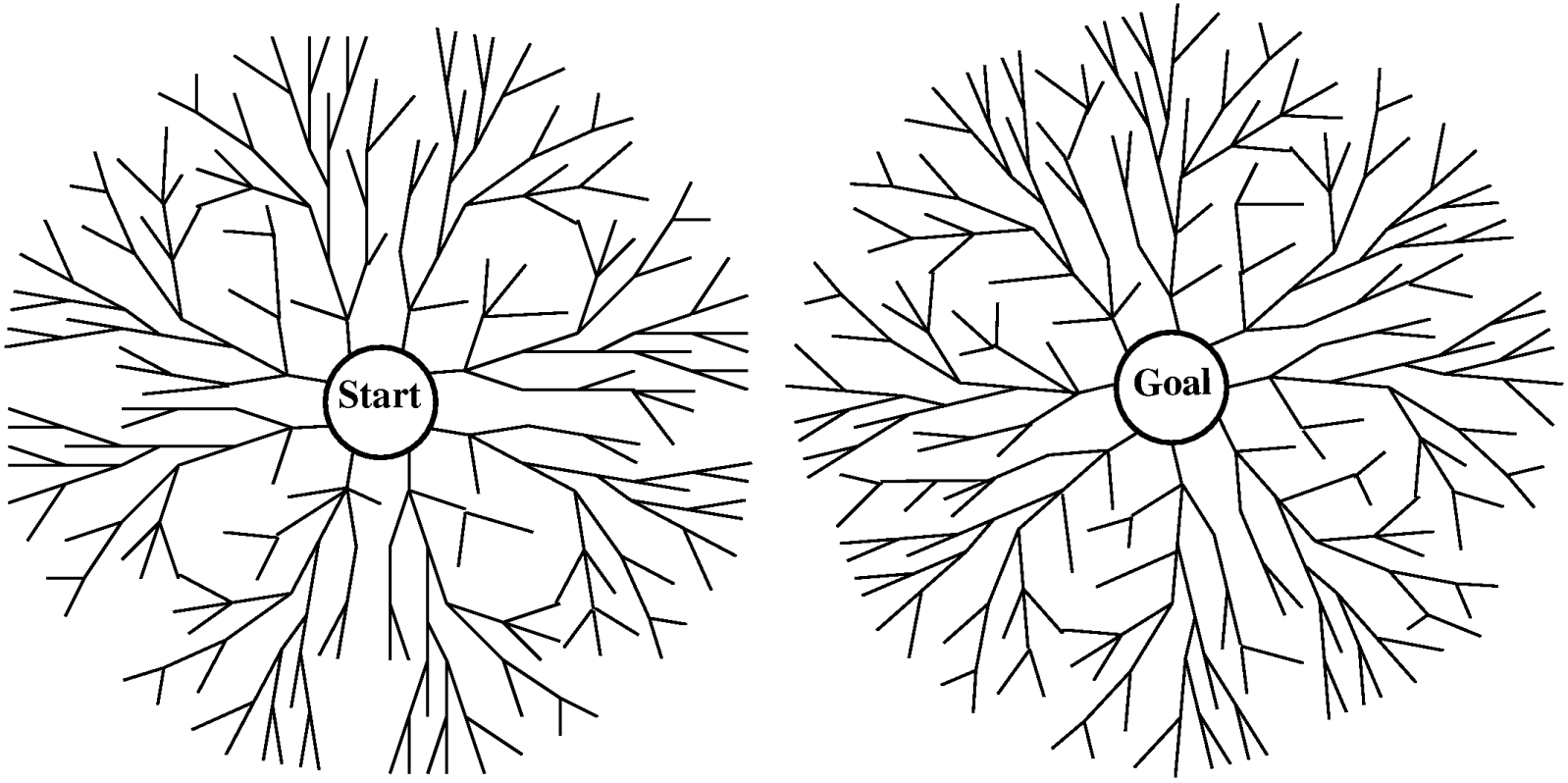
Beispiel für $b = 10$, $d = 5$:

Breitensuche	10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100
Iterative Tiefensuche	50 + 400 + 3.000 + 20.000 + 100.000 = 123.450

Zeitkomplexität: $O(b^d)$, *aber Platzkomplexität*: $O(b \cdot d)$.

→ Iterative Tiefensuche ist bzgl. der Zeitkomplexität in derselben Größenordnung wie die Breitensuche und i.allg. die bevorzugte Suchmethode bei großen Suchräumen mit unbekannter maximaler Suchtiefe.

Bidirektionale Suche (1)



- Sofern Vorwärts- und Rückwärtssuche symmetrisch sind, erreicht man Suchzeiten gemäß der Argumentation $O(2 \times b^{d/2}) = O(b^{d/2})$.
- Z.B. bei Breitensuche mit $b = 10$, $d = 6$ nur 22.200 Knoten statt 11.111.100!

Bidirektionale Suche (2)

Zu beachten ist:

- Die Operatoren sind nicht immer oder nur sehr schwer umkehrbar (Berechnung der Vorgängerknoten).
- In manchen Fällen gibt es sehr viele Zielzustände, die nur unvollständig beschrieben sind.
- Man braucht effiziente Verfahren, um zu testen, ob sich die Suchverfahren ”getroffen“ haben.
- Welche Art der Suche wählt man für jede Richtung (im Bild: Breitensuche, die z.B. selbst wieder hohe Komplexität hat)?

Vergleich der Suchstrategien

Kriterien:

- Zeitkomplexität
- Platzkomplexität
- Optimalität
- Vollständigkeit

Variable:

b: max. Verzweigungsfaktor, *d*: Tiefe der Lösung,

m: maximale Tiefe des Suchbaums (Suchtiefe), *l*: Tiefenlimit

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Beispiele für reale Probleme

- Routenplanung, Finden kürzester Pfade:
  - im Prinzip einfach (polynomiell lösbares Problem).
  - Schwierigkeit bei unbekannten, sich dynamisch verändernden Pfadkosten.
- Planung von Rundreisen (Tourenplanung) ($\rightarrow$ TSP):
  - eines der prototypischen NP-vollständigen Probleme.
- VLSI Layout:
  - auch ein NP-vollständiges Problem.
- Roboter Navigation (mit vielen Freiheitsgraden):
  - Schwierigkeit nimmt mit der Anzahl der Freiheitsgrade extrem zu.
  - weitere mögliche Komplikationen: Fehler bei Wahrnehmungen, unbekannte Umgebungen.
- Montageplanung:
  - Planung des Zusammenbaus von komplexen Objekten.
- Internetsuche:
  - Suche nach Antworten und verwandter Information im Internet.

Zusammenfassung

- Bevor ein Agent beginnen kann, eine Lösung zu suchen, muss er sein Ziel und darauf aufbauend sein Problem definieren.
- Eine Problembeschreibung umfasst fünf Komponenten: Zustandsraum, Anfangszustand, *Operatoren*, *Zieltest* und *Pfadkosten*. Ein *Pfad* vom Anfangszustand zu einem Zielzustand ist eine *Lösung* im Sinne einer *Aktionsfolge*.
- Es existiert ein *genereller Suchalgorithmus*, der benutzt werden kann, um Lösungen zu finden. Spezifische Varianten des Algorithmus benutzen verschiedene *Suchstrategien*.
- Suchalgorithmen werden auf Basis der Kriterien *Vollständigkeit*, *Optimalität*, *Zeit-* und *Platzkomplexität* beurteilt.