INF FAKULTÄT FÜR INFORMATIK

Intelligente Systeme

Heuristische Suchalgorithmen

Prof. Dr. R. Kruse C. Braune C. Moewes

{kruse,cmoewes,russ}@iws.cs.uni-magdeburg.de

Institut für Wissens- und Sprachverarbeitung

Fakultät für Informatik

Otto-von-Guericke Universität Magdeburg

Warum heuristische Suchalgorithmen?

- Suchprobleme werden häufig durch eine Baumsuche gelöst
- eine uninformierte Suche muss im schlechtesten Fall alle Knoten des Baumes expandieren
- unter Ausnutzung von Problemwissen (Mutmaßungen/Heuristiken) kann der Rechenaufwand meistens reduziert werden

Übersicht

Baumsuchalgorithmen

grundlegende Idee:

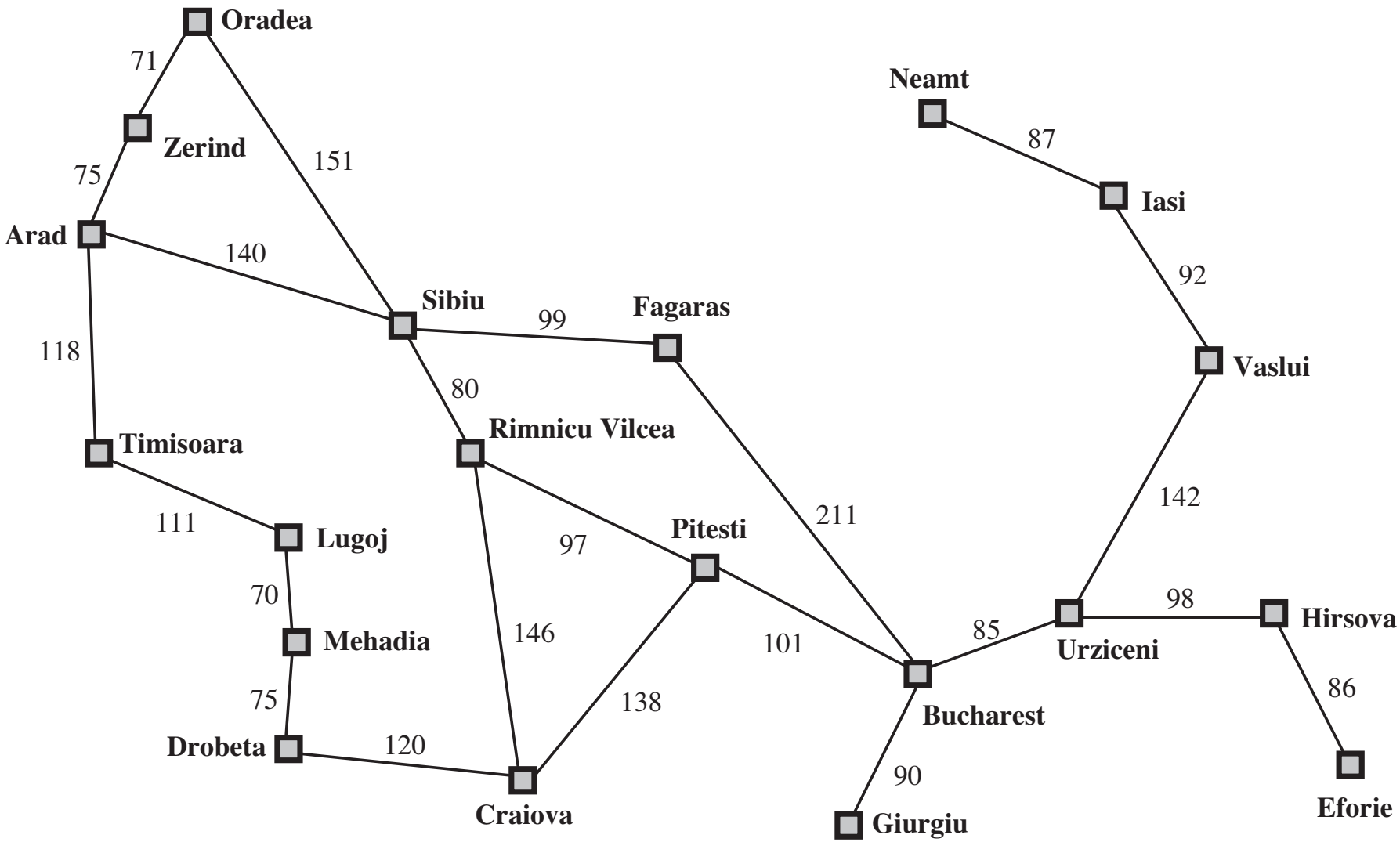
- offline, sogenannte simulierte Durforstung des Zustandsraums
- Erzeugung von Nachfolgern bereits erkundeter Zustände (sog. expandierte Zustände)

TREE-SEARCH

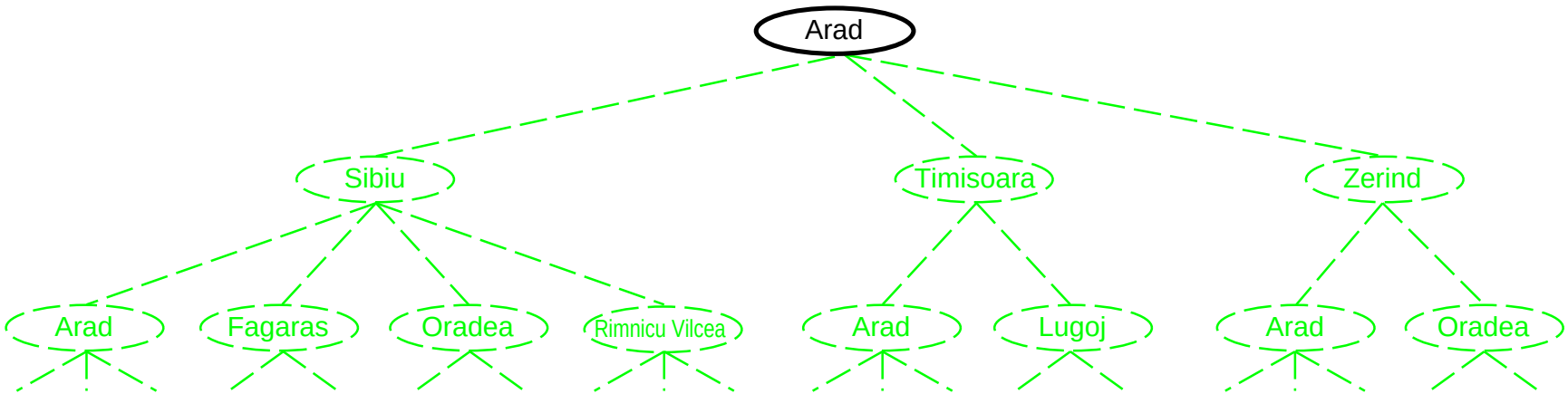
Eingabe: Problembeschreibung *problem*, Vorgehensweise *strategy*
Ausgabe: Lösung oder Fehler

```
 1: initialize search tree using initial state of problem
 2: while true {
 3:     if there are no candidates for expansion {
 4:         return failure
 5:     }
 6:     choose leaf node for expansion according to strategy
 7:     if node contains goal state {
 8:         return corresponding solution
 9:     } else {
10:         expand node and add resulting nodes to search tree
11:     }
12: }
```

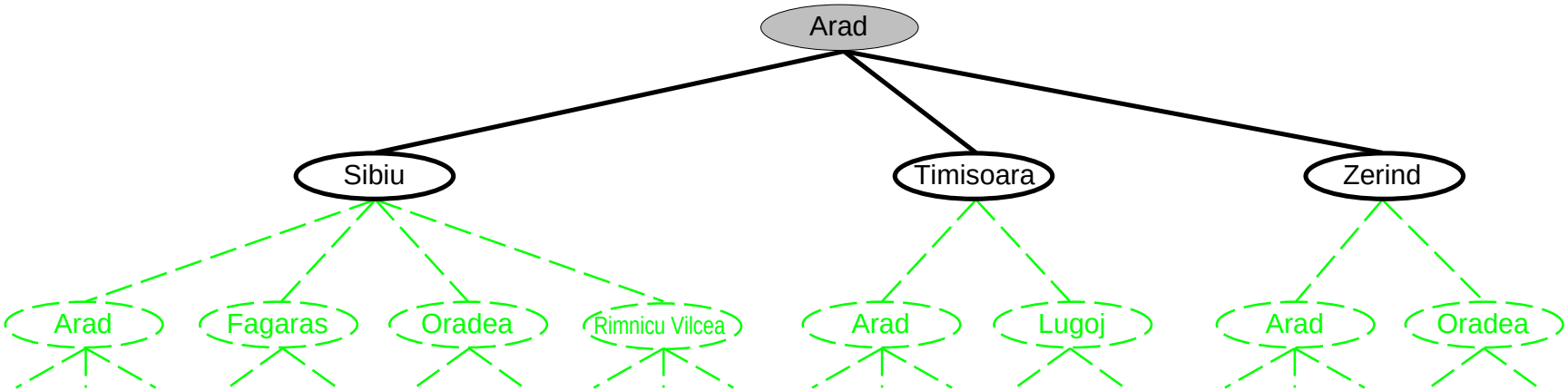
Beispiel: Routenplanung



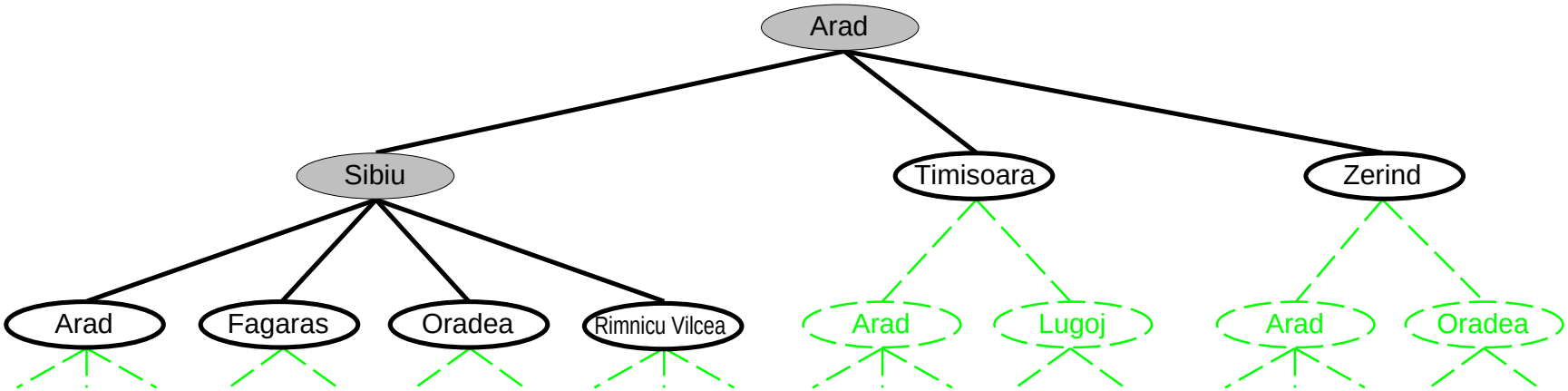
Beispiel: Baumsuche



Beispiel: Baumsuche



Beispiel: Baumsuche



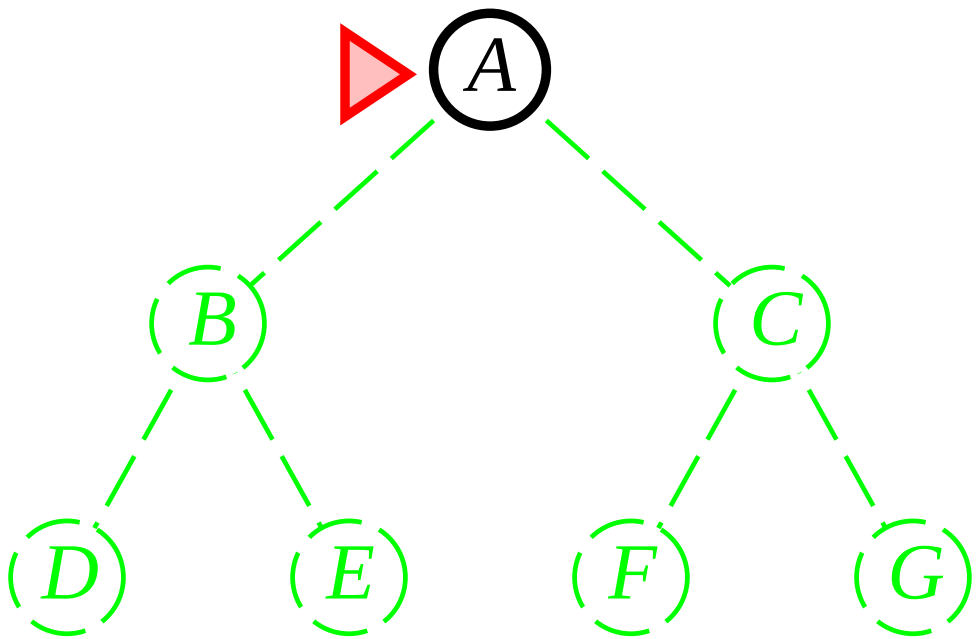
Uninformierte Suchstrategien

Uninformierte Suchstrategien nutzen nur verfügbare Informationen der Problemdefinition.

- Breitensuche
- uniforme Kostensuche
- Tiefensuche
- beschränkte Tiefensuche
- iterative Tiefensuche

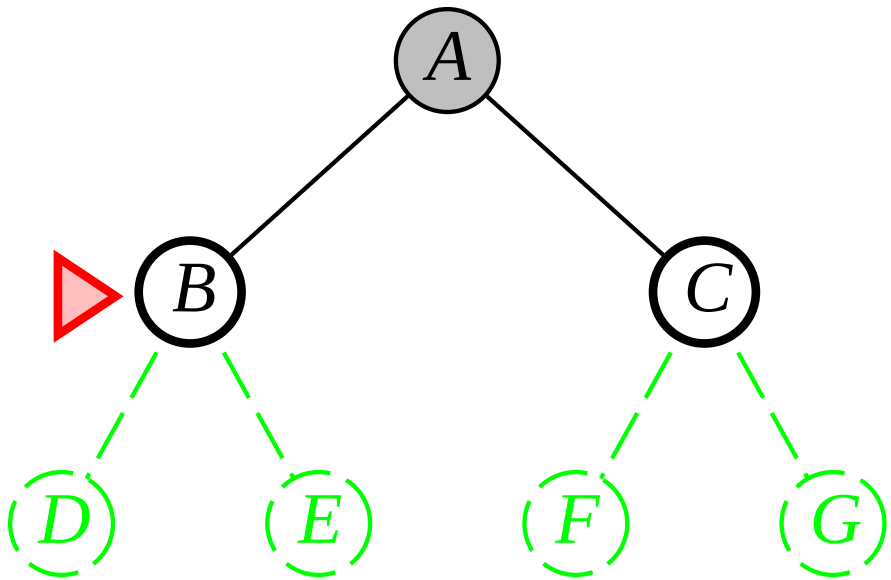
Breitensuche (engl. *breadth-first search*)

- Expandiere „seichtesten“, nicht-expandierten Knoten!
- Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



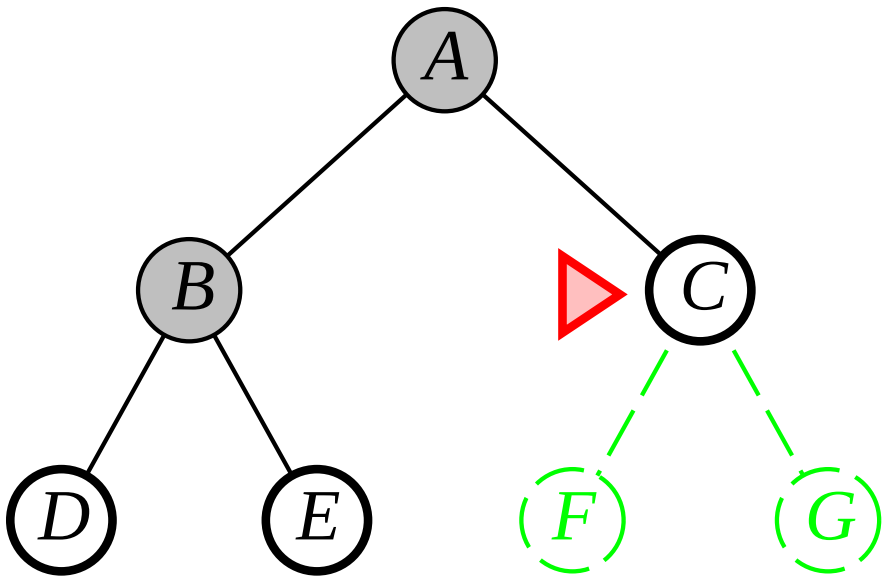
Breitensuche (engl. *breadth-first search*)

- Expandiere „seichtesten“, nicht-expandierten Knoten!
- Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



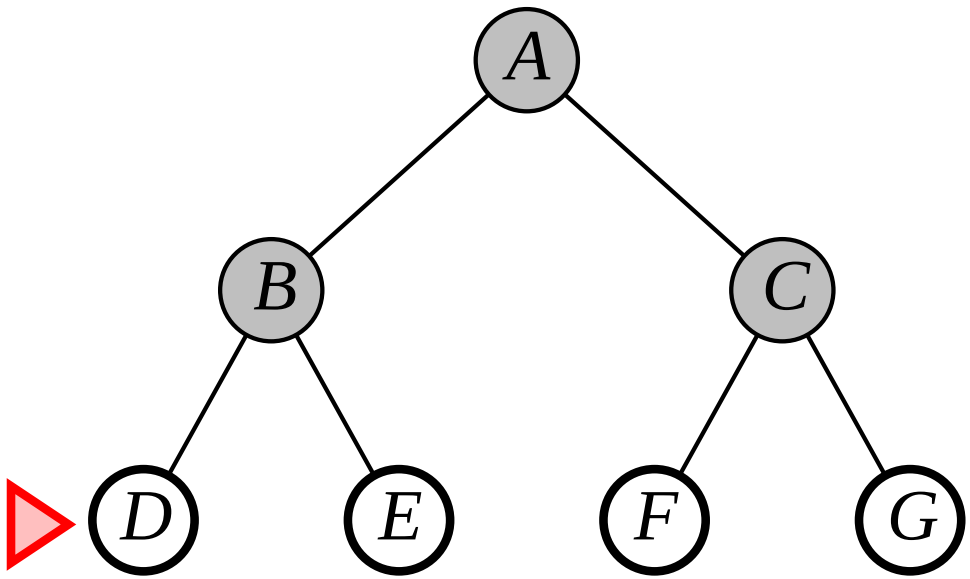
Breitensuche (engl. *breadth-first search*)

- Expandiere „seichtesten“, nicht-expandierten Knoten!
- Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



Breitensuche (engl. *breadth-first search*)

- Expandiere „seichtesten“, nicht-expandierten Knoten!
- Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



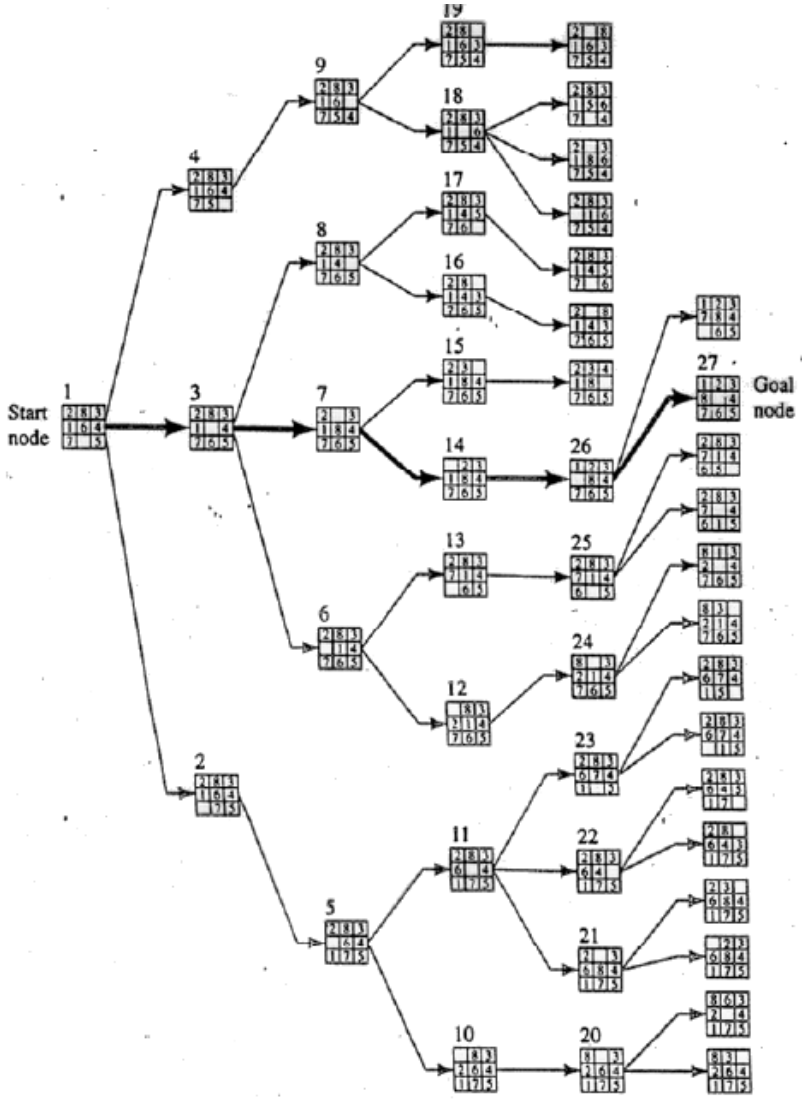
Breitensuche: 8-Puzzle

Start

2	8	3
1	6	4
7		5

Ziel

1	2	3
8		4
7	6	5



Breitensuche: Eigenschaften

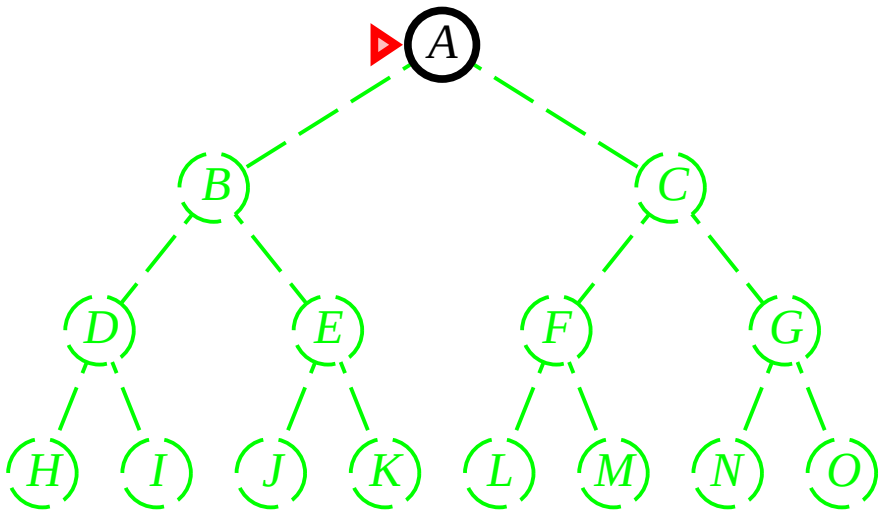
- vollständig: falls Verzweigungsfaktor b endlich
- Zeit: $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, also exponentiell in d (Tiefe der Lösung mit geringsten Kosten)
- Speicher: $O(b^{d+1})$ (behält jeden Knoten im Speicher)
- optimal: falls Kosten $= 1$ pro Schritt, generell nicht
- größtes Problem: Speicher
- es ist sehr leicht Knoten zu erzeugen, jedoch fehlt oft der Speicherplatz diese auch abzuspeichern: z.B. bei 100 MB/s sind das für 24 h ganze 8.5 TByte

Uniforme Kostensuche

- Expandiere nicht-expandierten Knoten mit geringsten Kosten
- Implementierung: *fringe* = Schlange absteigend sortiert nach Wegkosten
- äquivalent zur Breitensuche falls Schrittkosten alle gleich
- vollständig: falls Schrittkosten $\geq \varepsilon$
- seien $g(n)$ die bereits aufgenommenen Kosten um Knoten n zu erreichen
- Zeit: # Knoten mit $g \leq$ Kosten der optimalen Lösung $O(b^{\lceil C^*/\epsilon \rceil})$ wobei C^* Kosten der optimalen Lösung
- Speicher: # Knoten mit $g \leq$ Kosten der optimalen Lösung $O(b^{\lceil C^*/\epsilon \rceil})$
- optimal: ja, denn Knoten expandieren in aufsteigender Reihenfolge von $g(n)$

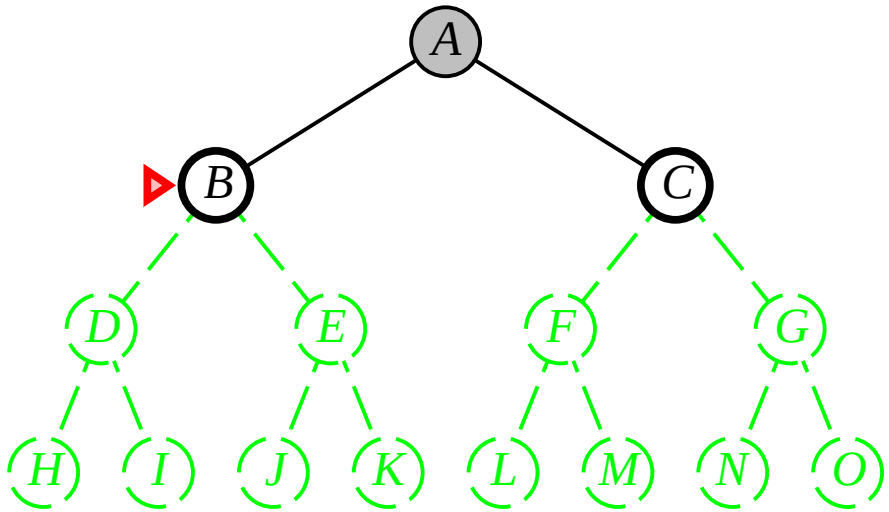
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



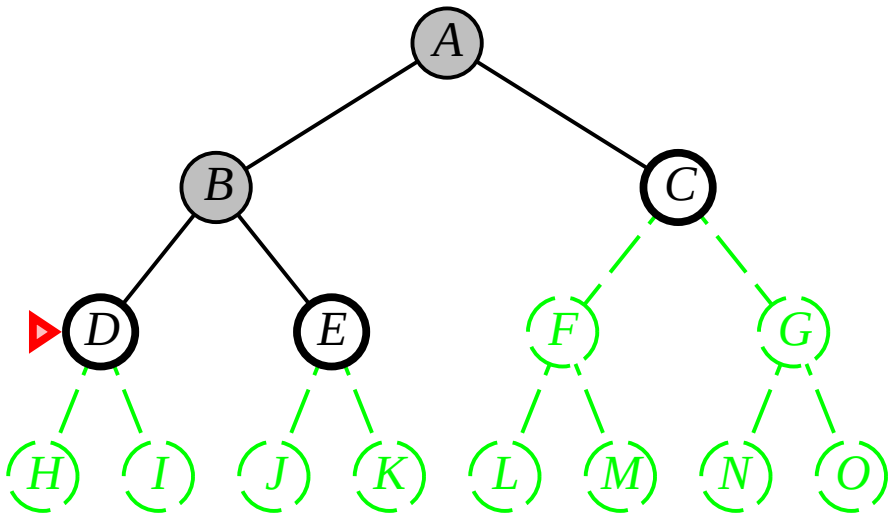
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



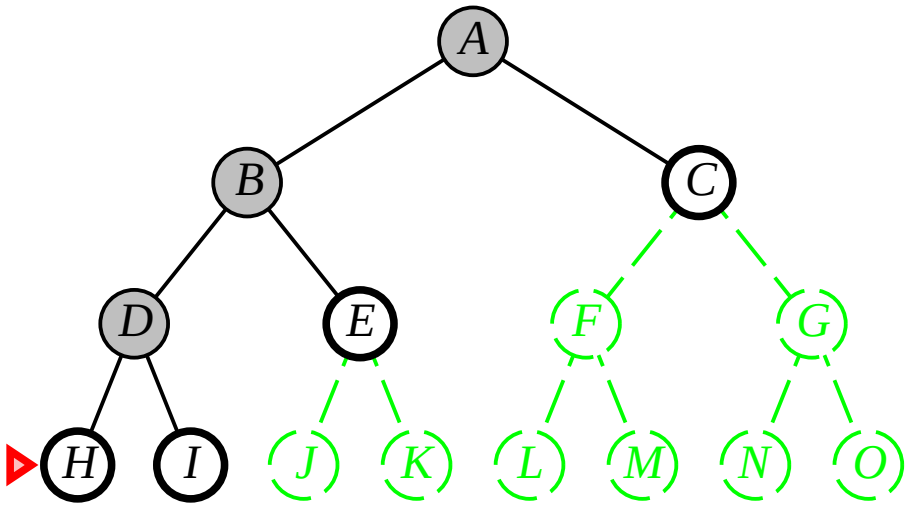
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



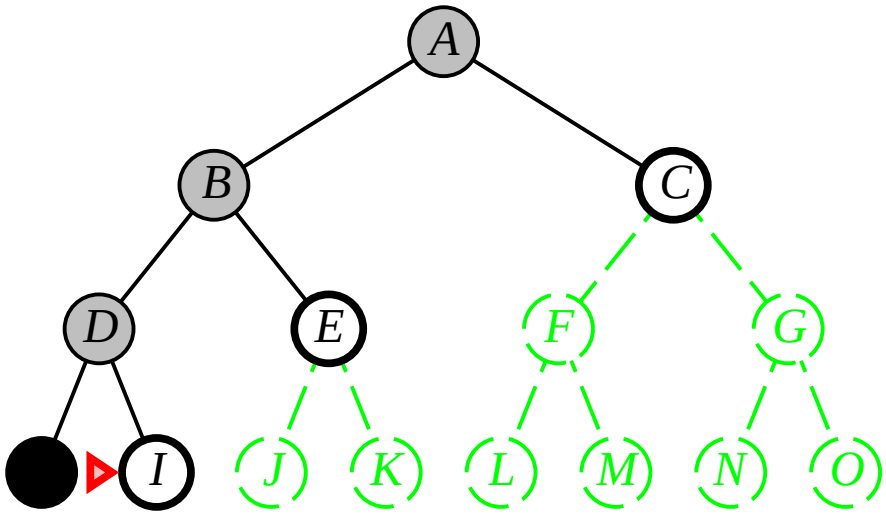
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



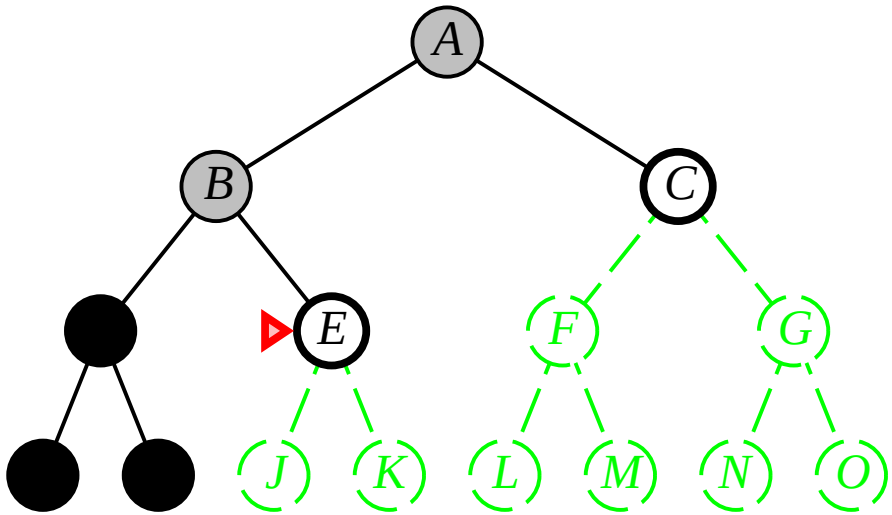
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



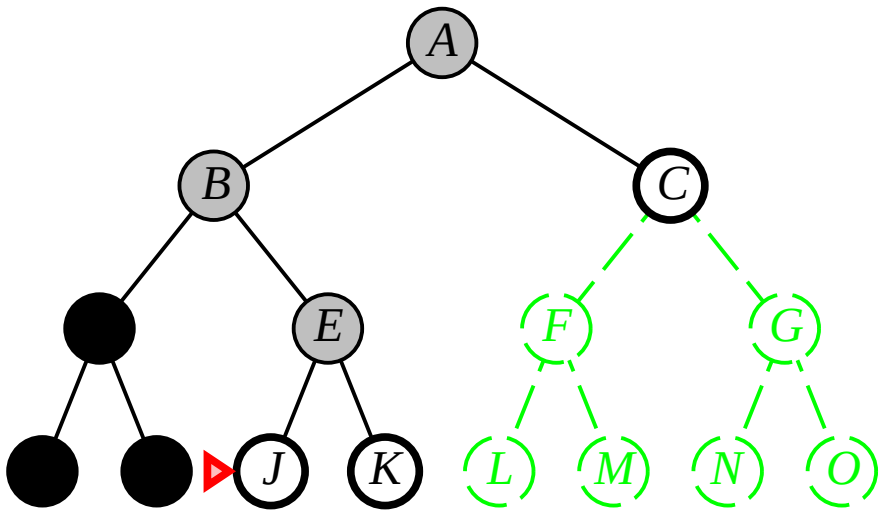
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



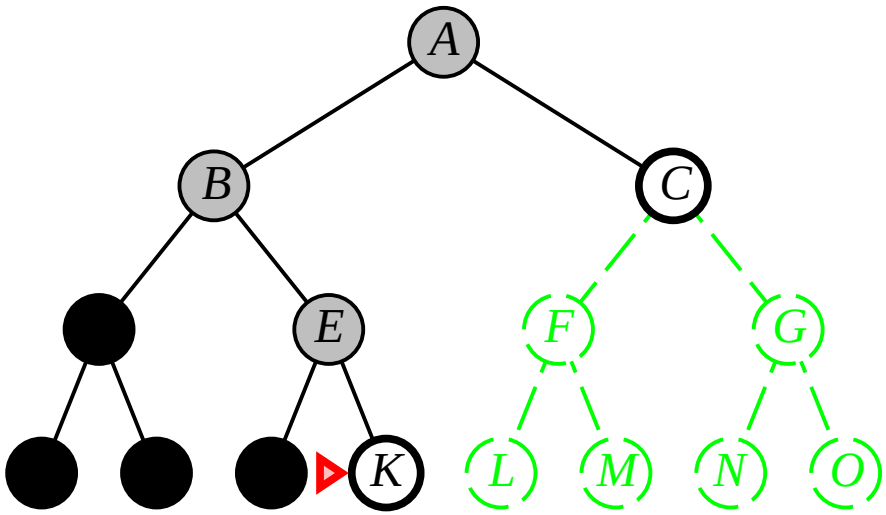
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



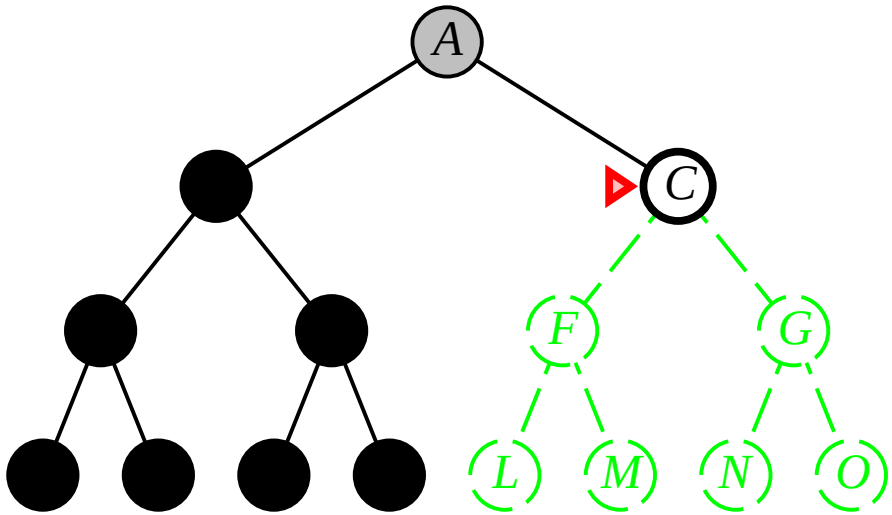
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



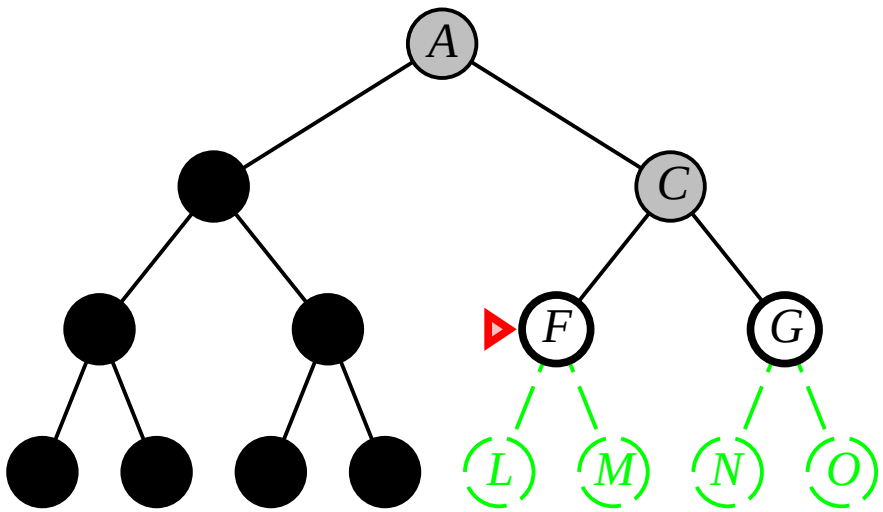
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



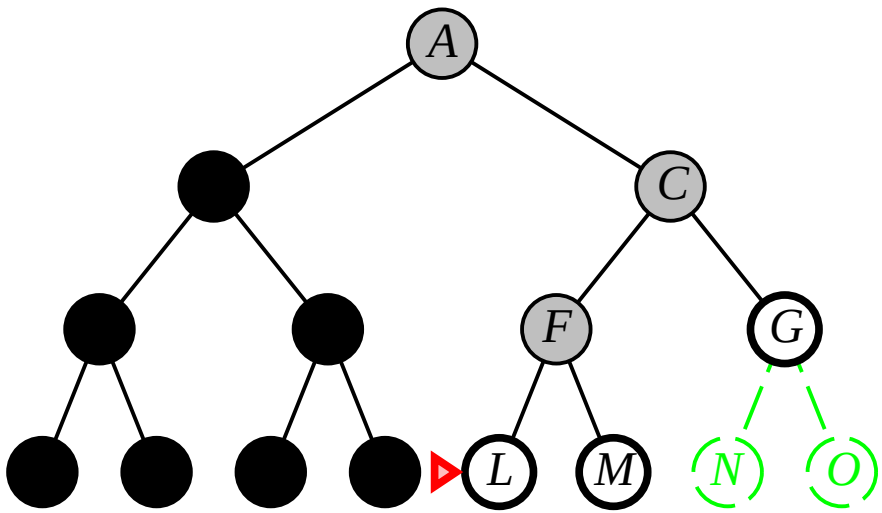
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



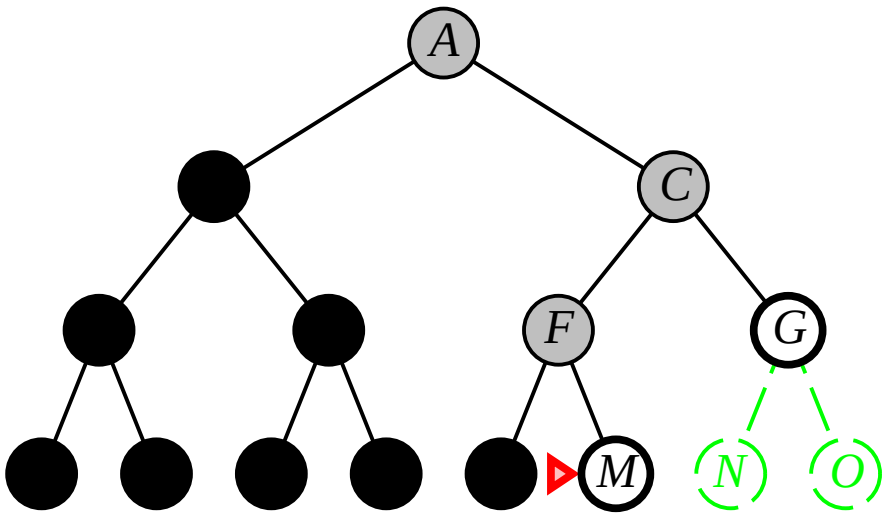
Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

- Expandiere tiefsten nicht-expandierten Knoten!
- Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche: Eigenschaften

- vollständig:
  - nein, versagt für unendlich-tiefe Räume (oder Räume mit Schleifen)
  - ja, für endliche Räume bei Vermeidung sich wiederholender Zustände im Pfad
- Zeit:
  - m maximale Tiefe des Zustandsraums (eventuell ∞)
  - $O(b^m)$, schrecklich falls $m \gg d$
  - falls viele Lösungen, dann u.U. viel schneller als Breitensuche
- Speicher: $O(bm)$, also linearer Speicher!
- optimal: nein

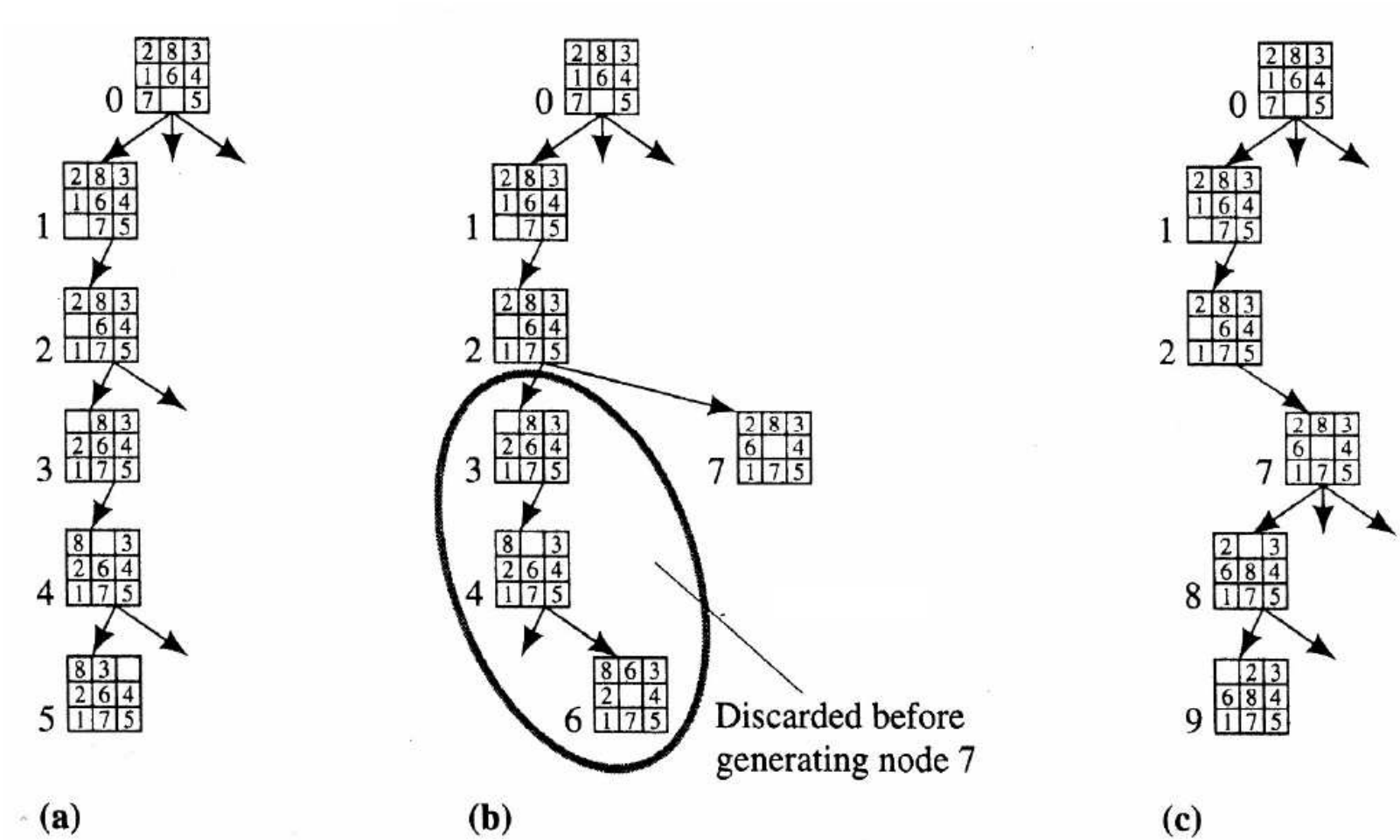
Beschränkte Tiefensuche

- Tiefensuche mit Tiefenbegrenzung l
- demnach: Knoten in Tiefe l haben keine Nachfolger

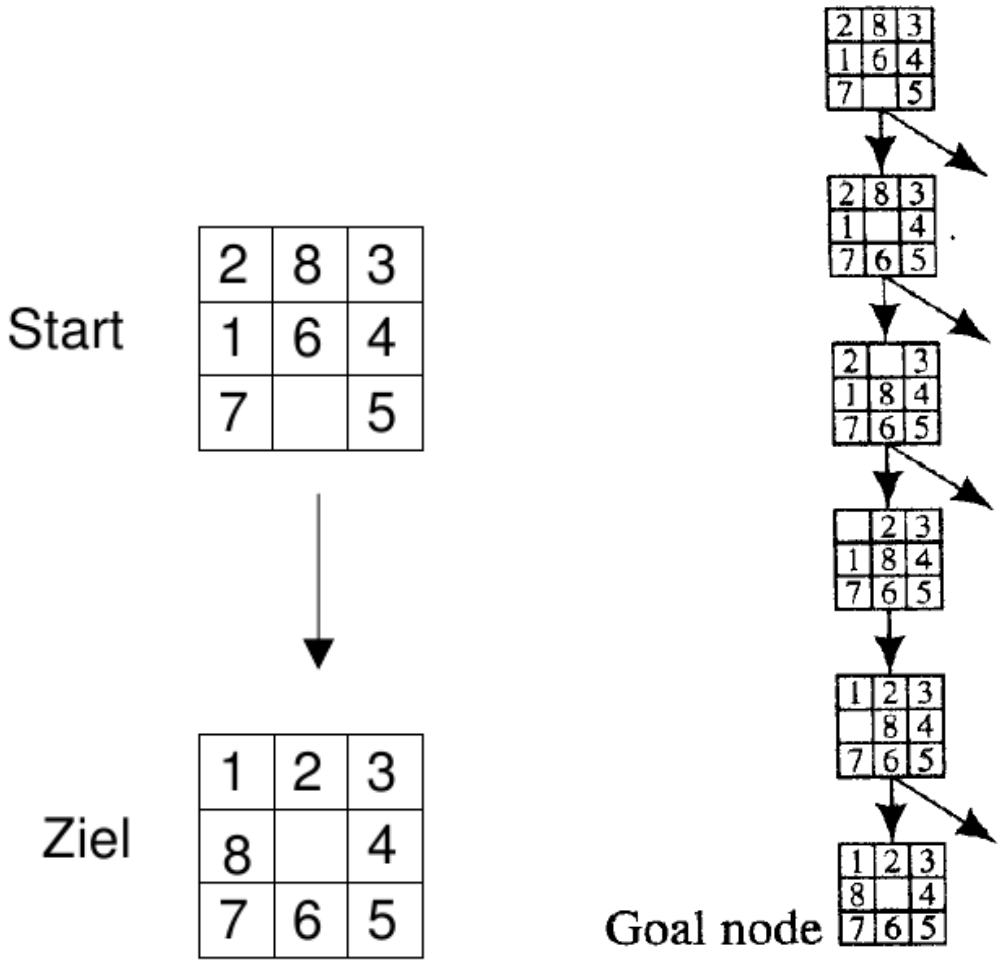
Beispiel: 8-Puzzle

- Operationsreihenfolge: links, oben, rechts, unten
- Tiefenbegrenzung: 5

Beschränkte Tiefensuche: 8-Puzzle



Beschränkte Tiefensuche: 8-Puzzle



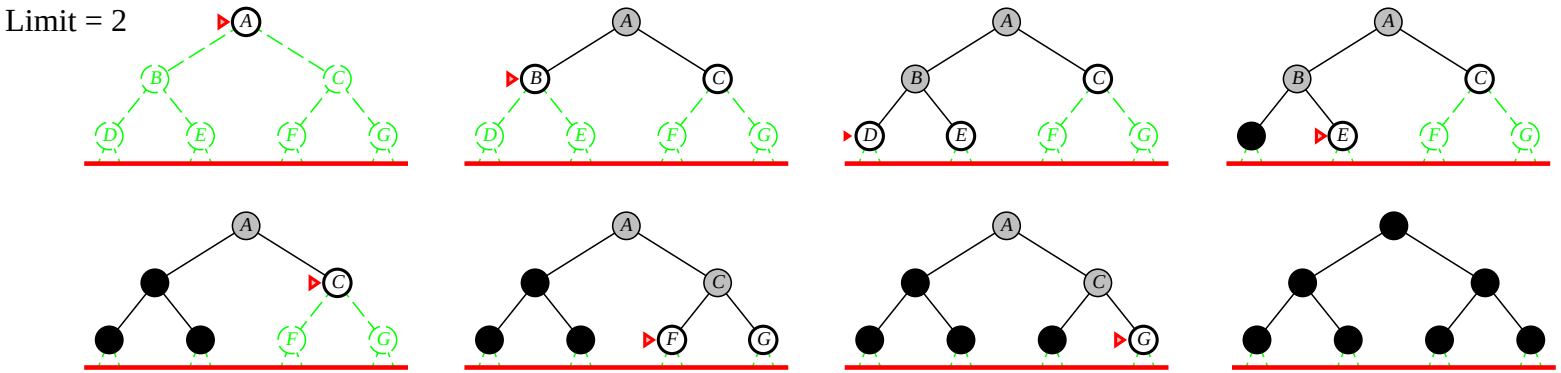
Iterative Tiefensuche

Limit = 0

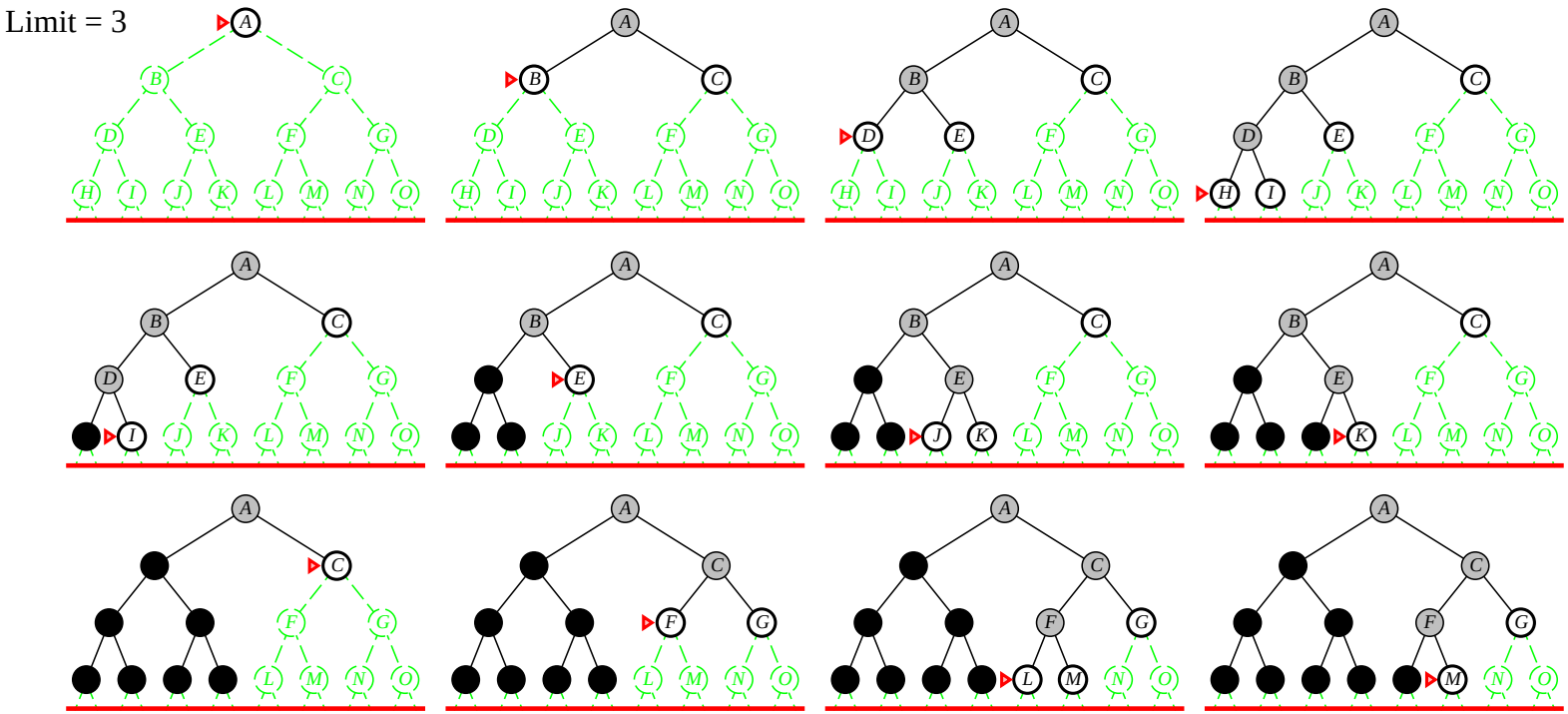
Iterative Tiefensuche

Limit = 1

Iterative Tiefensuche



Iterative Tiefensuche



Iterative Tiefensuche: Eigenschaften

- vollständig: ja
- Zeit: $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Speicher: $O(b \cdot d)$
- optimal:
  - ja, falls Schrittkosten $= 1$
  - kann um uniforme Kostenbäume erweitert werden
- numerischer Vergleich für $b = 10$ und $d = 5$ (Lösung im am weitesten rechten Blatt):

$$N(\text{IDS}) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$

$$N(\text{BFS}) = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100$$

- IDS besser, weil andere Knoten bei Tiefe $d = 5$ nicht expandiert
- BFS kann modifiziert werden, um auf Ziel zu testen wenn Knoten generiert wird

Zusammenfassung der Algorithmen

Kriterium	Breiten-suche	uniforme Kostensuche	Tiefen-suche	beschränkte Tiefensuche	iterative Tiefensuche
vollständig?	ja*	ja*	nein	ja, falls $l \geq d$	ja
Zeit	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Speicher	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
optimal?	ja*	ja	nein	nein	ja*

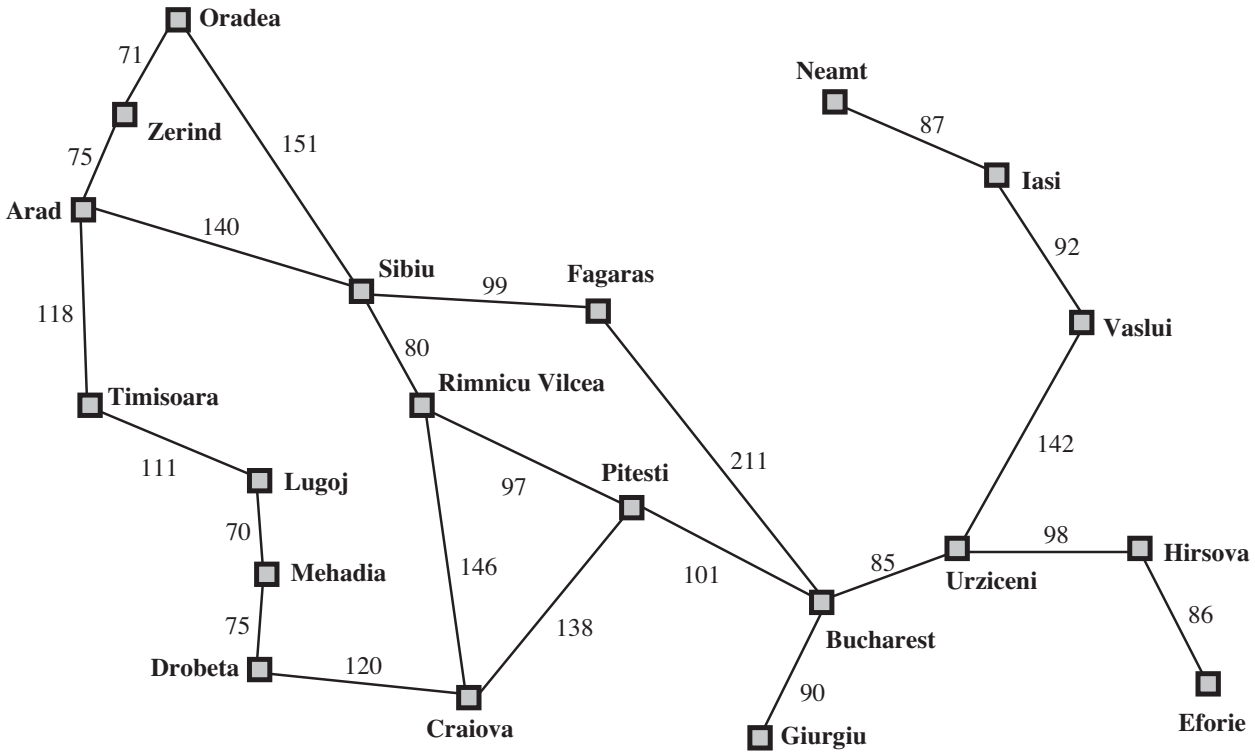
Übersicht

Bestensuche

Idee: nutze Bewertungsfunktion für jeden Knoten

- Schätzung, wie „wünschenswert/begehrt“ Knoten ist
- somit: Expansion des am wünschenswertesten (noch nicht expandierten) Knotens
- Implementierung:
  fringe = Queue absteigend sortiert nach „Begehrtheit“
- Spezialfälle: Greedy-Suche, A*-Algorithmus

Beispiel: Routenplanung mit Schrittkosten in km



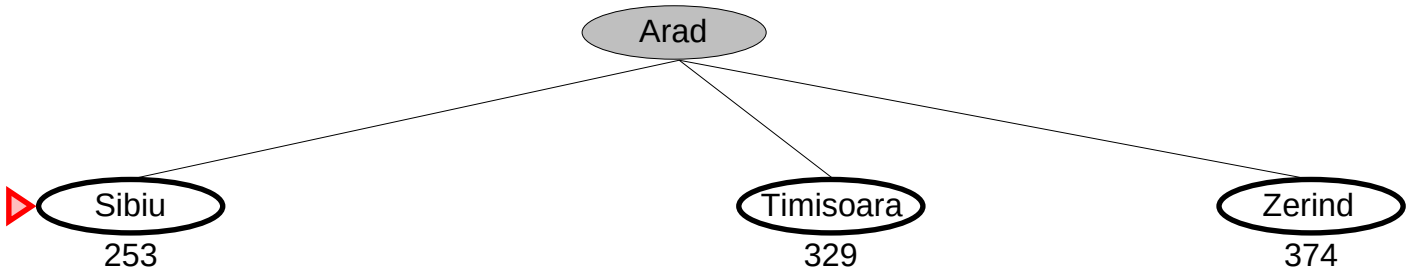
Luftlinie nach Bukarest	
Arad	366
Bukarest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy-Suche

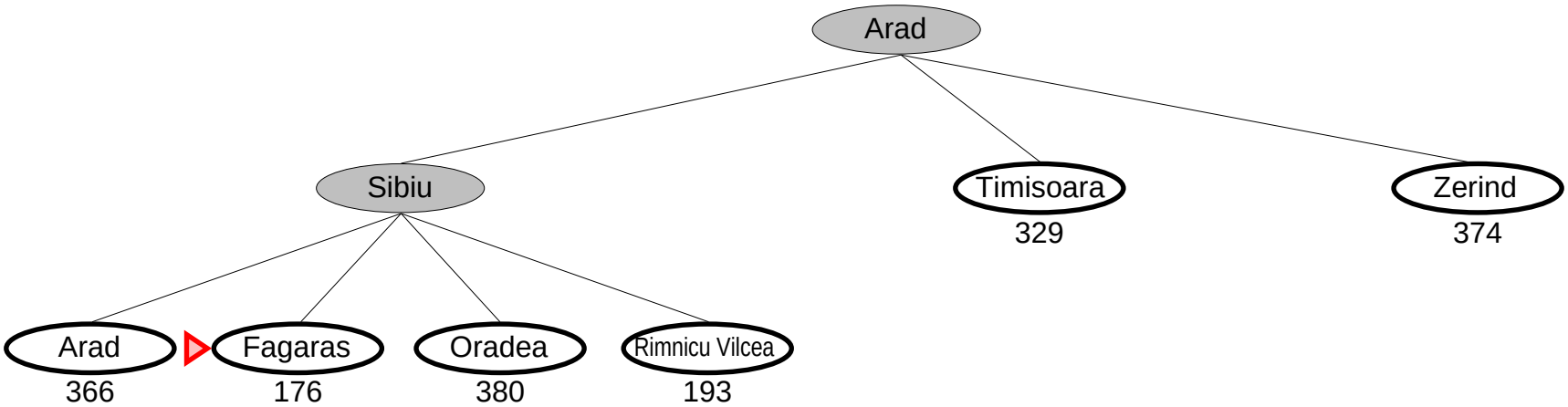
- Bewertungsfunktion $h(n)$ (Heuristik):
  Schätzung der Kosten von n zum nächsten Ziel
- z.B. $h_{\mathrm{LL}}(n) =$ Luftlinienabstand von n nach Bukarest
- Greedy-Suche expandiert Knoten der am nächsten am Ziel *scheint*

Beispiel: Greedy-Suche

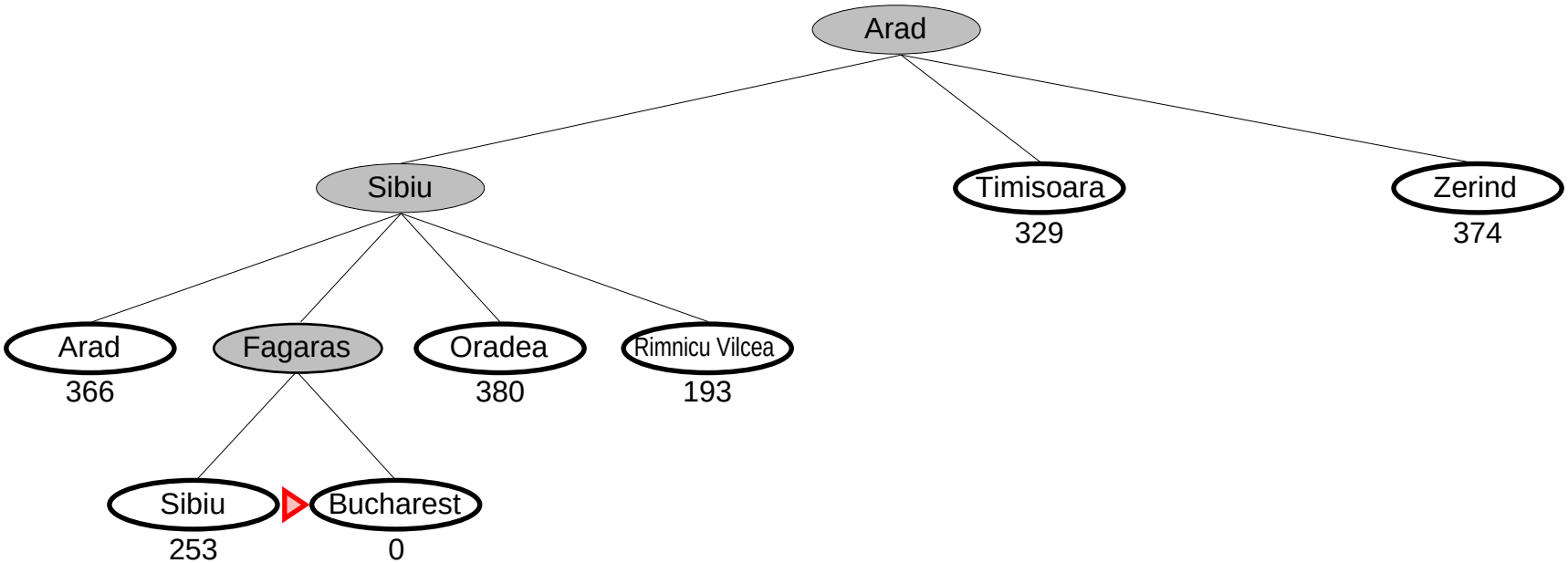
Beispiel: Greedy-Suche



Beispiel: Greedy-Suche



Beispiel: Greedy-Suche



Greedy-Suche: Eigenschaften

- vollständig:
  - nein, kann in Schleifen hängenbleiben, z.B.
    Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$ ...
  - ja, für endliche Räume bei Vermeidung sich wiederholender Zustände im Pfad
- Zeit: $O(b^m)$, mit guter Heuristik drastische Verbesserung
- Speicher: $O(b^m)$ (behält jeden Knoten im Speicher)
- optimal: nein

Übersicht

A*-Algorithmus

Idee: vermeide Expansion bereits teuer expandierter Pfade
Bewertungsfunktion $f(n) = g(n) + h(n)$

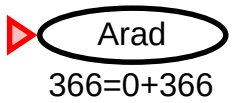
- $g(n)$ bereits aufgenommene Kosten um n zu erreichen
- $h(n)$ geschätzte Kosten von n zum Ziel
- $f(n)$ geschätzte Gesamtkosten des Pfades durch n zum Ziel

A*-Algorithmus benutzt *zulässige Heuristik*

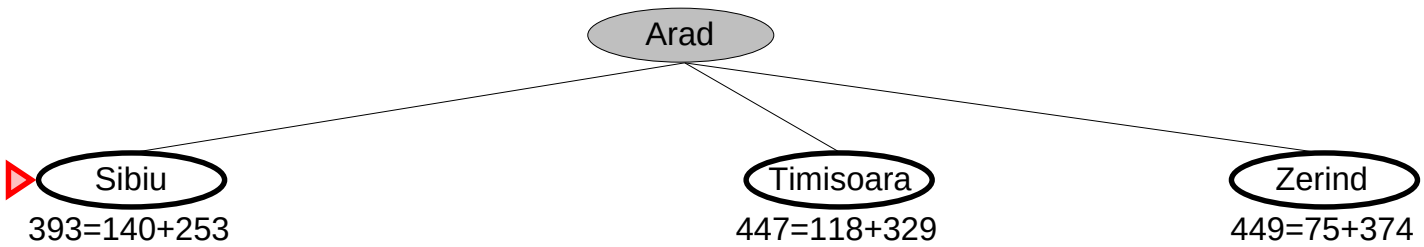
- also, $h(n) \leq h^*(n)$ wobei $h^*(n)$ **wahren** Kosten von n
- auch verlangt: $h(n) \geq 0$, also $h(G) = 0$ für beliebiges Ziel G
- z.B. $h_{\mathrm{LL}}(n)$ überschätzt wirkliche Wegstrecke nie!

Satz: A*-Algorithmus ist optimal.

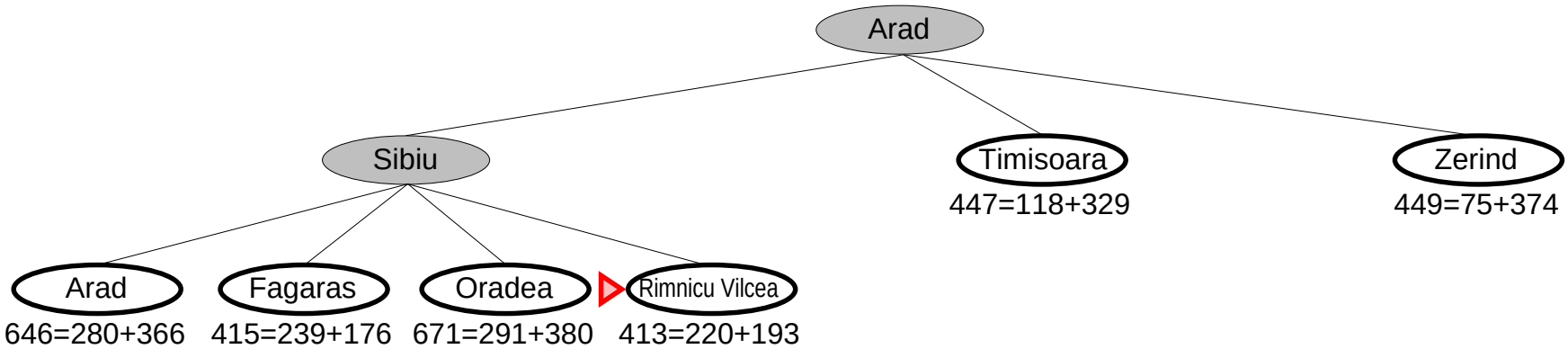
Beispiel: A*-Algorithmus



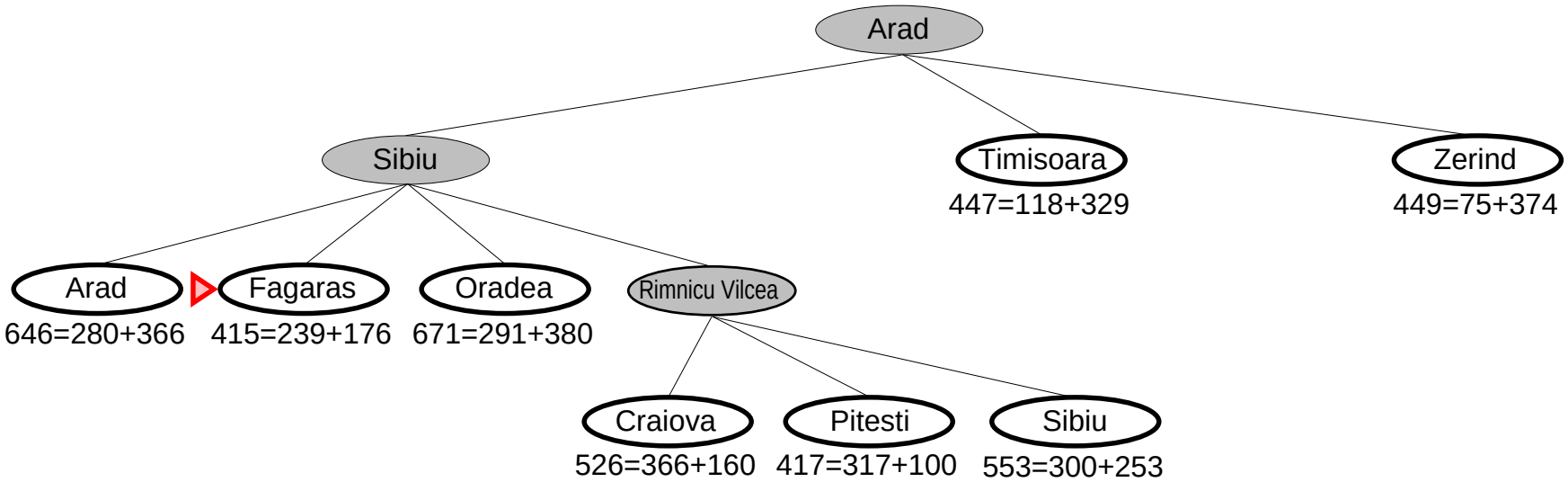
Beispiel: A*-Algorithmus



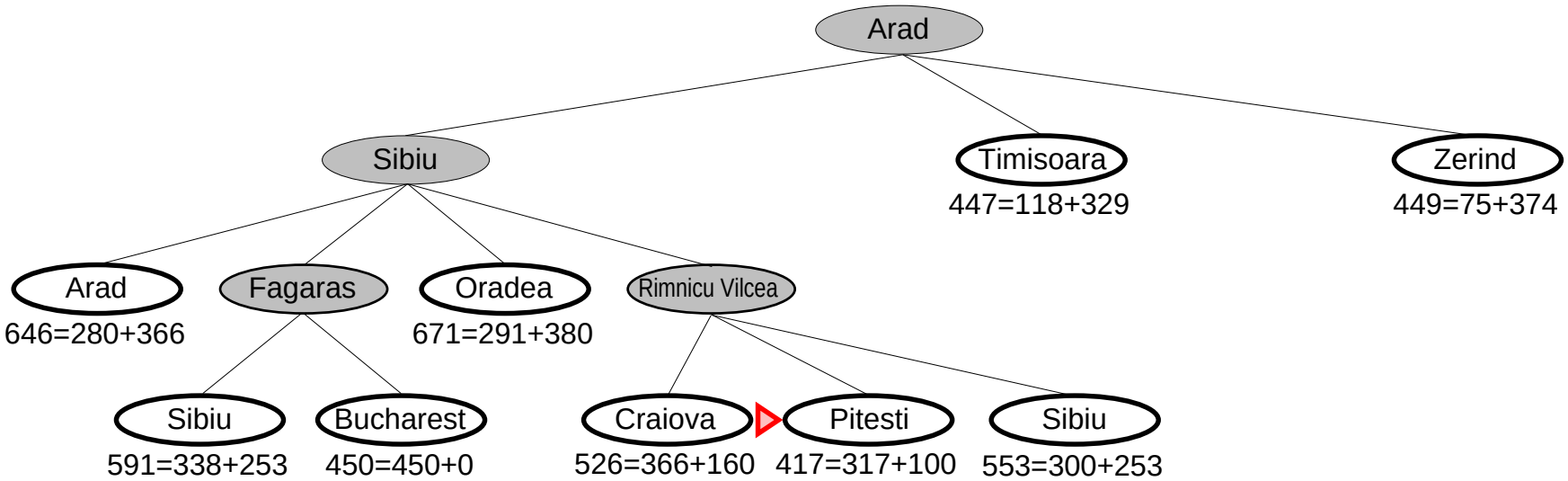
Beispiel: A*-Algorithmus



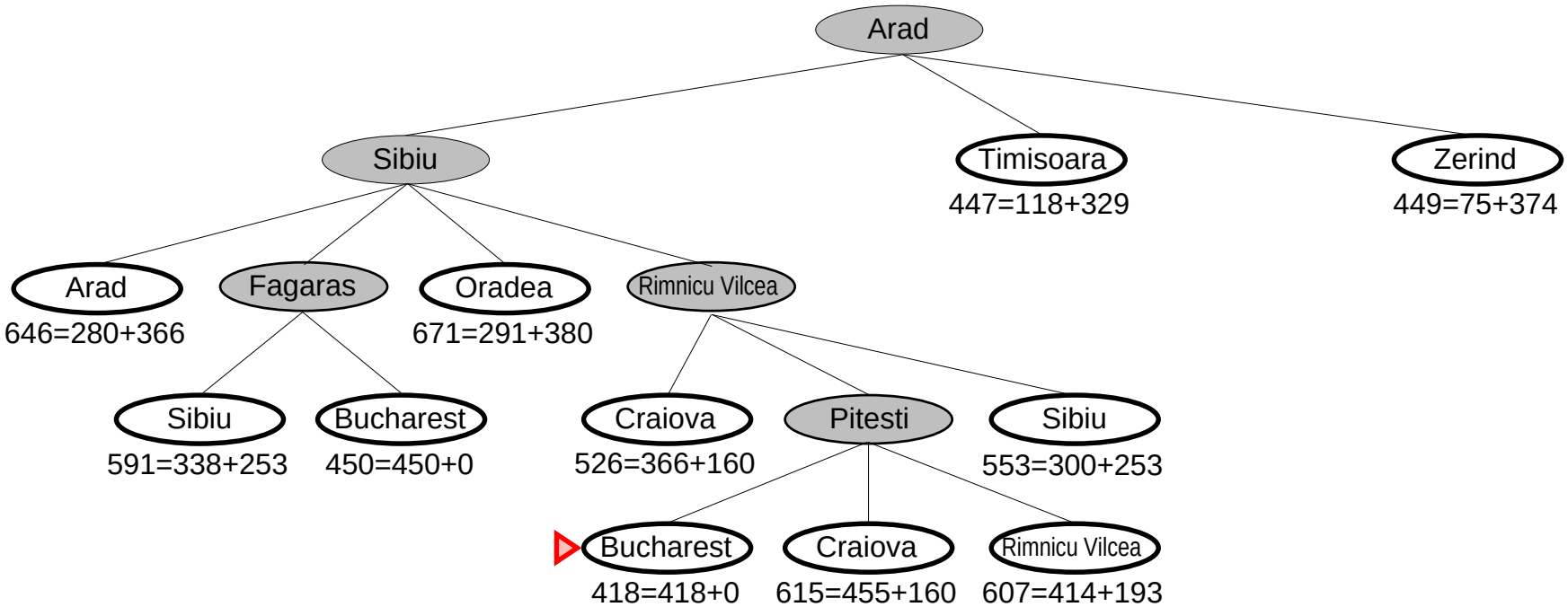
Beispiel: A*-Algorithmus



Beispiel: A*-Algorithmus



Beispiel: A*-Algorithmus



A*-Algorithmus: Gegeben

- Startzustand z_0
- Menge $O = \{o_1, \ldots, o_n\}$ von Operationen:
  liefern zu gegebenem Zustand Nachfolgezustand
  - i.A. nicht alle Operationen auf alle Zustände anwendbar
  - Operation liefert speziellen Wert $\perp$ (undefiniert) statt neuem Zustand, falls nicht anwendbar
- reellwertige Funktion `costs`:
  liefert für jede $o_i \in O$ zugehörigen Kosten
  - u.U. hängen Kosten vom Zustand ab
    (`costs` kann auch zweistellig sein)
- reellwertige Heuristikfunktion h
- Funktion `goal` stellt fest, ob Zustand = Ziel

A*-Algorithmus: Ablauf I

1. erzeuge (gericht.) Graphen $G = \{V, E\}$ mit $V := \{z_0\}$, $E := \emptyset$
   (G stellt den besuchten Teil des Suchraums und die besten bekannten Wege zum Erreichen eines Zustandes dar)

2. erzeuge Menge `open` mit $\texttt{open} := \{z_0\}$
   (`open` enthält die erreichten Zustände mit noch nicht erzeugten Nachfolgern)

3. erzeuge leere Menge `closed`
   (`closed` enthält die erreichten Zustände mit bereits erzeugten Nachfolgern)

A*-Algorithmus: Ablauf II

4. erzeuge Abbildung $g : V \to \mathbb{R}$ mit $z_0 \mapsto 0$ und sonst undefiniert (sog. Tiefenfaktor g: gibt Kosten der besten gefundenen Operationenfolgen zum Erreichen eines Zustandes von z_0 an)

A*-Algorithmus: Ablauf III

5. erzeuge Abbildung $e : V \to O$ für alle Zustände undefiniert
(e baut Lösung des Problems auf: e gibt an, durch welche Operationen ein Zustand von seinem Vorgänger aus erreicht wird)

6. wähle $z \in \texttt{open}$ mit $z \in \left\{x \mid f(x) = \min_{y \in \texttt{open}} f(y)\right\}$ wobei $f = g + h$
(wähle „erfolgversprechendsten“ Zustand gemäß h)

7. falls $\texttt{goal}(z)$, dann Lösung gefunden und somit lese Pfad aus G ab

8. entferne z aus open, d.h. $\texttt{open} := \texttt{open} \setminus \{z\}$
(Nachfolger von z im folgenden Schritt erzeugt)

A*-Algorithmus: Ablauf IV

9. für alle $o \in O$:

- $x := o(z)$ und $c := g(z) + \mathrm{costs}(o)$
- falls $x \neq \perp$, dann
  - falls $x \notin \texttt{open} \cup \texttt{closed}$, dann
    - ▷ $\texttt{open} := \texttt{open} \cup \{x\}$, $e(x) := o$, $g(x) = c$
    - ▷ erweitere G durch $V := V \cup \{x\}$ und $E := E \cup \{(z, x)\}$
  - falls $x \in \texttt{open} \cup \texttt{closed}$ und $c < g(x)$, dann
    - ▷ $e(x) := o$ $g(x) = c$
    - ▷ ersetze Vorgänger durch

    $E := (E \setminus \{(a, b) \mid b = x\}) \cup \{(z, x)\}$
    - ▷ falls $x \in \texttt{closed}$, dann prüfe rekursiv alle Zustände, die sich von x erreichen lassen und ersetze Vorgänger ggf. durch günstigere Vorgänger

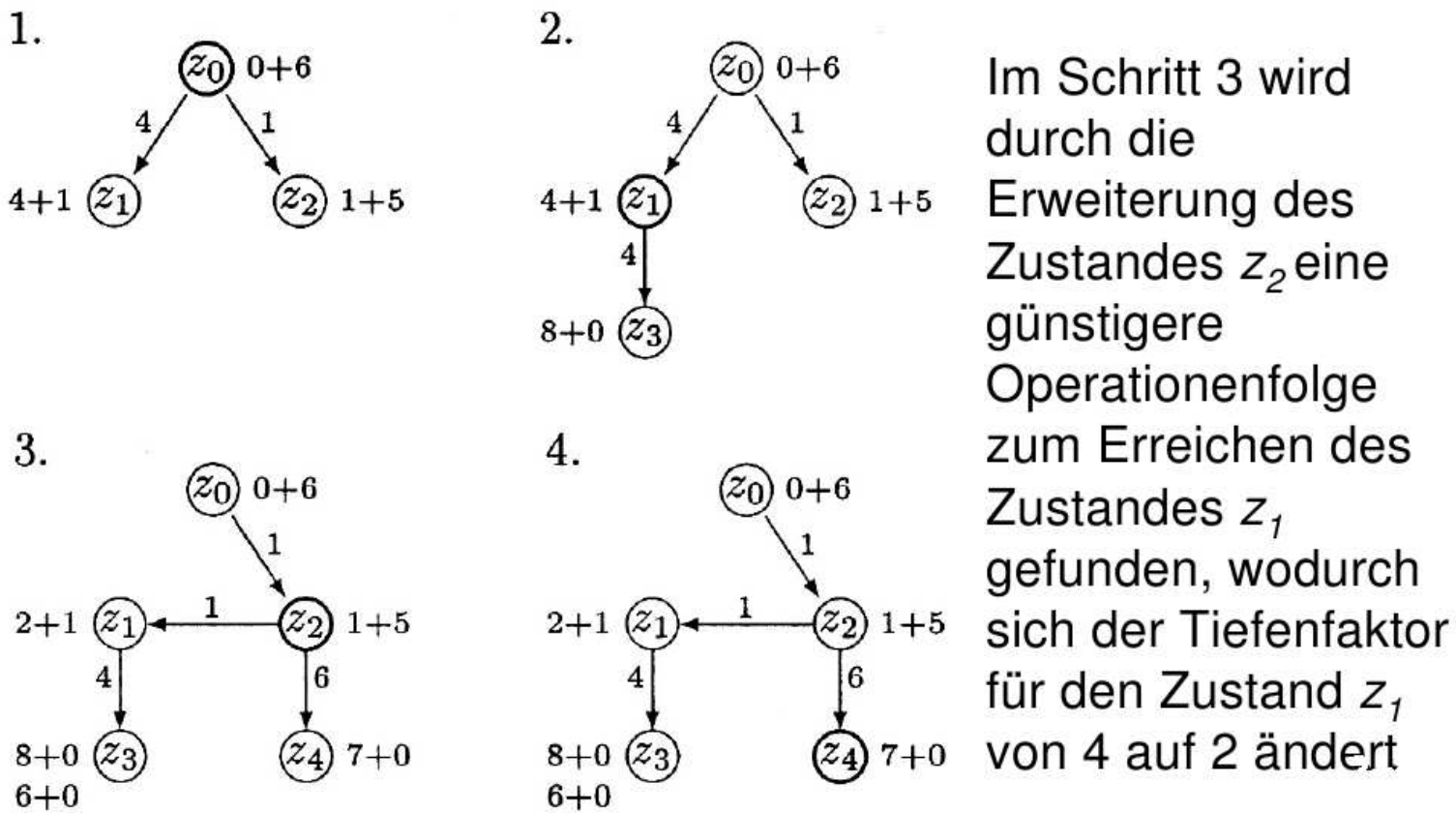
A*-Algorithmus: Ablauf V

10. nimm Zustand z in `closed` auf, also
$\texttt{closed} := \texttt{closed} \cup \{z\}$
(Nachfolger von z im vorhergehenden Schritt erzeugt)

11. falls `open` leer, dann Problem unlösbar und somit bricht A* ab, andernfalls gehe zu Schritt 6

A*-Algorithmus: Anpassung des Tiefenfaktors

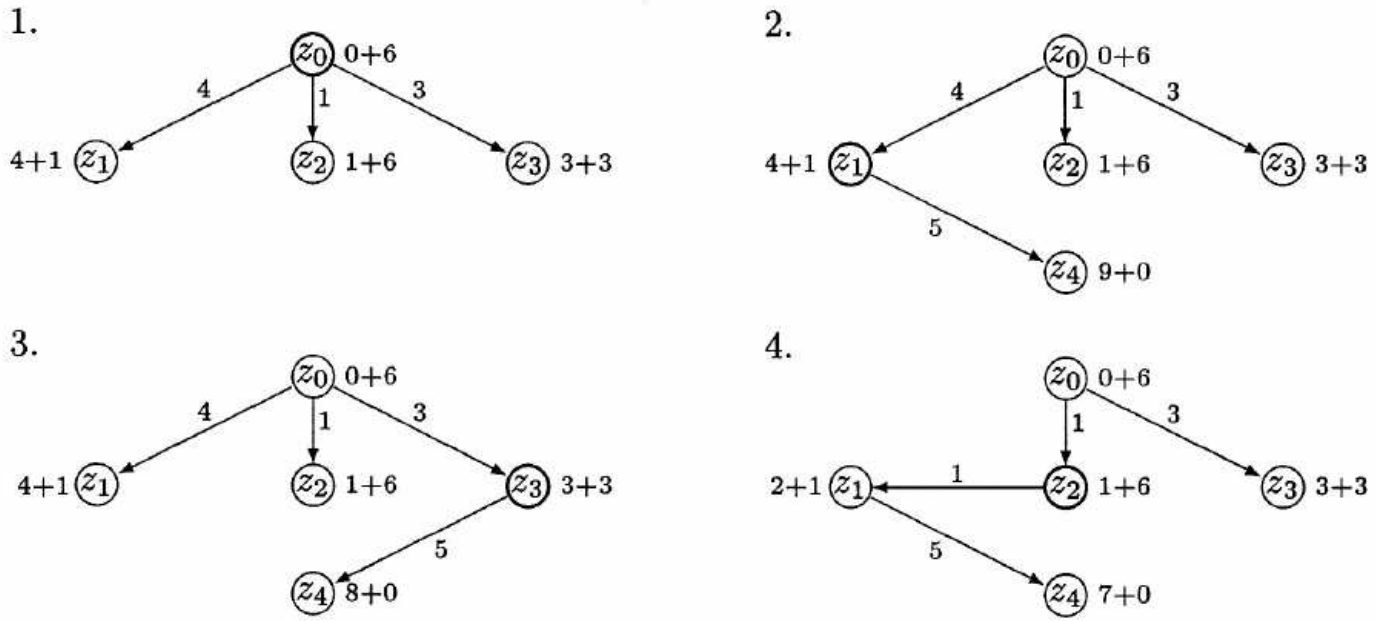
Notwendigkeit der Anpassung von Nachfolgezuständen (**9.**):



Im Schritt 3 wird durch die Erweiterung des Zustandes z_2 eine günstigere Operationenfolge zum Erreichen des Zustandes z_1 gefunden, wodurch sich der Tiefenfaktor für den Zustand z_1 von 4 auf 2 ändert

A*-Algorithmus: Anpassung des Tiefenfaktors

Notwendigkeit der (rekursiven) Anpassung *aller* Zustände (**9.**):



Im Schritt 3 wird durch die Erweiterung des Zustandes z_3 ein günstigerer Knoten z_4 gefunden. Daher wird die Kante (z_1,z_4) entfernt und stattdessen die Kante (z_3,z_4) eingefügt. Die Erweiterung von z_2 in Schritt 4 liefert aber einen kürzeren Weg nach z_1 (und z_4) und erfordert neue Zuordnung der Kante (kein Nachfolger!

A*-Algorithmus: Eigenschaften

- vollständig: ja, solange wie es unendlich mehr Knoten mit $f \leq f(G)$ gibt
- Zeit: exponentiell in [relativer Fehler in $h \times$ Länge der Lösung]
- Speicher: behält jeden Knoten im Speicher
- optimal: ja, A* kann nicht f_{i+1} expandieren bis f_i beendet

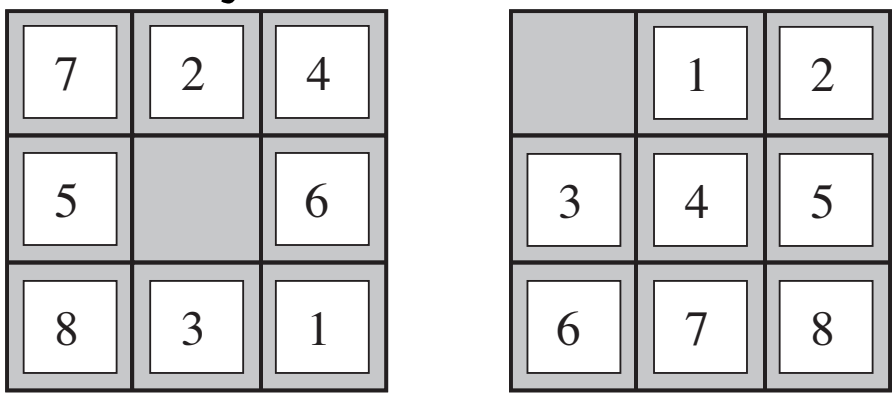
A* expandiert alle Knoten mit $f(n) < C^*$
A* expandiert einige Knoten mit $f(n) = C^*$
A* expandiert keine Knoten mit $f(n) > C^*$

Zulässige Heuristiken

z.B. für 8-Puzzle:

$h_1(n) =$ Anzahl der Plättchen an falscher Position

$h_2(n) =$ Summe der Manhattan-/City-Block-Abstände zw. falscher und gewünschter Position jedes Plättchens



$h_1(S) = 6$, h_1 für Startzustand (6 Plättchen an falscher Position)

$h_2(S) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$, h_2 für Startzustand

Dominanz

wenn h_1, h_2 zulässig und $h_2(n) \geq h_1(n)$ für alle n, dann $h_2 \succ h_1$ (h_2 **dominiert** h_1)
somit ist h_2 besser als h_1
typische Suchkosten:

- sei d Tiefe der Lösung mit geringsten Kosten
- für $d = 14$: iterative Tiefensuche ca. $3,5 \cdot 10^6$ Knoten
  $A^*(h_1) = 539$ Knoten, $A^*(h_2) = 113$ Knoten
- für $d = 24$: iterative Tiefensuche ca. $54 \cdot 10^9$ Knoten
  $A^*(h_1) = 39135$ Knoten, $A^*(h_2) = 1641$ Knoten

Satz: Gegeben 2 zulässige Heuristiken h_a, h_b.

$$h(n) = \max\{h_a(n), h_b(n)\}$$

ist auch zulässig und dominiert h_a, h_b.

Relaxierte Probleme

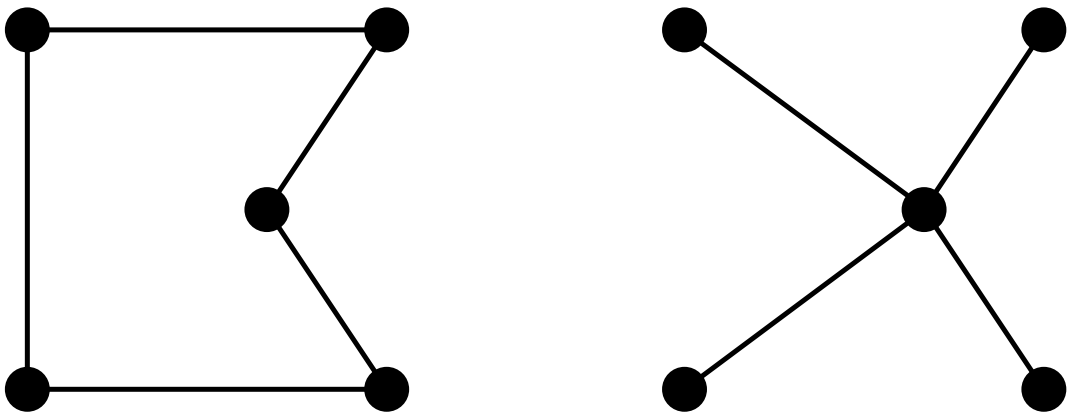
Wie erzeugt man zulässige Heuristiken?
Idee: Konstruktion **exakter** Lösungen einer relaxierten Version des Problems (Relaxierung: Weglassen oder Lockern von Bedingungen in Optimierungsproblemen), somit Ausnutzung der Kosten dieser Lösung als Heuristik

- falls Regeln des 8-Puzzles relaxiert, sodass Plättchen **überall** hin können, dann kürzeste Lösung mit $h_1(n)$
- falls Regeln des 8-Puzzles relaxiert, sodass Plättchen **zu jedem benachbarten Feld** können, dann kürzeste Lösung mit $h_2(n)$

Kosten der optimalen Lösung eines relaxierten Problems $\not>$
Kosten der optimalen Lösung des realen Problems

Relaxierte Probleme: TSP

Problem des Handlungsreisenden (engl. traveling salesman problem):
Finde kürzeste Rundreise, die alle Städte genau 1x besucht!



minimal aufspannender Baum (Berechnung in $O(n^2)$) ist untere Schranke der kürzesten (offenen) Rundreise

Zusammenfassung

- Heuristikfunktionen schätzen Kosten des kürzesten Pfads
- gute Heuristiken können Suchkosten dramatisch reduzieren
- Greedy-Suche expandiert Knoten mit kleinstem h
  - unvollständig und nicht immer optimal
- A*-Algorithmus expandiert Knoten mit kleinstem $g + h$
  - vollständig und optimal
  - auch optimal effizient (bis auf Unentschieden, für Vorwärtssuche)
- Erzeugung zulässiger Heuristiken durch exakte Lösungen relaxierter Probleme

Übersicht

Spiele vs. Suchprobleme

„unberechenbarer“ Gegner: Lösung = Strategie, die Zug für jede mögliche gegnerische Antwort spezifiziert
Zeitbegrenzung: unwahrscheinlich, dass Ziel gefunden wird, somit Näherungslösung
Geschichte:

- erste Überlegungen zu spielender Maschine (Babbage 1846)
- Algorithmen für perfektes Spiel (Zermelo 1912, Von Neumann 1944)
- endlicher Horizont, Näherungslösung (Zuse 1945, Wiener 1948, Shannon 1950)
- erstes Schachprogramm (Turing 1951)
- maschinelles Lernen zur Verbesserung der Bewertung (Samuel 1952–57)
- Stutzen erlaubt tiefere Suche (McCarthy 1956)

Arten von Spielen

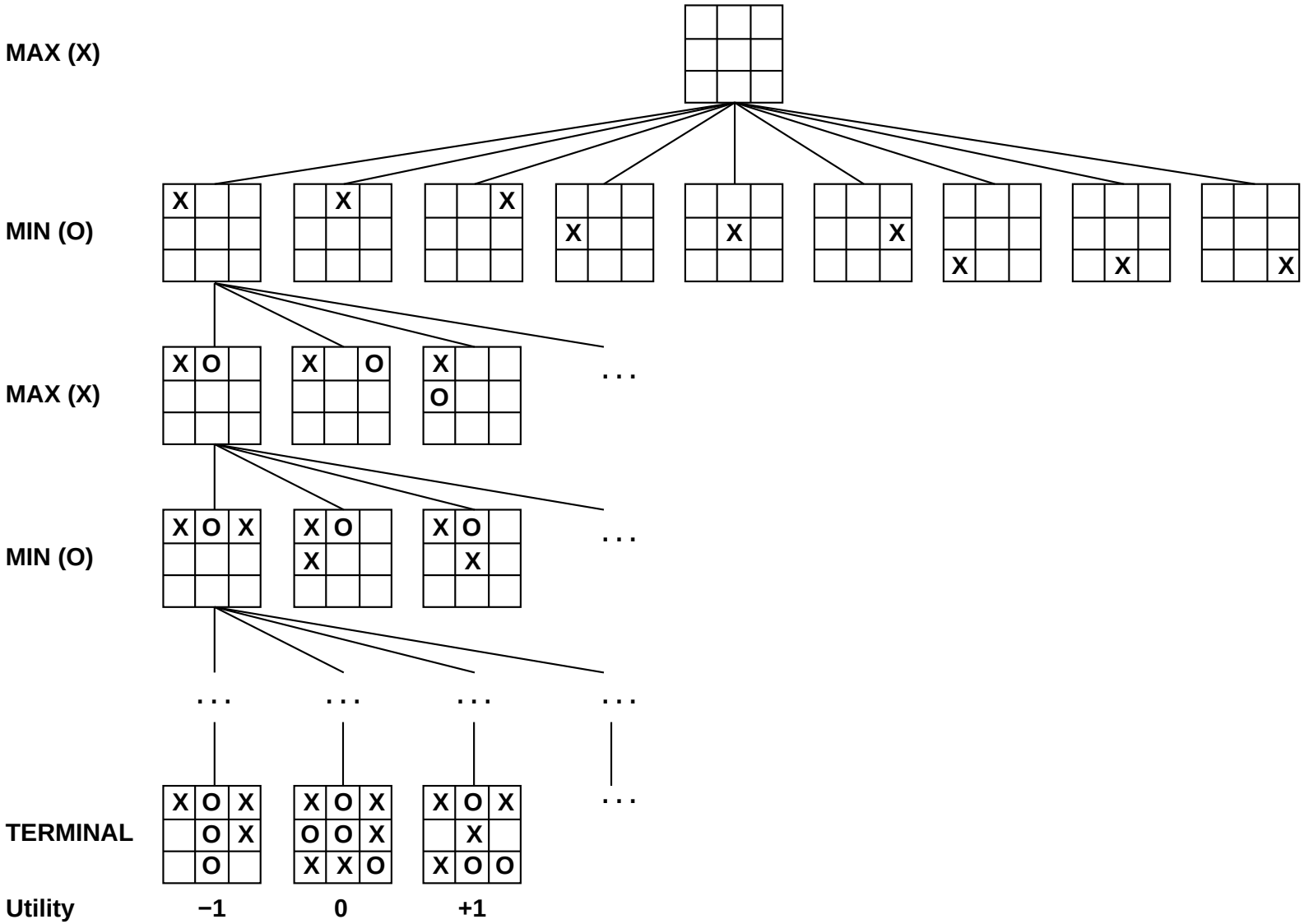
	deterministisch	**Glücksspiel**
vollständige Informationen	Schach, Dame, Go, Othello	Backgammon, Monopoly
unvollständige Informationen	Schiffe versenken, blindes Tic-Tac-Toe	Bridge, Poker, Scrabble, Nuklearer Krieg

blindes Tic-Tac-Toe:

- unvollständige Variante des Standardspiels
- jeder Spieler kann X *und* O setzen
- Gegner erfährt nur welches Feld, aber nicht ob X oder O
- Spieler mit erster Linie mit 3 gleichen Zeichen gewinnt

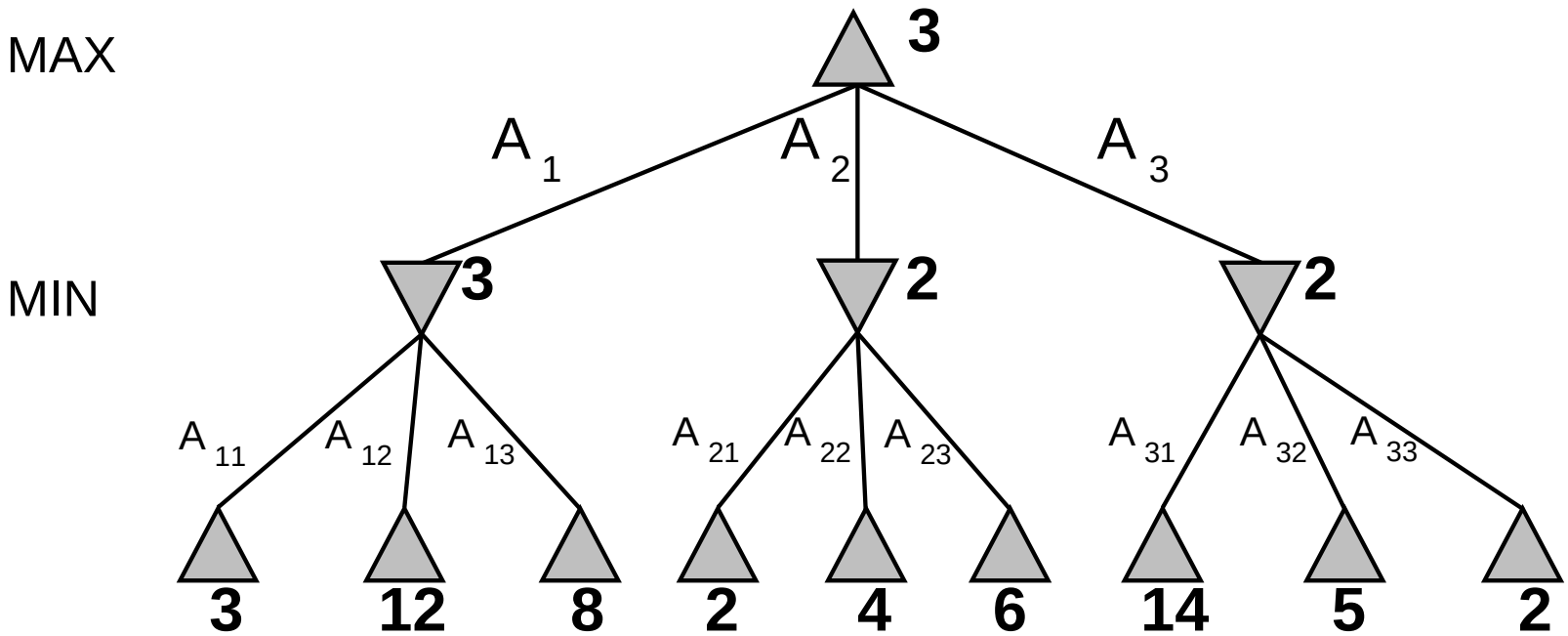
Übersicht

Spielbaum (2 Spieler, deterministisch, Runden)



Minimax

perfektes Spiel für deterministische Spiele mit vollständigen Infos
Idee: wähle Spielzug mit höchstem **Minimax-Wert**
= beste erreichbare Auszahlung gegen besten Spieler
z.B. 2-schichtiges Spiel:



Minimax-Algorithmus

MINIMAX-DECISION

Eingabe: *state*, momentaner Zustand im Spiel
Ausgabe: eine Aktion *action*

1: **return** die Aktion *a* in ACTIONS(*state*), die MIN-VALUE(RESULT(*a*, *state*)) maximiert

MAX-VALUE

```
1: if TERMINAL-TEST(state) {
2:     return UTILITY(state)
3: }
4: v ← −∞
5: for each a, s in SUCCESSORS(state) {
6:     v ←MAX(v,MIN-VALUE(s))
7: }
8: return v
```

MIN-VALUE

```
1: if TERMINAL-TEST(state) {
2:     return UTILITY(state)
3: }
4: v ← ∞
5: for each a, s in SUCCESSORS(state) {
6:     v ←MIN(v,MAX-VALUE(s))
7: }
8: return v
```

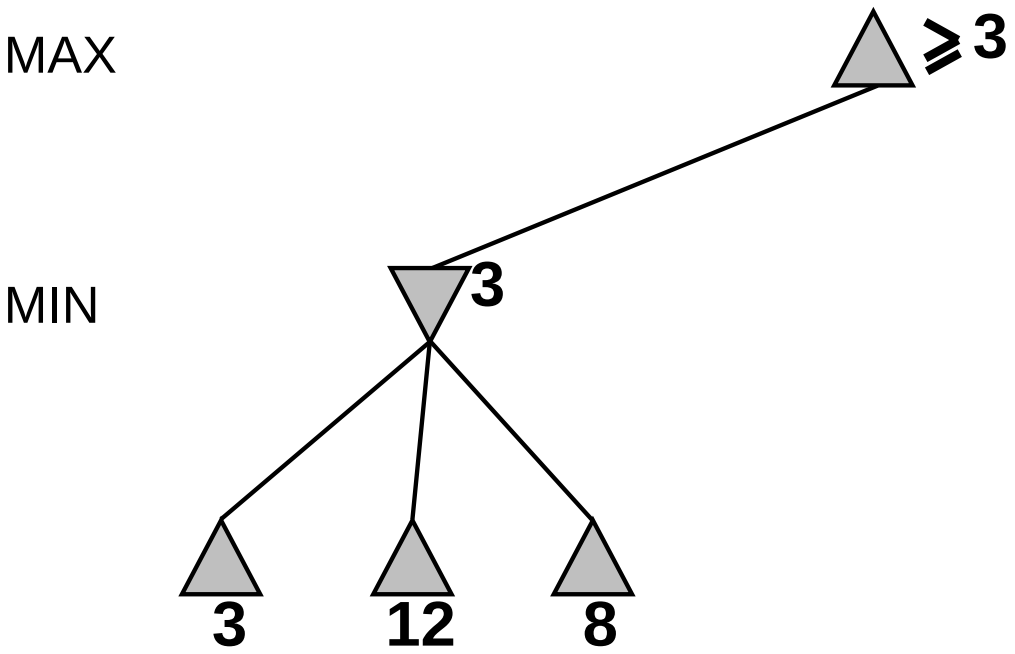
MiniMax: Eigenschaften

Zeit- und Speicherkomplexität gemessen anhand von

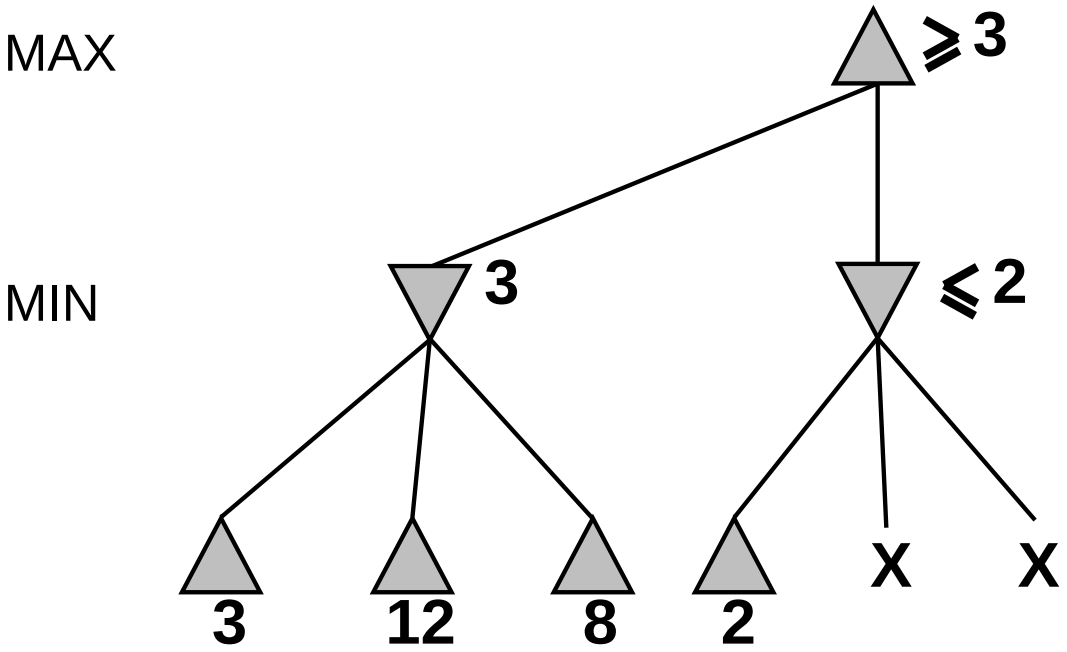
- b: maximalem Verzweigungsfaktor des Suchbaums
- m: maximaler Tiefe des Zustandsraums (eventuell ∞)
- vollständig: ja, falls Baum endlich
- optimal: ja, gegen optimalen Gegner
- Zeit: $O(b^m)$
- Speicher: $O(b \cdot m)$ (Tiefensuche)

für Schach: $b \approx 35, m \approx 100$ bei „realistischen“ Spielen
somit ist die exakte Lösung absolut nicht berechenbar
Aber: muss jeder Pfad exploriert werden?

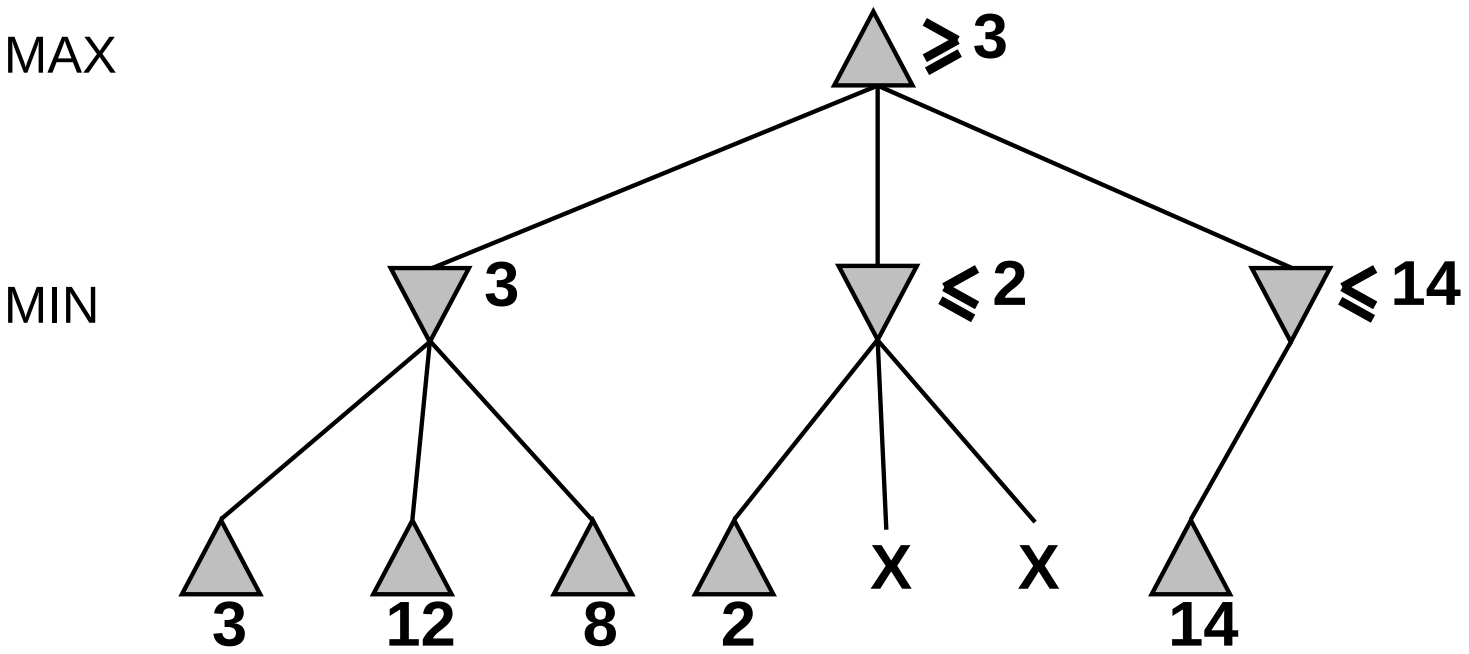
α–β-Stutzen



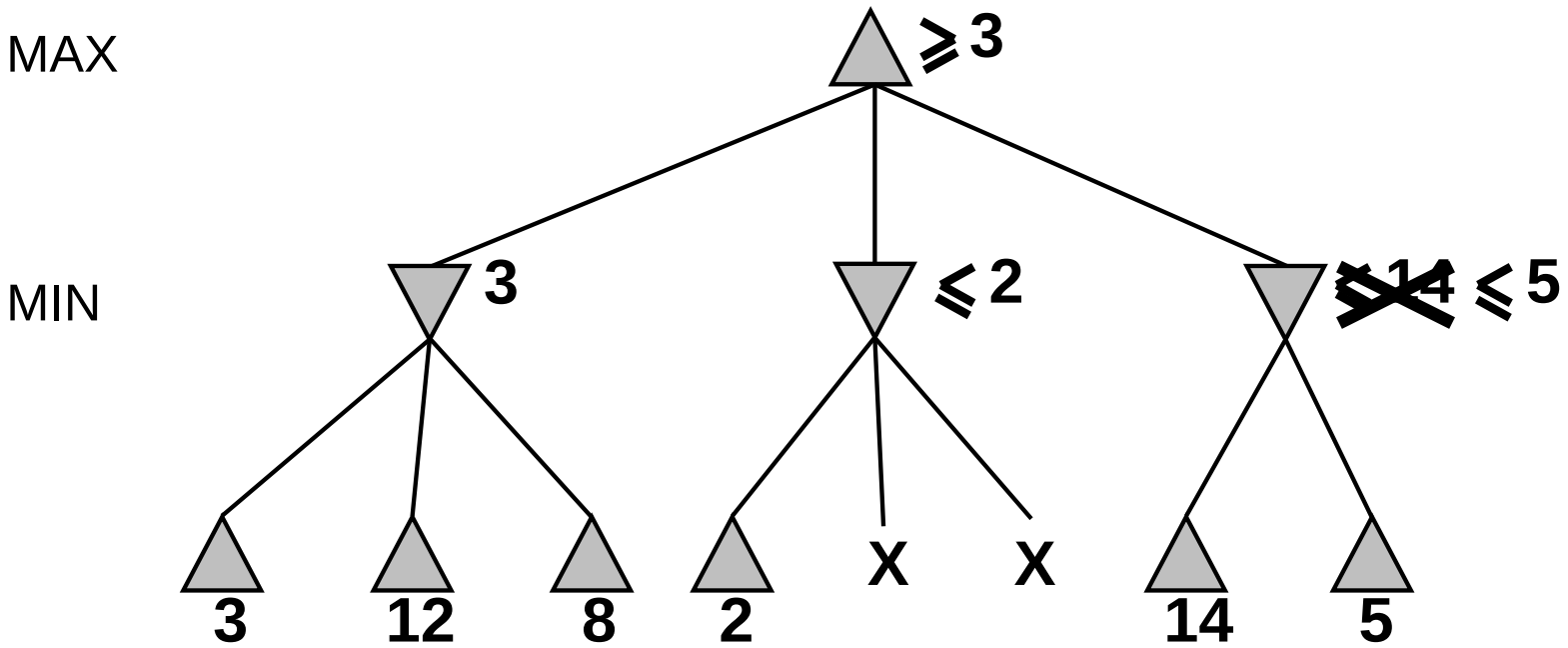
α–β-Stutzen



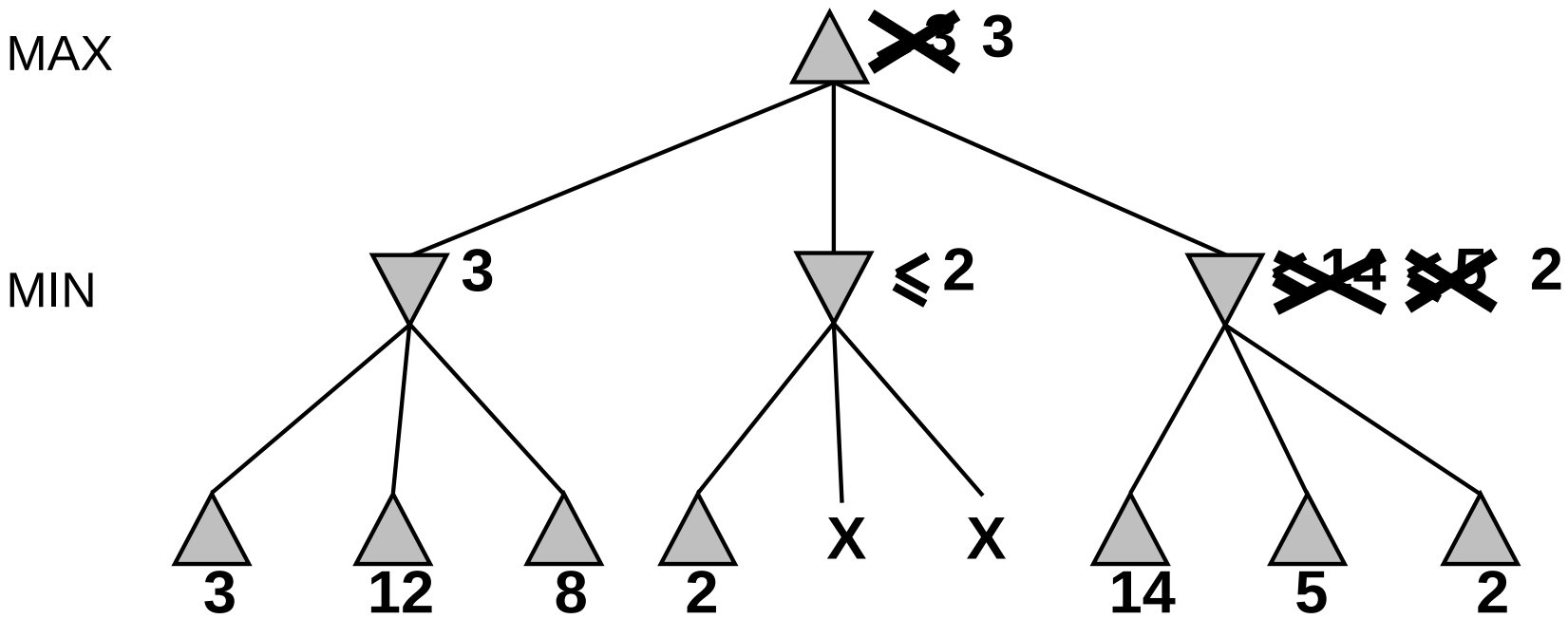
α–β-Stutzen



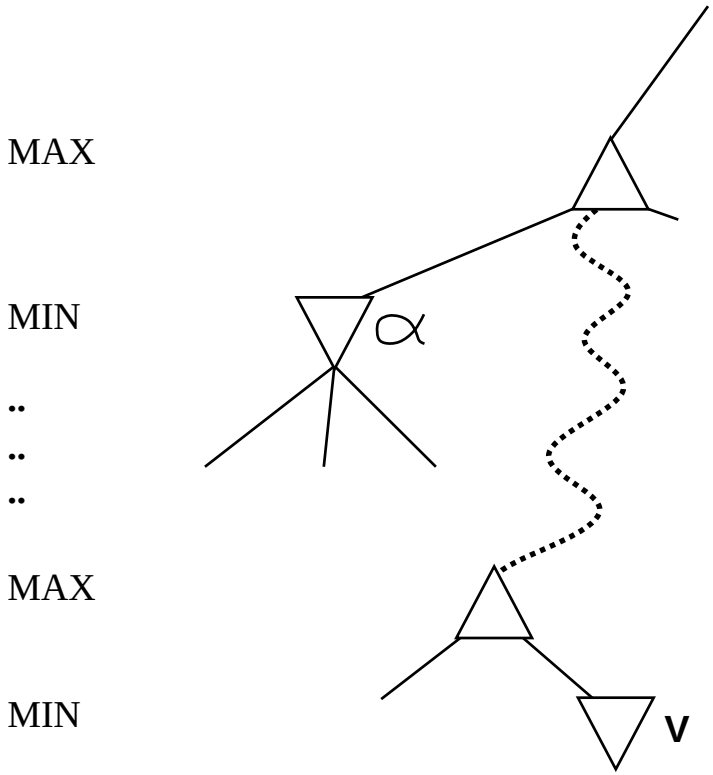
α–β-Stutzen



α–β-Stutzen



Warum heißt es α–β?



α: bester bisher gef. Wert (für $\textsc{Max}$) abseits des aktuellen Wegs
falls V schlechter als α, dann vermeidet $\textsc{Max}$ V und somit wird der Zweig gestutzt
β: analoge Definition für $\textsc{Min}$-Spieler

Der α–β-Algorithmus

ALPHA-BETA-DECISION

1: **return** die Aktion *a* in ACTIONS(*state*), die MIN-VALUE(RESULT(*a*, *state*)) maximiert

MAX-VALUE

Eingabe: *state*, momentaner Zustand im Spiel
α, Wert der besten Alternative für MAX entlang des Pfads zu *state*
β, Wert der besten Alternative für MIN entlang des Pfads zu *state*

```
 1: if TERMINAL-TEST(state) {
 2:     return UTILITY(state)
 3: }
 4: v ← −∞
 5: for each a, s in SUCCESSORS(state) {
 6:     v ←MAX(v,MIN-VALUE(s, α, β))
 7:     if v ≥ β {
 8:         return v
 9:     }
10:     α ← MAX(α, v)
11: }
12: return v
```

MIN-VALUE

1: genau wie MIN-VALUE aber mit vertauschten Rollen von α, β

α–β-Algorithmus: Eigenschaften

Stutzen hat **keine** Auswirkung auf Endergebnis

- gute Zugordnung verbessert Effektivität des Stutzens
- mit „perfekter“ Ordnung, Zeitkomplexität $= O(b^{m/2})$
- somit doppelte Suchtiefe
- für Schach: immer noch 35^{50} möglich

α–β-Algorithmus: Grenzen

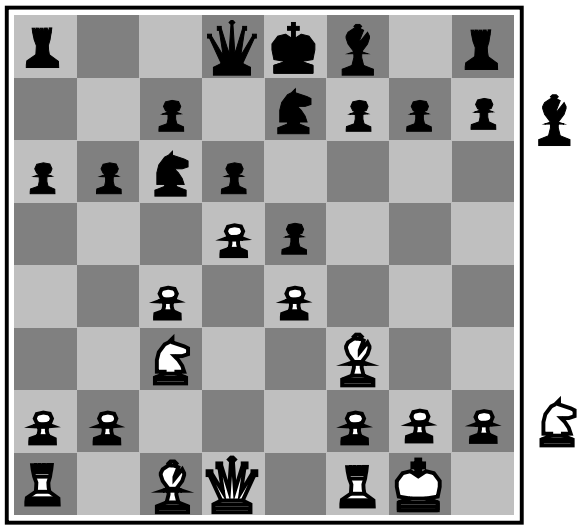
Standard-Ansatz:

- für gewöhnlich: Lösung zu tief im Suchbaum
- Utility mit Werten +1, 0 oder -1 nicht berechenbar
- nutze Cutoff-Test anstelle von Terminal-Test
  z.B. Tiefenbegrenzung (u.U. mit „stiller Suche“—sichtet interessante Pfade tiefer als „stille“)
- nutze Eval (Güteschätzung des Zustands) anstatt Utility:
  Bewertungsfunktion schätzt Begehrtheit einer Stellung

bei 100 Sekunden und 10^4 Knoten/Sekunde:

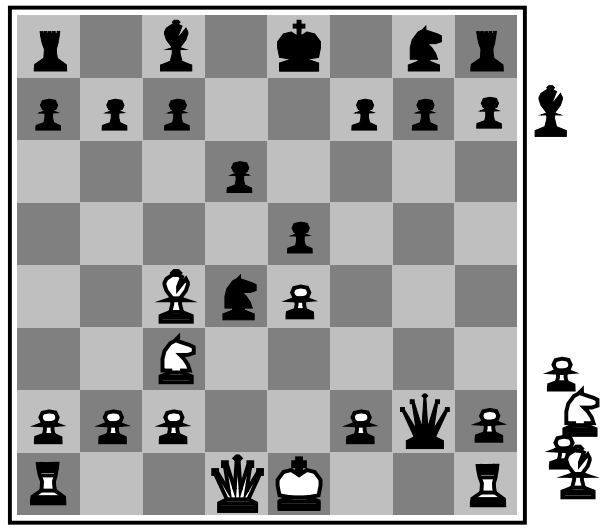
- 10^6 Knoten pro Zug $\approx 35^{8/2}$
- α–β berechnet 8 Halbzüge: ziemlich gutes Schachprogramm

Bewertungsfunktionen



Black to move

White slightly better



White to move

Black winning

für Schach: typischerweise linear gewichtete Summe von n **Merkmalen**

$$\text{Eval}(s) = \sum_{i=1}^{n} w_i \cdot f_i(s)$$

z.B. $f_1(s)$: Differenz von weißen und schwarzen Damen mit $w_1 = 9$

usw.

Deterministische Spiele in der Praxis

Dame:

- 1994 beendete Chinook die 40-jährige Herrschaft des Weltmeisters Marion Tinsley
- Endspieldatenbank mit perfekten Spielen aller Stellungen mit ≤ 8 Steinen ($\geq 443 \cdot 10^9$ Stellungen)

Schach:

- Deep Blue besiegte 1997 Weltmeister Garri Kasparow in 6 Spielen
- $200 \cdot 10^6$ Stellungen/Sekunde, sehr komplizierte Bewertung
- bis zu 40 Halbzüge tief (nicht veröffentlichte Methoden)

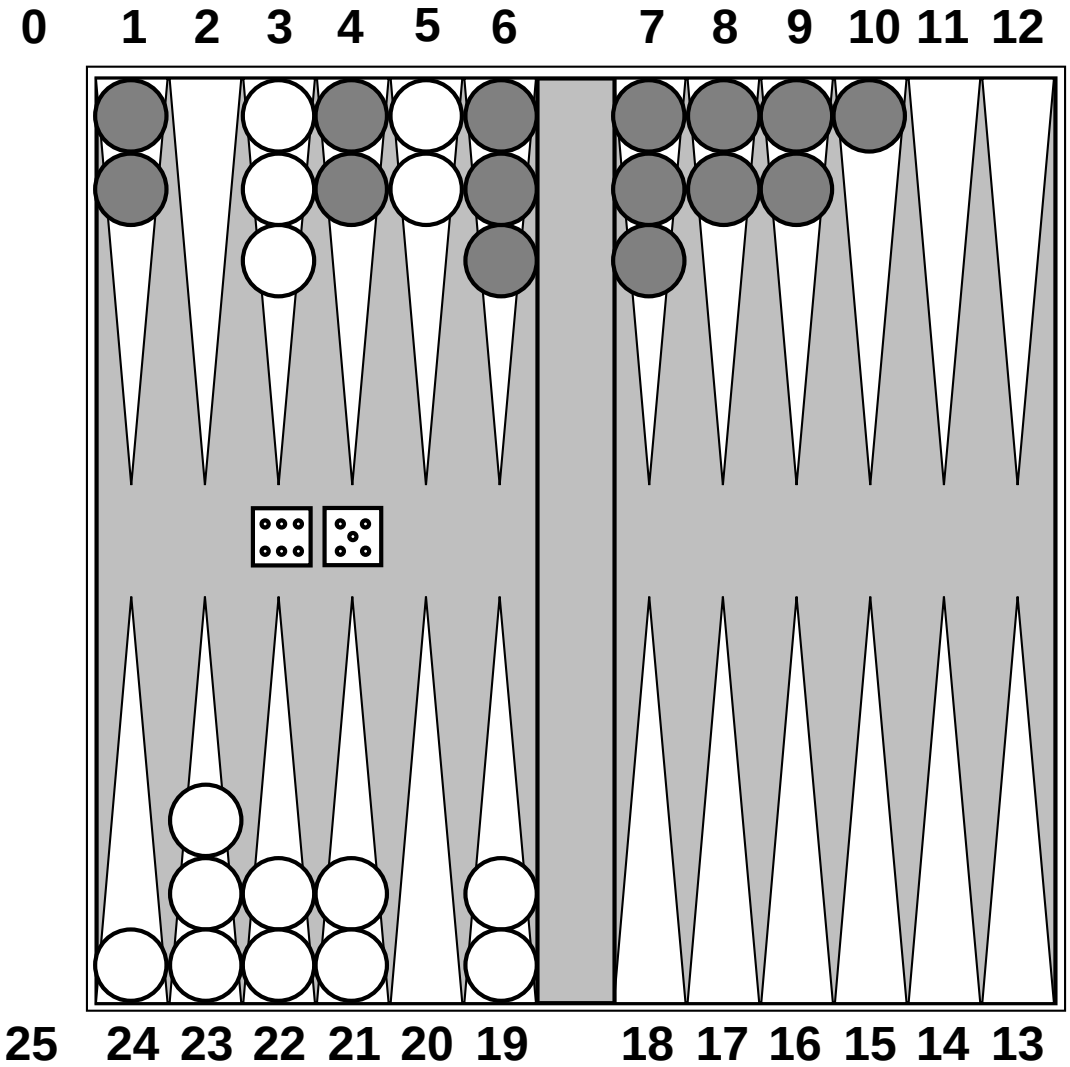
Othello: Wettbewerb nur unter menschlichen Meistern

Go:

- Wettbewerb nur unter menschlichen Meistern
- $b > 300$, daher Datenbanken mit Mustern für plausible Züge

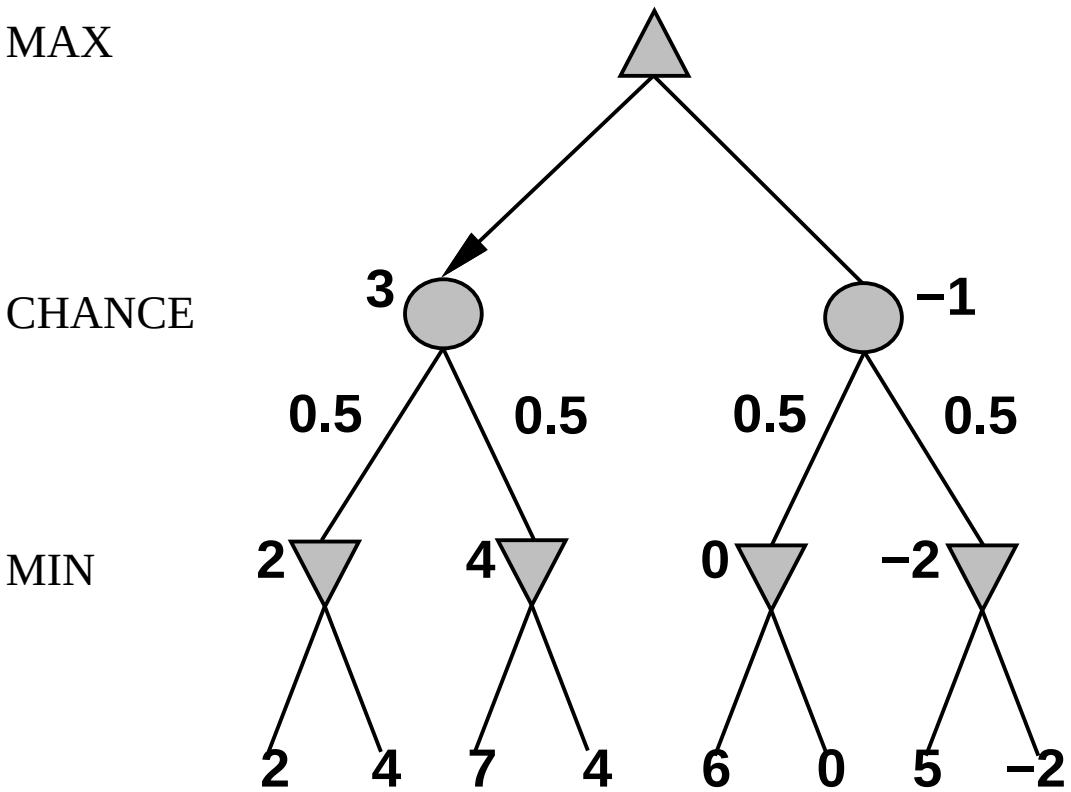
Übersicht

Glücksspiele: Backgammon



Glücksspiele allgemein

Zufall aufgrund von Würfeln, Mischen von Karten, etc.
vereinfachtes Beispiel mit Münzwurf:



Algorithmus für nichtdeterministische Spiele

ExpectiMinimax liefert perfektes Spiel
wie Minimax, nur Zufallsknoten müssen behandelt werden:

```
...
if state is a Max node {
    return highest ExpectiMinimax-Value of Successors(state)
}
if state is a Min node {
    return lowest ExpectiMinimax-Value of Successors(state)
}
if state is a chance node {
    return average of ExpectiMinimax-Value of Successors(state)
}...
```

Nichtdeterministische Spiele in Praxis

jeder Würfelwurf erhöht Verzweigungsfaktor b: 21 mögliche Würfe mit 2 Würfeln

Backgammon $\approx$ 20 erlaubte Züge (bei einem 1–1 Wurf können es 6000 sein)

$$\text{Tiefe } 4 \rightarrow 20 \cdot (21 \cdot 20)^3 \approx 1,2 \cdot 10^9 \text{ Zustände}$$

mit steigender Tiefe: Wahrscheinlichkeit sinkt geg. Knoten zu erreichen

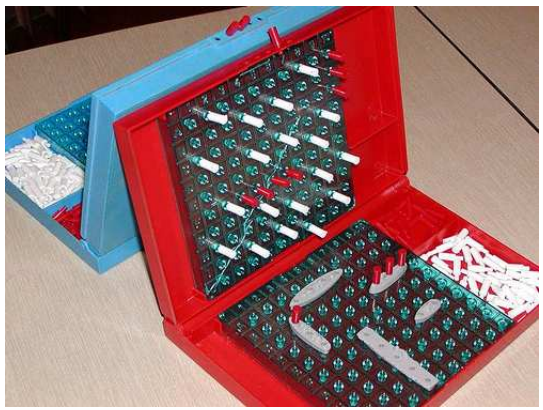
damit wird Wert des Vorgriffs vermindert

α–β-Stutzen viel weniger effektiv

TDGammon benutzt beschränkte Tiefensuche mit $d = 2 +$ sehr gute Eval: vergleichbar mit menschlichem Weltmeister

Spiele mit unvollständigen Informationen

- die meisten Kartenspiele (wie Bridge, Doppelkopf, Hearts, Mau-Mau, Poker, Siebzehn und vier, Skat) sind Nullsummenspiele mit unvollständigen Informationen
- auch einige Brettspiele (Schiffe versenken, Kriegspiel-Schach)



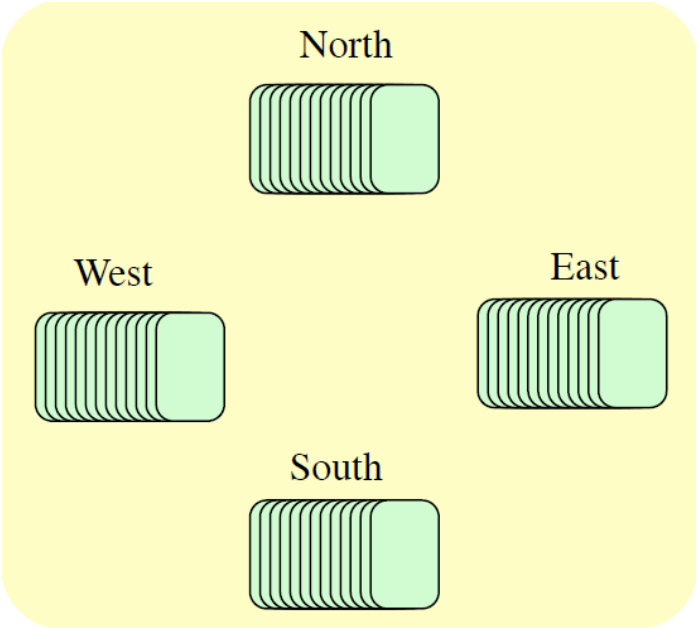
Beispiel: Bridge

4 Spieler

- Nord und Süd sind Partner
- Ost und West ebenfalls

Kartenstapel mit 52 Spielkarten
Phasen des Spiels:

- Karten ausgeben (gleichverteit auf alle 4 Spieler)
- Reizen (verhandeln, welche Farbe Trumpf ist)
- Spielen der Karten



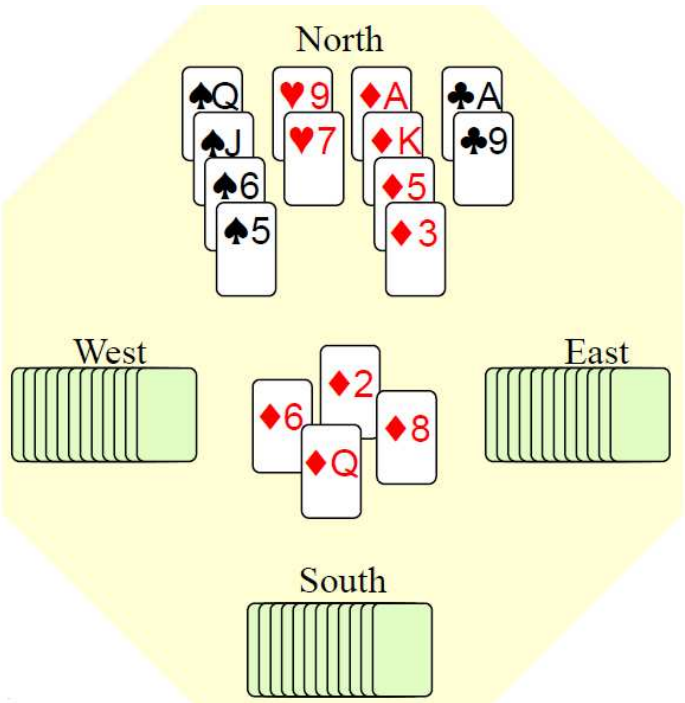
Bridge: Spielen der Karten

Alleinspieler: Spieler, der Trumpf wählt
Dummy: Partner des Alleinspielers

- deckt seine Karten auf
- Alleinspieler spielt seine Karten und die des Dummy

Stich: grundlegende Einheit des Spiels

- ein Spieler spielt Karte an
- anderen müssen bedienen (wenn möglich)
- Stich ist gewonnen bei höchster Karte der Farbe die angespielt wurde (solange niemand einen Trumpf gespielt hat)



läuft so weiter bis alle Karten gespielt wurden
Punkte basieren auf Anzahl der gereizten und gemachten Stiche

Bridge: Spielbaumsuche

unvollständige Informationen in Bridge:

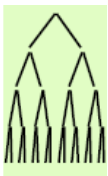
- unbekannt: welche Karten haben die anderen (bis auf Dummy)?
- viele mögliche Kartenverteilungen, sehr viele mögliche Züge

wenn man zusätzlichen Züge als zusätzliche Zweige im Suchbaum kodiert, dann erhöht sich der Verzweigungsfaktor b

Anzahl der Knoten ist exponential zu b

- worst case: ca. $6 \cdot 10^{44}$ Blattknoten
- im Durchschnitt: ca. 10^{24} Blattknoten

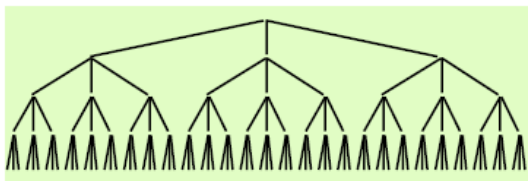
$b = 2$



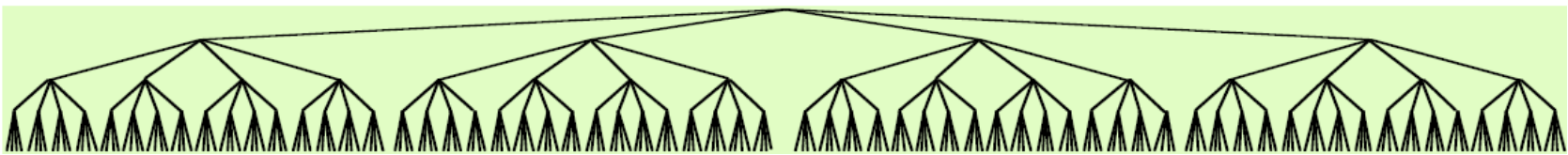
eine Partie Bridge dauert ca. 1,5 Minuten

- nicht genügend Zeit, um Baum zu durchsuchen

$b = 3$



$b = 4$



Ansatz: Monte-Carlo-Simulation

1. generiere viele Zufallshypothesen, die beschreiben wie die Karten verteilt sein können
2. generiere und durchsuche die so erzeugten Spielbäume
3. wähle die Aktion, die durchschnittlich die meisten Stiche macht

dieser Ansatz hat einige theoretische Probleme:

- die Suche ist nicht in der Lage zu unterscheiden zwischen: gewollte Züge, um Informationen zu sammeln und gewollte Züge, um andere zu täuschen

MC-Simulation kann Größe des Suchbaums um Faktor $5,2 \cdot 10^6$ verkleinern

- $6 \cdot 10^{44} / 5,2 \cdot 10^6 = 1.1 \cdot 10^{38}$ (immer noch viel)
- MC-Simulation allein reicht also nicht!
- für gewöhnlich mit *Zustandsaggregation* kombiniert

Transpositionstabelle

- es ist möglich einen gegebenen Zustand im Spiel auf mehreren Wegen zu erreichen
- diese heißen Transpositionen
- generell: nach n Schritten kann es höchstens $(n!)^2$ Transpositionen geben
- die meisten dieser Züge sind illegal, jedoch ist es wahrscheinlich, dass ein Programm einen Zustand mehrmals analysiert
- Lösung: Transpositionstabelle
- Hash-Tabelle von jedem bisher analysierten Zustand
- bei neuem Zustand wird überprüft, ob dieser schon ausgewertet wurde

Zustandsaggregation

modifizierte Version der Transpositionstabellen

- jeder Hash-Tabelleneintrag repräsentiert eine Menge von Positionen, die als gleichwertig gesehen werden
- z.B.: ♠$AQ532$ sei äquivalent zu ♠$AQxxx$

vor der Suche: erst einen Hash-Tabelleneintrag finden

- reduziert den Verzweigungsfaktor des Suchbaums
- berechneter Wert für einen Zweig wird in der Tabelle gespeichert und benutzt als Wert für ähnliche Zweige

momentane Bridge-Programme kombinieren dies mit MC-Simulation

z.B. GIB, das momentan beste Bridge-Programm

Beispiel: Poker

Unsicherheitsquellen:

- Kartenverteilung
- Strategie des Gegners (z.B. wann wird geblufft, wann ausgestiegen)

momentan viel KI für "Texas Hold'Em"

- 5 offene Karten in die Mitte des Tisches, die von jedem Spieler zur Bildung seiner Pokerhand verwendet werden können
- jeder Spieler darf nicht mehr als 2 seiner Hand-Karten verwenden

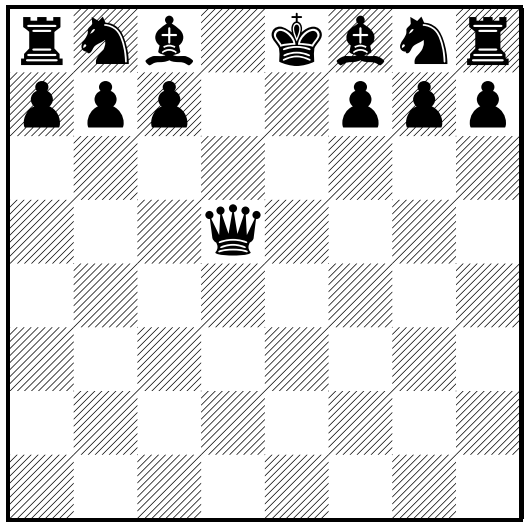
besten KI-Programme nähern sich allmählich den menschl. Experten

- Konstruktion eines statistischen Modells des Gegners (welche Einsätze sind unter welchen Umständen wahrscheinlich?)
- Kombination mit spieltheoretischen Techniken (lineare Programmierung und MC-Simulation)

Beispiel: Kriegspiel-Schach

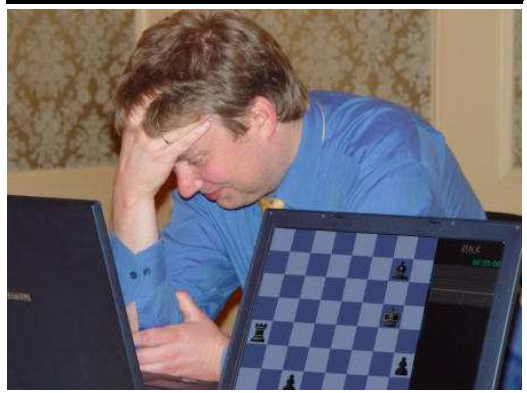
unvollständige Variante des Schach

- 1824 entwickelt von einem Preußischen Militäroffizier
- wurde bekannt als Aufgabe fürs Militär-Training
- Vorläufer von modernen Militärkriegsspielen



ähnlich einer Kombination aus Schach und Schiffe versenken

- Figuren starten in normaler Position, aber man sieht die Züge seines Gegners nicht



nur wie folgt kommt man an Informationen:

- man schlägt eine Figur oder verliert eine
- eigener König wird Schach gesetzt
- man macht einen ungültigen Zug

Beispiel: Kriegspiel-Schach

jeder Spieler versucht normale Schachzüge auszuführen

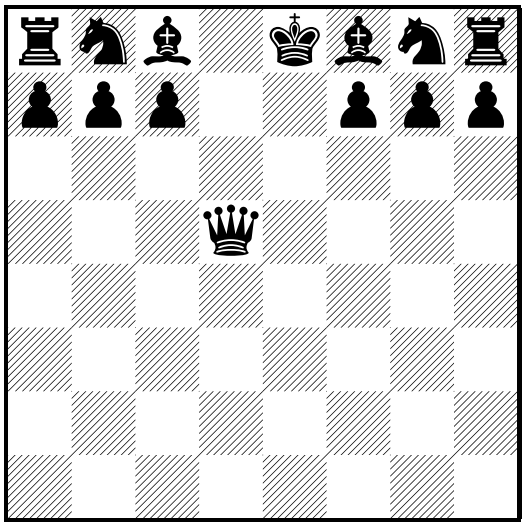
- falls der Zug ungültig ist, erfährt man, dass man einen anderen Zug versuchen soll

wenn eine Figure geschlagen wird, erfahren es beide Spieler

- man erfährt den Quadranten, jedoch nicht die Art der Figur

wenn ein legale Zug ein Schach, ein Schachmatt oder einen Patt für den Gegner verursacht, erfahren es beide Spieler

- man erfährt, ob Schach verursacht wurde durch lange/kurze Diagonale, Reihe, Linie, Springer (oder Kombination derer)



Beispiel: Kriegspiel-Schach

Größe der Informationsmenge (Menge aller Zustände, in denen man sich *möglicherweise* befindet):

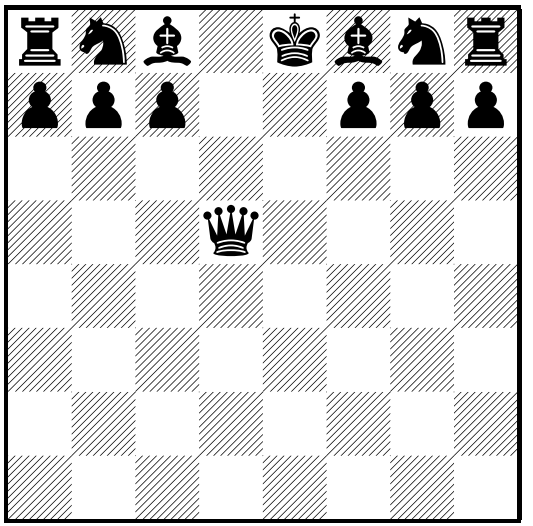
Schach:	1
Texas hold'em:	10^3
Bridge:	10^7
Kriegspiel:	10^{14}

bei Bridge oder Poker resultiert die Unsicherheit durch das zufällige Austeilen der Karten

- Wahrscheinlichkeitsverteilung ist leicht zu berechnen

beim Kriegspiel resultiert die Unsicherheit daher, dass die gegnerischen Züge nicht sichtbar sind

- kein einfacher Weg, um eine geeignete Wahrscheinlichkeitsverteilung zu bestimmen



Kriegspiel: MC-Simulation

grundlegende Idee: für $i = 1, \ldots, n$

1. generiere einen Suchbaum T mit perfekten Informationen durch Erraten der gegnerischen Züge
2. berechne einen Wert v für T durch Minimax-Suche
3. wiederhole Schritte 1 und 2 n-mal und gebe Mittelwert zurück

Probleme:

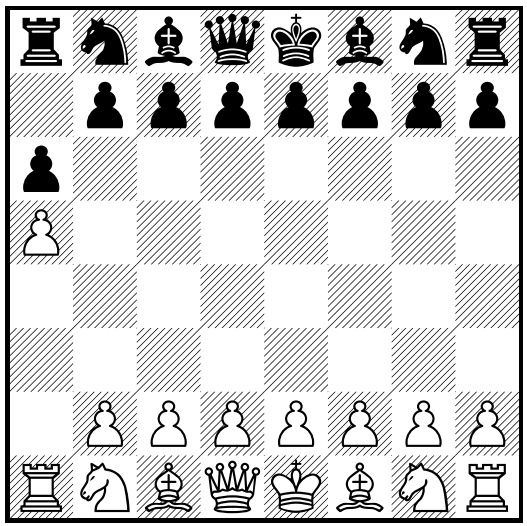
- schwer eine Sequenz zu generieren, die zu den bekannten Informationen konsistent ist (Zeit ist exponentiell in der Anzahl der Züge, die gespielt wurden)
- Kompromiss: beschränkte Anzahl von Bäumen, beschränkte Suchtiefe
- keine Schlussfolgerungen über „informationsgewinnende Züge“ möglich

Kriegspiel: Informationsmengen

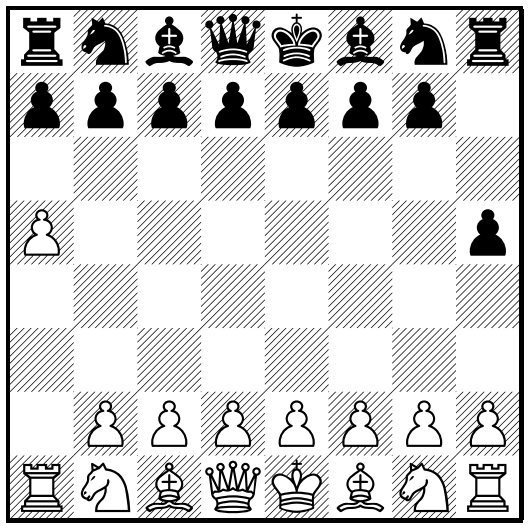
betrachten wir folgende Kriegspiel-Historie: $\langle a2-a4, h7-h6, a4-a5 \rangle$
Weiß weiß nicht, dass Schwarz den Zug $h7-h6$ gemacht hat

- Informationsmenge von Weiß hat 19 Zustände
- warum nicht 20? 15 versch. Bauernzüge, 4 versch. Pferdzüge

$\langle a2-a4, a7-a6, a4-a5 \rangle$



$\langle a2-a4, h7-h5, a4-a5 \rangle$

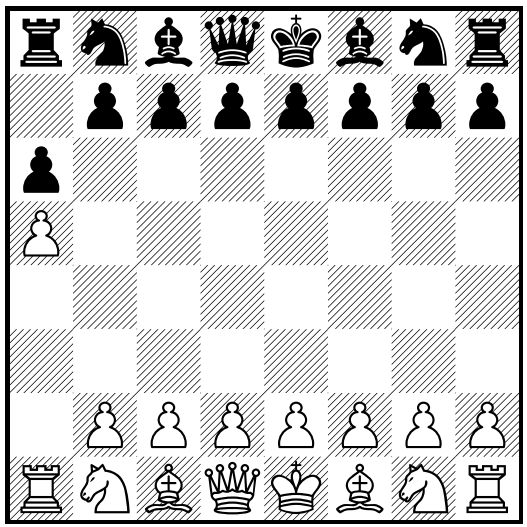


- in Kriegspiel mit Verzweigungsfaktor b existieren nicht mehr als b^n Zustände, wenn der Gegner n unbeobachtbare Züge macht

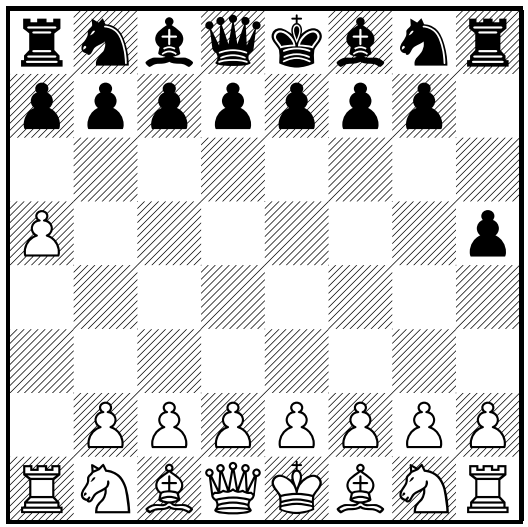
Kriegspiel: Informationsmengen

Einige Züge werden die Größe der Informationsmenge reduzieren

$\langle a2 - a4, a7 - a6, a4 - a5 \rangle$



$\langle a2 - a4, h7 - h5, a4 - a5 \rangle$



Schwarz hatte 20 versch. Mögl. für den ersten Zug (inkl. $h7 - h5$)

- bevor Weiß $a4 - a5$ zog, hatte Informationsmenge von Weiß 20 Zustände

$a4 - a5$ reduziert Informtionsmenge von Weiß von 20 auf 19 Zustände

- wenn Schwarz auf $h5$ wäre, könnte Weiß diesen Zug nicht machen

Kriegspiel: Informationsgewinnende Züge

Spieler nutzen häufig Bauern, um Informationen zu gewinnen

- Bauer zieht vorwärts, außer beim Schlagen
- beim Schlagen zieht er diagonal

Angenommen, es wird versucht, diagonal vorwärts zu kommen:

- wenn der Zug ungültig ist, wird es der Schiedsrichter sagen man weiß dann, dass der Gegner dort keine Figur hat und kann nochmal ziehen
- wenn der Zug gültig ist, weiß man, dass der Gegner dort eine Figur hatte und schlägt diese

Zusammenfassung

Die Erforschung von Spielen und deren Lösungen ist herausfordernd (und gefährlich zugleich. . . Ethik!)
Illustration verschiedener wichtiger Punkte über KI

- wenn Perfektion unerreichbar, dann approximieren
- Unsicherheit beschränkt Zuweisung von Werten zu Zuständen
- optimale Entscheidungen hängen ab vom Informationszustand, nicht vom wirklichen Zustand

Spiele sind für die Forschung in der künstlichen Intelligenz, was der Grand Prix für die Automobilentwicklung ist!