INF FAKULTÄT FÜR INFORMATIK

Intelligente Systeme

Heuristische Suchalgorithmen

Prof. Dr. R. Kruse C. Braune C. Doell

{kruse,cbraune,doell}@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

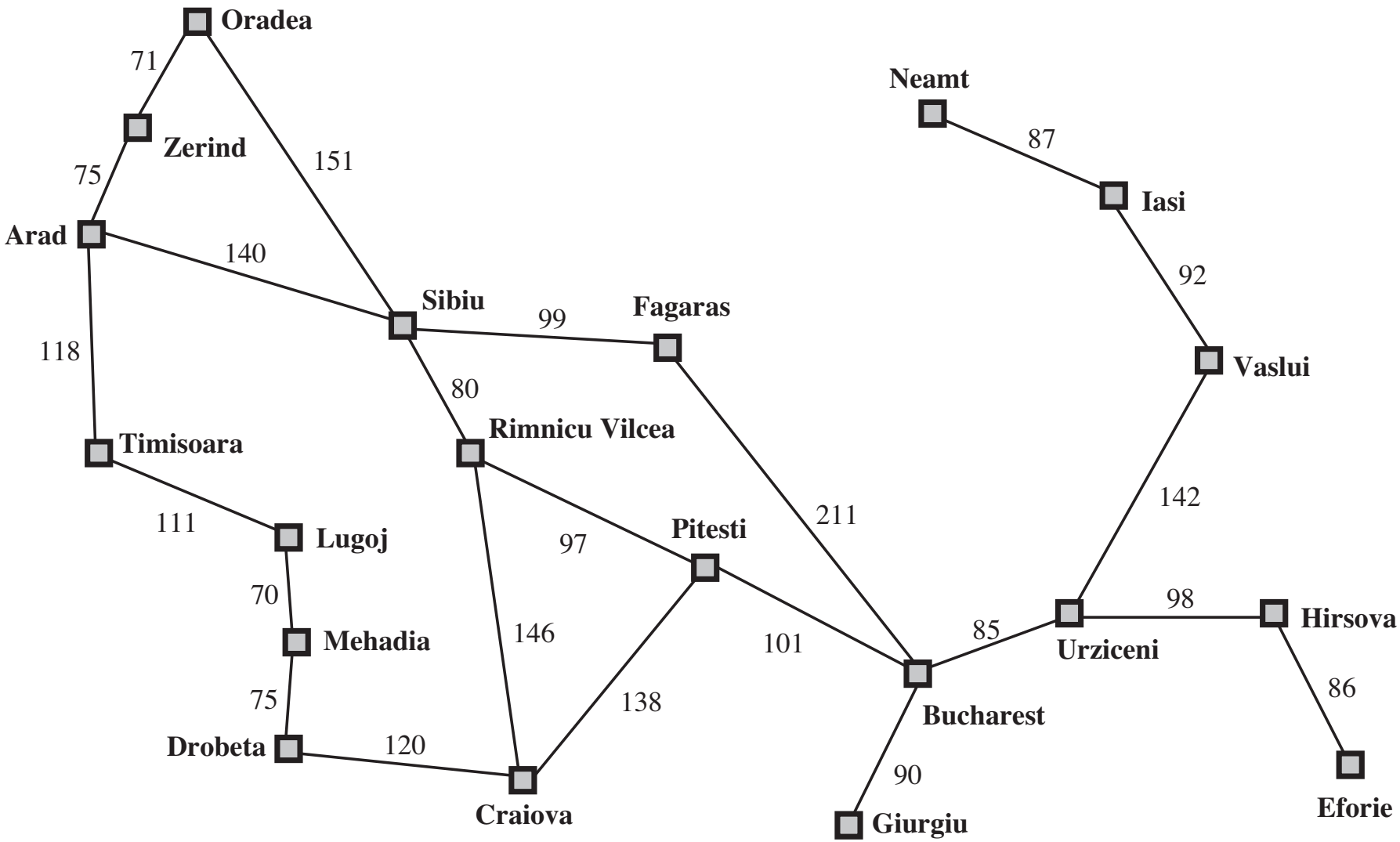
Erzeugung von Nachfolgern bereits erkundeter Zustände (sog. expandierte Zustände)

Tree-Search

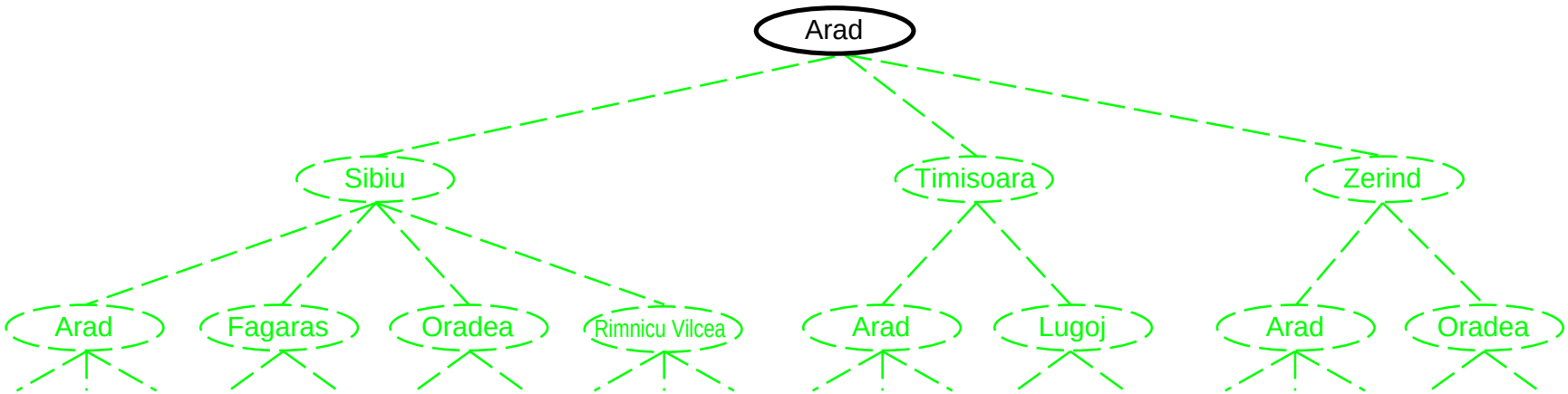
Eingabe: Problembeschreibung *problem*, Vorgehensweise *strategy*
Ausgabe: Lösung oder Fehler

```
 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: }
```

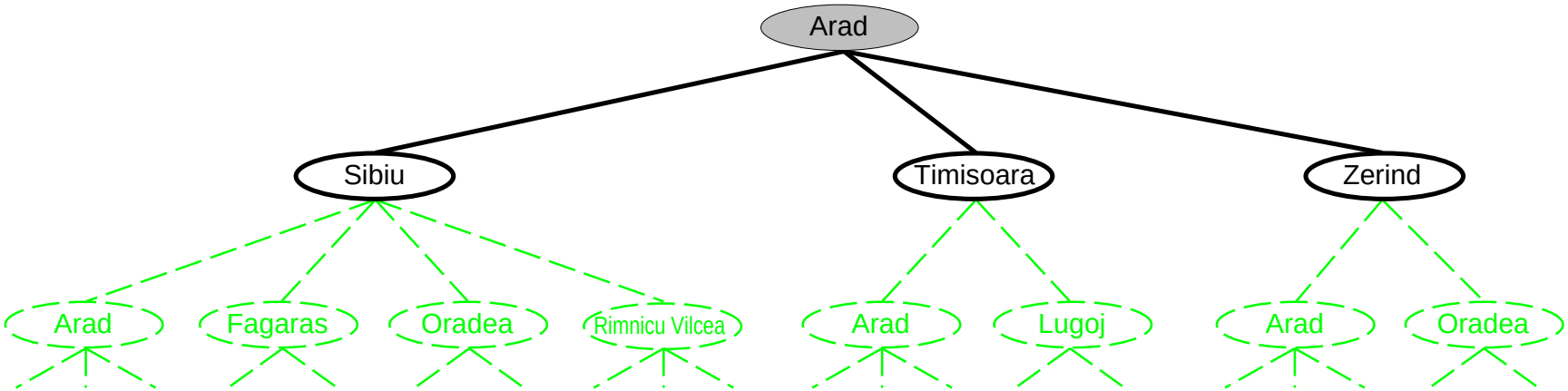
Beispiel: Routenplanung



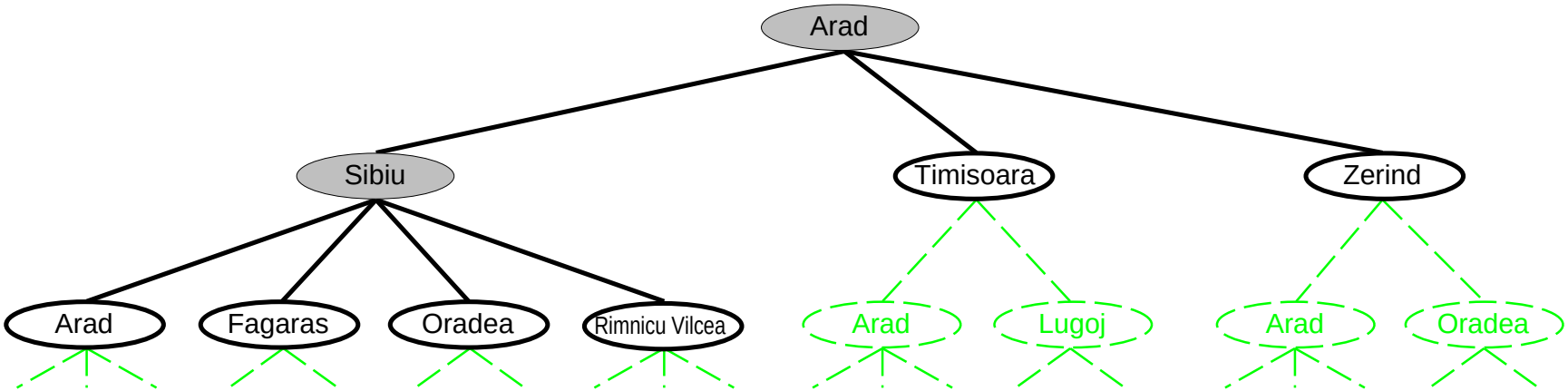
Beispiel: Baumsuche



Beispiel: Baumsuche



Beispiel: Baumsuche



Uninformierte Suchstrategien

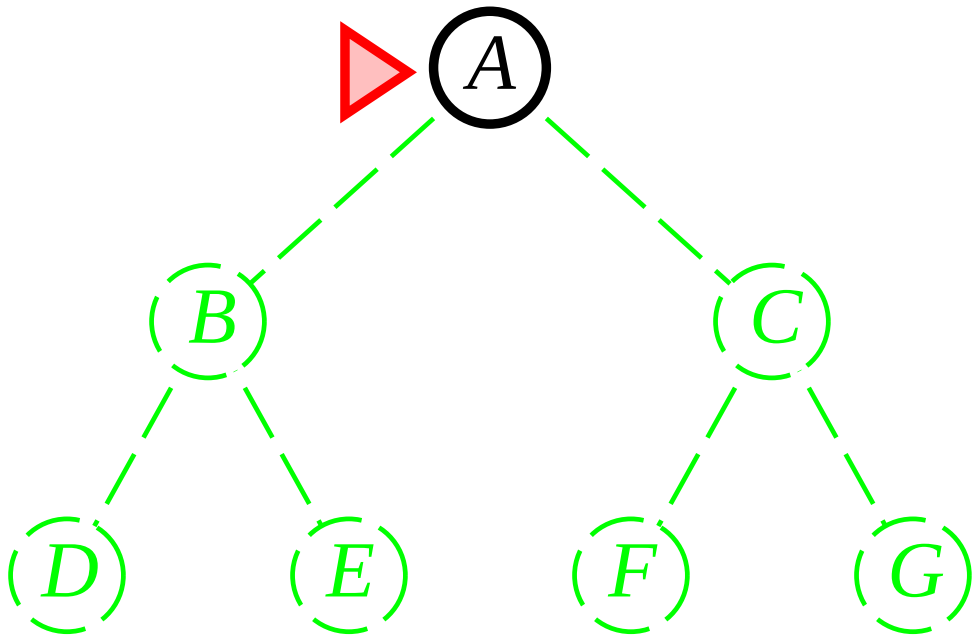
Uninformierte Suchstrategien nutzen nur verfügbare Informationen der Problemdefinition.

- Breitensuche
- Uniforme Kostensuche
- Tiefensuche
- Beschränkte Tiefensuche
- Iterative Tiefensuche

Breitensuche (engl. *breadth-first search*)

Expandiere „seichtesten“, nicht-expandierten Knoten!

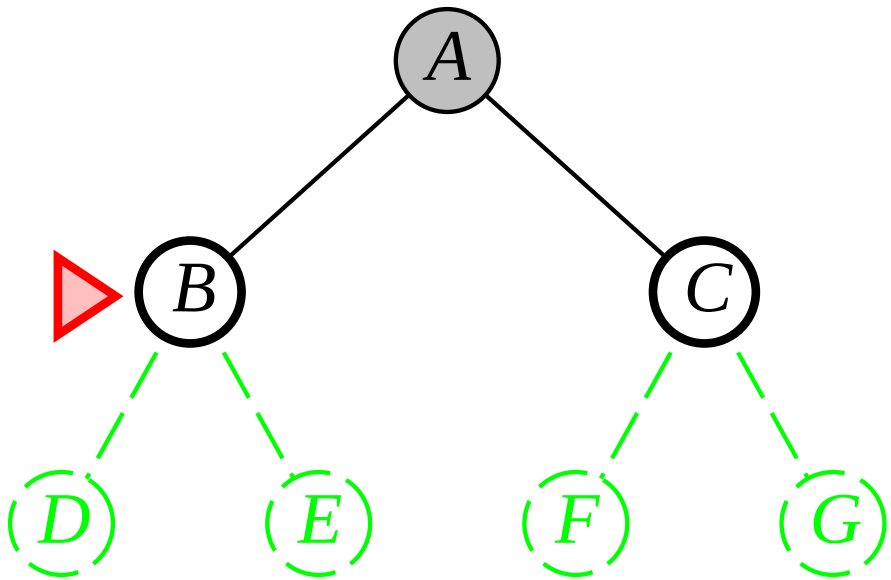
Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



Breitensuche (engl. *breadth-first search*)

Expandiere „seichtesten“, nicht-expandierten Knoten!

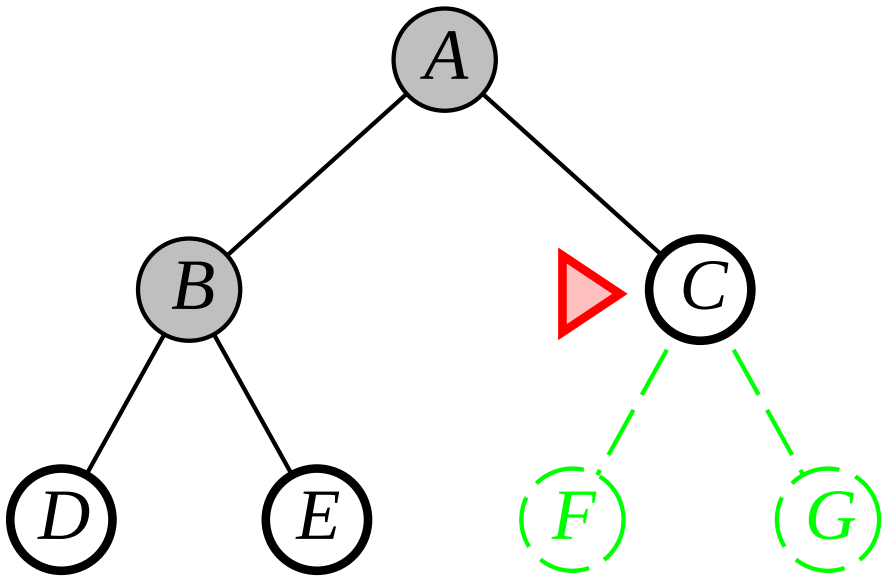
Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



Breitensuche (engl. *breadth-first search*)

Expandiere „seichtesten“, nicht-expandierten Knoten!

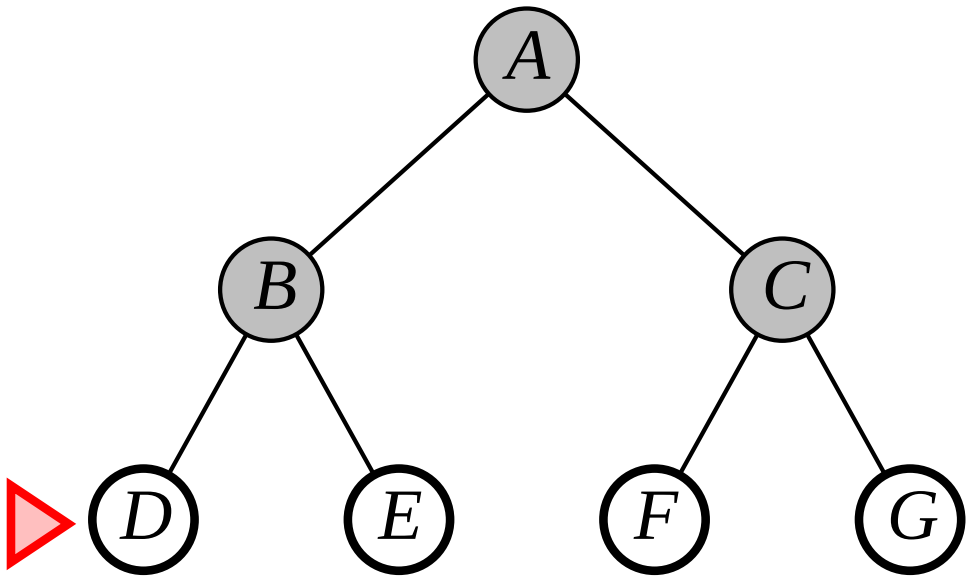
Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



Breitensuche (engl. *breadth-first search*)

Expandiere „seichtesten“, nicht-expandierten Knoten!

Implementierung: *fringe* = FIFO-Schlange (neue Nachfolger ans Ende)



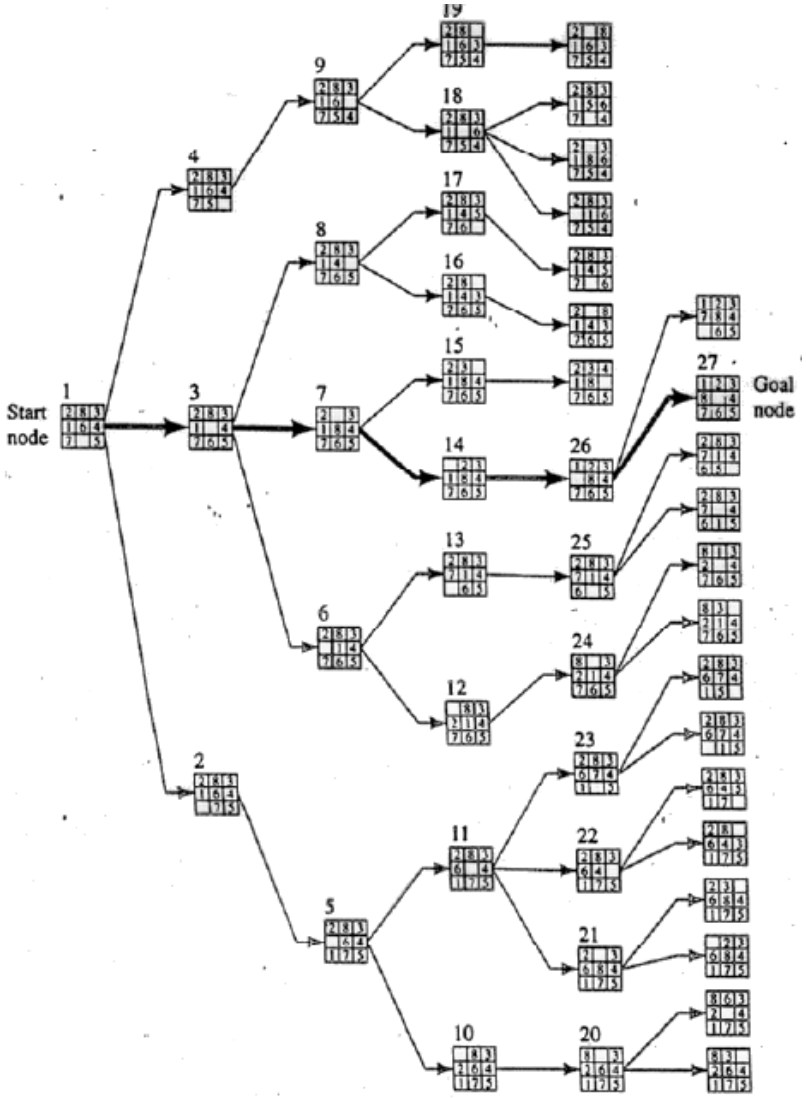
Breitensuche: 8-Puzzle

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)

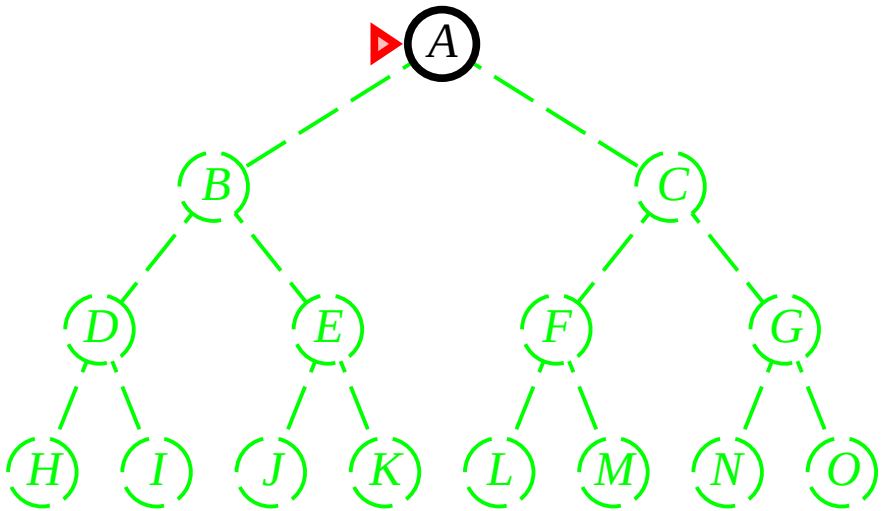
Optimal: falls Kosten $= 1$ pro Schritt, generell nicht

Größtes Problem: Speicher

Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

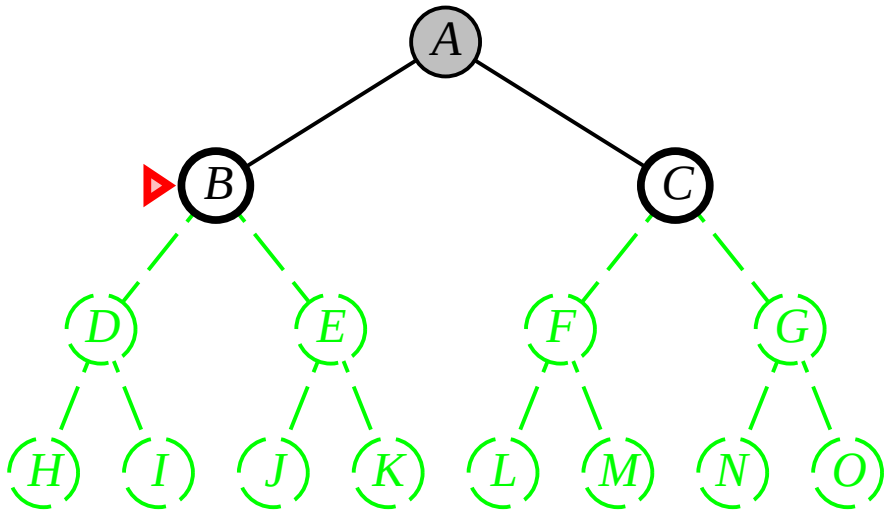
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

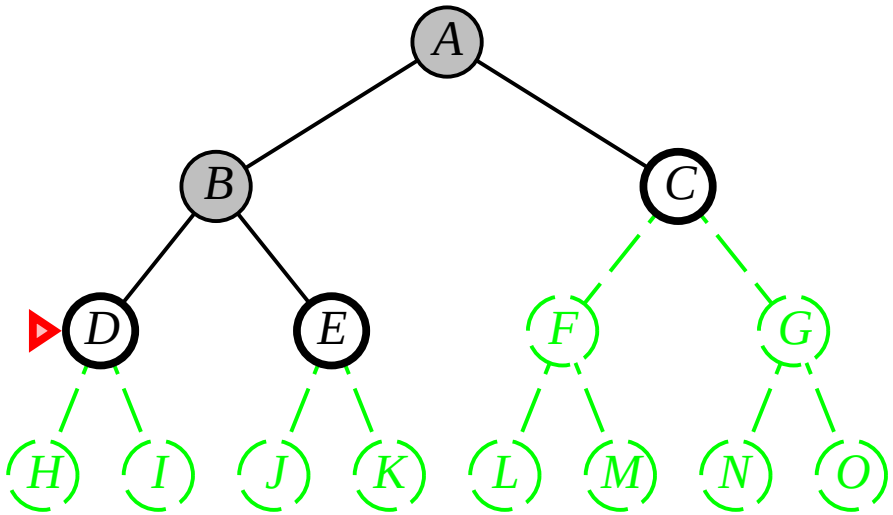
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

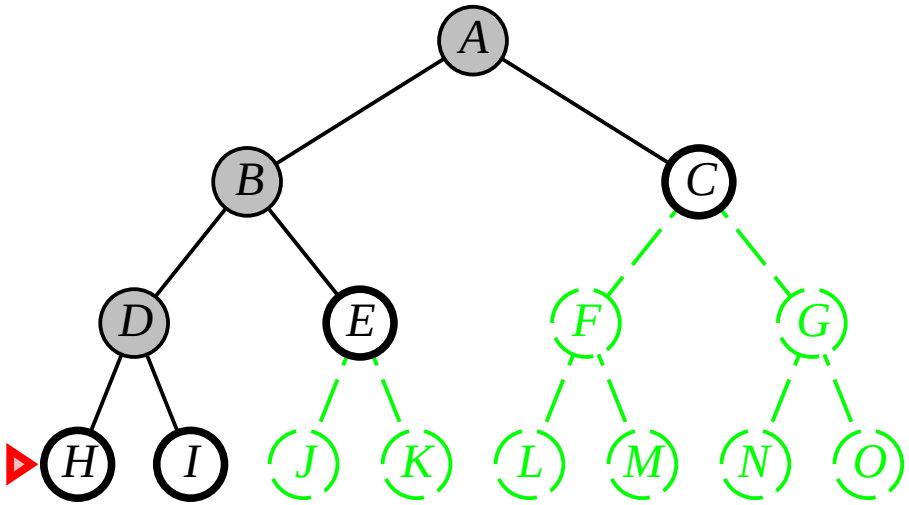
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

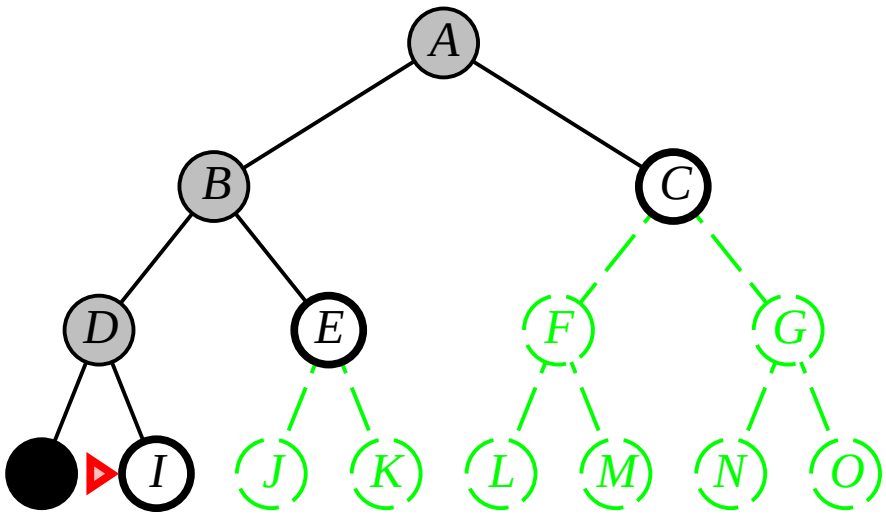
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

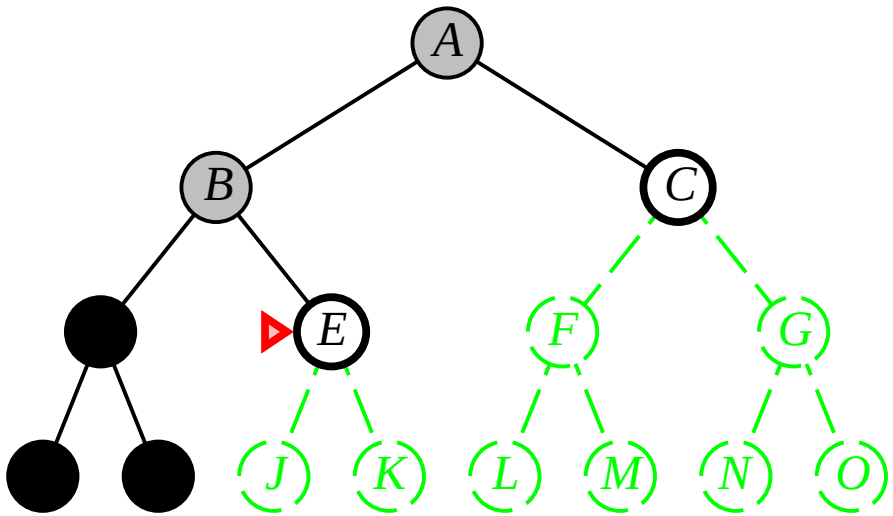
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

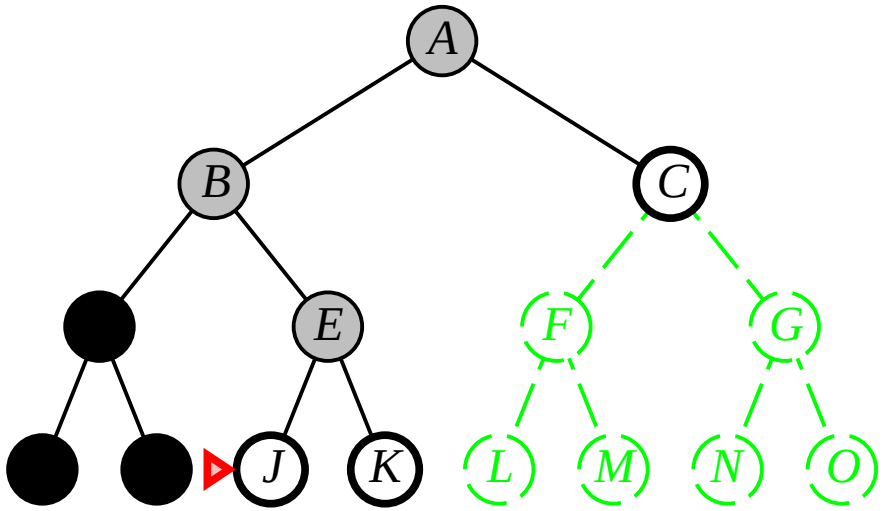
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

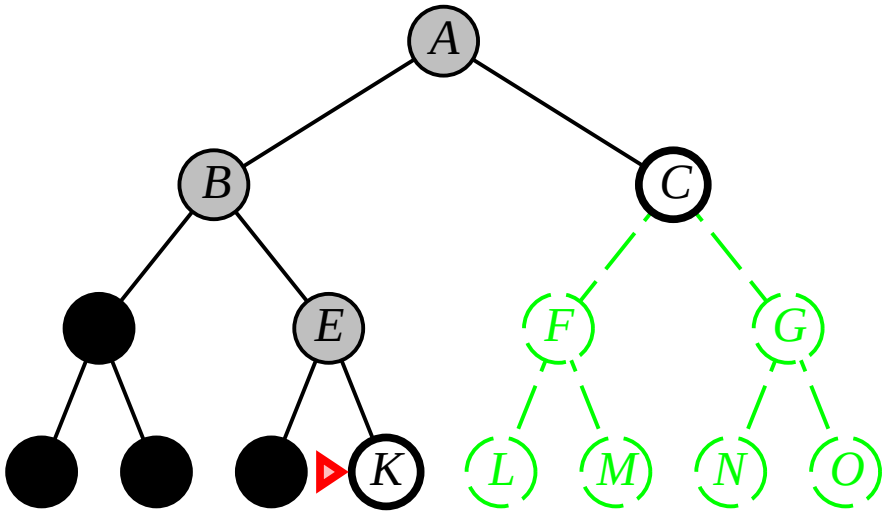
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

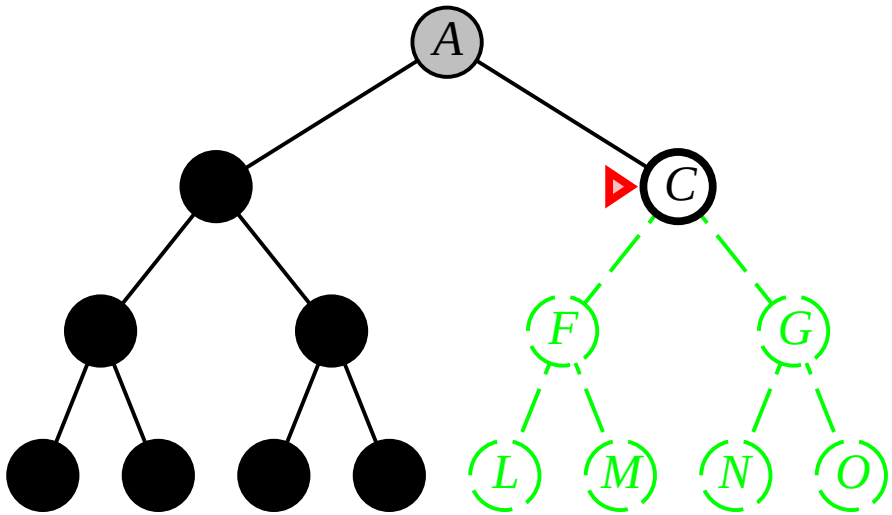
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

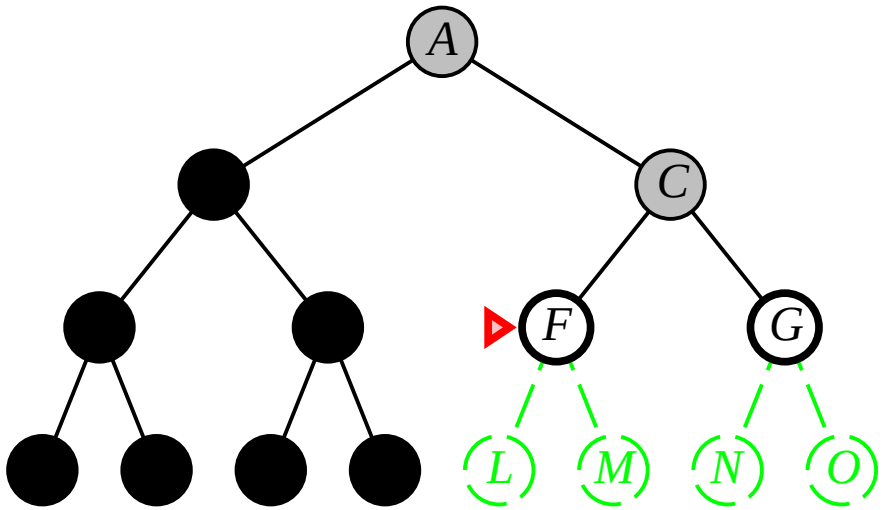
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

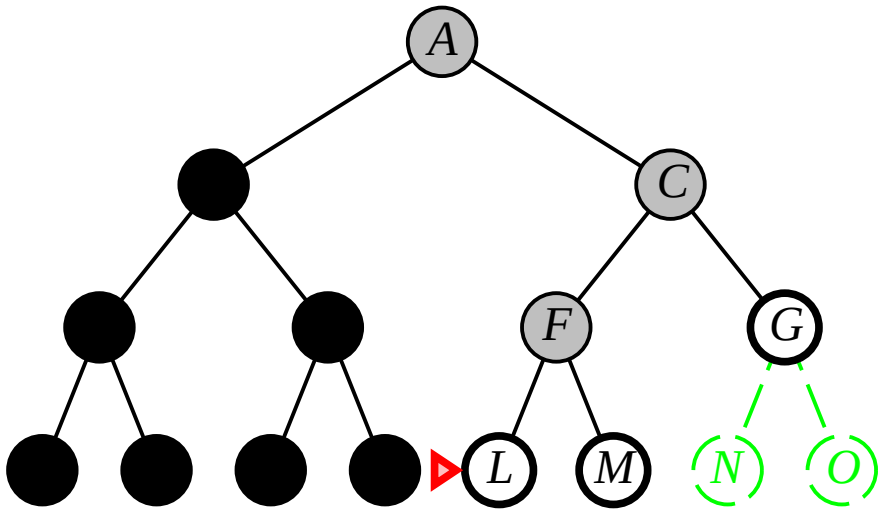
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



Tiefensuche (engl. *depth-first search*)

Expandiere tiefsten nicht-expandierten Knoten!

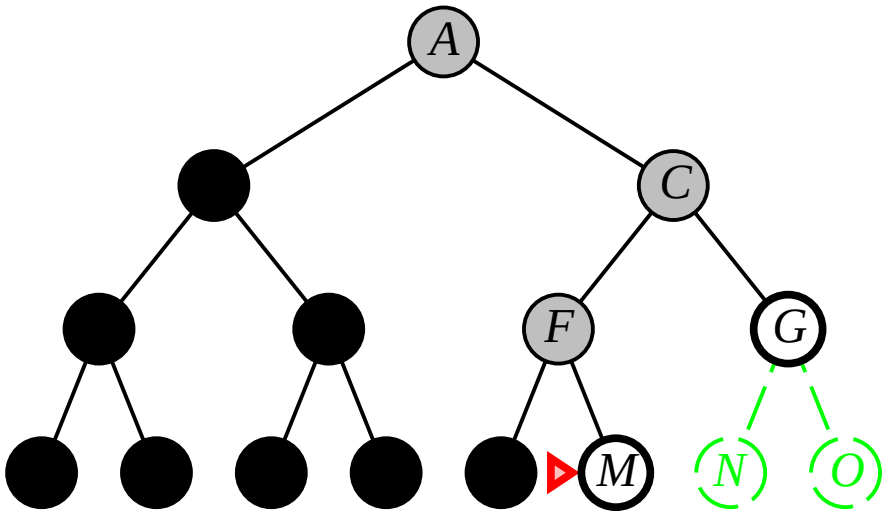
Implementierung: *fringe* = LIFO-Schlange (Nachfolger nach vorn)



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

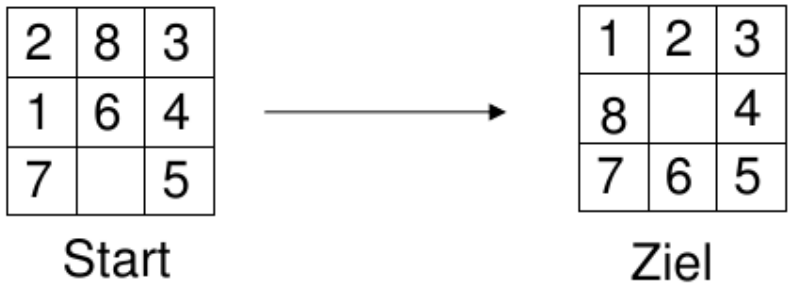
Tiefensuche mit Tiefenbegrenzung l

Demnach: Knoten in Tiefe l haben keine Nachfolger

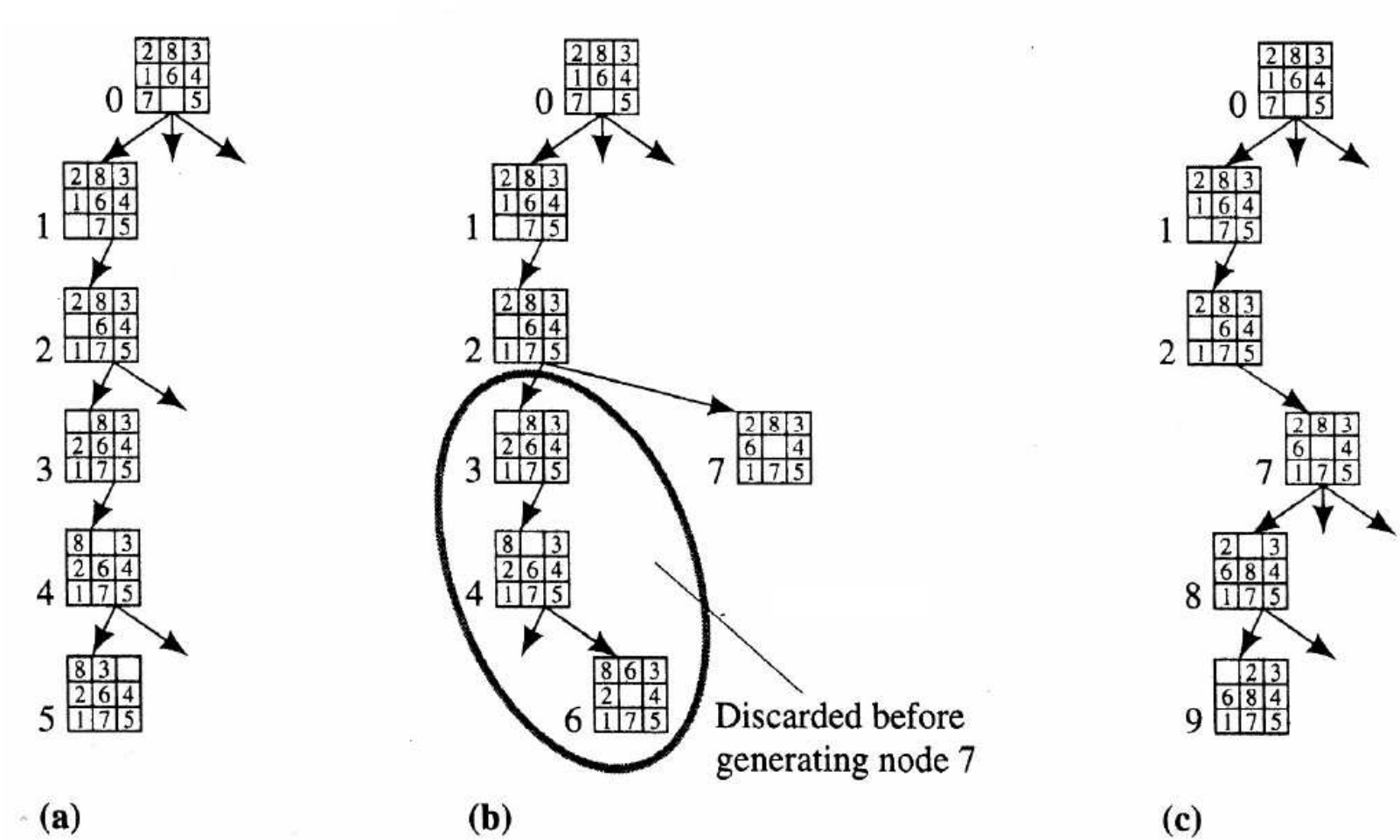
Beispiel: 8-Puzzle

Operationsreihenfolge: links, oben, rechts, unten

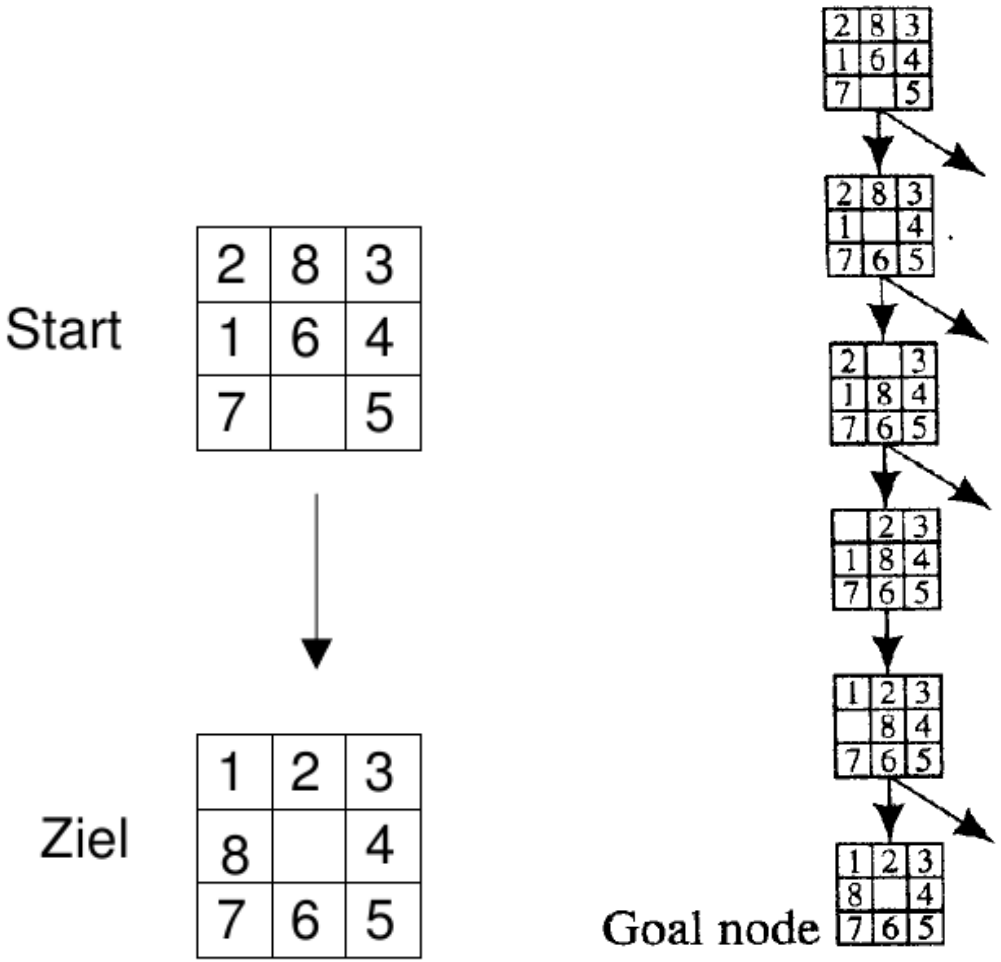
Tiefenbegrenzung: 5



Beschränkte Tiefensuche: 8-Puzzle



Beschränkte Tiefensuche: 8-Puzzle



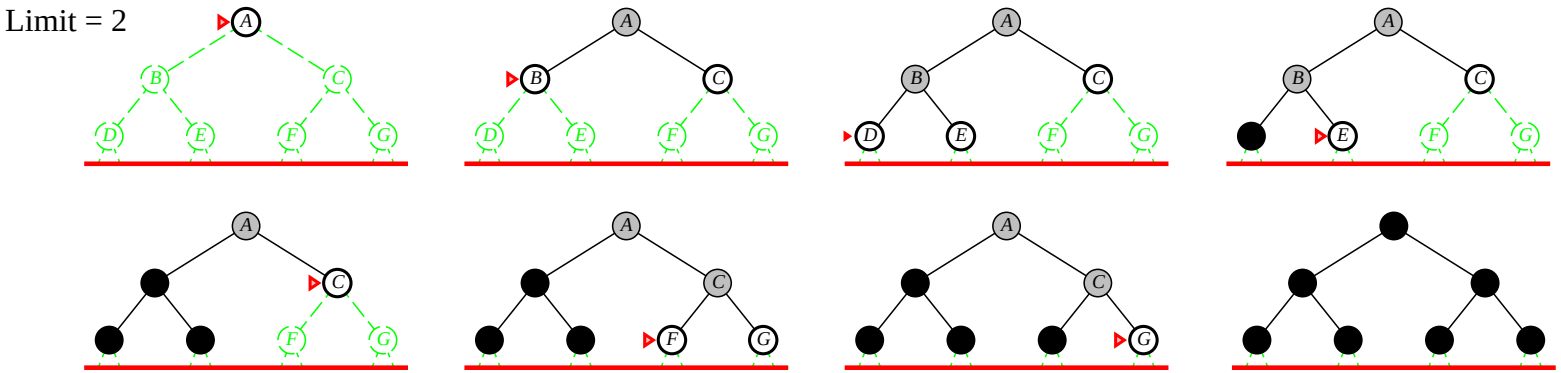
Iterative Tiefensuche

Limit = 0

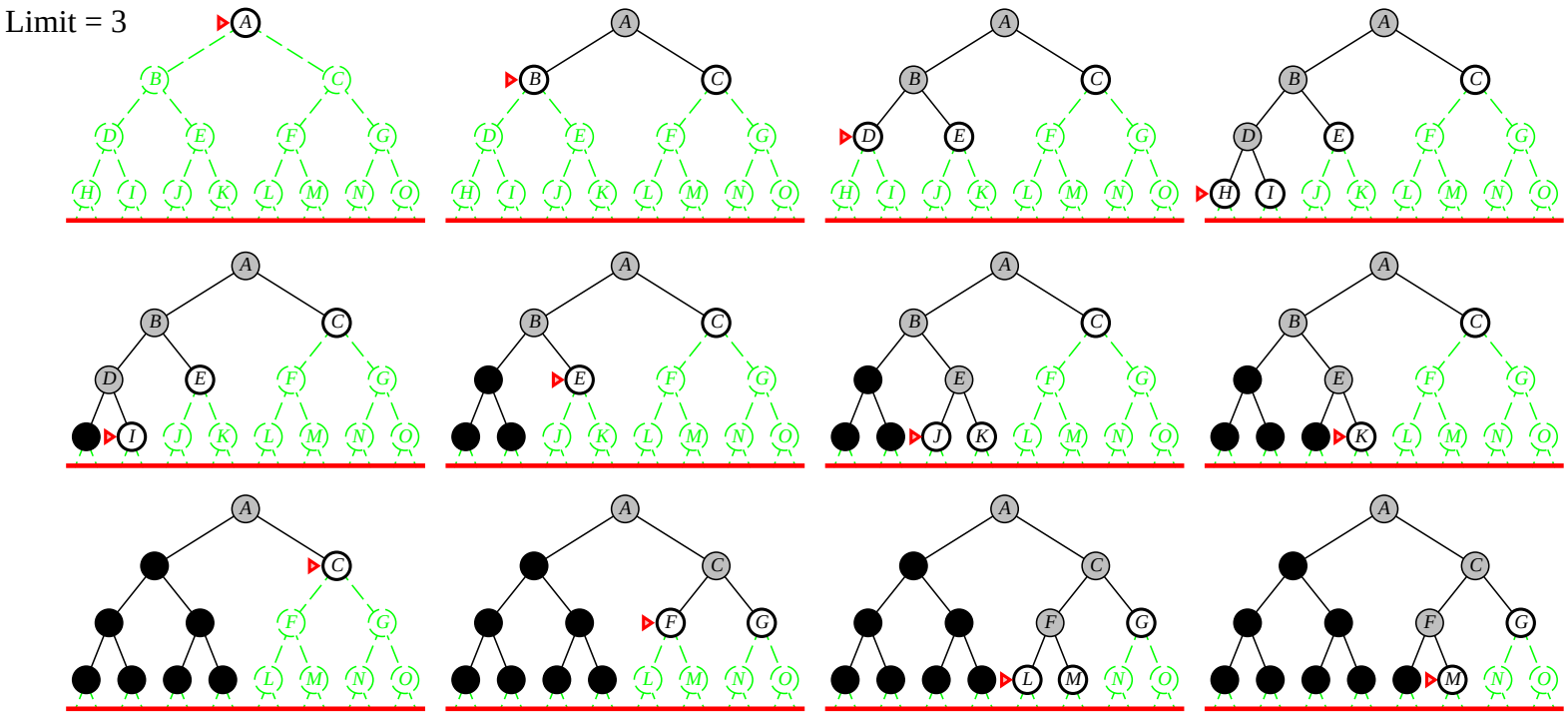
Iterative Tiefensuche

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

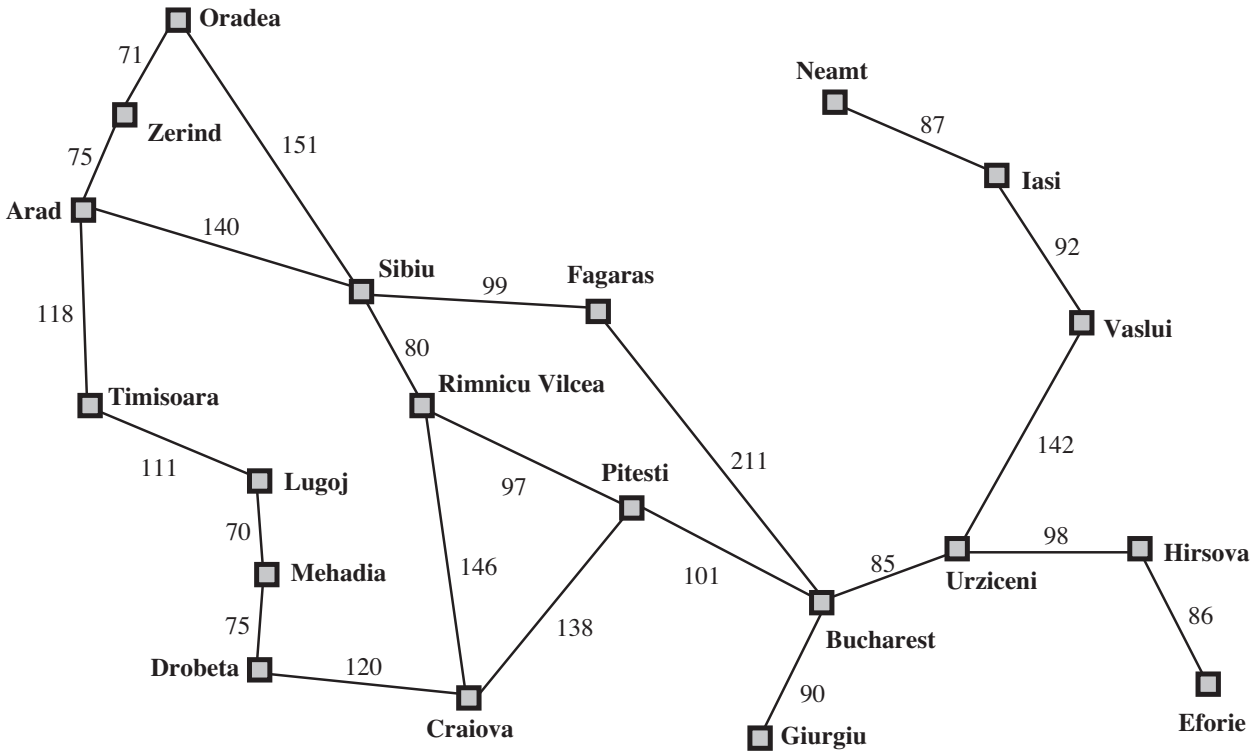
Schätzung, wie „wünschenswert/begehrt“ Knoten ist

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

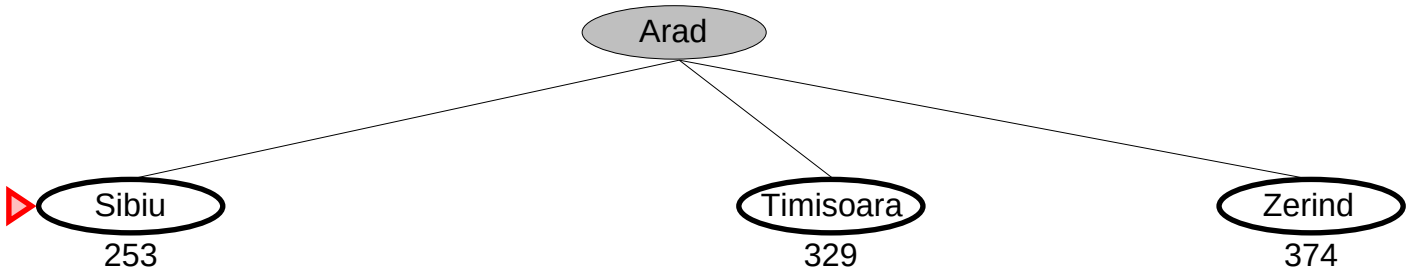
Bewertungsfunktion $h(n)$ (Heuristik):
Schätzung der Kosten von n zum nächsten Ziel

Z.B. $h_{\mathrm{LL}}(n) =$ Luftlinienabstand von n nach Bukarest

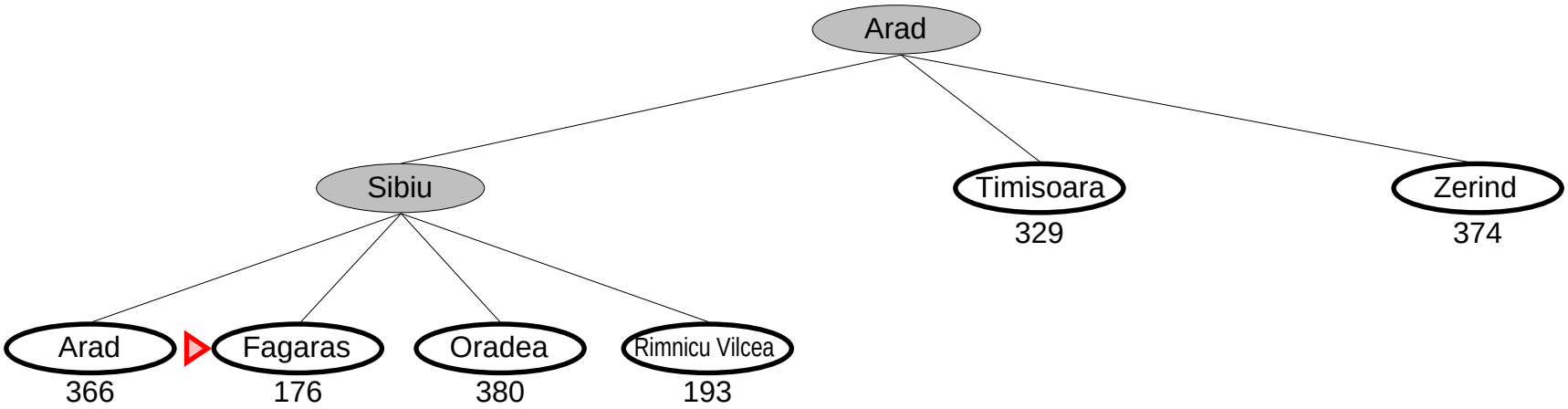
Greedy-Suche expandiert Knoten der am nächsten am Ziel *scheint*

Beispiel: Greedy-Suche

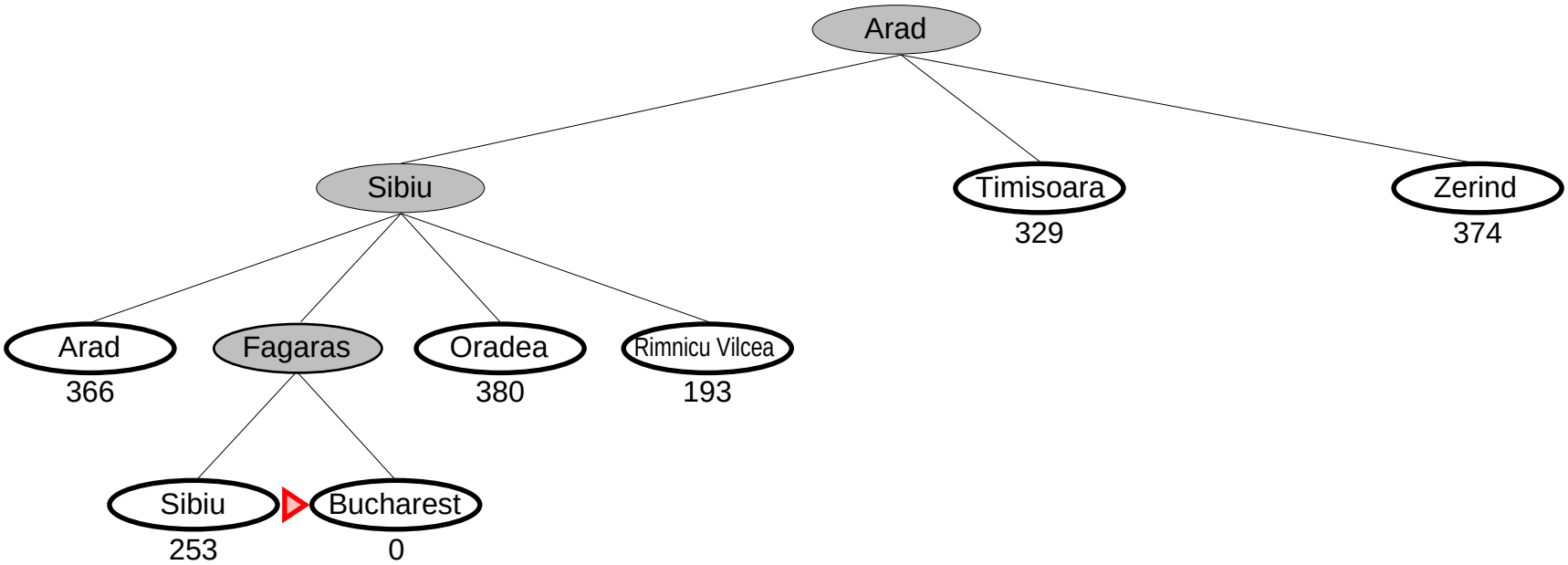
Beispiel: Greedy-Suche



Beispiel: Greedy-Suche



Beispiel: Greedy-Suche



Greedy-Suche: Eigenschaften

Vollständig:

- Nein, kann in Schleifen hängenbleiben, z.B.
  Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \ldots$
- 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 $f(n) = g(n) + h(n)$

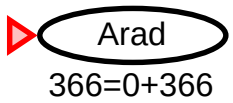
- $g(n)$ bereits aufgenommene Kosten um n zu erreichen
- $h(n)$ geschätzte Kosten von n zum Ziel
- $f(n)$ geschätzte Gesamtkosten des Pfades durch n zum Ziel

A*-Algorithmus benutzt *zulässige Heuristik*

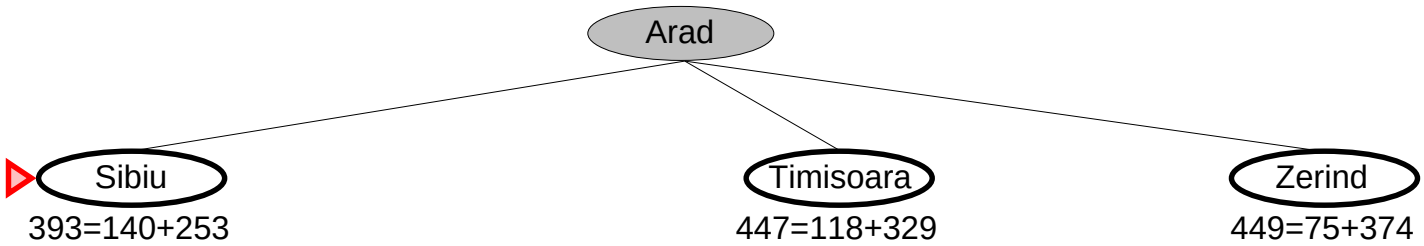
- Also, $h(n) \leq h^*(n)$ wobei $h^*(n)$ **wahren** Kosten von n
- Auch verlangt: $h(n) \geq 0$, also $h(G) = 0$ für beliebiges Ziel G
- Z.B. $h_{\mathrm{LL}}(n)$ überschätzt wirkliche Wegstrecke nie!

Satz: A*-Algorithmus ist optimal.

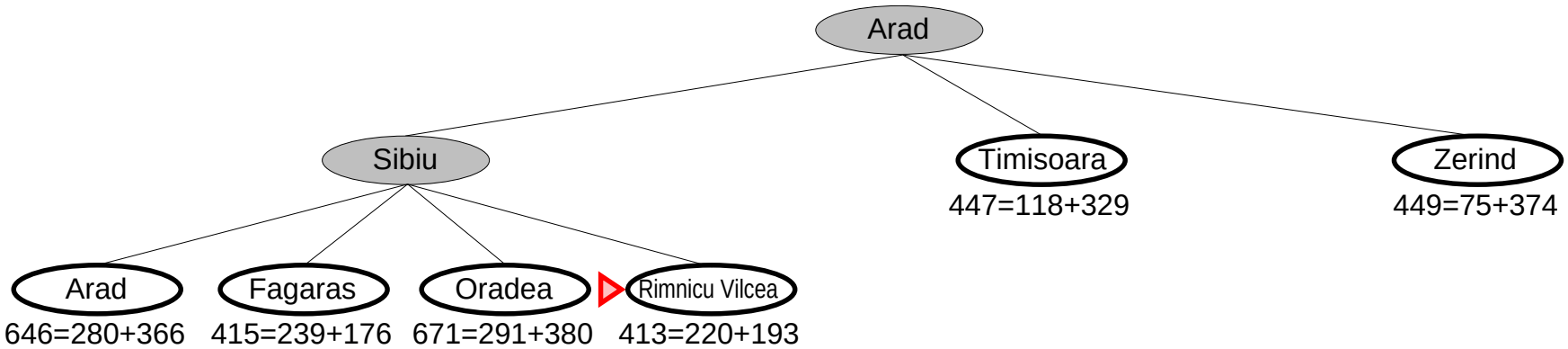
Beispiel: A*-Algorithmus



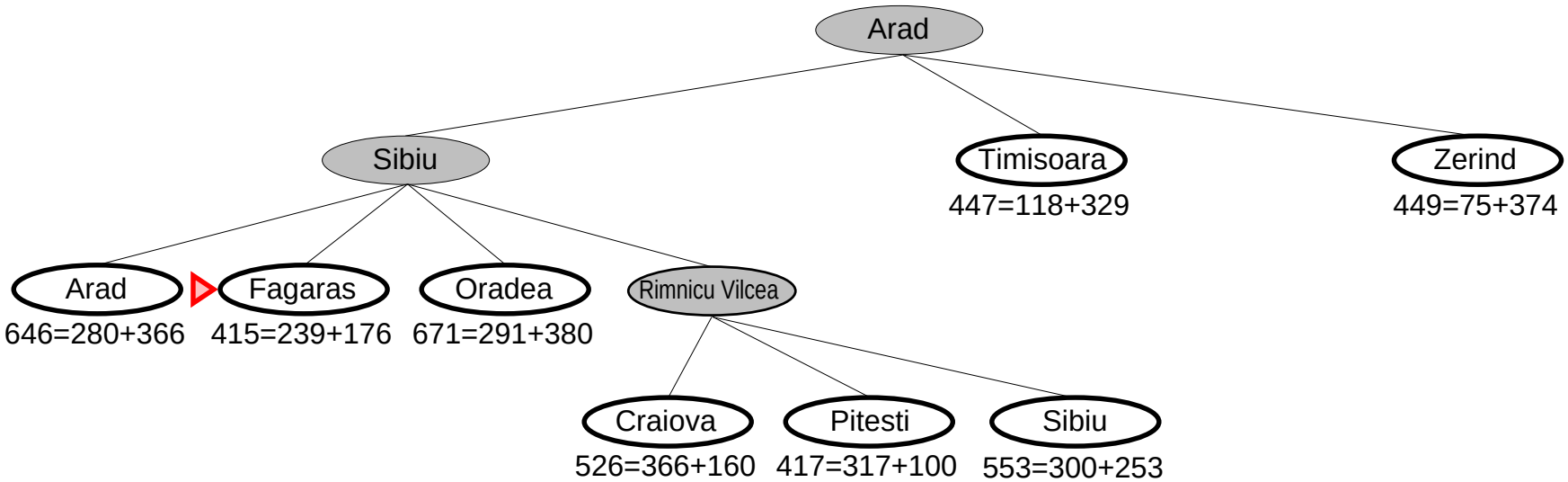
Beispiel: A*-Algorithmus



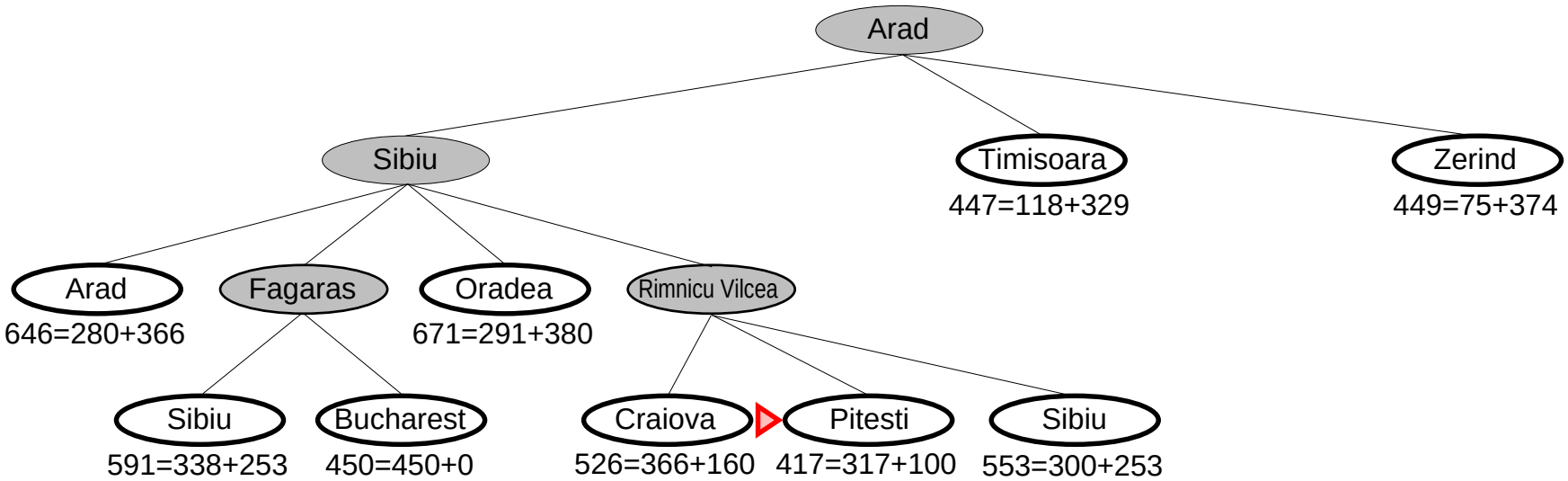
Beispiel: A*-Algorithmus



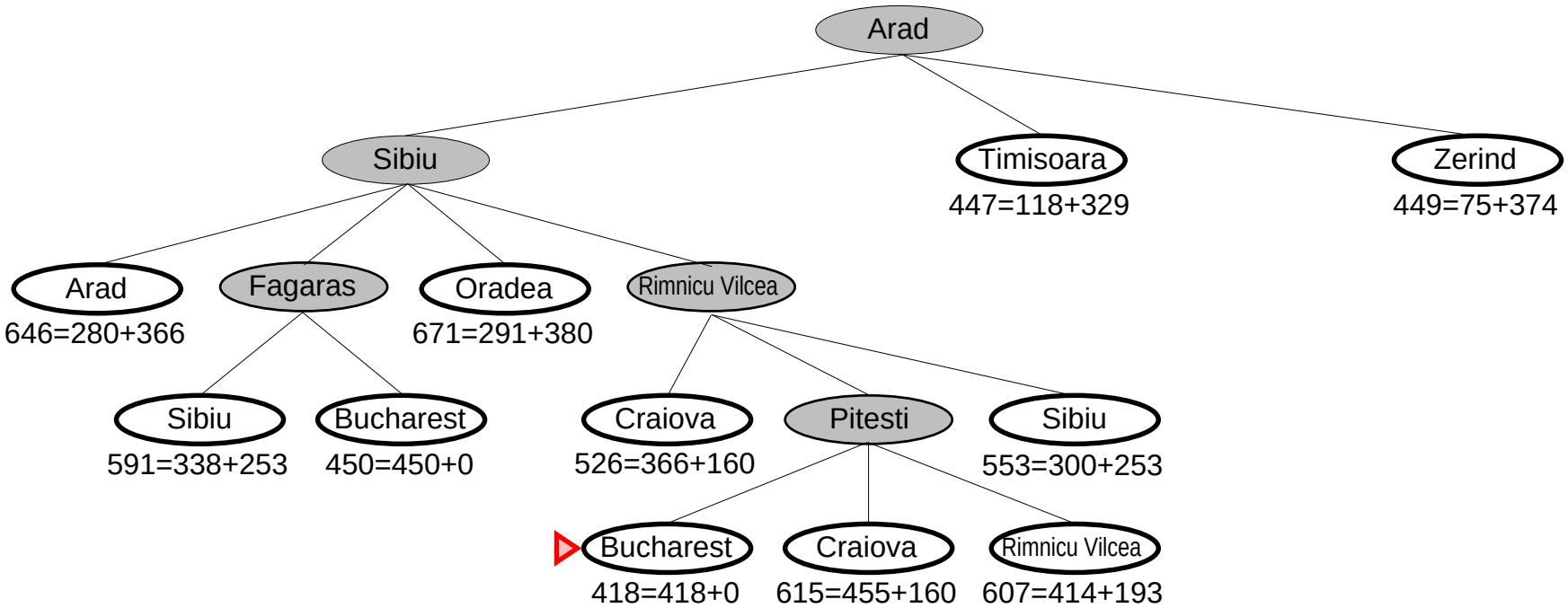
Beispiel: A*-Algorithmus



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 `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)

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 II

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 III

9. für alle $o \in O$:
   - $x := o(z)$ und $c := g(z) + \text{costs}(o)$
   - 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

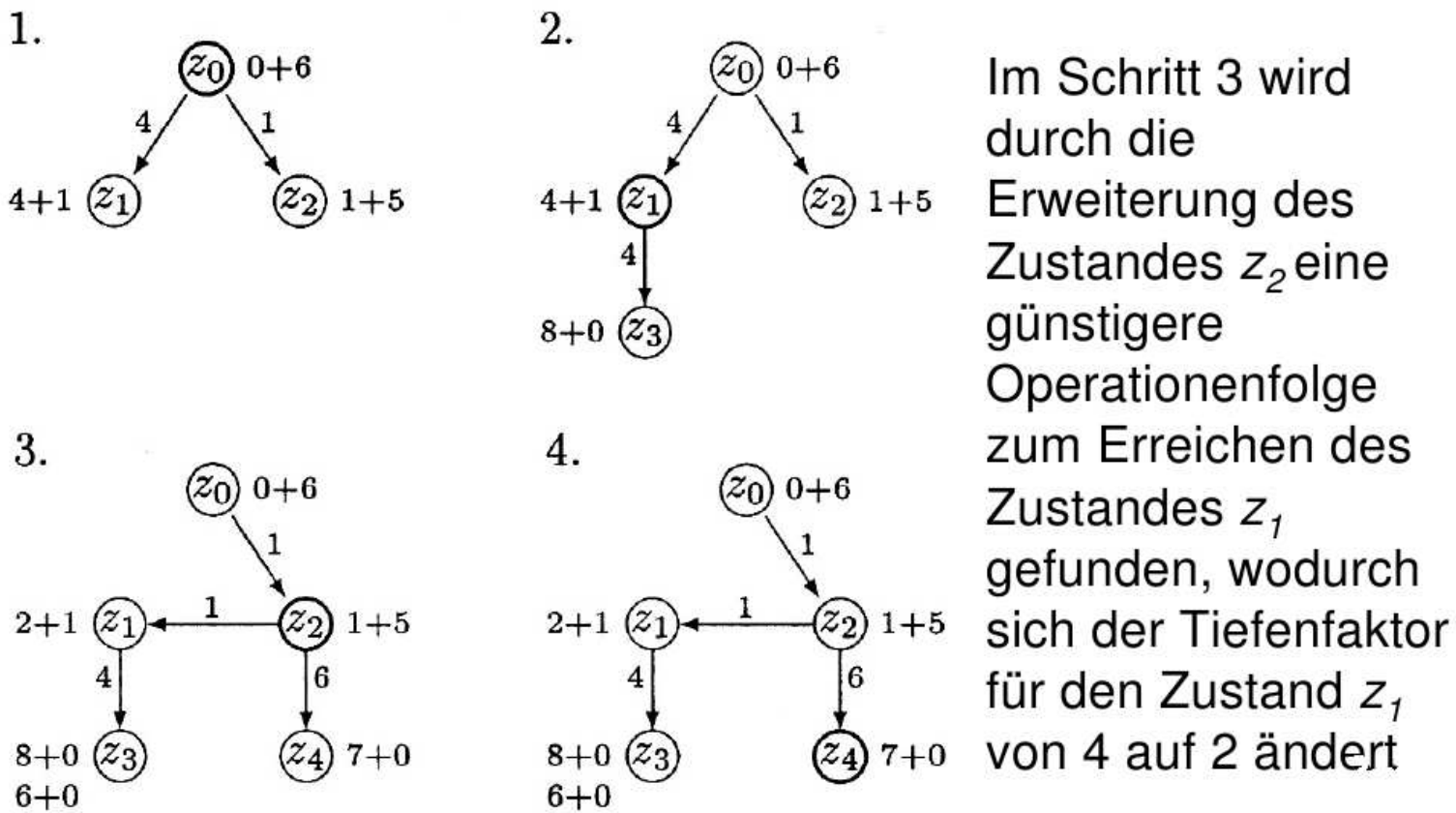
A*-Algorithmus: Ablauf IV

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

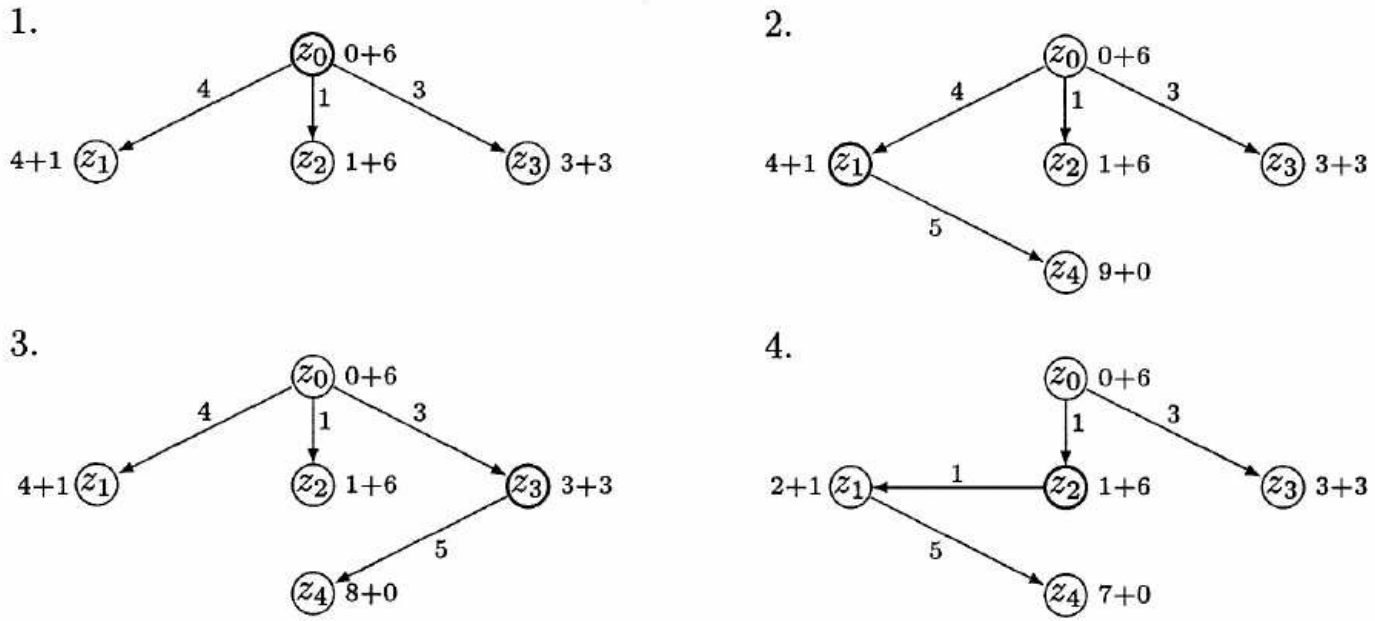
Notwendigkeit der Anpassung von Nachfolgezuständen (**9.**):



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

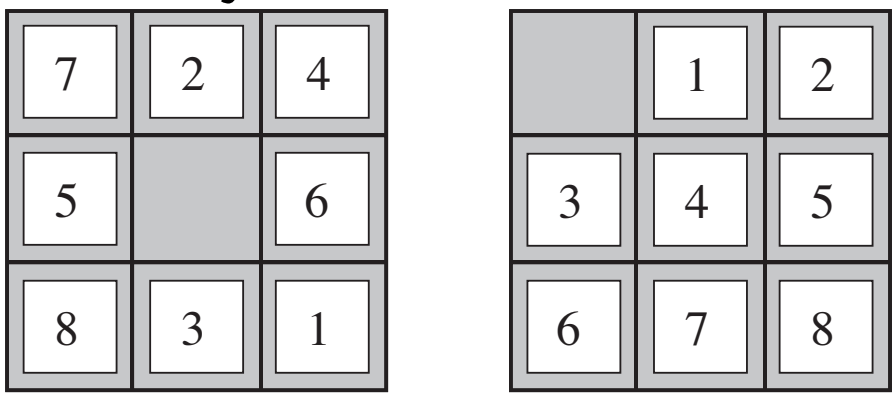
A* expandiert alle Knoten mit $f(n) < C^*$
A* expandiert einige Knoten mit $f(n) = C^*$
A* expandiert keine Knoten mit $f(n) > C^*$

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

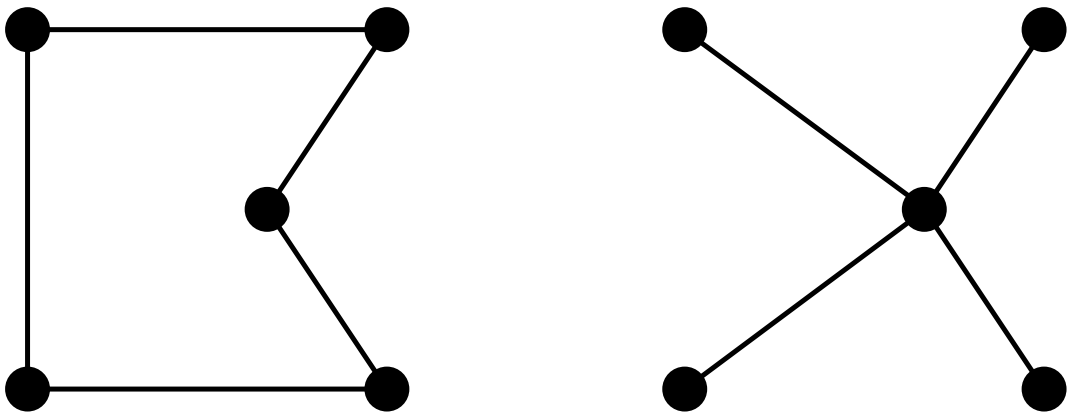
Wie erzeugt man zulässige Heuristiken?
Idee: Konstruktion **exakter** Lösungen einer relaxierten Version des Problems (Relaxierung: Weglassen oder Lockern von Bedingungen in Optimierungsproblemen), somit Ausnutzung der Kosten dieser Lösung als Heuristik

> 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

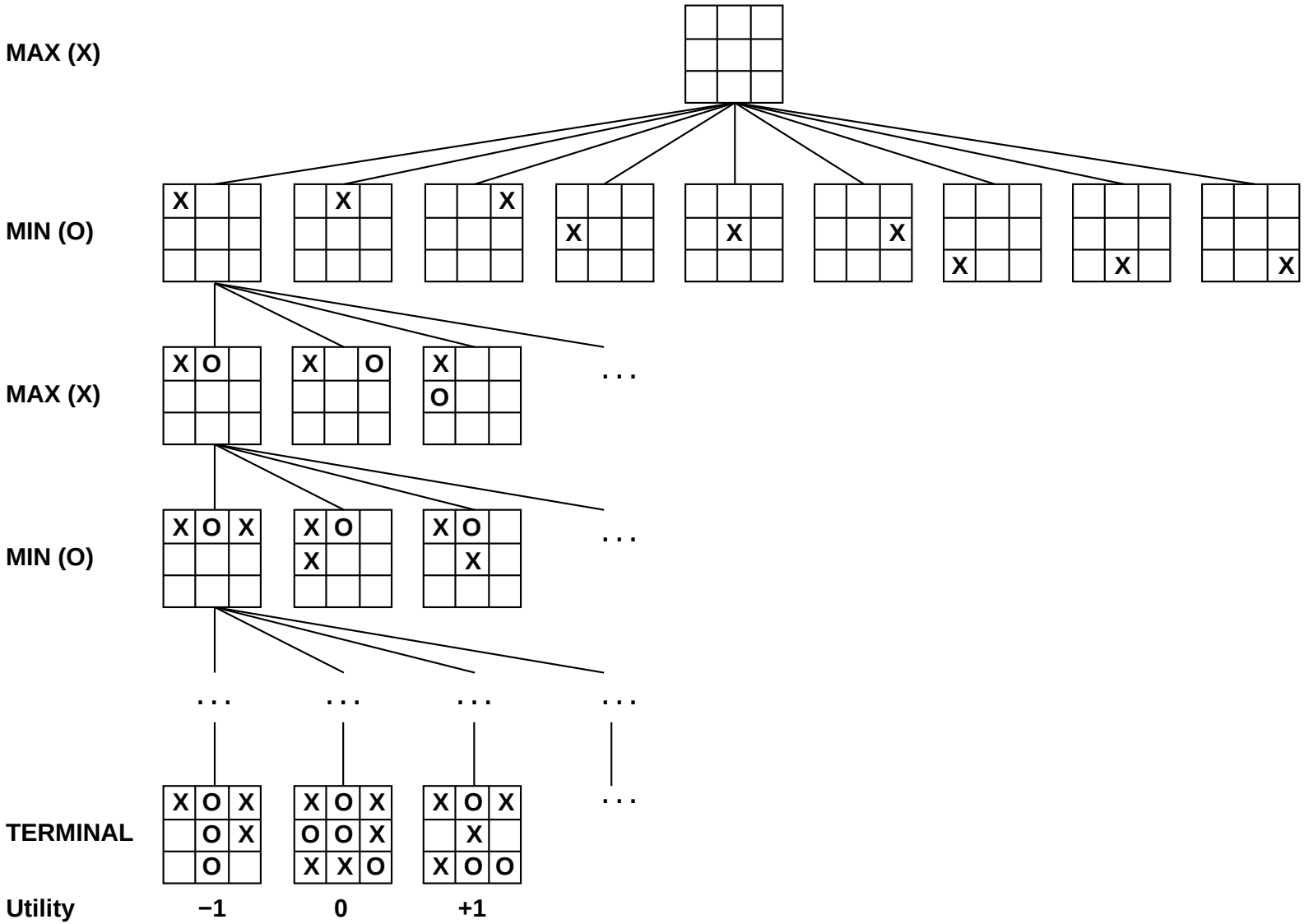
Arten von Spielen

	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

Spielbaum (2 Spieler, deterministisch, Runden)



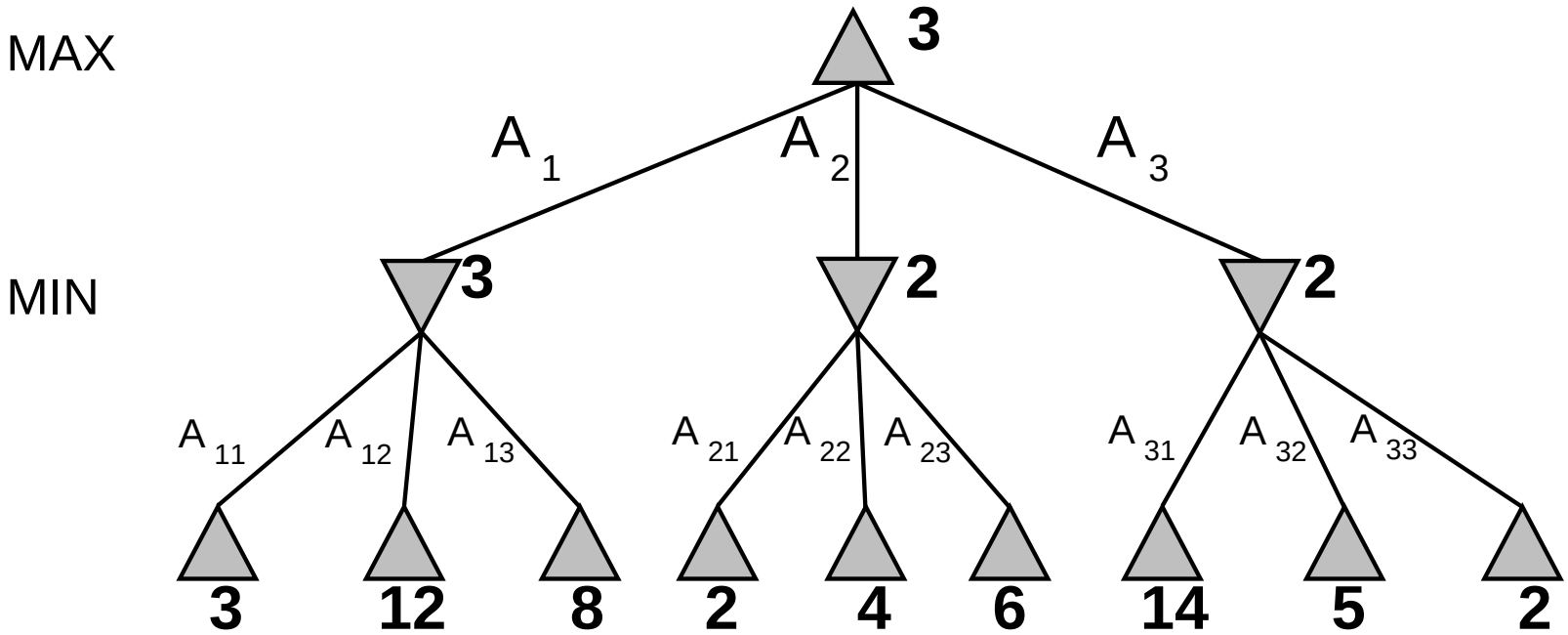
Minimax

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: Ein Beispiel

Zweipersonenspiel: Erster Spieler wählt eine von mehreren Taschen aus und erhält von seinem Gegenspieler den Gegenstand mit dem geringstem Wert aus dieser Tasche.

Minimax-Algorithmus durchsucht alle Taschen vollständig.

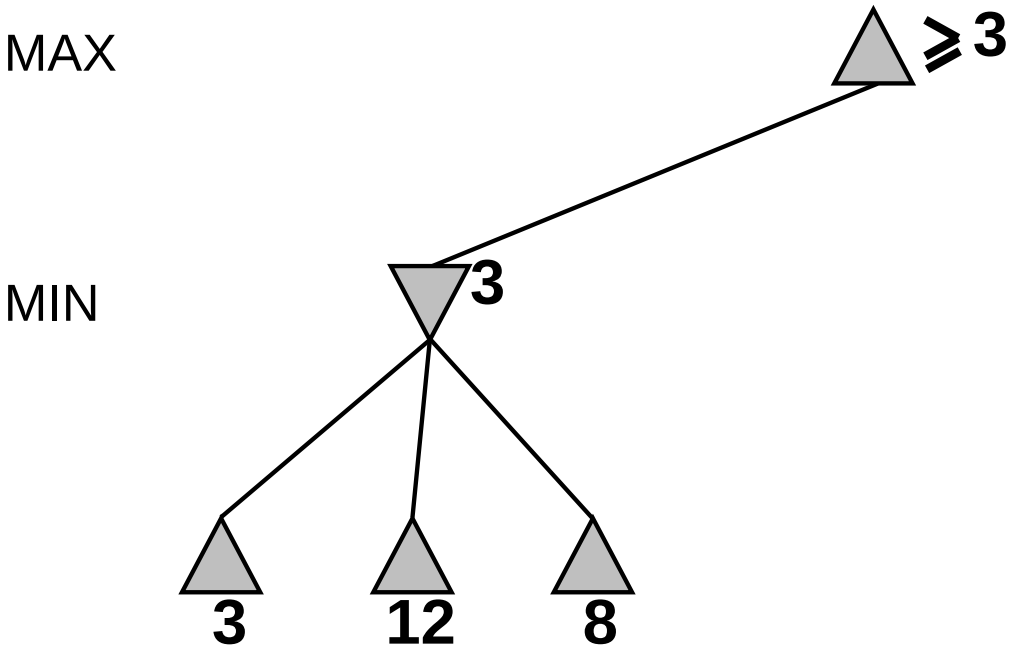
α–β-Stutzen durchsucht zunächst nur die erste Tasche vollständig nach dem Gegenstand mit minimalem Wert.

In allen weiteren Taschen wird nur solange gesucht, bis der Wert eines Gegenstands dieses Minimum unterschreitet.

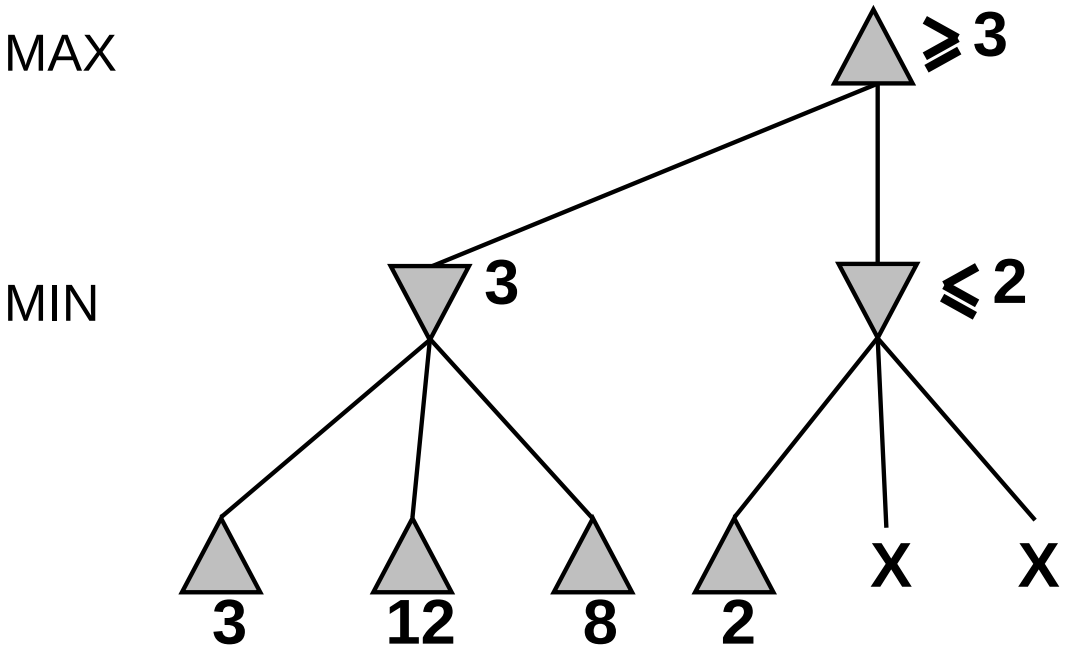
Ist dies der Fall, wird die Suche in dieser Tasche abgebrochen und die nächste Tasche untersucht.

Andernfalls ist diese Tasche eine bessere Wahl für den ersten Spieler und ihr minimaler Wert dient für die weitere Suche als

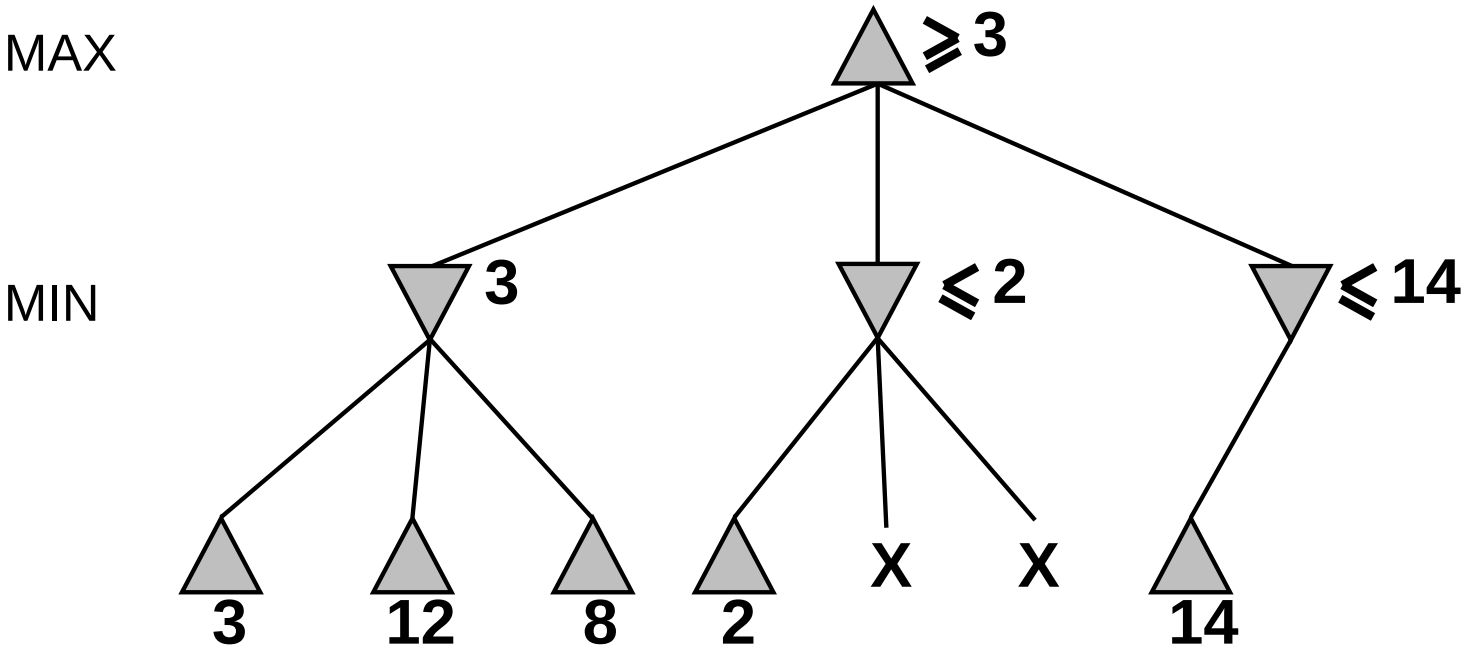
α–β-Stutzen



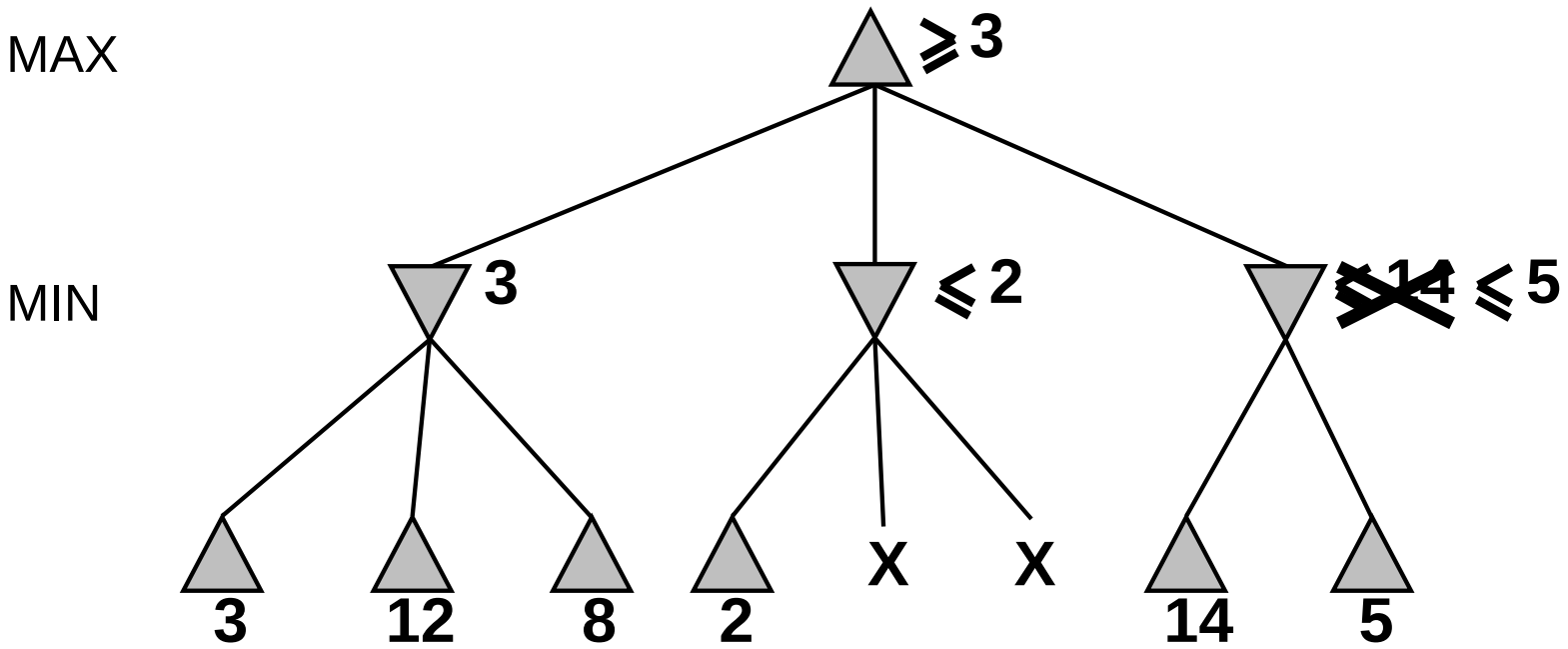
α–β-Stutzen



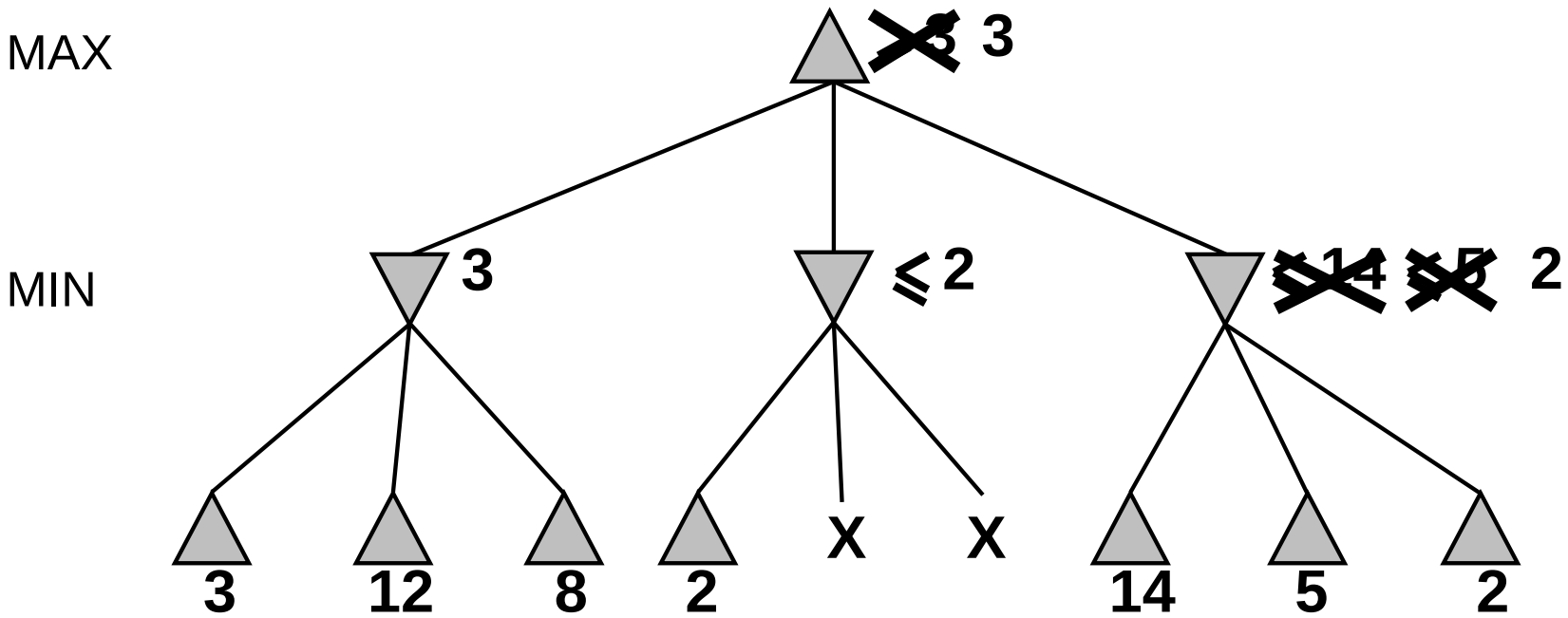
α–β-Stutzen



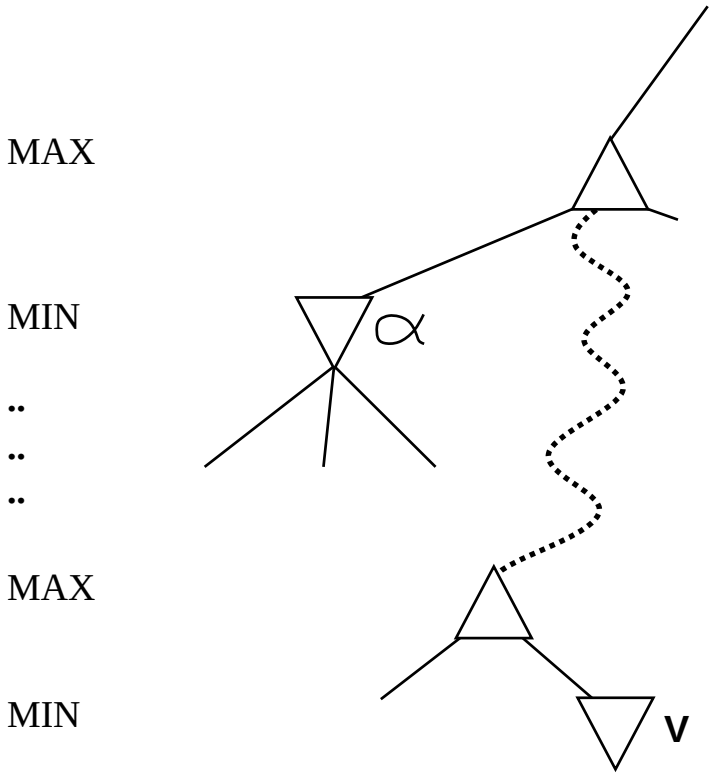
α–β-Stutzen



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

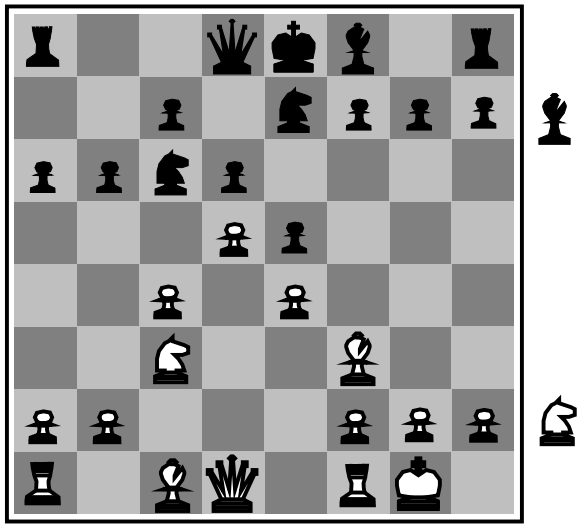
Standard-Ansatz:

- 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:

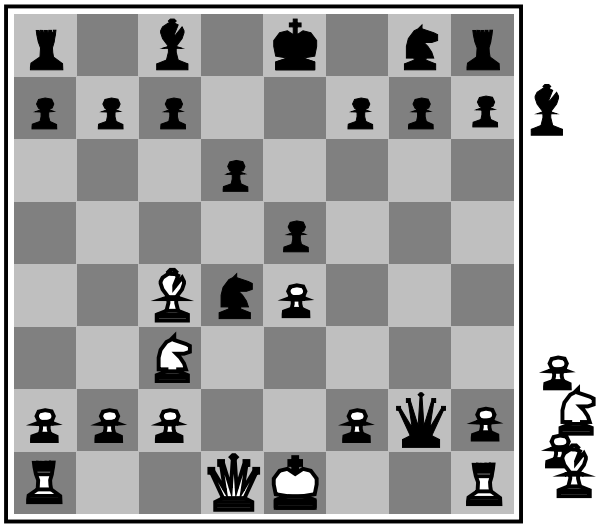
- 10^6 Knoten pro Zug $\approx 35^{8/2}$
- α–β berechnet 8 Halbzüge: ziemlich gutes Schachprogramm

Bewertungsfunktionen



Black to move

White slightly better



White to move

Black winning

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

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

Deterministische Spiele

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:

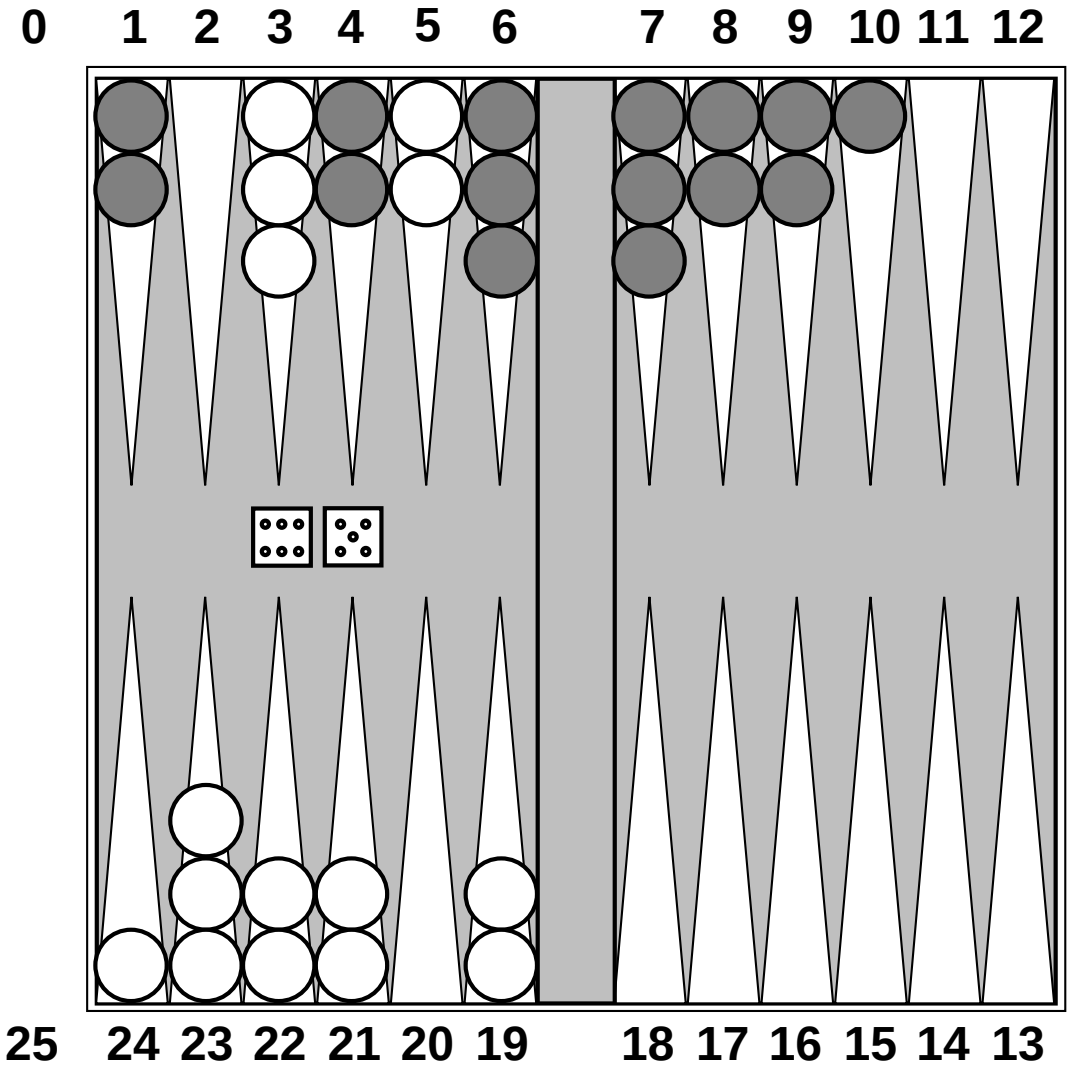
- Deep Blue besiegte 1997 Weltmeister Garri Kasparow in 6 Spielen
- $200 \cdot 10^6$ Stellungen/Sekunde, sehr komplizierte Bewertung
- Bis zu 40 Halbzüge tief (nicht veröffentlichte Methoden)

Othello: Wettbewerb nur unter menschlichen Meistern

Go:

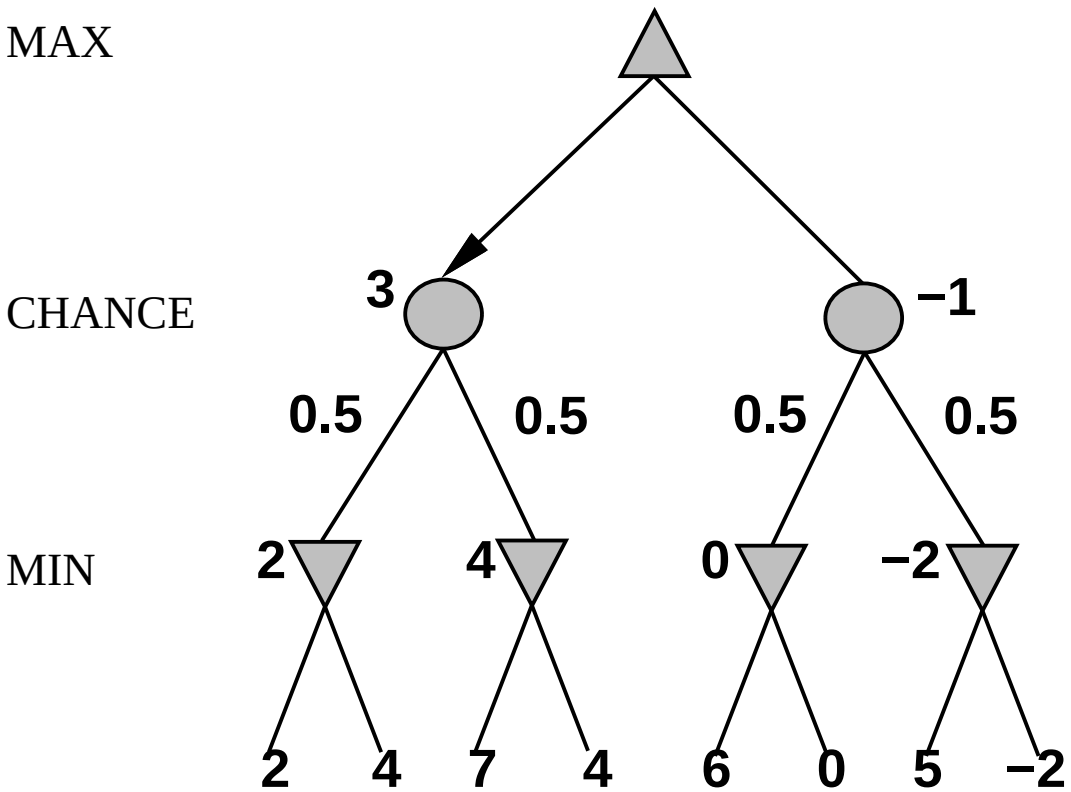
- Wettbewerb nur unter menschlichen Meistern
- $b > 300$, daher Datenbanken mit Mustern für plausible Züge

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

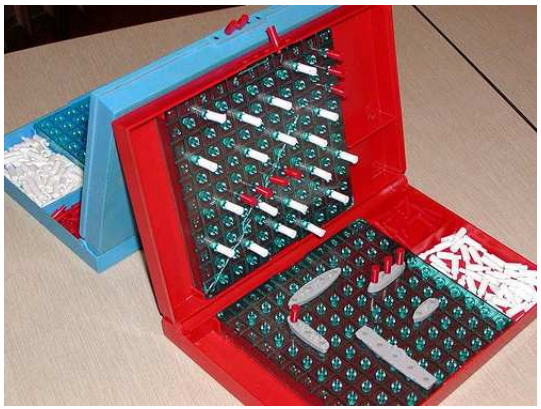
α–β-Stutzen viel weniger effektiv

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).



Nullsummenspiel: Spiel, bei dem $\sum$ der Gewinne und Verluste aller Spieler $= 0$

Übersicht

Was ist ein BEP?

BEP $= < \mathcal{X}, \mathcal{C}, \mathcal{D} >$

- $\mathcal{X}$ ist ein endliche Menge von Variablen $X_1, \ldots, X_n$.
- $\mathcal{C}$ ist eine endliche Menge von Bedingungen $C_1, \ldots, C_m$.
- $\mathcal{D}$ ist eine nicht-leere Menge $D_1, \ldots D_n$ von möglichen Werten für jede Variable X_i.

Ein *Zustand* ist definiert als *Zuweisung* von Werten zu einigen oder allen Variablen.

Eine *konsistente Zuweisung* ist eine Zuweisung, die keine der Bedingungen verletzt.

Was ist ein BEP?

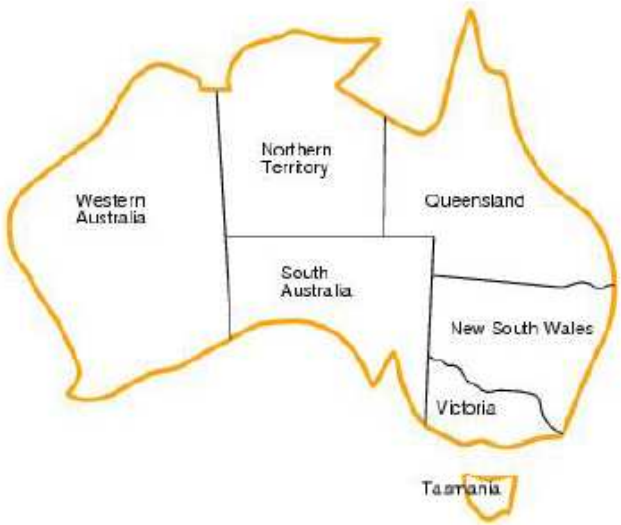
Eine Zuweisung ist *vollständig*, wenn jeder Variablen ein Wert zugewiesen wurde.

Eine Lösung für ein BEP ist eine vollständinge und konsistente Zuweisung.

In einigen Fällen muss durch die Lösung außerdem eine Zielfunktion maximiert werden.

BEP Beispiel: Graphfärbungsproblem

Variablen:
$X_1 = WA, X_2 = NT, X_3 = Q, X_4 = NSW, X_5 = V, X_6 = SA$ und $X_7 = T$



Wertebereiche:
$D_i = \{\text{rot, grün, blau}\}$
für $i = 1, \ldots, 7$.

Bedingungen:
benachbarte Regionen müssen unterschiedliche Farben haben,
z.B.: $WA \neq NT$ oder
$(WA, NT) \in \{(rot, grn), (rot, blau), (grn, rot), \ldots\}$

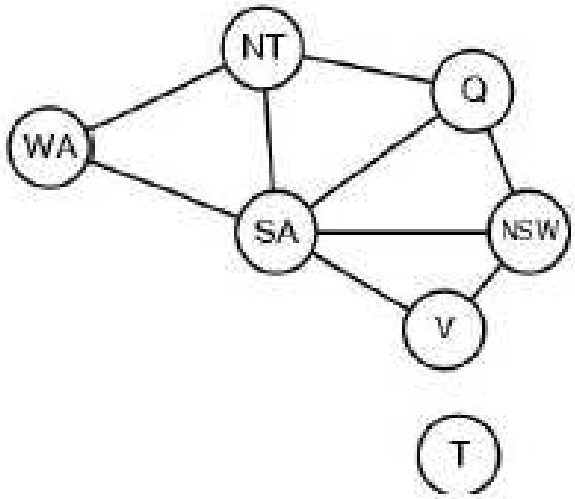
BEP Beispiel Graphfärbungsproblem

Eine Lösung für ein BEP ist eine vollständinge und konsistente Zuweisung, z.B.: $\{WA = red, NT = grn, Q = rot, NSW = grn, V = rot, SA = blau, T = grn\}$.

Bedingungsgraph

In einem binären BEP kommen in jeder Bedingung höchstens zwei Variablen vor.

Solche Probleme lassen sich durch einen Bedingungsgraphen modellieren: Knoten sind Variablen, Kanten zeigen Bedingungen.



Hier: Tasmanien ist ein unabhängiges Teilproblem.

Varianten von BEPs

Diskrete Variablen

- Endlicher Wertebereich der Größe d $\Rightarrow O(d^n)$ mögliche Zuweisungen
  z.B. Bool'sche BEPs, SAT ($\mathbb{NP}$-vollständig)
- Unendlicher Wertebereich (ganze Zahlen, Strings, etc.)
  z.B. Maschinenbelegungsprobleme, Variablen sind Start- bzw. Endzeiten für Aufträge), lineare Bedingungen lösbar, nicht-lineare unentscheidbar

Kontinuierliche Variablen

- Start- und Endzeiten von Beobachtungen des Hubbleteleskops
- Lineare Bedingungen, die in Polynomialzeit durch ganzzahlige Optimierung gelöst werden können

Varianten von Bedingungen

Einstellige Bedingunen beinhalten nur eine Variable
z.B.: $SA \neq grn$

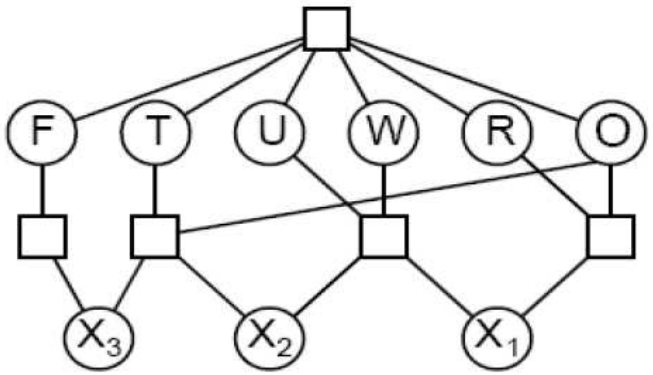
Zweistellige Bedingungen beinhalten Paare von Variablen
z.B.: $SA \neq WA$

Höherstellige Bedingungen beinhalten drei oder mehr Variablen
z.B.: kryptoarithmetische Spaltenbedingungen

Präferenzen (weiche Bedingungen), z.B. “rot ist besser als grün”, sind oft durch Kosten für Variablenzuweisungen darstellbar (Optimierung mit Nebenbedingungen)

Beispiel: Kryptoarithmetik

$$\begin{array}{r} \,T\;W\;O \\ +\;T\;W\;O \\ \hline F\;O\;U\;R \end{array}$$



hypergraph

Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Constraints
alldiff(F, T, U, W, R, O)
$O + O = R + 10 \cdot X_1$, etc.

Beispiel: Echte Probleme

Zuweisungsprobleme: Wer gibt welche Vorlesung?

Schedulingprobleme: Welche Vorlesung findet wann und wo statt?

Hardwarekonfiguration

Exceltabellen

Logistische Probleme

BEP als einfaches Suchproblem

Ein BEP kann als einfaches Suchproblem aufgefasst werden

Inkrementelle Formulierung:

- Startzustand: Die leere Zuweisung $\{\}$
- Nachfolgerfunktion: Weise einer bisher nicht belegten Variable einen Wert zu, sodass kein Konflikt entsteht
- Zieltest: Die aktuelle Zuweisung ist vollständig
- Pfadkosten: konstante Kosten für jeden Schritt

BEP als einfaches Suchproblem

Dieses Vorgehen gilt für alle BEPs!

Da die Lösung bei Tiefe n (wenn es n Variablen gibt) liegt, kann Tiefensuche genutzt werden

Pfad ist irrelevant, also kann auch komplette Zustandsrepräsentation genutzt werden

Verzweigungsgrad b des Suchbaums in der Wurzel ist $n \cdot d$

$b = (n - l) \cdot d$ in Tiefe l, also gibt es $n! d^n$ Blätter (aber nur d^n vollständige Zuweisungen)

Grund: BEPs sind kommutativ (die Reihenfolge der Zuweisungen hat keine Auswirkungen auf die Lösung

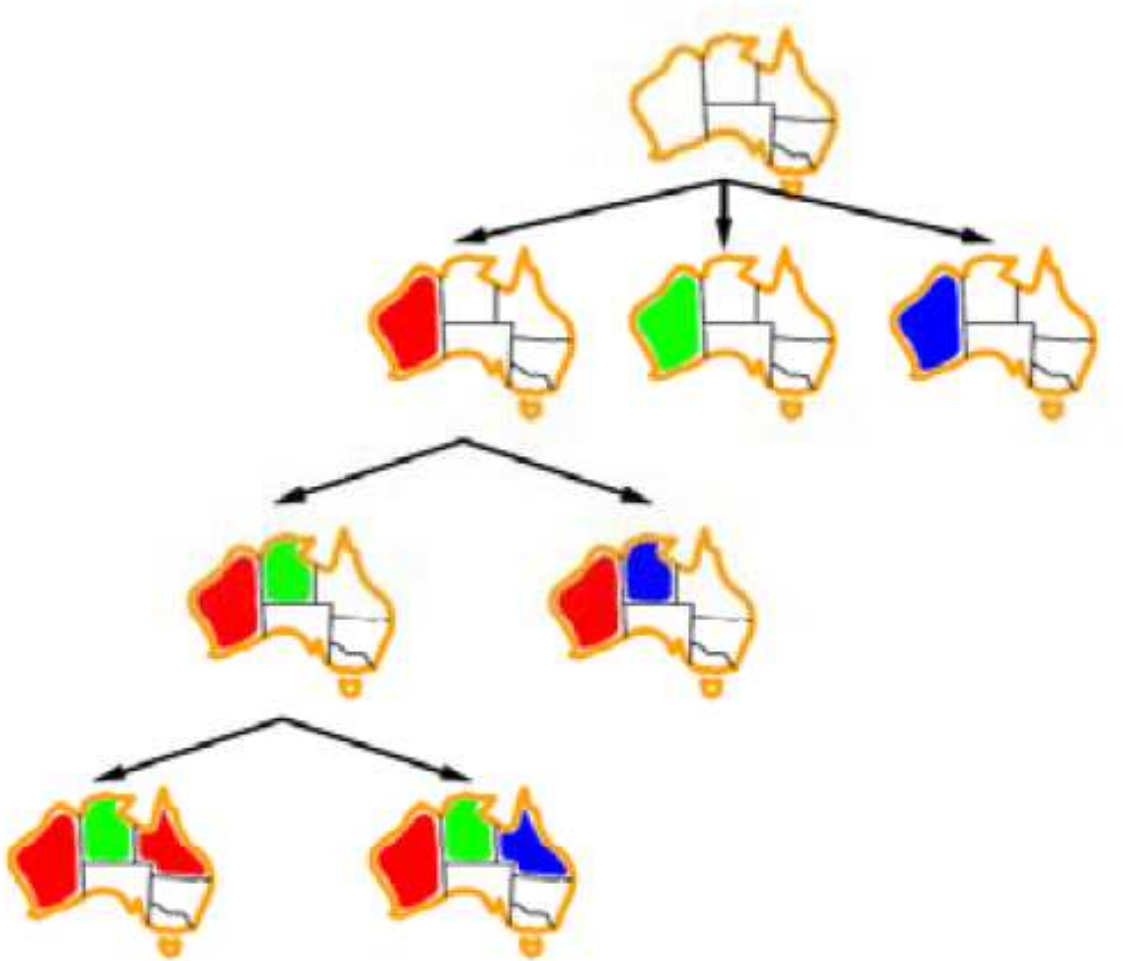
Backtracking

Tiefensuche

Wählt Werte für jeweils eine Variable pro Schritt und geht zurück wenn keine gültigen Schritte mehr möglich sind

Uninformiertes Suchverfahren (im generellen keine gute Performance)

Beispiel: Backtracking



Backtracking Effizienzverbesserung

Drei Hauptfragen:

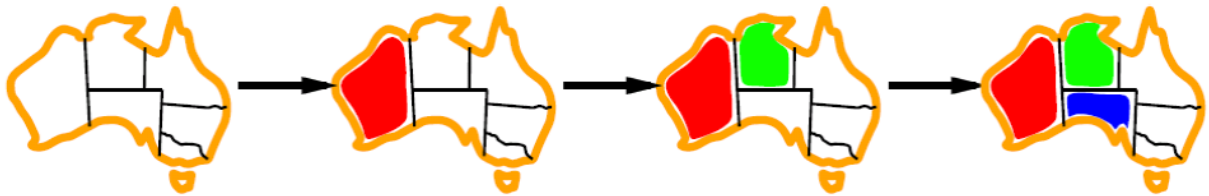
Welche Variable sollte als nächstes belegt werden und in welcher Reihenfolge soll man die Werte durchprobieren?

Welche Implikationen ergeben sich aus einer Belegung für die übrigen, noch nicht belegten Variablen?

Wenn ein Pfad *scheitert*, kann die Suche es vermeiden, diesen Fehler in zukünftigen Pfaden zu wiederholen?

Minimum Remaining Values

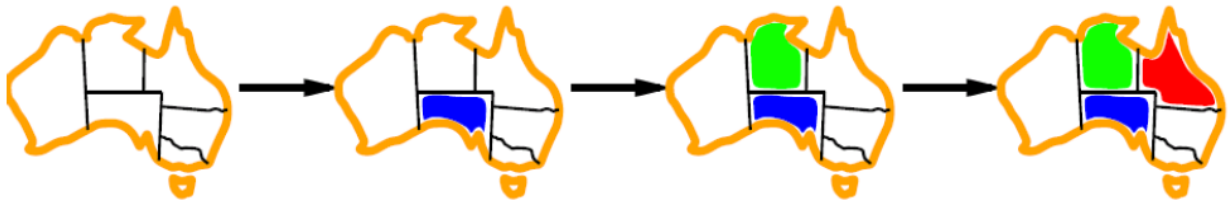
Die *minimum remaining values* Heuristik (am stärksten durch Bedingungen eingeschränkte Variable) belegt die Variable als nächste, bei der die wenigsten Werte noch übrig sind.



Die Heuristik beantwortet die Frage, welche Variable als nächstes belegt werden soll.

Most Constraining Variable

Die *most constraining vaariable* Heuristik belegt die Variable als nächste, die in den meisten Bedingungen für andere Variable auftaucht.



Die Heuristik beantwortet die Frage, welche Variable als nächstes belegt werden soll.

Least Constraining Value

Die *least constraining value* Heuristik belegt eine gewählte Variable mit dem Wert, der die wenigstens Bedingungen für die übrigen Variablen.

Die Heuristik beantwortet die Frage, welche Wert für eine Variable gewählt werden soll.

Beispiel: Sudoku

Sudoku ist ein BEP

Variablen: $X_{11}, X_{12}, \ldots, X_{99}$

Wertebereiche: $\{1, 2, \ldots, 9\}$

Bedingungen: alle Zeilen, alle Spalten und alle 3×3-Blöcke müssen jede Zahl genau einmal enthalten

Die MRV-Heuristik füllt die Zelle mit den wenigsten möglichen Werten

Die MCV-Heuristik füllt die Zelle, die die meisten anderen, freien Zellen beeinflusst

Die LCV-Heuristik wählt den Wert (für eine gegebene Zelle), der die meisten Möglichkeiten für die übrigen Zellen lässt.

Vorwärtsprüfung

Können wir einen unvermeidlichen Fehler frühzeitig entdecken? Und ihn dann vermeiden?

Vorwärtsprüfung: Führe eine Liste von verbleibenden, möglichen Werten für freie Variablen

Beende die Suche, wenn irgendeine Variable keine gültigen Werte mehr hat.

Funktioniert sehr gut zusammen mit der MRV-Heuristik

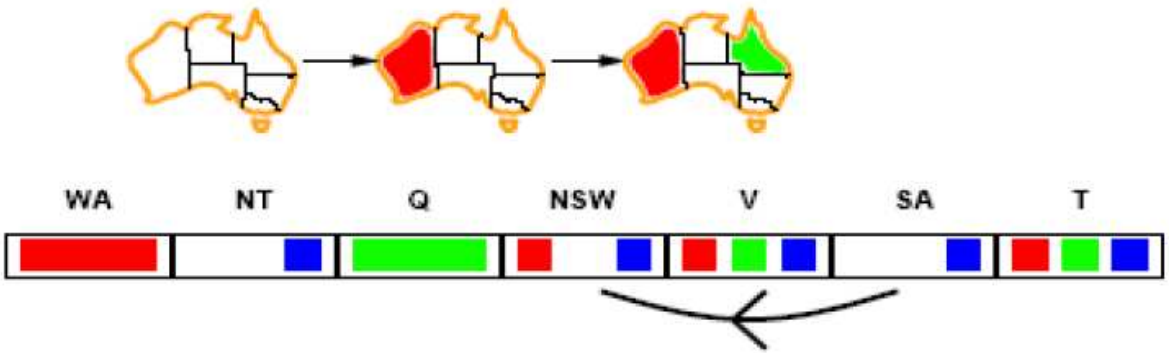
Bedingungsverbreitung

Vorwärtsprüfung findet nicht alle Fehler.

Wenn für zwei Variablen die selben, sich ausschließenden Belegungen verbleiben, wird das von der Vorwärtsprüfung nicht erkannt.

Durch wiederholtes Verbreiten der Bedingungen, können diese lokal besser durchgesetzt werden.

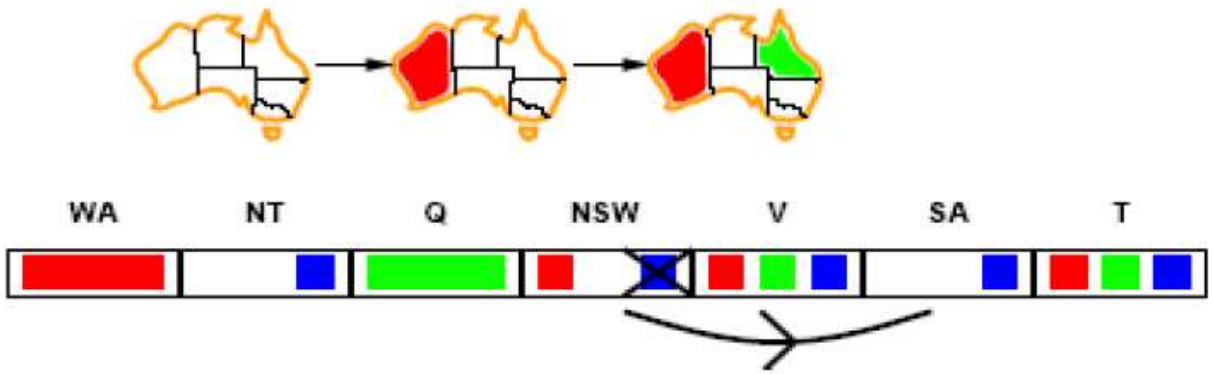
Kantenkonsistenz



Eine Kante $X \rightarrow Y$ ist genau dann konsistent, wenn für jeden Wert x von X mindestens ein Wert y für Y erlaubt ist.

$SA \rightarrow NSW$ ist konsistent $\leftrightarrow SA = \text{blau} \wedge NSW = \text{rot}$

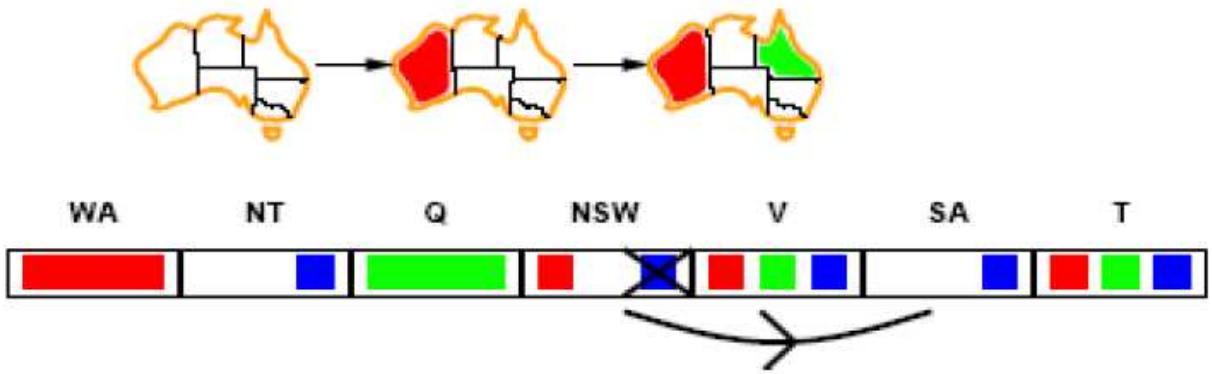
Kantenkonsistenz



Eine Kante $X \rightarrow Y$ ist genau dann konsistent, wenn für jeden Wert x von X mindestens ein Wert y für Y erlaubt ist.

Umgekehrt: $NSW \rightarrow SA$ ist konsistent
$\leftrightarrow NSW = \text{red} \wedge SA = \text{blau}$ und $NSW = \text{blau} \wedge SA = ???$.

Kantenkonsistenz



Kante kann konsistent gemacht werden, indem *blau* von den möglichen Werten für *NSW* entfernt wird.

Dadurch können weitere Inkonsistenzen entstehen und der Prozess muss für die übrigen Kanten wiederholt werden.

Diese Heuristik erkennt Fehler früher und kann daher nachg jeder Zuweisung als eine Art *Präprozessor* verwendet werden.

k-Konsistenz

Kantenkonsistenz erkennt nicht alle Inkonsistenzen

Stärkere Formen der Verbreitung können durch k-Konsistenzen definiert werden

Ein BEP ist k-konsistent, wenn für jede Menge von $k-1$ Variablen und jede konsistente Zuweisung zu diesen Variablen, einer beliebigen k-ten Variable ein gültiger Wert zugewiesen werden kann.

Formen: 1-Konsistenz (Knotenkonsistenz), 2-Konsistenz (Kantenkonsistenz), 3-Konsistenz (Pfadkonsistenz)

k-Konsistenz

Ein Graph ist *stark k-konsistent*, wenn er für jedes $j \leq k$ j-konsistent ist.

Das ist ideal, denn eine Lösung kann dann in $\mathcal{O}(nd)$ gefunden werden.

Aber: No-Free-Lunch! Jeder Algorithmus der n-Konsistenz herstellt, braucht mindestens $\mathcal{O}(2^n)$ im schlimmsten Fall