

# Künstliche Intelligenz

Prof. Dr. Dirk Krechel  
Hochschule RheinMain



Hochschule **RheinMain**  
University of Applied Sciences  
Wiesbaden Rüsselsheim Geisenheim

- Einführung
- Symbolische Verfahren, Logik
  - Aussagenlogik, Prädikatenlogik
  - Horn Logik, Prolog
- **Suchen und Bewerten**
  - **Problemlösen durch Suche**
    - **Uninformierte Suche**
    - Heuristische Suche
    - Spielbäume
- Lernen
  - Entscheidungstheorie
  - Naive Bayes
  - Entscheidungsbäume
  - Neuronale Netze
  - unüberwachtes Lernen

- Modellierung

- Viele Probleme darstellbar als

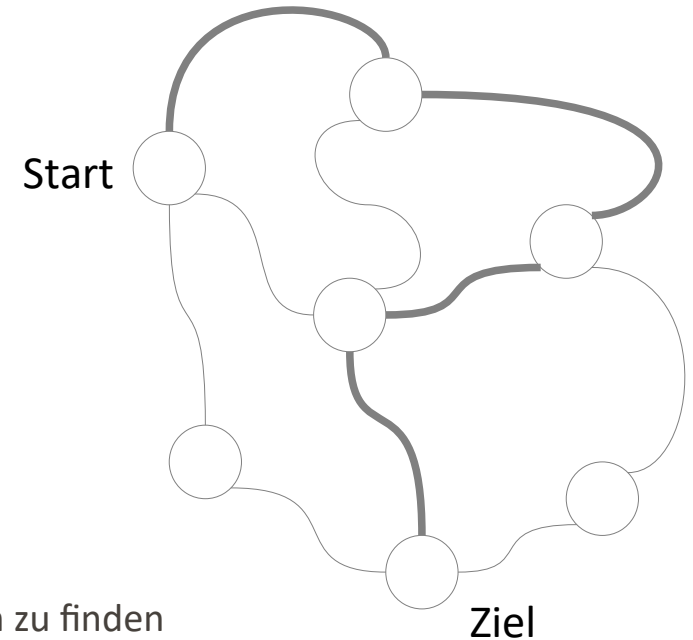
- Zustandsraum (Umgebung), (un)endliche Menge diskreter Zustände
    - Ausgewiesener Startzustand und Zielzustand

- Handlungskette zur Zielerreichung

- Sequentielle Ausführung (Reihenfolge) von elementaren Schritten
    - Jeder Schritt/Handlung, (un)endliche Menge von Operatoren, überführt Zustand in anderen Zustand
    - Beginn bei Startzustand, letzter Schritt überführt in Zielzustand
  - Problemlösen heißt eine richtige Reihenfolge von Handlungen zu finden die Startzustand in Zielzustand überführt

- Problemlösungsstrategien

- Uninformierte Suche
  - Heuristische Suche
  - Stochastische Suche

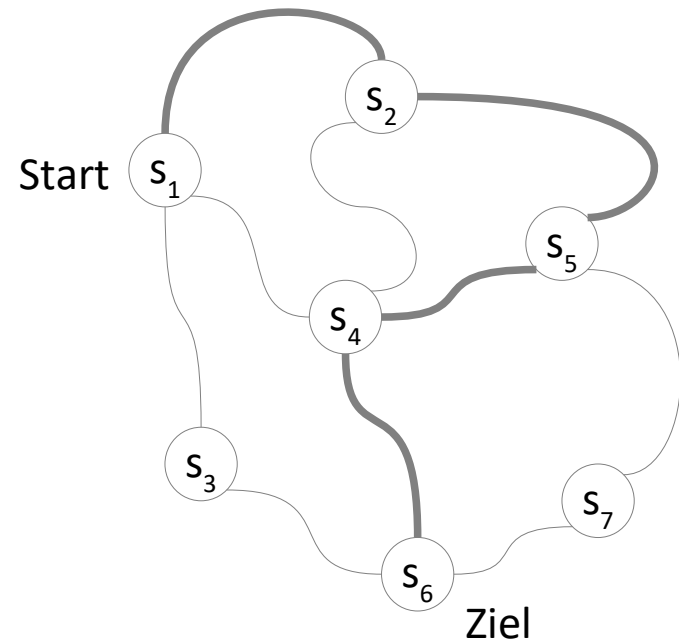


# \* Suchen – Begriffsbildung

- Suchproblem
  - Zustandsraum  $S = \{s_1, \dots, s_n\}$ , Menge von Zuständen
  - Operatoren (Zustandsübergänge) als partielle Funktionen  $o: S \rightarrow S$ 
    - Ein Operator  $o$  führt einen Zustand  $s_i$  in einen Zustand  $s_j$  wenn  $o(s_i) = s_j$
  - Initialzustand  $s_i \in S$ , Startzustand
  - Zielbeschreibung  $G: S \rightarrow \text{Bool}$ , Ziel ist mit  $s$  erreicht wenn  $G(s)$  gilt
    - Alternativ Menge von Zielzuständen  $S_G$  angeben
- Pfade
  - Ein *Pfad*  $p$  ist eine geordnete Operatorenfolge  $\langle o_1, \dots, o_n \rangle$
  - Hintereinanderausführung:  $p(s_i) = s_j$  gdw  $o_n(o_{n-1}(\dots o_1(s_i) \dots)) = s_j$
  - Zustand  $s_j$  ist *erreichbar* von  $s_i$  falls ein Pfad  $p$  von  $s_i$  nach  $s_j$  existiert
- Pfadkosten  $g(p)$ 
  - Meist Summe der einzelnen Operatoren  $g(\langle o_1, \dots, o_n \rangle) = g(o_1) + \dots + g(o_n)$
  - Kosten können von besuchten Zuständen abhängen
  - Kosten können uniform sein, Pfadkosten = Länge des Pfades

# \* Suchproblem und Lösung

- Ein *Suchproblem* besteht aus
  - Zustandsmenge  $S$
  - Ein Initialzustand  $S_I$
  - Zielbeschreibung  $G$
  - Operatorenmenge  $O$
  - Pfadkostenfunktion  $g$
- Eine *Lösung* eines Suchproblems ist
  - ein Pfad  $p$ , der den Initialzustand  $S_I$  in einen Zustand  $s_G \in S_G$  überführt, der die Zielbedingung erfüllt
- Lösungskosten einer Lösung ist
  - der Wert der Pfadkostenfunktion von  $S_I$  nach  $s_G$



$$S = \{s_1, s_2, s_4, s_4, s_5, s_6, s_7\}$$

$$S_I = s_1$$

$$S_G = \{s_6\}$$

$$s_G = s_6$$

$$\text{Lösung } p = \langle o_{12}, o_{25}, o_{54}, o_{46} \rangle$$

$$\text{Uniforme Kosten: } \forall i,j: g(o_{ij}) = 1$$

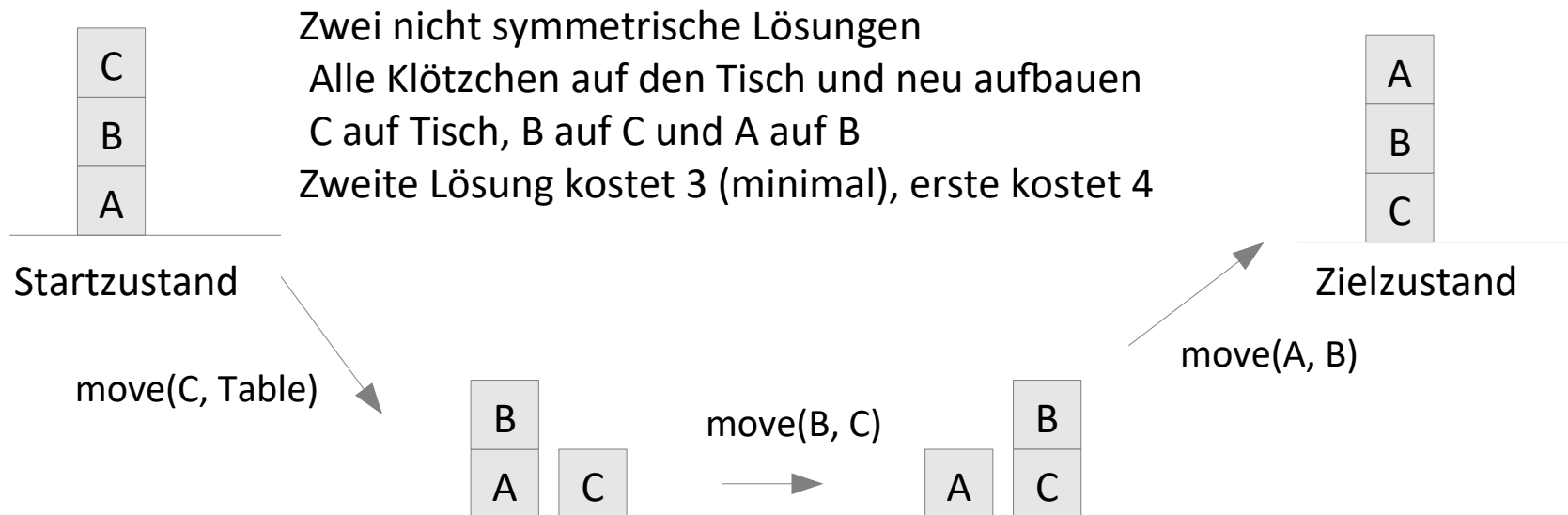
$$g(p) = 4$$

# \* Suchprobleme – Eigenschaften

- Problem: Riesige Suchgraphen, Beispiel PL1
- Ziel: schnell Lösungspfad finden
  - Suchgraph verkleinern
  - „Geschickt“ suchen
- Eigenschaften
  - Zustandsraum durch Operatoren verbunden ist  
Baum oder Graph (Zyklen)
    - Zum Beispiel etwas wegnehmen oder wegnehmen und zurücklegen
  - Äquivalente Zustände
    - Zustände, die dasselbe Problem anders beschreiben
    - Beispiele: Symmetrie, Listen statt Mengen bei beliebiger Reihenfolge
  - Kostenfunktionen, ein Pfad oder optimaler Pfad
- Probleme
  - Zyklen vermeiden, wenn unvermeidlich erkennen
    - Keine Pfade mit Zyklen
  - Äquivalente Zustände vermeiden
    - Problemmodellierung anpassen

# \* Suchprobleme – Beispiel Blocksworld

- Zustandsmenge: Anordnung von Klötzchen auf einem Tisch
- Operatoren: Klötzchen vom Turm nehmen oder drauf legen
- Startzustand, Zielzustand: Zwei beliebige Anordnungen
- Pfadkosten: Uniform, Anzahl der Klötzchenbewegungen



Pfad der zweiten Lösung: <move(C, Table), move(B,C), move(A, B)>

# \* Suchprobleme – Beispiel 8-Puzzle

- Zustandsmenge: Anordnung von 8 Kacheln auf 3x3 Feld
- Operatoren: Bewegung von Kacheln auf freies Feld
- Startzustand, Zielzustand: Zwei beliebige Anordnungen
- Pfadkosten: Uniform, Anzahl der Kachelbewegungen

5	4	
6	1	8
7	3	2

Startzustand

NP-vollständig  
9!/2 erreichbare Zustände  
bei 4x4 ~1.3 Billionen  
bei 5x5  $\sim 10^{25}$  –  
noch heute schwer  
zu lösen

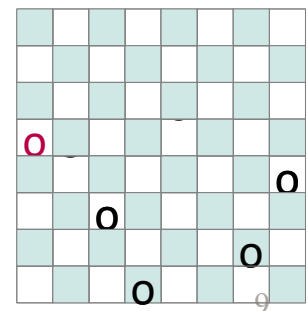
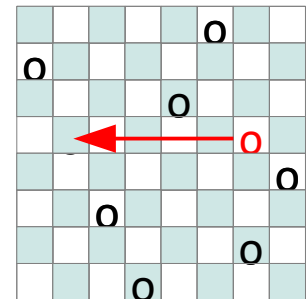
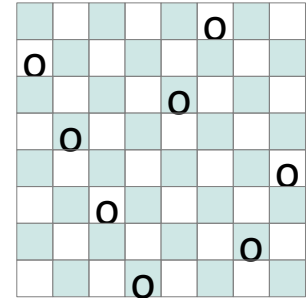
1	2	3
8		4
7	6	5

Zielzustand



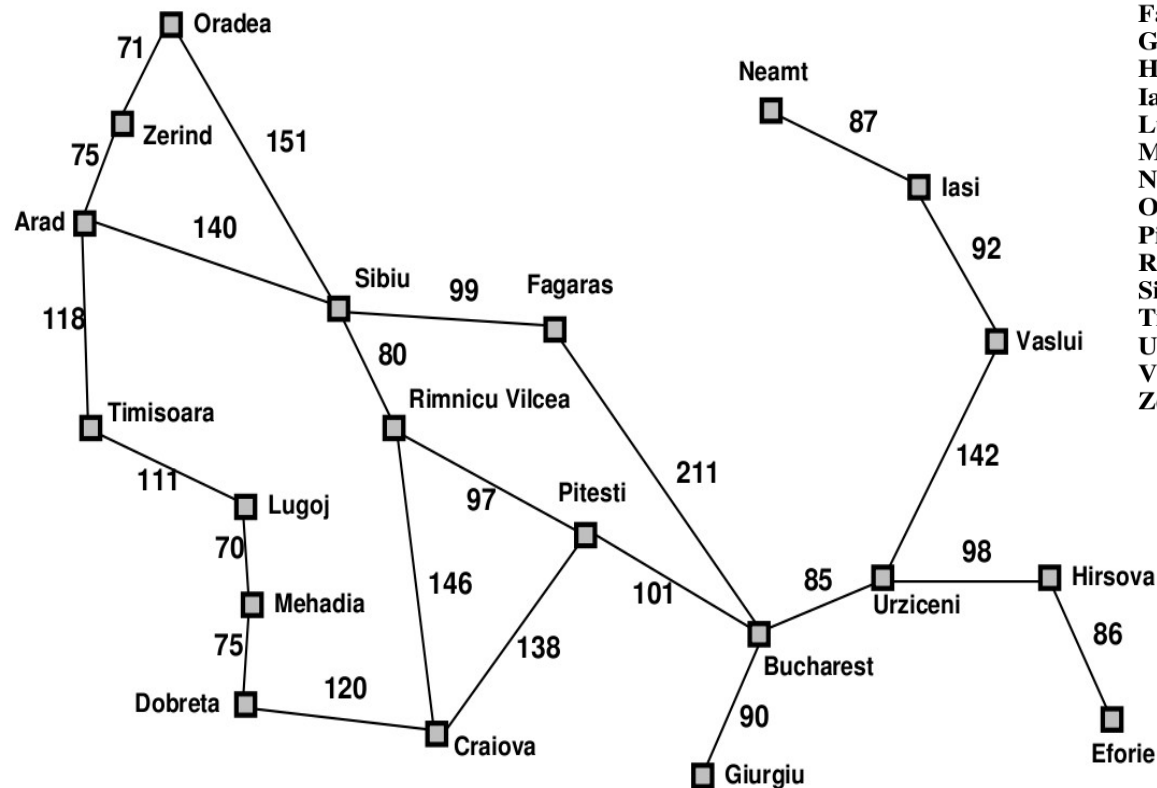
# \* Suchprobleme – N-Damen

- N-Damen Problem
  - N Damen auf einem Schachbrett der Größe NxN verteilen
  - Constraints
    - Keine Dame attackiert eine andere
  - Problemgrößen
    - Standardbeispiel hat Größe 8
- Formulierungen – unterschiedlicher Zustandsraum
  - Vollständig: Ändern
    - Alle Damen auf dem Brett (potentiell sich angreifend)
    - Jeder Operator setzt eine Dame um
    - Größe des Zustandsraums  $64 \cdot 63 \cdot \dots \cdot 57 \sim 1.8 \times 10^{14}$
  - Inkrementell: Erweitern
    - Je eine Dame in einer Reihe
    - Damen auf unteren Reihen (ohne Lücken)
    - Jeder Operator fügt Dame nicht attackierend hinzu
    - Deutlich kleinerer Suchraum: 2057



# \* Suchprobleme – Routenplanung

- Zustandsmenge: Städte
- Operatoren: Strassenfahrt Stadt zu Stadt
- Startzustand, Zielzustand:  
Zwei beliebige Städte
- Pfadkosten:  
je Übergang unterschiedlich,  
Strassenkilometer,  
Gesamtkosten  
Gesamtstrassenkilometer



Luftlinienentfernung  
zu Bucharest

Arad	366
Bucharest	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

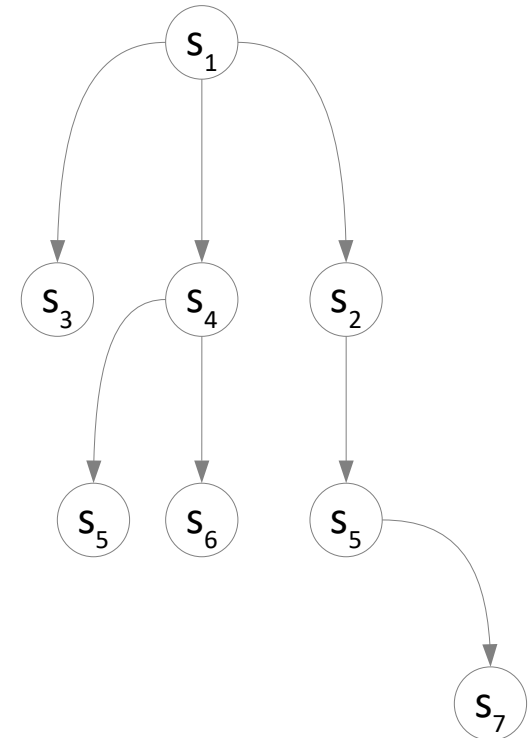
Quelle: Artificial Intelligence, Russel, Norvig,  
<http://www.cs.berkeley.edu/~russell/slides/>, für alle weiteren Routenplanerbeispiele

# Uninformierte Suche

- Suchen: Finde Pfad vom Startzustand zu einem Zielzustand
- Ansatz: Systematische Suche im Zustandsraum
  - Beginne beim Initialzustand (Vorwärtsgerichtete Suche)
  - Bestimme Nachfolgezustände
    - Mögliche Operatoren anwenden bis Zustand, der Zielbedingung erfüllt, erreicht
  - Suchstrategie
    - Bestimmt die Reihenfolge, in der Nachfolgezustände betrachtet werden
  - Uniformiert
    - Kein Verwenden von zusätzliches Bereichswissen
- Rahmenbedingungen
  - Vollständigkeit?: Wenn eine Lösung existiert, dann wird sie gefunden
  - Optimalität?: Finde die „beste“ Lösung, zum Beispiel geringste Kosten
  - Technische Rahmenbedingungen
    - Zeitkomplexität: Wie lange dauert es?
    - Speicherkomplexität: Wie viele Zustände muss ich gleichzeitig halten?

# \* Suchbaum

- Wird während des Suchens implizit oder explizit aufgebaut
- Besteht aus Knoten und Kanten
  - Knoten repräsentieren Zustände
  - Kanten repräsentieren Operatoranwendungen
  - Nachfolgerknoten wurde durch Operator von Vorgängerknoten aus erreicht
  - Wurzel ist Startzustand
  - Blattknoten sind noch nicht *expandiert* oder kein Operator ist mehr anwendbar
  - Zielknoten ist ein Blatt
- Pfadkosten und Tiefe
  - Bei jedem Knoten die Kosten des einen Pfads Wurzel bis Knoten
  - Tiefe ist Pfadkosten bei uniformen Kosten
- [Könnte auch ein Suchgraph sein]



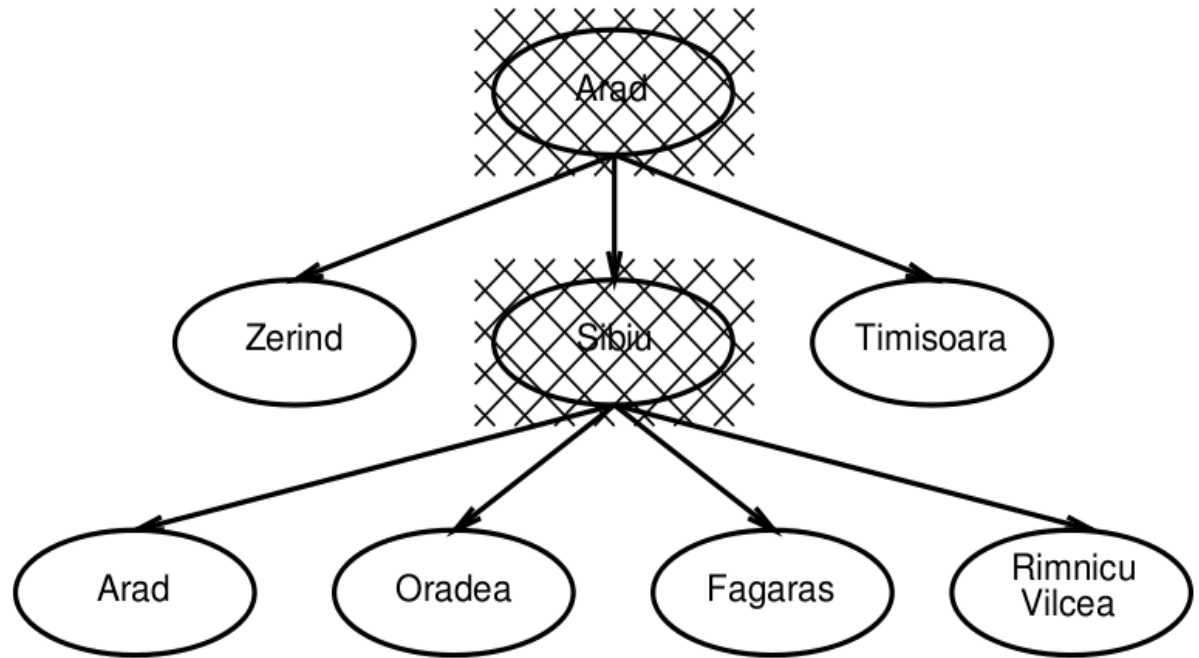
# \* Beispiel – Expansion Routenplanung

- Suche Route von Arad nach Bucharest

- Arad und Sibiu sind beide expandiert, alle Nachfolger sind im Suchbaum

- Achtung

- Zustandsraum  $\neq$  Knoten im Suchbaum
- Es können welche fehlen (die noch nicht erreicht worden sind)
- Es könnten doppelte vorkommen (ein Graph, im Beispiel Arad)
- Zustandsraum ist unendlich, man kann auf Strassen hin und her fahren



# \* Universeller Suchalgorithmus

- Warteschlange
  - Blätter, noch abzuarbeitende Knoten
  - Meist deque, double ended queue

- Solange noch Knoten abzuarbeiten

- Hole Knoten aus Queue
- Expandiere Knoten

- Alle möglichen Nachfolger durch Menge von Operatoren
- Vermeide Wiederholungen

- Füge nur noch nicht erreichte Knoten hinzu
  - Vermeide Endlosschleifen in der Berechnung

- Strategie – Art des Hinzufügens an Warteschlange

- Ans Ende (gegenüber der Stelle an der rausgeholt wird), Breitensuche
- An den Anfang (an der Stelle an der rausgeholt wird), Tiefensuche

```
def search((start, expand, strategy,
is_goal)
    queue = [start]
    reached = [start]
    while queue:
        state = queue.pop()
        if is_goal(state):
            return state # eine Lösung
        reached.push(state)
        ex = expand(state, ops)
        newex = [s for s in ex if ex not in reached]
        queue = strategy(queue, newex)
    return None # keine Lösung
```

# \* Vermeide wiederholte Zustände

- Wiederholte Zustände

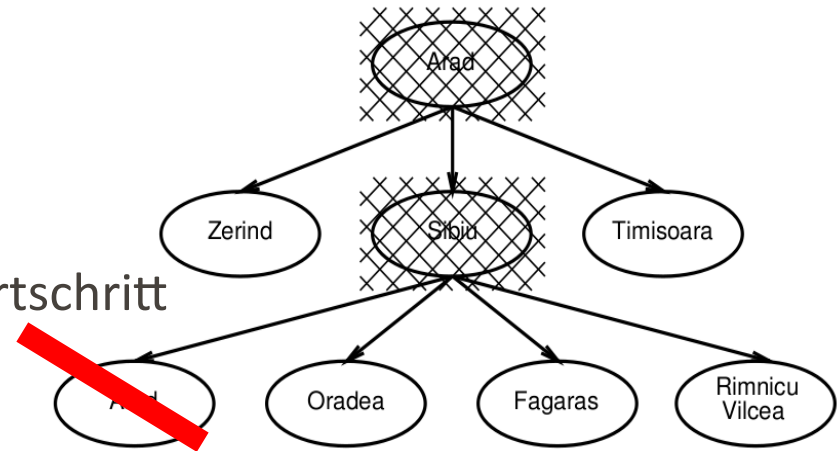
- Wiederholung falls mehrere Pfade zu einem Zustand
- Immer dann, wenn Operationen umkehrbar

- Wiederholte Zustände bringen keinen Fortschritt

- Bei Erkennen Wiederholung kann Suchbaum beschnitten werden
- Verursacht aber zusätzlichen Berechnungsaufwand

- Mögliche Ansätze

- Modellerierung so, dass Zustände sich nicht wiederholen, oft reicht es nicht zu einem Zustand zurückkehren von dem man gerade gekommen ist
- Nicht zu Zustand zurückkehren, der auf Pfad zum Knoten liegt
  - Man muss sich nur Knoten auf dem Pfad merken
- Nicht zu einem Zustand gehen, der bereits im Suchbaum vorhanden ist
  - Speicherung aller Zustände, hohe Speicherkomplexität
  - Effizient implementierbar (Mengen mit Bäumen oder Hashing)



# ✳ Beispiel – N-Damen-Problem

- Initial
  - Keine Dame gesetzt
  - 8 Damen
- Check
  - Neues gesetzt
  - Weder in Spalte noch in Diagonale?
- Expand
  - Alle 8 möglichen Positionen in der Zeile
  - Nur konfliktfreie
- Ziel wenn 8
- Strategie
  - Tiefensuche
  - An den Anfang

```
start, N = [], 8 # keine Damen drin, 8-Damen Problem
```

```
def check(new_state):  
    pos = new_state[0]  
    if pos in new_state[1:]: # auf gleicher Spalte  
        return False  
    for i in range(1, len(new_state)): # gleiche Diagonale  
        if pos+i == new_state[i] or pos-i == new_state[i]:  
            return False  
    return True
```

```
def expand(state):  
    result = [] # mögliche Folgezustände  
    for pos in range(1, N+1):  
        new_state = [pos] + state # neuer Zustand  
        if check(new_state):  
            result.append(new_state)  
    return result
```

```
unisearch.search(start, expand, depthfirst, is_goal)
```

```
def is_goal(state): return len(state) == N  
def depthfirst(queue, nodes): return nodes+queue
```



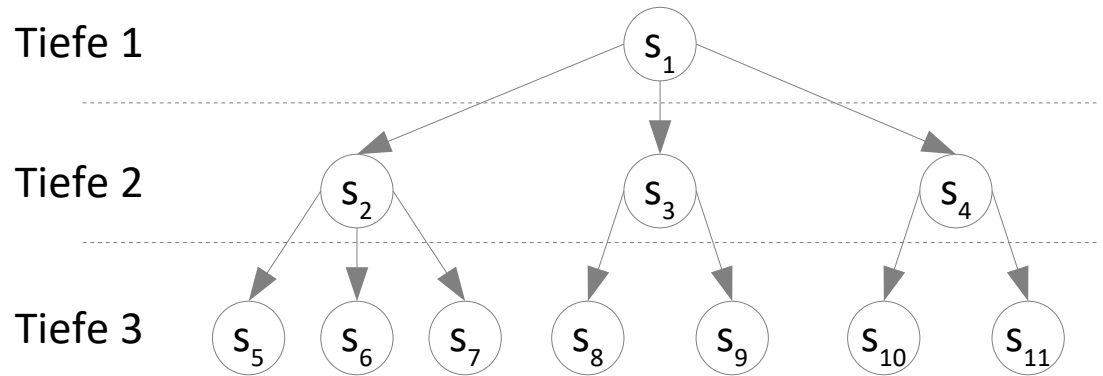
# \* Breitensuche

- Strategie

- Erst alle Knoten einer Tiefe, dann das nächsttiefere Level
- Reihenfolge: 1,2,3,4,5,6....
  - Reihenfolge innerhalb der Tiefe kann variiert werden, z. B. 1, 4,3,2, 11, 10, ...
- Umsetzung: Einfügen der neuen Knoten ans Ende der Warteschlange

- Eigenschaften

- Vollständige Strategie, Optimalität bei uniformen Pfadkosten
- Aufwand: Annahme fester Verzweigungsgrad  $b$  (Anzahl Nachfolger)  
 $1 + b + b^2 + \dots + b^{d-1} \in O(b^d)$ ,  
bei Lösung auf Tiefe  $d$   
Fast alle Knoten im Speicher,  
Speicheraufwand größtes Problem
- Beispiel:  $b=10$ , 100 Byte/Knoten,  
1000 Knoten pro Sekunde



**def breadthfirst(queue, nodes): return queue+nodes**

Tiefe	Knoten	Zeit	Speicher
4	11.111	11s	1 MB
8	$10^8$	31h	11 GB
10	$10^{10}$	128d	1 TB
12	$10^{12}$	35y	111 TB
14	$10^{14}$	3500y	11 PB

# \* Uniforme Kostensuche

- Strategie

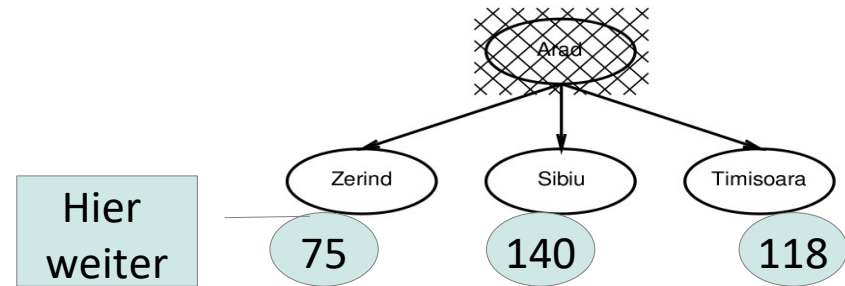
- Expandiere Knoten mit geringsten Kosten in Warteschlange

- Ersetze Warteschlange durch Prioritätswarteschlange oder
- Füge Knoten nach Kosten aufsteigend sortiert ein

- Sinnvoll, wenn Kosten je Schritt unterschiedlich sind, zum Beispiel Routenplanung

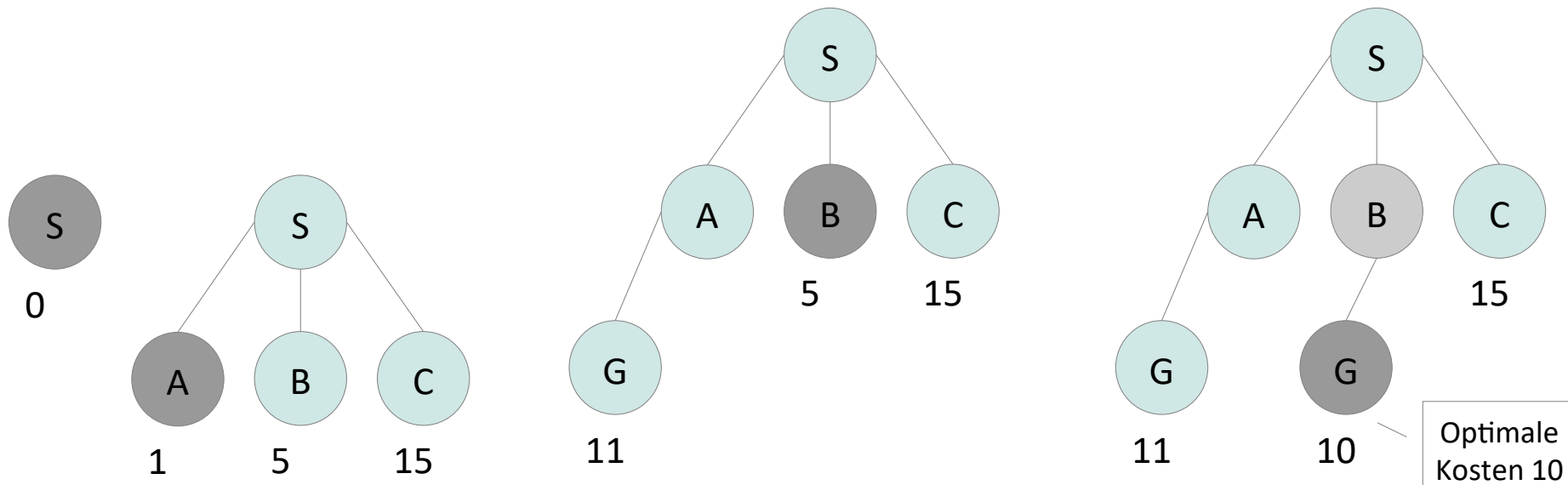
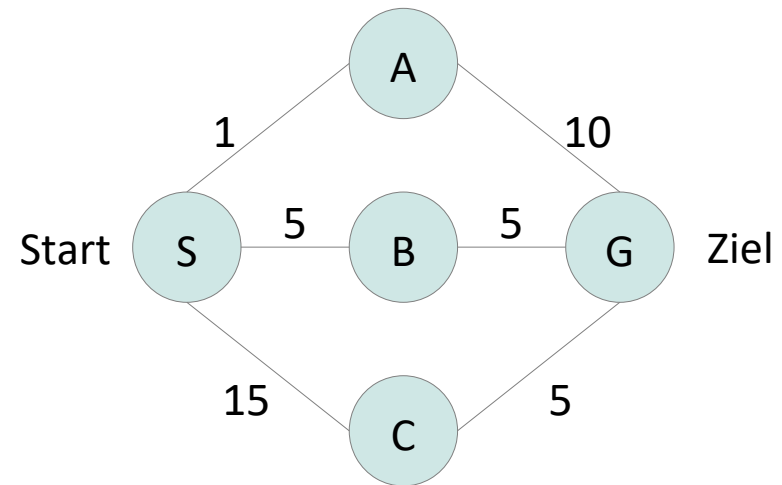
- Eigenschaften

- Vollständige Strategie
- Optimalität bei positiven Pfadkosten
  - Beachte, dass erst aufgehört wird, wenn Zielzustand aus Queue kommt, und nicht gleich wenn er in der Menge der expandierten Knoten ist
- Aufwand wie bei Breitensuche:  
Zeitkomplexität  $O(b^d)$   
Speicherkomplexität  $O(b^d)$



# \* Uniforme Kostensuche – Beispiel

- Zustandsraum mit Kosten je Operatoranwendungen
  - 5 Zustände: S, A, B, C, G
  - Startzustand S
  - Zielzustand G
- Suchbaum



# \* Tiefensuche

- Strategie

- Immer den zuletzt hinzugefügten Knoten zuerst

- Erst in die Tiefe, nur bei Misserfolg in die Breite

- Reihenfolge: 1,2,5,6,7,3,8,9,4, ....

- Reihenfolge innerhalb der Tiefe kann variiert werden, z. B. 1,4,11,10,2,7, ...

- Umsetzung: Einfügen der neuen Knoten an Anfang der Warteschlange

- Eigenschaften

- Nicht vollständig bei unendlichen Suchbäumen

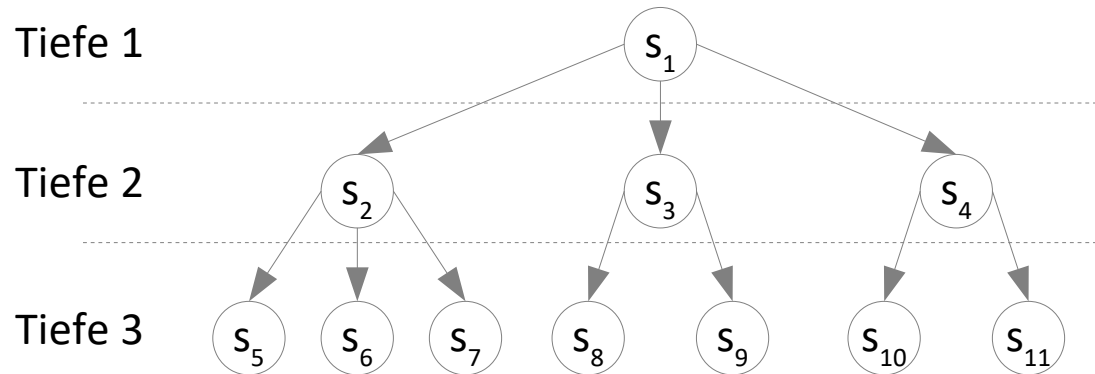
- Kann in unendlichem Pfad stecken bleiben
  - Häufig effektiv, wenn es viele Lösungen gibt

- Keine Optimalität

- Aufwand:

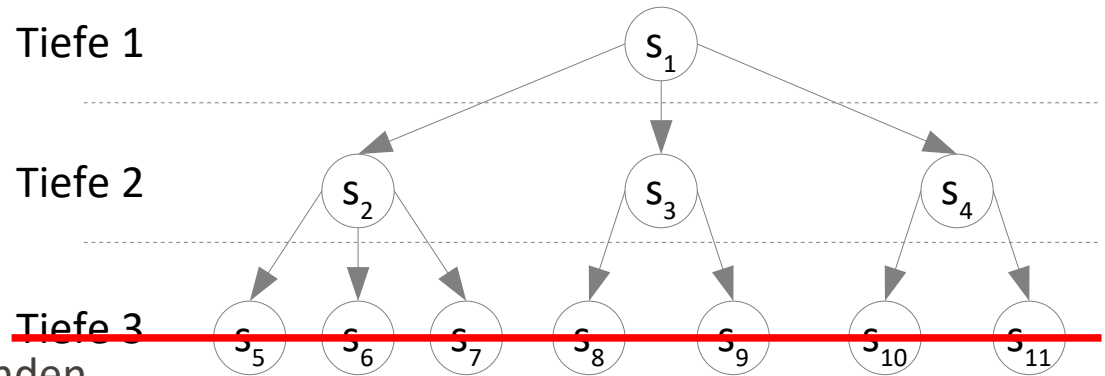
Zeitkomplexität  $O(b^d)$

Speicherkomplexität  $O(d*b)$ , viel besser als Breitensuche



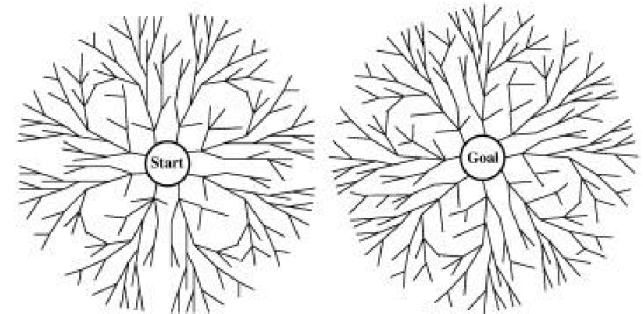
# ✳ Beschränkte Tiefensuche

- Idee: Abschneiden des Suchbaums bei Erreichen einer bestimmten Tiefe  $t$ 
  - Falls Lösung bis Tiefe  $t$  existiert, dann wird diese gefunden
  - Speicherkomplexität bleibt mit  $O(b \cdot t)$  gering!
  - Keine Optimalität
  - Implementierung: Tiefe mitführen, ab Tiefe  $t$  nicht mehr in Warteschlange
- Iterative Tiefensuche (iterative deepening)
  - Beschränkte Tiefensuche schrittweise mit höherer Tiefe wiederholen
  - Kombination der Vorteile von Tiefensuche und Breitensuche
    - Vollständig und optimal bei uniformen Kosten
    - Entgegen der Intuition nicht signifikant mehr Rechenaufwand, gleiche Zeitkomplexität  $O(b^d)$
    - Geringer Speicherbedarf  $O(d \cdot b)$
  - Gut geeignet für große Suchräume ohne Tiefenbeschränkung



# ✱ Bidirektionale Breitensuche

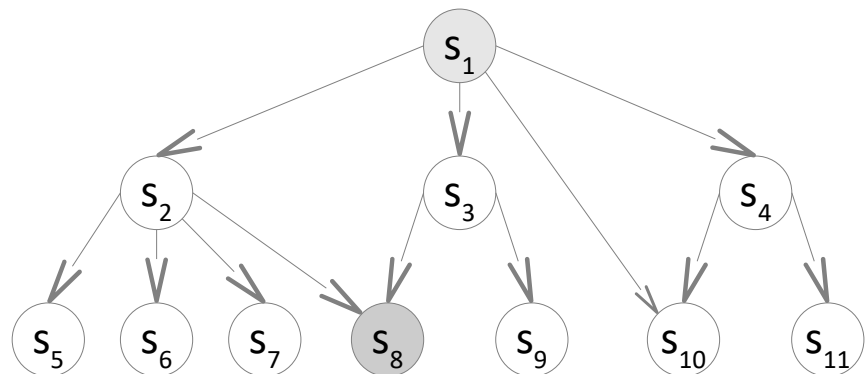
- Idee: Gleichzeitig suchen
  - Suche beginnt sowohl vom Startzustand als auch vom Zielzustand (wenn der eindeutig ist)
  - Ende ist erreicht, wenn sich zwei Suchzweige in der Mitte treffen
- Voraussetzung
  - Operatoren müssen umkehrbar sein, Vorgängerzustände sind zu bestimmen für Gesamtlösung
- Umsetzung
  - Terminierungsbedingung aufwendiger: Test ob Knoten schon im anderen Suchbaum enthalten
- Eigenschaften
  - Wenn  $b$  in beide Richtungen gleich ist, dann Zeitkomplexität:  $O(b^{d/2})$ , Speicherkomplexität:  $O(b^{d/2})$
  - Vollständigkeit und Optimalität bei uniformen Pfadkosten



Quelle: Artificial Intelligence, Russel, Norvig

# \* Vorwärtssuche versus Rückwärtssuche

- Suchrichtung
  - Statt von Start zu Ziel kann man auch von Ziel zu Start suchen
  - Frage der Modellierung
- Voraussetzung
  - Umkehrung der Operatoren muss möglich sein
  - Definition von Umkehroperatoren ( $o^{-1}$ )
- Welche Richtung wählen?
  - Problemabhängig
  - Falls unterschiedlicher Verzweigungsgrad, dann meist geringerer Verzweigungsgrad vorteilhaft
- Beispiel
  - $s_1$  nach  $s_8$   
Verzweigungsgrad 4
  - $s_8$  nach  $s_1$   
Verzweigungsgrad 2



# \*Uninformierte Suchverfahren – Vergleich

Kriterium	Breiten- suche	Uniforme Kostensuche	Tiefen- suche	Iterative Tiefensuche	Beschränkte Tiefensuche	Bidirektionale Breitensuche
Zeit	$O(b^d)$	$O(b^d)$	$O(b^m)$	$O(b^d)$	$O(b^t)$	$O(b^{d/2})$
Speicher	$O(b^d)$	$O(b^d)$	$O(b \cdot m)$	$O(b \cdot d)$	$O(b \cdot t)$	$O(b^{d/2})$
Optimalität	ja <sup>1</sup>	ja	nein	ja <sup>1</sup>	Nein	ja <sup>1</sup>
Vollständig- keit	ja	ja	nein	ja	ja, wenn $t \geq d$	ja

b = Verzweigungsgrad  
 d = Tiefe der Lösung  
 m = Maximale Tiefe  
     des Suchbaums  
 t = Tiefenlimit MaxTiefe

<sup>1</sup> nur bei uniformen Kosten



- Einführung
- Symbolische Verfahren, Logik
  - Aussagenlogik, Prädikatenlogik
  - Horn Logik, Prolog
- **Suchen und Bewerten**
  - **Problemlösen durch Suche**
    - Uninformierte Suche
    - **Heuristische Suche**
    - Spielbäume
  - Information Retrieval
- Lernen
  - Entscheidungstheorie
  - Naive Bayes
  - Entscheidungsbäume
  - Neuronale Netze
  - unüberwachtes Lernen

# \* Heuristische Suche, Bestensuche

- Problem
  - Uninformierte Suche generiert *blind* Zustände und testet, keine problemspezifischen Strategien
  - Häufig sehr lange Laufzeiten
- Lösung – Heuristische Suche mit Vorwissen
  - Vorwissen hilft bei der Steuerung, generiere *gute* Zustände
  - Vermeide Sackgassen, effizientere Suche

# \* Heuristische Suche, Bestensuche

- Verfahren – Bestensuche
  - Auswahl zu expandierender Knoten bisher nach Pfadkosten
    - Breitensuche: geringste Gesamtkosten bis zum aktuellen Knoten zuerst
    - Tiefensuche: höchste Gesamtkosten bis zum aktuellen Knoten zuerst
  - Bestensuche – statt Fakt ab Start; Heuristik bis Ziel
    - Heuristikfunktion (geschätzte Kosten bis Ziel) ist Vorwissen
  - Greedy-Suche
    - Wähle Knoten zur Expansion mit geringsten geschätzten Kosten bis Ziel
  - A\*-Suche
    - Wähle Knoten zur Exp. mit geringsten geschätzten Gesamtkosten bis Ziel

# \*-Heuristik, Schätzfunktion

- Schätzfunktion/Heuristik  $h(s): S \rightarrow \mathbb{Q}$ 
  - Berechnet für jeden Knoten einen Schätzwert der Kosten bis zum Ziel
  - Schätzwert: genaue Kosten nicht bekannt (warum sonst Suchen?)
  - Optimistisch (unter) oder pessimistisch (über tatsächlichen Kosten)
  - Für alle  $s$ :  $h(s) \geq 0$ ; für Zielknoten  $z$ :  $h(z) = 0$
  - Meist kann eine optimistische Schätzfunktion angegeben werden

# \* Heuristik, Schätzfunktion

- Beispiel: Problem zur Routenfindung

- Heuristikfunktion

- ist Entfernung

- in Luftlinie

- Kosten ist

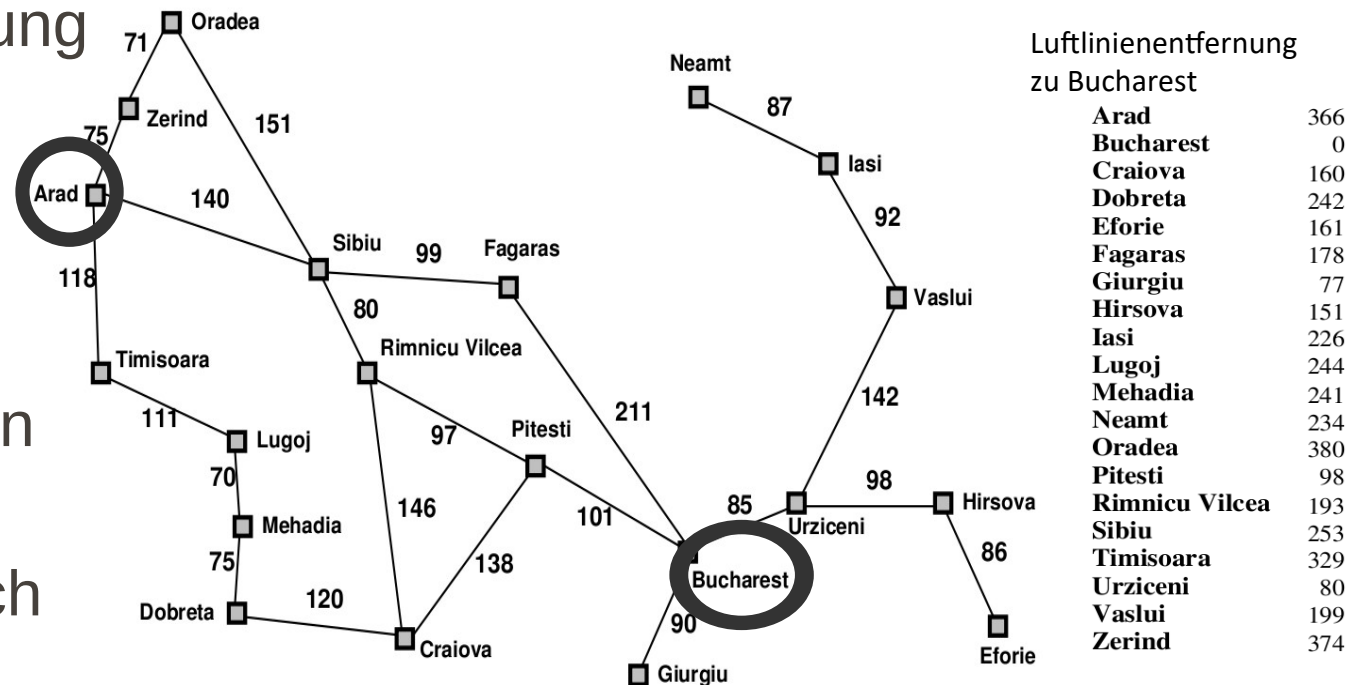
- Länge

- des

- gewichteten

- Pfads

- Optimistisch



# \* Gierige (Greedy) Suche

- Bewertungsfunktion, Heuristik
  - Geschätzte Kosten bis zum Ziel
- Umsetzung Bestensuche
  - Prioritätswarteschlange statt Warteschlange
  - Ergebnis der Kostenfunktion als Wert

# \* Gierige (Greedy) Suche

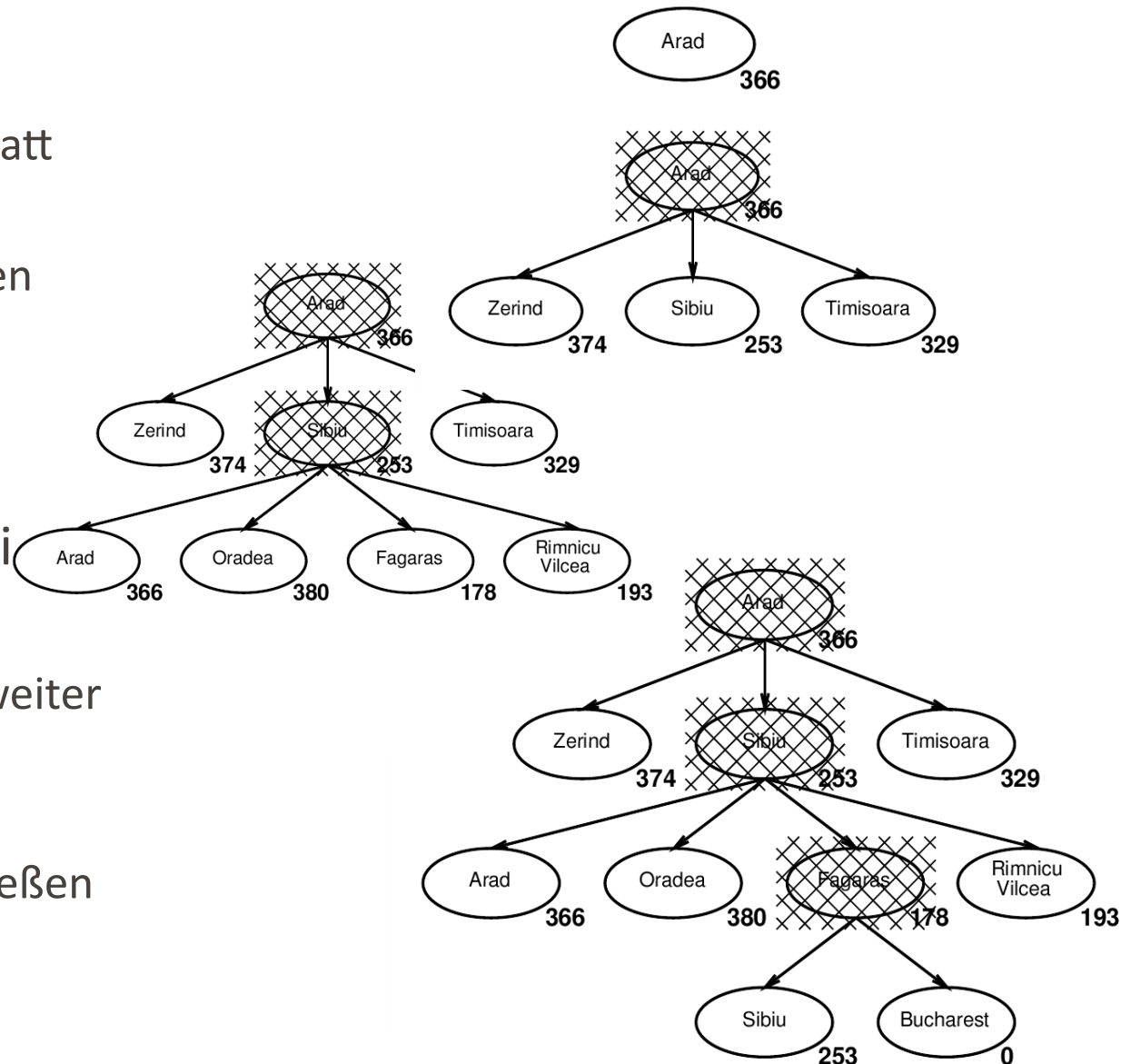
- Beispiel

## Routenplanung

- Nicht optimal, 450 statt 418
- Gesamtkosten werden nicht betrachtet
- Wenige Knoten expandiert

- Achtung, Beispiel lasi nach Fagaras

- Doppelte Zustände weiter möglich
- Erreichte Zustände merken und ausschließen



# \* Gierige Suche – Bewertung

- Greedy Search, ähnlich zu Tiefensuche
  - Häufig wird tiefster Pfad weiter verfolgt, außer bei Sackgassen oder falschen Richtungen
  - Auswahl anderer Pfad/Knoten möglich, wenn jede mögliche Auswahl ein Rückschritt ist
  - Nicht optimal
  - Unvollständig, wenn mehrfaches Erreichen der Zustände nicht eliminiert wird
  - Zeitkomplexität:  $O(b^m)$
- Anders als Tiefensuche
  - Speicherkomplexität:  $O(b^m)$   
da alle Knoten im Speicher verbleiben müssen
  - Zeit- und Speicherbedarf kann meist durch gute Schätzfunktion stark reduziert werden
- Verbleibendes Problem: Optimalität
  - Ok für Online-Probleme (Zustandsraum nicht sichtbar/planbar)

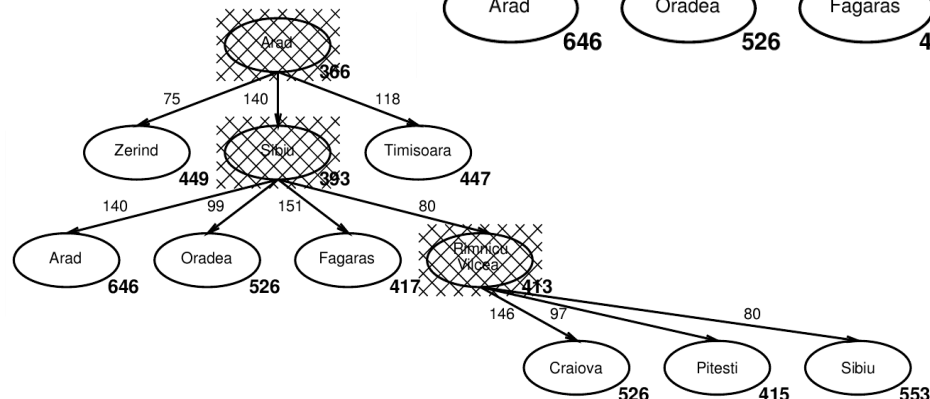
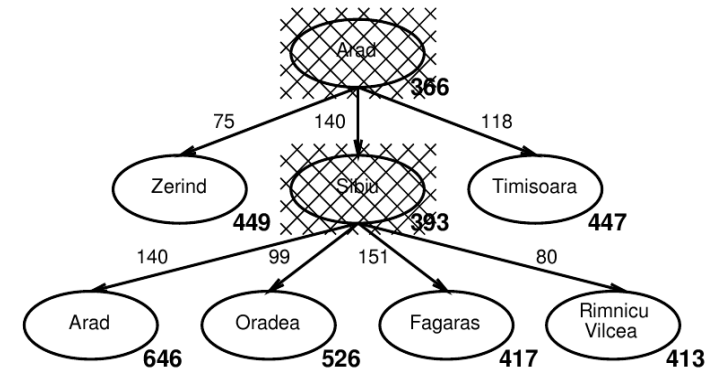
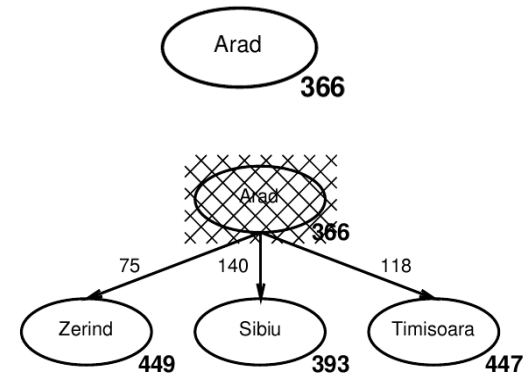


# \*A\*-Suche

- Bewertungsfunktion, Heuristik
  - Bisherige Kosten  $[g(s)]$  + geschätzte Kosten bis zum Ziel  $[h(s)]$
  - $f(s) = g(s) + h(s)$ , für alle  $s$ :  $f(s), h(s) \geq 0$
  - $h(s)$  wie bei gieriger Suche
  - Optimalität falls  $h$  optimistisch, also  $h(s)$  immer kleiner gleich Kosten optimaler Pfad von  $s$  zu Ziel;

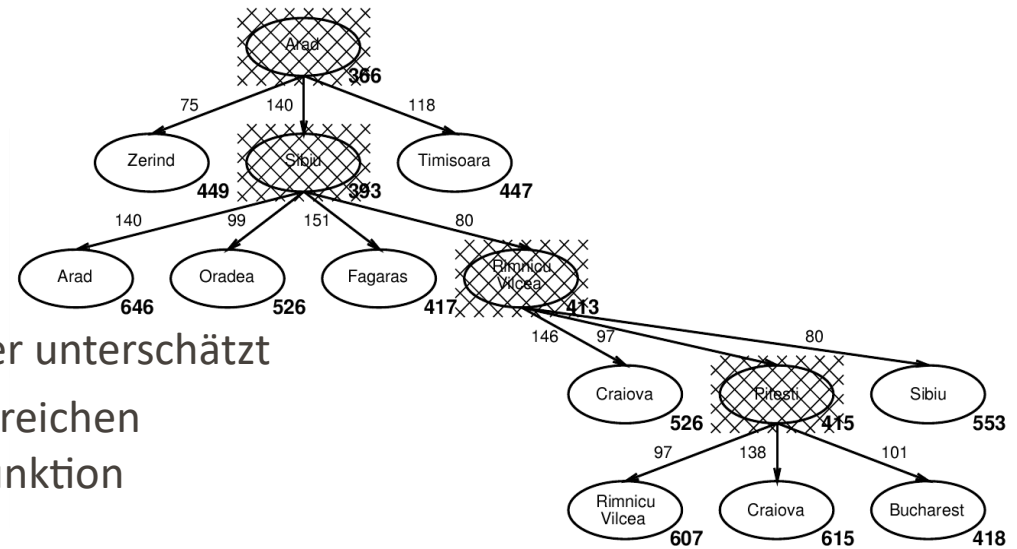
- Umsetzung
  - Prioritätswarteschlange
  - $f(s)$  als Wert

- Beispiel Routenplanung
  - Optimal 418
  - Gesamtkosten werden betrachtet
  - Wenige Knoten expandiert

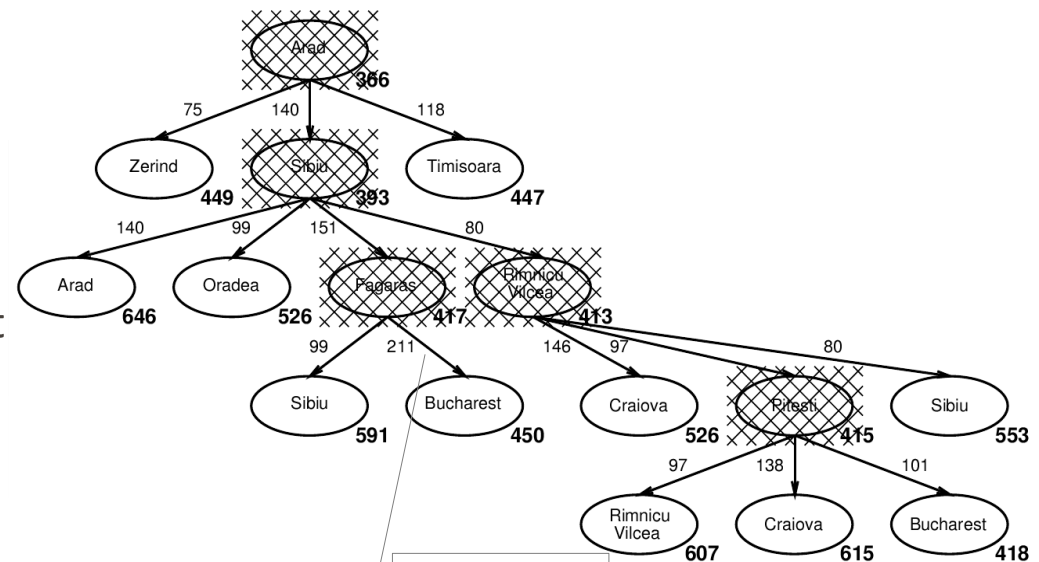


# \*A\*-Suche – Bewertung

- A\*-Suche, ähnlich Breitensuche
  - Nach aktuellem Wissen kürzester Pfad expandieren
  - Auswahl anderer Pfad/Knoten möglich,
  - Optimal, wenn Heuristik-Anteil immer unterschätzt
  - Vollständig, auch bei mehrfachem Erreichen der Zustände (wegen Anteil Kostenfunktion zurückgelegte Strecke)



- Endlicher Zustandsraum oder diskrete Kostenfunktion vorausgesetzt
- Zeitkomplexität:  $O(b^m)$
- Speicherkomplexität:  $O(b^m)$ , bleibt im schlechtesten Fall schlecht
- Zeit- und Speicherbedarf kann meist durch gute Schätzfunktion stark reduziert werden

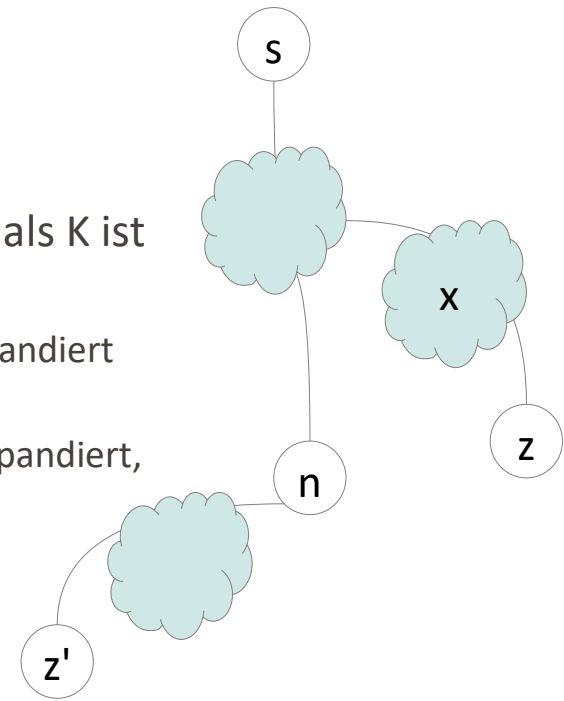


wegen  
Optimalität

- Das Standard Suchverfahren

# \*Optimalität A\*

- Behauptung: A\* ist optimal
  - Das heißt A\* findet einen Pfad mit minimalen Pfadkosten K vom Startknoten s zu einem Zielknoten z
- Beweis: Widerspruch
  - Annahme: A\* ist nicht optimal
  - Dann findet A\* einen Zielknoten z' (z=z' ist möglich) auf einem Pfad ungleich dem optimalen Pfad, also mit größeren Pfadkosten  $K' > K$
  - Sei n der erste Knoten ungleich z, dessen Pfadkosten größer als K ist und der expandiert wird
    - Der Knoten muss existieren, da z' ja auf dem suboptimalen Pfad expandiert wurde, damit größer und ein Kandidat ist
    - Alle Knoten in x inklusive z auf dem optimalen Pfad werden nicht expandiert, da nach Annahme z' als Lösung identifiziert wird
  - Für alle Knoten x auf dem kürzeren Pfad gilt nun
    - $f(x) = g(x) + h(x) \leq f(z)$ , da h unterschätzt
    - Aber  $f(z) < f(n)$  nach Annahme und  $f(n) \leq f(z')$ , also  $f(x) < f(n) \leq f(z')$
  - Wenn aber  $f(x) < f(n)$ , dann wird jeder Knoten x auf dem optimalen Pfad vor n expandiert – Widerspruch



# \* Informiertheit einer Heuristikfunktion

- Ziel
  - Bewertung einer Heuristikfunktion
  - Wann liefert eine Heuristikfunktion eine bessere (schnellere) Suche?
- Definition: *Dominierung*
  - Gegeben seien zwei unterschätzende (zulässige) Heuristiken  $h_1$  und  $h_2$
  - Wir sagen  $h_1$  *dominiert*  $h_2$  wenn für alle  $x \in S$  gilt  $h_1(x) \geq h_2(x)$
- Satz: Monotonie
  - Wenn die Heuristik  $h_1$  die Heuristik  $h_2$  dominiert, dann wird jeder Knoten, der durch die A\* Suche unter Verwendung von  $h_1$  expandiert wird auch durch die A\* Suche unter Verwendung von  $h_2$  expandiert
- Folgerung
  - A\* arbeitet mit  $h_1$  keineswegs langsamer als mit  $h_2$ , aber eventuell viel viel schneller, da weniger Knoten expandiert werden könnten
  - $h_1$  heißt auch *besser informiert* als  $h_2$
  - Bei mehreren zulässigen Heuristikfunktionen können wir für eine dominierende Heuristik für jeden Knoten das Maximum verwenden

# \* Heuristiken – Beispiel für 8-Puzzle

- Eigenschaften 8-Puzzle
  - Verzweigungsfaktor  $b \approx 3$ ; in Ecke 2, Rand 3, Mitte 4
  - Durchschnittliche Lösungslänge 20
  - Erschöpfend  $3^{20}$ ; doppelte weg  $9!/2 = 181.440$  Zustände
- Heuristik  $h_1$ : Anzahl Kacheln falsch
  - Im Startzustand zum Beispiel 7
    - $1 + 1 + 1 + 1 + 1 + 1 + 0 + 1 = 7$
- Heuristik  $h_2$ : Entfernung Kacheln von Zielposition
  - Summe der Manhattan-Distanzen
  - Im Startzustand zum Beispiel 18
    - $2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$
  - $h_2$  dominiert  $h_1$
- Vergleich – Experiment; Russel, Norvig
  - Lösungslänge 14, Tiefensuche\* > 3 Millionen,  $A^*(h_1)$  539,  $A^*(h_2)$  113 Knoten
  - Lösungslänge 24, Tiefensuche\* ?,  $A^*(h_1)$  39135,  $A^*(h_2)$  1641 Knoten

5	4	
6	1	8
7	3	2

Startzustand

1	2	3
8		4
7	6	5

Zielzustand

\* Tiefensuche als Iterative Deepening

# \* Speicherbegrenzte A\*-Suche

- Problem
  - Speicherverbrauch, da alle nicht expandierten Knoten noch im Speicher gehalten werden müssen (Verhalten Breitensuche)
- Lösungsansatz Iterative Deepening, IDA\*
  - Statt Begrenzen auf Suchtiefe Begrenzen auf maximale Kosten  $f$
  - Schrittweises Erhöhen der maximalen Kosten auf den kleinsten Wert, der im vorherigen Lauf aufgrund der Größe nicht betrachtet wurde
  - Optimalität bleibt erhalten, diskrete Kostenfunktion
  - Speicherbedarf abhängig von maximaler Tiefe bei gegebener Kostengrenze
  - Problem: Viele kleine Vergrößerungen maximaler Kosten; Lösung durch größere Sprünge bei Verzicht Optimalität

# \* Speicherbegrenzte A\*-Suche

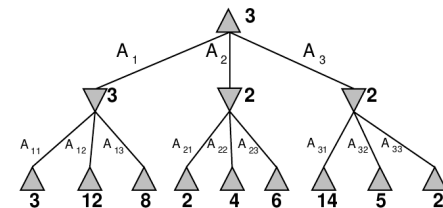
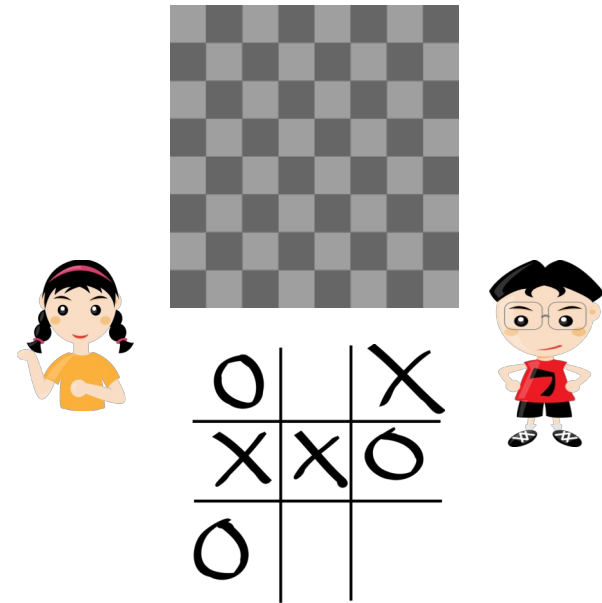
- Lösungsansatz Simplified Memory Bounded, SMA\*
  - Angabe einer maximalen Speichergröße für die Prioritätswarteschlange
  - Löschen der *schlechteren* Knoten (hoher f-Wert) wenn Queue voll
  - Optimal und vollständig, wenn flachster Lösungsweg in Queue passt
  - Ansonsten beste Lösung, die der vorhandene Speicher zulässt

- Einführung
- Symbolische Verfahren, Logik
  - Aussagenlogik, Prädikatenlogik
  - Horn Logik, Prolog
- **Suchen und Bewerten**
  - **Problemlösen durch Suche**
    - Uninformierte Suche
    - Heuristische Suche
    - Lokale Suche
    - **Spielbäume**
- Lernen
  - Entscheidungstheorie
  - Naive Bayes
  - Entscheidungsbäume
  - Neuronale Netze
  - unüberwachtes Lernen



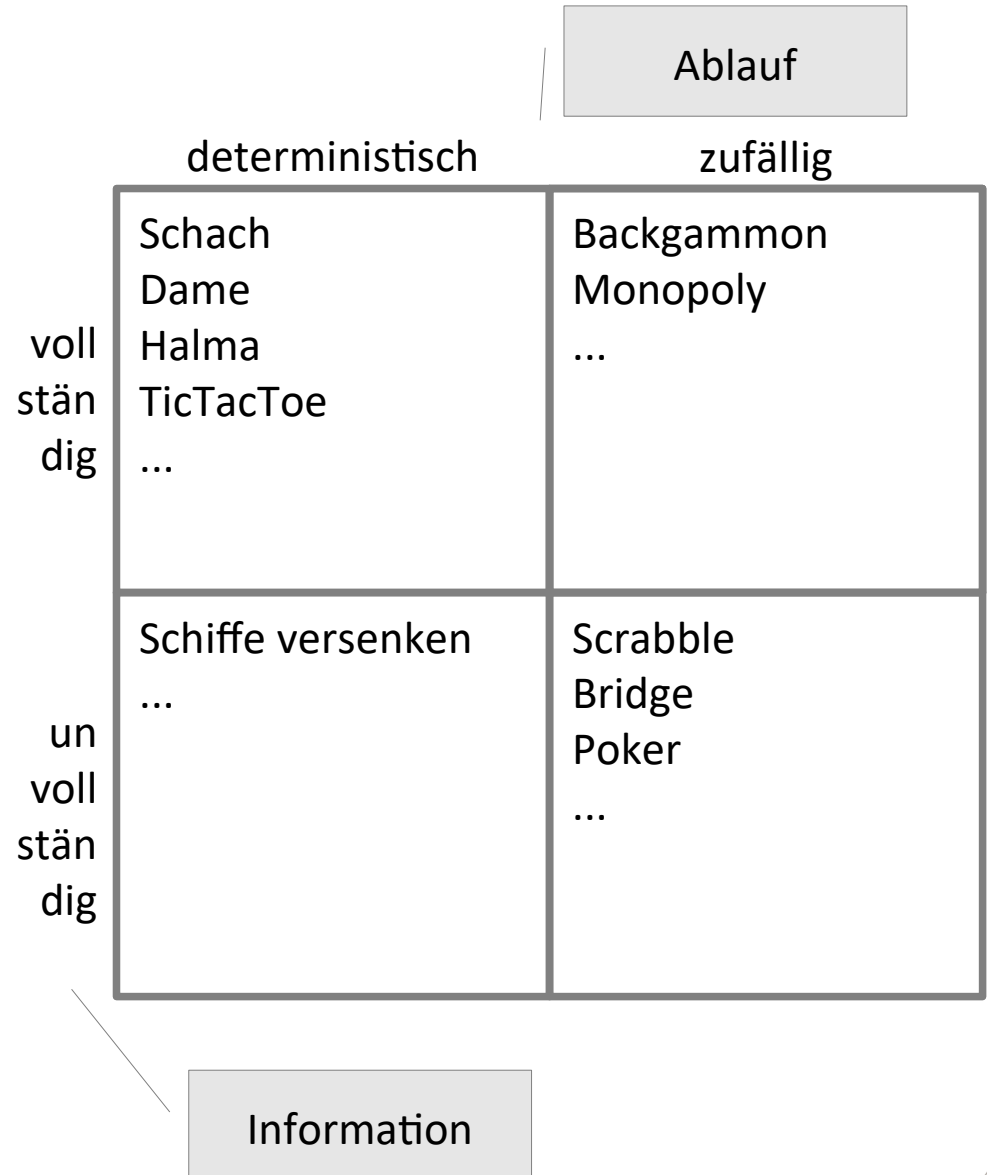
# \* Spielbäume

- Problem
  - Mehrere Agenten agieren, Aktion ändert Welt (für andere Agenten)
  - Adversariale Suchräume, Spiele
  - Meist zwei Spieler, abwechselnd ziehen
- Lösung
  - Suchbäume
  - Heuristiken
- Ansätze, Verfahren
  - Minimax, Alpha-Beta-Kürzung
  - Bewertungsfunktionen für Spiele, Suchbaumbeschränkungen
  - Zufallselemente, ExpectMinimax
  - Ansätze hochwertiger Spielverfahren



# \* Spiele

- Mehrere Spieler
  - *Zweisspieler*
  - Mehrspieler
- *Nullsummenspiele*
  - Einer gewinnt, der andere verliert
  - (Kooperatives Verhalten ab drei Spielern möglich)
- Information
  - *Vollständige*
  - Unvollständige Information über aktuellen Zustand
- Ablauf
  - *Deterministisch*
  - Zufallselemente



# \* Deterministisch, Zwei-Spieler, Nullsumme

- Spieldefinition

- *Zwei Spieler*, Min und Max
- (Zustandsraum als mögliche Brettpositionen)
- Abwechselnd ziehen (Verändern der Position) von Min und Max
- *Ausgangszustand*: Brettposition plus „Wer ist am Zug“
- *Nachfolgerfunktion*:
  - Abbildung Position x Zug auf Menge von Positionen
  - Zug ist Menge gültiger Züge für jeden Spieler am Zug
- *Endetest*: Ist Spiel vorbei?, *Endzustände*
- *Nutzenfunktion*: Bewertung bei Spielende

O		X
X	X	O
O		

- Beispiel: Tic-Tac-Toe

- 9! Zustände (1. Zug 9 Möglichkeiten, 2. Zug 8, ...) realistisch weniger, da vorher Endzustand erreicht
- Nachfolgerfunktion: Wähle freie Position und markiere mit eigenem Zeichen
- Endetest: Drei gleiche Zeichen in Spalte, Zeile oder Diagonale oder voll
- Nutzenfunktion: 1 (Sieg), 0 (Unentschieden), -1 (Niederlage)

O	O	X
X	X	O
O		

O		X
X	X	O
O	O	

O		X
X	X	O
O		O

# \* Optimale Strategie – Minimax

- Voraussetzung

- Zwei perfekte Spieler, die Spiel bis Ende vollständig durchschauen

- Mini-Beispiel

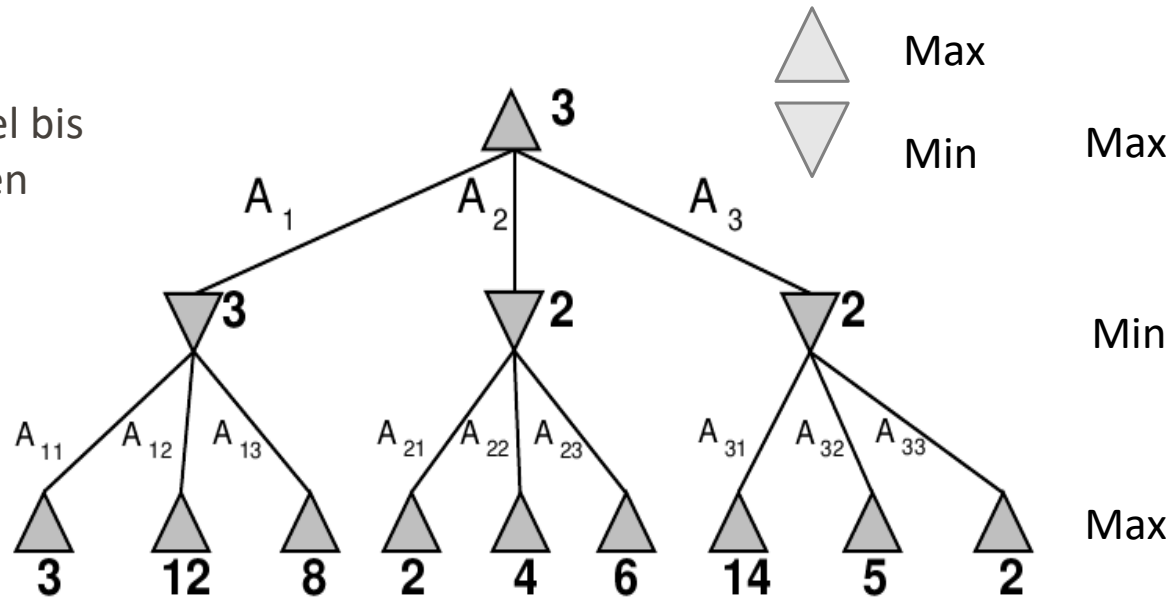
- Genau zwei Züge, Max beginnt
- Max hat größte Bewertung, Min kleinste als Ziel

- Vollständige Durchdringung des Spiels möglich

- Wenn Max  $A_1$  zieht und Min  $A_{11}$ , dann hat Max den Wert 3,
- Wenn Max  $A_1$  zieht und Min  $A_{12}$ , dann hat Max den Wert 12, ...

- Minimax-Algorithmus

- Nach  $A_1$ , würde Min  $A_{11}$  wählen; Nach  $A_2$  Min  $A_{21}$ ; Nach  $A_3$  Min  $A_{33}$
- Für Max ist das der Wert des Zugs  $A_1$ ,  $A_2$ ,  $A_3$  respektive
- Max wählt  $A_1$
- Allgemein: Eine Ebene maximieren, dann minimieren, ....



# \* Minimax-Algorithmus

- Minimax-Algorithmus

- Bewerte am Ende
- Ansonsten für alle Nachfolger
  - Berechne Wert des nächsten Layers
  - Vertausche dabei min/max

- Eigenschaften

- *Vollständig* für endliche Bäume bzw. endliche Gewinnstrategien
- *Optimal* nur gegen perfekten Gegner
- Zeitkomplexität:  $O(b^m)$ ,  $b$  Verzweigungsfaktor,  $m$  maximale Tiefe
- Speicherkomplexität:  $O(bm)$ , Tiefensuche

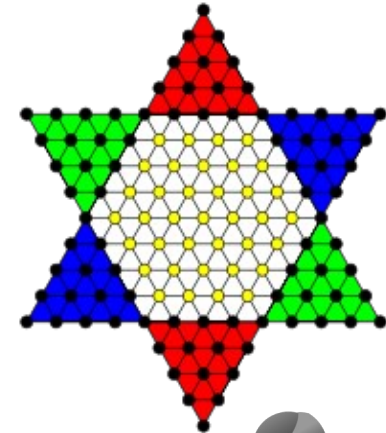
- Problem – Zeitkomplexität

- Beispiel Schach:  
 $b \approx 35$ ,  $m \approx 100$ , normalerweise nicht mehr praktisch lösbar
- Ansatz: Nicht jeden Pfad verfolgen

```
def minimax(start, expand, evaluate,
m=max):
    if evaluate(start) != None:
        return evaluate(start), start
    n_m = min if m == max else max
    layer = []
    for n in expand(start):
        val,_ = minimax(n, expand, evaluate, n_m)
        layer.append((val, n))
    return m(layer)
```

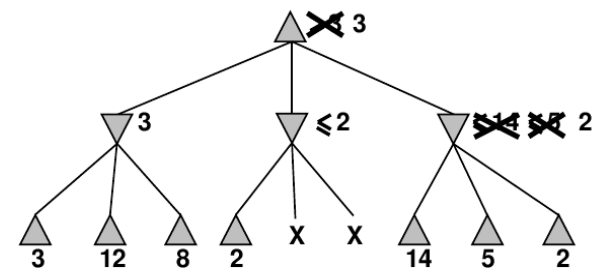
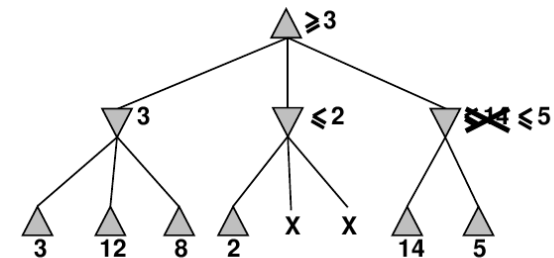
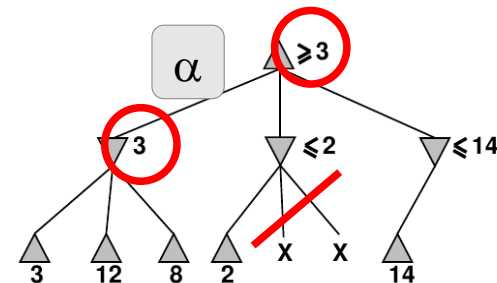
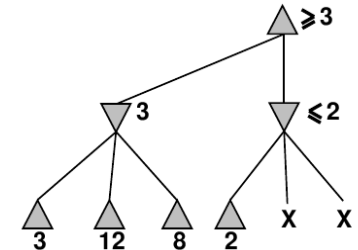
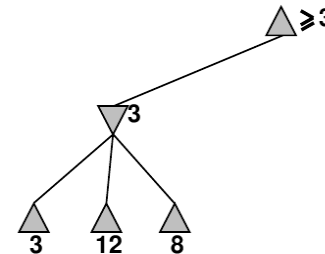
# \* Mehrspieler-Spiele

- Mehrspieler Spiele
  - Halma, Mensch-Ärgere-Dich-Nicht, ...
- Anpassung Minimax
  - Jeder Knoten im Suchbaum erhält Vektor mit einem Eintrag je Spieler
    - In einem Drei-Spieler Spiel mit Spielern A, B, C den Vektor  $(v_A, v_B, v_C)$
  - Für Endzustände repräsentiert jeder Wert den Wert des Ergebnisses für den jeweiligen Spieler
    - Nur ein Wert in Zwei-Spieler Spielen, Wert des anderen ist Gegenteil
  - Auswahl bei „perfektem“ Spieler durch Maximieren auf der Ebene, bei der er oder sie am Zug ist
- Besonderheiten Mehrspieler – Allianzen
  - Spiel mit drei Gegnern, einer ist überlegen
  - Allianzen sind sinnvoll. Schwächere haben nur eine Chance, wenn mit Allianz Stärkere besiegt werden
  - Passiert automatisch bei vollständig durchschauten Spiel



# \* Alpha-Beta-Kürzung

- Ziel
  - Teilbäume Knoten nicht bearbeiten
  - Gleiches Ergebnis wie Minimax
- Verfahren: Alpha-Beta-Kürzung
  - Berechne Untergrenze  $\alpha$  (max-Ebene)
  - Berechne Obergrenzen darunter (min)
  - Sobald Obergrenze kleinergleich Untergrenze: Abschneiden (cutoff)
  - Umgekehrt (min/max) mit  $\beta$
- Beispiel
  - Nach Evaluation von  $A_{1*}$  weiß Max, dass mindestens 3 erreicht werden kann
  - Wenn jetzt bei  $A_2$  und dann  $A_{21}$  bestenfalls 2 herauskommt, dann muss  $A_2$  nicht weiter expandiert werden
  - Bei  $A_3$  muss weiter evaluiert werden



# \* Alpha-Beta-Kürzung – Umsetzung

- Abwechselnd maximieren, dann minimieren
- alpha: maximieren
  - Check Ende, nur evaluieren
  - Für alle Nachfolger
    - Bewerte, Aufruf beta
    - Wenn cutoff, dann abbrechen; Wert egal, wird verworfen
    - Ansonsten kontinuierlich Maximum bestimmen
  - Falls durchlaufen, dann maximales Ergebnis zurück
- Effekt auf Laufzeit
  - Abhängig von Reihenfolge Expansion der Knoten
  - Im Idealfall (auf einer Ebene immer gleich richtig geraten) halber Verzweigungsgrad,  $O(b^{m/2})$

```
def alphabeta(start, expand, evaluate):  
    return alpha(start, expand, evaluate, None, None)
```

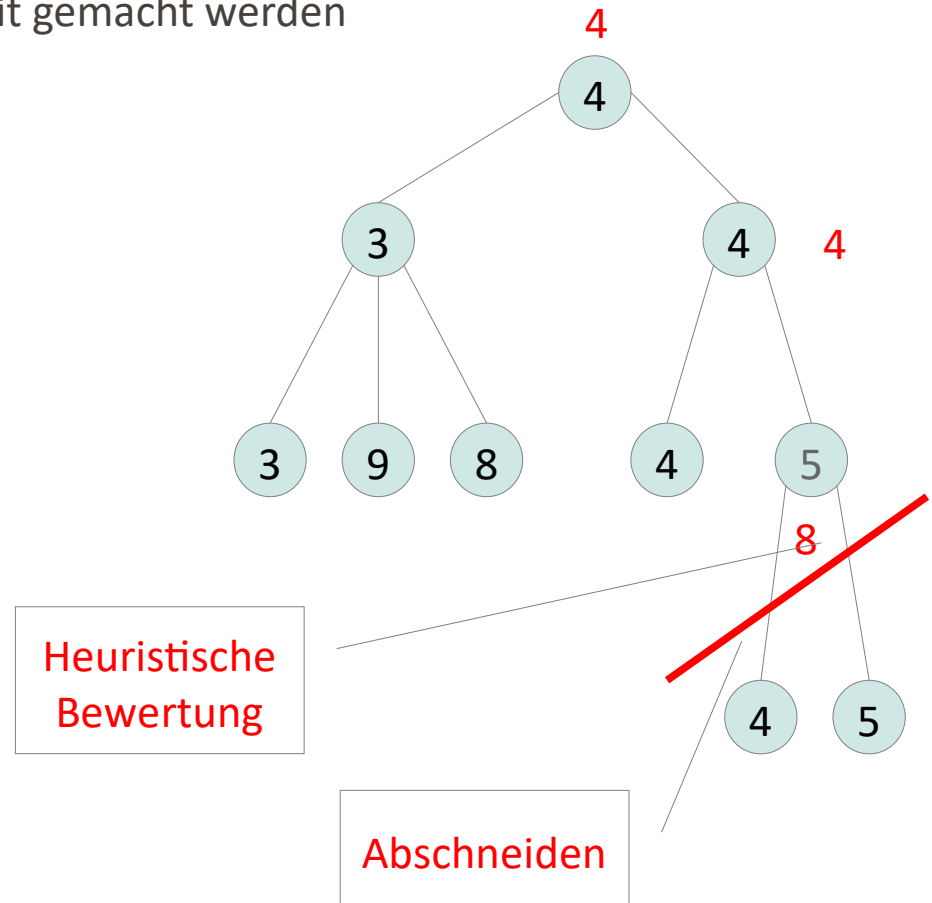
```
def alpha(s, expand, evaluate, a, b):  
    if evaluate(s) != None:  
        return (evaluate(s), s)  
    an = None  
    for n in expand(s):  
        v,_ = beta(n,expand,evaluate,a,b)  
        if b != None and v > b: # cutoff  
            return v, n # wird verworfen  
        if a == None or v > a:  
            a, an = v, n  
    return a, an
```

```
def beta(s, expand, evaluate, a, b):  
    if evaluate(s) != None:  
        return (evaluate(s), s)  
    bn = None  
    for n in expand(s):  
        v,_ = alpha(n,expand,evaluate,a,b)  
        if a != None and v < a: # cutoff  
            return v, n # wird verworfen  
        if b == None or v < b:  
            b, bn = v, n  
    return b, bn
```



# \* Einschränkungen Suchbaum

- Suchraum meist zu groß
  - Auch mit Alpha-Beta-Reduktion
  - Meist feste Zeitvorgabe, die nicht überschritten werden, ein Zug *muss* nach einer gewissen Zeit gemacht werden
- Suchraum beschränken
  - Kein Durchschauen bis zum Ende, keine perfekte Evaluation
  - Bewertungsfunktion statt perfekte Evaluation
    - Berechnen einer relativen Zahl, die Spielzustand aus eigener Sicht bewertet
    - Meist Summe von einzelnen Features
    - Heuristik, wie bei informierter Suche
  - Abschneiden (Cutoff) von Zweigen
    - Anhand von Heuristiken
    - z.B. schlechtester Wert bisher
  - Zustandsbibliotheken
- Verzicht auf richtigen Zug



# \* Bewertungsfunktion

- Bewertung eines Zustands
  - Unsicherheit aufgrund beschränkter Systemressourcen
  - Bewertungsfunktion meist gewichtete Summe von zu extrahierenden Bewertungen von Einzelfeatures
- Beispiel: Schach
  - Bewertung anhand der Anzahl und Wertigkeit (Punkte) der Figuren

Alternativer  
Ansatz  
Dame,  
maschinelles  
Lernen



1



3



5

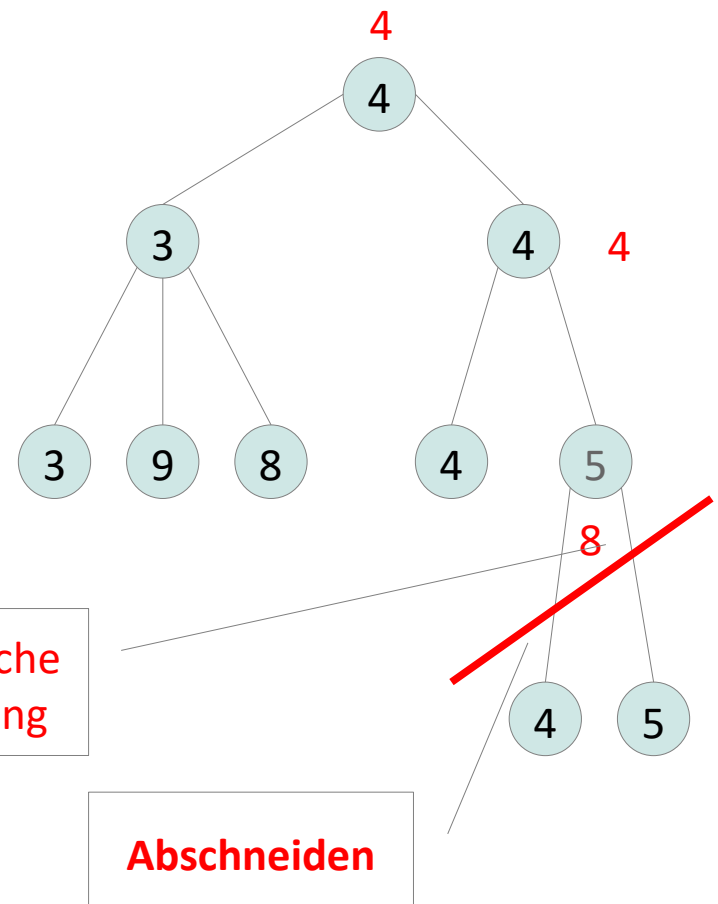


9

- Gewichtung anhand Erfahrung beim Schachspiel
- Gute Gewinnchancen, bei >1 Vorteil, fast sicherer Sieg bei >3 Vorteil
- Andere „Features“ können in (Milli-)Punkte umgerechnet werden
  - „Gute Bauernstruktur“, „Sicherheit des Königs“, ...
- Achtung – nur eine Heuristik
  - Sinnlos inmitten einer festen Austausch-Zugfolge
  - Summieren bedeutet, dass die Features unabhängig voneinander sind (?)
  - Gewichtung über Spieldauer gleich, aber z.B. Springerpaar im Endspiel mehr als doppelt so viel wert wie ein einzelner Springer

# \* Suchbaum beschneiden

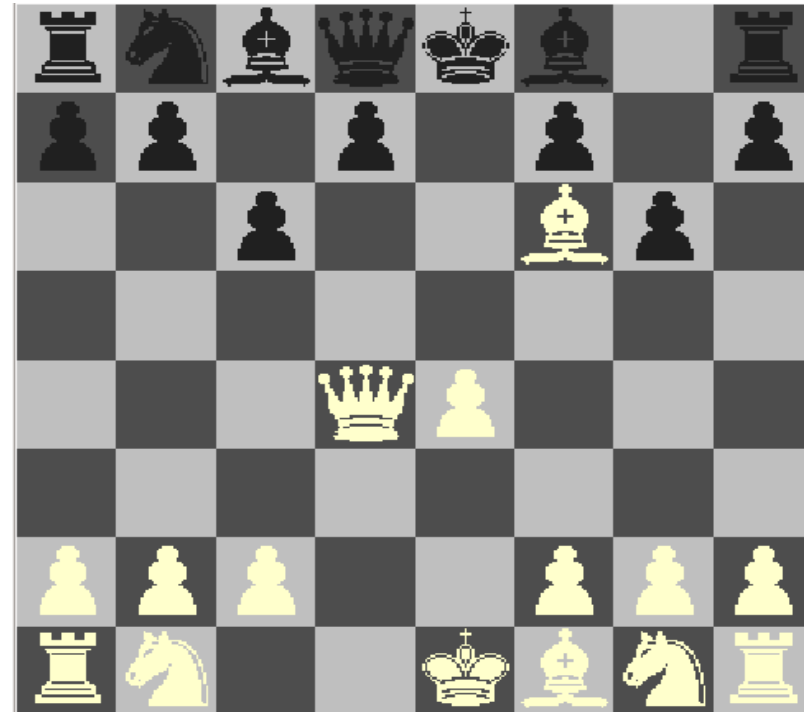
- Abbrechen der Tiefensuche
  - Statt Test auf Endzustand Test auf abbrechen/beenden
  - Bei Endzuständen wird natürlich auch abgebrochen
- Verschiedene Ansätze
  - Feste maximale Tiefe
  - Iterative Tiefensuche mit Zeitlimit
- Verbesserungen
  - Nur in Ruhephasen im Spiel (nicht mitten in einer Zugtauschfolge, die typischerweise tief aber nicht breit sind) abbrechen
  - Horizonteffekte vermeiden (Zustände, bei denen ein schlechter Zug unausweichlich ist, aber die Suche präferiert diese Entscheidung hinter den Suchhorizont zu schieben)
  - „Hoffnungslose“ oder symmetrische Züge nicht beachten



# \* Beispiel – Schach

- Bewertung

- Voller Satz Figuren ist  
 $8 + 2 \cdot 5 + 4 \cdot 3 + 9 = 39$
- Schwarz:  
Es fehlt Bauer und Springer,  
35
- Weiss:  
Es fehlt Bauer,  
38
- Und es sieht nicht so  
gut aus für Schwarz ...

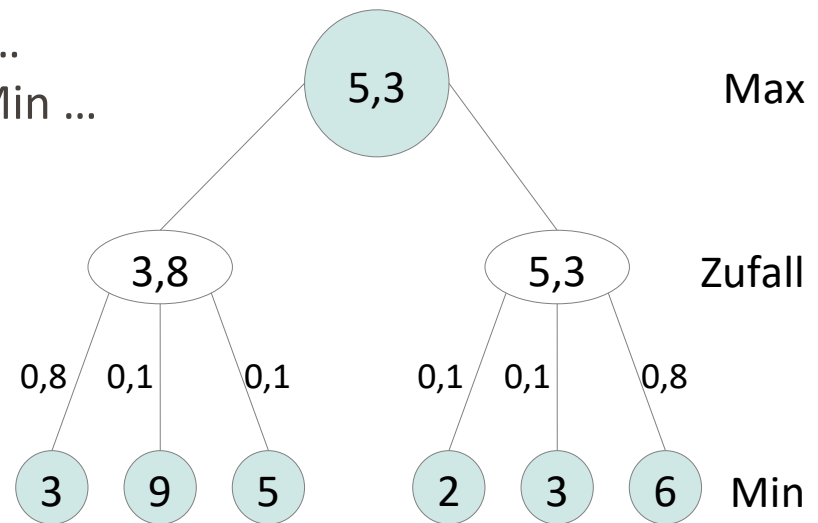


- Schachprogramme

- ca. 1 Million Knoten pro Sekunde, ca. 200 Millionen Knoten pro Zug
- Statt Minimax (ca. 6 Halbzüge) mit Alpha-Beta-Kürzung ca. 10 Halbzüge
- Viel Engineering und Tuning
  - Aktuell ca. 14 Halbzüge
  - Umfangreiche Eröffnungs- und Endspielbibliotheken

# \* Zufallselemente – Expectiminimax

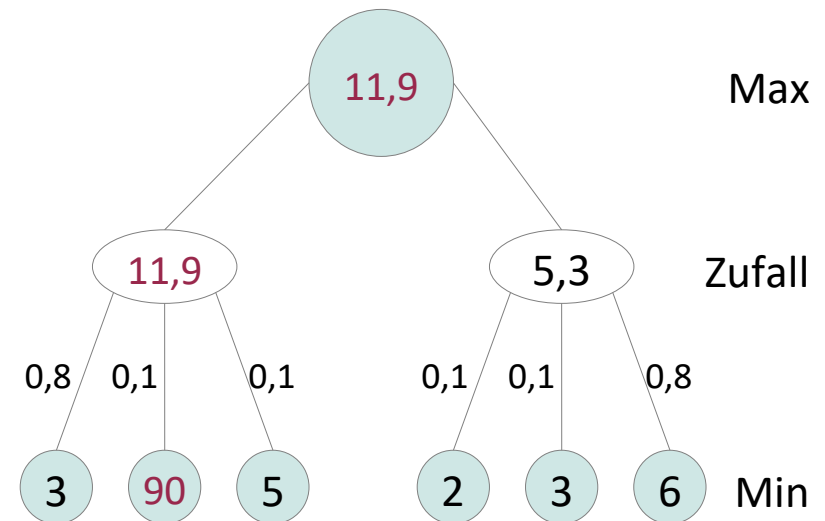
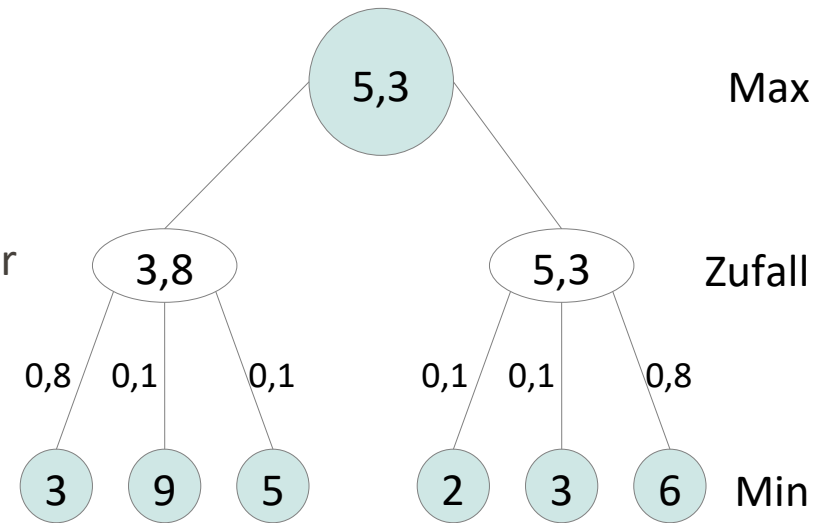
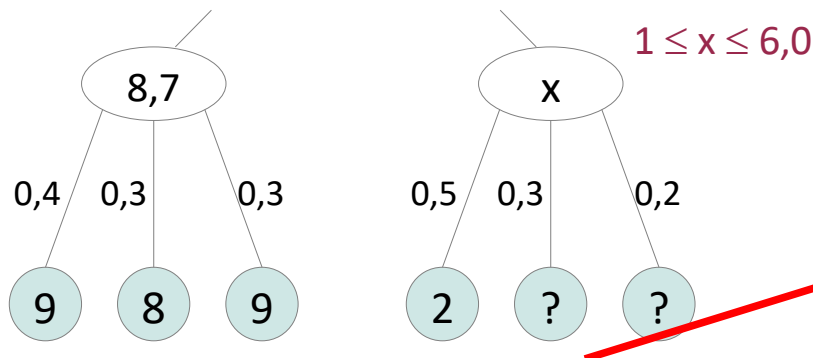
- Spiele mit Zufallselementen
  - Zum Beispiel Würfelspiele wie Backgammon
  - Nicht mehr deterministisch
  - Die Anzahl und Art der zulässigen Züge wird durch Zufall bestimmt
- Einführung von Zufallsknoten
  - Statt Max gefolgt von Min, Max, Min ...  
Max, Zufall, Min, Zufall, Max, Zufall, Min ...
  - Ein Zufallsknoten auf der Zufallsebene für jedes mögliche Ergebnis
  - Mit Einzelwahrscheinlichkeiten gewichteter Gesamtwert der Zufallsebene (Erwartungswert)
  - Verfahren: Expectiminimax



# \* Zufallselemente – Suchraumbegrenzung

## • Techniken Suchraumbegrenzung

- Vorhandene Techniken mit Anpassung einsetzbar
- Achtung: Bewertungsfunktion muss linear dem tatsächlichen relativen Wert des Zustands entsprechen
  - Bei Minimax war es reine Entscheidungsfunktion (egal ob 1, 2 oder 1, 10)
- Alpha-Beta-Kürzung:
  - Nicht einfach möglich, keine „wahrscheinliche“ Zugfolge
  - Einführung von Intervallen (Werte  $\leq 10$ )



# \* Unbekannte Informationen

- Unbekannte Information
  - Nicht alle relevanten Informationen sind bekannt
  - Kann behandelt werden wie zufällige Information
  - Beispiel: Kartenspiele (Skat, ...)
    - Wissen/Wahrscheinlichkeiten über Karten der Gegner ändert sich.  
Es wird eine Farbe nicht bekannt, ein Stich nicht gemacht, ...
  - Informationen, die bekannt sind zur Entscheidungsfindung nutzen
  - Expectiminimax mit gewichtetem Mittelwert aller möglichen Situationen
- Beispiel: Wegekreuzung/Entscheidungsbaum
  - Max will Gold nicht Geld aber nicht sterben
  - Max kann sich an jedem Knoten entscheiden ob links oder rechts
  - Wenn Baum bekannt, dann dreimal rechts
  - Wenn Baum nur bekannt bis Level von Knoten C, dann?
    - Normale Menschen gehen links und nehmen Geld

