

Problem - lösen mittels Suche — Suchalgorithmen

(Kap. 3 in RN)

Wir betrachten Algorithmen die einen Suchbaum über dem Zustandsraum (state space) aufbauen.

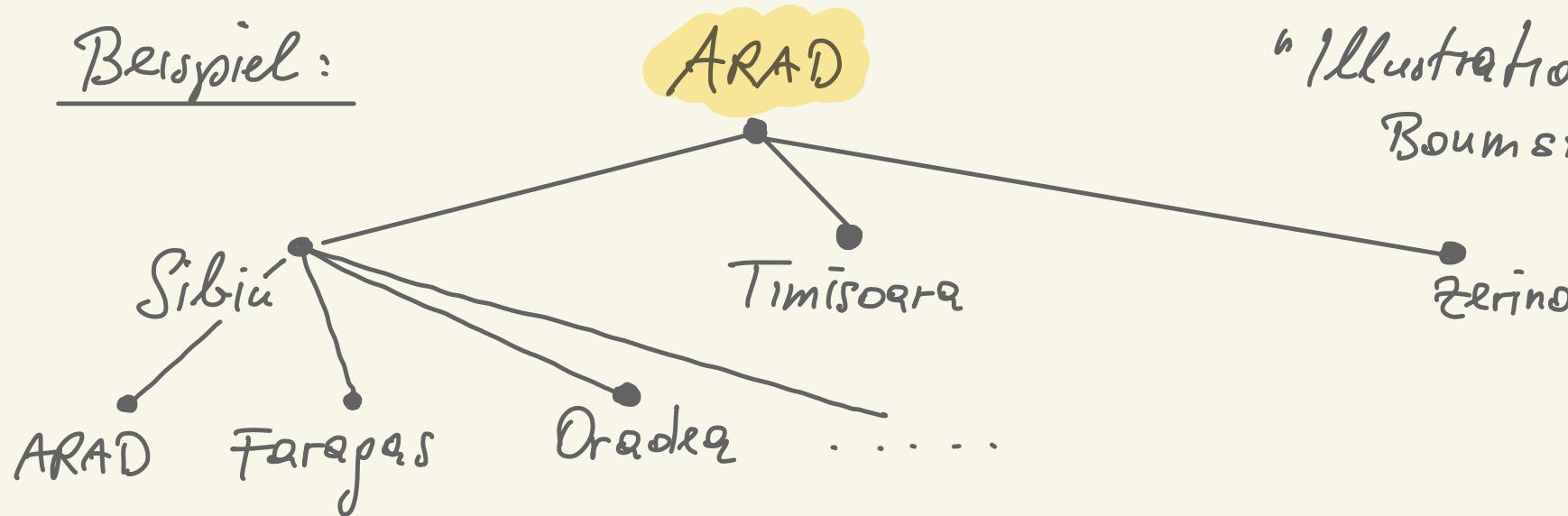
wir versuchen einen Pfad in diesem Baum zu finden der, ausgehend von einem Startknoten unser Ziel erreicht.

Knoten im Baum = Zustand im Zustandsraum

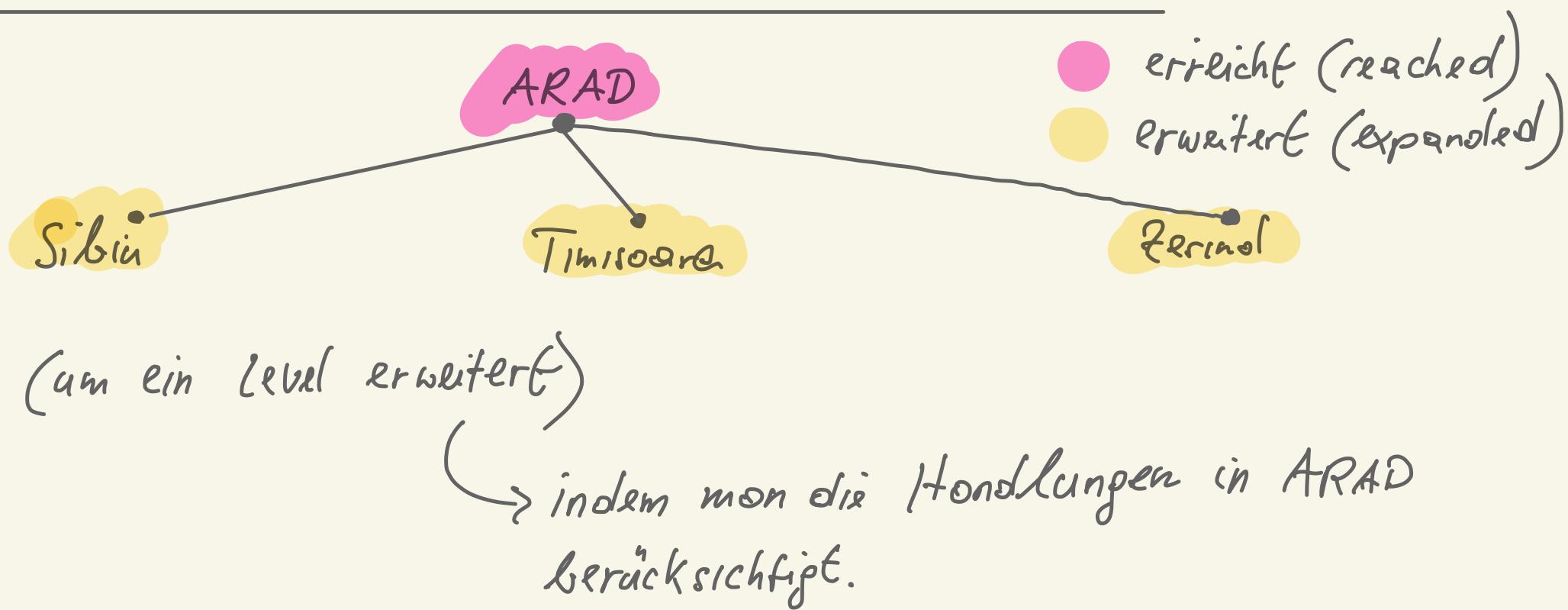
Kanten im Baum = Handlungen (actions)

Der Wurzelknoten (root node) im Baum ist der Initialzustand!

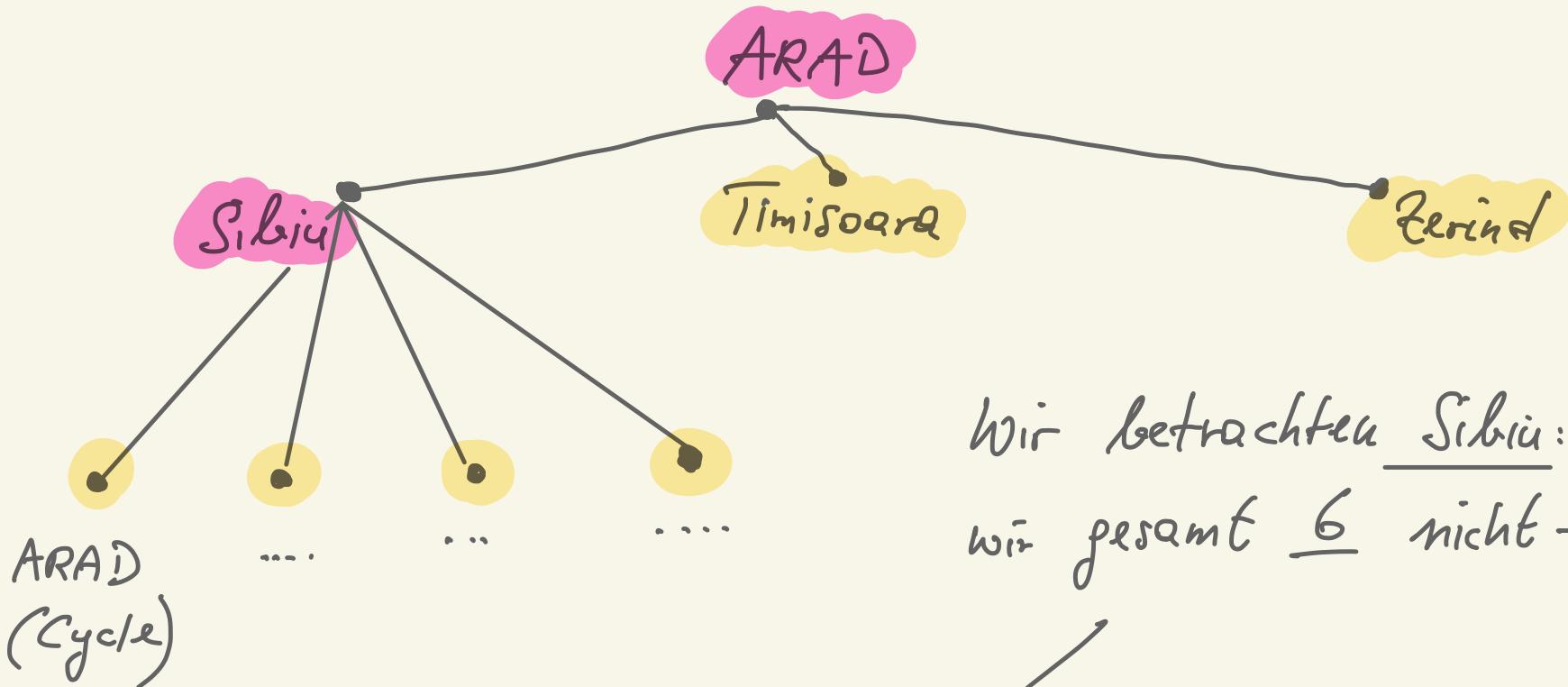
Beispiel:



"Illustration
Boumstruktur"



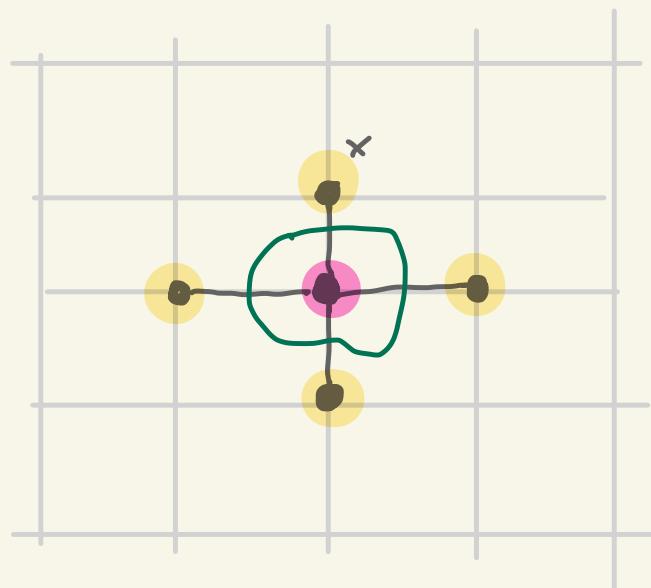
Jeder Kind-knoten (child-node) $\{Sibiu, Timisoara, Zerind\}$ hat ARAD als Eltern-knoten (parent-node).



wir betrachten Sibiu: hier haben wir gesamt 6 nicht-erweiterte Knoten.

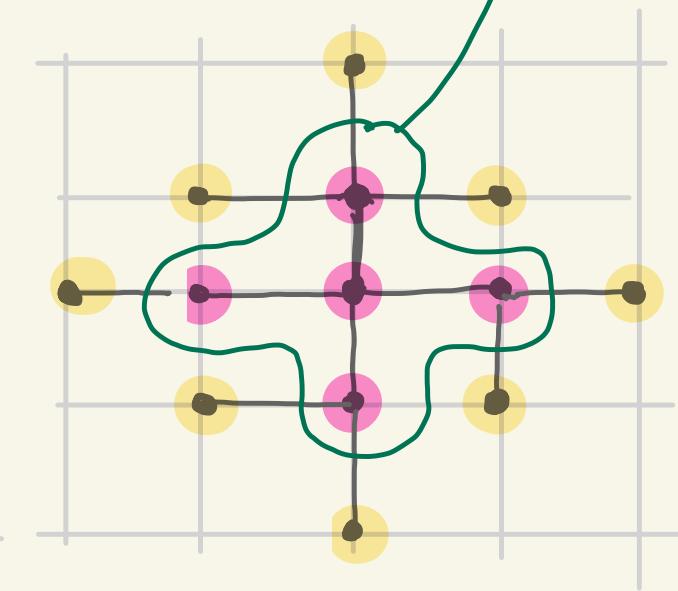
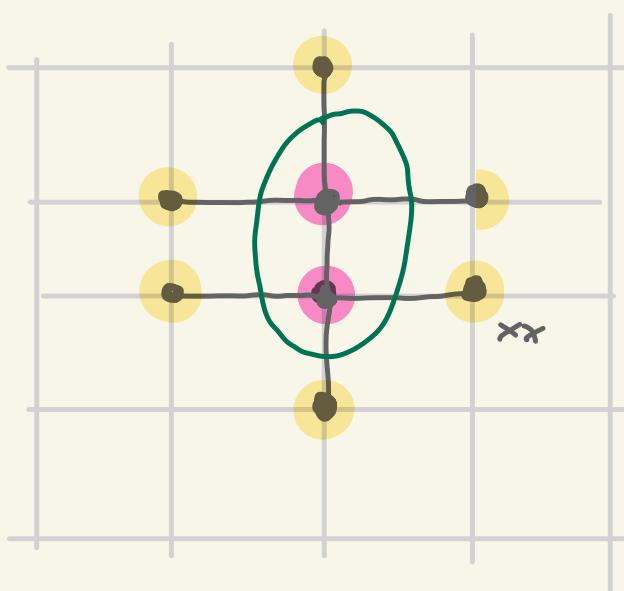
wir nennen diese Knoten die **Front** (frontier).

Beispiel (cinterior vs. frontier)



FRONT

obersten
Knoten erweitert



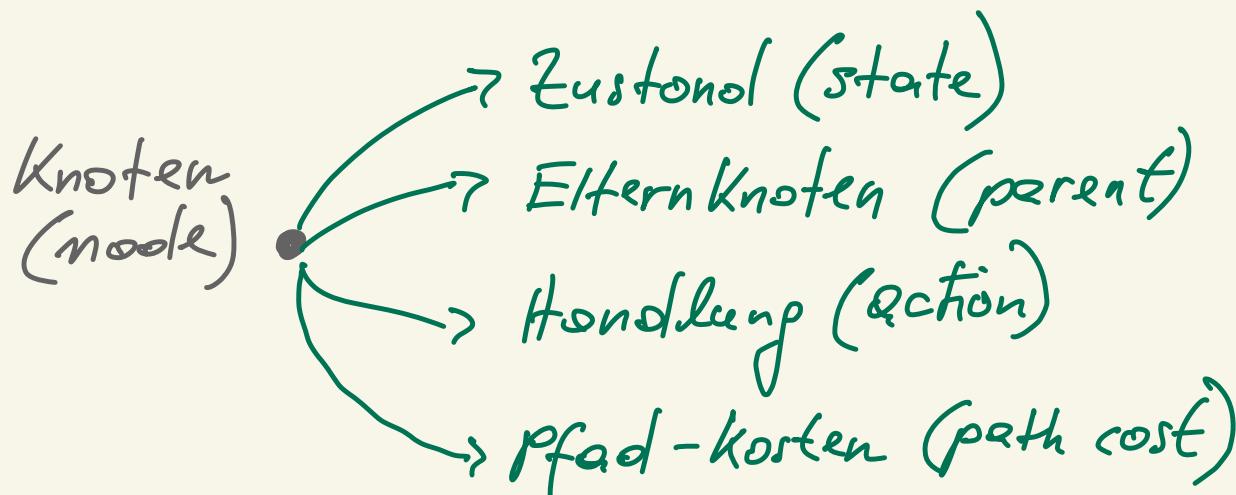
erweitern den "Rest"
(Uhrzeigersonn, startend von xx)

Best-First Search

Wir wählen Knoten n mit minimalem Wert einer Evaluierungsfunction $f(n)$. Also, in jedem Schritt wählen wir einen Knoten der Front mit minimalem $f(n)$, überprüfen ob dieser bereits unter Ziel ist, oder erweitern den Knoten. "Erweitern" generiert Kind-knoten die wir zur Front hinzufügen (sofern nicht schon erreicht, z.B.: ARA^D in unserem Beispiel; kann auch neu hinzugefügt werden, sofern die Pfadkosten geringer sind).

Je nach $f(n) \rightarrow$ anderen Suchalgorithmen

Was benötigen wir?



Um die Front (frontier) zu speichern, benötigen wir eine Art von Warteschlange (queue). Wir müssen ...

- ▷ überprüfen können ob die Warteschlange leer ist (IS-EMPTY)
- ▷ den "obersten" Knoten entfernen und zurückgeben können (POP)
- ▷ den "obersten" Knoten zurückgeben können (TOP)
- ▷ einen Knoten zur Warteschlange hinzufügen können (ADD)

Redundante Pfade:

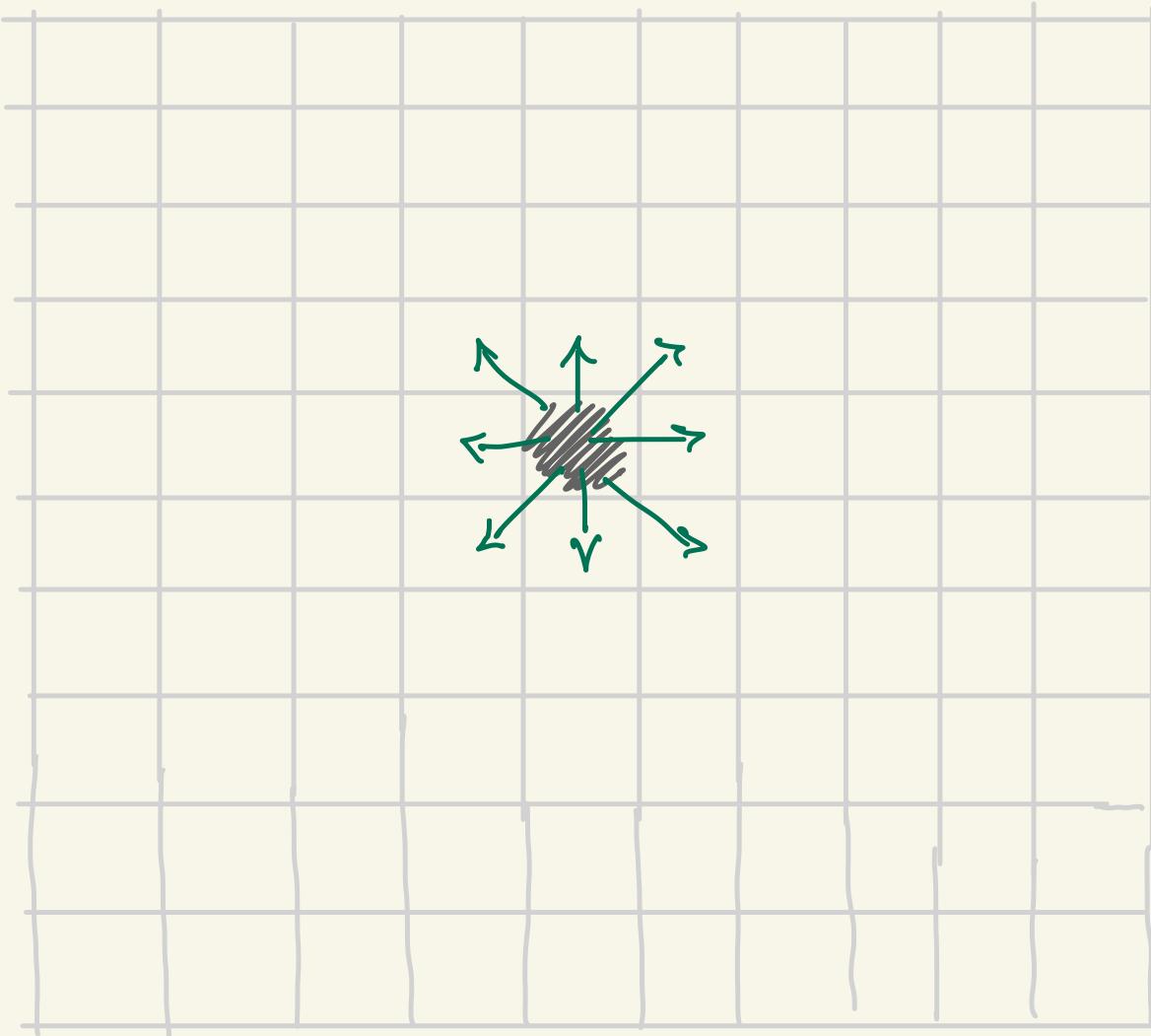
z.B.: ARAD → SIBIU

ARAD → ZERIND → ORADEA → SIBIU

Suchalgorithmen die redundante Pfade berücksichtigen nennt man
Graph - Suchalgorithmen; falls nicht auf redundante Pfade geprüft
wird, nennen wir die Baum - ähnliche Suche (graph-based search vs.
tree - search).

Bsp.:

→
erlaubter Zug



man erreicht alle
Zellen in 9 Zügen
ool. weniger!

Anzahl aller Pfade der Länge $9^2 \approx 8^9$, da 8 mögliche Nachboren
ca. 130 Millionen!
→ da weniger Züge möglich an Grenzen!

Eine durchschnittliche Zelle kann durch ca. 1.3 Mil. $\frac{\text{~} \approx 130 \text{ M}}{10 \cdot 10}$
Pfade oder Länge ϱ erreicht werden.

ϱ

Wie können wir die Performance von Suchalgorithmen bewerten?

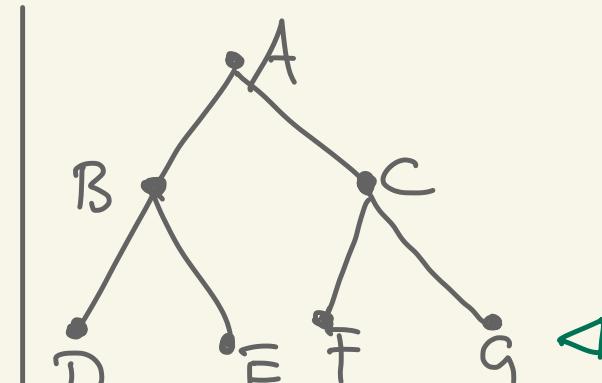
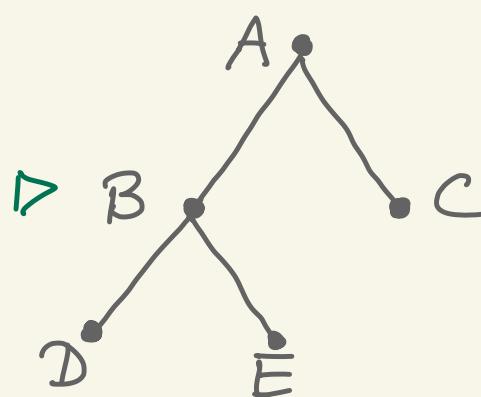
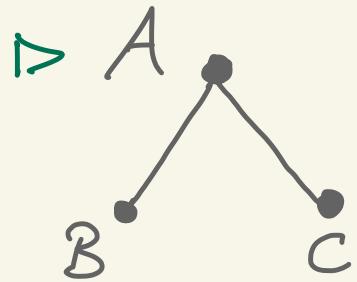
- ▷ **Vollständigkeit (completeness)**: findet man eine Lösung wenn es eine gibt, bzw. meldet der Algorithmus einen Fehler wenn es keine gibt.
- ▷ **Kosten - Optimalität (cost-optimal)**: findet man eine Lösung mit minimalen Kosten unter allen Lösungen.
- ▷ **Zeit - Komplexität (time-complexity)**: Anzahl der betrachteten Zustände und Aktionen.
- ▷ **Platz - Komplexität (space-complexity)**: wieviel Speicher benötigen wir?

Beispiel : Breadth-First-Search (Breitensuche) BFS

$$f(n) = \text{Tiefe von Knoten } n$$

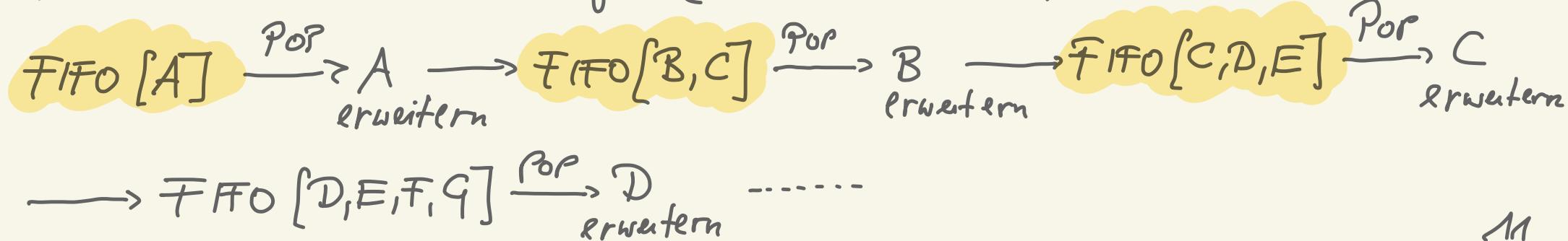
↳ # Handlungen um Knoten n zu erreichen!

Brauchbar wenn Handlungen gleiche Kosten haben.



.....

FRONT.... FIFO Warteschlange (First-In First Out)



BFS ist kosten-optimal für Probleme wo alle Handlungen die gleichen Kosten haben.

ABER, sehr Speicherplatz-Intensiv!

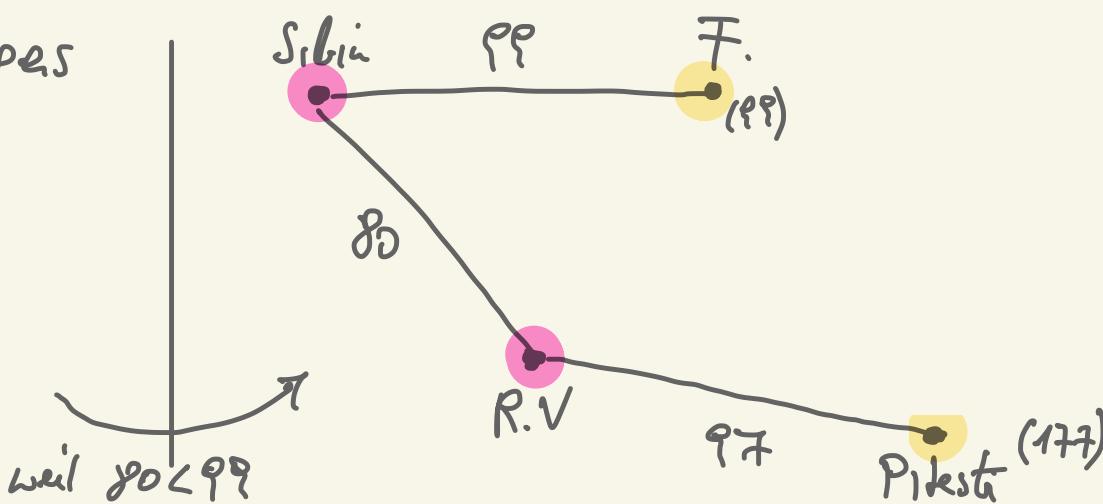
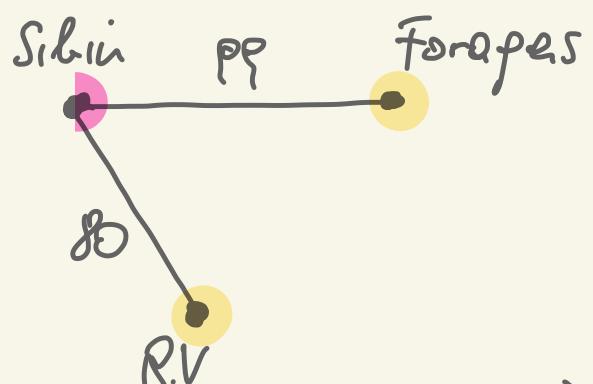
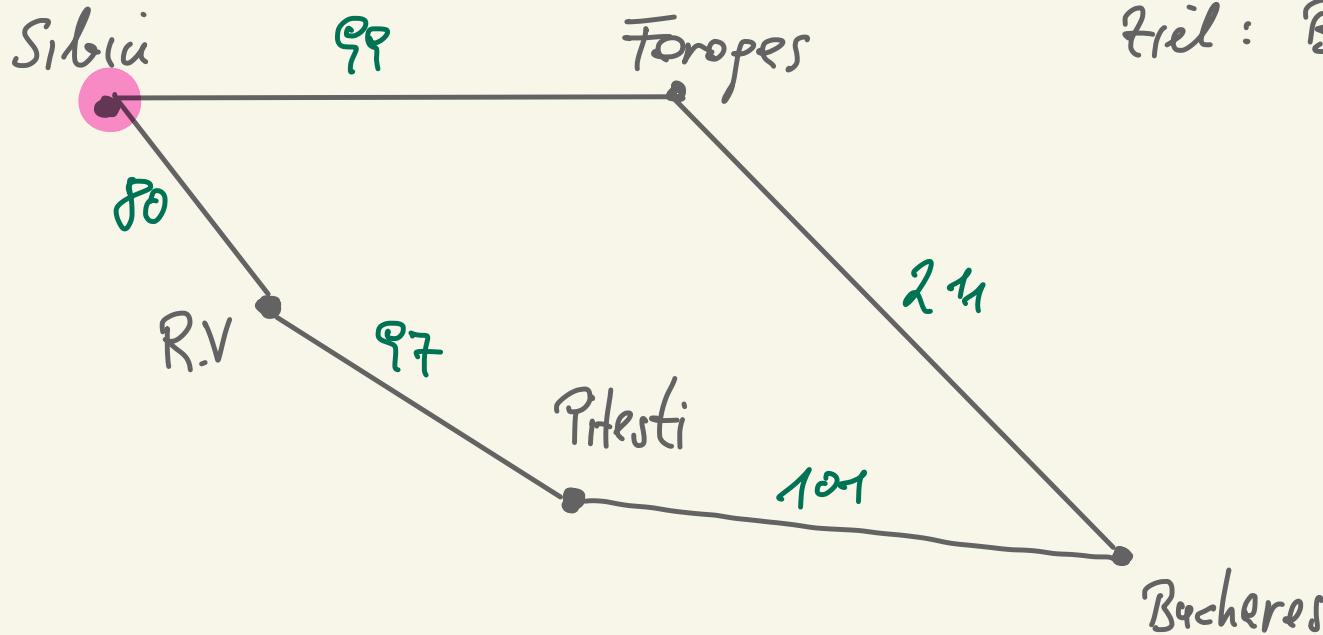
Bsp.: generiert z.B. jeder Knoten 10 Kind Knoten (child nodes),
haben wir bei Tiefe d

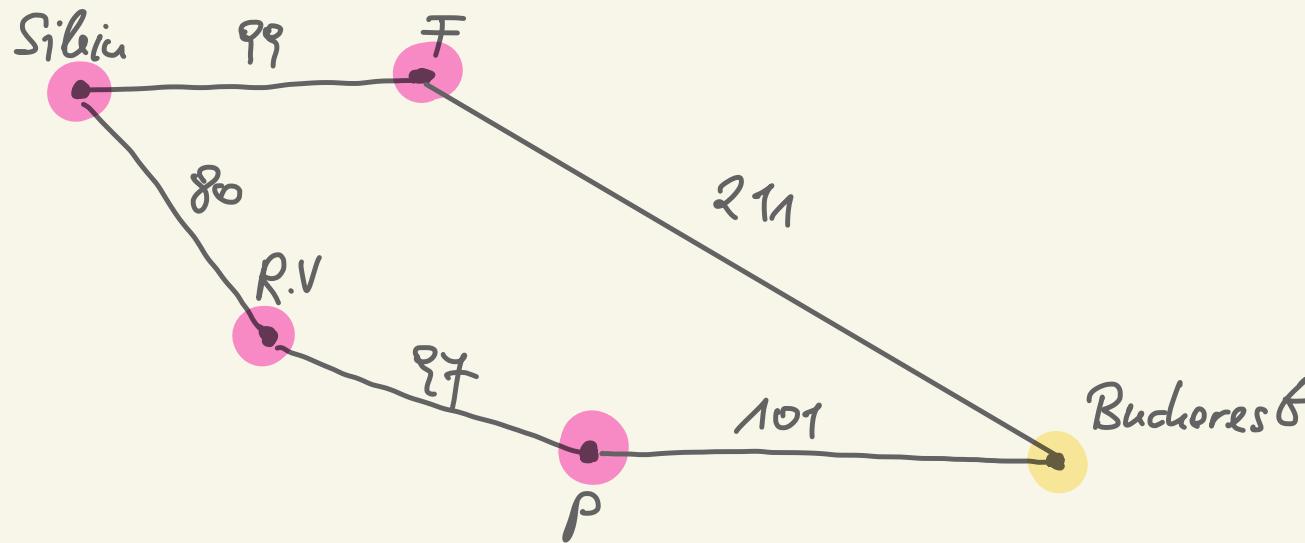
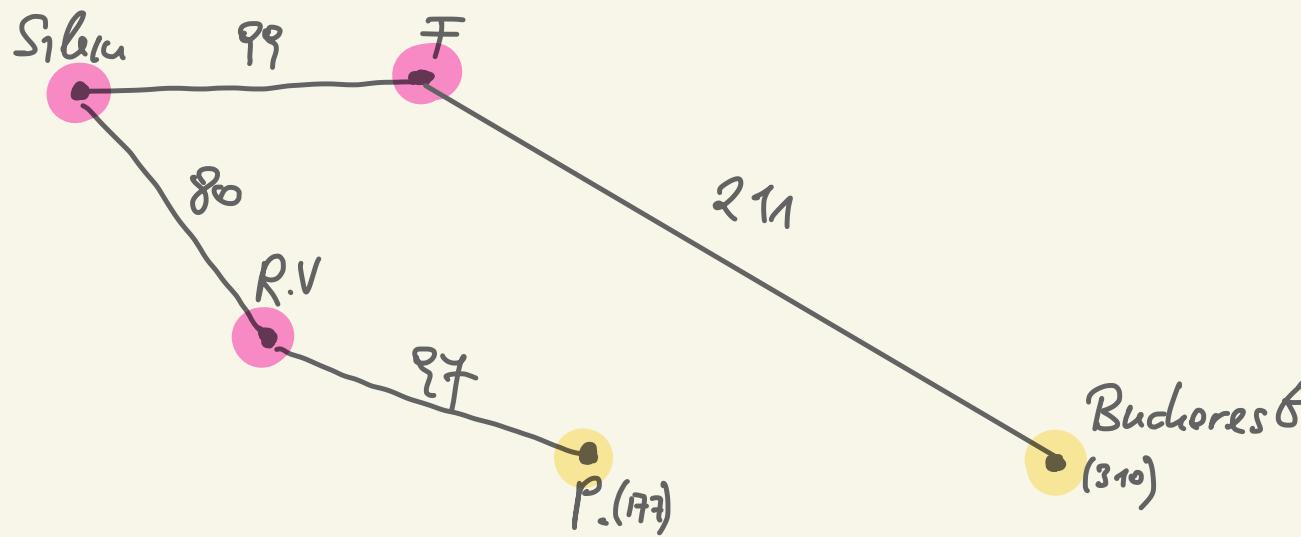
$$1 + b + b^2 + b^3 + \dots + b^d$$

knoten. Setzen wir $b=16$! Bei $\underbrace{d=10}_{\text{Tiefe}}$: 16^{10} Knoten!!!

Weiteres Beispiel (Dijkstra Algorithmus)

$f(n)$ = Kosten von der Wurzel (root node) zu Knoten n!





Jetzt würde Bucharest erweitert werden (aktueller Pfad mit geringsten Kosten ist $S \rightarrow R.V. \rightarrow P \rightarrow B$ mit $80 + 97 + 101 = 278!$). Wir überprüfen ob Bucharest unser Ziel ist → Algorithmus terminiert!

Alle bisher besprochenen Such-Algorithmen sind Instanzen sogenannter **uninformierter** Such-Algorithmen; also jene wo nicht bekannt ist wie weit ein Zustand vom Ziel entfernt ist. Hat man Heuristiken über die "Distanz" zum Ziel, nennt man solche Such-Algorithmen **informierte** Such-Algorithmen.

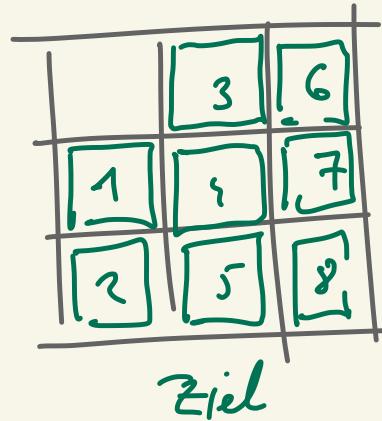
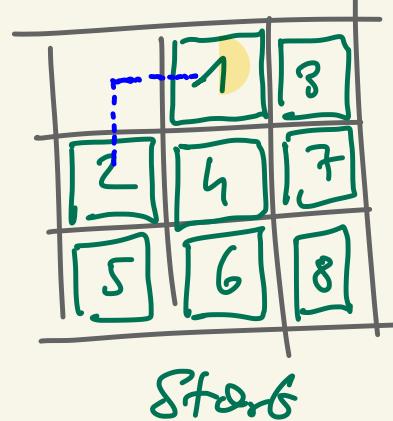
z.B.: A* Such-Algorithmus

Geschätzte Kosten des günstigsten Pfades von n zum Ziel!

$$f(n) = \underbrace{g(n)}_{\text{Pfad-Kosten von Start-Knoten zu Knoten } n} + \underbrace{h(n)}_{\text{Geschätzte Kosten des günstigsten Pfades von }} \\$$

- h muss eine Zulässigkeitsbedingung erfüllen, nämlich die Kosten von n zum Ziel nicht zu überschätzen.

8-Puzzle:



Heuristik: Manhattan-Distanz
jeder Kachel zur Zielposition!

Z.B.: $2 + 1 + 1 + 0 + 1 + 3 + 0 + 0$

Suche in Komplexen Umgebungen

(Zuvor: voll beobachtbar, deterministisch, statisch, bekannt, ...)

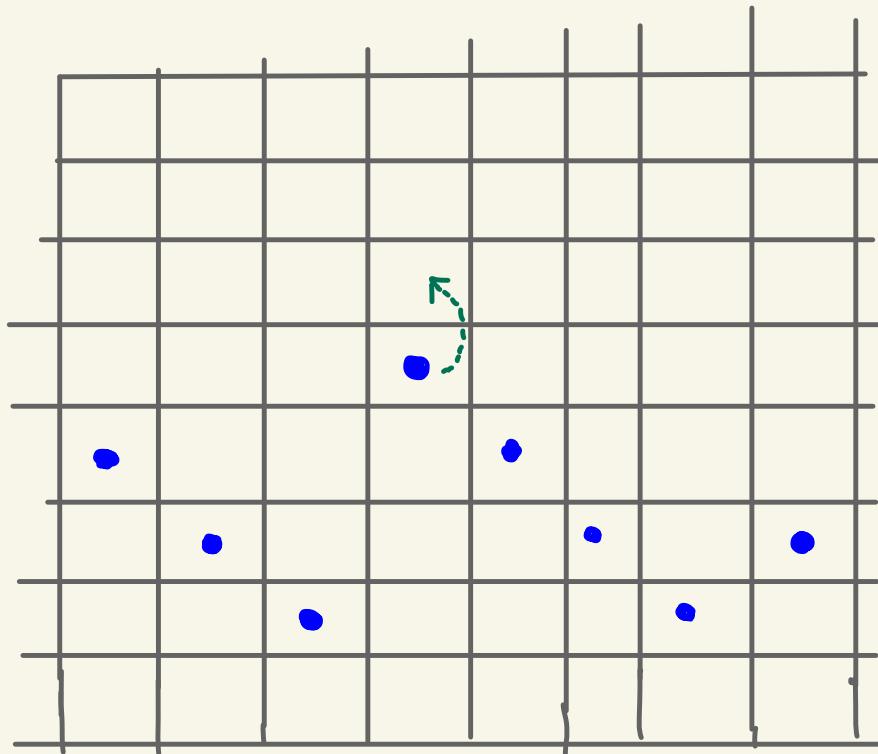
Optimierungsprobleme und lokale Suche

Manchmal interessiert uns nur der "finale" Zustand, nicht der Pfad!

Lokale Suche (Local Search) funktioniert indem man, ausgehend von einem initialen Start-Zustand, nur in den Nachbarzuständen sucht (ohne sich den Pfad zu merken od. auch welche Zustände man schon erreicht hat).

Beispiel : 8-puzzles

• Königshinzen



Aktuelle Kosten

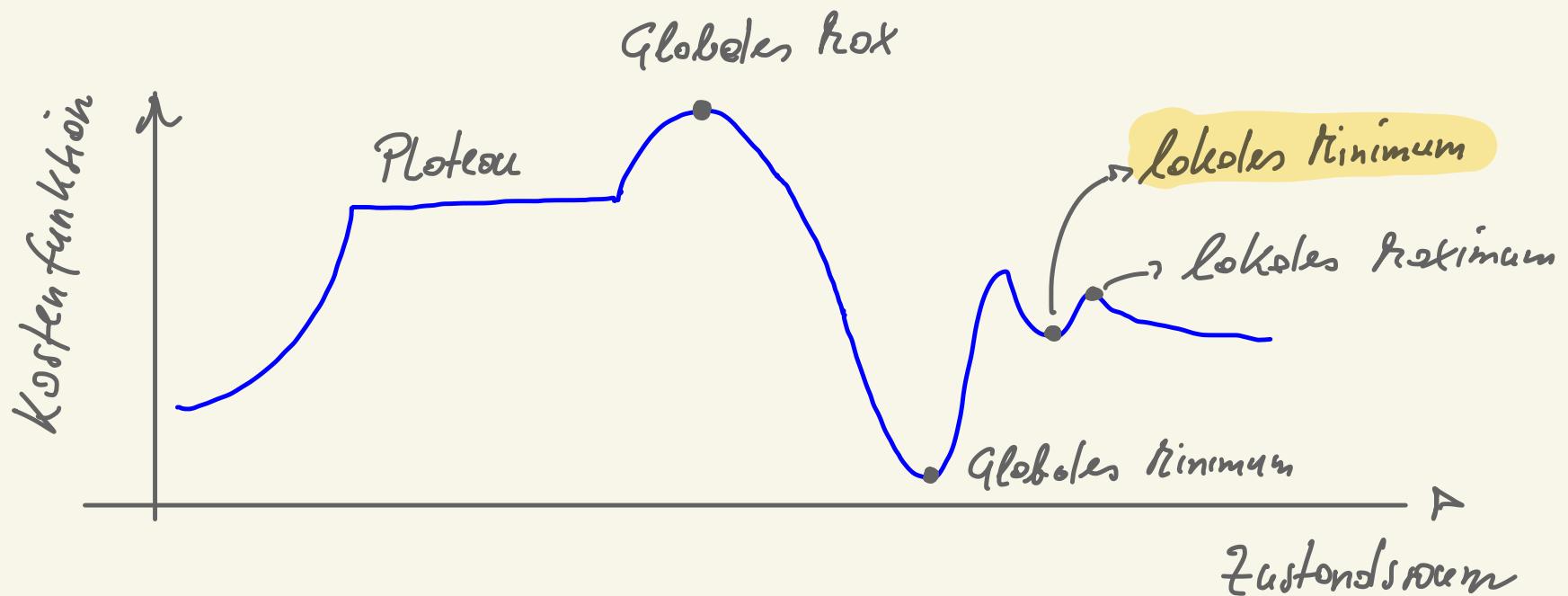
(17)

Kosten = # Paaren von • die sich diagonal, als auch über Spalten u. Zeilen hinweg angreifen (sich wenn eine Figur dazwischen ist!)

18

Vorteile: geringe Speicherbedarf & man findet oft brauchbare Lösungen (auch in unendlichen Zustandsräumen).

Nachteil: viel. durchsucht man nie Teile des Zustandsraums wo eine Lösung (Teilzustand) liegt.



Hill - Climbing

Man merkt sich den aktuellen Zustand und geht zu jenem Nachbarzustand mit dem höchsten Wert einer Kostenfunktion; iteratives Verfahren welches terminiert wenn es keinen Nachbarzustand mehr mit höheren Kosten gibt.

(Probleme sind lokale Maxima od. Plateau's, siehe Zeichnung)

Mögliche Erweiterungen:

- Stochastic Hill Climbing: zufällig aus möglichen Zustandswechseln auswählen;
- Random - Restart Hill Climbing: mehrmals von zufälligen Initialzuständen starten (und jene Lösung mit max. Kosten auswählen).

Simulated Annealing

„Ähnlich wie Hill-Climbing, aber anstatt den LOKAL BESTEN Zustandswechsel zu machen, wählt man zufällig nach folgendem Prinzip: ist der neue Zustand „besser“ (gerüptere Kosten), wird dieser akzeptiert; ist dies nicht so, wird er mit Wahrscheinlichkeit < 1 akzeptiert; Diese Wahrscheinlichkeit sinkt exponentiell mit ΔE , d.h. mit der Differenz (ΔE) um die sich der Zustand verschlechtert. Zusätzlich: Temperatur Parameter T , welcher anfänglich hoch ist und sich im Laufe des Simulated Annealing verringert.

$$e^{\frac{\Delta E}{T}}$$

Bsp.: $x = \text{aktueller Zustand}$

$y \in \text{Umpembung}(x)$

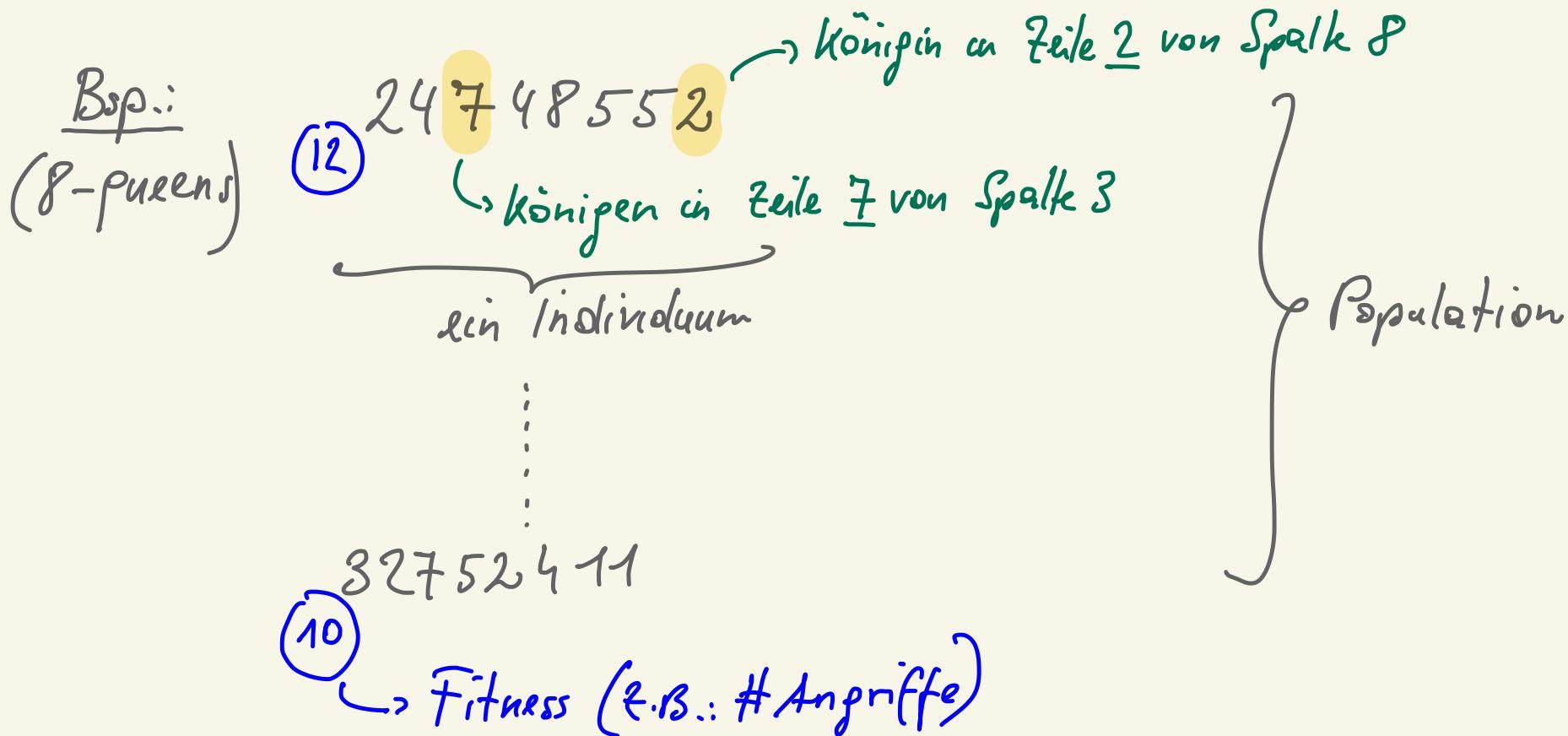
$\text{Kosten}(y) \leq \text{Kosten}(x) \Rightarrow \text{Zustandswechsel akzeptiert}$

$\text{Kosten}(y) > \text{Kosten}(x) \Rightarrow \text{Akzeptiert mit}$
 $\text{Wahrscheinlichkeit}$

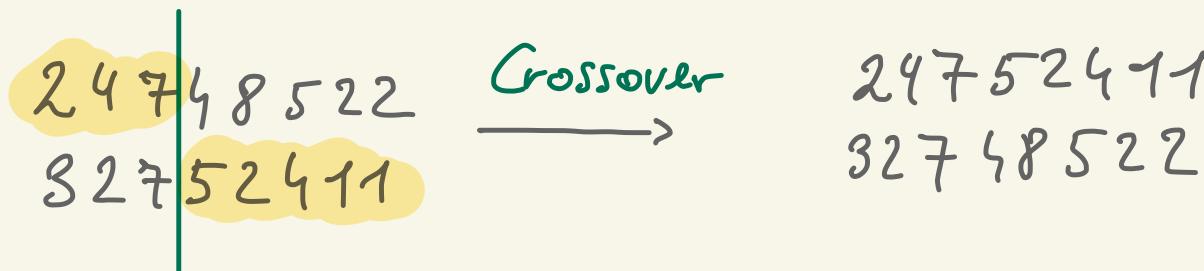
$$e^{-(\text{Kosten}(y) - \text{Kosten}(x)) / T}$$

Evolutionäre Algorithmen

Grundprinzip: Population von Individuen (also Zuständen); aus dieser Population generieren die "fittesten" Nachkommen (also neue Zustände), usw.

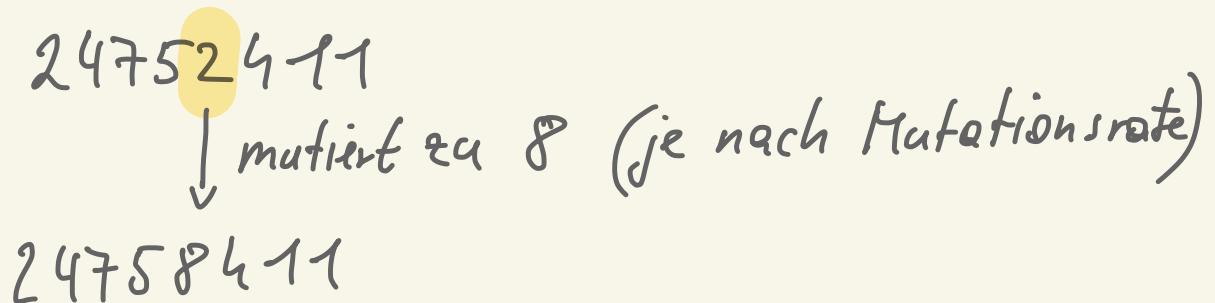


Als Beispiel: Selektion der 2 Fittesten



↳ wird z.B. zufällig ausgewählt (Rekombinationspunkt)

Weiters gibt es die Möglichkeit zur "Mutation":



Wichtige Parameter:

■ Größe der Population

■ Repräsentation der Individuen

- String über endlichem Alphabet \Rightarrow genetic algorithms
(wie in anderem Beispiel)

- Sequenz reeller Zahlen \Rightarrow evolution strategies

- als Computer Programm \Rightarrow genetic programming

- Selektionsprozess (proportional zum Fitness Wert)

- Rekombinationspunkt (für Crossover Operation)

- Mutationsrate

- Generierung der neuen Population (z.B.: 50% Fitteste Eltern + Nachkommen, etc.)

Suche bei nicht-deterministischen Handlungen

In partiell-beobachtbaren Umgebungen, ist es dem Agenten nicht genau bekannt in welchem Zustand sich die Umgebung befindet.

Ist die Umgebung nicht-deterministisch, ist es dem Agenten nicht genau bekannt in welchem Zustand sich die Umgebung nach einer Handlung befindet. Jene Zustände die vom Agenten als möglich angesehen werden nennt man **belief-states**.

In solchen Situationen ist eine Lösung nicht mehr eine reine Sequenz von Handlungen, sondern ein **Alternativplan (contingency plan)**, od. auch **Strategie**.

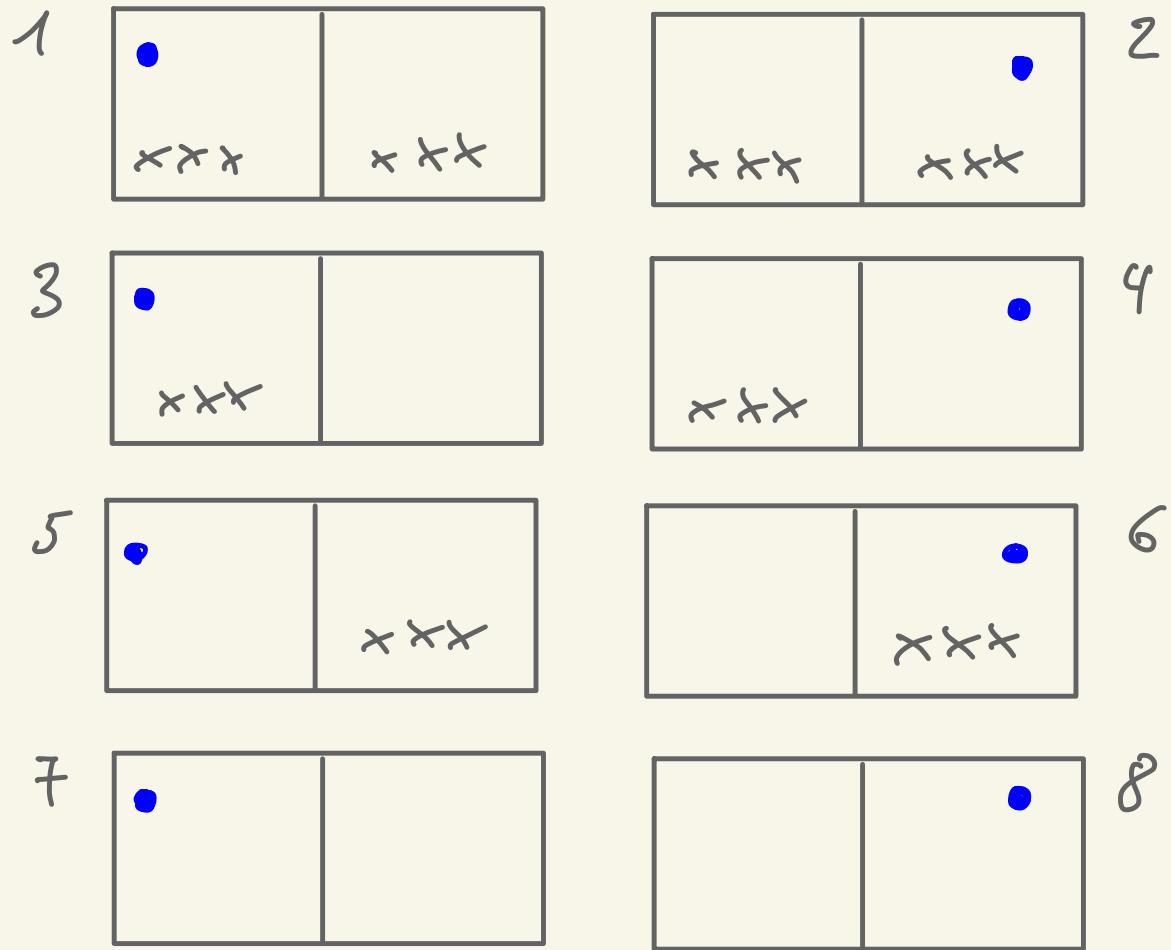
Bsp. (erweiterte Staubsaugerwelt)

Handlungen: 'Saugen', 'Links', 'Rechts'
 (Actions)

- ist eine Kachel schmutzig, reinigt die Handlung 'Saugen' die Kachel, aber auch möglicherweise die Kachel daneben
- ist die Kachel sauber, führt die Handlung 'Saugen' möglicherweise zu einer schmutzigen Kachel

⇒ Übergangsmodell nun nicht mehr durch eine RESULT Funktion gegeben (z.B.: $\text{RESULT}(1, \text{'Saugen'}) = 5$, siehe Grafik), sondern durch eine RESULTS Funktion der Form

$$\text{RESULTS}(1, \text{'Saugen'}) = \{5, 7\}$$

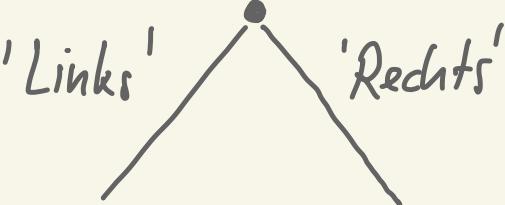


Sind wir in Zustand 1, führt keine einzelne Sequence an Handlungen zu einer Lösung, aber der Plan

\Leftrightarrow Lösungen sind Bäume anstatt Sequenzen)

Eine mögliche Variante: AND-OR-TREES

1) OR Knoten

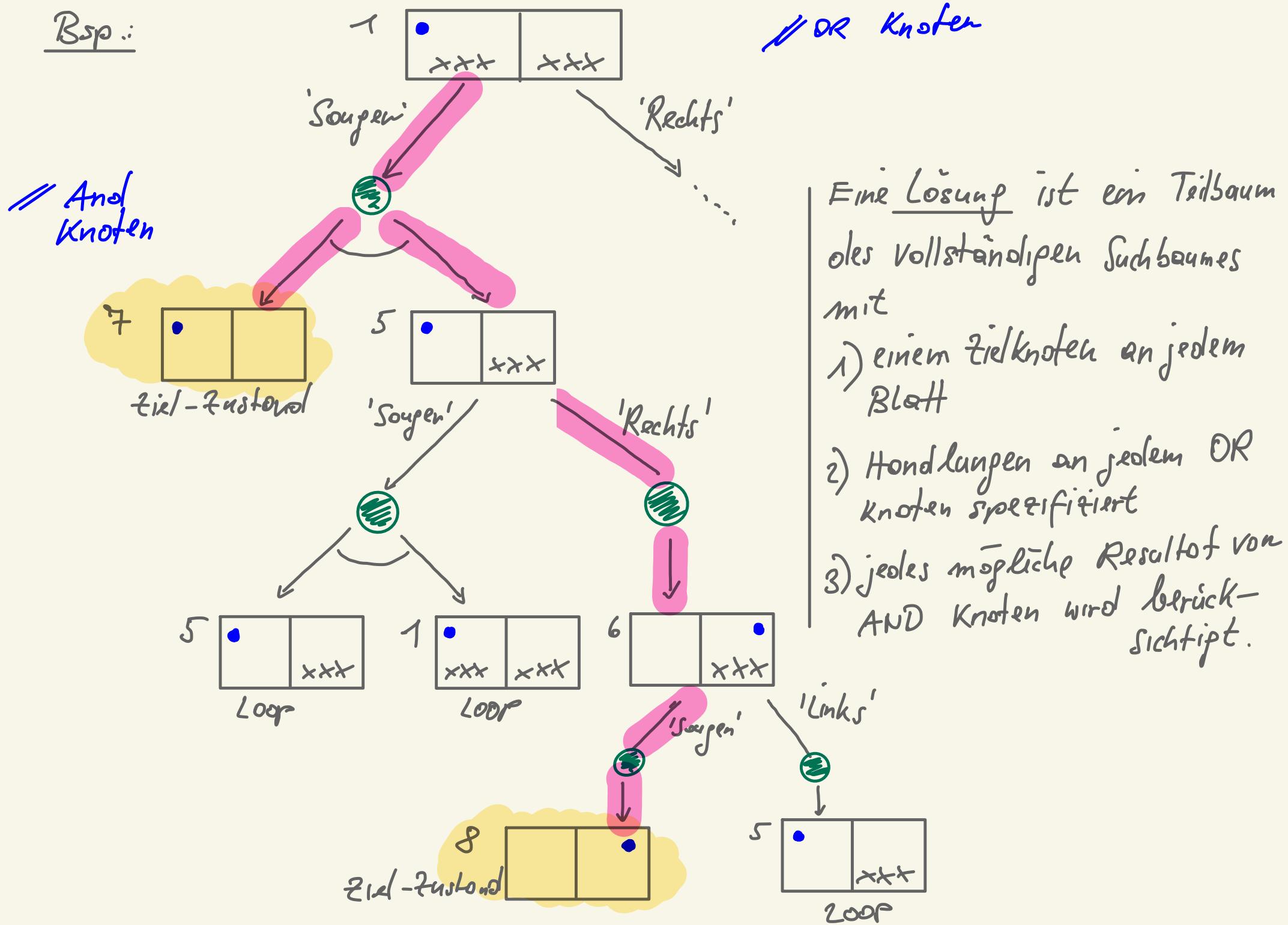


(Agent wählt Handlung aus)

2) AND Knoten

Umggebung bestimmt das Resultat einer Handlung, z.B. in Zustand 1 führt 'Sagen' zu den Belief States $\{5, 7\}$. Also brauchen wir einen Plan für 5 und 7.

Bsp.:



Suche ohne Beobachtungen (Bsp. von partiell beobachtbarer Umgebung)

Bsp.: Sensorlose Staubsaugerwelt (deterministisch). Agent kennt seine Umgebung, jedoch nicht seine Position od. die Verteilung von Schmutz.

→ Die initialen Belief States sind $\{1, 2, 3, \dots, 8\}$. Bewegt sich der Agent nach rechts (Handlung: 'Rechts'), sind die Belief States $\{2, 4, 6, 8\}$; (bekannt auch ohne Sensorik).

Hätte man [Rechts, Saugen] wäre der Agent entweder in Zustand 4 od. 8, also Belief States $\{4, 8\}$.

! ... mit [Rechts, Saugen, Links, Saugen] erreicht der Agent einen Zielzustand, egal wo er startet!

Eine Lösung in einem sensorlosen Problem ist also eine Reihe
an Handlungen (kein Plan da wir nichts wahrnehmen).