

Problem - Lösen mittels Suche (Suchalgorithmen)

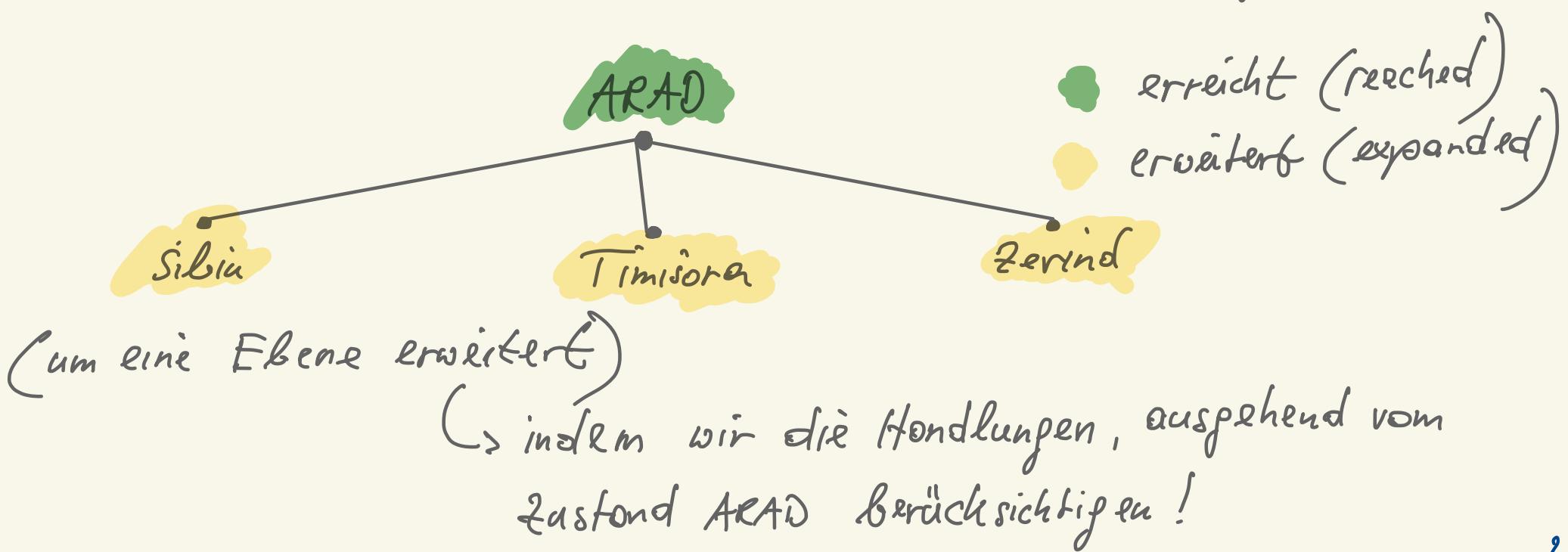
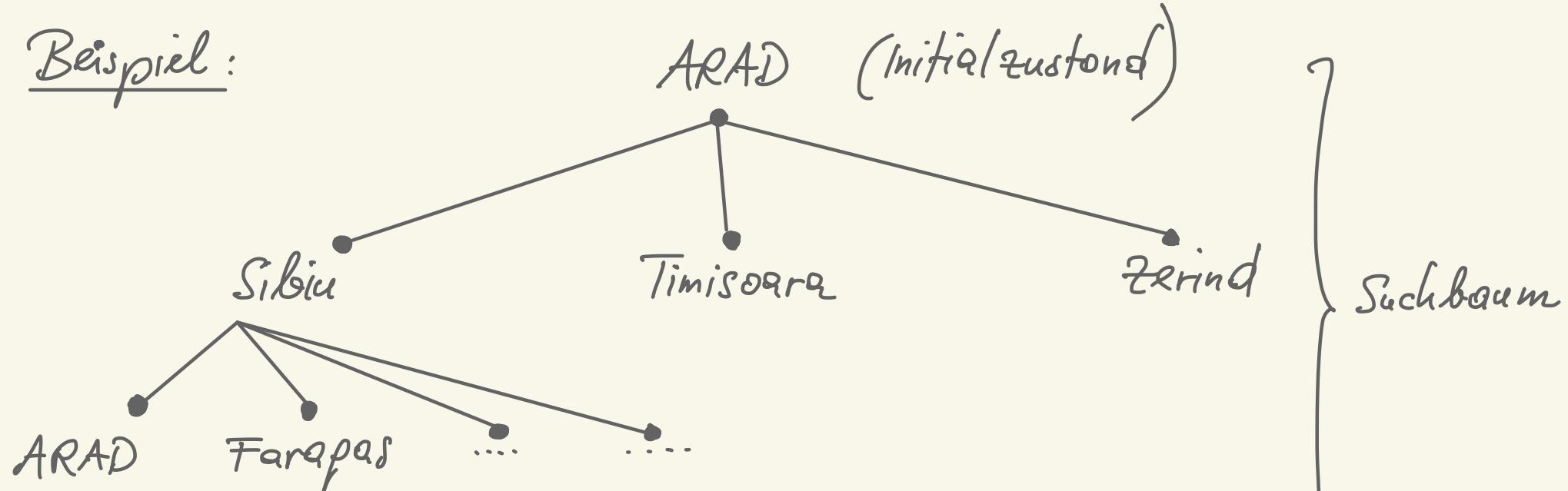
Wir betrachten Algorithmen die einen **Suchbaum** über dem Zustandsraum aufbauen.
(state space)

Wir versuchen einen Pfad in diesem Baum (Suchbaum) zu finden der, ausgehend von einem Startknoten, unser Ziel erreicht.

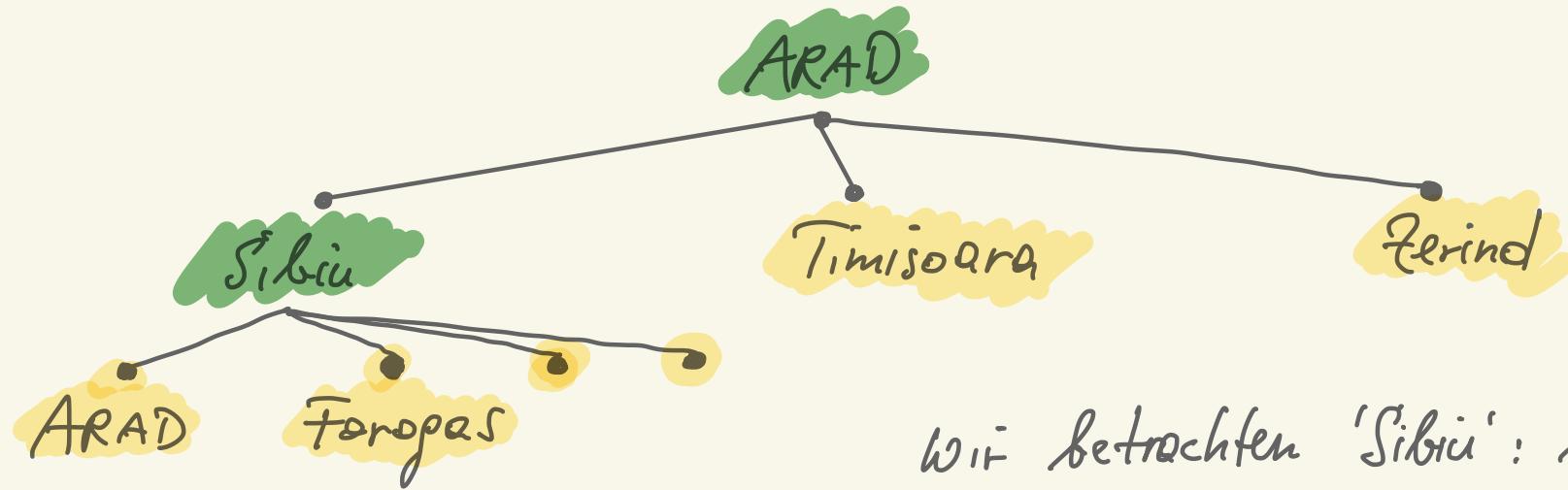
Knoten (im Baum) Zustand im Zustandsraum
 Kanten (im Baum) Handlungen

Der Wurzelknoten (root node) im Baum ist unser Initialzustand.

Beispiel:

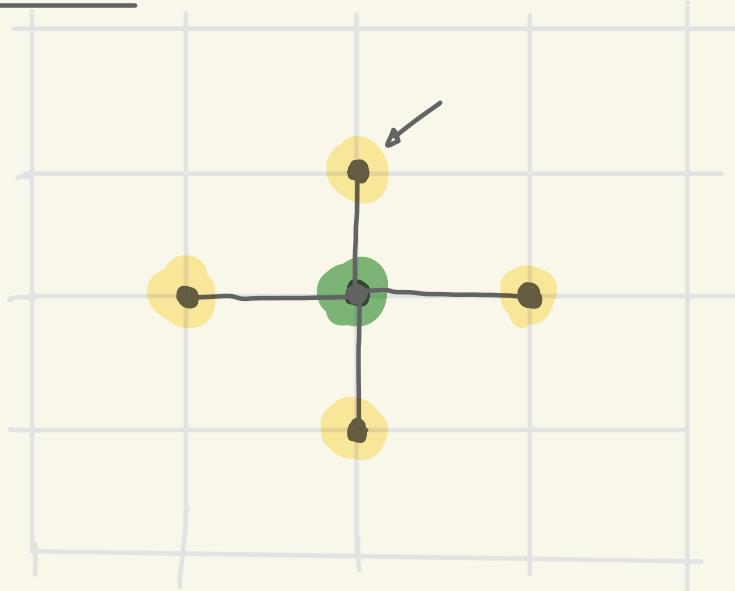


Jeder Kind-Knoten (child node) $\{Sibiu, Timisoara, Zerind\}$ hat ARAĐ als Eltern-Knoten (parent node).

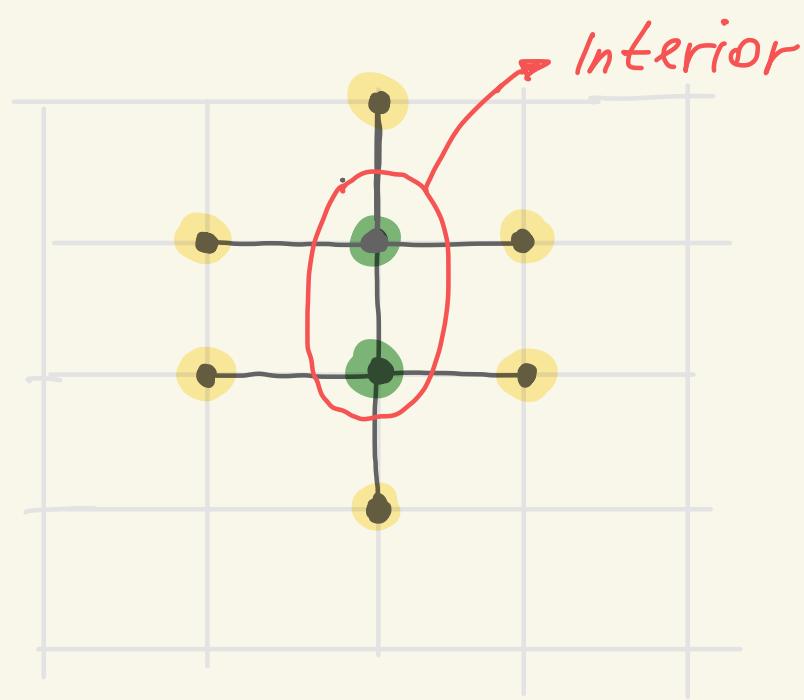


wir betrachten 'Sibiu': hier haben wir gesamt 6 noch nicht selbst erweiterbare Knoten. Wir nennen diese Knoten die FRONT (frontier).

Skizze :



Front



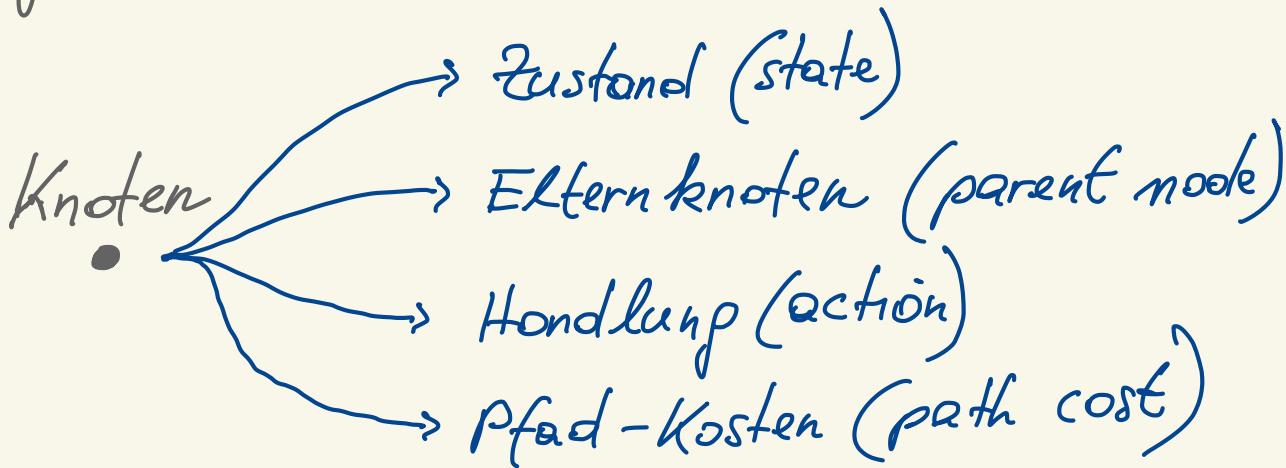
erweitert

Best-First Search

Wir wählen Knoten n mit minimalem Wert einer Evaluierungsfunktion $f(n)$. Also, in jedem Schritt wählen wir einen Knoten der Front mit minimalem $f(n)$, überprüfen ob der Knoten bereits unser Ziel ist, oder erweitern den Knoten. Erweitern generiert Kind-Knoten (child nodes) die wir zur Front hinzufügen (sofern nicht schon erreicht, z.B. A* im Beispiel; es könnte jedoch sein, dass der Knoten neu hinzugefügt wird, sofern die Pfadkosten geringer sind).

Je nach Wahl von $f \rightarrow$ anderer Suchalgorithmus

Was benötigen wir?



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

- ▷ 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)

Anmerkung (redundante Pfade):

E.B.: ARAD → SIBIU

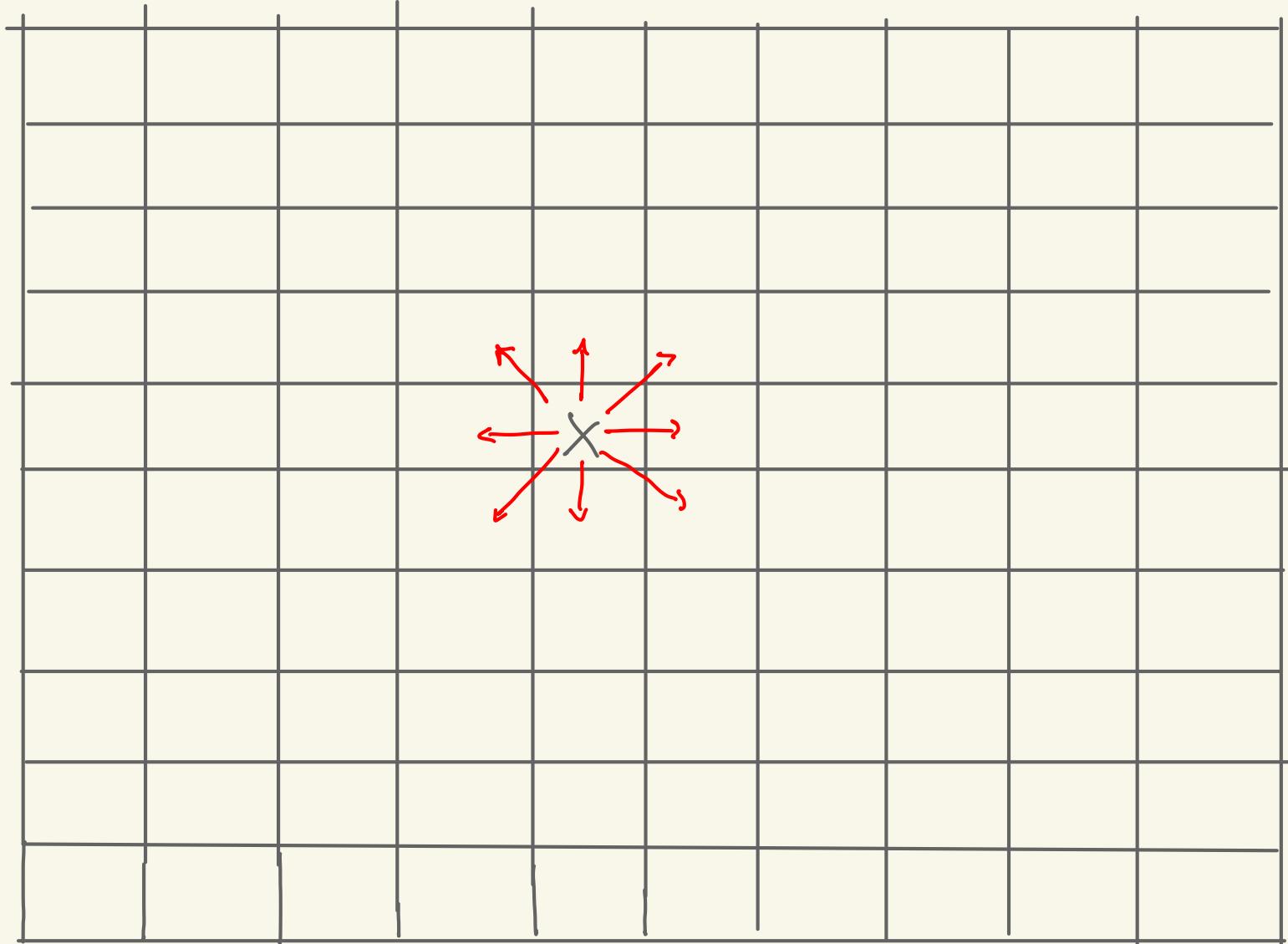
ARAD → ZERIND → ORADEA → SIBIU

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

Beispiel :

10x10 Gitter

Handlung
→



Hier erreicht alle Zellen mit 9 Handlungen , oder weniger .

Anzahl der Pfade der Länge $\ell \approx 8^{\ell}$
(ca. 130 Millionen)

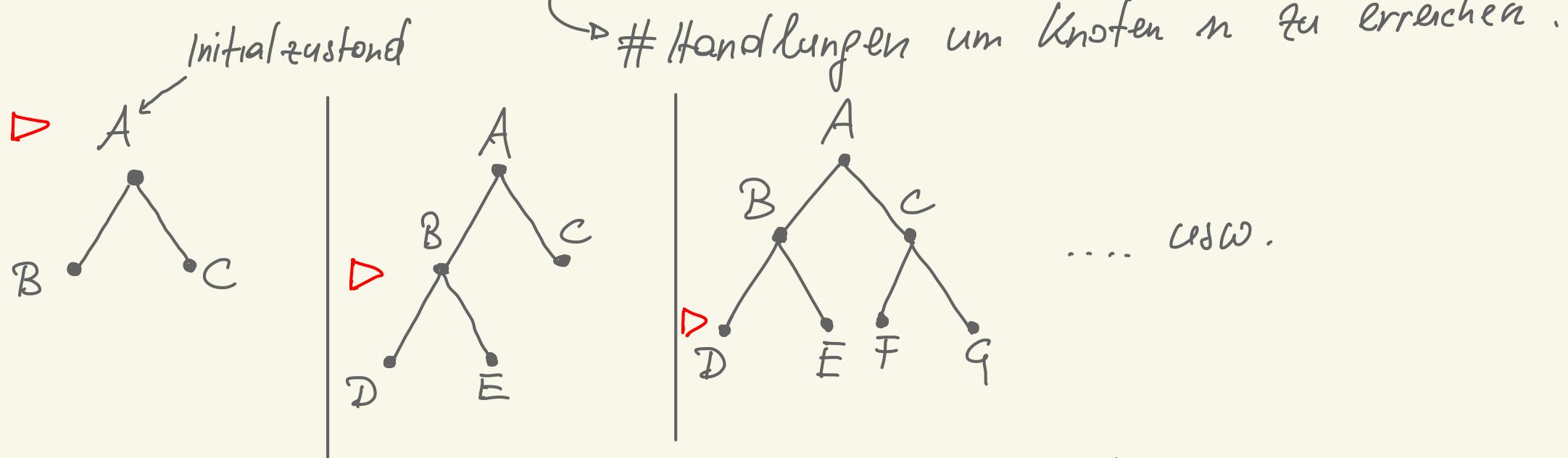
↳ da 8 mögliche Nachbarn
→ da wir am Rand weniger Möglichkeiten hätten.

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.²
- **Kosten - Optimalität** (cost-optimal): findet man eine Lösung mit minimalen Kosten unter allen Lösungen.
- **Zeit - Komplexität** (time-complexity): Anzahl der betrachteten Zustände und Handlungen.
- **Platz - Komplexität** (space-complexity): wie viel Speicher benötigen wir?

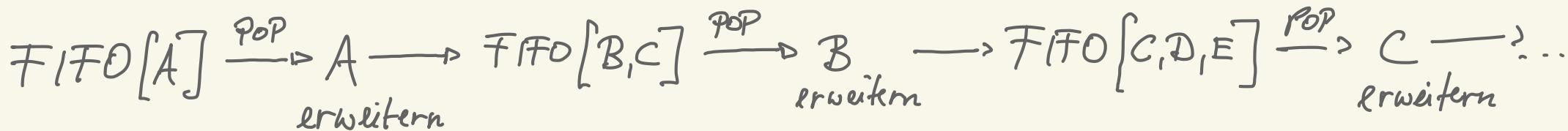
Beispiel: Breadth-First Search (Breitensuche) BFS

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



Brauchbar wenn Handlungen die gleichen Kosten haben.

FRONT als **FIFO** (First-In-First-Out) Warteschlange:



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

ABER, sehr Speicherplatz intensiv!

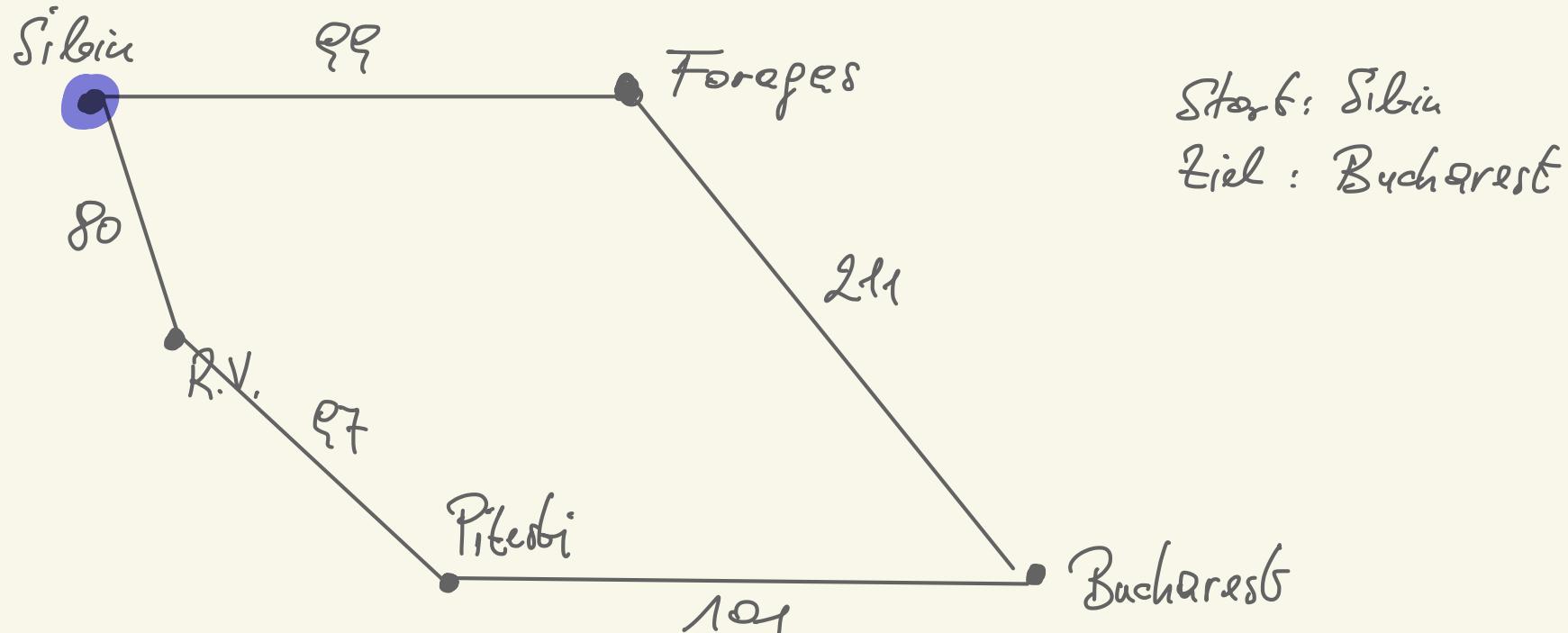
Bsp. generiert jeder Knoten \textcircled{b} Kind-Knoten (child nodes), haben wir bei Tiefe d exponentielles Wachstum!

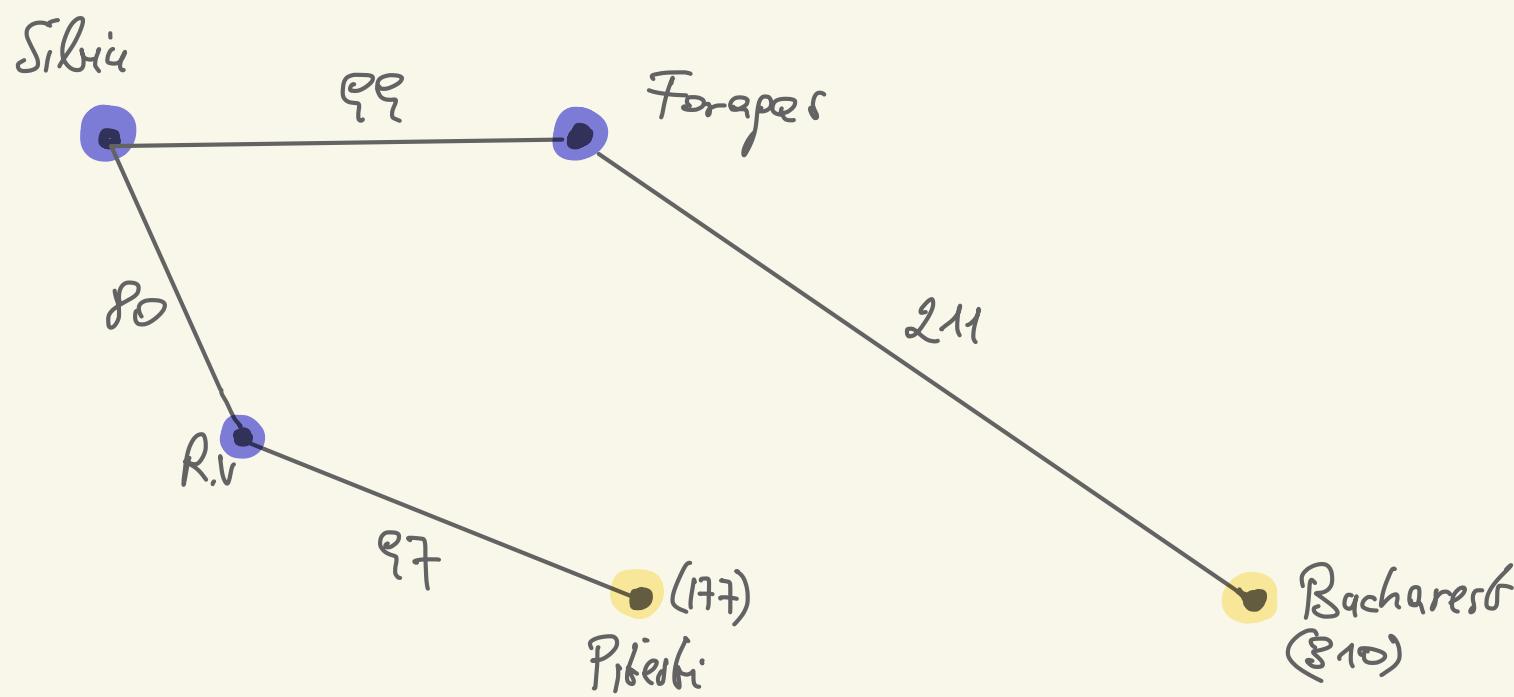
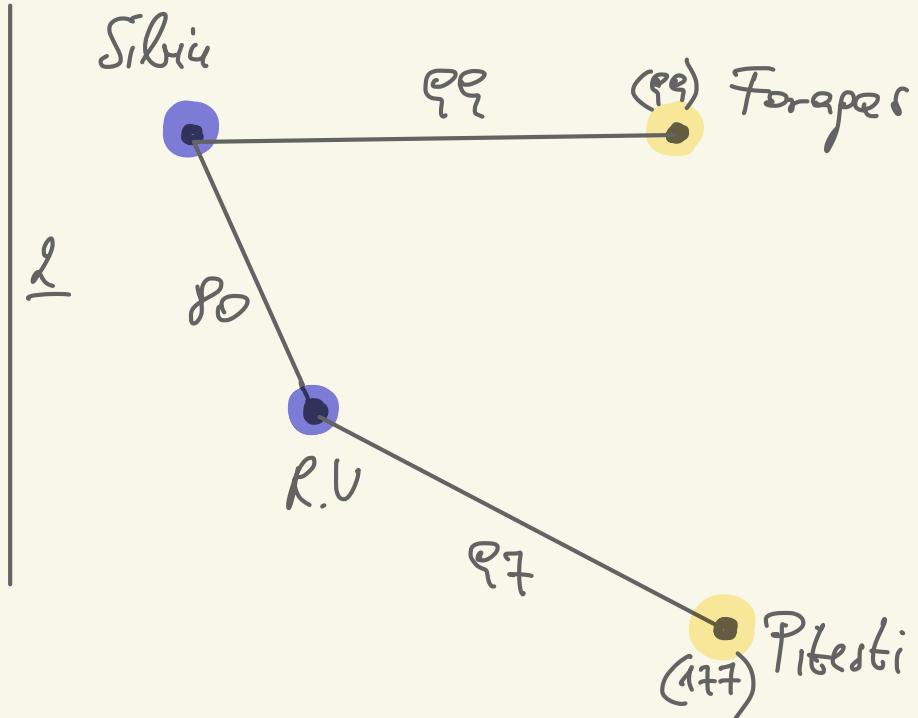
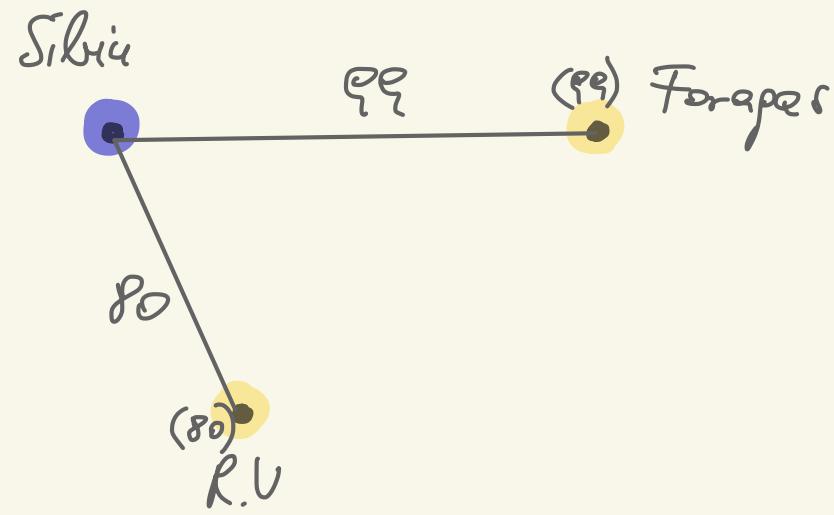
$$1 + b + b^2 + \dots + b^d$$

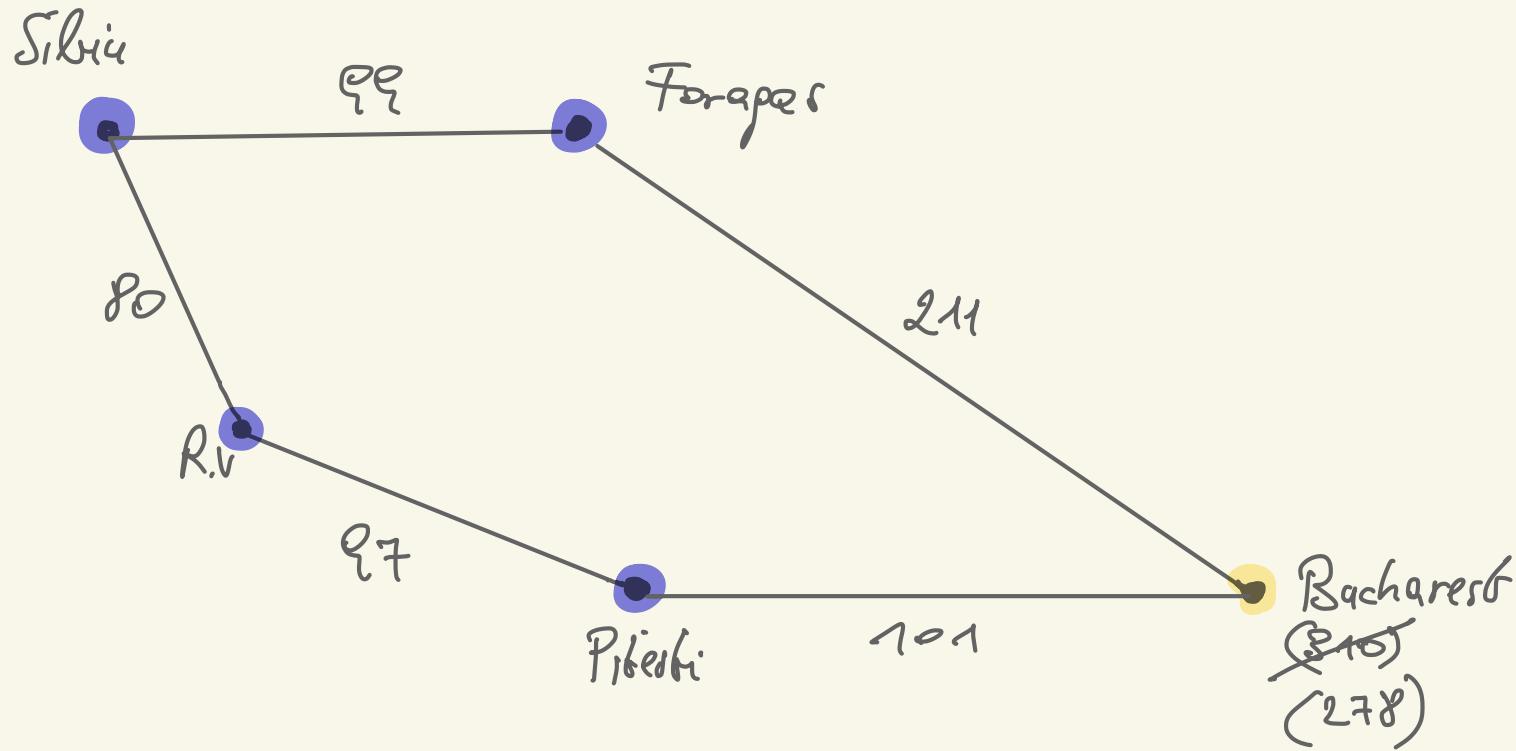
Knoten. Sagen wir $b=16$. Bei $\underbrace{d=10}_{\text{Tiefe}}$: $\sim 16^{10}$ Knoten

Weiters Beispiel : Dijkstra - Algorithmus

$f(n)$ = "Kosten von Wurzelknoten zu Knoten n"







5 Bucharest erreicht; Ist Ziel? Ja! Pfadkosten: 278! (Algorithmus terminiert)

Alle bisherigen Such-Algorithmen sind Instanzen sogenannter uninformierter Suche (also jene wo nicht bekannt ist, wie weit ein Zustand vom Zielzustand entfernt ist). Hat man Heuristiken über die Distanz zum Ziel, nennt man solche Such-Alg. informierte Such-Alg.

z.B.: A* Such-Alg.
 geschätzte Kosten des günstigsten Pfades von
 n zum Ziel!

$$f(n) = \underbrace{g(n)}_{\text{Pfad-Kosten von Start/Wurzel Knoten zu}} + \underbrace{h(n)}_{\text{Knoten } n}$$

pfad-kosten von Start/Wurzel Knoten zu
 Knoten n

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

Bsp.:

(8-puzzle)

		1	3
2	4	7	
5	6	8	

Startzustand (n)

	3	6
1	4	7
2	5	8

Zielzustand

Höpflche Heuristik: Manhattan-Distanz jeder Kachel zur Zielposition

Bsp.: (Startzustand zum Ziel): $\underbrace{2}_{\text{Kachel } \boxed{1}} + 1 + 1 + 0 + 1 + 3 + 0 + \underbrace{0}_{\text{Kachel } \boxed{8}} = 8$

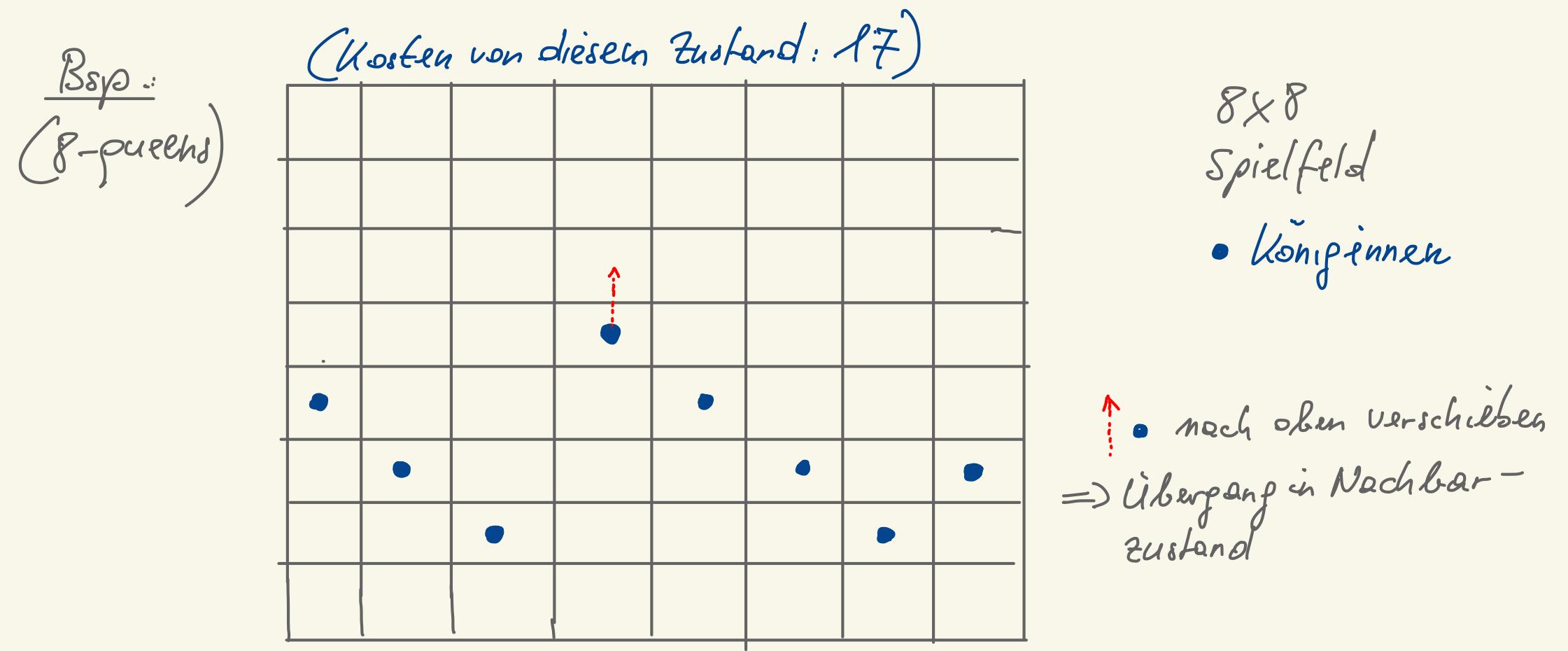
Suche in komplexen Umgebungen

1

Optimierungsprobleme & lokale Suche

Manchmal interessiert uns nur der "finale Zustand", nicht der Pfad dorthin.

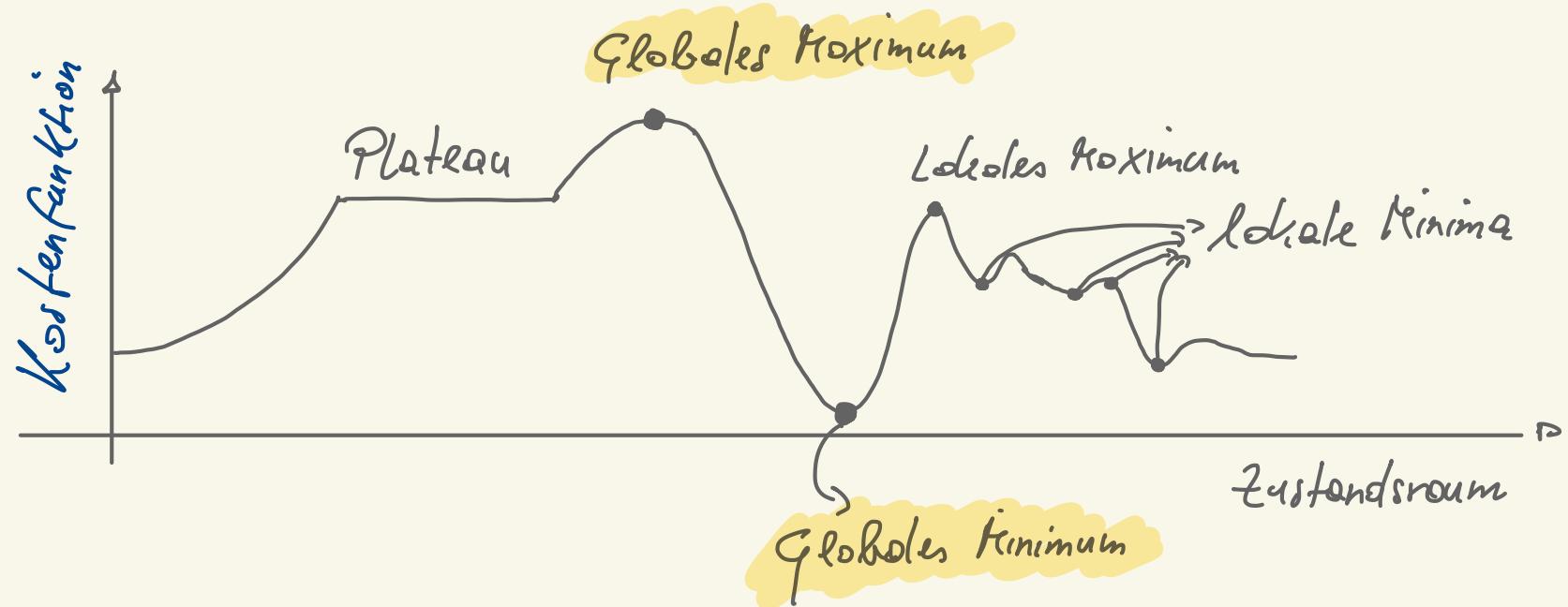
Lokale Suche (local search) funktioniert indem man, ausgehend von einem Start-Zustand, nur in den Nachbarzuständen sucht (ohne sich den Pfad zu merken oder auch welche Zustände man schon erreicht hat).



Kosten = # Paare von Königinen die sich diagonal, als auch über Spalten u. Zeilen hinweg anstreifen (auch wenn eine F. für dazwischen ist).

Vorteile: geringer Speicherplatzbedarf & man findet "brauchbare" Lösungen (auch in unendlich großen Zustandsräumen).

Nachteile: viell. durchsucht man nie Teile des Zustandsraumes wo eine Lösung (Zielzustand) 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 einem höheren Wert gibt.

Mögliche Erweiterungen :

- Stochastic Hill - Climbing : zufällig aus möglichen Zustandswechseln auswählen;
- Random - Restart Hill - Climbing : mehrmals von zufälligen initialen Zuständen starten (und jenen finalen Zustand nehmen mit mak. Kosten).

Simulated Annealing (für Minimierungsprobleme)

Ähnlich wie Hill-Climbing, aber anstatt den lokal besten Zustandswechsel zu machen, wählt man "zufällig" nach folgendem Prinzip: ist der neue Zustand besser (also geringere 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 Annealings verringert

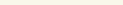
$$e^{-\frac{\Delta E}{T}}$$

Bsp.: x = aktueller Zustand

$y \in \underbrace{\text{Umgebung}(x)}$

Nachbarschaftszustände von x

- 1) Kosten(y) \leq Kosten(x) \Rightarrow Zustandswechsel wird akzeptiert
- 2) Kosten(y) $>$ Kosten(x) \Rightarrow Akzeptieren mit Wahrscheinlichkeit
 $e^{-(\text{kosten}(y) - \text{kosten}(x)) / T}$

Evolutionäre Algorithmen  einer bestimmten Größe

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

Bsp.: (8-queens) ⑫

Population

2 4 7 4 8 5 5 2
Königin in Zeile 2 von Spalte 8

12 ↘ Königin in Zeile 7 von Spalte 3

ein Individuum (also ein Zustand)

3 2 7 5 2 4 1 1

10 ↗ Fitness (d.h.: #Angriffe)

Als Beispiel: Selektion der 2 "Fittesten"

$$\begin{array}{r|rr} 2 & 4 & 7 & 4 & 8 & 5 & 5 & 2 \\ \hline 3 & 2 & 7 & 5 & 2 & 4 & 1 & 1 \end{array} \xrightarrow{\text{Crossover}} \begin{array}{r} 24752411 \\ 32748552 \end{array}$$

"Crossover Point
(wird zufällig gewählt)
Rekombinationspunkt

Weiters gibt es die Möglichkeit zur "Mutation":

$$\begin{array}{r} 24752411 \\ \downarrow \text{mutiert z.B. zu 8} \end{array} \left(\text{je nach } \underbrace{\text{Mutationsrate}}_{\text{Parameter}} \right)$$

24758411

Wichtige Parameter

- Größe der Population
- Repräsentation der Individuen
 - String (Zeichenkette) über endlichem Alphabet
→ **genetic algorithms**
(wie in unserem Beispiel)
 - Sequenz von reellen Zahlen ⇒ **evolution strategies**
 - als Computer Programm ⇒ **genetic programming**
- Selektionsprozess (z.B.: proportional zum Fitness-Wert)
- Crossover-Operation u. Crossover-Point
- Mutationsrate
- Generierung der neuen Population (z.B.: 50% Fitteste Eltern + Nachkommen, etc.)

