

SUCHALGORITHMEN

Wir betrachten Algorithmen die einen Suchbaum (search tree) über dem Zustandsraum aufbauen.

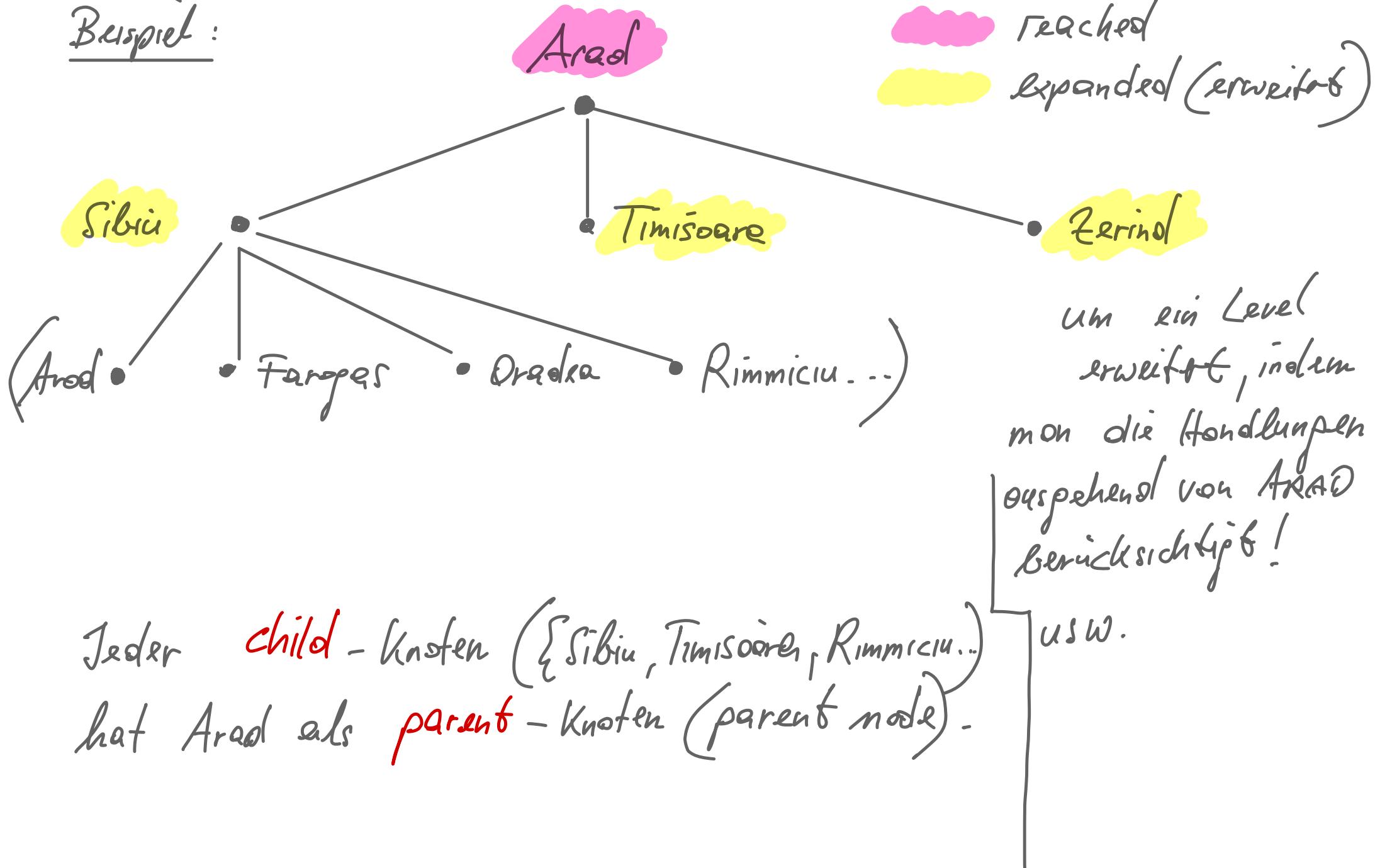
Wir versuchen einen **Pfad** in diesem Baum zu finden, welcher ausgehend von einem **Startknoten (root node)** unser Ziel erreicht.

Knoten im Baum = Zustände im Zustandsraum

Kanten im Baum = Handlungen

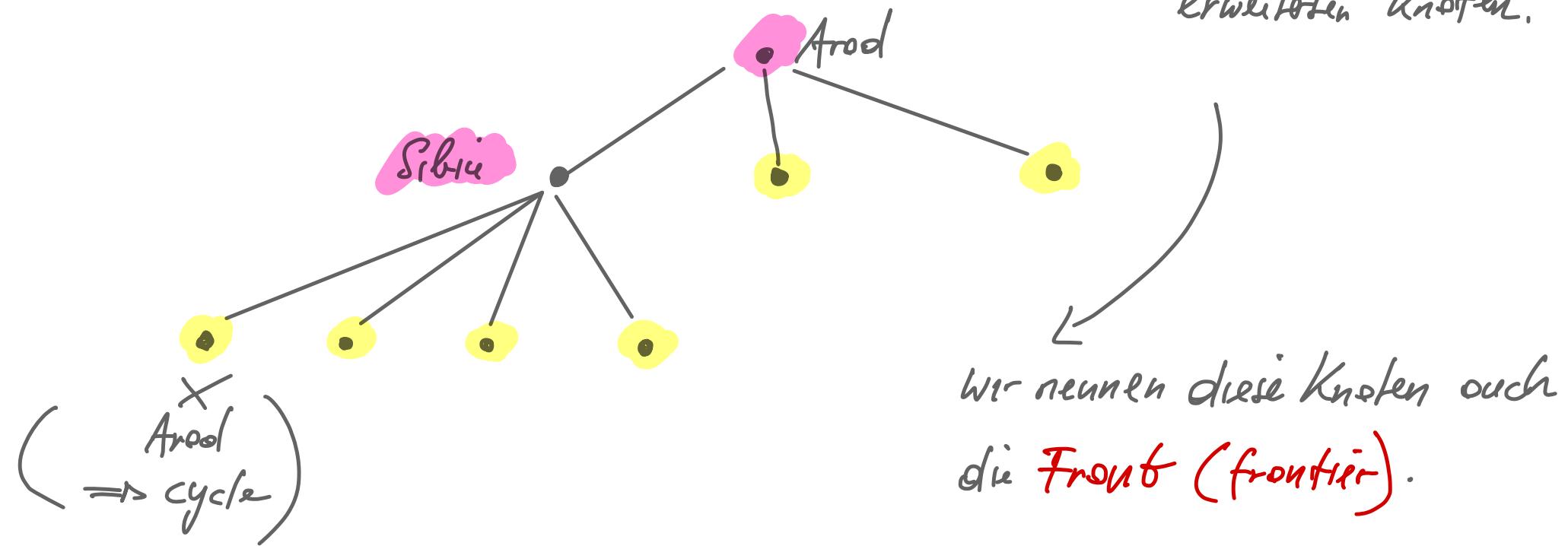
Der Wurzelknoten/Startknoten ist der Initialzustand (z.B.: A₀₀)

Beispiel:



Wie entscheidet man sich, welchen child-Knoten man als Nächster betrachtet?

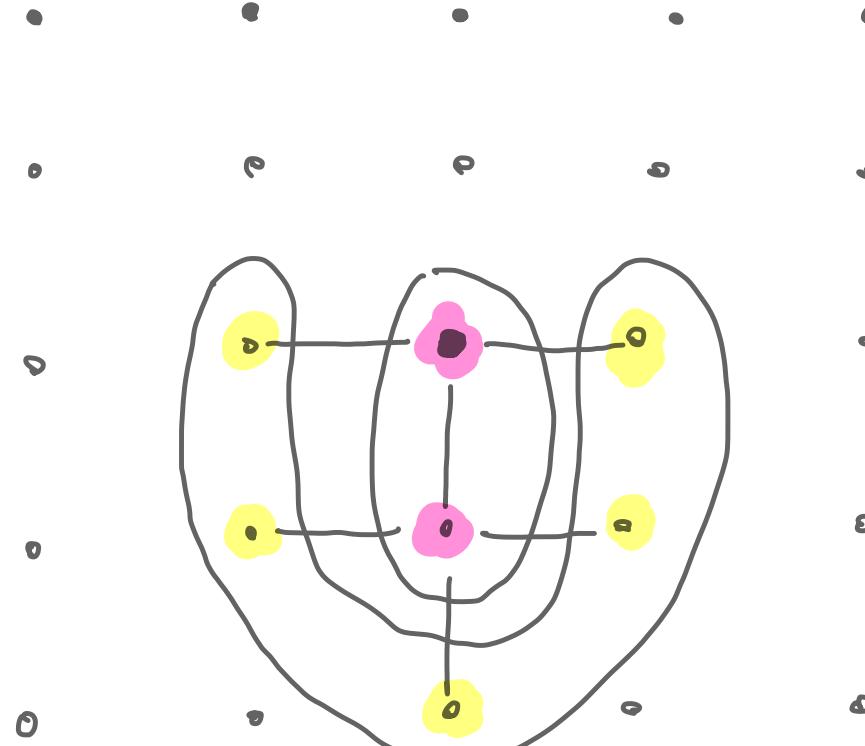
Nehmen wir an, wir betrachten Sibia: wir haben eine Menge an 6 nicht erweiterten Knoten.



Wie können wir die Performance von Suchalgorithmen quantifizieren?

- ▷ Vollständigkeit (completeness): findet man eine Lösung wenn es eine gibt? meldet der Alg. einen Fehler wenn es keine Lösung gibt?
- ▷ Kosten-optimal (cost-optimal): Finden wir eine Lösung mit minimalen Kosten unter allen Lösungen.
- ▷ Zeit-komplexität (time complexity): Anzahl an betrachteten Zuständen und Handlungen.
- ▷ Platz-komplexität (space complexity): wieviel Speicher benötigen wir?

Anm. zu frontier:

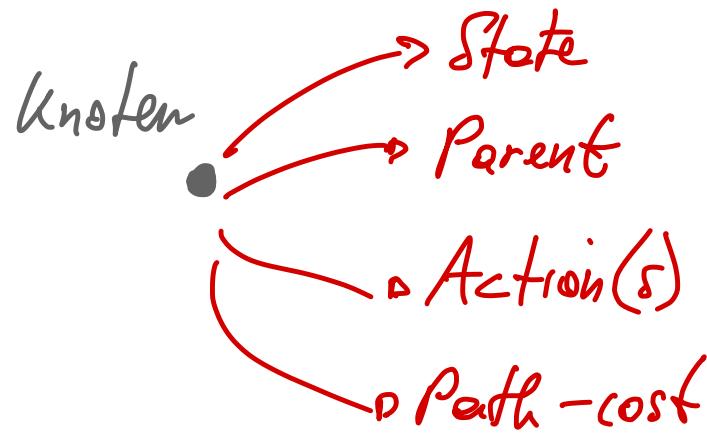


Beispiel : Best-First Search

Wir wählen einen 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 unser Ziel ist, odl. wir erweitern den Knoten. "Erweitern" generiert Child-Knoten die wir zur Front hinzufügen (sofern nicht schon erreicht, z.B., Arod; einen Knoten kann jedoch auch wieder neu hinzugefügt werden sofern die Pfadkosten geringer sind).

Je nach Wahl von $f(n) \Rightarrow$ unterschiedliche Suchalgorithmen.

Bsp. Implementierung: Was benötigen wir?



Um die Front zu speichern benötigen wir eine Art Warteschlange (queue).

D Wir müssen überprüfen können ob die Queue leer ist (IS-EMPTY).

D Wir müssen den obersten Knoten entfernen und zurückgeben können (POP)

▷ wir müssen den obersten Knoten zurückgeben können (TOP)

▷ wir müssen Knoten hinzufügen können (ADD)

(Ann.: z.B.: priority queue)

Such-Alg. die redundante Pfade berücksichtigen nennt man
graph-Suchalgorithmen; falls nicht auf redundante Pfade
geprüft wird, nennt man dies Baum-ähnliche Suchalgorithmen
(tree-like search).

Ahm: Natürlich ist bei Baum-ähnlicher Suche der Zustandsraum immer noch der gleiche Graph, aber wir behandeln ihn wie einen Baum mit immer nur einem Pfad von einem Knoten zurück zur Wurzel.

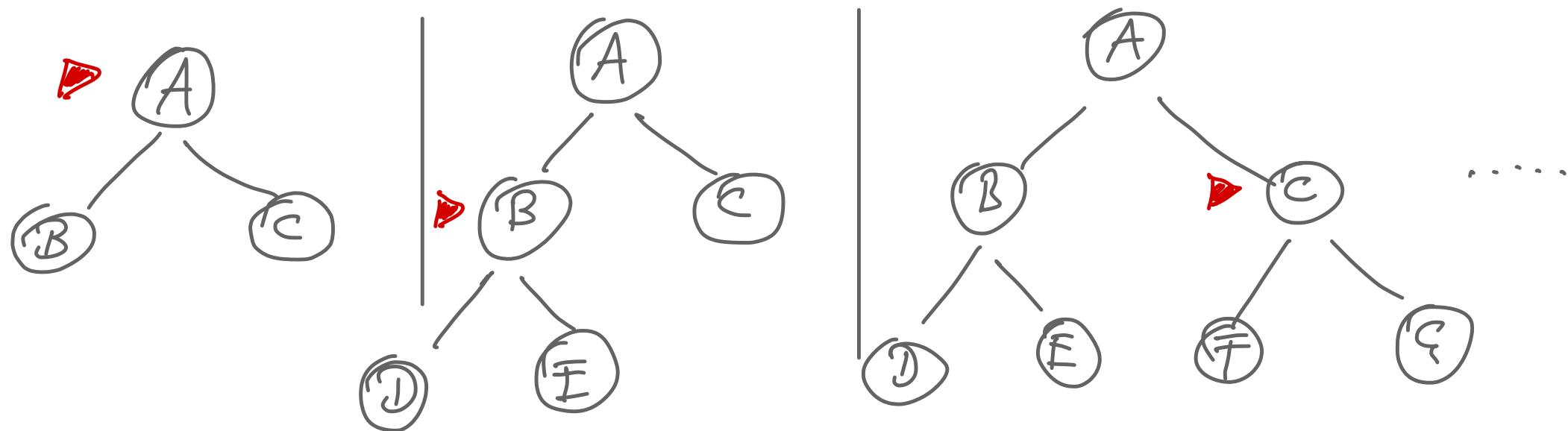
Beispiel (Breadth-First Search)

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

\rightarrow # Handlungen um Knoten n zu erreichen

Brauchbar, wenn alle Handlungen die gleichen Kosten haben.

Prinzip: Wir erweitern zuerst den Wurzel-Knoten (root node), dann alle "child"-nodes, dann alle "child"-nodes oder "child"-nodes, usw.



Extrem Speicher-intensiv! Generiert jeder Knoten beispielweise **b** "child"-nodes, haben wir bei Tiefe $d+1$:

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

→ exponentielles Wachstum

Andres Beispiel: $f(h) = \text{Kosten von Wurzel zu Knoten } n$

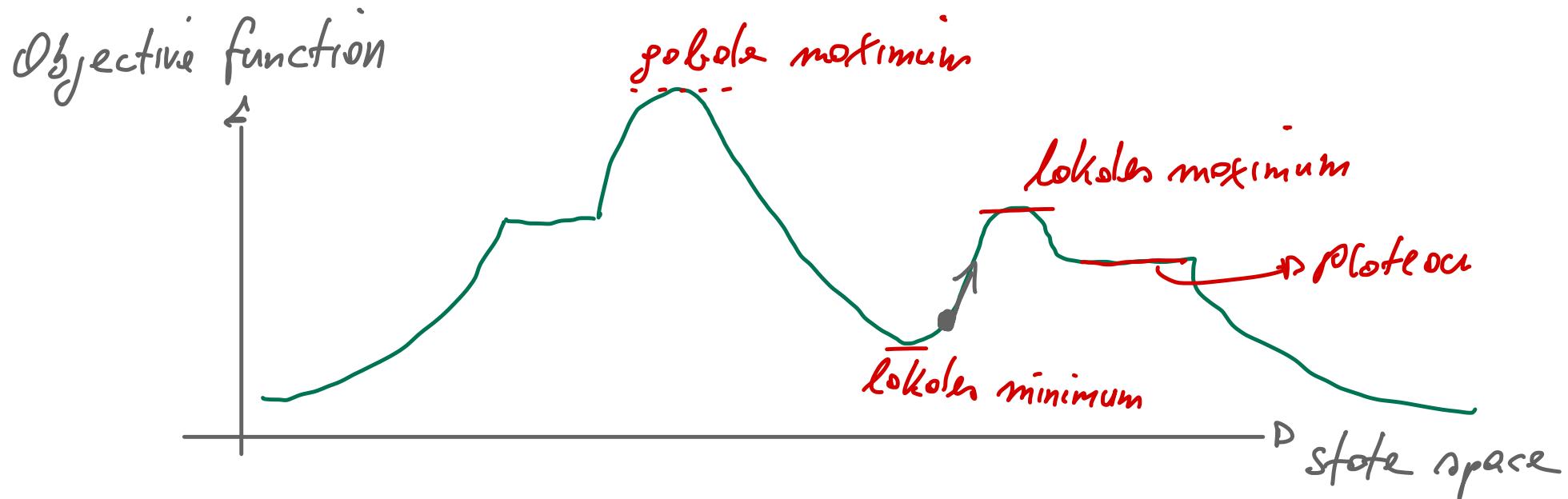
(Dijkstra Algorithmus)

Lokale Suche & Optimierungsprobleme

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

Vorteile: geringer Speicherplatzbedarf & man findet oft brauchbare Lösungen (auch in unendlichen Zustandsräumen).

Nachteil: viel. durchsucht man meist den Teil des Zustandsraumes wo eine Lösung liegt.

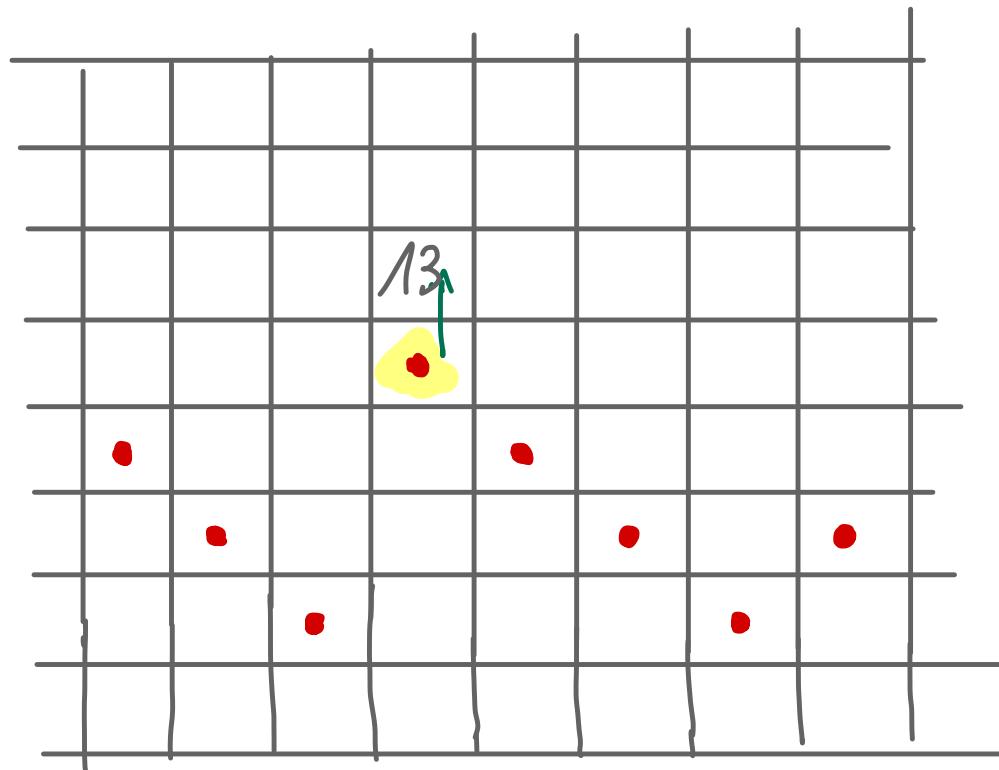


Ist unser Ziel den "höchsten" Punkt zu finden \Rightarrow Hill-climbing
Algorithmen; wollen wir Kosten minimieren \Rightarrow Gradient descent
Algorithmus;

Hill - Climbing

Prinzip: man merkt sich den aktuellen Zustand und geht zu jinem
Nachbarzustand mit dem höchsten Wert (der Objective
function; siehe Grafik oben) \Rightarrow iteratives Verfahren;
terminiert wenn kein Nachbarzustand einen höheren Wert
hat als der aktuelle Zustand.

Beispiel (8 -queens):



• = Queen

Aktuelle Kosten: 17

(# Posen die sich über Spalten, Zeilen u. diagonal attackieren – auch wenn eine Figur dazwischen)

Simulated Annealing: Ähnlich zu Hill-Climbing, aber erlaubt den lokalen besten Zustandswechsel zu machen, wählt man zufällig. Ist der neue Zustand besser, wird dieser akzeptiert; wenn nicht, wird er mit Wahrscheinlichkeit < 1 akzeptiert. Die W-Kü \ddot{u} b sinkt exponentiell mit ΔE , d.h. mit der Differenz um die sich der Zustand verschlechtert. Zusätzlich: Temperatur Parameter T (dieser ist anfänglich hoch und wird im Laufe des Verfahrens verringert).

$$e^{\frac{\Delta E}{T}} \quad (\text{Akzeptanz - Wahrscheinlichkeit})$$

Boltzmann Verteilung

Anm.: x - aktueller Zustand, $y \in \text{Umfgebung}(x)$: falls

(Minimierungs-
problem)

Kosten(y) \leq Kosten(x) \Rightarrow y wird akzeptiert

Kosten(y) $>$ Kosten(x) \Rightarrow y — n — mit

Wahrscheinlichkeit

$$\ell = \frac{\text{Kosten}(y) - \text{Kosten}(x)}{T}$$

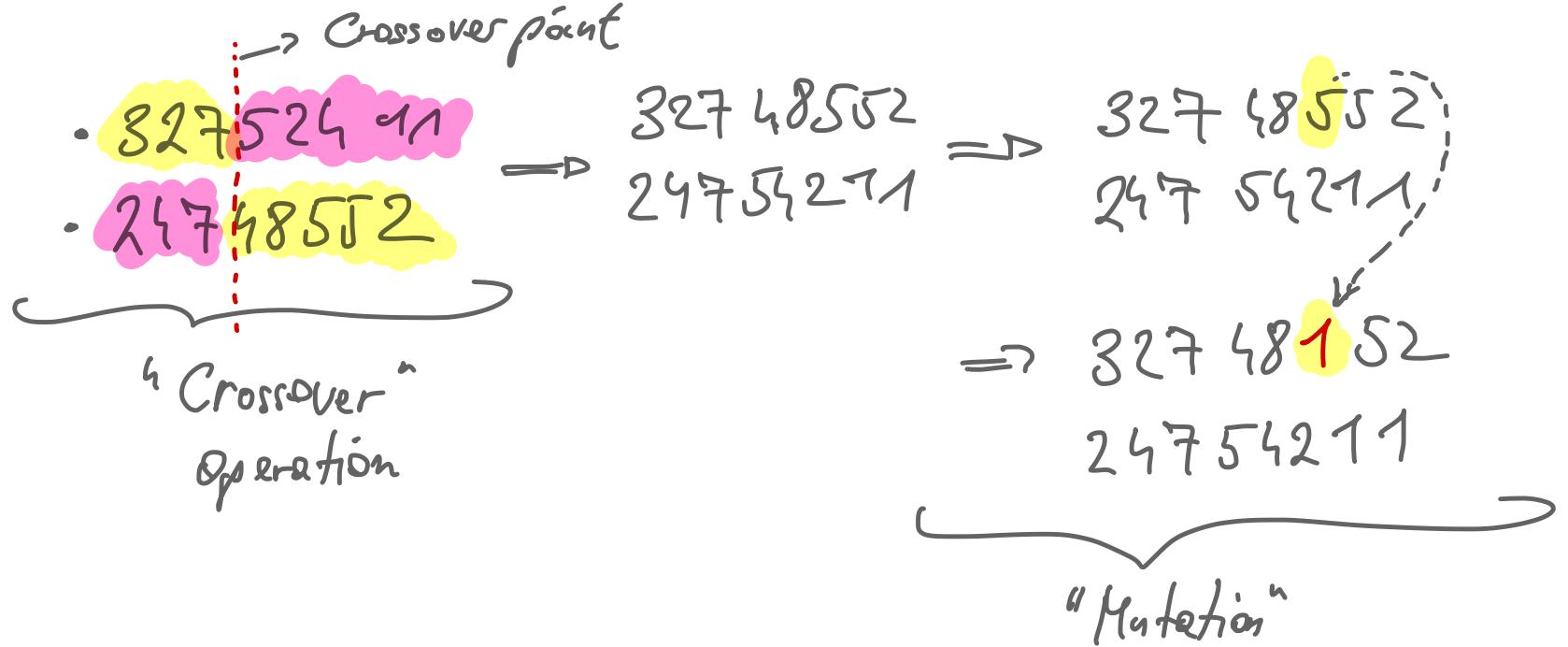
Evolutionäre Algorithmen

Grundprinzip: Population von Individuen (repräsentieren die Zustände); in dieser Population generieren die "Fittesten" Nachkommen (also neue Zustände).

Beispiel: (8-Punkte) 

Konjunktiv I (also \tilde{f} -Fälle) in
Zeile 2

\Rightarrow nächste Folie



Wichtige Parameter:

(i) Größe der Population

(ii) Repräsentation der Individuen:

▷ String über endlichen Alphabet (siehe Beispiel)

⇒ "genetische Algorithmen" (genetic algorithms)

▷ Sequenz reeller Zahlen
→ "Evolution strategies"

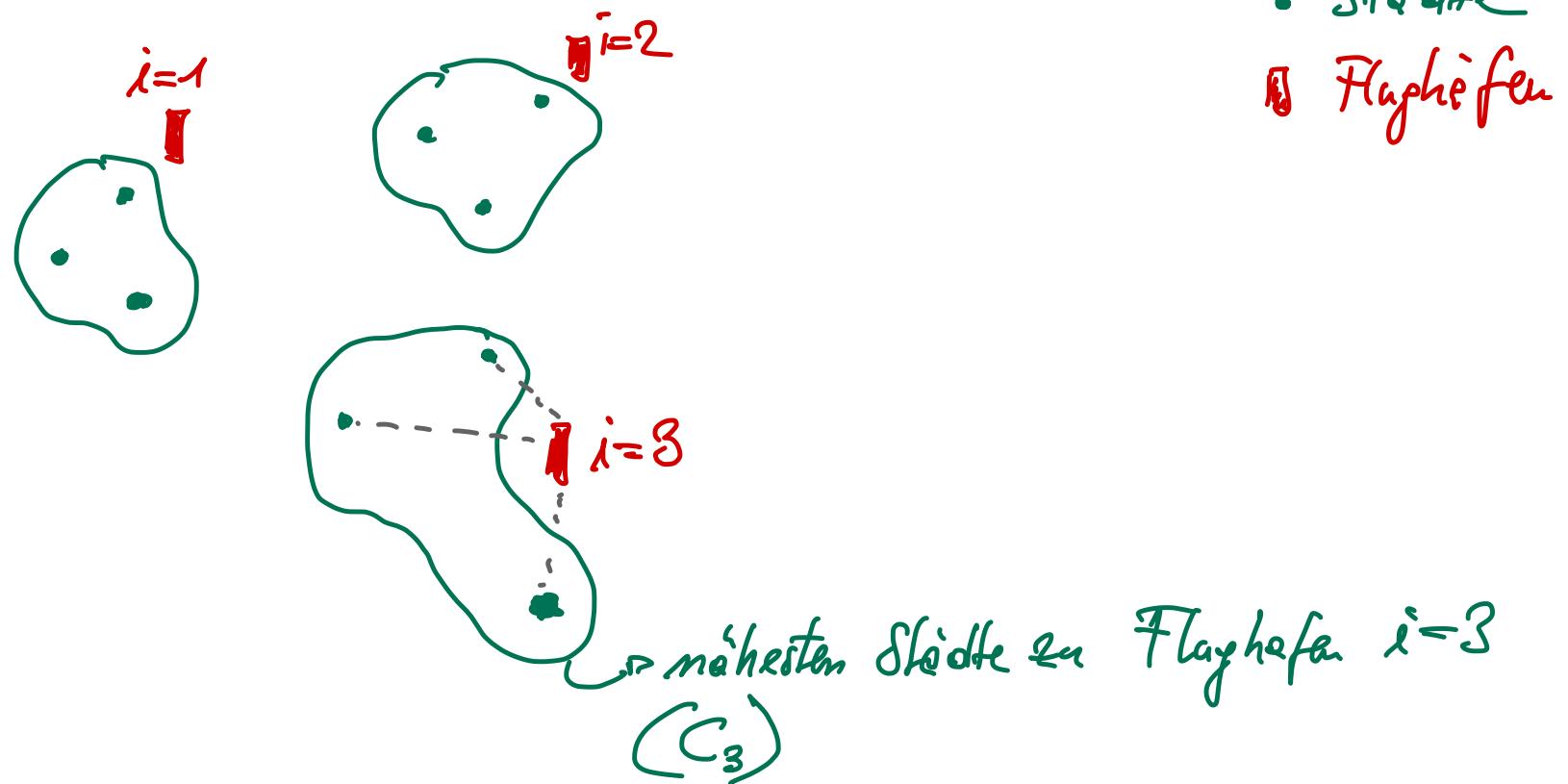
▷ als Computerprogramm

⇒ "genetic programming"

- (iii) Selektionsprozess (z.B.: α zum Fitness Wert)
- (iv) Rekombinationspunkt (crossover point)
- (v) Mutationssrate
- (vi) Generierung neuer Individuen; z.B.: nur die nach Fitness-Wert am höchsten bewerteten produzieren neue Nachkommen und die bilden die neue Population; andere Variante: man nimmt auch Individuen aus der Elterngeneration d.h.g.

Lokale Suche in stetigen Zustandsräumen

Beispiel: 3 Flughäfen in Rumänien platzieren, sodass die Summe der Luftlinien-Distanzen zu den jeweils nächsten Städten minimal ist.



Koordinaten $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ sind reelle Zahlen.

Wir haben eine Funktion in 6 Variablen:

$$f \underbrace{(x_1, y_1, x_2, y_2, x_3, y_3)}_{\vec{x}}$$

$$f(\vec{x}) = \sum_{i=1}^3 \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

Problem: Wenn wir einen der Flughäfen weit verschieben, ändert sich C_i .

wir könnten den Gradienten von f nutzen, d.h.,

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \dots, \frac{\partial f}{\partial y_3} \right)^T$$

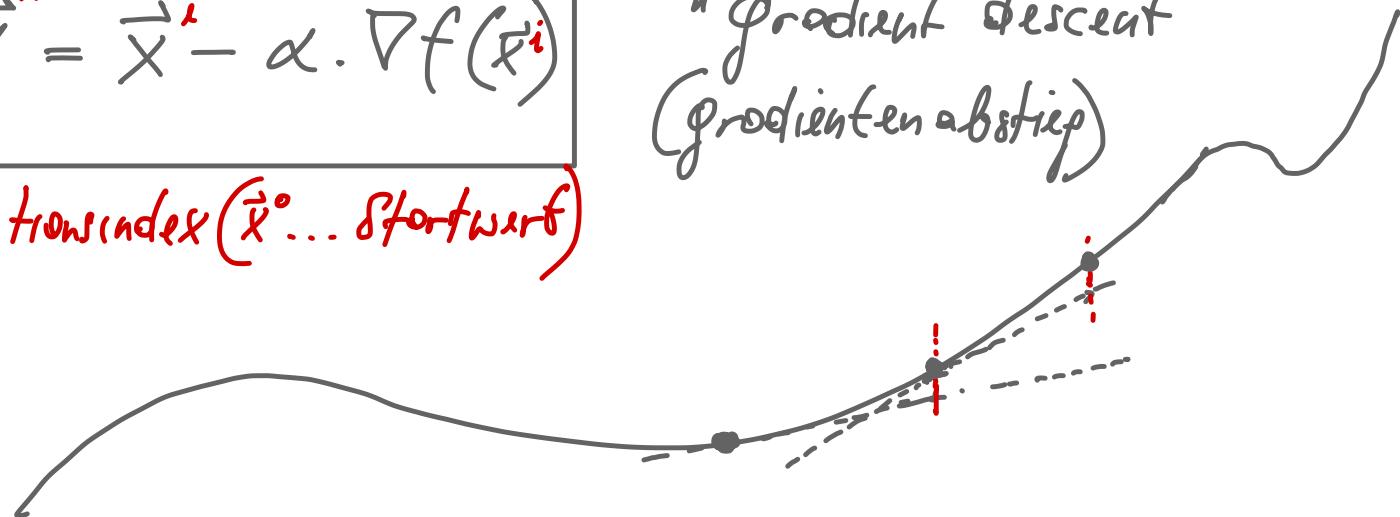
und $\nabla f = 0$ lösen. Bei 3 Städten nicht geschlossen lösbar.

Aber

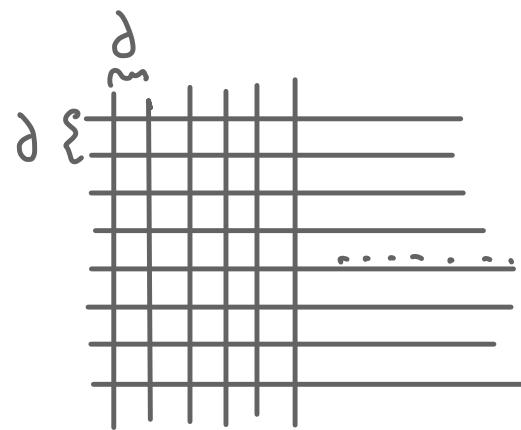
$$\vec{x}^{i+1} = \vec{x}^i - \alpha \cdot \nabla f(\vec{x}^i)$$

i ... Iterationsindex (\vec{x}^0 ... Startwert)

"Gradient descent"
(Gradientenabstieg)



Alternativ dazu: Diskretisieren und mit bekannten Lösungsverfahren
arbeiten.



Suche bei nicht-deterministischen Aufforderungsumgebungen

Der Agent kennt nicht genau den Folgezustand einer Handlung.
Die Zustände die vom Agenten als möglich angesehen werden, nennt man
belief states.

Eine Lösung in einer solchen Situation ist keine Sequenz mehr, sondern
ein **Alternativplan (contingency plan)**; auch Strategie genannt.

Beispiel (abgeändert): Handlungen RIGHT, LEFT, SUCK

- Ist eine Kachel schmutzig, reinigt 'SUCK' die Kachel, aber möglicherweise auch die Kachel daneben.
- —n— sauber, führt 'SUCK' möglicherweise zu einer schmutzigen Kachel.