Die heuristische Funktion *h(n)*

(„Mama, wie weit müssen wir'n noch!?“)

Zulässigkeit: *h* unterschätzt die tatsächlichen Kosten *h** :

$h(n) \leq h^*(n)$

Monotonie: *h*-Wert von Knoten *n* zu Nachfolger *n'* nimmt höchstens um tatsächliche Aktionskosten ab:

$h(n) \leq c(n,a,n') + h(n')$ (*c*(*n*,*a*,*n'*): Kosten mit *a* von *n* zu *n'*)

Monotonie impliziert Zulässigkeit, aber nicht umgekehrt.

Dominanz: h_2 dominiert h_1 (ist „besser informiert“) :

$h_2(n) \geq h_1(n)$ für alle *n*, und beide sind zulässig

Knotenbewertung ***f*(*n*)=*g*(*n*):** Werte nur Pfadkosten

Knotenbewertung ***f*(*n*)=*h*(*n*):** Werte nur Restkostenschätzung

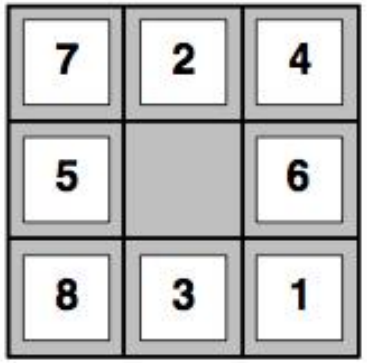
Knotenbewertung ***f*(*n*)=*g*(*n*)ă*h*(*n*):** Werte beide Anteile

Beispiele für *h*-Funktionen im Verschiebespiel

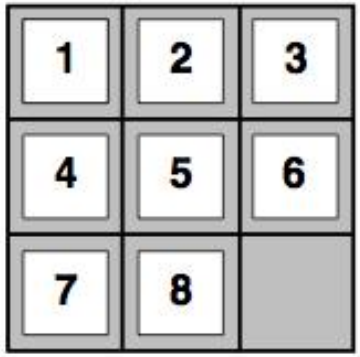
***h*(*n*) = 0**: monoton, daher zulässig
… und maximal uninformiert

h*(*n*) =** Zahl falsch liegender Plättchen in ***n:
zulässig (nicht monoton!)
Beispiel: *h*(Startzustand) = 6

h*(*n*) =** „Manhattan-Distanz“ in ***n:
(Summe der Schritte je Plättchen)
zulässig (nicht monoton!)
Beispiel: *h*(Startzustand) = 14
(= 4+0+3+3+1+0+2+1)



Startzustand



Zielzustand

Bestensuche (*greedy best-first*)

… ein Beispiel für einen *greedy* („gierigen“) Algorithmus

- Geh aus von Suchalgorithmen-Schema Folie 122,
- bewerte Knoten durch ***f(n)=h(n)*** (zulässig oder nicht)
- sortiere in Suchfront nach Knotenwerten (billigste vor)

➥ ende bei zuerst gefundenem Zielknoten

☹ Zeitbedarf: $O(b^m)$, wenn m Maximaltiefe des Baums

☹ Speicherbedarf: $O(b^m)$ (da alle Knoten im Speicher)

☹ Unvollständig (da anfällig für „Sackgassen“)

☹ Nicht optimal (siehe folgendes Beispiel)

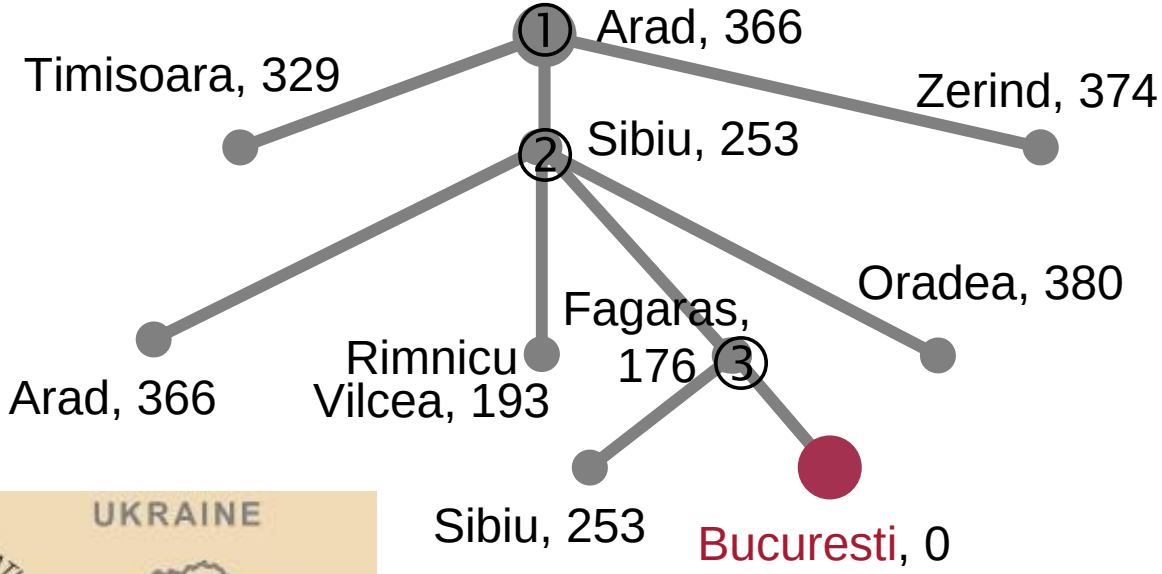
Zeit und Speicher praktisch oft besser bei guter *h*-Funktion!

Worst-Case-Komplexitätsbetrachtungen sind inadäquat für (gute) Heuristikfunktionen!

Beispiel: Bestensuche beim Reiseproblem

Luftlinienentfernungen nach Bucuresti

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Pfadkosten: 450
(s. Wegentfernungen Folie 145)
optimal: 418
(s. Folie 146)

Der Algorithmus A*

- Geh vor wie bei *uniform-cost*-Suche (Standard-Kostensuche),
- bewerte Knoten durch $f(n)=g(n)+h(n)$, wobei $h(n)$ zulässig
- sortiere bei INSERTALL nach Knotenwerten (billigste vor)
- ende erst, wenn ein Zielknoten expandiert werden müsste

☹ Zeitbedarf: $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$ (A* ≈ *uniform cost* für $h(n)=0$)

☹ Speicherbedarf: $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$ (alle Knoten im Speicher)

☺ Vollständig (Details folgen)

☺ Optimal (Details folgen)

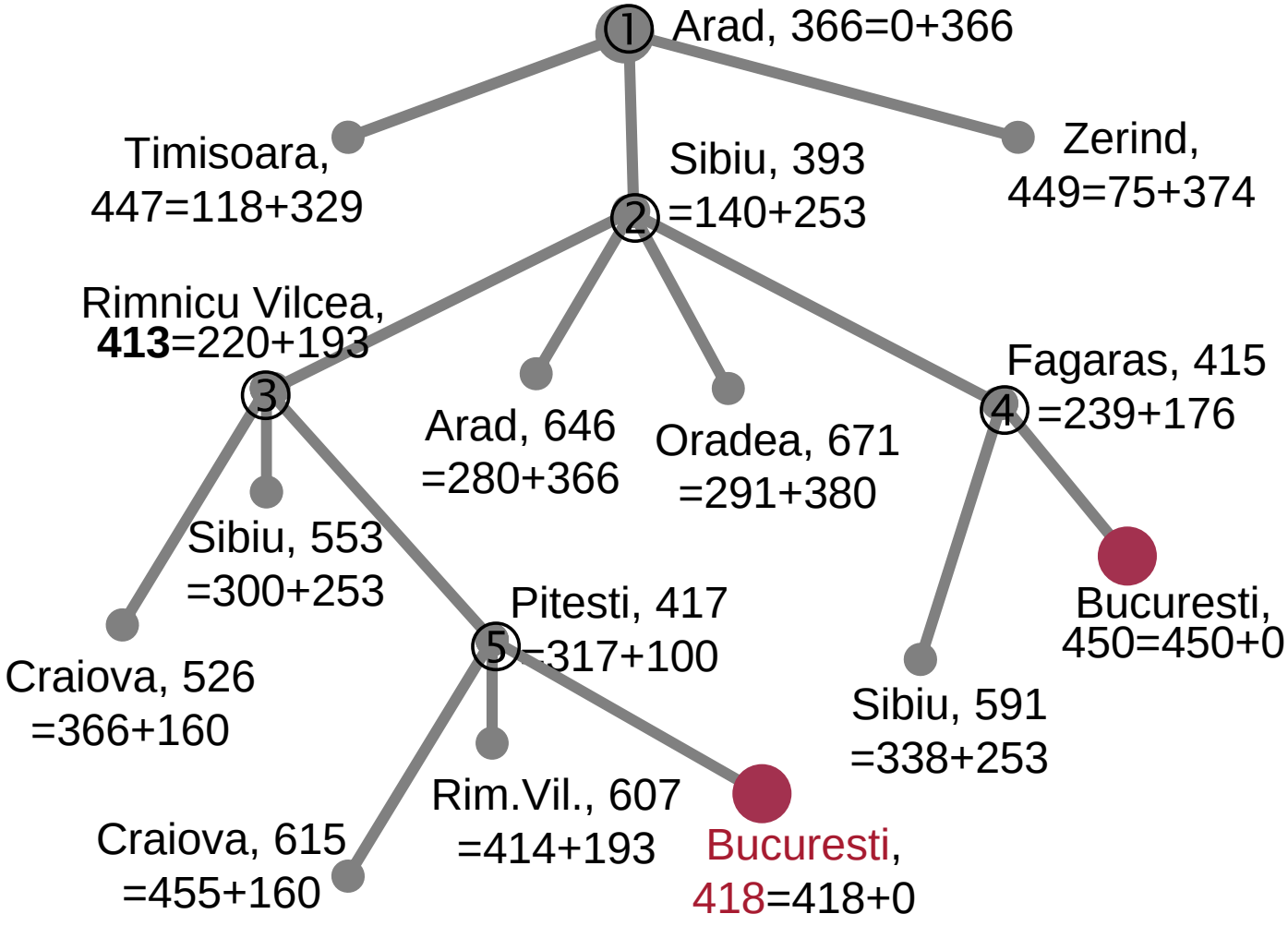
Nils Nilsson, *1933

- Vollständigkeit & Optimalität schon früh bewiesen (1968)
- Nochmal: *worst-case*-Betrachtungen inadäquat für (gute) Heuristiken

Beispiel: A* beim Reiseproblem

Luftlinienentfernungen
nach Bucuresti

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Optimalität und Vollständigkeit von A*

Ist für alle Knoten n Bewertung $f(n)=g(n)+h(n)$ für zuläss. h, existiert eine Lösung, sind alle Aktionskosten >0 und ist $b<\tilde{a}$, so findet A* einen optimalen Lösungsknoten.

Beweisskizze (Widerspruchsbeweis)

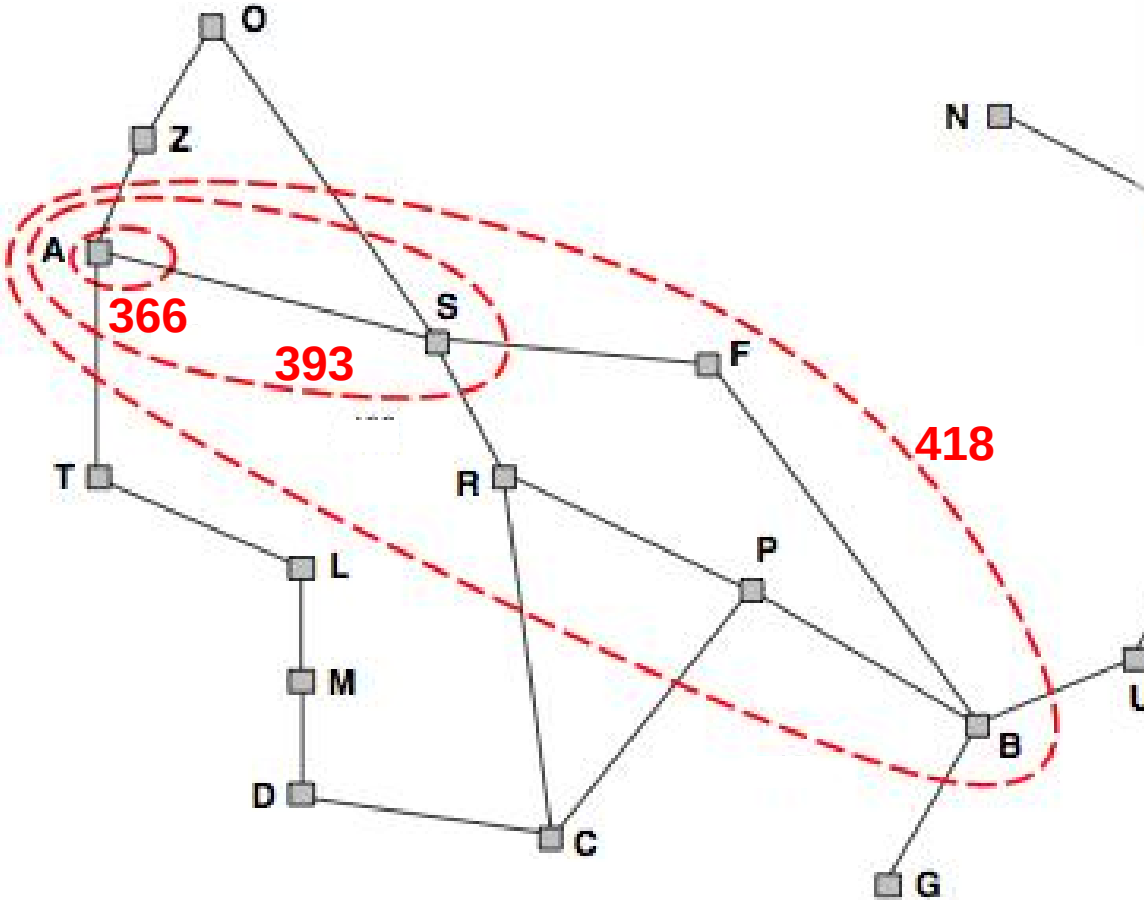
Sei C^* Kosten einer optimalen Lösung im Knoten L, d.h. $f(L)=g(L)+h(L)=C^*+0$
Annahme: A* terminiert mit Lösung in $L^+ \neq L$, wobei $f(L^+) = C^+ > C^*$.

Nach Vorschrift müsste A* alle *fringe*-Knoten mit Kosten < C^+ expandiert haben; wenn A* einen Lösungsknoten expandieren müsste, terminiert er.
Wegen Zulässigkeit von h: Alle Vorgänger k von L haben Kosten $f(k) \leq C^* < C^+$
L und L^+ haben mindestens einen gemeinsamen, voll expandierten Vorgänger.
Folglich sind alle Vorgänger von L expandiert, folglich war L in *fringe*.

Da $f(L) < f(L^+)$, terminiert A* mit der Lösung in L.

Qualitatives Verhalten von A*

A* propagiert wachsende „*f*-Konturen“ ab Start, wobei $f_i < f_{i+1}$



- expandiert alle n: $f(n)<C^*$
- exp. einige n: $f(n)=C^*$
- exp. kein n: $f(n)>C^*$
- A* ist optimal effizient unter den optimalen Algorithmen, gegeben h

Andererseits (schlecht bei vielen Zielknoten im Suchraum): A* macht „Breitensuche durch alle plausiblen Pfade“!

Einfluss der Informiertheit

Ist h_1 informierter als h_2,
so expandiert A*/h_2 mindestens alle die Knoten im Suchraum, welche A*/h_1 expandiert

(Hier ohne Beweis)

Iterierte Tiefensuche mit $f(n)=g(n)+h(n)$: IDA*

- Gleiches Vorgehen wie bei Einheitskosten: Beschränkte Tiefensuche mit Schranke erhöht in £-Schritten
- Das £ ist hier kleinste vorkommende Operatorkosten (70 im Rumänien-Beispiel)
- Statt Tiefe $d(n)$ verwende $f(n)=g(n)+h(n)$ als Knotenwert

☹ Zeitbedarf: $O(b^{(1+\lfloor C^*/£ \rfloor)})$ (analog A* und *uniform-cost*)

☺ Speicherbedarf: $O(b(1+\lfloor C^*/£ \rfloor))$

☺ Vollständig (wie A*)

☺ Optimal (wie A*)

- Oft wird strikte Optimalität aufgegeben für größere Schrittweite w>£: Lösung dann optimal bis auf w–£

A* mit Speicherbegrenzung: SMA*

*Simplified Memory-bounded A**

Gegeben feste Speicherkapazität für M Knoten, sodass

$$b(1+\lfloor C^*/\varepsilon \rfloor) << M << b^{(1+\lfloor C^*/£ \rfloor)}$$

- Wenn: M Knoten im Speicher, Lösung nicht gefunden, und (M+1)-ter Knoten wäre zu speichern:
  - wähle den schlechtest bewerteten *fringe*-Knoten *H* aus mit Bewertung *f(H)*
    (falls mehrere Knoten gleichwertig, nimm zuerst generierten)
  - für den Vorgängerknoten *G* von *H* notiere, dass Kosten für Unterbaum Richtung *H* mindestens *f*(*H*)
  - verdränge *H* aus dem Speicher

☺ Vollständig und optimal, falls optimaler Pfad <*M* Schritte

Zustandsraumsuche vs. Lösungsraumsuche

- Vielfach gibt es im Suchraum nicht nur eine einzige Lösung unter sehr vielen „unfertigen“ Zuständen,
- sondern sehr viele Lösungen sehr unterschiedlicher Qualität,
- und einige (typischerweise „schlechte“ oder „falsche“) Lösungen sind sehr einfach (direkt – ohne Suche) konstruierbar.
  (Beispiel: Terminfestlegung für ein Gruppentreffen)
- „Dasselbe“ Problem kann auf beide Arten darstellbar sein.

Beispiel: *n*-Damen-Problem

- Zustandsraumsuche startet mit leerem *n*x*n*-Brett; findet Folge von *n* Aktionen „Setze_Dame(*i,y*)“, bis *n* Damen unbedroht auf dem Brett
- Lösungsraumsuche startet mit beliebiger Verteilung von *n* Damen in *n* Spalten; verbessert aktuelle Verteilung, bis frei von Bedrohung

„Schlechte Lösung“ des 4-Damen-Problems

➥ Lokale Suche

Lokale Suche I: Bergsteigen (*hill climbing*)

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] < VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

Knoten sind hier Lösungen unterschiedlicher Qualität

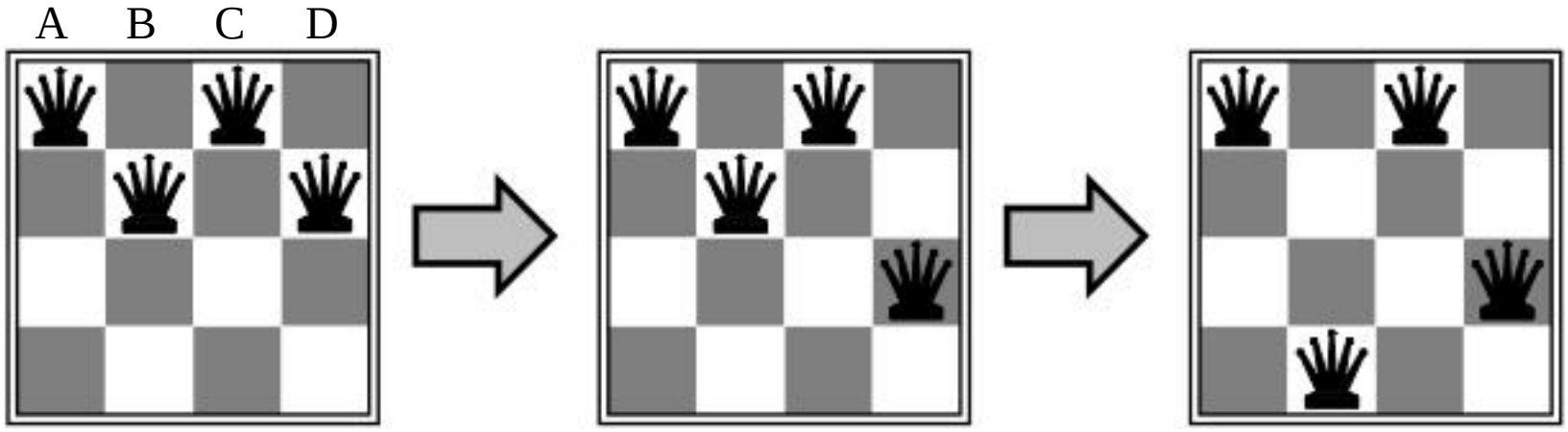
In der Basisversion ist hier ≤ gemeint!

- Nachfolger entstehen durch lokale Veränderung der Lösung
- Die Bewertung muss Fehler in der Lösung „bestrafen“

Beispiel n-Damen-Problem

- Nachfolger durch Verziehen einer Dame in ihrer Spalte
- Bewertung zählt Damenpaare *D*, die sich (direkt oder verdeckt) bedrohen:

  VALUE(node) = 1/(1+|*D*|)

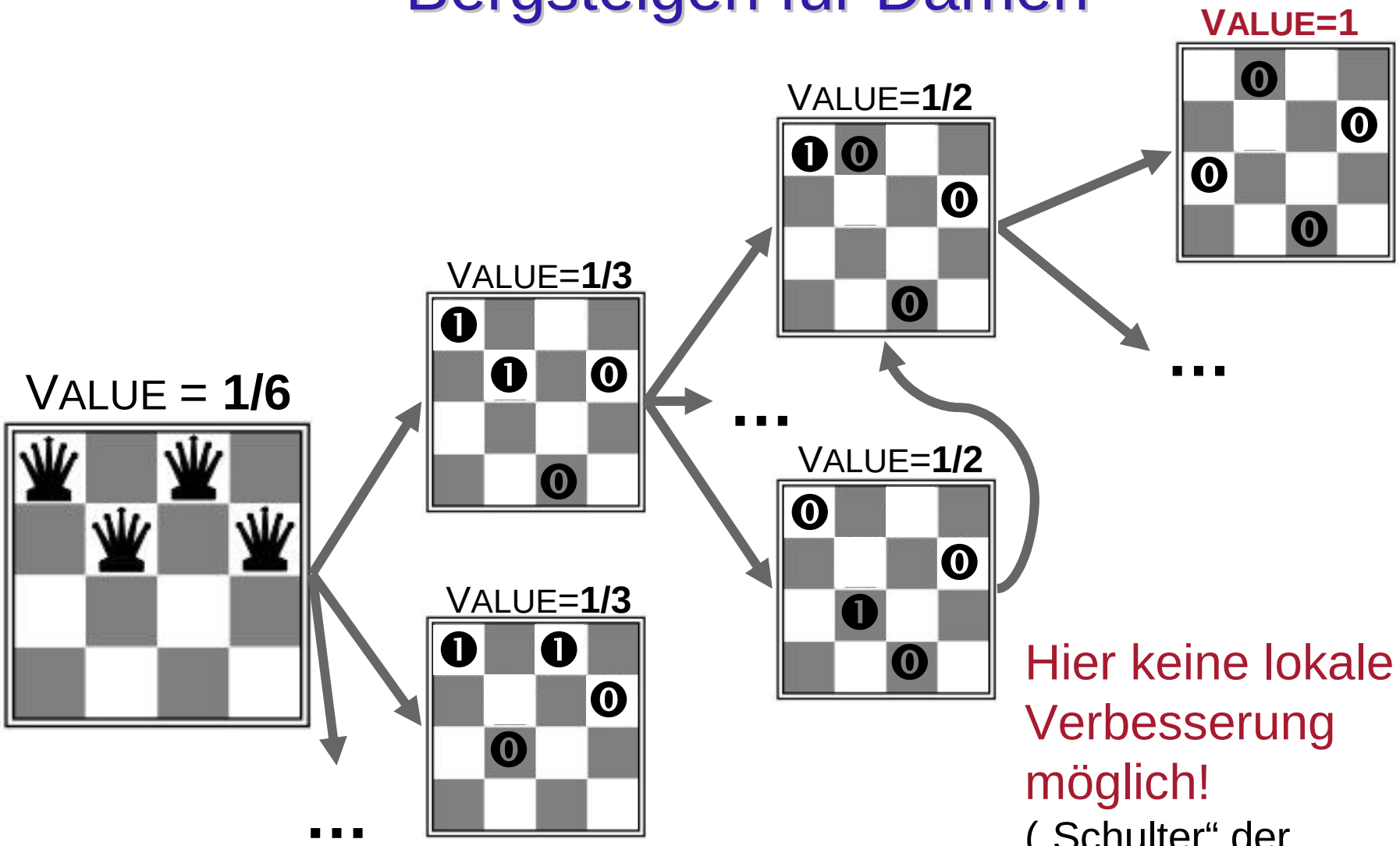


VALUE = 1/6
(AB, AC, BC, BD, CD)

VALUE = 1/4

VALUE = 1/2

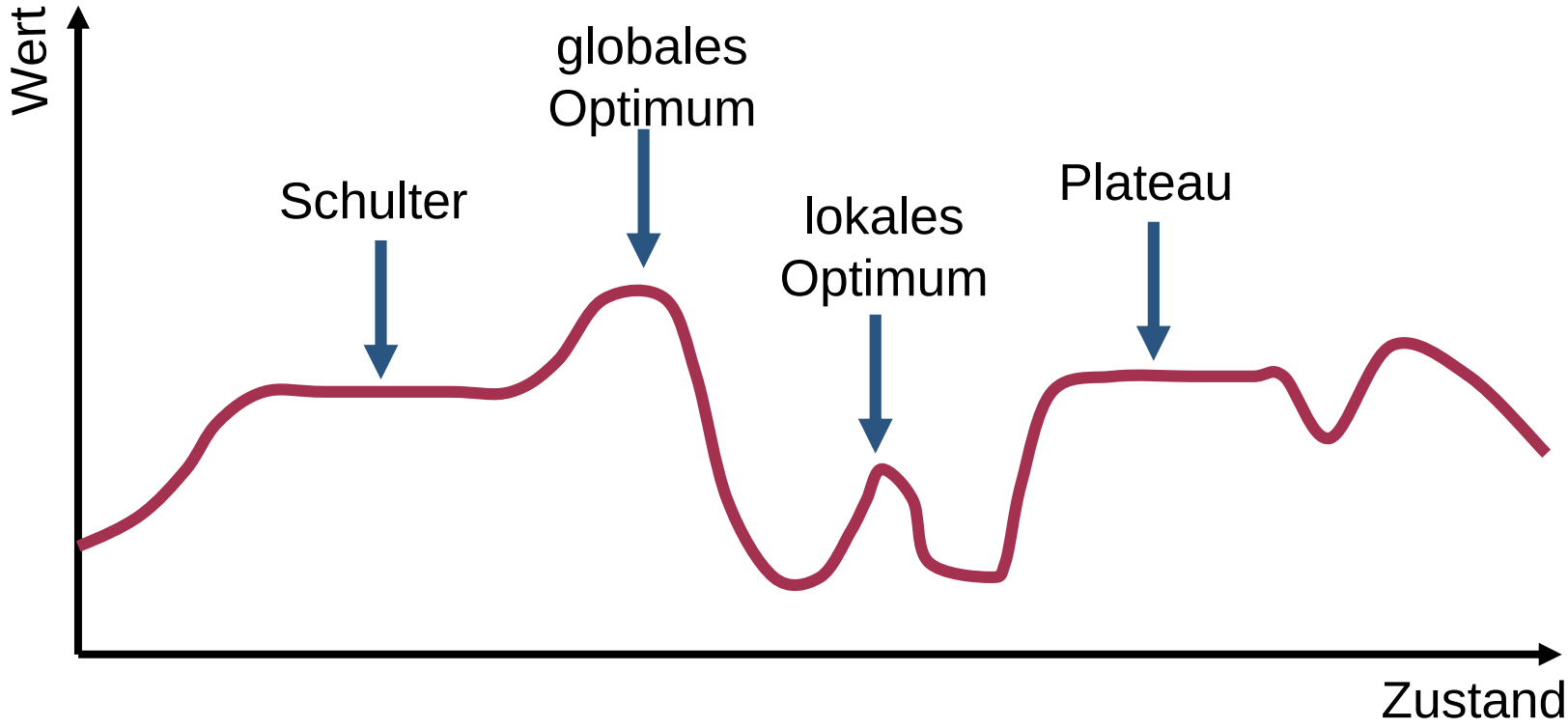
Bergsteigen für Damen



Hier keine lokale Verbesserung möglich!

(„Schulter“ der VALUE-Funktion)

Probleme auf dem Weg zu globalem Optimum



Bergsteigen findet im Allgemeinen kein globales Optimum!

➥ g.Opt. oder l.Opt. oder Schulter oder Plateau

Eigenschaften des Bergsteig-Algorithmus

Seien L die Zustände im Lösungsraum,
V_L die in L vorkommenden Bewertungen,
N die Maximalzahl von Nachbarn.

- ☺ Speicherbedarf: $O(1)$ (nur aktueller Knoten im Speicher)
  - ➥ lokales Suchverfahren
- ☺ Zeitbedarf: $O(N|V_L|)$ bei $|V_L|<\infty$ (sonst Terminierung unsicher)
- ☹ Nicht optimal / vollständig

Laufzeit ist praktisch meist sehr kurz

Wege aus dem lokalen Optimum

- Akzeptiere notfalls gleichguten Nachfolger
  - ↪ Bewältigt Schulter; Gefahr von Endlosschleife

Unterschiedliche Varianten von stochastischem Verhalten

- *Random restart*: Löse das Problem mehrfach mit zufällig gewähltem Startwert, nimm das beste Ergebnis
- *Random Walk*: Wähle zufällig einen der Nachfolger
  (merke global den bisher besten; gib den zurück bei externem Abbruch)
- *Simulated annealing*: Akzeptiere Verschlechterung abhängig von ihrem Betrag und von der Laufzeit des Algorithmus (s.u.)
- Genetische Algorithmen: Erzeuge Knoten stochastisch durch gewichtete Variation & Kombination vorhandener Knoten (s.u.)

Laufzeitabhängige Verschlechterungstoleranz

Im Prinzip Bergsteigen entsprechend VALUE-Änderung ΔE ,
aber:

- Akzeptiere auch negatives ΔE (Verschlechterung),
  aber umso seltener, je größer es ist
- Akzeptiere Verschlechterungen umso seltener,
  je höher die Laufzeit (Anzahl Durchläufe) *t* der Prozedur

Akzeptanzkriterium für negatives ΔE im Prinzip:

$$e^{t\Delta E} > \text{random}[0,1]$$

oder analog: akzeptiere Verschlechterung mit W'keit $e^{t\Delta E}$

Statt *t* verwende *s(t)* (monoton wachsend), um Einfluss von Laufzeit vs. Betrag d. Verschlechterung zu gewichten (*cooling schedule*)

Bei Ertel im Kapitel
Neuronale Netze

Simulated Annealing

anneal = ausglühen, härten, vergüten

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

schedule[*t*]≈1/*t*

stochastisches Verhalten

Eigenschaften des *Simulated Annealing*

- ☺ Speicherbedarf: $O(1)$ (nur aktueller Knoten im Speicher)
  ➥ lokales Suchverfahren
- 😐 Zeitbedarf: abhängig vom *cooling schedule*
  (praktisch: Zeit- oder Qualitäts-Schranke)
- 😐 Asymptotisch optimal in Abhängigkeit vom *cooling schedule*
  (nichttrivialer Beweis!)
- 😐 Asymptotisch vollständig in Abh. vom *cooling schedule*

- Theoretisch hängt alles an einem guten *cooling schedule*
- Praktisch ist das Verfahren robust