# Grundlagen der Künstlichen Intelligenz

## 4 Informierte Suche & lokale Suche

Heuristiken, Greedy Search, A*, Graph Search,
Hill Climbing, Simulated Annealing, genetische Algorithmen

*Volker Steinhage*

# Inhalt

- Bestensuche
  - Greedy Seach
  - A*
  - IDA*
- Lokale Suche
  - Hill Climbing
  - Simulated Annealing
  - Genetische Algorithmen

# Bestensuche

Suchverfahren unterscheiden sich durch die *Strategie* zur Auswahl des Knotens im Suchbaum, der als nächstes expandiert werden soll.

Letzte Vorlesung

- Uninformierte Suche: *starre Strategien* ohne Nutzung von problemspezifischer Information.

heutige Vorlesung

- Informierte Suche: Nutzung von problemspezifischer Information
  - über Kosten von Knoten als Station auf Weg von Start zum Ziel,
  - über *Evaluierungsfunktion f(n),* die jedem Knoten *n* eine reelle Zahl zuweist.

***Bestensuche*** (*best-first search*) als allg. Ansatz der Umsetzung der inform. Suche:

- Suchverfahren, das den Knoten mit dem besten *f*-Wert expandiert,
- wenn *f* eine Kostenfunktion ist: Expansion von Knoten mit minimalem *f*-Wert.

Voraussetzung heute: Bestensuche

# Generischer Algorithmus

**function** BEST-FIRST-SEARCH( *problem,* EVAL-FN) **returns** a solution sequence
  **inputs**: *problem*, a problem
    *Eval-Fn*, an evaluation function
  *Queueing-Fn* $\leftarrow$ a function that orders nodes by EVAL-FN
  **return** GENERAL-SEARCH( *problem, Queueing-Fn*)

$\Rightarrow$ Wenn f immer richtig ist, brauchen wir nicht zu suchen!

Vereinbarung: Bei gleichem Wert erfolgt Einordnen des neuen Knotens vor den Knoten mit demselben Wert

zur Erinnerung:

**function** GENERAL-SEARCH( *problem,*QUEUING-FN) **returns** a solution, or failure
  *nodes* $\leftarrow$ MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
**loop do**
  **if** *nodes* is empty **then return** failure
  *node* $\leftarrow$ REMOVE-FRONT(*nodes*)
  **if** GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
  *nodes* $\leftarrow$ QUEUING-FN(*nodes*, EXPAND(*node*,OPERATORS[*problem*]))
**end**

# Gierige Suche (Greedy Search)

Tatsächlich ist eine vollständige Bewertung eines Knotens i. A. *nicht* gegeben:

- Pfadkosten von *Start* zu Knoten *n* sind i .A. bekannt:

  → Pfadkostenfunktion *g(n)* wie bei uniformer Kostensuche in Kap. 3.

- Pfadkosten von Knoten *n* zu *Ziel* sind i. A. *nicht* bekannt:

  → *Schätzung* dieser Pfadkosten durch eine

  heuristische Funktion *h(n)* = *geschätzter* Abstand von *n* zum Ziel

  - *h(n)* prinzipiell beliebig, aber mit *h(n)* = 0, falls *n* Zielknoten.

- Bestensuche mit *f(n)* = *h(n)* heißt gierige Suche

  - Beispiel Routensuche: *h(n)* = Luftlinienentfernung zwischen zwei Orten

# Heuristiken

- Das Wort Heuristik ist vom griechischen Verb $\varepsilon\upsilon\rho\iota\sigma\kappa\varepsilon\iota\nu\alpha$ für „finden“ abgeleitet (vgl. auch „*eureka!*“ für „ich habe gefunden“)

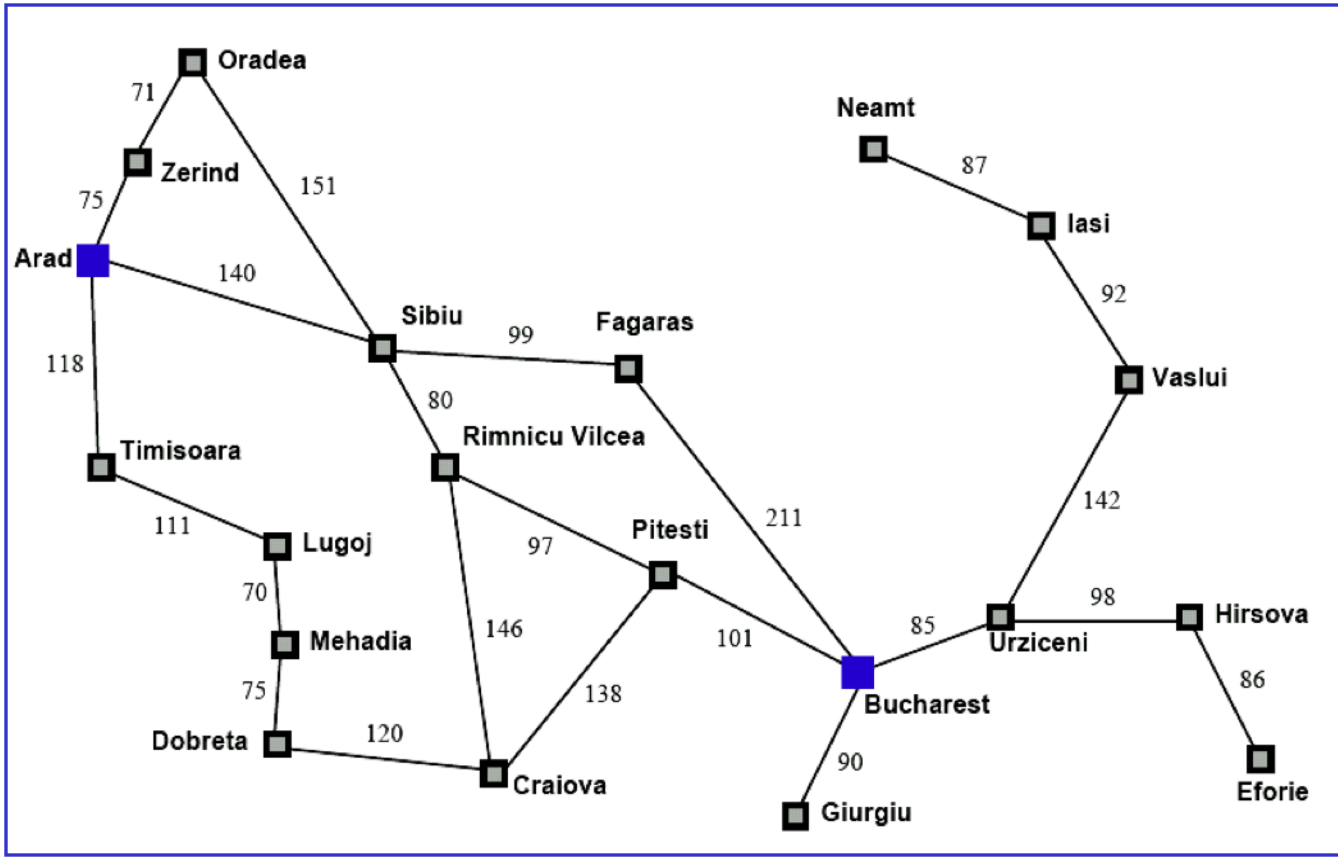
  und wurde vom Mathematiker George Polya* eingeführt, um Problem-lösungstechniken zu beschreiben.

- In der KI gibt es *zwei* Bedeutungen:
  - Heuristiken sind problemspezifische Methoden zur Beschleunigung der Suche $\rightarrow$ Verlust der Generalität.
  - Heuristiken sind schnelle, aber u.U. unvollständige Methoden, um Probleme zu lösen [Newell, Shaw, Simon 1963] (gierige Suche z.B. ist tatsächlich i. A. nicht vollständig).

$\rightarrow$ Auf jeden Fall ist eine Heuristik problemspezifisch und fokussiert die Suche!

* George (György) Pólya (* 13. Dezember 1887 in Budapest; † 7. September 1985 in Palo Alto)

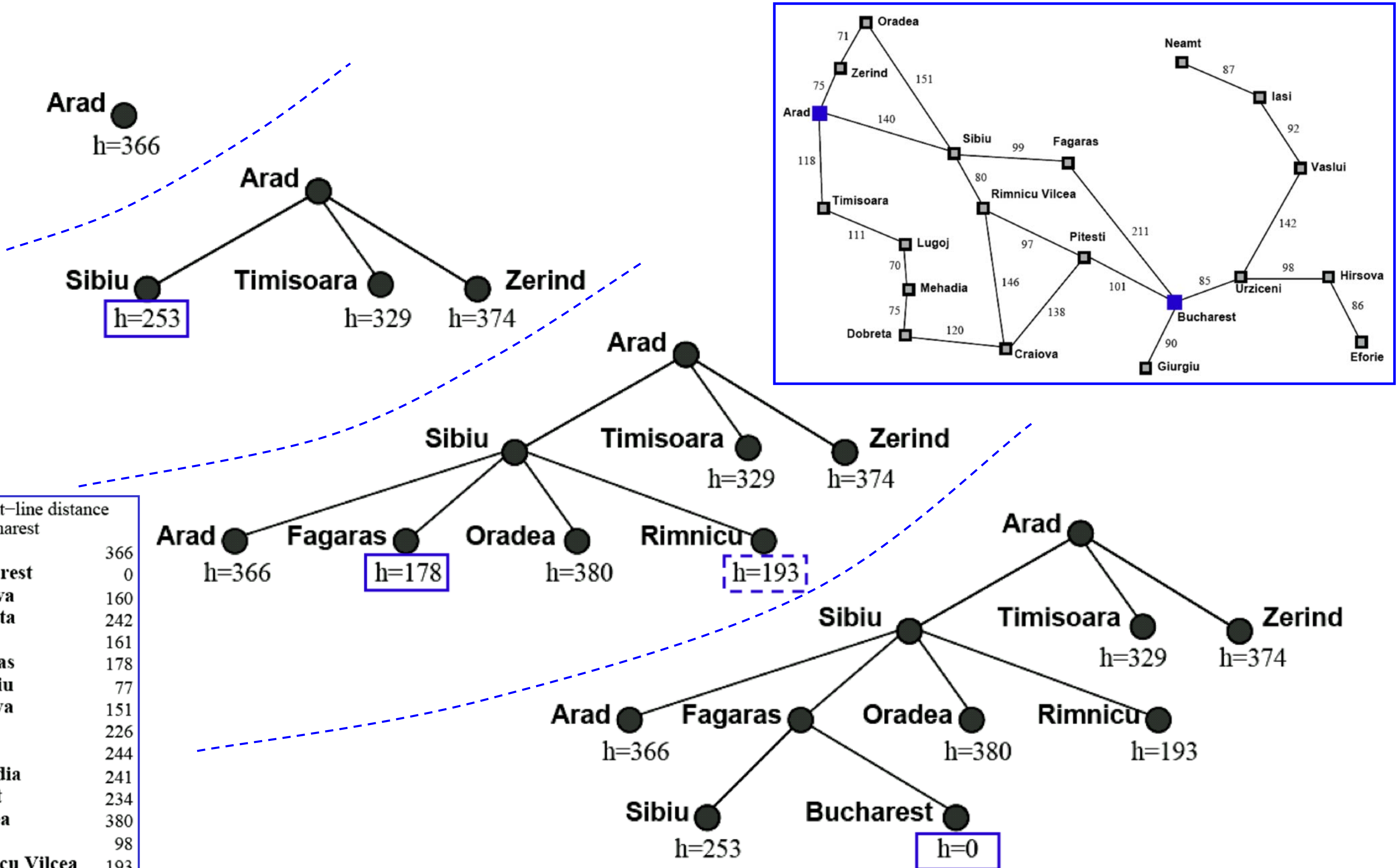
# Beispiel für gierige Suche




| Straight−line distance to 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 |

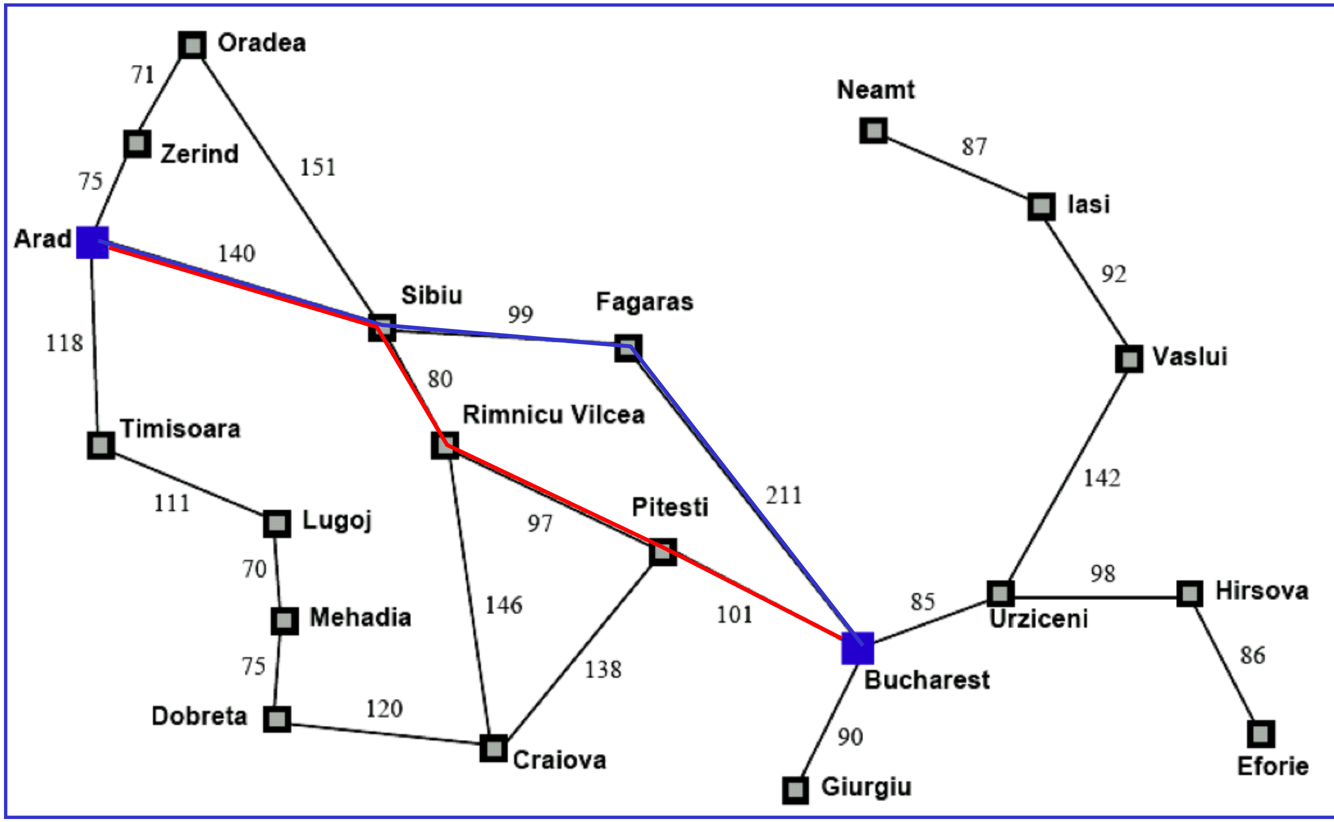
- Links: Karte Rumäniens mit Straßendistanzen in km
- Rechts: Angabe von Luftliniendistanzen nach Bukarest in km
- Heuristikfunktion $h_{LLD}(n)$ = Luftliniendistanz von Knoten *n* zum Zielknoten *Bukarest*
- Aufgabe: kürzester Weg von Arad nach Bukarest

# Gierige Suche für kürzesten Weg von *Arad* nach *Bukarest*




| Straight−line distance to 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 |

# Beispiel für gierige Suche




| Straight–line distance to 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 |

- Gierige Suche: *Arad – Sibiu – Fagaras – Bukarest* = 450 km
- aber kürzester Weg: *Arad – Sibiu – Rimnicu V. – Pitesti – Bukarest* = 418 km

→ "The strategy prefers to take the biggest bite possible out of the remaining cost to reach the goal, without worrying about wether it will be the best in the long run – hence the name *greedy search*." (Russel/Norvig 1st Ed. of Art. Intelligence)

# A*: Minimierung der gesamten Pfadkosten

A* verbindet uniforme Kostensuche mit gieriger Suche:

- *g(n)* = tatsächliche Kosten vom *Anfangszustand* bis Knoten *n*
- *h(n)* = geschätzte Kosten von Knoten *n* bis zum nächsten *Ziel*

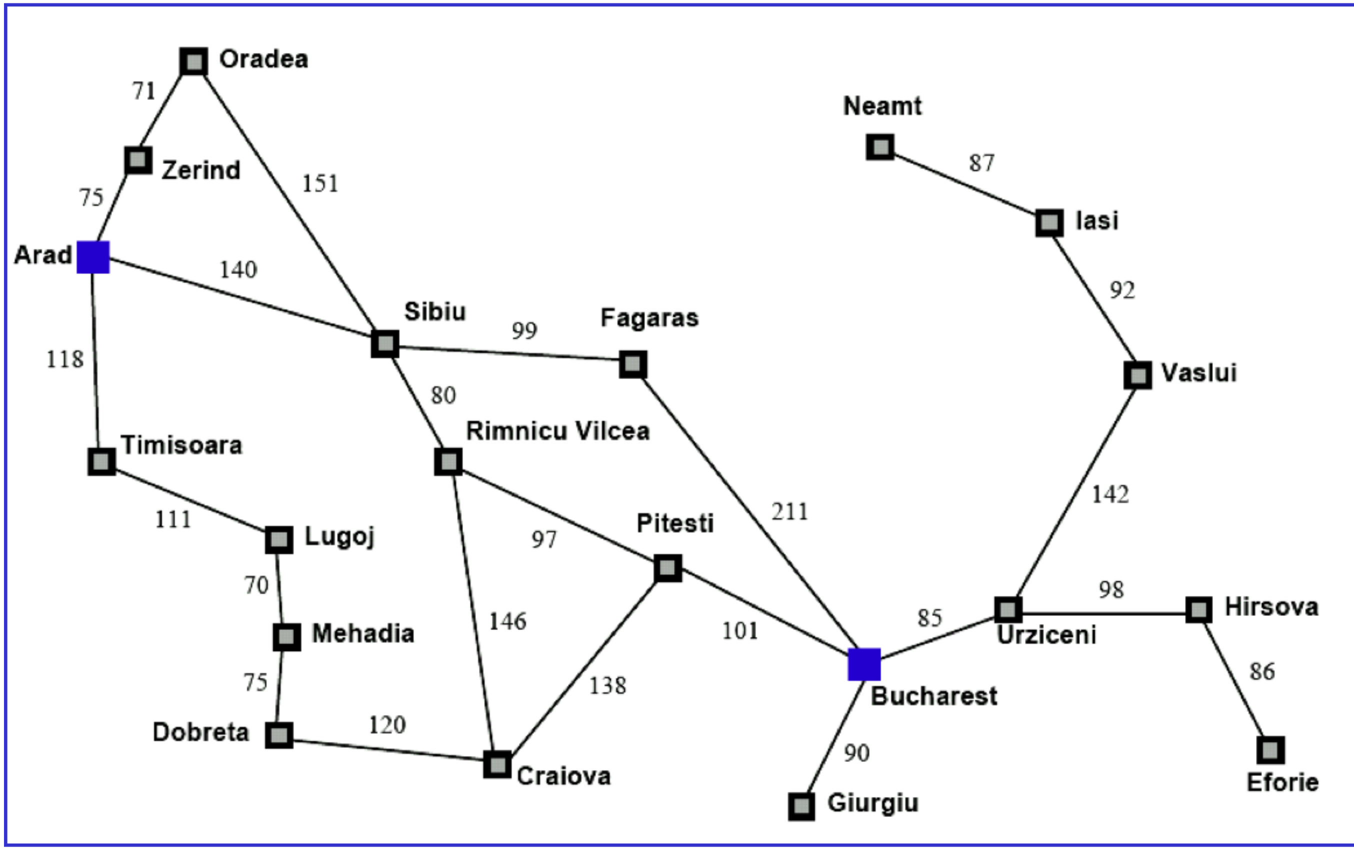
→ *f(n) = g(n) + h(n),* d.h. geschätzte Kosten des günstigsten *Gesamtpfades*, der durch Knoten n verläuft

Zulässigkeit von *h(n)* für A*:

- *Sei h*(n)* = tatsächliche Kosten des *optimalen Pfades* von *n* zum nächsten Ziel.
- Die heurist. Funktion *h* heißt zulässig, wenn für alle Knoten *n* gilt: $h(n) \leq h^*(n)$:
- Für die Optimalität von A* muss *h* zulässig sein

  → im Bspl. ist Luftliniendistanz $h_{LLD}$ zulässige Heuristik für Routensuche

# Beispiel für A*: Suche nach kürzestem Weg von *Arad* nach *Bukarest*

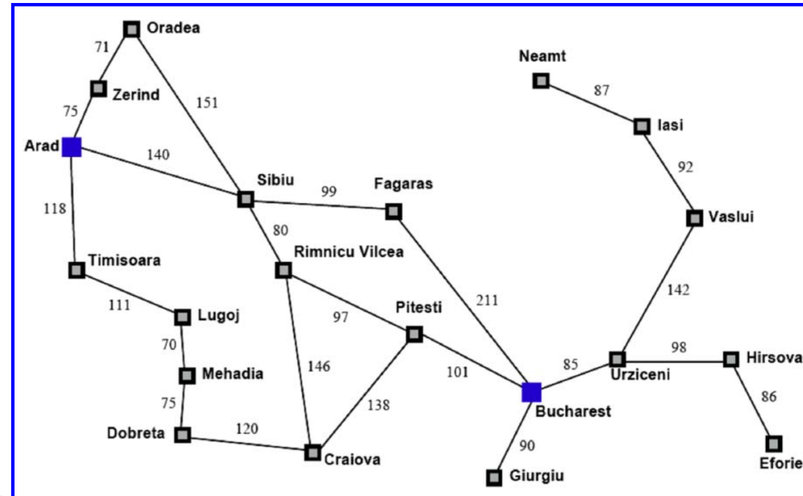


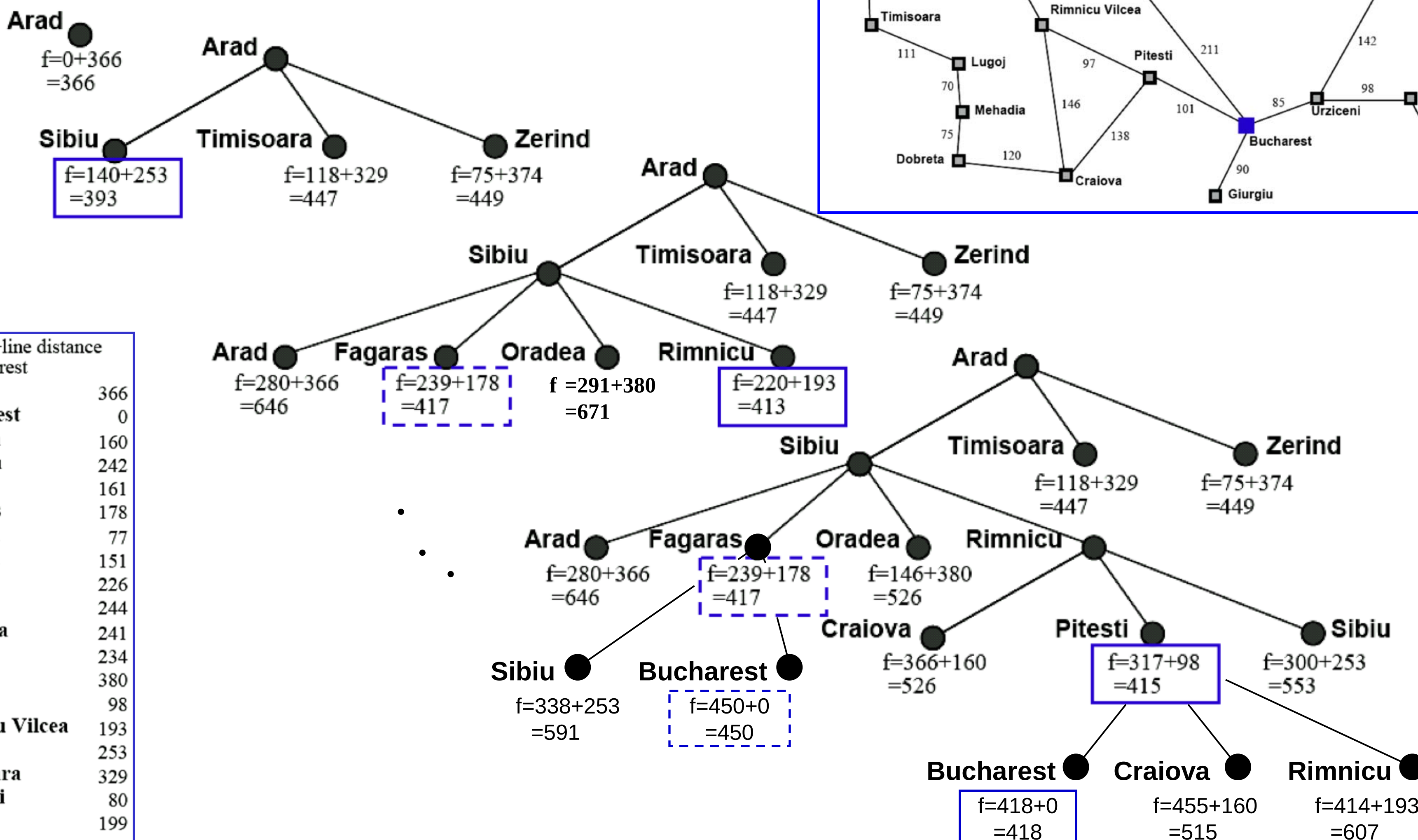

Straight−line distance to Bucharest

| | |
|---|---|
| **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 |

- Links: Karte Rumäniens mit Straßendistanzen in km
- Rechts: Angabe von Luftliniendistanzen nach Bukarest in km

# Beispiel für A*: Suche nach kürzestem Weg von *Arad* nach *Bukarest*



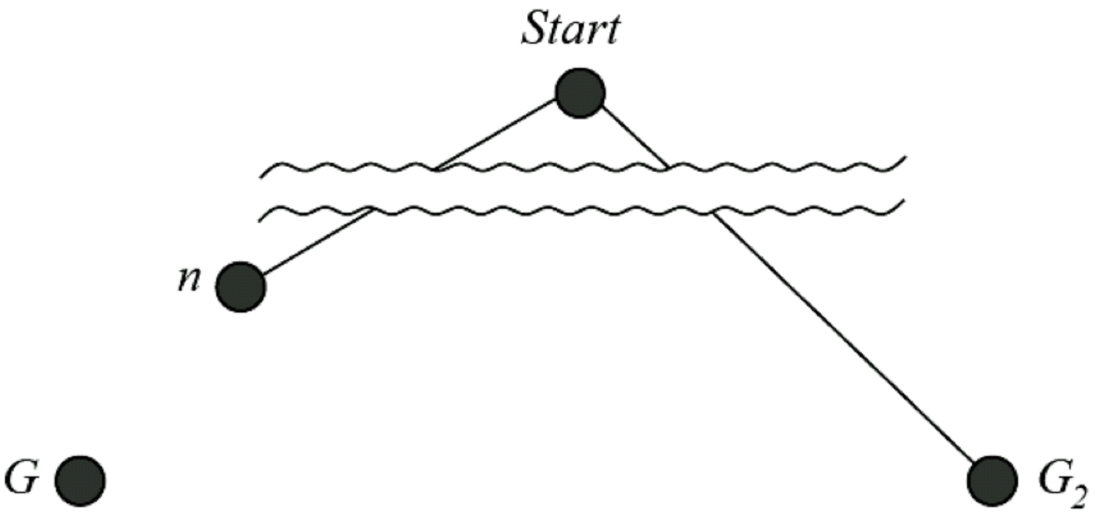

| Straight−line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 178 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 98 |
| Rimnicu Vilcea | 193 |
| Sibiu | 253 |
| Timisoara | 329 |
| Urziceni | 80 |
| Vaslui | 199 |
| Zerind | 374 |

# Optimalität von A* (1)

*Beh.:* Die <u>erste</u> von A* gefundene Lösung ist eine mit minimalen Pfadkosten.

*Indirekter Beweis:* Wir nehmen an, dass

- es einen Zielknoten $G$ mit optimalen Pfadkosten $f^*$ gibt,
- aber A* einen anderen Zielknoten $G_2$ mit $g(G_2) > f^*$ gefunden hat.



# Optimalität von A* (2)

*Beh.:* Die erste von A* gefundene Lösung ist eine mit minimalen Pfadkosten.

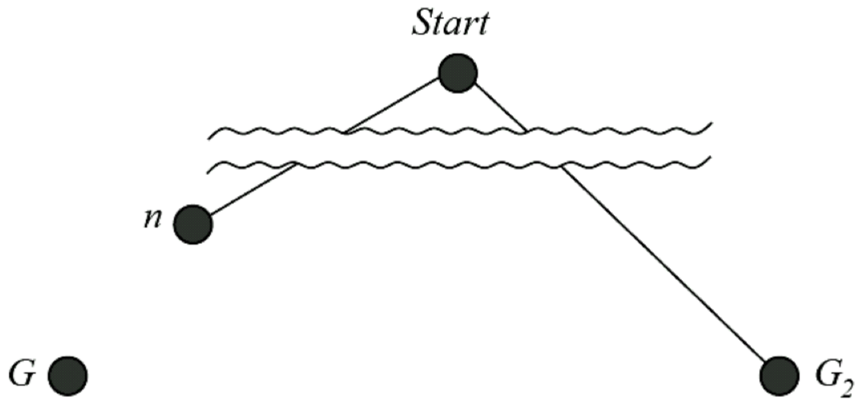
*Indirekter Beweis:* Wir nehmen an, dass

- es einen Zielknoten $G$ mit optimalen Pfadkosten $f^*$ gibt,
- aber A* einen anderen suboptimalen Zielknoten $G_2$ mit $g(G_2) > f^*$ findet.

*Bew.:* Sei *n* ein Knoten in der Liste *nodes, der* auf dem optimalen Pfad vom *Start* nach *G* liegt, aber eben *noch nicht expandiert* wurde.

- Da *h* zulässig ist, folgt: $f(n) \leq f^*$.
- Da *n* nicht vor $G_2$ expandiert wurde, muss gelten $f(G_2) \leq f(n)$ und somit $f(G_2) \leq f^*$.
- Wegen $h(G_2) = 0$ folgt daraus, dass $g(G_2) \leq f^*$.

→ Widerspruch zur Annahme!



# Vollständigkeit und Komplexität von A*

*Vollständigkeit:* Wenn eine Lösung existiert, findet A* diese, sofern

- jeder Knoten nur endlich viele Nachfolgerknoten hat und
- positive Konstante *d* exist., so dass jeder Operator mindest. die Kosten *d* hat.

  $\rightarrow$ nur endlich viele Knoten $n$ mit $f(n) \leq f^*$.

---

*Komplexität* (*worst case* – i.A. deutlich besser bei guten Heuristiken!):

- Sofern der Fehler der Heuristik maximal logarithmisch mit den tatsächlichen Pfadkosten wächst, also $|h^*(n) - h(n)| \leq O(\log h^*(n))$, werden nur subexponentiell viele Knoten expandiert.
- Wächst der Fehler proportional zu den Pfadkosten, liegt – im worst case – exponentielles Wachstum mit zunehmender Ableitungstiefe vor.

# A* und Monotonie (1)

Im Beispiel der A*-Suche von Arad nach Bukarest gilt für alle Pfade:

$f(n') \geq f(n)$, wenn $n'$ Nachfolgeknoten von $n$

Allgemein gilt für Nachfolgeknoten $n'$ von $n$ in einem Pfad:

- Eine Heuristik $h(n)$ heißt monoton, gdw. $f(n') \geq f(n)$ für alle Pfade
- oder: Eine Heuristik $h(n)$ heißt monoton, gdw. $h(n') + c(n,a,n') \geq h(n)$, für alle Pfade mit Kosten $c$ einer Aktion $a$, die von $n$ zu $n'$ führt

# Suchbäume und Wiederholungen

Bislang Suchtechniken, welche einen explizten Suchbaum verwenden!

Dabei bislang Ignoranz bzgl. wiederholt erzeugter Zustände

- Bei einigen Problemformulierungen treten keine Wiederholungen auf: z.B. 8-Damen-Problem mit Formulierung durch spaltenweise Belegung $\Rightarrow$ jeder Zustand ist nur über einen Pfad erreichbar.
- Insbes. bei Problemen mit *umkehrbaren* Operationen sind Wiederholungen unvermeidbar: z.B. Schiebeblock-Puzzle, Routenplanung.

Wiederholbare Zustände können ein *lösbares Problem unlösbar machen*, indem

- auch bei endlichen Wiederholungen begrenzte *Zeit- oder Speicherressourcen überschritten werden,*
- *Endlosschleifen* bei unvollständigen Strategien wie z.B. Tiefensuche die Lösung unmöglich machen.

# Erkennung von Wiederholungen

- Klar: Erkennung von Wiederholungen durch Vergleich der Zustände der zu expandierenden Knoten mit denen bereits expandierter Knoten!

⇝ Bei Erkennung einer Gleichheit liegen zwei verschiedene Pfade zu ein und demselben Zustand vor. Ein Pfad kann verworfen werden.

Algorithmisch:

- Bislang führte *General-Search* für die Suche im Suchbaum eine Liste *nodes*, die alle zu expandierenden Knoten nach entsprechender Strategie geordnet speicherte und aktualisierte.
- Die Erweiterung *Graph-Search* hält eine weitere Liste *closed*, die die Zustände aller bereits expandierten Knoten speichert und damit Zyklen vermeidet.

# Graph-Search vs. General Search

**function** GRAPH-SEARCH (*problem*, QUEUING-FN) returns a solution, or a failure
    *closed* ← initially empty
    *nodes* ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
**loop do**
    **if** *nodes* is empty **then return** failure
    *node* ← REMOVE-FRONT(*nodes*)
    **if** GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
    **if** STATE(*node*) is not in *closed* **then**
        add STATE**[***node***]** to *closed*
        *nodes* **←** QUEUING-FN(*nodes*,EXPAND(*node*,OPERATORS[*problem*]))
**end**

**function** GENERAL-SEARCH( *problem,* QUEUING-FN) **returns** a solution, or failure
    *nodes* ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
**loop do**
    **if** *nodes* is empty **then return** failure
    *node* ← REMOVE-FRONT(*nodes*)
    **if** GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
    *nodes* ← QUEUING-FN(*nodes*, EXPAND(*node*,OPERATORS[*problem*]))
**end**

# Optimalität von Graph-Search

```
function GRAPH-SEARCH (problem, QUEUING-FN) returns a solution, or a failure
    closed ← initially empty
    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    if STATE(node) is not in closed then
        add STATE[node] to closed
        nodes ← QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
end
```

- Algm. *Graph-Search* verwirft immer den zuletzt entdeckten Pfad und ist daher nicht optimal, wenn dieser der günstigere ist.
- Optimal ist *Graph-Search*, wenn die beiden optimalen Strategien der uniformen Kostensuche oder der Breitensuche mit konst. Schrittkosten verwendet werden.
- Iterative Tiefensuche benutzt Tiefensuche und kann daher einem suboptimalen Pfad zu einem Zustand folgen. Daher muss *Graph-Search* testen, ob der neue Pfad zum selben Zustand günstiger ist und entsprechend Tiefe und Pfadkosten korrigieren.

# Komplexität von Graph-Search

**function** GRAPH-SEARCH (*problem*, QUEUING-FN) returns a solution, or a failure
    *closed* ← initially empty
    *nodes* ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
**loop do**
    **if** *nodes* is empty **then return** failure
    *node* ← REMOVE-FRONT(*nodes*)
    **if** GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
    **if** STATE(*node*) is not in *closed* **then**
        add STATE**[***node***]** to *closed*
        *nodes* ← QUEUING-FN(*nodes*,EXPAND(*node*,OPERATORS[*problem*]))
**end**

- *Graph-Search* merkt sich jeden Zustand ⇝ *Graph-Search* erkundet letztlich den Zustandsgraphen (⇝ Name). Bei Problemen mit vielen Wiederholungen ist *Graph-Search* viel effizienter als die mit *General Search* realisierte Baumsuche. Worst-Case-Zeit und die Speicheranforderungen sind proportional zur Größe des Zustandsraumes, was deutlich kleiner sein *kann* als $O(b^d)$.
- Der Einsatz von *closed* bedeutet aber, dass Tiefensuche und iterative Tiefensuche nicht mehr lineare Speicheranforderungen $O(b \cdot m)$ bzw. $O(b \cdot d)$ haben.

# A* und Graph-Search

```
function GRAPH-SEARCH (problem, QUEUING-FN) returns a solution, or a failure
    closed ← initially empty
    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    if STATE(node) is not in closed then
        add STATE[node] to closed
        nodes ← QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
end
```

- A* in *Graph-Search* gewährleistet auch bei zulässiger Heuristik *h(n)* nicht die Optimalität, da der zweite verworfene Pfad der günstigere sein kann.

- Erst die *Monotonie* von *h(n)* gewährleistet die Optimalität von *Graph-Search* , da der optimale Pfad in jedem wiederholten Zustand immer der erste ist, der verfolgt wird.

# A* und Monotonie (2)

Bespiel:

- Eine Heuristik *h(n)* sei zulässig, aber nicht monoton:
- *n‘* sei Nachfolgeknoten von *n* mit
  - $g(n) = 3$, $h(n) = 4$ ↝ $f(n) = 7 \leq f^*$ ,
  - $g(n') = 4$, $h(n') = 2$ ↝ $f(n') = 6 \leq f^*$ ↝ $f(n') \geq f(n)$ gilt nicht!

$f^*$ = wahre optimale Pfadkosten von Start zu Ziel

Da *im Suchbaum* jeder Pfad durch *n‘* auch durch *n* geht und *h*(*n*) zulässig:

↝ $f(n) = 7 \leq f^*$

↝ $f(n') = 7$ ist auch für $f(n')$ keine Überschätzung von $f^*$.

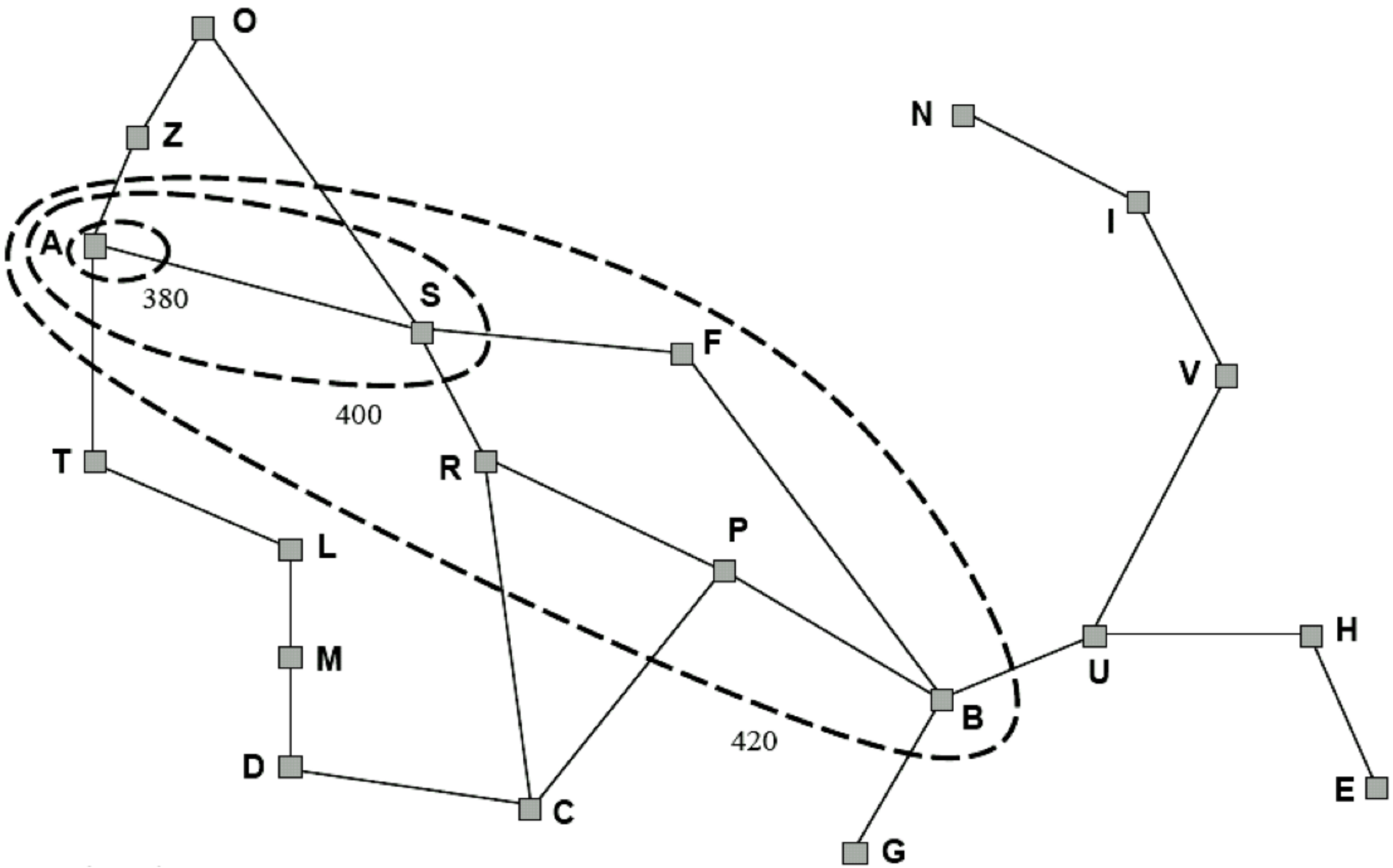
Somit ist Monotonie erzielbar durch folg. Wahl (PathMax-Gleichung):

$$f(n') = max(f(n), g(n') + h(n')).$$

↝ Monotone *f*-Kosten lassen graphische Darstellung des Vorgehens von A* durch *Konturen* zu.

# Konturen in A* (1)

Bei nicht fallenden f-Kosten, sind im Suchraum *Konturen* mit *f*-Werten darstellbar. Eine Kontur ist mit einem f-Grenzwert assoziiert. Die Kontur umfasst alle Knoten, deren *f*-Werte kleiner oder gleich dem Grenzwert der Kontur sind:

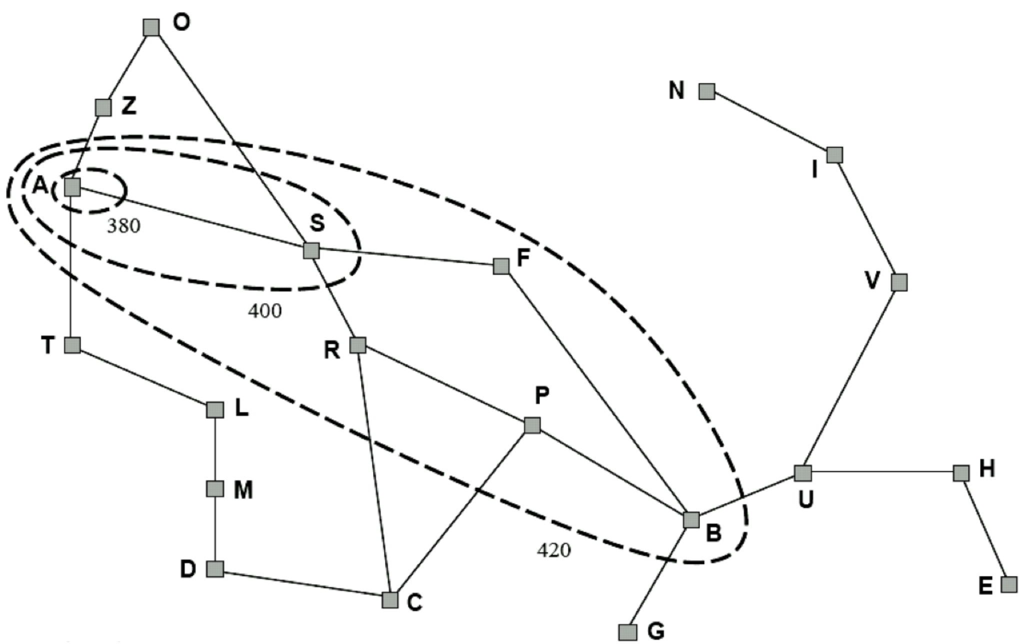



Konturen für *f* = 380, 400, 420

# Konturen in A* (2)

A* expandiert ausgehend von Startknoten immer Blattknoten mit niedrigstem *f(n)*

- Knotenexpansion erfolgt in konzentrischen Bändern mit wachsendem *f(n)*
- bei *uniformer Kostensuche* (also *h(n) = 0*) eher kreisrunde Bänder
- bei *guter Heuristik h*(*n*) schmalere Bänder um optimalen Lösungspfad mit Orientierung zum Ziel.

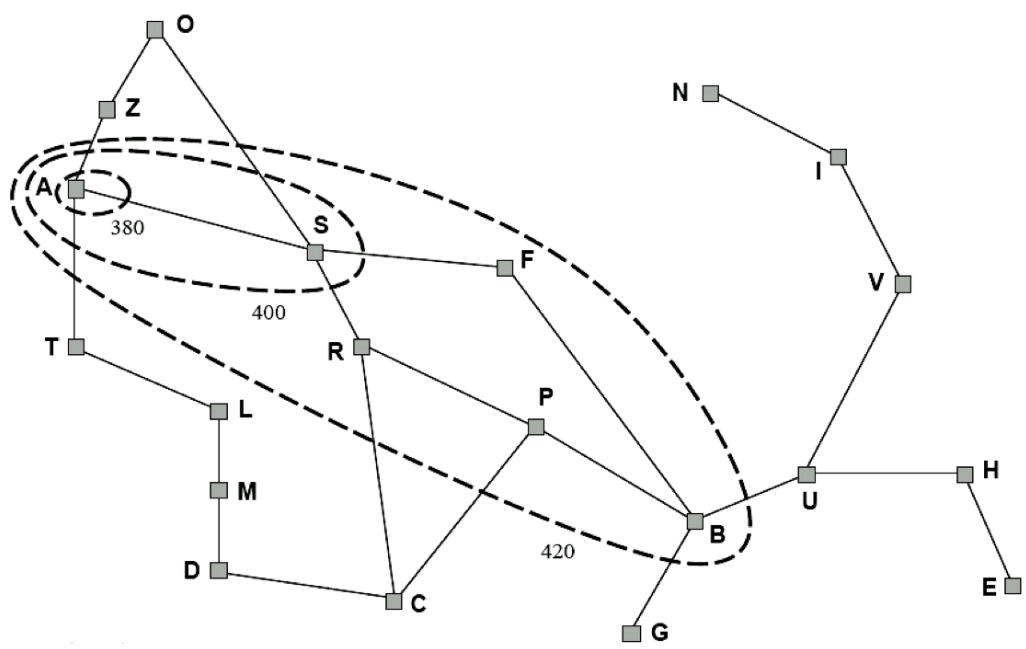


# Konturen in A* (3)

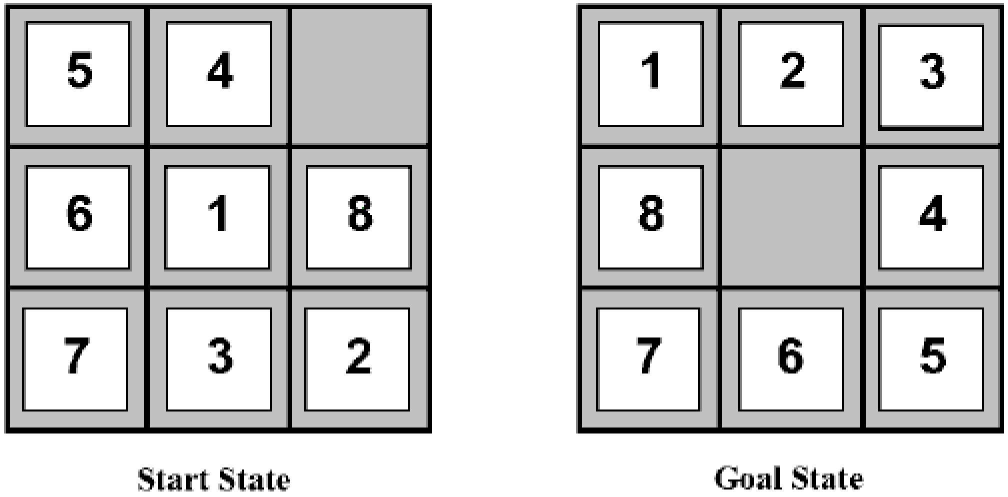
Bezeichne *f** die Kosten des optimalen Lösungspfades, dann gilt:

- A* expandiert alle Knoten mit f(n) < f* ,
- A* expandiert ggf. einige Knoten auf der „Zielkontur“ (*f*(*n*) = *f**) vor Auswahl des Zielknotens,

→ da aufeinander folgende Konturen höhere *f*- und *g*-Kosten haben, ist die erste Lösung eine optimale Lösung!



# Vergleich von heuristischen Funktionen




- $h_1$ = Zahl der Felder in falscher Position

  $h_1$ ist zulässig, $h_1(Start) = 7$

- $h_2$ = Summe der Distanzen der Felder zur Zielposition (*Manhattan-Distanz*)

  $h_2$ ist zulässig, $h_2(Start) = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$

- Heuristik $h_2$ dominiert Heuristik $h_1$: $h_2(n) \geq h_1(n)$ für alle $n$

# Empirische Bewertung der heuristischen Genauigkeit

Messung über *effektiven Verzweigungsfaktor b** (*effective branching factor*):

- Sei *N* die Zahl erzeugter Knoten bei Lösungstiefe *d,*

- Dann ist *b** ist der Verzweigungsfaktor, den ein *einheitlicher* Baum der Tiefe d bräuchte, um N+1 Knoten aufzunehmen:

$$N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d.$$

- Für gute Heuristiken liegt *b** nahe bei 1. Damit ist die Lösung von relativ großen Probleminstanzen möglich.

# Empir. Auswertung im Vergleich mit *uninform. iterativer Tiefensuche*

Geg.: Schiebeblock-Puzzle mit Heuristiken $h_1$ (# falsche Positionen) und $h_2$ (Manhattan-Distanzen zu richtigen Positionen).

Evaluierung: 1.200 zufällige Probleme mit geraden Lösungslängen *d* von 2 bis 24 gleich verteilt in 100 Problemen pro Lösungslänge d.

Bewertungsgrößen: (1) Anzahl expandierter Knoten (Suchkosten), (2) eff. Verzweigungsfaktor

| | Search Cost | | | Effective Branching Factor | | |
|---|---|---|---|---|---|---|
| *d* | IDS [1] | A*($h_1$) | A*($h_2$) | IDS | A*($h_1$) | A*($h_2$) |
| 2 | 10 | 6 | 6 | 2.45 | 1.79 | 1.79 |
| 4 | 112 | 13 | 12 | 2.87 | 1.48 | 1.45 |
| 6 | 680 | 20 | 18 | 2.73 | 1.34 | 1.30 |
| 8 | 6384 | 39 | 25 | 2.80 | 1.33 | 1.24 |
| 10 | 47127 | 93 | 39 | 2.79 | 1.38 | 1.22 |
| 12 | 364404 | 227 | 73 | 2.78 | 1.42 | 1.24 |
| 14 | 3473941 | 539 | 113 | 2.83 | 1.44 | 1.23 |
| 16 | – | 1301 | 211 | – | 1.45 | 1.25 |
| 18 | – | 3056 | 363 | – | 1.46 | 1.26 |
| 20 | – | 7276 | 676 | – | 1.47 | 1.27 |
| 22 | – | 18094 | 1219 | – | 1.48 | 1.28 |
| 24 | – | 39135 | 1641 | – | 1.48 | 1.26 |

[1] IDS = Iterative deepening depth-first search

# Speicherbegrenzte heuristische Suche: IDA*

- Problem: A* muss alle erzeugten Knoten im Speicher halten und kann bei hohen Problemgrößen am Speicheraufwand scheitern.
- Motivation: *Iterative Tiefensuche* reduziert den Bedarf an Speicherplatz beträchtlich, ohne dass die Laufzeit übermäßig erhöht wird.

→ Idee der Kombination: *Iterative A*-Tiefensuche* (*Iterative-Deepening-A** - IDA*) als Variante mit *f*-Kostenschranken anstelle von Tiefenschranken.

- *IDA** ist vollständig und optimal mit denselben Bedingungen wie A*, zeigt aber Speicheranforderungen, die nur proportional zu dem längsten explorierten Pfad sind:

  Mit $k_{min}$ für die minimalen Aktionskosten, $b$ für den *mittleren* Verzweigungs-faktor und $f^*$ für die opt. Lösungskosten benötigt IDA* Speicher für maximal $b \cdot f^*/k_{min}$ Knoten.

# Vorteile von IDA*

- *Iterative A*-Tiefensuche* reduziert den Bedarf an Speicherplatz beträchtlich, ohne dass die Laufzeit übermäßig erhöht wird.

- *Aber: Zeitkomplexität* erhöht sich beträchtlich, wenn die Bewertungsfunktion *f viele verschiedene Werte* annehmen kann $\Rightarrow$ häufiger „Neustart“

$\rightarrow$ Lösung durch sog. $\varepsilon$ -Zulässigkeit:

- In jeder Iteration Erhöhung von *f-limit* um einen festen Wert $\varepsilon$

$\rightarrow$ Anzahl der Iterationen nur noch prop. zu $1/\varepsilon$

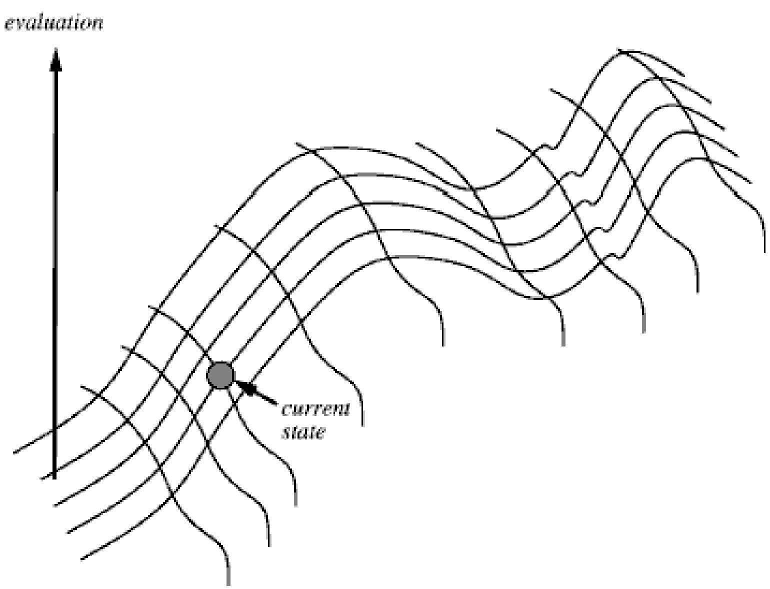
$\rightarrow$ Lösungen können nur um Betrag $\varepsilon$ schlechter als das Optimum sein

Ein solcher Algorithmus heißt dann $\varepsilon$ -zulässig.

- *IDA** war viele Jahre der einzig bekannte, optimale, heuristische Algorithmus mit Speicherbeschränkung.

# Lokale Suche

- Für viele Probleme ist es *irrelevant*, *wie* man zum Zielzustand kommt – nur der Zielzustand selber ist interessant (z. B. 8-Damen Problem, VLSI Design, TSP).
- Wenn sich außerdem ein Qualitätsmaß für Zustände angeben lässt, kann man lokale Suche benutzen, um Lösungen zu finden.
- Lösung ist also nicht mehr die Aktionsfolge zur Erreichung des Zielzustandes, sondern „nur noch“ der Zielzustand selbst.




- Idee: Man fängt mit einer zufällig gewählten Konfiguration an und verbessert diese schrittweise.

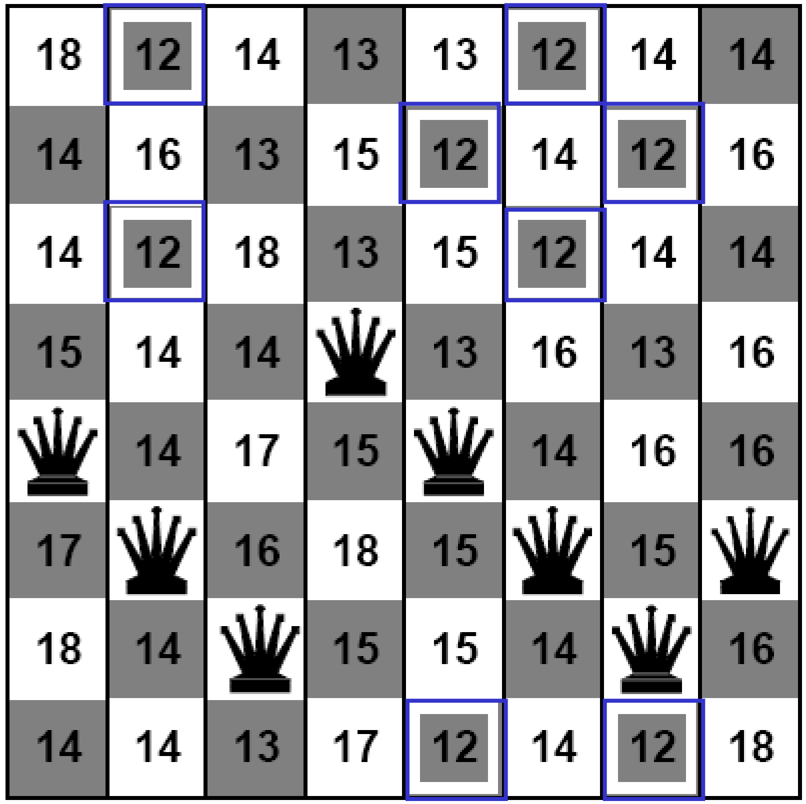
  ⇝ Hill Climbing bei positiver Nutzenbewertung.

# Hill Climbing

```
function HILL-CLIMBING( problem) returns a solution state
    inputs: problem, a problem
    static: current, a node
            next, a node
    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
            next ← a highest-valued successor of current
            if VALUE[next] < VALUE[current] then return  STATE(current)
            current ← next
    end
```

Bemerkung: Bei Zustandsbewertung durch Kosten ist eine Kostenminimierung das Ziel und entsprechend der „Talabstieg“ zu wählen.

# Beispiel: 8-Damen-Problem (1)

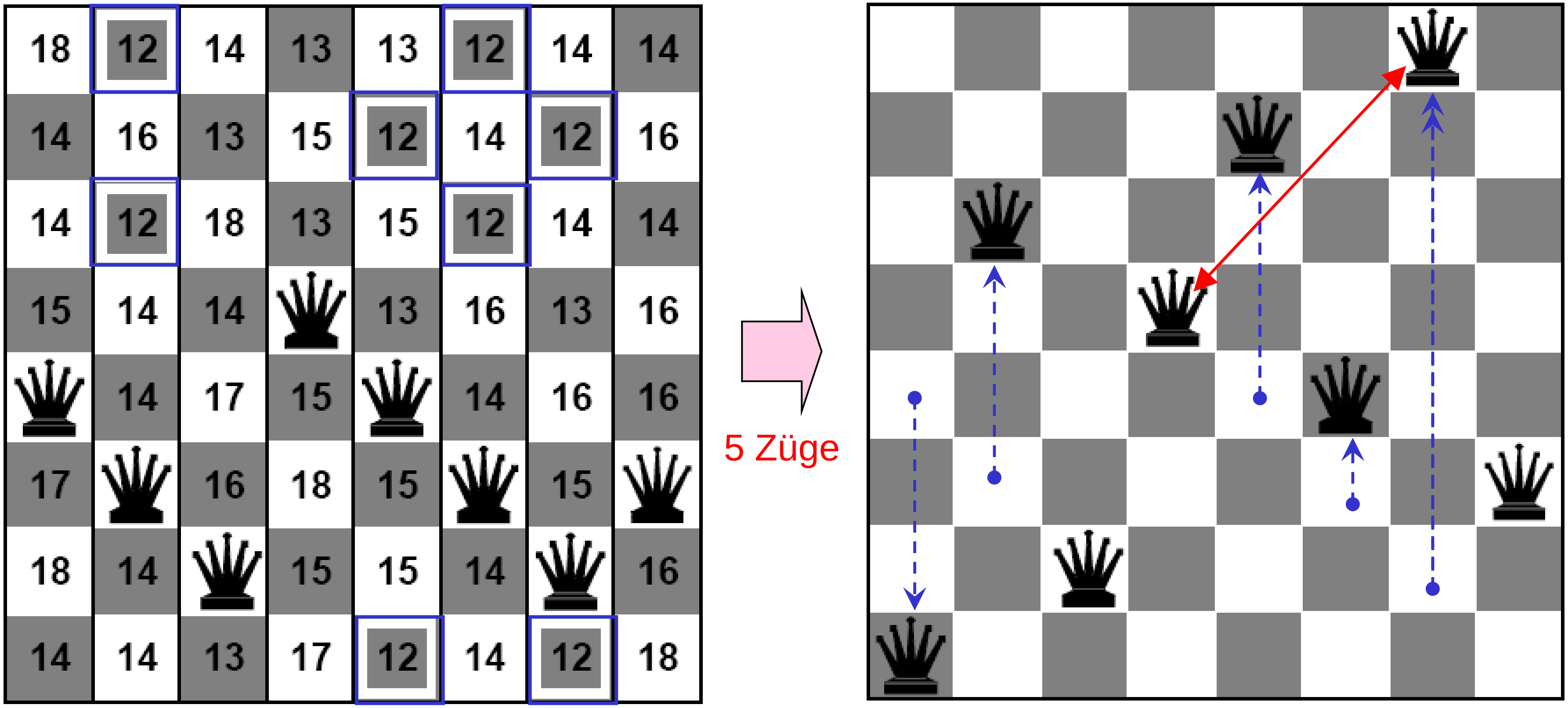



Hier *„valley descending“* – wir sprechen aber in der allg. Diskussion weiterhin von *„hill climbing“*

- Heurist. Funktion *h*: Zahl der *Damenpaare*, die sich bedrohen (in Abb.: h=17).
- Dargestellt sind auch die h-Werte von mögl. *Nachfolgezuständen*, die durch das Verschieben der Damen *innerhalb ihrer Spalten* erreicht werden.
- Hill-Climbing wählt (i. A. zufällig) einen der besten Nachfolgezustände,

⇝ hier mit h=12.

# Beispiel: 8-Damen-Problem (2)




Problem: *Hill-Climbing* kann in einem *lokalen Optimum* stecken bleiben:

- Hier lokales Optimum nach 5 Zügen aus Zustand in letzter Folie mit h=1.
- ***Jeder*** weitere Spaltenzug liefert mind. einen zusätzlichen Konflikt und damit einen schlechteren h-Wert.
- Also keine Verbesserung möglich … und *keine Lösung* gefunden!

# Probleme bei lokaler Suche

- Lokale Maxima: Der Algorithmus gibt eine suboptimale Lösung aus.
- Plateaus: Hier kann der Algorithmus nur zufällig herumwandern.
- Grate und Sattelpunkte: Ähnlich wie Plateaus.

Lösungen:

- Neustarts, wenn keine Verbesserung mehr erzielt wird.
- Rauschen „injizieren“ (Random walk).
- Tabu-Suche: die letzten *n* angewandten Operatoren nicht anwenden.

Welche Strategien (mit welchen Parametern) erfolgreich sind (auf einer Problemklasse), ist meist nur empirisch zu bestimmen.

# Simuliertes Abkühlen

Bei *Simulated-Annealing* erfolgt das „Injizieren“ von Rauschen systematisch: erst stark, dann abnehmend.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   static: current, a node
           next, a node
           T, a "temperature" controlling the probability 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^(ΔE/T)
```

„*Abkühlungsplan*“

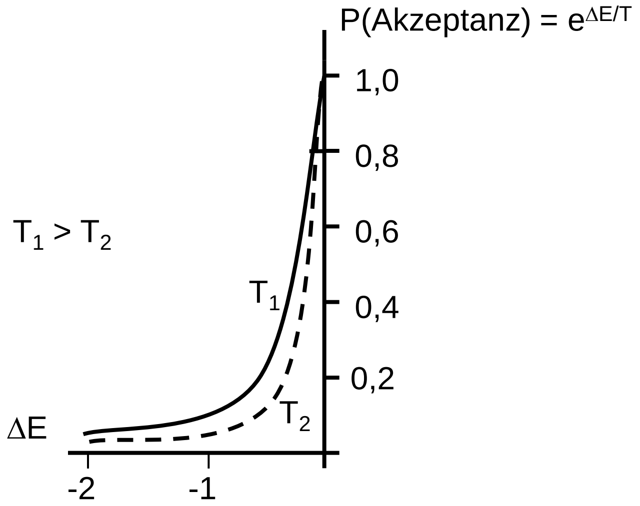
Zufällige Wahl des Nachfolgezustands *next*

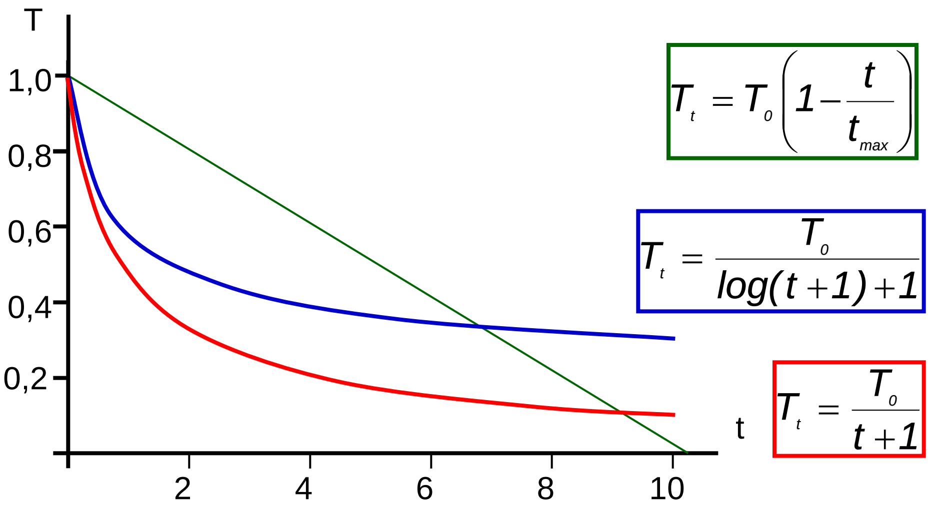
„*Energiedifferenz*“ ΔE

Val.[next] > Val.[current] ↝ ΔE > 0 ↝ Akzeptanz von *next,* andernfalls nur probabil. Akzeptanz von *next.*

Einsatz seit 1983 für VLSI Layout und andere Optimierungsprobleme.

# Akzeptanz-Rate und Annealing Schedules

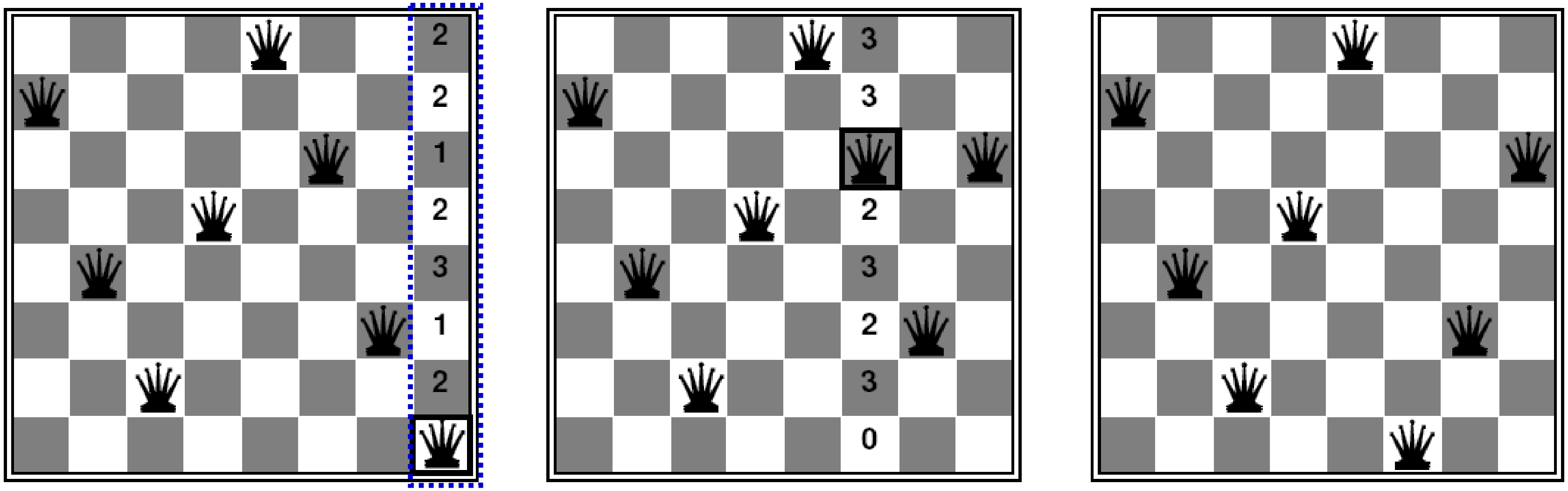



- Akzeptanz-Wahrscheinlichkeit kontrolliert Stochastizität der Suche:
  - verschlechternde Änderungen werden mit Wahrscheinlichkeit < 1 akzeptiert,
  - Wahrscheinlichkeit der Akzeptanz sinkt exponentiell mit $\Delta$E.
- Temperaturfunktionen bestimmen die Qualität der Lösungen:
  - logarithmisch langsames Abkühlen führt mit Wahrscheinlichkeit 1 zum globalen Minimum $\rightarrow$ erschöpfende Suche!

# Beispiel: Heuristisches Reparieren beim 8-Damen Problem

- Allg. steht *Heuristisches Reparieren* (*HR*) für Verfahren, die die Zahl der erfüllten Bedingungen einer Problemstellung maximieren bzw. die Zahl der entsprechenden Konflikte minimieren
- Start: Zufällige oder möglichst konsistente Startpositionierung aller Damen.
- *HR*: iteratives Umsetzen von Damen in Konfliktpositionen auf neue Positionen mit minimaler Zahl neuer Konflikte → Min-Conflicts-Heuristik.
- *HR* erlaubt Lösung von 1-Millionen-Damen-Problem in ca. 50 Schritten!!!




Lösung des 8-Damen-Problems mit Min-Conflicts-Heuristik in zwei Schritten!

# Genetische Algorithmen (GA)

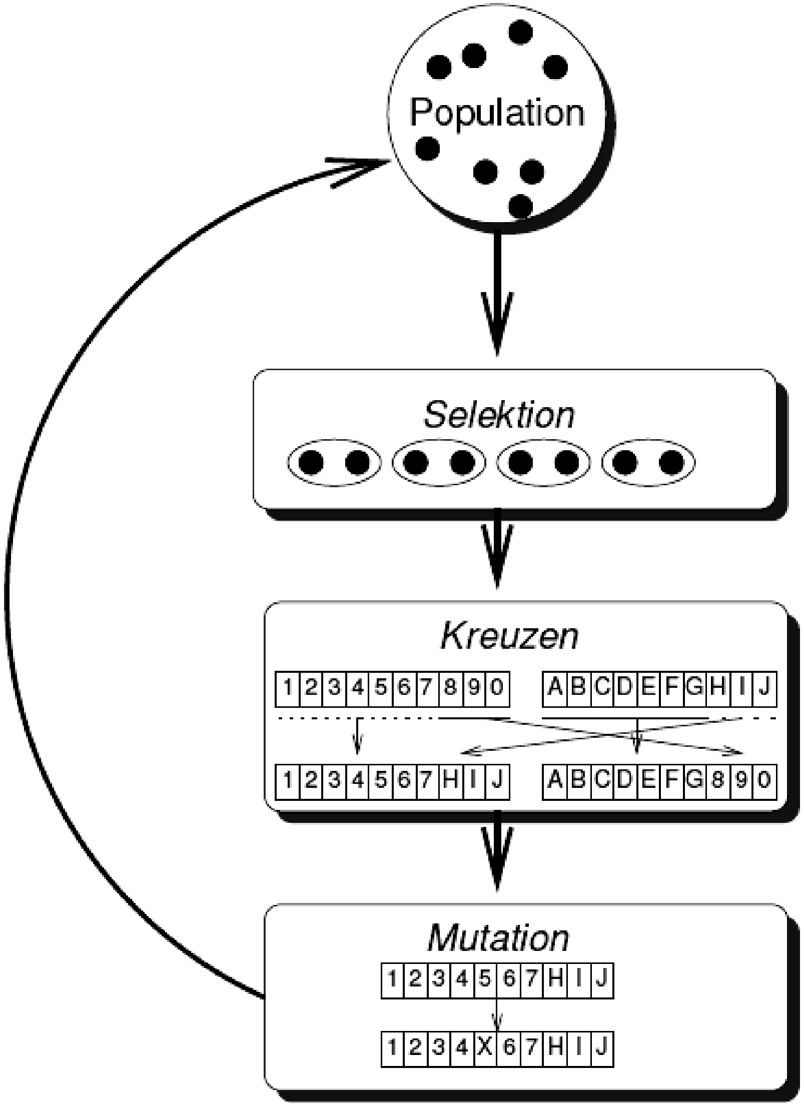
Die Evolution scheint sehr erfolgreich zu sein, gute Lösungen zu produzieren.

Idee: Ähnlich wie bei der Evolution, sucht man Lösungen, indem man erfolgreiche Lösungen „kreuzt", „mutiert" und „selektiert".

Ingredienzen:

- Kodierung von Konfigurationen als Zeichenketten oder Bit-Strings.
- „Fitness"-Funktion, welche die Güte von Konfigurationen beurteilt.
- Population von Konfigurationen

# Selektieren, Kreuzen und Mutieren




Selektion von Individuen anhand der Fitness-Funktion und Paarung

Festlegung wo aufgebrochen wird und neu zusammenfügen

Mit einer gewissen kleinen Wahrscheinlichkeit wird etwas im String geändert.

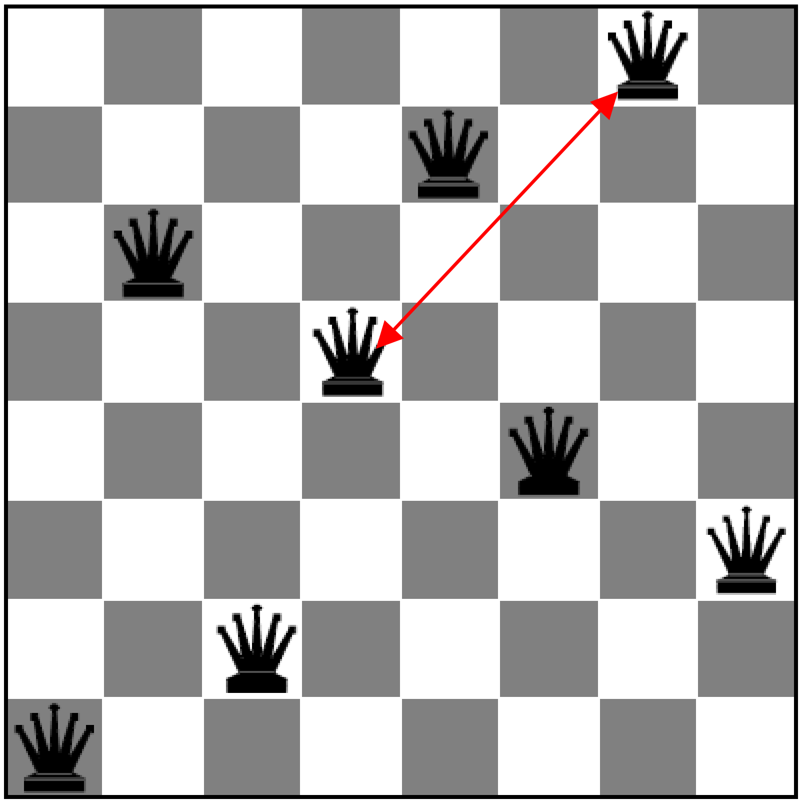
Viele Variationen: Wie wird Selektion angewandt, welche Art der Kreuzung wird benutzt, usw.

Wird z.B. erfolgreich für das Erstellen von Phantombildern eingesetzt. Fitness wird durch Zeugen manuell bestimmt.

# GA für 8-Damen-Problem: Kodierung durch Strings

Beispiel:

- 8-Damen Problem kodiert als Zeichenkette (String) von 8 Zahlen.
- Fitness berechnet sich aus der Anzahl der *paarweisen Nichtangriffe*.
- Population besteht aus einer Menge von Anordnungen der acht Damen



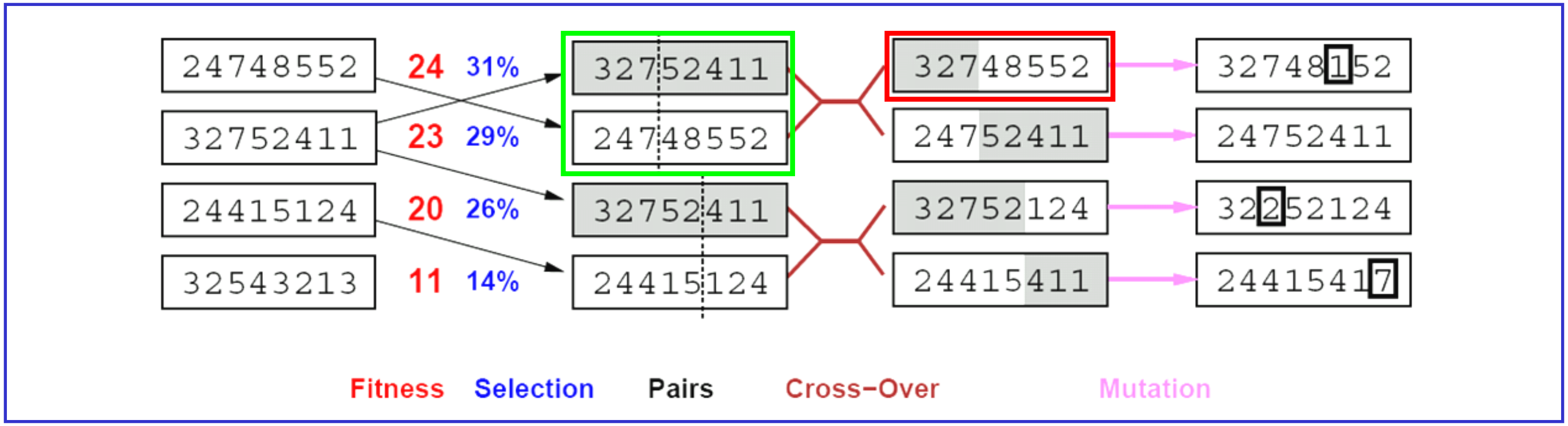
1 6 2 5 7 4 8 3

Kodierung als String

Fitness = 27

(also nur ein Konflikt)

# GA für 8-Damen-Problem: Fitness & Operationen




**Zustände** kodiert durch Strings mit 8 Zeichen aus {1,...,8}.

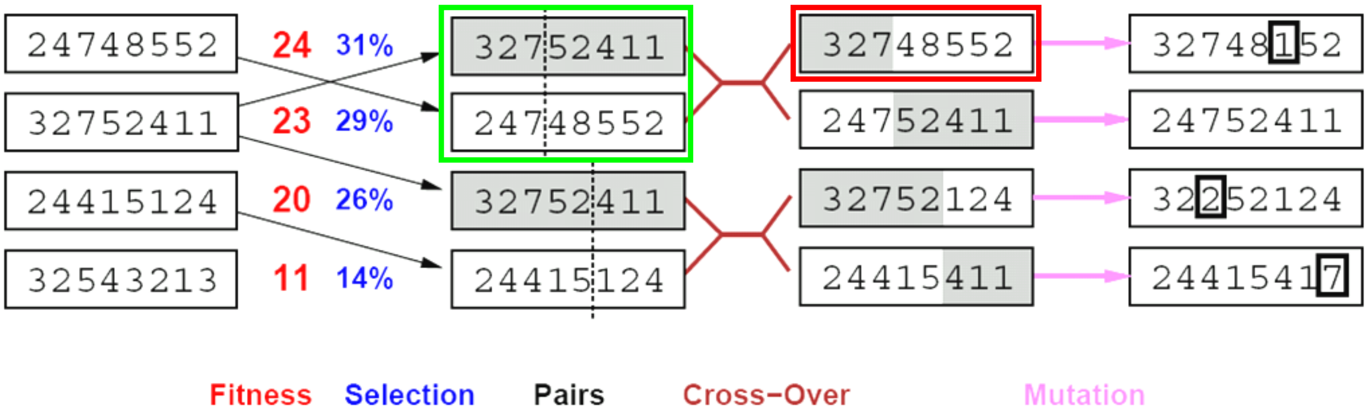
**Fitness** = Zahl der konfliktfreien Damepaare (Lösungen haben Fitness = 28).

**Selektion** durch zufällige Wahl von Paaren nach Auswahlwahrscheinlichkeiten, die proportional zur Fitness sind.

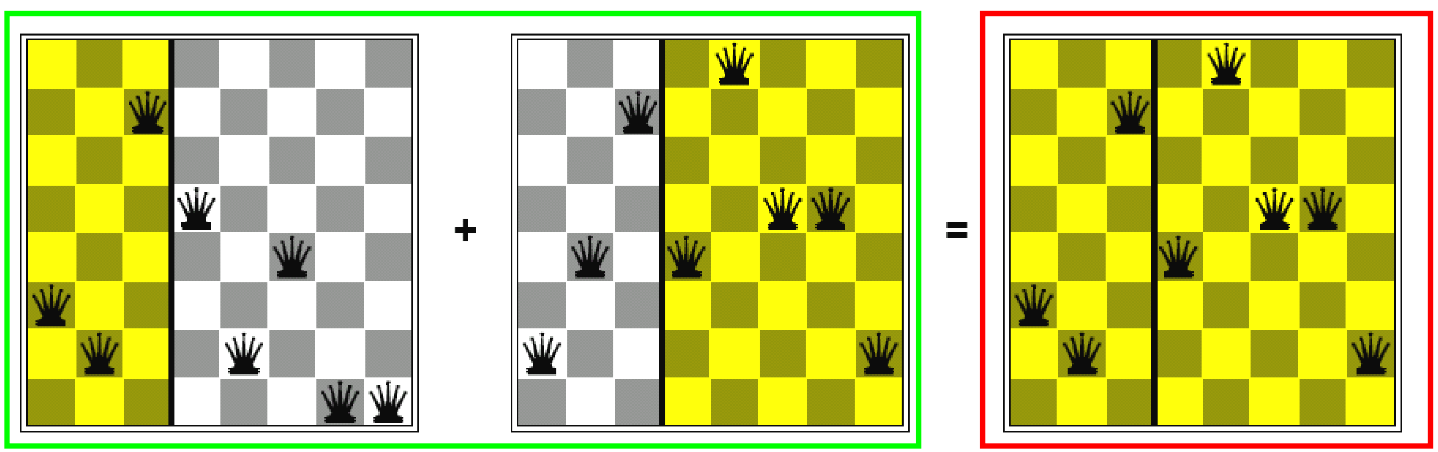
**Kreuzung** durch zufällige Wahl eines Kreuzungspunktes (Cross-Over) – im Bspl. nach 3. Zeichen beim 1. Paar bzw. nach 5. Zeichen beim 2. Paar.

**Mutation** durchzufälliges Verändern einer Position mit vorgeg. Wahrscheinlichkeit.

# GA für 8-Damen-Problem: Bspl. einer Kreuzung




Konfiguration der ersten Kreuzung entsprechend:



# Genetischer Algorithmus für 8-Damen-Problem (4)

**function** GENETIC-ALGORITHM (*population*,FITTNESS-FN) **returns** an individual
inputs:*population*, a set of individuals
FITTNESS-FN, a function to measure the fitness of an individual
repeat
*new_population* ← empty set
**loop for** *i* **from** SIZE(*population*) **do**
*x* ← RANDOM-SELECTION(*population*,FITTNESS-FN)
*y* ← RANDOM-SELECTION(*population*,FITTNESS-FN)
*child* ← REPRODUCE(*x,y*)
**if** (small random probability) **then** *child* ← MUTATE(*child*)
add *child* to *new_population*
*population* ← *new_population*
**until** some individual is fit enough, or enough time has elapsed
**return** the best individual in *population*, according to FITTNESS-FN

Initialisierung der neuen Generation

Zuf. Auswahl der Eltern gem. Fitness

Erzeugung von Kind – ggf. mit Mutation

Auswahl des besten Individuums als Lösung

---

**function** REPRODUCE(*x,y*) **returns** an individual
inputs: *x,y*, parent individuals
*n* ← LENGTH(*x*)
*c* ← random number from 1 to *n*
**return** APPEND(SUBSTRING(*x,1,c*),SUBSTRING(*y,c+1,n*))

Hier – im Ggs. zum bislang gezeigten 8-Damen-Bspl. – nur ein Ergebnis pro Kreuzung.

# Zusammenfassung

- *Heuristiken* fokussieren die Suche.
- *Bestensuche* expandiert die jeweils aktuell am besten bewerteten Knoten zuerst.
- Durch Minimierung der heurist. Kosten h(n) zum Ziel erhalten wir *gierige Suche*.
- Minimierung von f(n) = g(n) + h(n) kombiniert uniforme Kostensuche mit gieriger Suche. Ist h(n) *zulässig*, resultiert die vollständige und optimale *A**-Suche.
- Graphensuche erkennt Wiederholungen von Zuständen.
- *IDA** ist eine Kombination von iterativer Tiefensuche und A*.
- *Lokale Suche* arbeitet immer nur auf einem Zustand und versucht, diesen schrittweise zu verbessern.
- *Genetische Algorithmen* ahmen die Evolution nach, indem sie gute Lösungen miteinander kombinieren.