HOCHSCHULE MAINZ
Wirtschaft
Business

Grundlagen der Informatik
Such-Algorithmen und Komplexität

University of
Applied Sciences

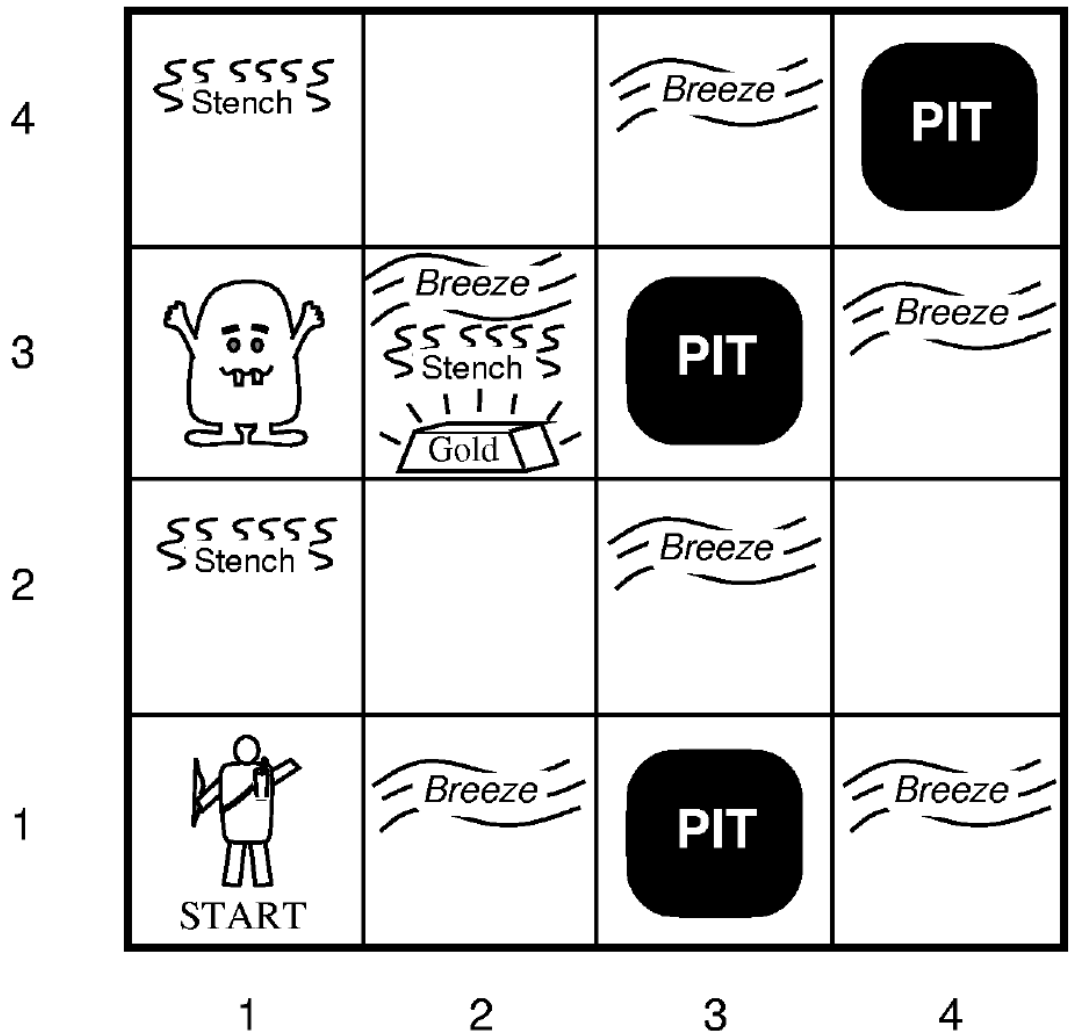
Agenda

1. Problemformulierung
2. Bewertungskriterien zur Unterscheidung von Suchverfahren
3. Allgemeine Suchstrategie und Datenstruktur
4. Uninformierte Suche
5. Informierte Suche

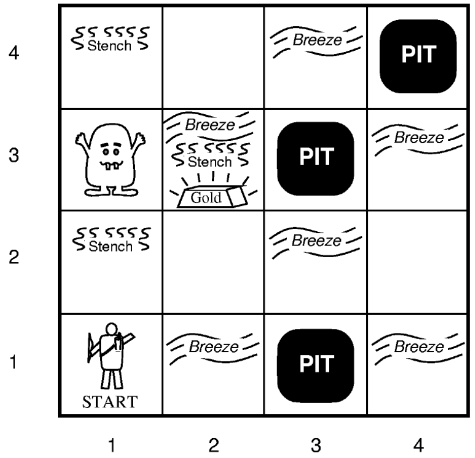
Agenda

1. **Problemformulierung**
2. Bewertungskriterien zur Unterscheidung von Suchverfahren
3. Allgemeine Suchstrategie und Datenstruktur
4. Uninformierte Suche
5. Informierte Suche

Die Wumpus-Welt

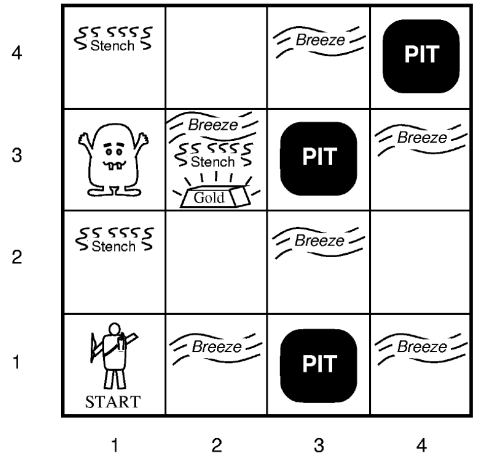


Die Wumpus-Welt



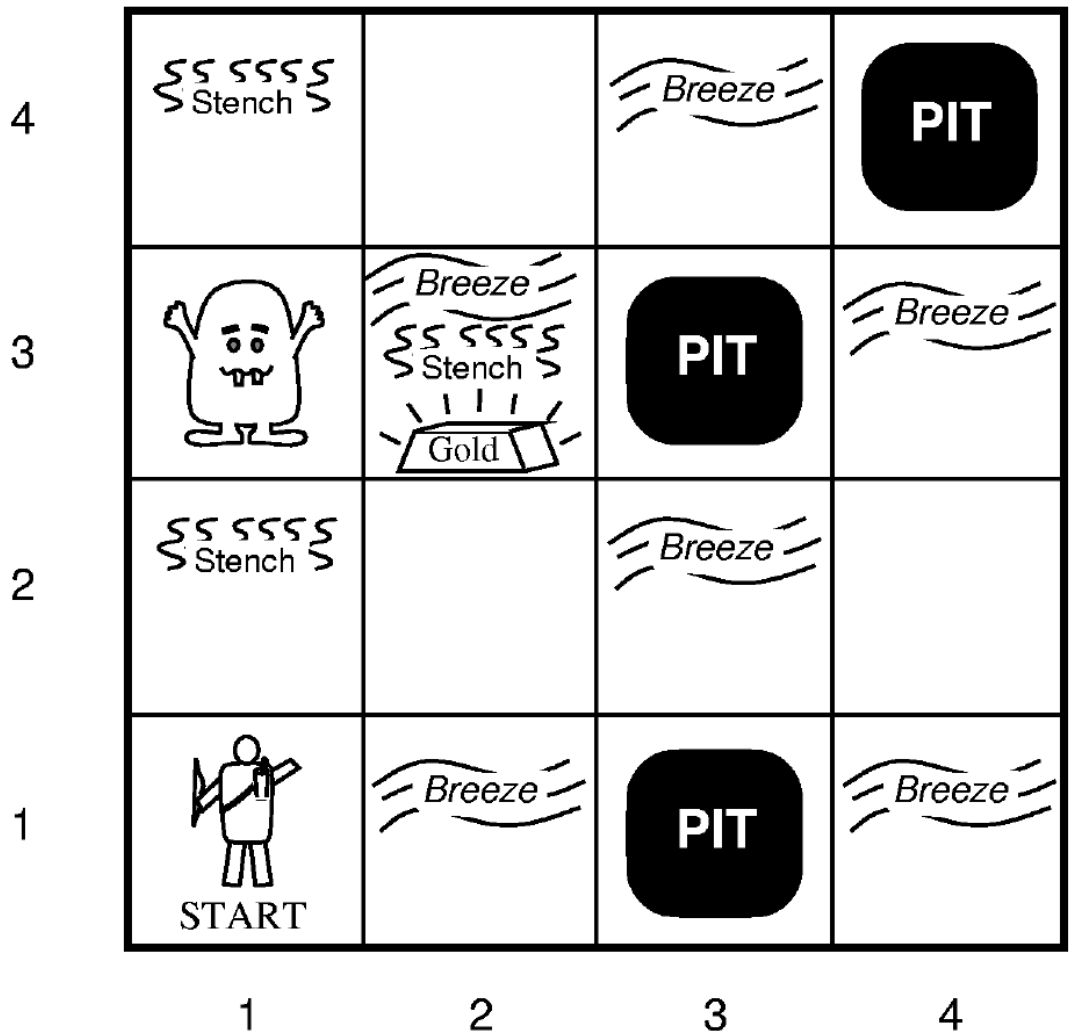
- Ein 4×4 Feld
- Im Kästchen, in dem der Wumpus sich befindet, und in den direkten Nachbarkästchen nimmt man einen üblen Geruch wahr (Stench)
- In den Kästchen neben einer Fallgrube (Pit) nimmt man einen Luftzug wahr (Breeze)
- Im Kästchen mit dem Gold glitzert es (Glitter)
- Wenn der Agent in eine Wand läuft, bekommt er einen Schlag
- Wenn der Wumpus getötet ist, hört man es überall (Todesschrei)
- Wahrnehmungen werden als 5-Tupel dargestellt:
  z. B. [Stench, Breeze, Glitter, None, None] bedeutet, dass es stinkt, zieht und glitzert, aber es gab weder einen Schlag noch einen Todesschrei
- Der Agent kann seinen Standort nicht wahrnehmen!

Die Wumpus-Welt



- Wir steuern einen Agenten
- **Anfangszustand:**
  Agent in [1,1] nach Osten schauend, irgendwo 1 Wumpus, 1 Haufen Gold und 3 Fallgruben
- **Ziel:** Hole das Gold und verlasse die Höhle
- **Aktionen:**
  - gehe vorwärts,
  - 90° nach rechts drehen,
  - 90° nach links drehen,
  - greife ein Objekt im selben Kästchen,
  - schieße (es gibt nur einen Pfeil),
  - verlasse die Höhle (funktioniert nur in Kästchen [1,1])
- Der Agent stirbt, wenn er in eine Fallgrube fällt oder dem lebenden Wumpus begegnet (selbes Feld)

Die Wumpus-Welt



Die Wumpus-Welt

Zustand	Wahrnehmung [Stench, Breeze, Glitter, Bump, Scream]	Aktionen
[1,1]	[None, None, None, None, None]	Vorwärts (nach Osten)
[2,1]	[None, Breeze, None, None, None]	180° drehen und vorwärts (zurück zu 1,1), dann nochmal vorwärts (nach Westen)
[1,1]	[None, None, None, Bump, None]	Drehen um 90° nach rechts, gehe vorwärts (nach Norden)
[1,2]	[Stench, None, None, None, None]	…
…	…	…

Problemlösende Agenten

Für einen (intelligenten) Agenten ist notwendig:

– Formuliere Ziel (Zielzustände)
– Problem-Formulierung:
  Darstellung der „Welt“ und von möglichen Handlungsoptionen („Weltzustandsraum“)
– Wissen des Agenten (über seinen eigenen Zustand, über Auswirkungen von Aktionen, …)?

– Gegeben: Anfangszustand
– Gesucht: Erreichen eines bestimmten Ziel-Zustands durch Ausführen geeigneter Aktionen

=> Finden einer geeigneten (optimalen) Strategie, d. h. einer geeigneten *Aktionsfolge* und *Ausführung* dieser Folge

Das Finden einer geeigneten Strategie kann als *Optimierungsproblem* betrachtet werden.
Im Folgenden wollen wir uns mit möglichen *Problemrepräsentationen* und *Suchstrategien* beschäftigen.

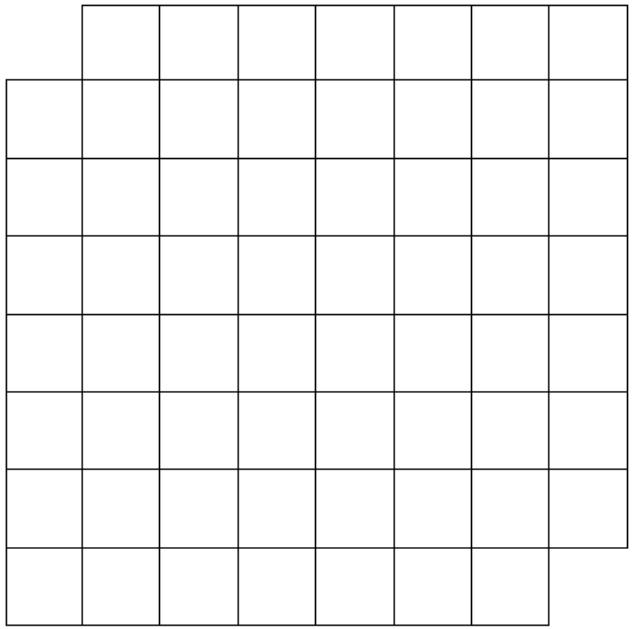
Problemlösende Agenten

- Formulierung des *Ziels*
  Weltzustände mit bestimmten Eigenschaften (Zielzustände)
- Festlegen des *Weltzustandsraums*
  (wichtig: nur die relevanten Aspekte, Abstraktion)
- Festlegen der *Aktionen*, die einen Weltzustand in einen anderen überführen
- Bestimmung des *Problemtyps*, der abhängig vom *Wissen* über Weltzustände und Aktionen ist (Einzustandsproblem, Mehrzustandsproblem usw.) -> Zustände im Suchraum
- Bestimmung der Kosten für das Suchen (*Suchkosten*, Offline-Kosten) und der Ausführungskosten (*Pfadkosten*, Online-Kosten)

Achtung: Die Art der Problemformulierung kann einen großen Einfluss auf die Schwierigkeit der Lösung haben.

Beispiel für Zielformulierung

Gegeben ist ein $n \times n$ (hier 8×8) Brett, bei dem an den beiden diagonal gegenüberliegenden Ecken je ein Feld entfernt wurde:



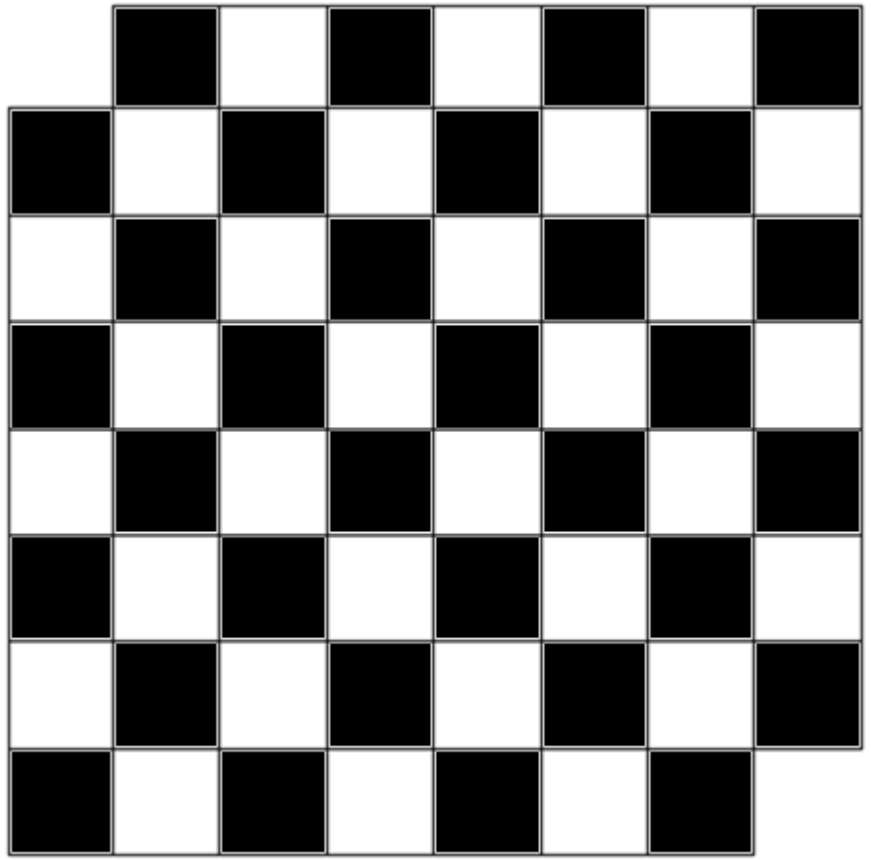
Ziel: Das gegebene Brett mit Dominosteinen belegen, die jeweils zwei benachbarte Felder überdecken, so dass alle Felder abgedeckt werden.

=> Lösung?

Beispiel für Zielformulierung

Alternative Zielformulierung:
Kann man ein Brett, auf dem es $n^2/2$ schwarze und $n^2/2 - 2$ weiße Felder gibt, mit Dominosteinen belegen, die jeweils ein weißes und ein schwarzes Feld überdecken, so dass alle Felder abgedeckt sind?

… nein, natürlich nicht!



Problemtypen: Wissen über Zustände und Aktionen

Einzustandsproblem

- Weltzustand vollständig bekannt (z. B. wo Schmutz liegt, wo Wände sind, usw.)
- eigener Zustand bekannt,
- die Folgen der Aktionen sind bekannt

=> der Agent weiß immer, in welchem Weltzustand er ist

=> der Agent kann einen Plan aufstellen, der ihn in einen Zielzustand bringt

Mehrzustandsproblem

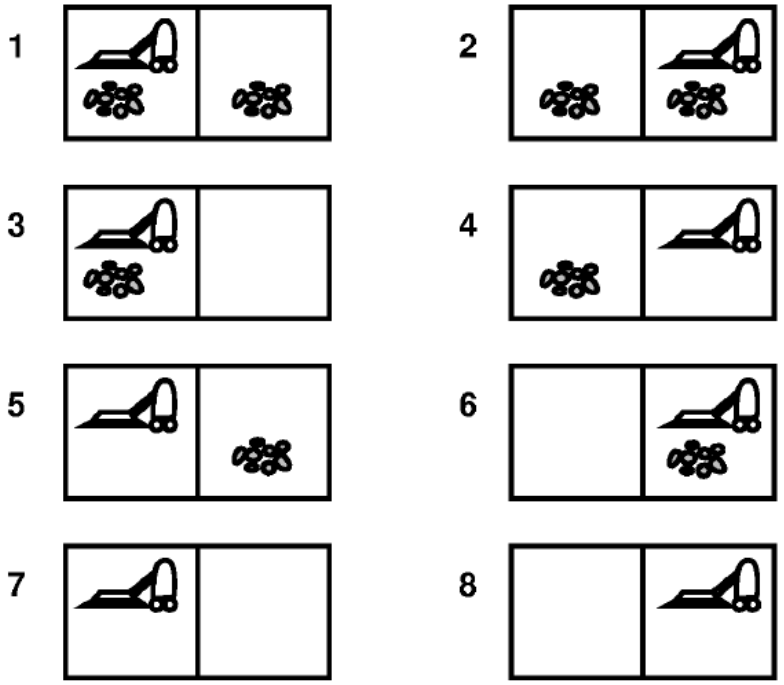
- Weltzustand nicht vollständig bekannt, mögliche Zustände sind jedoch bekannt (z. B. ist nicht klar, wo der Schmutz sich befindet, aber es ist klar, dass Schmutz auf 0 bis 2 der Felder sein kann)
- die Folgen der Aktionen sind bekannt

=> der Agent weiß nur, in welcher Menge von Weltzuständen er ist

=> jede weitere Aktion verkleinert die Menge der Weltzustände, in der sich der Agent momentan befinden kann, oder lässt sie zumindest gleich

Problemtypen am Beispiel der Staubsaugerwelt

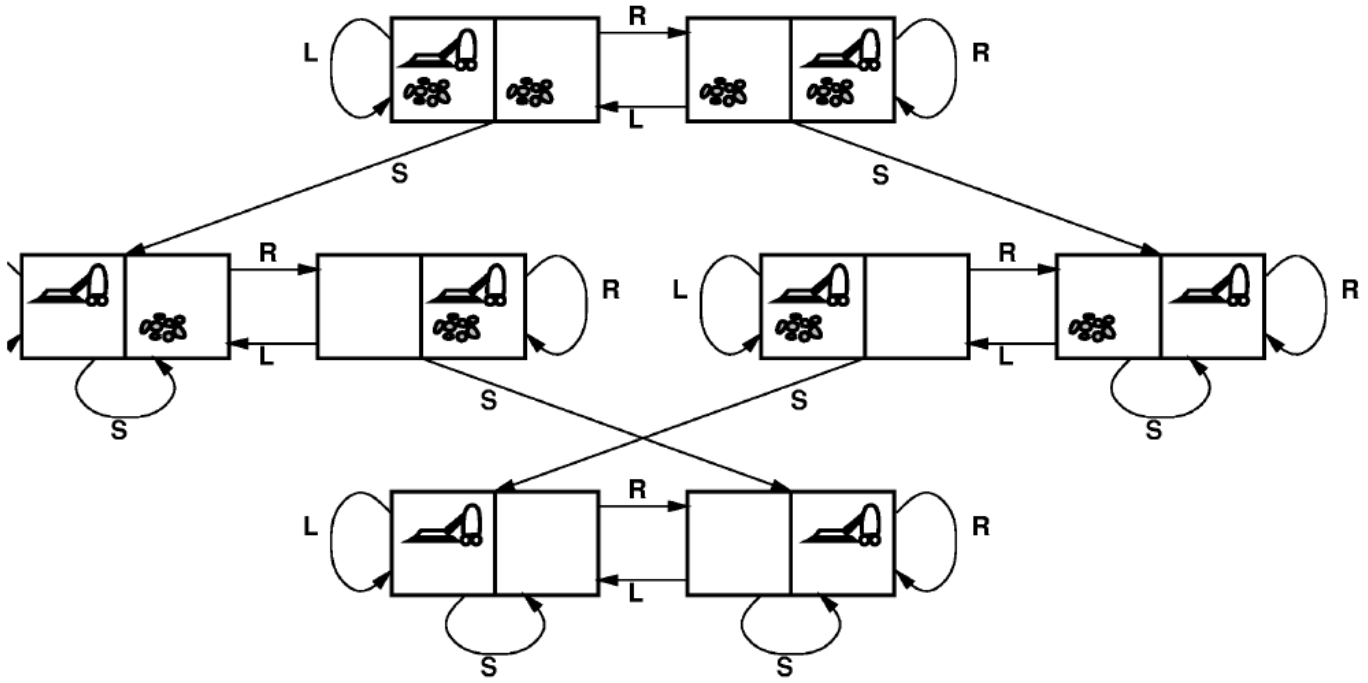
– Weltzustandsraum: 2 räumliche Positionen, beide können verschmutzt sein oder nicht
  => 8 Weltzustände



- Aktionen: links (L), rechts (R), saugen (S)
- Ziel: kein Schmutz in den Räumen
- Pfadkosten: pro Aktion 1 Einheit

Problemtypen am Beispiel der Staubsaugerwelt - Einzustandsproblem

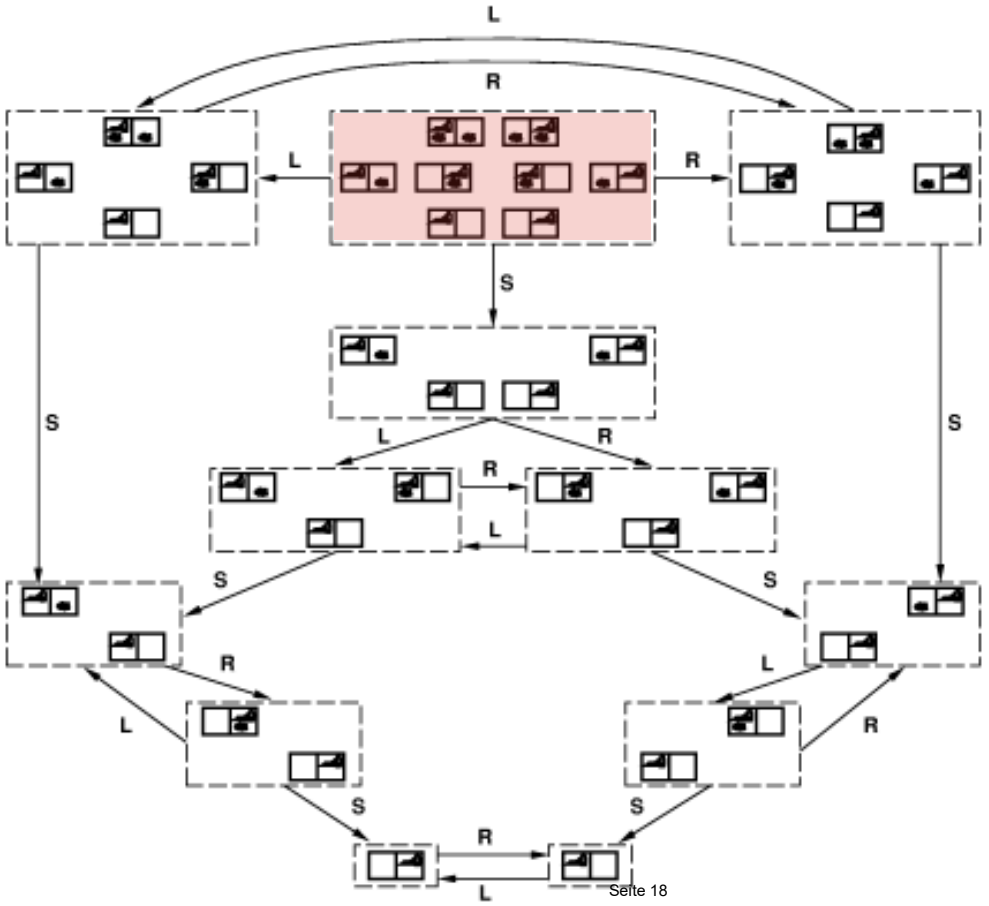
– Welt vollständig zugänglich, Staubsauger weiß immer, wo er und der Schmutz sind
– Problemlösen reduziert sich auf die Suche nach einem Pfad vom Anfangszustand zu einem Zielzustand



– Zustände für die Suche: Die Weltzustände 1–8

Problemtypen am Beispiel der Staubsaugerwelt - Mehrzustandsproblem

- Falls der Staubsauger keine Sensoren besitzt, weiß er nicht, wo er ist und wo Schmutz ist
- Trotzdem kann er das Problem lösen
- Zustände sind dann Wissenszustände:



- Zustände für die Suche: Potenzmenge der Weltzustände 1–8

Problemtypen: Wissen über Zustände und Aktionen

Kontingenzproblem

- Agent kennt seinen momentanen Zustand
- auch *mögliche* Aktionen und ihre Wirkung sind bekannt
- allerdings kann der Agent nicht davon ausgehen, dass alle Aktionen erfolgreich verlaufen (z. B. weiß der Staubsauger nicht, ob „saugen“ immer funktioniert)
- Agent muss also beobachten, was passiert, und dann weiter entscheiden

=> es ist unmöglich, eine komplette Sequenz von Aktionen zur Lösung im voraus zu bestimmen, da nicht alle Informationen über Zwischenzustände vorliegen

Explorationsproblem

- Zustandsraum und Effekte der Aktionen nicht vollständig bekannt (z. B. weiß der Staubsauger nicht, wie viele Räume es gibt, wo Wände sind oder was „rechts“ bedeutet)
- der Agent ist daher gezwungen zu probieren, um zu lernen
- Schwer!

Begriffe zur Problemformulierung

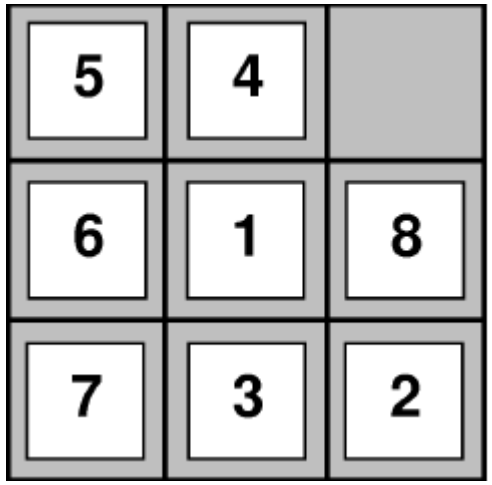
- Problemformulierung:
  - Anfangszustand: Zustand, von dem der Agent glaubt, anfangs zu sein (auch „Menge möglicher Anfangszustände“)
  - Zustandsraum: Menge aller möglichen Zustände
  - Operator: Definiert unter welchen Voraussetzungen eine Aktion anwendbar ist und welche Zustandsänderung sie bewirkt ($S \rightarrow S'$)
  - Zieltest: Test, ob die Beschreibung eines Zustands einem Zielzustand entspricht

- Pfadkosten: Kostenfunktion *g* über Pfade. Setzt sich üblicherweise aus der Summe der Kosten der Aktionen zusammen
- Pfad: Sequenz von Aktionen, die von einem Zustand zu einem anderen führen
- Lösung: Pfad von einem Anfangs- zu einem Zielzustand
- Suchkosten: Zeit- und Speicherbedarf, um eine Lösung zu finden

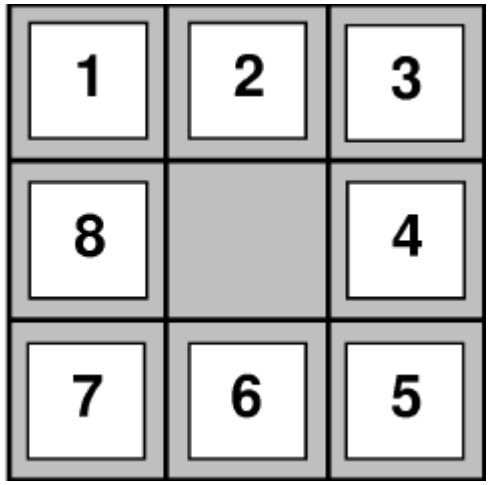
Problemformulierung: Beispiel Schiebepuzzle

Zustände:

- Beschreibung der Lage jedes der 8 Kästchen und (aus Effizienzgründen) des Leerkästchens
- Operator: „Verschieben“ des Leerkästchens nach links, rechts, oben und unten
- Zieltest: Entspricht aktueller Zustand dem rechten Bild?
- Pfadkosten: Jeder Schritt kostet 1 Einheit



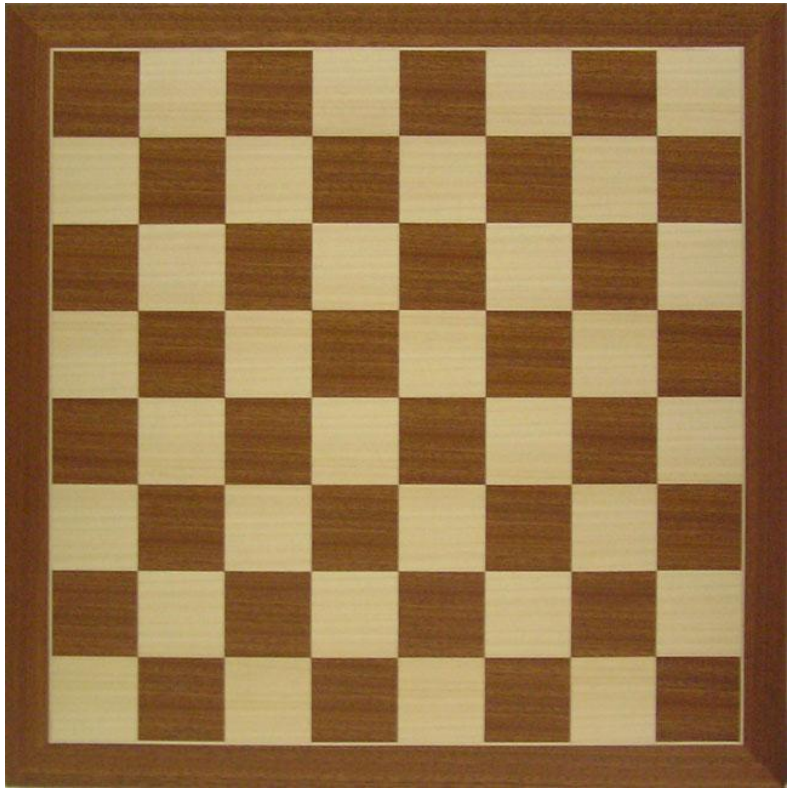
Startzustand



Zielzustand

Problemformulierung: Beispiel 8 Damen Problem

- Es sollen acht Damen auf einem Schachbrett so aufgestellt werden, dass keine zwei Damen einander schlagen können (die Figurenfarbe wird dabei ignoriert)
- Zieltest: 8 Damen auf dem Brett, keine angreifbar
- Pfadkosten: 0 (nur die Lösung interessiert)



Problemformulierung: Beispiel 8 Damen Problem

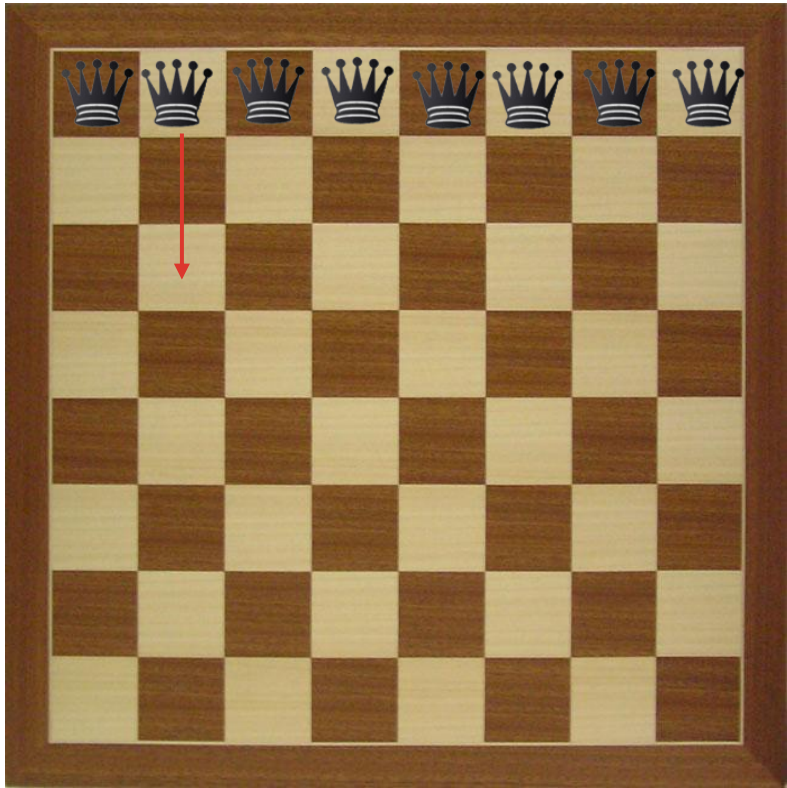
- Es sollen acht Damen auf einem Schachbrett so aufgestellt werden, dass keine zwei Damen einander schlagen können (die Figurenfarbe wird dabei ignoriert)
- Zieltest: 8 Damen auf dem Brett, keine angreifbar
- Pfadkosten: 0 (nur die Lösung interessiert)
- Darstellung 1:
  - Zustände: beliebige Anordnung von 0–8 Damen
  - Operatoren: setze eine der Damen aufs Brett
  - Problem: 64^8 Zustände

Problemformulierung: Beispiel 8 Damen Problem

- Es sollen acht Damen auf einem Schachbrett so aufgestellt werden, dass keine zwei Damen einander schlagen können (die Figurenfarbe wird dabei ignoriert)
- Zieltest: 8 Damen auf dem Brett, keine angreifbar
- Pfadkosten: 0 (nur die Lösung interessiert)
- Darstellung 2:
  - Zustände: Anordnung von 0–8 Damen in unangreifbarer Stellung
  - Operator: Setze eine der Damen so weit wie möglich links unangreifbar auf das Brett
  - Problem: wenige Zustände (2057), aber manchmal keine Aktion möglich

Problemformulierung: Beispiel 8 Damen Problem

- Es sollen acht Damen auf einem Schachbrett so aufgestellt werden, dass keine zwei Damen einander schlagen können (die Figurenfarbe wird dabei ignoriert)
- Zieltest: 8 Damen auf dem Brett, keine angreifbar
- Pfadkosten: 0 (nur die Lösung interessiert)
- Darstellung 3:
  - Zustände: 8 Damen auf dem Brett, eine in jeder Spalte
  - Operatoren: verschiebe eine angegriffene Dame in derselben Spalte



Problemformulierung: Beispiel Missionare und Kannibalen

Informelle Problembeschreibung:

- An einem Fluss haben sich 3 Kannibalen und 3 Missionare getroffen, die alle den Fluss überqueren wollen
- Es steht ein Boot zur Verfügung, das maximal zwei Leute aufnehmen kann
- Es sollte nie die Situation auftreten, dass an einem Ufer Missionare und Kannibalen sind und dabei die Anzahl der Kannibalen die Anzahl der Missionare übertrifft

=> Finde eine Aktionsfolge, die alle an das andere Ufer bringt

Problemformulierung: Beispiel Missionare und Kannibalen

Formelle Problembeschreibung:

- Zustände:
  - Ein Zustand wird durch die folgenden drei Variablen beschrieben:
    - $x =$ Anzahl der Missionare am Ausgangsufer,
    - $y =$ Anzahl der Kannibalen am Ausgangsufer,
    - $z =$ Position des Boots (1 = Boot am Ausgangsufer, 0 = Boot am Zielufer)
  - Zustände (x, y, z) sind Tripel mit $x, \mathrm{y} \in \{0,1,2,3\}$, und $\mathrm{z} \in \{0,1\}$
  - Anfangszustand: $(3, 3, 1)$
- Operatoren:
  Von jedem Zustand aus entweder einen Missionar, einen Kannibalen, zwei Missionare, zwei Kannibalen, oder einen von jeder Sorte über den Fluss bringen (5 Operatoren)
- Beachte: nicht jeder Zustand ist erreichbar, z. B. (0,0,1) und einige sind illegal.
- Zielzustand: $(0,0,0)$
- Pfadkosten: 1 Einheit pro Flussüberquerung

Suche allgemein für Missionare und Kannibalen

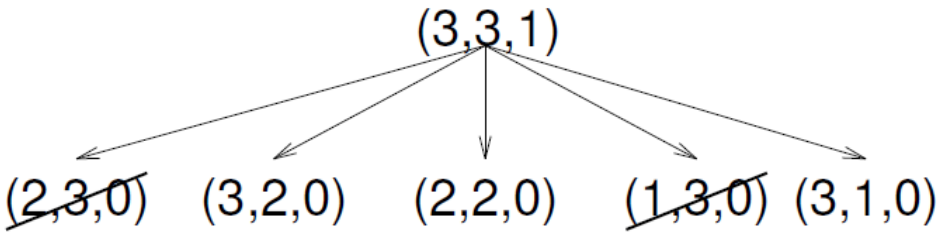
– Ausgehend vom Anfangszustand schrittweise alle Nachfolgezustände erzeugen

=> *Suchbaum*

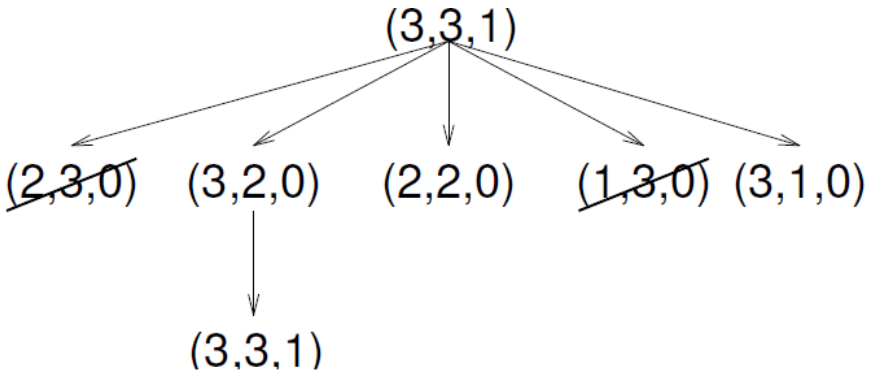
(a) Anfangszustand

(3,3,1)

(b) nach Expansion von (3,3,1)



(c) nach Expansion von (3,2,0)



Beispiele für reale Probleme

- Routenplanung, Finden kürzester Pfade: Im Prinzip einfach (polynomiales Problem).
  Komplikationen bei unbekannten, sich dynamisch verändernden Pfadkosten (Beispiel: Routenplanung in Norwegen)
- Planung von Rundreisen (TSP): Eines der prototypischen NP-vollständigen Probleme
- VLSI Layout: Auch ein NP-vollständiges Problem
- Roboter Navigation (mit vielen Freiheitsgraden):
  Schwierigkeit nimmt mit der Anzahl der Freiheitsgrade extrem zu.
  Weitere mögliche Komplikationen: Fehler bei Wahrnehmungen, unbekannte Umgebungen
- Montageplanung: Planung des Zusammenbaus von komplexen Objekten

Agenda

1. Problemformulierung
2. **Bewertungskriterien zur Unterscheidung von Suchverfahren**
3. Allgemeine Suchstrategie und Datenstruktur
4. Uninformierte Suche
5. Informierte Suche

Bewertungskriterien zur Unterscheidung von Suchverfahren

- Vollständigkeit:
  Wird immer eine Lösung gefunden, sofern es eine gibt?
- Optimalität:
  Findet das Verfahren immer die beste Lösung?
- Zeitkomplexität:
  Wie lange dauert es (im schlechtesten Fall / worst case), bis eine Lösung gefunden ist?
- Platzkomplexität:
  Wieviel Speicher benötigt die Suche (im schlechtesten Fall / worst case)?

Vollständigkeit

– Ein **Suchalgorithmus** heißt **vollständig**, wenn er **garantiert** eine Lösung findet, **sofern eine existiert**.

Formal:

– Sei P ein Problem mit Zustandsraum S, Startzustand $s_0 \in S$,und Zielzuständen $S_G \subseteq S$.
Ein Algorithmus A ist **vollständig**, wenn gilt:

$$\left(\exists s_g \in S_G\right) \Rightarrow A \textit{ findet } s_g$$

Optimalität

Sei X die Menge aller **zulässigen Lösungen** (auch **Feasible Set** genannt), und

$$f: \mathrm{x} \to \mathbb{R} \; \forall \; x \in X$$

eine **Zielfunktion**, die jeder zulässigen Lösung einen Wert zuordnet (z. B. Kosten, Gewinn, etc.).

Dann ist

– $x^* \in X$ *mit* $f(x^*) \leq f(x) \; \forall x \in X$ *(Minimierungsproblem)* bzw.
– $x^* \in X$ *mit* $f(x^*) \geq f(x) \; \forall x \in X$ *(Maximierungsproblem)*

eine **optimale Lösung** (Zielfunktion wird **minimiert** oder **maximiert**).

Beispiel:
Eine kostenoptimale Produktion in einem Unternehmen: Wir fangen gar nicht an zu produzieren (Kosten von 0€). (Optimal ist also nicht immer betriebswirtschaftlich sinnvoll.)

Laufzeit und Speicherbedarf

- Gegeben: ein Algorithmus (z. B. ein Suchverfahren)
- Frage: wie wächst seine Laufzeit / sein Speicherbedarf mit der Größe des Problems?
- „Wie skaliert der Algorithmus“

- Bei der Bestimmung der Zeitkomplexität reicht es nicht aus, einfach die Zeit zur Berechnung einer bestimmten Eingangsbelegung experimentell zu messen, weil die Antwortzeiten dann abhängig von der CPU Taktfrequenz, der Rechnerauslastung etc. sind

- Idee: Zählen der Anzahl durchzuführender primitiver Operationen und Ausdrücke als Funktion, abhängig von der Größe der Eingabe n
- Eingabe: Maß für die Größe des Problems (z. B. Anzahl Elemente die wir sortieren sollen)
- Detailgrad der Abschätzung muss nur gering sein, denn uns interessiert das „Big Picture“ – skaliert der Algorithmus gut oder schlecht?

Laufzeit und Speicherbedarf

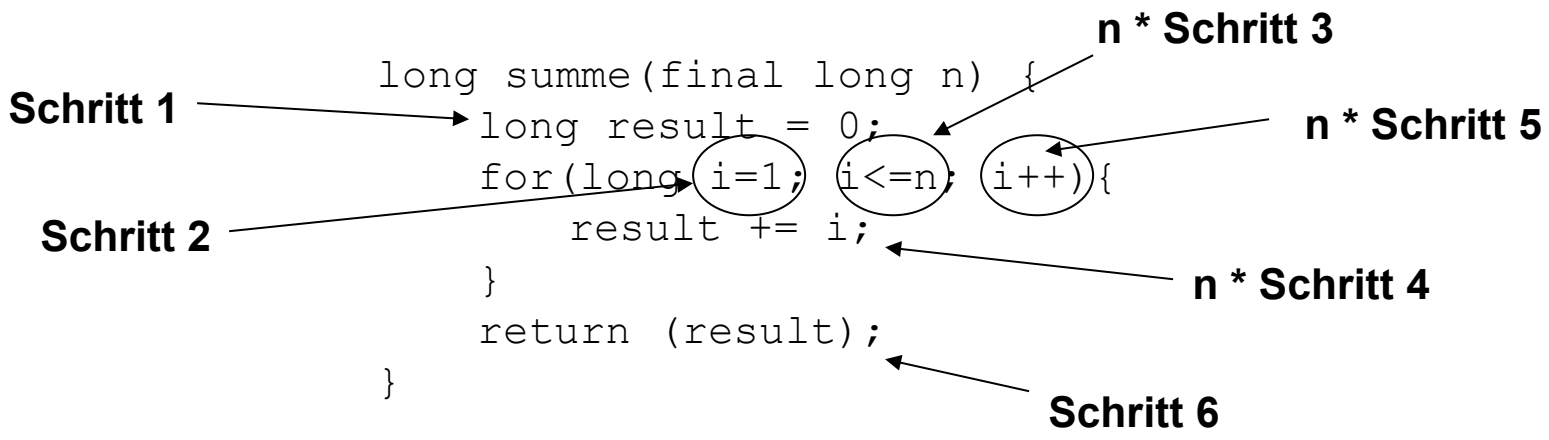
Gegeben:

```
/* Liefert Summe der Zahlen von 1 bis n */
long summe(final long n) {
        long result = 0;
        for(long i=1; i<=n; i++){
                    result += i;
        }
        return (result);
}
```

Frage: Wie gut ist *summe*?
Wieviel Zeit braucht summe in Abhängigkeit von n?

Laufzeit und Speicherbedarf

- Schrittanzahl ist 3+3*n, wobei die Schrittdauer unklar ist
- Wir nennen die Angabe der (realen) Schrittanzahl analytische Betrachtung der Komplexität
- Die Größenordnung der Schrittanzahl steigt linear mit dem Eingabewert n.
- Bei Effizienzfragen stellt sich immer die Frage, ob man einen effizienteren Algorithmus finden kann
- Was wäre eine bessere Komplexitätsklasse? Gibt es einen entsprechenden Algorithmus für das gegebene Problem?



Die $\mathcal{O}$-Notation (Landau Notation)

- Zur Analyse des asymptotischen Verhaltens von Algorithmen verwendet man häufig das $\mathcal{O}$-Kalkül

- Maß für
  - die Anzahl der Elementarschritte (Zeitkomplexität) oder
  - der Speichereinheiten (Platzkomplexität),
- die ein Algorithmus benötigt, in Abhängigkeit von der Größe des gegebenen Problems n

Ordnung einer Funktion

- Es seinen $f, g\colon \mathbb{N}_0 \to \mathbb{R}^+$ zwei Funktionen
- f ist von der Ordnung g, falls es eine Konstante $c > 0$ und eine natürliche Zahl n_0 gibt mit:

$$f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$$

- Mit anderen Worten: Für hinreichend große n wächst $f(n)$ (abgesehen von einem konstanten Faktor c) höchstens so schnell wie $g(n)$

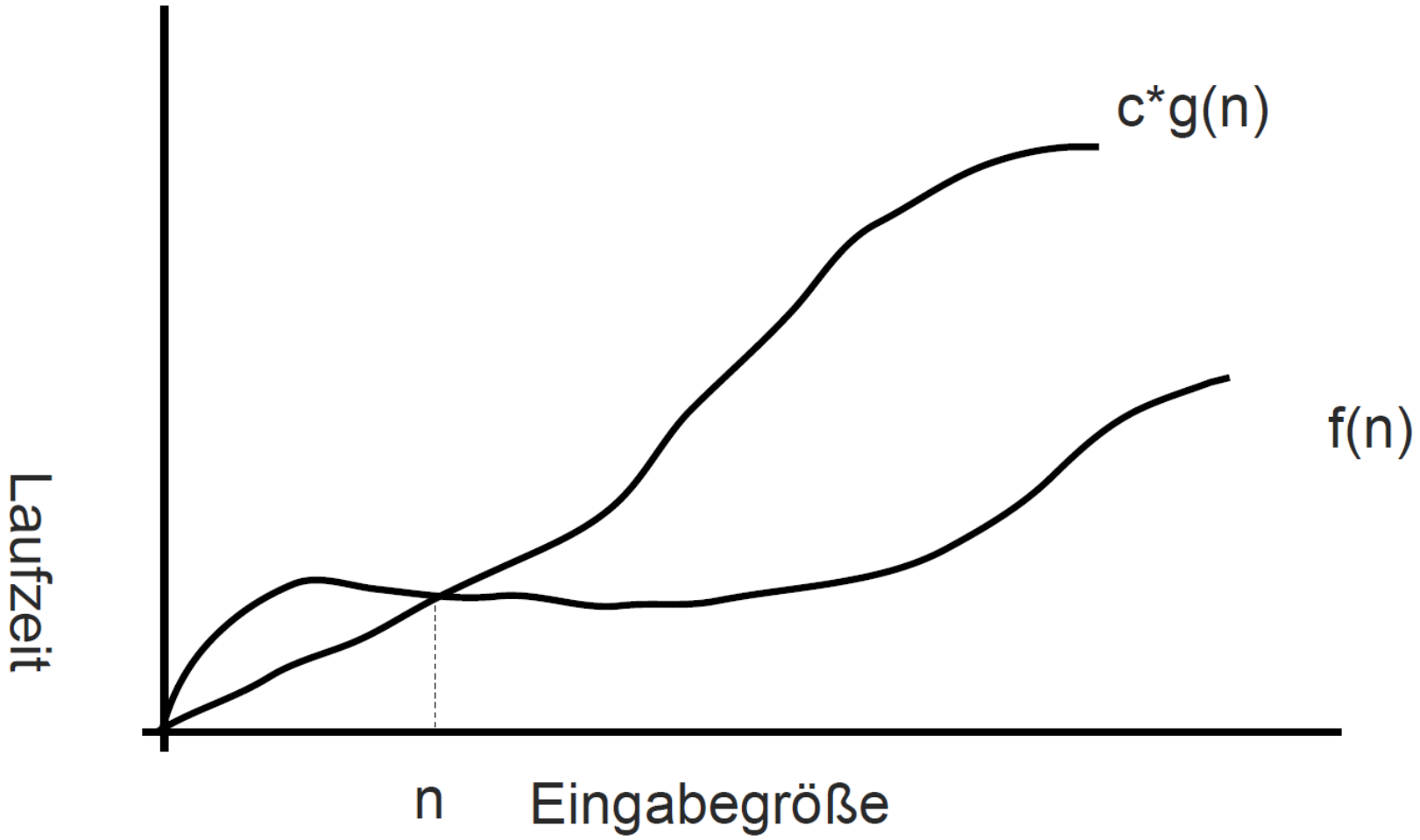
- Mit $\mathcal{O}(g)$ wird die Menge der Funktionen f bezeichnet, die von der Ordnung g sind:

$$\mathcal{O}(g) = \{f\colon \mathbb{N}_0 \to \mathbb{R}^+ | \exists c > 0 \wedge n_0 \in \mathbb{N}_0 \colon f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0\}$$

- Falls $f \in \mathcal{O}(g)$ ist, sind alle Funktionswerte von f ab einer festen Stelle n_0 kleiner als die von g (abgesehen von der Konstanten c)

- Insofern stellt $\mathcal{O}(g)$ die obere Grenze der Laufzeit bzw. des Speicher-Bedarfs dar („Worst Case“)

Illustration



Beispiel $\mathcal{O}(n^2)$

- Gegeben sei die Funktion $f(n) = 3n^2 + 7n + 9$
- Es gilt $f \in \mathcal{O}(n^2)$
- Beweis:

$$
\begin{aligned}
f(n) &= 3n^2 + 7n + 9 \\
&\leq 3n^2 + 7n^2 + 9n^2 \; \forall \, n \geq 0 \\
&\leq 9n^2 + 9n^2 + 9n^2 \; \forall \, n \geq 0 \\
&\leq 27n^2 \; \forall \, n \geq 0
\end{aligned}
$$

- Allgemein gilt für jedes Polynom $p_k(n)$ vom Grade k : $p_k \in \mathcal{O}(n^k)$

Beispiel: maximale Abschnittssumme

Aufgabe: Auffinden einer maximale Abschnittssumme in einem Array von n ganzen Zahlen

Die maximale Abschnittssumme ergibt sich aus der *Teilfolge von aufeinanderfolgenden Zahlen, die unter allen möglichen Teilfolgen die größte Summe* liefert.

Hat man z. B. die folgende Zahlenreihe:

-59, 52, 46, 14, -50, 58, -87, -77, 34, 15

so liefert die folgende Teil-Zahlenfolge die maximale Abschnittssumme:

52 + 46 + 14 + -50 + 58 = 120

Beispiel: maximale Abschnittssumme

– $\mathcal{O}(n^3)$, „kubischer Algorithmus“

```
int maxfolge1(int z[], int n) {
    int i, j, k, sum, max = −10000000;
    for (i = 0; i < n; i++)
        for (j = i; j < n; j++) {
            sum = 0;
            for (k = i; k <= j; k++)
                sum += z[k];
            if (sum > max)
                max = sum;
        }
    return max;
}
```

Beispiel: maximale Abschnittssumme

- Grundidee des ersten Algorithmus, aber bereits errechnete Summen werden gespeichert
- So sparen wir uns die äußerste Schleife und kommen mit zwei Schleifen aus
- $\mathcal{O}(n^2)$, „quadratischer Algorithmus“

```
int maxfolge2(int z[], int n) {
    int i, j, sum, max = −10000000;
    for (i=0; i<n; i++) {
        sum = 0;
        for (j=i; j<n; j++) {
            sum += z[j];
            if (sum > max)
                max = sum;
        }
    }
    return max;
}
```

Beispiel: maximale Abschnittssumme

- $\mathcal{O}(n)$, „linearer Algorithmus“
- Merken der bisher größte Abschnittssumme in *gesamtmax*
- Berechnung der Abschnittssumme des aktuellen Teilstücks in *endesumme*
  - Falls bei Addition der nächsten Zahl *endesumme* > *gesamtmax*, wird dies die neue maximale Abschnittssumme (*gesamtmax* := *endesumme*)
  - Falls *endesumme* durch Addition der nächsten Zahl negativ wird setze *endesumme* := 0

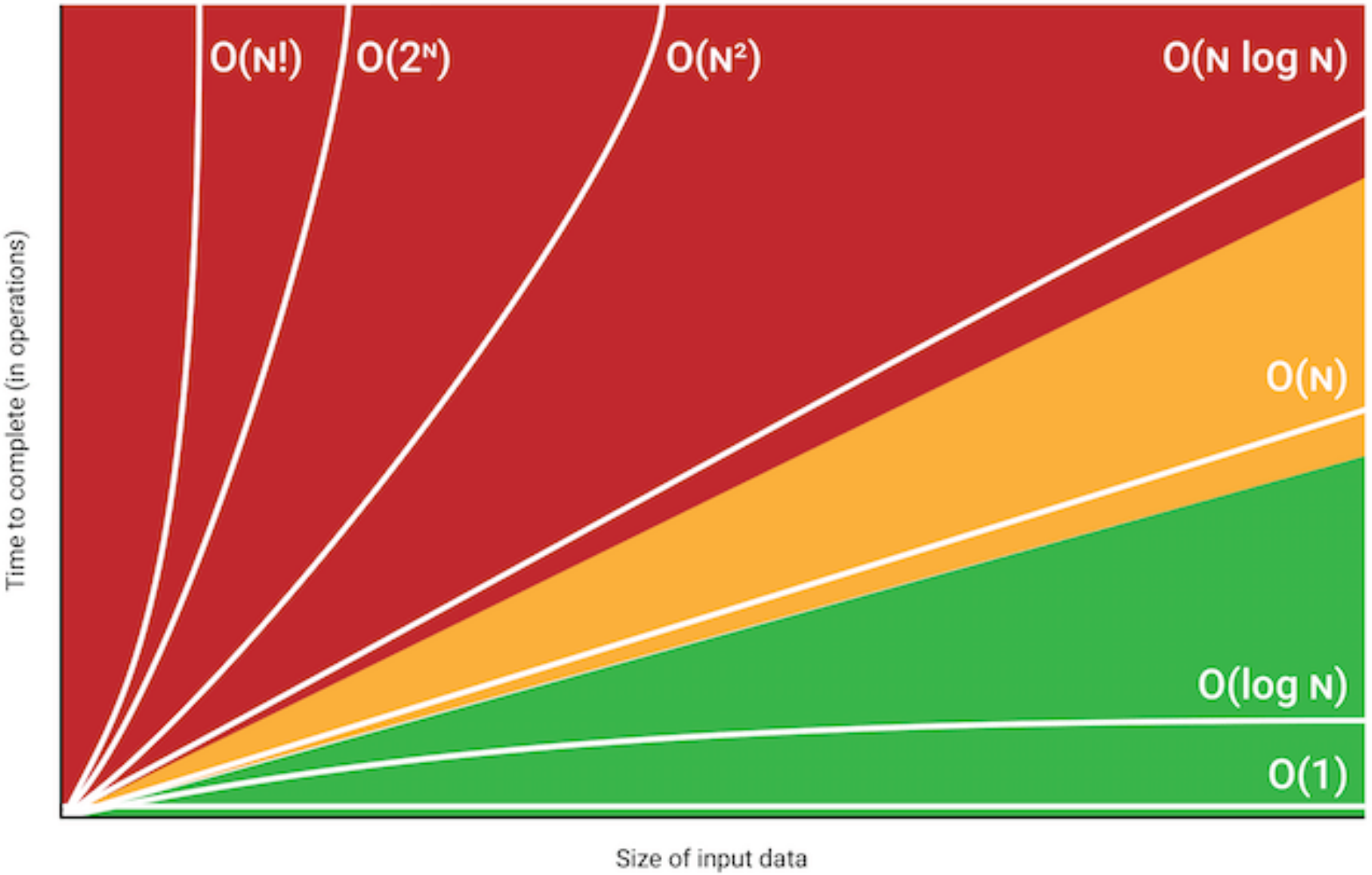
```
int maxfolge3(int z[], int n) {
    int i, s, gesamtmax = −10000000, endesumme = 0;
    for (i=0; i<n; i++) {
        endesumme = ((s=endesumme+z[i]) > 0) ? s : 0;
        if (endesumme > gesamtmax)
            gesamtmax = endesumme;
    }
    return gesamtmax;
}
```

Einen Algorithmus mit einem noch günstigeren Zeitverhalten als „proportional zu n“ kann es für die Ermittlung einer maximalen Abschnittssumme nicht geben, da ja auf jede Zahl der Folge zumindest einmal zugegriffen werden muss.

Typische Klassen

- Hinweis: Man sollte eine Funktion immer **so genau wie möglich von oben abschätzen**. Wert der Schätzung sonst gering.

- Typische Klassen:
  - Logarithmisch $\mathcal{O}(\log n)$
  - Linear $\mathcal{O}(n)$
  - Quadratisch $\mathcal{O}(n^2)$
  - Polynomial $\mathcal{O}(n^k)$ $(k \geq 1)$
  - Exponentiell $\mathcal{O}(a^n)$ $(a > 1)$



Typische Klassen

- Bei einer algorithmischen Komplexität $> \mathcal{O}(\mathrm{n}^3)$ ist die Suche nach einem besseren Algorithmus angeraten
- Für viele Probleme gibt es keinen solchen optimalen Algorithmus und es werden dann nicht deterministische Algorithmen oder auch Approximationen verwendet, die dann jedoch in polynomieller Laufzeit eine Näherungslösung liefern

	n				
	1	10	100	1000	10000
$log_e n$	0	2	5	7	9
n	1	10	100	1000	10000
$n log_e n$	0	23	460	6908	$9 * 10^4$
n^2	1	100	10000	10^6	10^8
n^3	1	1000	10^6	10^9	10^{12}
e^n	3	$2 * 10^4$	$3 * 10^{43}$	$2 * 10^{434}$	$9 * 10^{4342}$

Bewertung der $\mathcal{O}$-Notation

- $\mathcal{O}$-Notation ist eine asymptotische Betrachtung der oberen Grenze der Laufzeit, „Worst Case“
- Konstante wird vernachlässigt → kann groß sein
- Praktische Erfahrung ist wichtig:
  - Komplexitätsklassen sind für die praktische Bewertung der Lösbarkeit eines Problems nicht genau genug
  - Laufzeitverhalten eines Algorithmus kann aber einen guten Anhaltspunkt liefern
  - Es gibt Algorithmen, die im Durchschnitt (Average Case) sehr gute Laufzeiten haben, aber im Worst Case sehr schlecht abschneiden
  - Was sind realistische Problemgrößen? („Kleine“ Problemgröße vs asymptotische Laufzeitbetrachtung)
  - Tradeoff Qualität vs. Laufzeit → spezialisierte Algorithmen oder Heuristiken in Erwägung ziehen

Agenda

1. Problemformulierung
2. Bewertungskriterien zur Unterscheidung von Suchverfahren
3. **Allgemeine Suchstrategie und Datenstruktur**
4. Uninformierte Suche
5. Informierte Suche

Rückblick: Suche allgemein für Missionare und Kannibalen

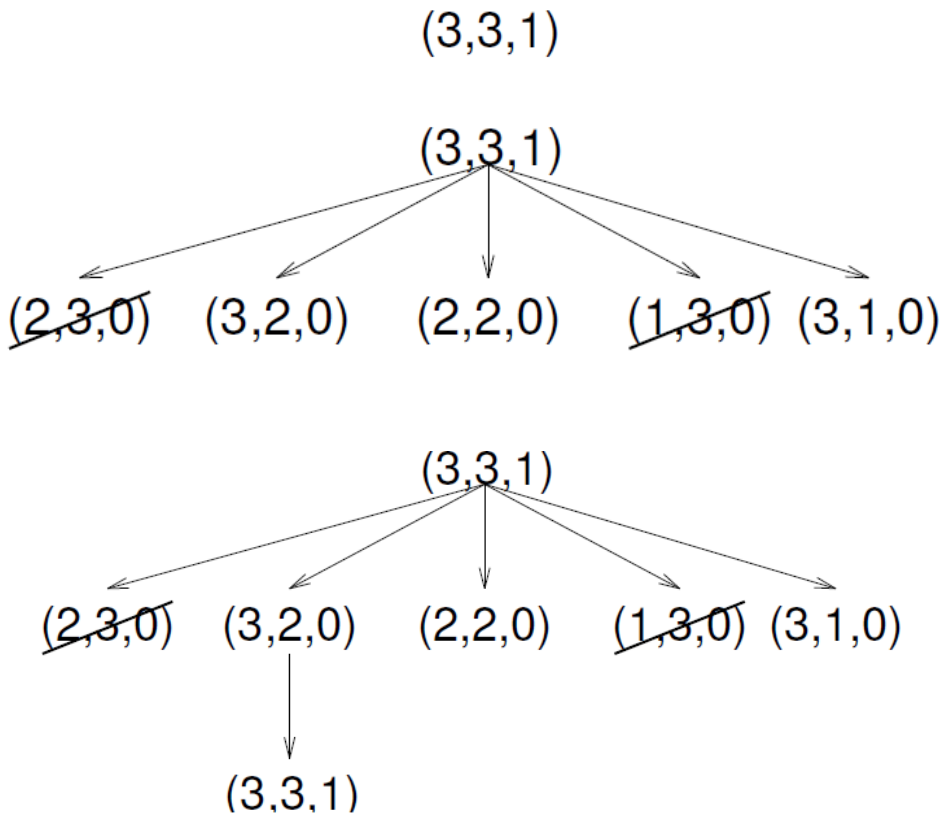
– Ausgehend vom Anfangszustand schrittweise alle Nachfolgezustände erzeugen

=> *Suchbaum*

(a) Anfangszustand

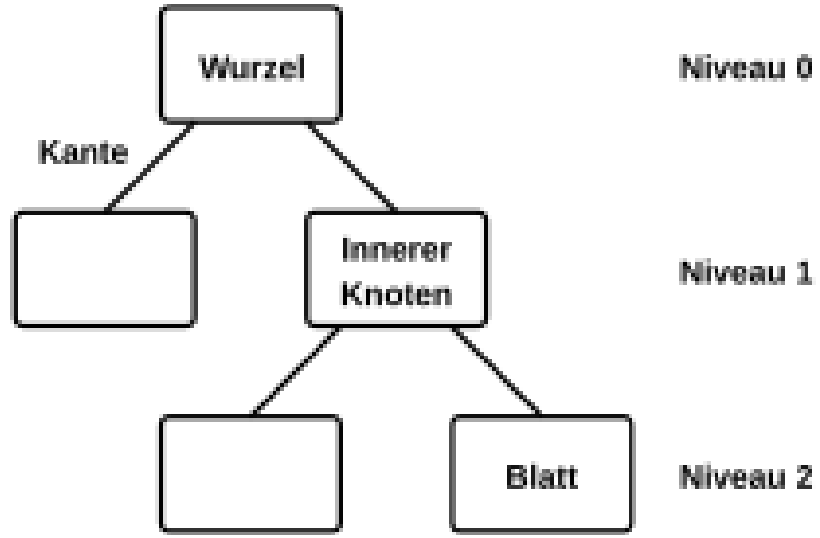
(b) nach Expansion von (3,3,1)

(c) nach Expansion von (3,2,0)



Baum

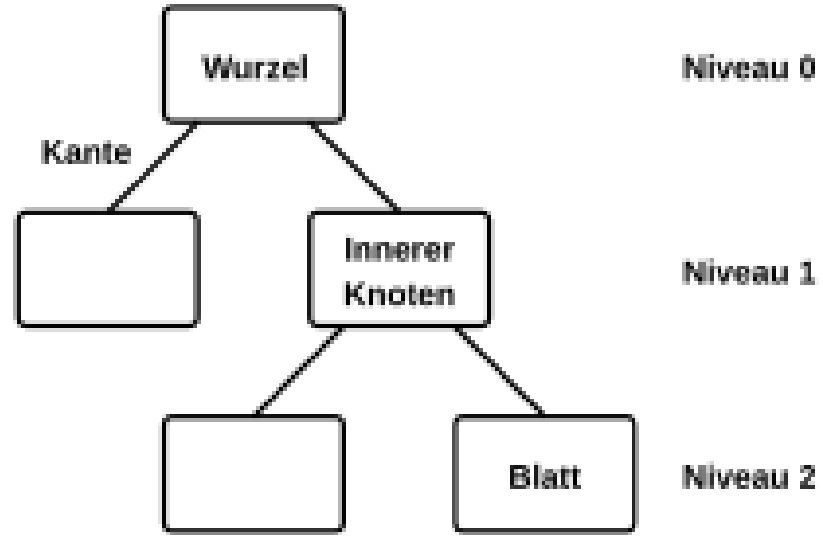
- In der Informatik ist ein Baum eine Datenstruktur, mit der sich hierarchische Strukturen abbilden lassen
- Ein Baum ist ein zusammenhängender kreisfreier ungerichteter Graph
- Ein Baum besteht aus
  - einer Menge von Knoten und
  - einer Menge von Kanten, die jeweils zwei Knoten verbinden



Baum

- Ein bestimmter Knoten des Baums wird als **Wurzel** bezeichnet: ein einzelner Knoten, der keine Vorfahren hat
- Analog sind **Blätter** die Knoten, die keine Nachfolger haben
- Zusätzlich gibt es **innere Knoten**, die Vorgänger („Elternknoten“) und Nachfolger („Kindsknoten“) haben

- Blätter und innere Knoten sind durch eine Kante mit genau einem anderen übergeordneten Knoten verbunden, jeder dieser Knoten hat also genau einen Elternknoten
  => Ein eindeutiger Pfad verläuft von der Wurzel zu jedem Knoten
  (insbesondere gibt es keine „losgelösten“ Knoten)
  => Es gibt keine Zyklen/„Kreise“



Allgemeine Suchprozedur

function GENERAL-SEARCH(*problem,strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion
    then return failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state
    then return corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

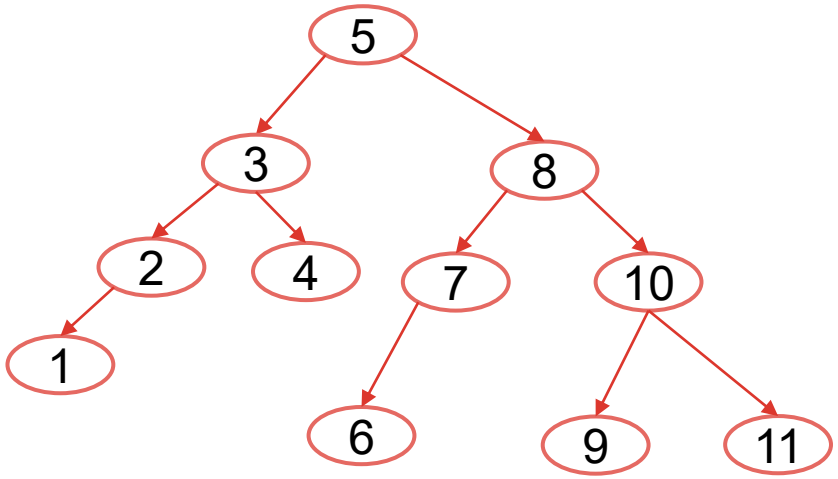
Implementierung des Suchbaums

Benötigte Datenstruktur für Knoten im Suchbaum sind in der Regel:

- State: Zustand des Zustandsraums
- Parent-Node: Vorgängerknoten
- Operator: Operator, der den aktuellen Knoten erzeugt hat
- Depth: Tiefe im Suchbaum
- Path-Cost: Pfadkosten bis zu diesem Knoten

Beispiel: Binärer Suchbaum

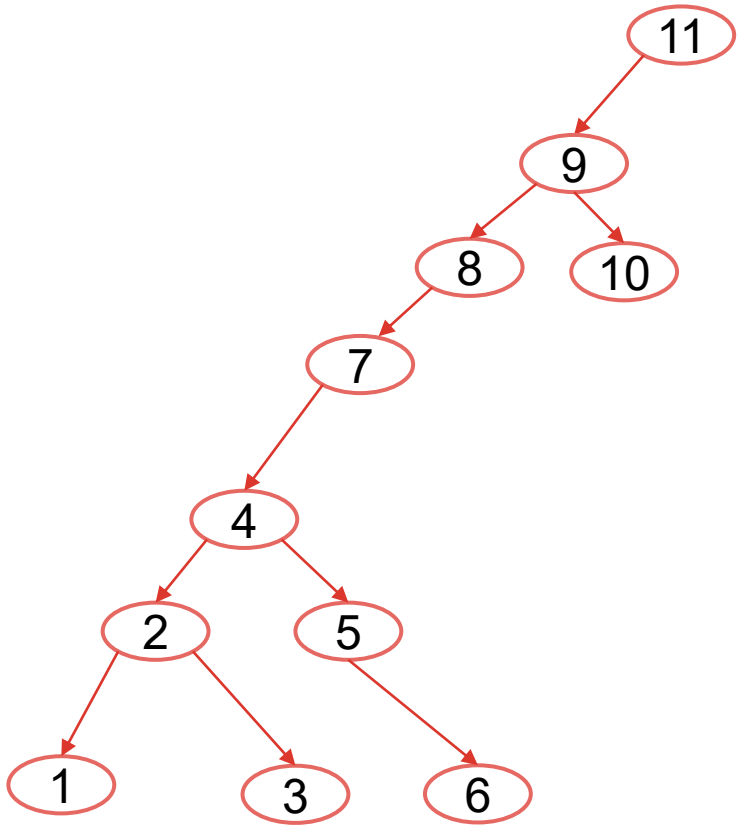
Falls jeder Knoten in einem Baum nur eine bestimmte Anzahl n von direkten Nachfolgern haben darf, die in einer bestimmten Reihenfolge vorliegen müssen, spricht man von einem n-ären Baum.
Im Binärbaume können die Knoten nur *höchstens zwei* direkte Nachkommen haben, z. B.:



Wie lange dauert die Suche nach einem Element?

Beispiel: Binärer Suchbaum

– Aufwand Suche bei ungünstiger Struktur: $\mathcal{O}(n)$



Beispiel: Binärer Suchbaum

- Sofern der Suchbaum ein geordneter Binärbaum ist, geht es schneller
- Suchprozess:
  - Starte bei der Wurzel des Baums
  - Ist der gesuchte Wert gleich dem Wert der Wurzel, gebe die Wurzel zurück
  - Ist der gesuchte Wert kleiner dem in der Wurzel, fahre rekursiv mit dem linken Teilbaum fort
  - Ist der gesuchte Wert größer dem in der Wurzel, fahre rekursiv mit dem rechten Teilbaum fort
- Im Worst Case benötigen wir für die Suche in einem geordneten Binärbaum T mit Höhe $H(T)$ dann $\mathcal{O}(H(T))$ Operationen (z. B. erfolglose Suche)

- Ist der Binärbaum zusätzlich auch balanciert, (alle Blattknoten haben in etwa die gleiche Tiefe) liegt der Aufwand der Suche bei $\mathcal{O}(\log_2(H(T)))$
- Es ist allerdings nicht ganz einfach, den Baum beim Einfügen balanciert zu halten (hierzu verwendet man oft spezielle Bäume, wie z. B. „B-Bäume“)

Agenda

1. Problemformulierung
2. Bewertungskriterien zur Unterscheidung von Suchverfahren
3. Allgemeine Suchstrategie und Datenstruktur
4. **Uninformierte Suche**
5. Informierte Suche

Suchstrategien

Uninformierte Suche (Blind Search):

– Entscheidungen basieren allein auf der Struktur des Suchraums (z. B. Tiefe, Reihenfolge)
– Der Agent hat **keine** zusätzlichen Informationen über den Abstand zum Ziel außer der Problemdefinition selbst (d. h. Anfangszustand, Operatoren, Zieltest)
– Es wird insbesondere keine Heuristik verwendet

Informierte / Heuristische Suche:

– Nutzt zusätzliche Informationen, die eine Schätzung der Entfernung oder der Kosten bis zum Ziel liefern (Heuristiken)
– Ziel: Suche effizienter machen, indem vielversprechende Zustände bevorzugt werden

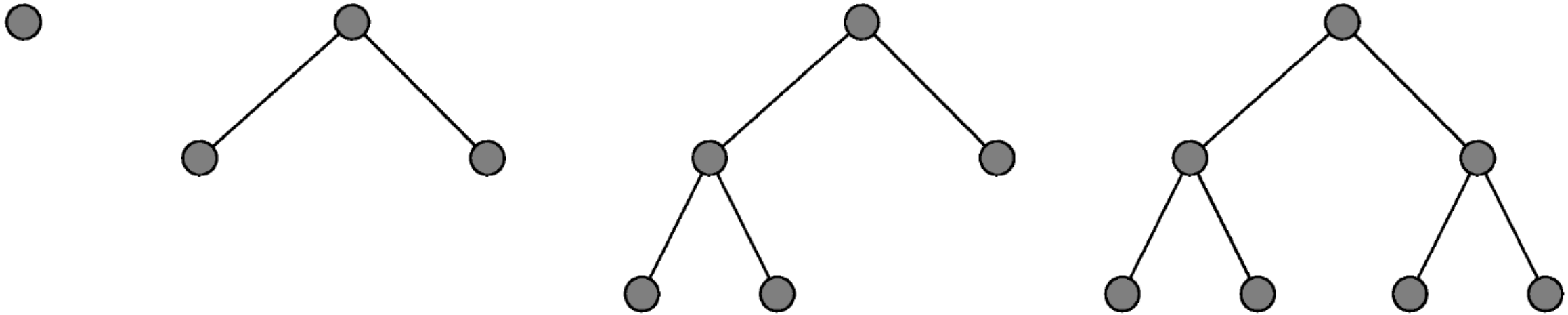
Uninformierte Suchverfahren

Typische Verfahren:

– Breitensuche, uniforme Kostensuche, Tiefensuche,
– tiefenbeschränkte Suche, iterative Tiefensuche,
– bidirektionale Suche

Breitensuche

– Expandiere Knoten in der Reihenfolge, in der sie erzeugt werden



– Findet immer die flachste Lösung (vollständig)
– Die Lösung ist optimal, wenn Pfadkosten eine nichtfallende Funktion der Knotentiefe ist (z. B. wenn jede Aktion identische, nichtnegative Kosten hat)

Breitensuche

- Allerdings sind die Kosten sehr hoch!
- Sei *b* der maximale Verzweigungsfaktor, *d* die Tiefe eines Lösungspfads
- Dann müssen maximal $1 + b + b^2 + b^3 + \ldots + b^d$ Knoten expandiert werden, also $\mathcal{O}(b^d)$

- Beispiel: $b = 10$, 1000 Knoten/s; 100 Bytes/Knoten:

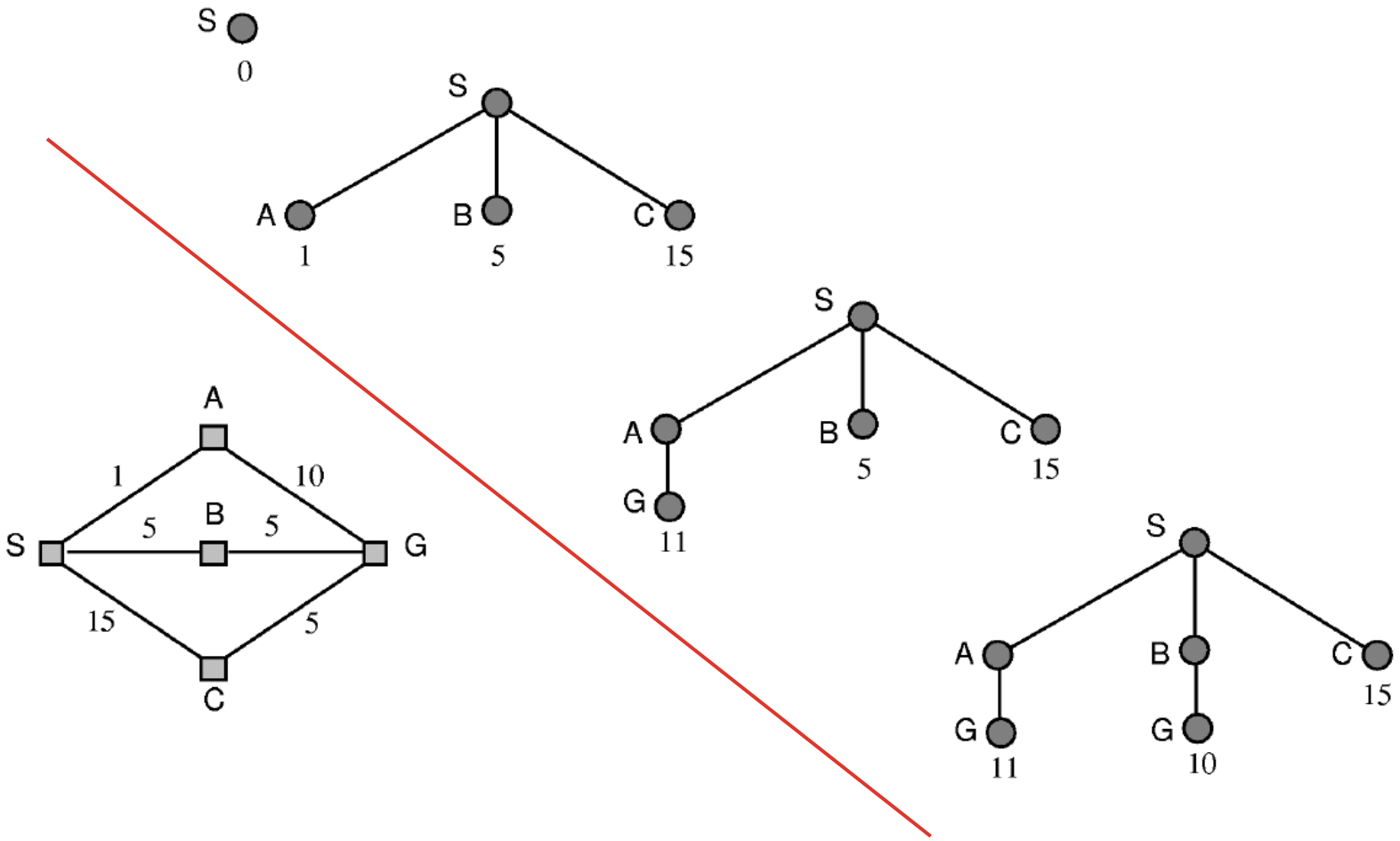
Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Uniforme Kostensuche

In Abwandlung der Breitensuche wird immer der Knoten n mit den geringsten Pfadkosten $g(n)$ expandiert.

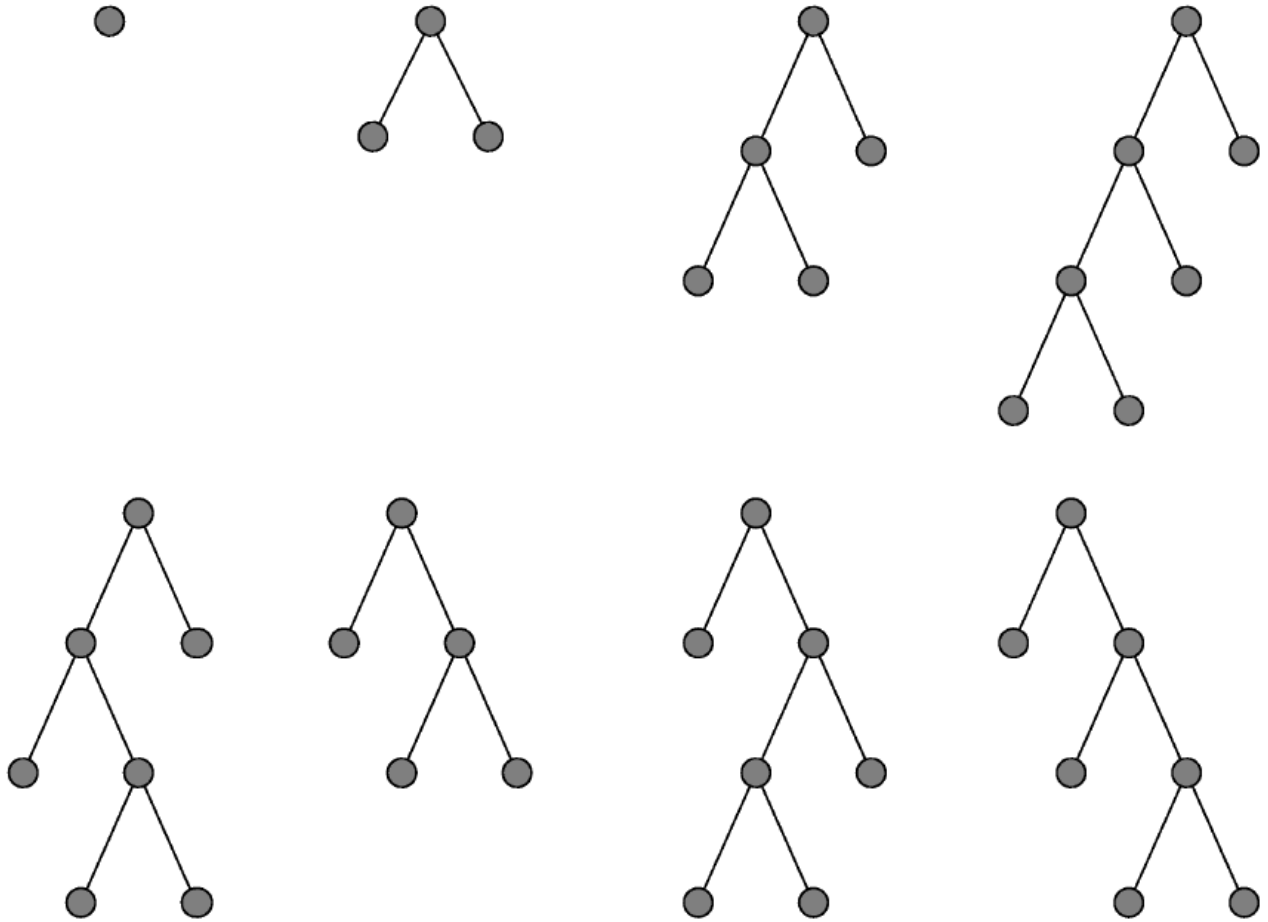
Findet immer die günstigste Lösung, falls $g(n+1) \geq g(n)$ für alle n.

Komplexität?



Tiefensuche

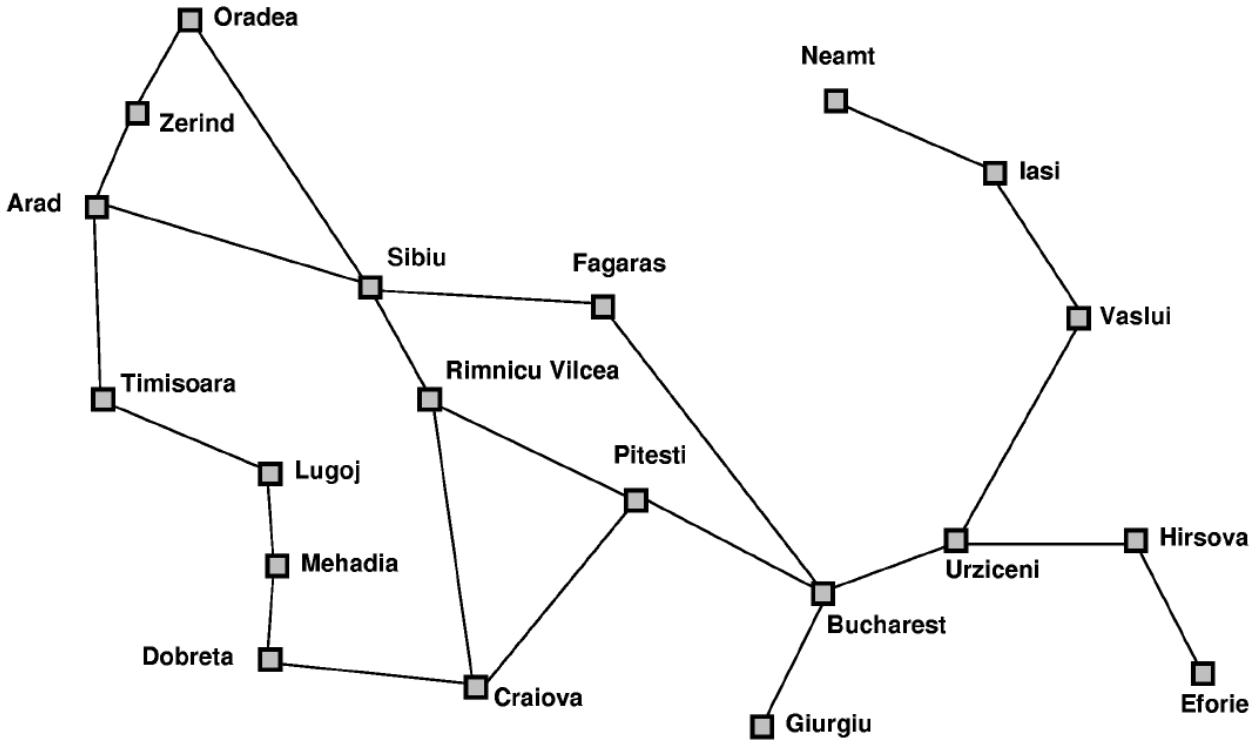
- Expandiere immer einen nicht expandierten Knoten mit maximaler Tiefe
- Beispiel (Knoten der Tiefe 3 haben keine Nachfolger):



- Komplexität?

Tiefenbeschränkte Suche

- Es wird nur bis zu einer vorgegebenen Pfadlänge Tiefensuche durchgeführt.
- z. B. Wegeplanung: Bei *n* Städten ist die maximale Tiefe *n* − 1
- Für den Weg von Arad nach Bucharest reicht die maximale Tiefe 9 (*Durchmesser* des Problems)

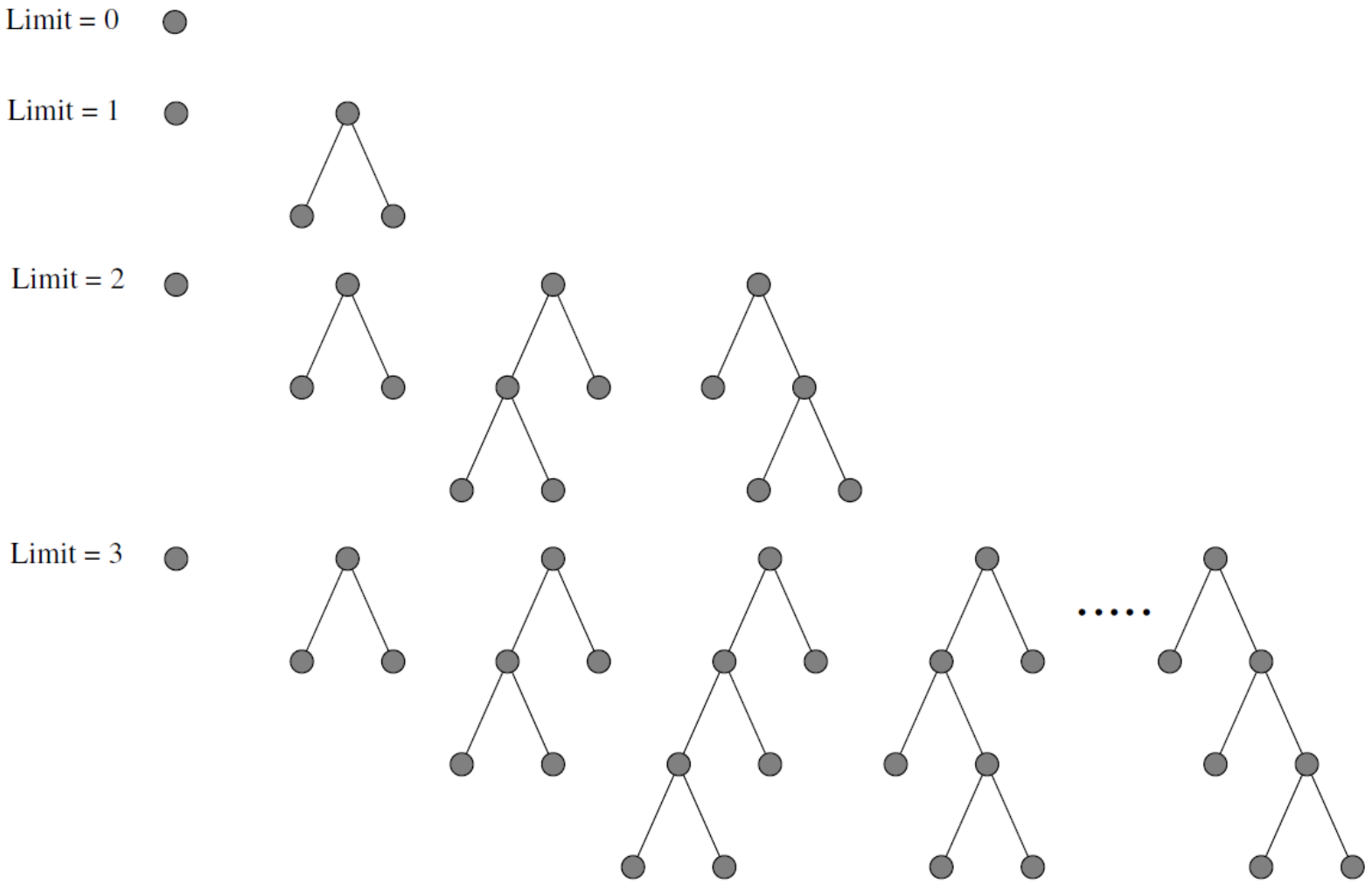


Iterative Tiefensuche

– kombiniert Tiefen- und Breitensuche
– optimal und vollständig wie Breitensuche, aber weniger Speicherplatzbedarf

function ITERATIVE-DEEPNING-SEARCH(*problem*) **returns** a solution sequence
inputs: *problem*, a problem
for *depth* ← 0 **to** ∞ **do**
    if DEPTH-LIMITED-SEARCH(*problem, depth*) succeeds
    then return its result
end
return failure

Iterative Tiefensuche - Beispiel



Komplexitätsbetrachtung

Zahl der Expansionen:

Iterative Tiefensuche	$(\mathrm{d}+1)+\mathrm{db}+(\mathrm{d}-1)\mathrm{b}^2+\ldots+3\mathrm{b}^{\mathrm{d}-2}+2\mathrm{b}^{\mathrm{d}-1}$
Breitensuche	$1+\mathrm{b}+\mathrm{b}^2+\mathrm{b}^3+\ldots+\mathrm{b}^{\mathrm{d}}$

Beispiel mit $b=10, d=5$

Iterative Tiefensuche	$6+50+400+3.000+20.000+100.000=123.456$
Breitensuche	1 + 10 + 100 + 1.000 + 10.000 + 100.000 = 111.111

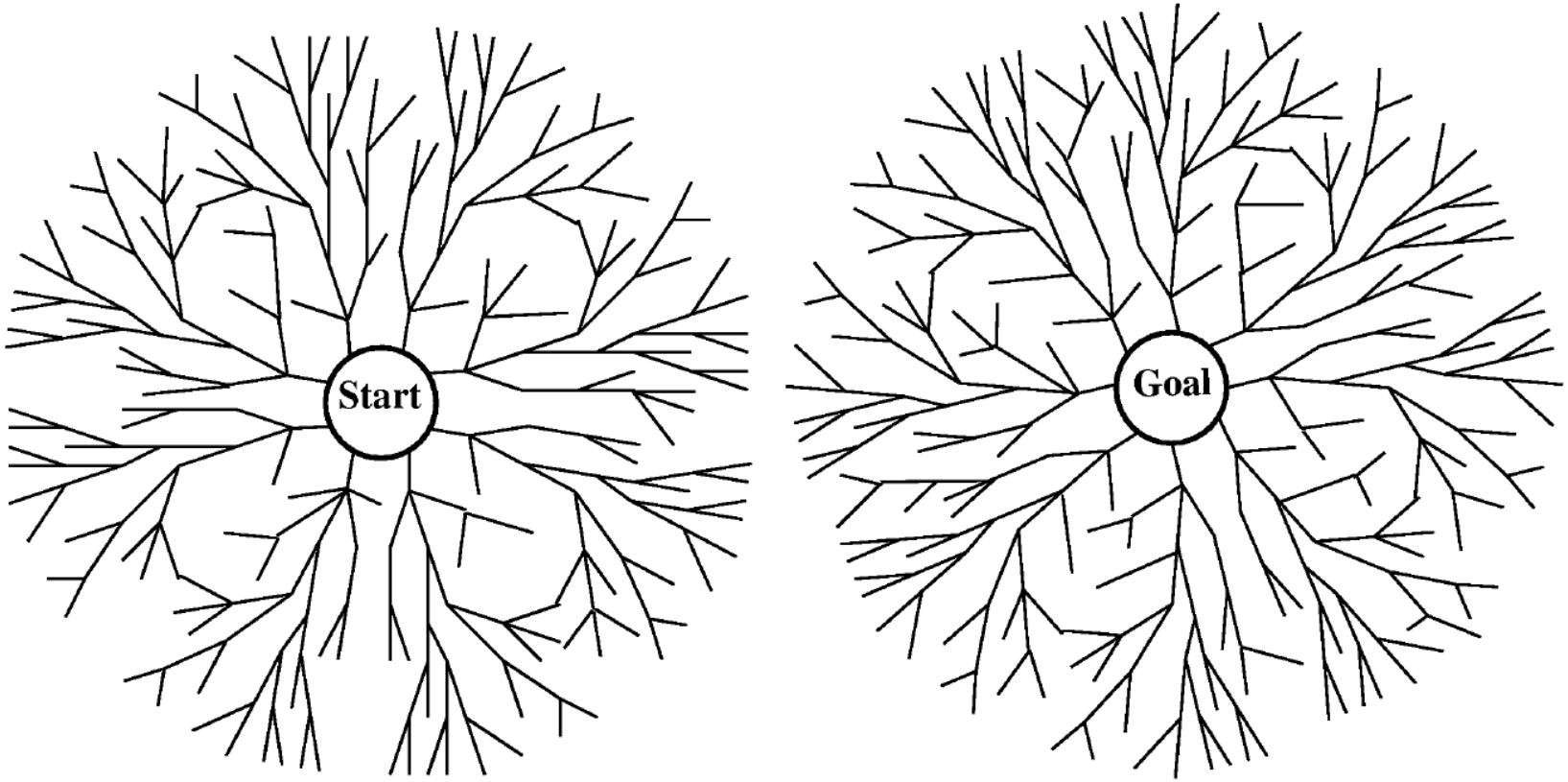
In diesem Fall werden bei iterativer Tiefensuche nur 11% mehr Knoten expandiert als bei der Breitensuche, wobei der Platzbedarf erheblich niedriger ist!

Zeitkomplexität: $O(\mathrm{b}^{\mathrm{d}})$, Platzkomplexität: $O(bd)$

=> Iterative Tiefensuche ist nicht viel schlechter und im Allgemeinen die bevorzugte Suchmethode bei großen Suchräumen mit unbekannter maximaler Suchtiefe

Bidirektionale Suche

- Sofern Vorwärts- und Rückwärtssuche symmetrisch sind, erreicht man Suchzeiten von $O(2\mathrm{b}^{\mathrm{d}/2}) = O(\mathrm{b}^{\mathrm{d}/2})$
- z. B. für $b = 10,\ d = 6$ statt 1.111.111 nur 2.222 Knoten!



Probleme mit bidirektionaler Suche

- Die Operatoren sind nicht immer oder nur sehr schwer umkehrbar (Berechnung der Vorgängerknoten).
- In manchen Fällen gibt es sehr viele Zielzustände, die nur unvollständig beschrieben sind. Beispiel: Vorgänger des „Schachmatt“
- Man braucht effiziente Verfahren, um zu testen, ob sich die Suchverfahren „getroffen“ haben
- Welche Art der Suche wählt man für jede Richtung (im Bild: Breitensuche, die nicht immer optimal ist)?

Vergleich der Suchstrategien

	Breiten-suche	Uniforme Kostensuche	Tiefensuche	Tiefen-beschränkte Suche	Iterative Tiefensuche	Bidirektionale Suche
Zeit-komplexität	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Platz-komplexität	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimalität	✅	✅	❌	❌	✅	✅
Vollständigkeit	✅	✅	❌	✅, wenn $l \geq d$	✅	✅

b: Verzweigungsfaktor,
d: Tiefe der Lösung,
m: maximale Tiefe des Suchbaums,
l: Tiefenlimit

Agenda

1. Problemformulierung
2. Bewertungskriterien zur Unterscheidung von Suchverfahren
3. Allgemeine Suchstrategie und Datenstruktur
4. Uninformierte Suche
5. **Informierte Suche**

Suchstrategien

Uninformierte Suche (Blind Search):

– Der Agent hat **keine** zusätzlichen Informationen über den Abstand zum Ziel
– Es wird insbesondere keine Heuristik verwendet

Informierte / Heuristische Suche:

– Nutzt zusätzliche Informationen, die eine Schätzung der Entfernung oder der Kosten bis zum Ziel liefern (Heuristiken):
  Information über Kosten von gegebenem Knoten bis zum Ziel in Form einer Evaluierungsfunktion h, die jedem Knoten eine reelle Zahl zuweist
– Ziel: Suche effizienter machen, indem vielversprechende Zustände bevorzugt werden

Bestensuche (best-first search)

– Suchverfahren, das Knoten mit dem „besten“ *h*-Wert expandiert
– Generischer Algorithmus:

function BEST-FIRST-SEARCH(*problem*, EVAL-FN) **returns** a solution sequence
  inputs: *problem*, a problem
    Eval-Fn, an evaluation function

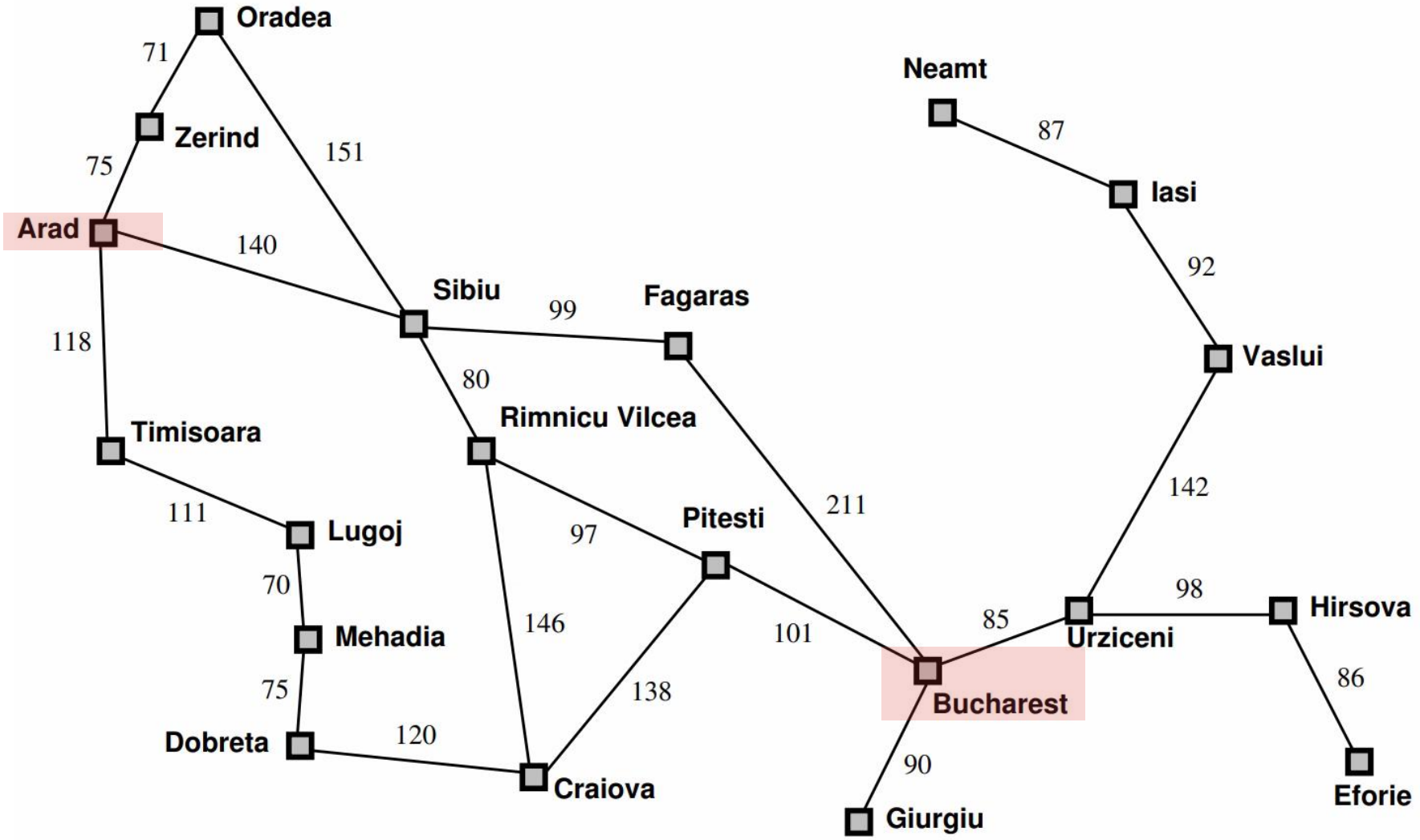
  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(*problem*, *Queueing-Fn*)

– Doch wie sieht Eval-Fn aus?

Gierige Suche (Greedy Search)

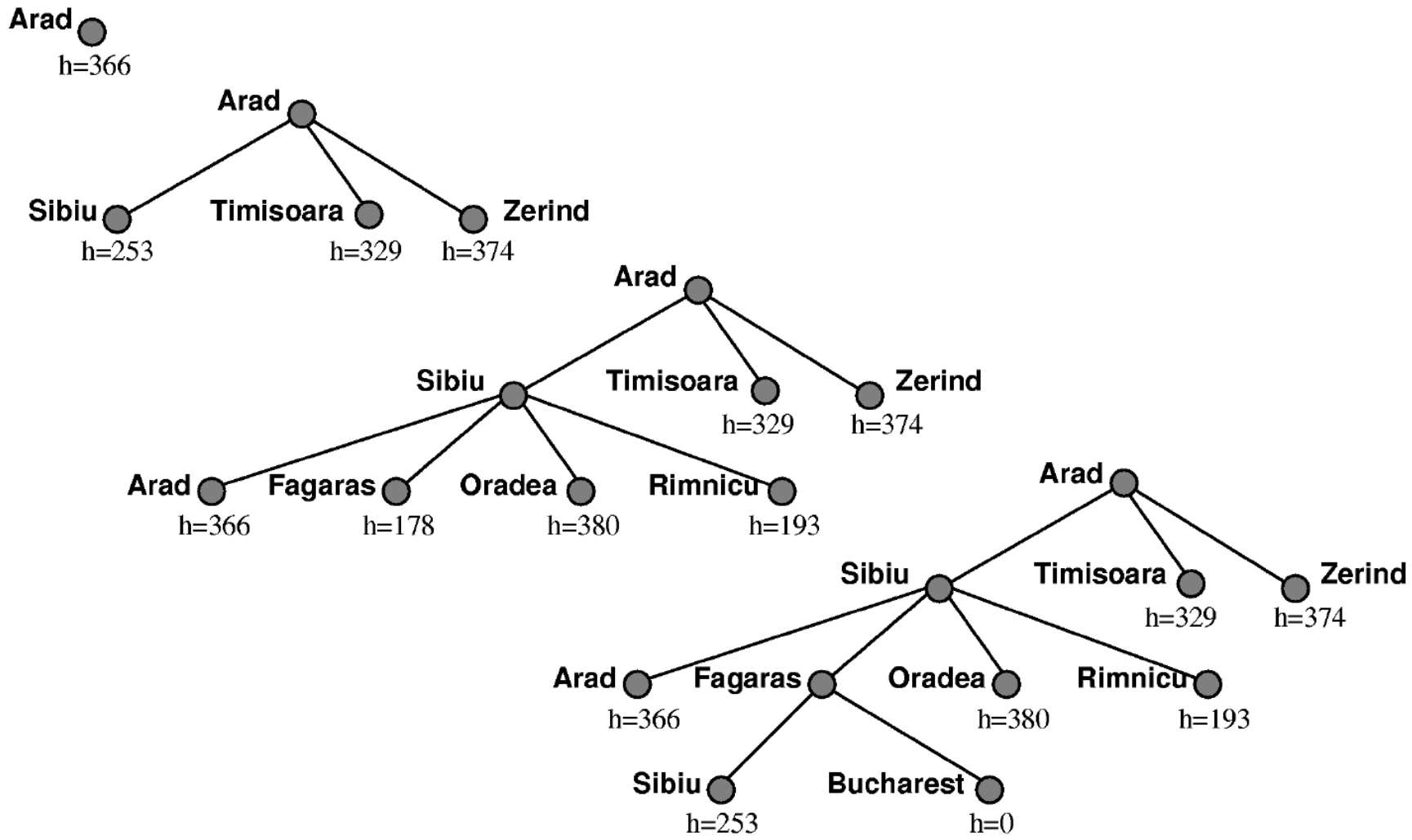
- Eine Möglichkeit die „Güte“ von Knoten zu beurteilen ist es, ihren Abstand zum Ziel zu schätzen
- $h(n)$ = geschätzter Abstand von n zum Ziel
- Einschränkung für h: $h(n) = 0$ falls n ein Zielknoten ist
- Eine Bestensuche mit dieser Funktion heißt „gierige Suche“
- Beispiel Routensuche: h = Luftlinienentfernung zwischen zwei Orten

Beispiel für gierige Suche von Arad nach Bucharest



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 von Arad nach Bucharest



Heuristiken

- Die Evaluierungsfunktion h im Falle gieriger Suche wird auch heuristische Funktion oder Heuristik genannt
- Das Wort Heuristik stammt aus dem Griechischen Wort für „auffinden, entdecken“
- Heuristiken beschleunigen im Normalfall die Suche aber die Suche ist im Allgemeinen nicht vollständig
- Eine Heuristik ist spezifisch für eine Klasse von Problemen (z. B. kürzeste Wege) und fokussiert die Suche

A*: Minimierung der geschätzten Pfadkosten

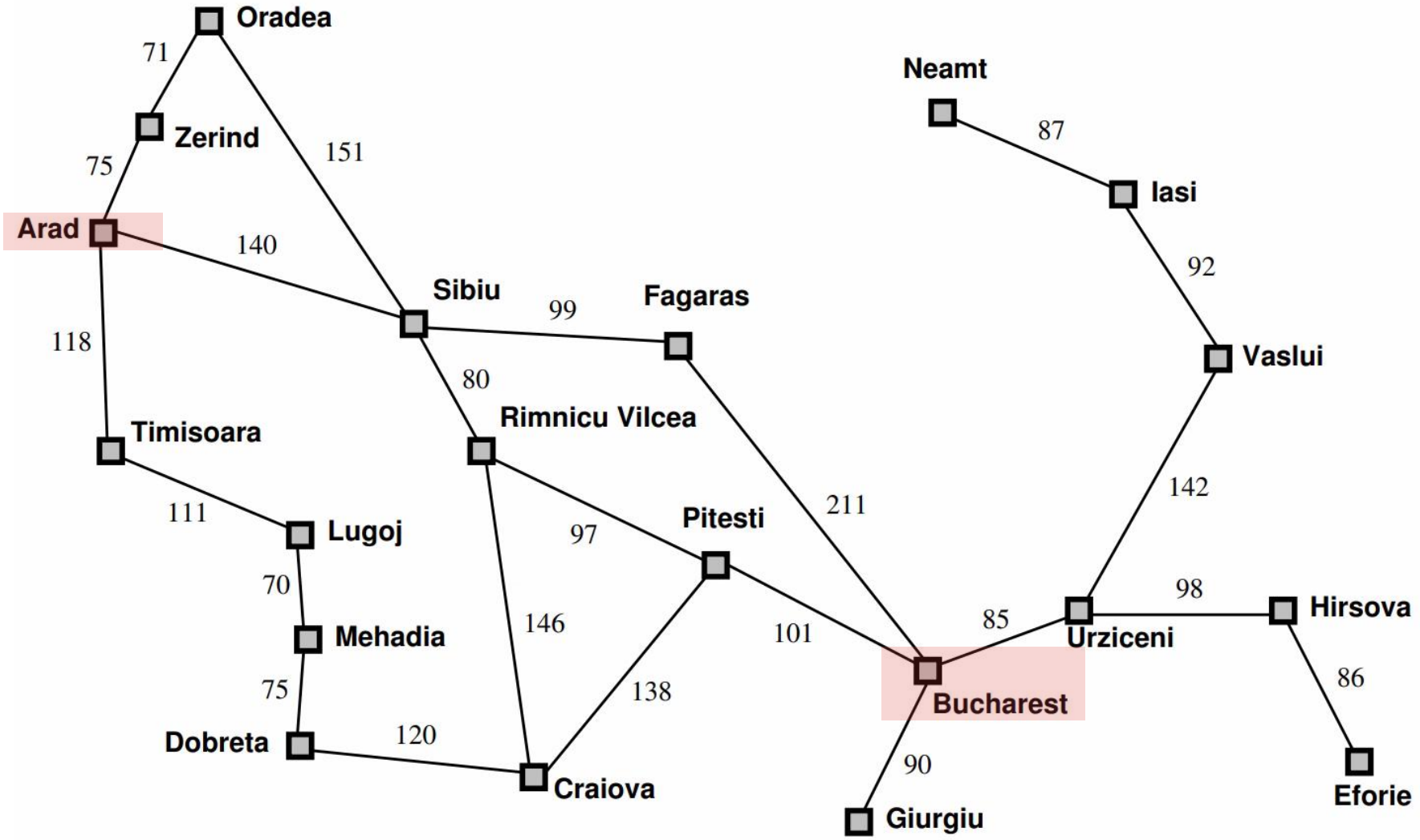
- A* verbindet uniforme Kostensuche mit gieriger Suche
- $g(n) =$ tatsächliche Kosten vom Anfangszustand bis n
- $h(n) =$ geschätzte Kosten von n bis zum nächstgelegenen Ziel
- $f(n) = g(n) + h(n)$, d. h. geschätzte Kosten des günstigsten Pfades, der durch n verläuft.

- Seien $h*(n)$ die tatsächlichen Kosten des optimalen Pfades von n zum nächstgelegenen Ziel
- h heißt zulässig, wenn für alle n gilt:

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

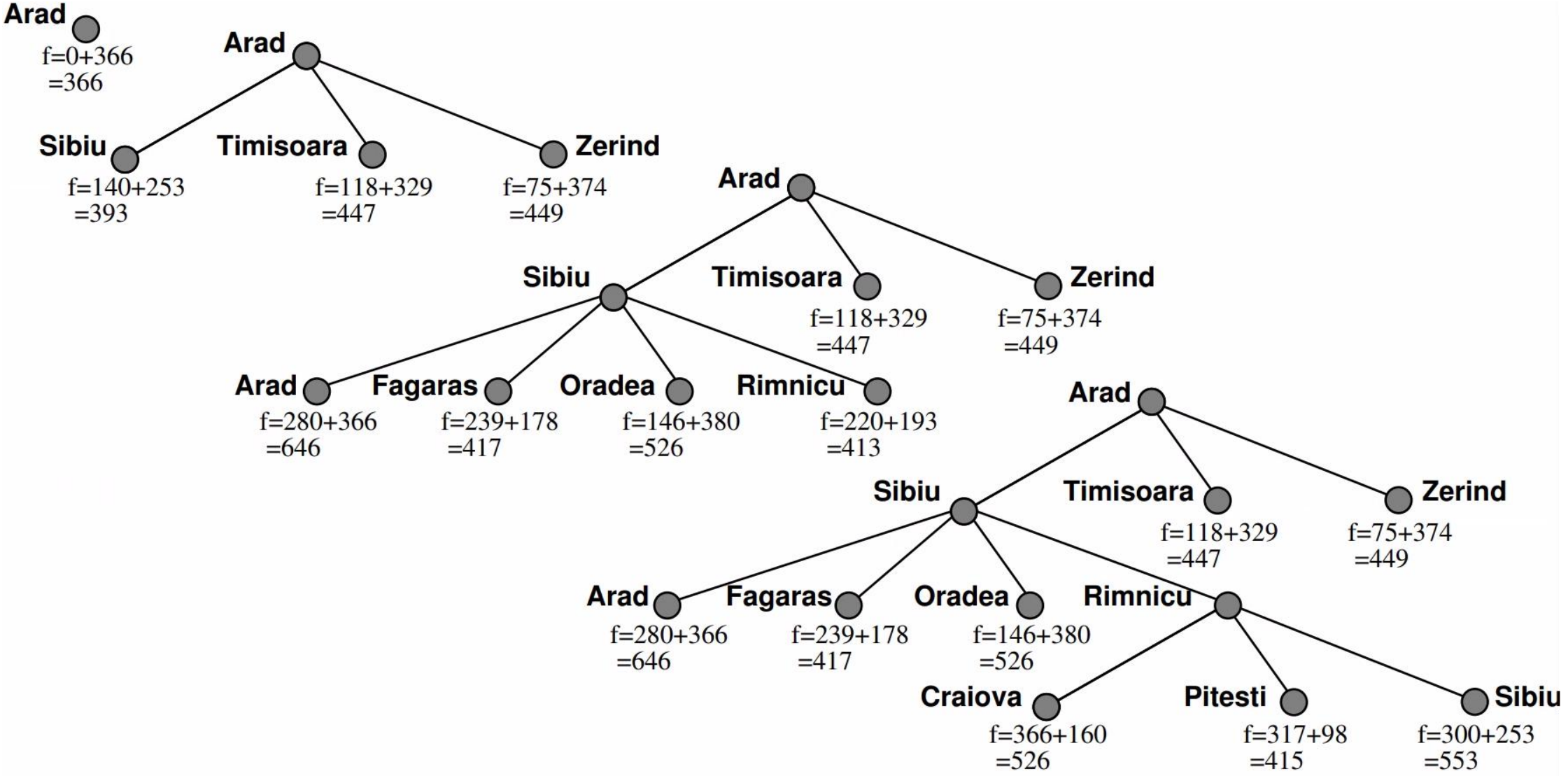
- Wir verlangen für A*, dass h zulässig ist (Luftlinienentfernung ist z. B. zulässig, weil kleiner oder gleich dem optimalen Pfad)

Beispiel für A*-Suche von Arad nach Bucharest



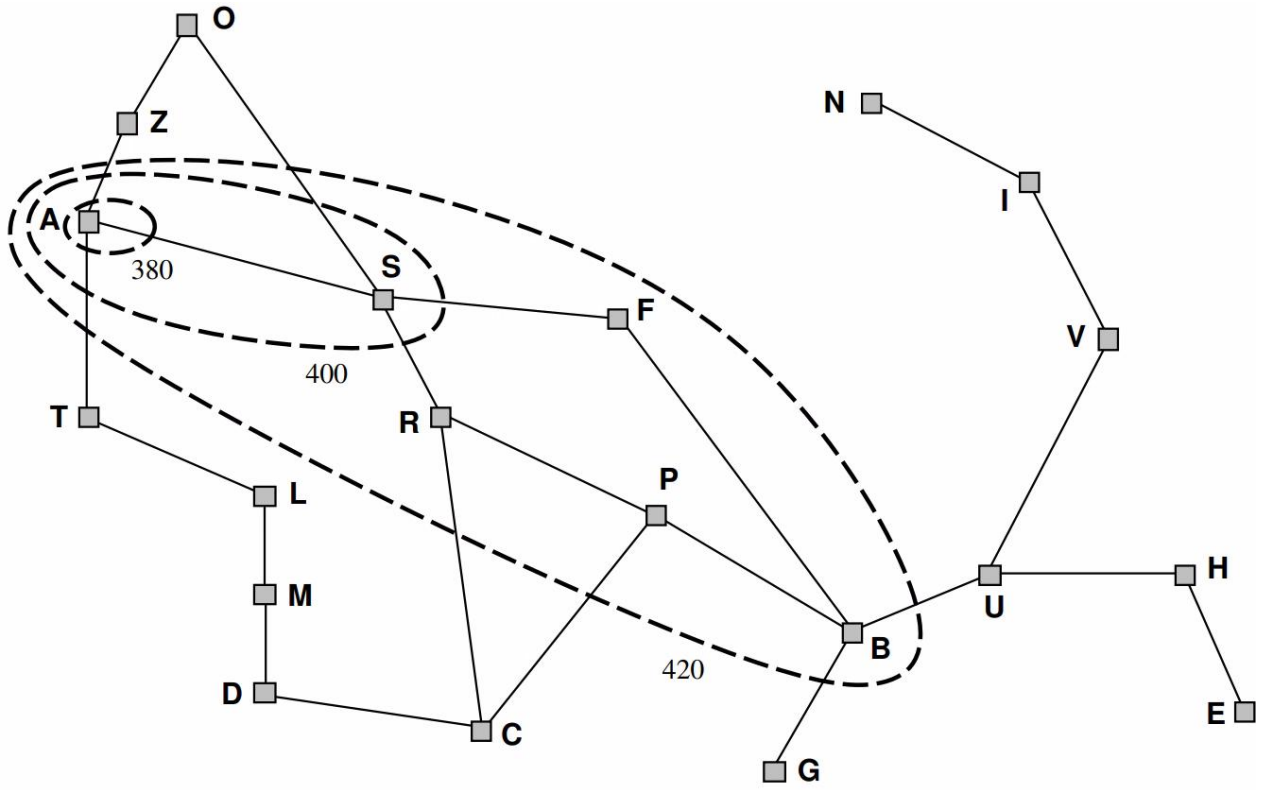
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Beispiel für A*-Suche von Arad nach Bucharest

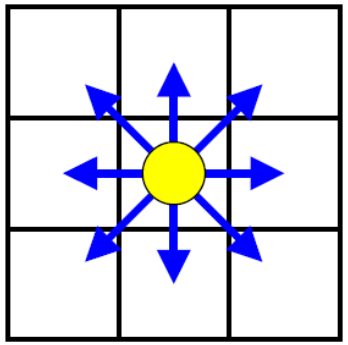


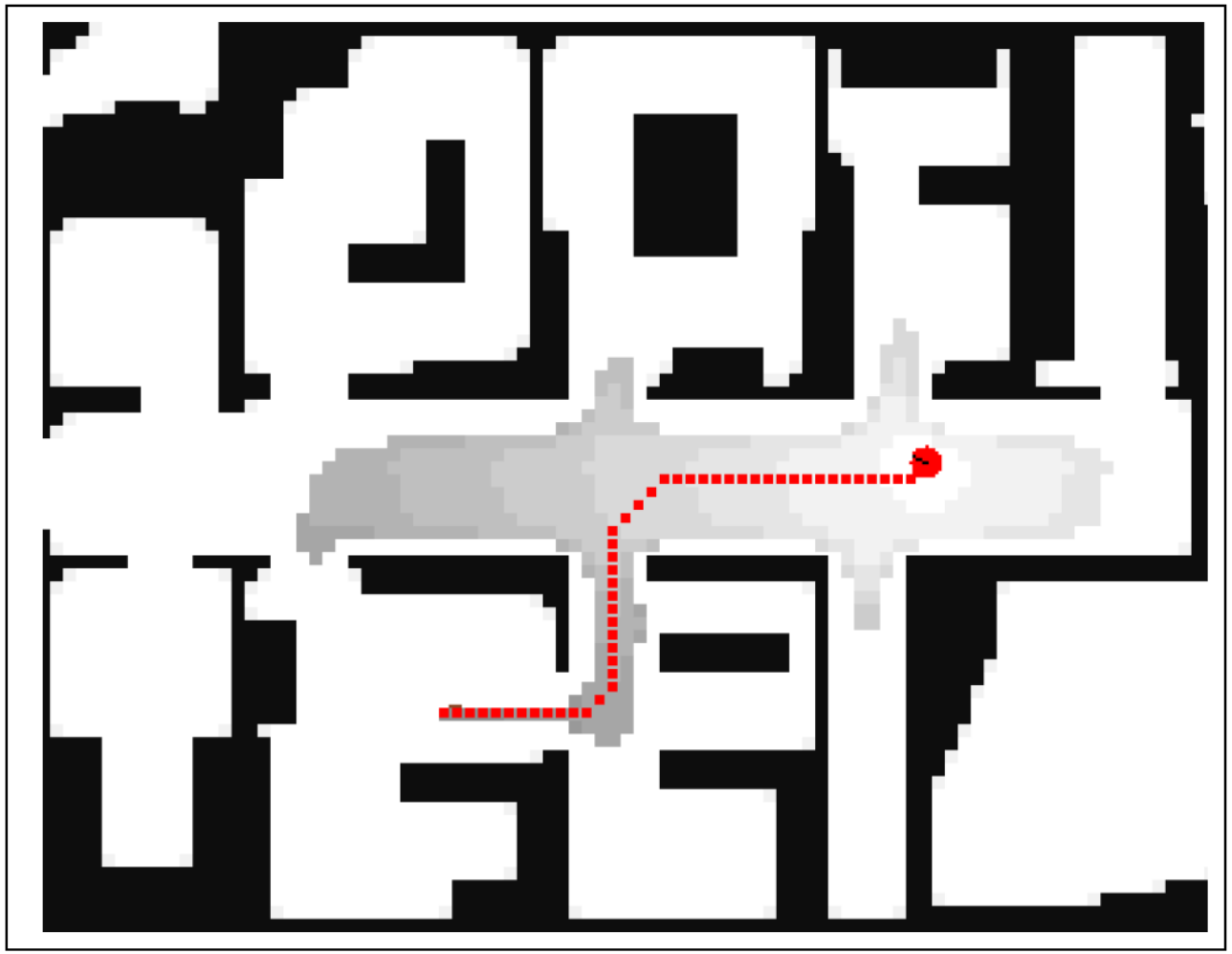
A*-Suche anschaulich

– Innerhalb des Suchraums ergeben sich Konturen, in denen jeweils für einen gegebenen f-Wert alle Knoten expandiert werden (hier Konturen für $f = 380, 400, 420$):

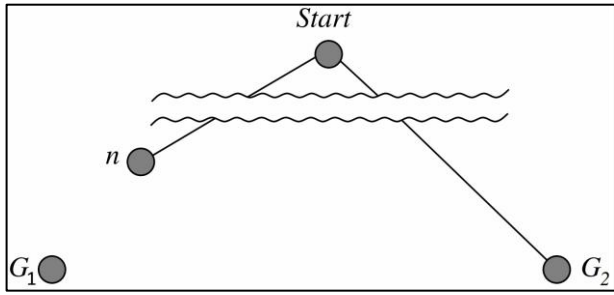


Beispiel: A* zur Pfadplanung für Roboter in einer Grid-Welt





Optimalität von A*



Behauptung: Die erste von A* gefundene Lösung ist eine mit minimalen Pfadkosten
Beweis:
Wir behaupten das Gegenteil, nämlich die Möglichkeit, dass A* einen suboptimalen Knoten G_2 als Lösung gefunden hat:
Sei G_1 ein Zielknoten mit optimalen Pfadkosten K^* und der Knoten G_2 eine suboptimale Lösung.

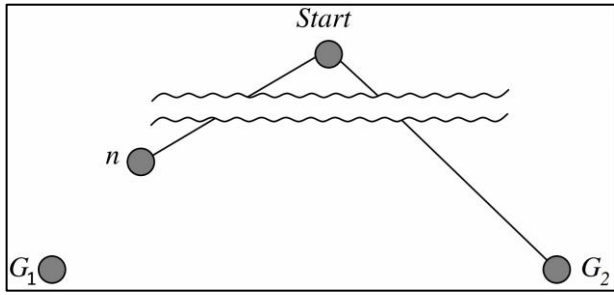
Da die Heuristik die Kosten bis zum Ziel nie überschätzt, gilt für jeden Zielknoten und insbesondere für G_2:

$$h(G_2) = 0$$

Da G_2 eine suboptimale Lösung ist, gilt für die Kosten C_2:

$$C_2 = f(G_2) = g(G_2) + h(G_2) = g(G_2) > K*$$

Optimalität von A*



Sei n ein beliebiger Knoten auf dem optimalen Pfad vom Start nach G_1, der noch nicht expandiert wurde. Die Heuristik schätzt mit $\mathrm{h(n)}$ die tatsächlichen Kosten oder unterschätzt sie. Also gilt:

$$\mathrm{f(n) = g(n) + h(n) \leq K*}$$

Da n nicht vor G_2 expandiert wurde, muss gelten

$$\mathrm{f(G_2) < f(n)}$$

und somit:

$$\mathrm{f(G_2) \leq K^*}$$

Das ist allerdings ein Widerspruch zu unserer ursprünglichen Annahme, dass G_2 eine suboptimale Lösung ist!

Optimalität von A*

Dies bedeutet insgesamt, dass der A*-Algorithmus die suboptimale Lösung G_2 nicht wählt, solange sich Knoten eines optimalen Pfades in der Liste der zu evaluierenden Knoten befinden (da bei jedem Schritt der Knoten mit minimalem f-Wert erweitert wird).

Eine suboptimale Lösung würde also erst gewählt werden, nachdem die Knoten jeder optimalen Lösung besucht worden wären.

Dazu kommt es nicht, da der A*-Algorithmus stets die erste gefundene Lösung ausgibt und dann terminiert.

Vollständigkeit und Komplexität

Vollständigkeit:

A* findet eine Lösung, falls es eine gibt, unter der Voraussetzung, dass

- jeder Knoten nur *endlich* viele Nachfolgerknoten hat und
- es eine positive Konstante δ gibt, so dass jeder Operator mindestens die Kosten δ hat.

=> nur endlich viele Knoten n mit $\mathrm{f}(n) \leq \mathrm{K} *$

Komplexität:

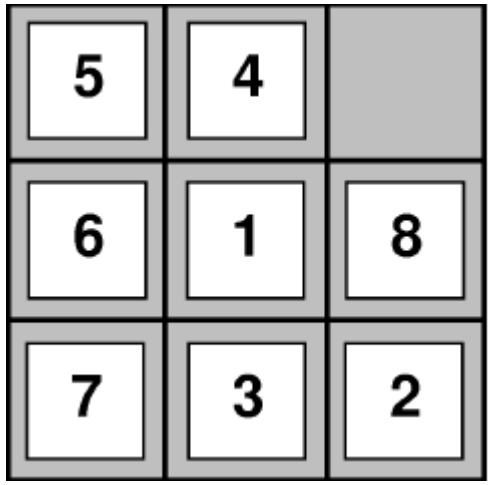
a) Normalfall: Exponentielles Wachstum, weil der Fehler proportional zu den Pfadkosten ist
b) Wenn die Heuristik $h(n)$ sehr genau ist, also der Unterschied zum wahren Wert $h*(n)$ höchstens logarithmisch wächst, dann gilt: $|h*(n) - h(n)| \leq \mathcal{O}(\log(h*(n)))$
   In diesem Fall werden *subexponentiell* viele Knoten expandiert.

Mit anderen Worten: **Je genauer die Heuristik, desto langsamer wächst der Suchbaum.**
Wenn die Heuristik ungenau ist, kommen viele weitere Knoten in Betracht und müssen evaluiert werden, was den Aufwand deutlich erhöht.

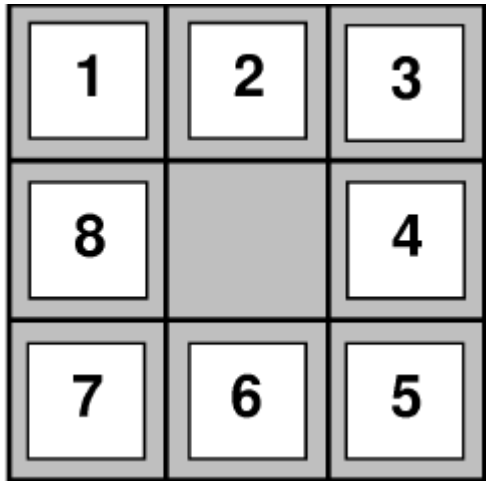
Wiederholung: Beispiel Schiebepuzzle

Zustände:

- Beschreibung der Lage jedes der 8 Kästchen und (aus Effizienzgründen) des Leerkästchens
- Operator: „Verschieben“ des Leerkästchens nach links, rechts, oben und unten
- Zieltest: Entspricht aktueller Zustand dem rechten Bild?
- Pfadkosten: Jeder Schritt kostet 1 Einheit



Startzustand



Zielzustand

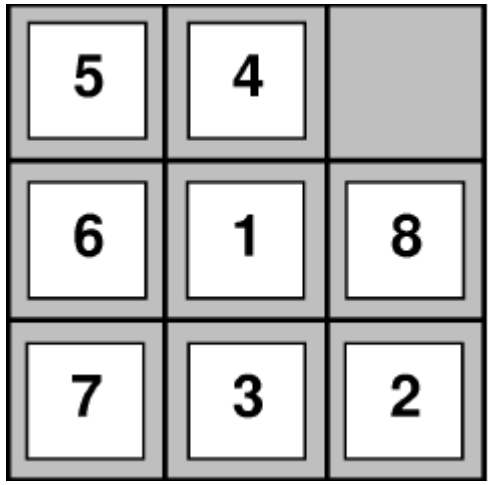
=> Lösung?

Heuristik für Schiebepuzzle

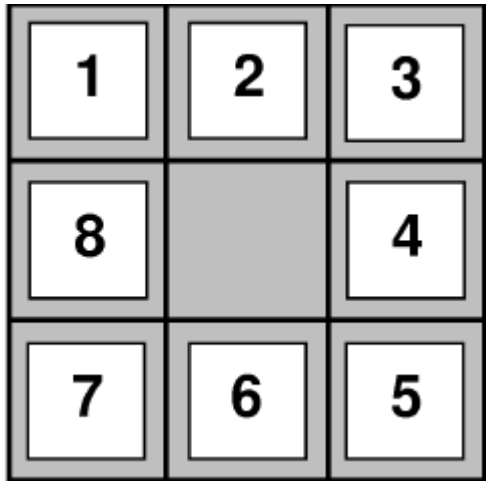
Vorschlag:

- h_1 = Zahl der Kästchen in falscher Position
- h_2 = Summe der Distanzen der Kästchen zu ihrer Zielposition (*Manhattan-Distanz*)

Welche heuristische Funktion bevorzugen Sie?



Startzustand



Zielzustand

Empirische Auswertung Schiebepuzzle

- d= Abstand vom Ziel (wie „schlecht“ ist der Ausgangszustand)
- Gezeigt ist jeweils der Durchschnitt über 100 Instanzen

	Search Cost			Effective Branching Factor		
d	IDS	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

Iterative A*-Tiefensuche: IDA*

- Weitere Varianten möglich, z. B.: Kombination von IDS und A*
- man durchsucht wiederholt f-Konturen per Tiefensuche, beginnend mit der Schranke $h(Start)$
- in jeder Iteration wird die Schranke auf das kleinste überschrittene f angehoben

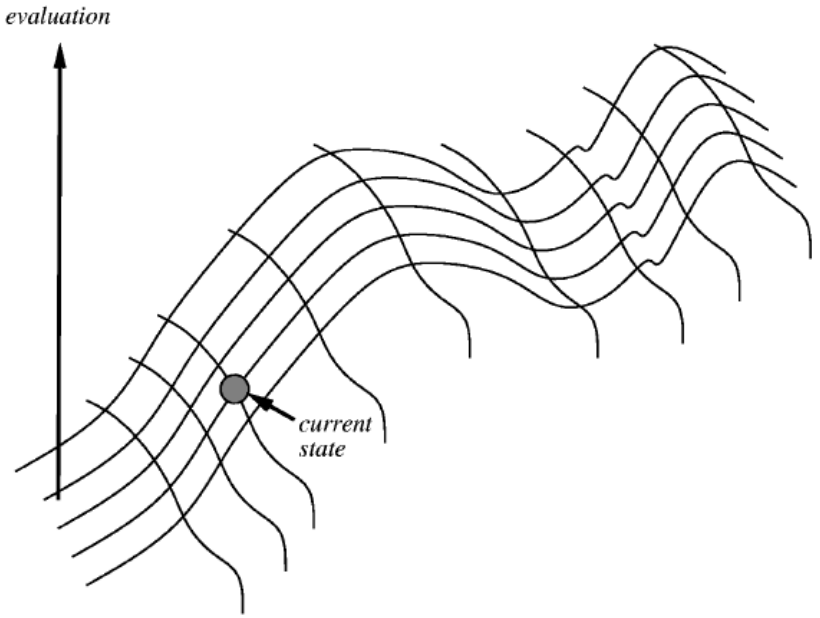
```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f- COST limit
          root, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f- COST(root)
  loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f- COST limit
  inputs: node, a node
          f-limit, the current f- COST limit
  static: next-f, the f- COST limit for the next contour, initially ∞

  if f- COST[node] > f-limit then return null, f- COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
  return null, next-f
```

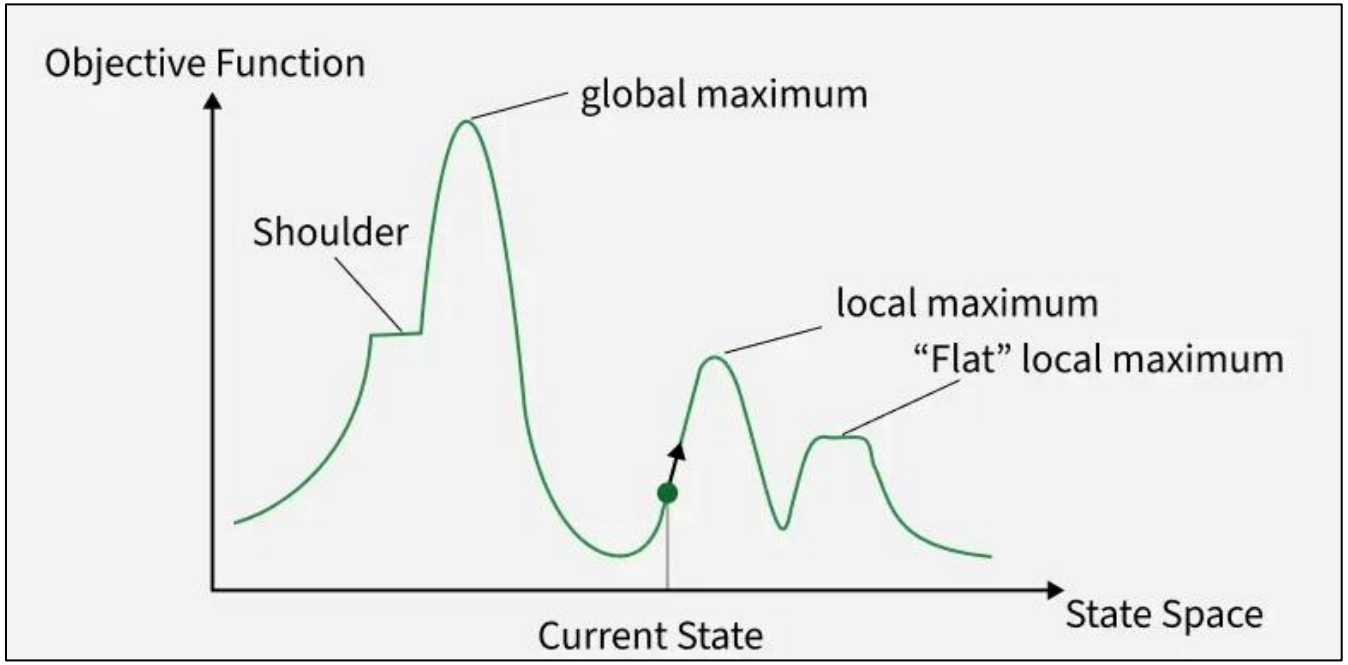
Lokale Suche

- Für viele Probleme ist es irrelevant, wie man zum Zielzustand kommt – nur der Zielzustand selber ist interessant (8-Damen Problem, VLSI Design, TSP)
- Wenn sich außerdem ein Qualitätsmaß für Zustände angeben lässt, kann man lokale Suche benutzen, um Lösungen zu finden
- Idee: Man fängt mit einer zufällig gewählten Konfiguration an und verbessert diese schrittweise
  => „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 current
     current ← next
  end
```



Probleme bei lokaler Suche



Probleme:

– Lokale Maxima: Der Algorithmus gibt eine suboptimale Lösung aus
– Plateaus und Grate: Hier kann der Algorithmus nur zufällig herumwandern

Verbesserungsmöglichkeiten:

– Neustarts, wenn keine Verbesserung mehr
– Rauschen „injizieren“ („Random Walk“)
– Tabu-Suche: Die letzten n angewandten Operatoren erstmal nicht mehr anwenden

Welche dieser Strategien (mit welchen Parametern) erfolgreich sind (auf einer Problemklasse), kann man meist nur empirisch bestimmen

Treshold Accepting

- Treshold Accepting ist eine einfache lokale Suche, bei der auch kleinere Verschlechterungen der Lösung akzeptiert werden
- Es akzeptiert also alle Variationen, bei denen die Verschlechterung unterhalb eines Schwellenwerts *T* bleibt

```
Wähle einen anfänglichen Schwellwert T > 0
Wähle einen Schwellwertfaktor TF (0 < TF < 1)
Wähle eine anfängliche Konfiguration C
C_best := C
Wiederhole bis Abbruchbedingung erfüllt
   Wähle neue Konfiguration C' (ausgehend von C)
   Falls Qualität(C') > Qualität(C_best)
      C_best := C'
   Falls Qualität(C') > Qualität(C) - T
      C := C'
   T := T * TF
Gib C_best aus
```

Simuliertes Abkühlen

- Im Simulated-Annealing-Algorithmus (SA) erfolgt das „Injizieren“ von Rauschen ebenfalls wie in Threshold Accepting systematisch: Erst stark, dann abnehmend
- SA akzeptiert Verschlechterungen (unter Berücksichtigung der Threshold) allerdings nur mit einer bestimmten – im Verlauf der Optimierung kleiner werdenden – Wahrscheinlichkeit (Threshold Accepting immer)

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*
    Δ*E* ← VALUE[*next*] – VALUE[*current*]
    if Δ*E* > 0 **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Simuliertes Abkühlen

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*
    Δ*E* ← VALUE[*next*] – VALUE[*current*]
    if Δ*E* > 0 **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

HOCHSCHULE MAINZ
Wirtschaft
Business

Vielen Dank für Ihre Aufmerksamkeit!

Prof. Dr. Dirk Schweim
Professur für Wirtschaftsinformatik

Hochschule Mainz
University of Applied Sciences
Lucy-Hillebrand-Str. 2
55128 Mainz, Germany

T +49 6131 628-3315
E Schweim@HS-Mainz.de
W www.hs-mainz.de/schweim

University of
Applied Sciences