3. Suche als Problemlöseverfahren

1. Was ist KI?
2. Logik und Inferenz
3. **Suche als Problemlöseverfahren**
4. Schließen unter Unsicherheit
5. Maschinelles Lernen
6. Ausblick: „Rationale“ Roboter

Suche in der Informatik

Typische Problemstellung

„Ist ein Datensatz in einer Datenbank vorhanden?“

Naive Lösung

Alle Datensätze der Reihe nach durchsuchen.

Zeit: $O(n)$ für n Datensätze in der DB

Info A, Kap. 6

Bessere Lösung (wie in Informatik A gelernt)

Datensätze clever sortiert speichern (z.B. Baum);
Sortierung beim Suchen nutzen.

Zeit: $O(\log n)$

➥ D.E. Knuth, Bd.3: *Sorting and Searching*; Informatik A

Suche in der KI

Typische Problemstellung

Ziele verfolgender Agent überlegt den nächsten Schritt,
und den Folgeschritt,
und den … … — bis zum Ziel

Lösungsweg konstruieren, nicht Ziel nachschlagen!

Naive Lösung

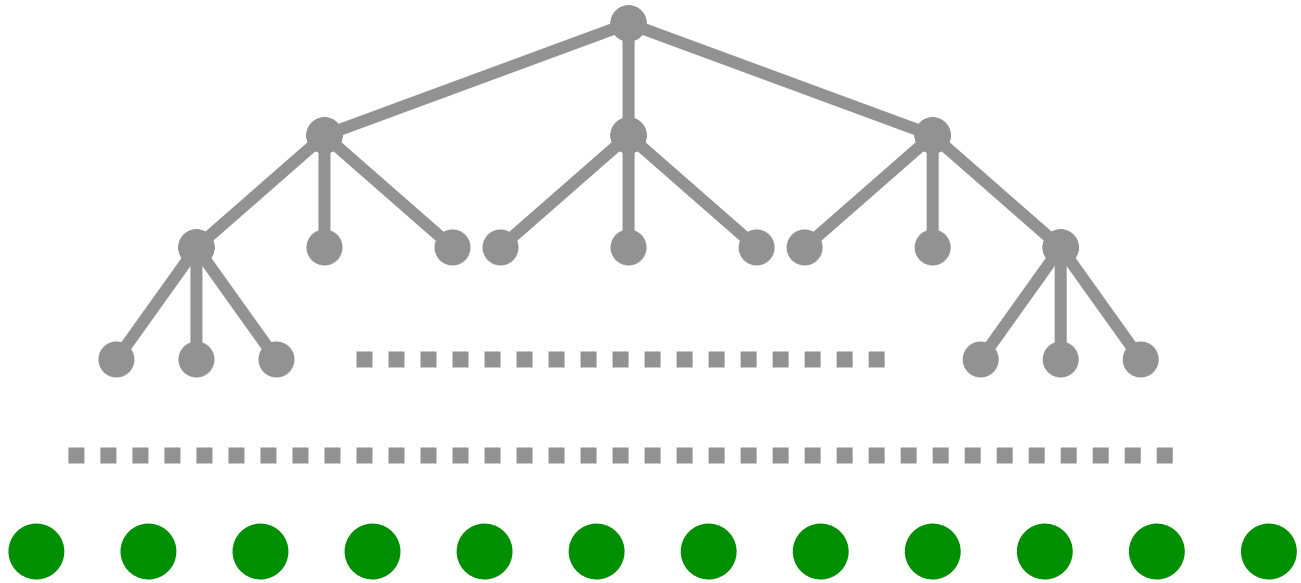
Alle Sequenzen von Schritten der Reihe nach durchprobieren.

Zeit: …

Exkurs über Knoten und Blätter in Bäumen

Verzweigungsfaktor b=3

Tiefe



d=0

d=1

d=2

d=3

d

$O(b^d)$ # Knoten der Tiefe d: b^d

$O(b^d)$ # alle Knoten bis einschl. Tiefe d:

$$\sum_{i=0}^{d} b^i = \frac{b^{d+1}-1}{b-1}$$

Ende Exkurs!

Komplexität der naiven Suche

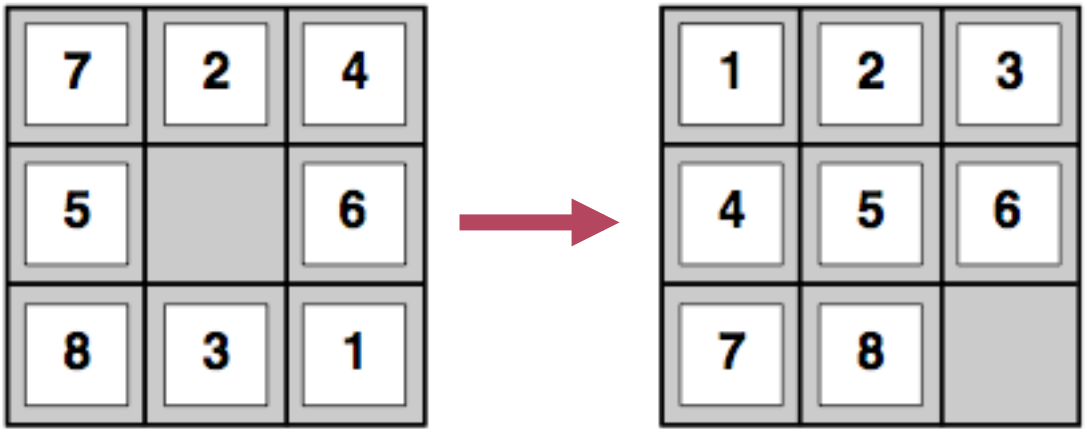
Alle Sequenzen von Schritten der Reihe nach durchprobieren.

Zeit: $O(b^d)$ bei „erster“ Lösung in Tiefe d

Speicher: dito (alle Knoten im Speicher)

Bessere/Andere Lösungen … folgen!

Beispielproblem I: Verschiebespiel



Startzustand **Zielzustand**

Zustand

Sequenz der Zahlen/ Leerfeld auf den 9 Feldern

Aktionen

Left, Right, Up, Down (Verschiebung des Leerfelds, wenn‘s geht)

Kosten

Konstant (1) pro Aktion

Suchproblem

6.1

Definition 6.2 *Ein Suchproblem wird definiert durch folgende Größen*

Zustand: *Beschreibung des Zustands der Welt, in dem sich ein Suchagent befindet.*

Startzustand: *der Initialzustand, in dem der Agent gestartet wird.*

Zielzustand: *erreicht der Agent einen Zielzustand, so terminiert er und gibt (falls gewünscht) eine Lösung aus.*

Aktionen: *Alle erlaubten Aktionen des Agenten.*

Lösung: *Der Pfad im Suchbaum vom Startzustand zum Zielzustand.*

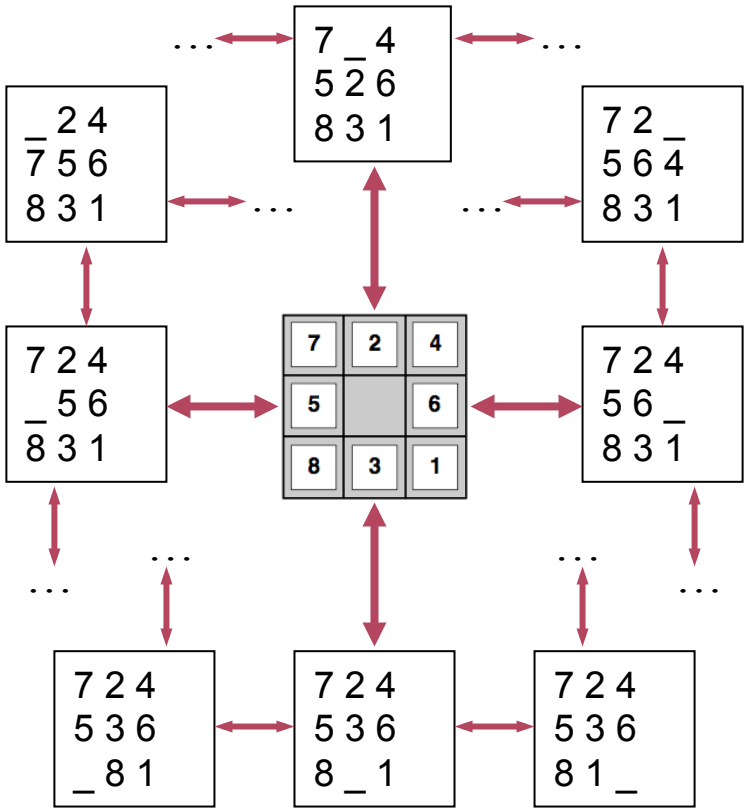
Kostenfunktion: *ordnet jeder Aktion einen Kostenwert zu. Wird benötigt, um kostenoptimale Lösungen zu finden.*

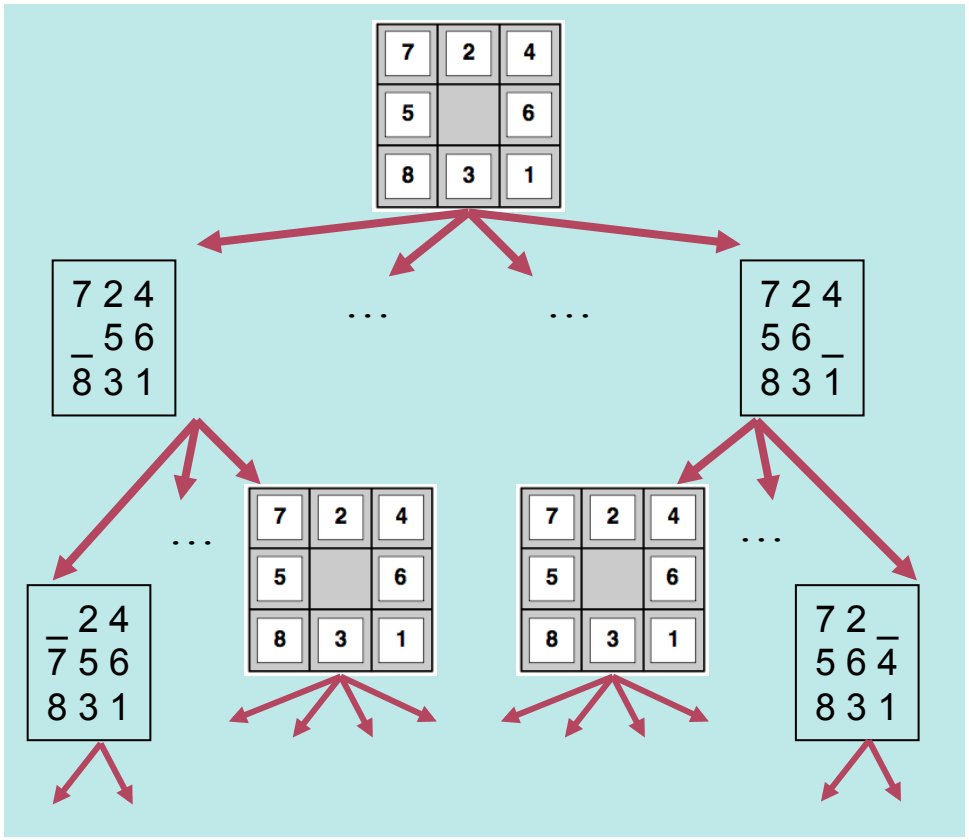
Zustandsraum: *Menge aller Zustände.*

Suchbaum: *Zustände sind Knoten, Aktionen sind Kanten.*

Problemraum und Suchraum

… kann strukturell unterschiedlich gewählt werden



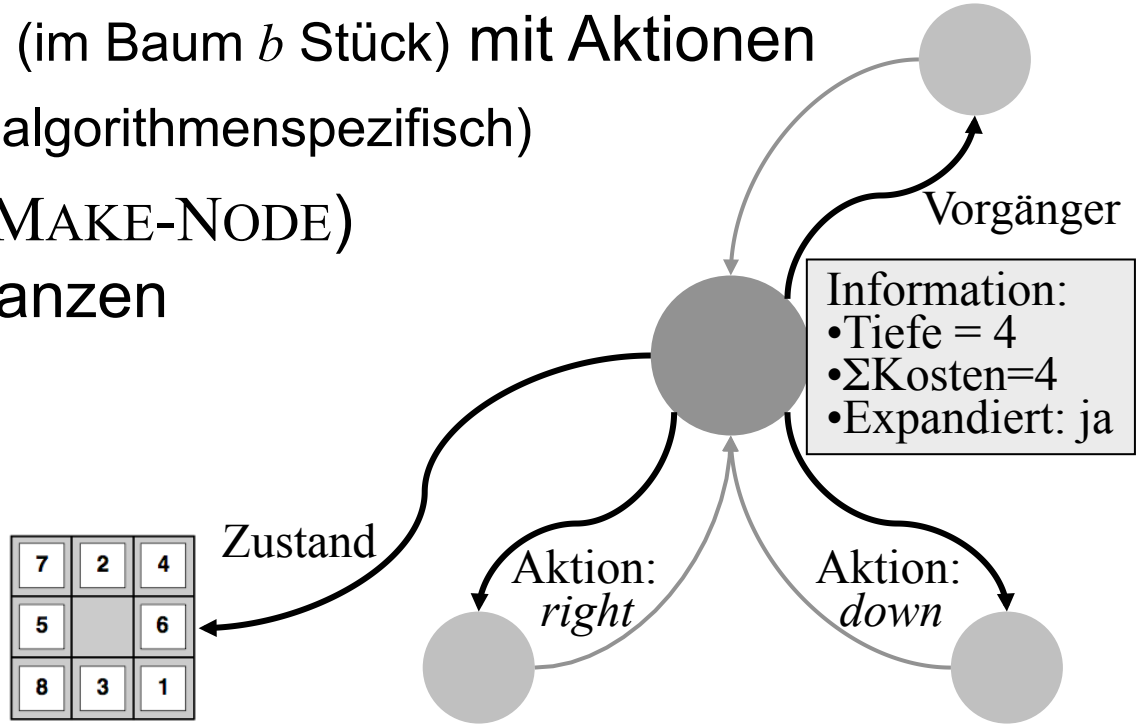


Traversiere **Graph** als **Baum**

Zyklen werden zu unendlichen Pfaden!

Zustände und Knoten

- **Zustände**: „Schnappschüsse“ der Welt
- **Knoten**: Datenobjekte, die Zustände repräsentieren und weitere Information enthalten, z.B.:
  - Vorgängerknoten (im Baum 1)
  - Nachfolgerknoten (im Baum b Stück) mit Aktionen
  - Verwaltungsinfo. (algorithmenspezifisch)
- Konstruktorfunktion (MAKE-NODE) generiert Knoten-Instanzen bei der Suche



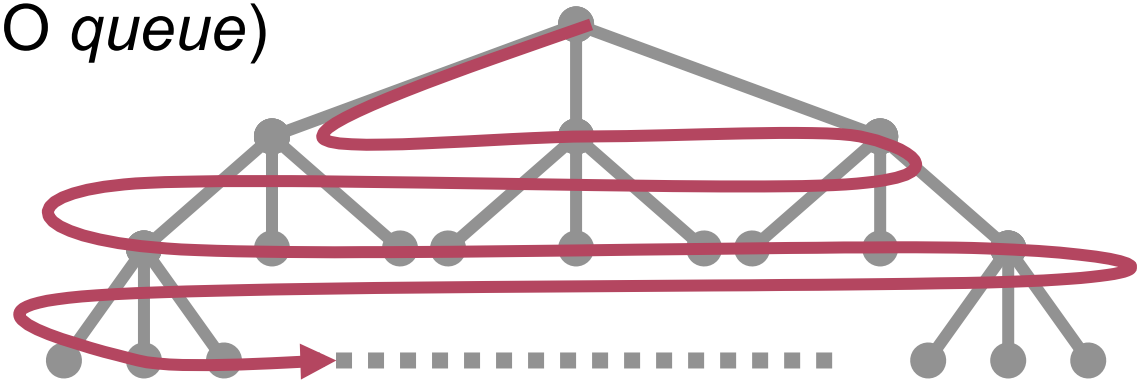
3.1 Uninformierte Suche

- Alle Baum-Suchalgorithmen folgen dem Zyklus
  1. Ist die „Suchfront“ leer, terminiere („keine Lösung“)
  2. Sonst nimm vordersten Knoten n aus aktueller Suchfront
  3. Ist n Zielknoten, terminiere („Lösung gefunden“)
  4. Sonst erzeuge die Nachfolger von n („**expandiere** n“), sortiere sie in die Suchfront ein, mach weiter bei 1.
- Bei Ertel erst als Algorithmenschema für Heuristische Suche eingeführt – passt aber für viele Such-Arten (nicht Spiele mit Gegner, s.3.3!)
- „Uninformierte“ Suche verwendet in Schritt 4 schematische Sortierung (z.B. FIFO, LIFO);
- „heuristische“ Suche (3.2) versucht, das schlauer zu machen

Breitensuche

Siehe Vorlesung Informatik A!

Sortierung der Suchfront implementiert
als Warteschlange (FIFO *queue*)

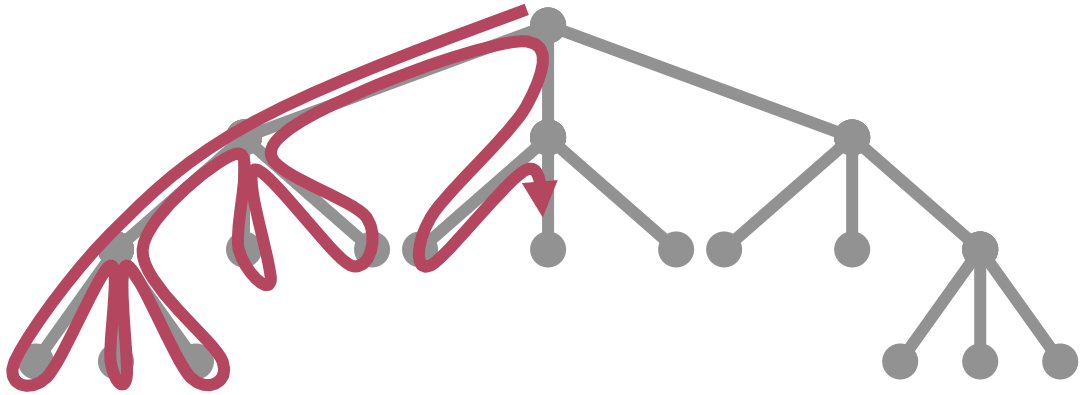


- ☹ Zeitbedarf: $O(b^{d+1})$ (Exponent d falls Zieltest beim Einfügen – konstante Aktionskosten!)
- ☹ Speicherbedarf: $O(b^{d+1})$ (Exponent d falls Zieltest beim Einfügen – konst. Aktionskosten)
- ☺ Vollständig: Wenn Lösung existiert, wird sie gefunden
- ☺ Optimal: Wenn Lösung gefunden, dann ist es eine beste

Tiefensuche

Siehe Vorlesung Informatik A!

Sortierung der Suchfront implementiert
als Keller (LIFO *stack*)

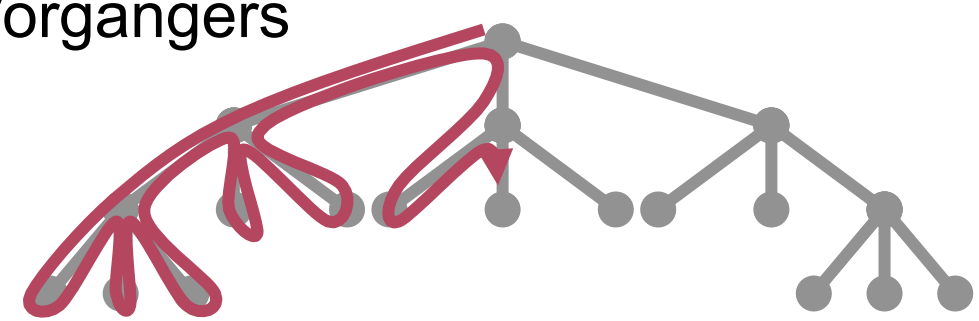


- ☹ Zeitbedarf: $O(b^m)$, wenn m Maximaltiefe des Baums
- ☺ Speicherbedarf: $O(bm)$
- ☹ Unvollständig
- ☹ Nicht optimal

Tiefensuche „taucht ab“ auf unendlichen Suchpfaden!

Varianten von Tiefensuche I: Backtracking

- Erzeuge immer nur 1 Nachfolgeknoten bei Expansion
- Merke je Knoten, welche Nachfolger schon erzeugt sind
- Bei Scheitern an Knoten k erzeuge nächsten Nachfolger des k-Vorgängers



- Erhält das qualitative Verhalten von Tiefensuche

☺ Speicherbedarf: $O(m)$! (m Maximaltiefe des Baums)

- Technik der Wahl bei hohem Verzweigungsfaktor

Varianten II: Tiefenbeschränkung

- Gib ein globales Tiefenlimit l vor

- Erhält das qualitative Verhalten von Tiefensuche
- Speicher: $O(bl)$ (bzw. $O(l)$ bei Backtracking); Zeit $O(b^l)$

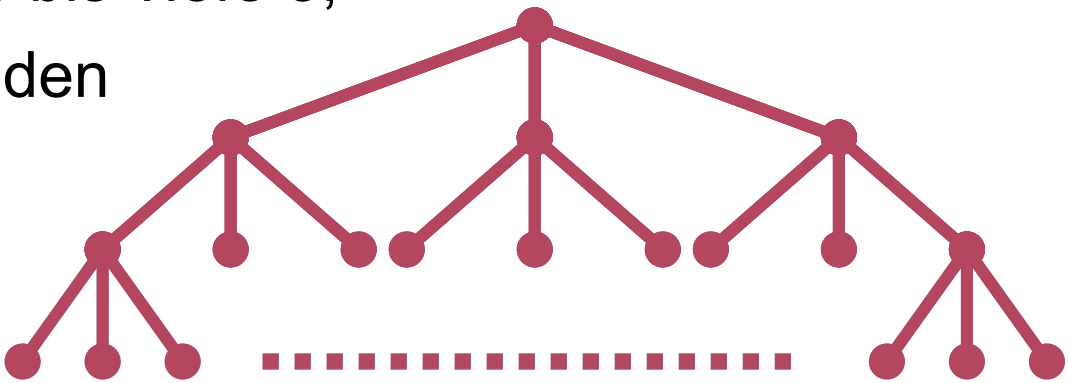
☹ Findet keine Lösung in Tiefen $d>l$

☺ Terminiert sicher bei endlichem Verzweigungsfaktor

Natürlich auch bei Breitensuche anwendbar!

Var. III: Iterierte beschränkte Tiefensuche

- Mach Tiefensuche bis Tiefe 1;
- mach **erneut** Tiefensuche bis Tiefe 2;
- mach **erneut** Tiefensuche bis Tiefe 3;
- … usw., bis Lösung gefunden



☺ Vollständig

☺ Optimal bei konstanten Aktionskosten

☺ Speicherbedarf: $O(bd)$

… aber ist das nicht riesige Zeitverschwendung?

Zeitkomplexität der Iterierten Tiefensuche

… oder: Wie oft muss man schlimmstenfalls einen Knoten anpacken, wenn eine „erste“ Lösung in Tiefe d liegt?

$\sum_{i=0}^{d}\sum_{j=0}^{i} b^j$ -mal!

Wir erinnern uns: $\sum_{j=0}^{i} b^j = \frac{b^{i+1}-1}{b-1}$

$$\sum_{i=0}^{d}\sum_{j=0}^{i} b^j = \sum_{i=0}^{d}\frac{b^{i+1}-1}{b-1}$$

$$= \frac{1}{b-1}\left[b\sum_{i=0}^{d} b^i - \sum_{i=0}^{d} 1\right]$$

$$= \frac{1}{b-1}\left[b\frac{b^{d+1}-1}{b-1} - (d+1)\right] = \frac{b^{d+2}-b}{(b-1)^2} - \frac{d+1}{b-1} \in O(b^d)$$

Eigenschaften der Iterierten Tiefensuche

- ☺ Vollständig
- ☺ Optimal bei konstanten Aktionskosten
- ☺ Speicherbedarf: $O(bd)$
- 😐 Zeitbedarf: $O(b^d)$ wie Breitensuche
  (akzeptabel für vollständiges und optimales Verfahren
  mit realistischem Speicherbedarf)

Iterierte Tiefensuche oder Varianten ist oft die Methode der Wahl!

3.2 Heuristische Suche

… führt zu **informierten Suchverfahren**!

Bisher (3.1): Reihenfolge der Bearbeitung hängt ab von

Wert eines Knotens $g(n)$: Knotentiefe bzw. Pfadkosten

Jetzt: Berücksichtige (zusätzlich) für Knoten Schätzung, wie weit es noch zu einem Ziel ist
(„Mama, wie weit müssen wir ‘n noch!?“)

Wert $h(n)$: Schätzung der Kosten von n bis zu einem Ziel
(Voraussetzung immer: $h(n)=0$ für Zielknoten n.)

Aktionskosten

Voraussetzung bisher:

Alle Aktionen verursachen gleiche Kosten bei Ausführung

- ➥ **Tiefe** eines Knotens im Suchbaum
  entspricht „Herstellungskosten“ des entspr. Zustands

Nun:

Aktionen verursachen möglicherweise unterschiedliche Kosten

- ➥ **Pfadkosten** eines Knotens im Suchbaum/Suchgraphen
  entspricht „Herstellungskosten“ des entspr. Zustands

Beispiele

Route planen (TSP!), Computer konfigurieren, Stundenplan erstellen

Beispielproblem II: Das Reiseproblem

Zustand: Merkmal $in(x)$ für Ort x

Startzustand: $in(Arad)$

Zielzustand: $in(Bucuresti)$

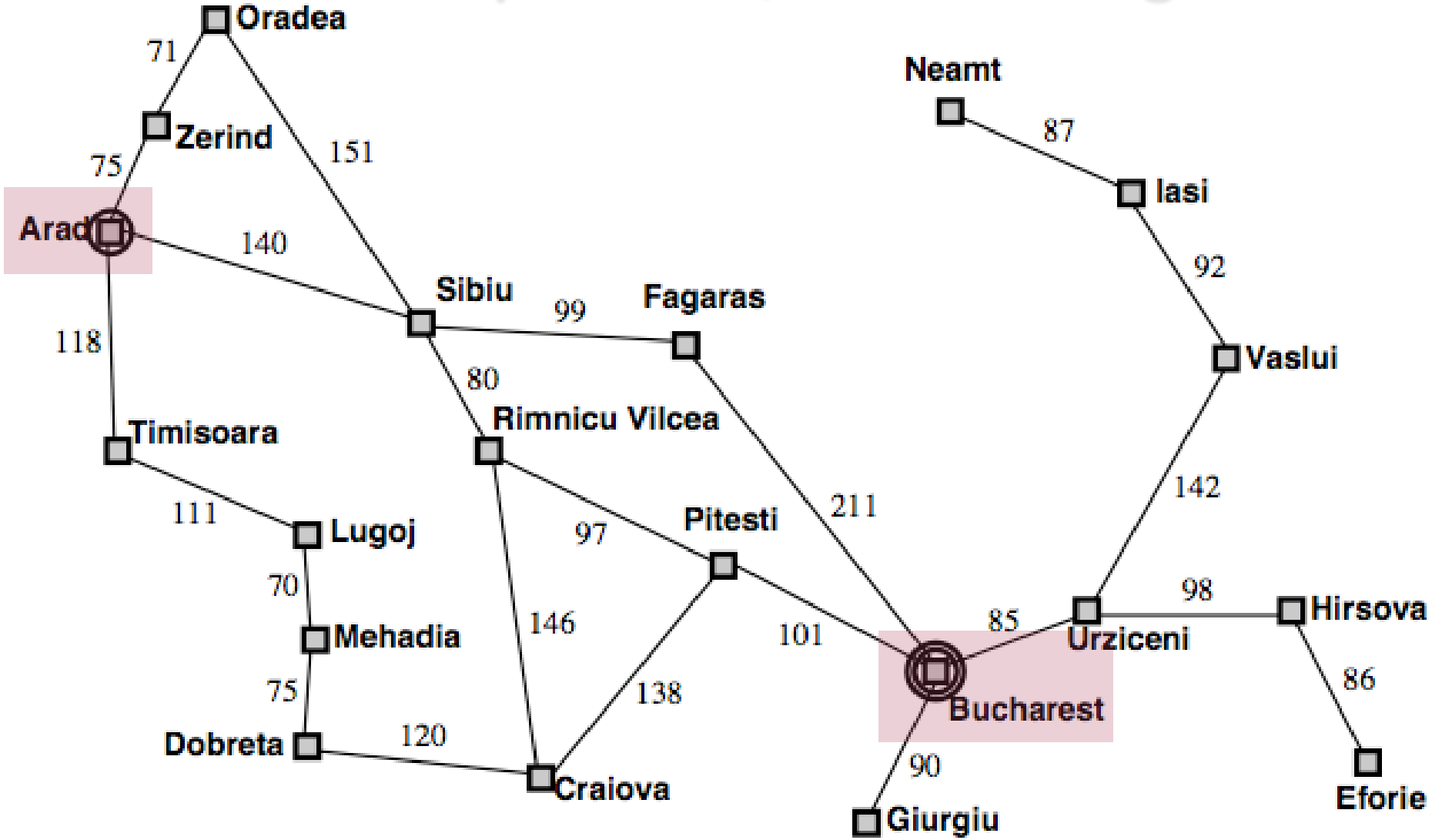
Aktionen: $go(x,y)$: fahre von Ort x nach Ort y
Wirkung: vorher galt $in(x)$; danach gilt $in(y)$

Kosten: Straßenkilometer zwischen x und y für Aktion $go(x,y)$

Nebenbedingung

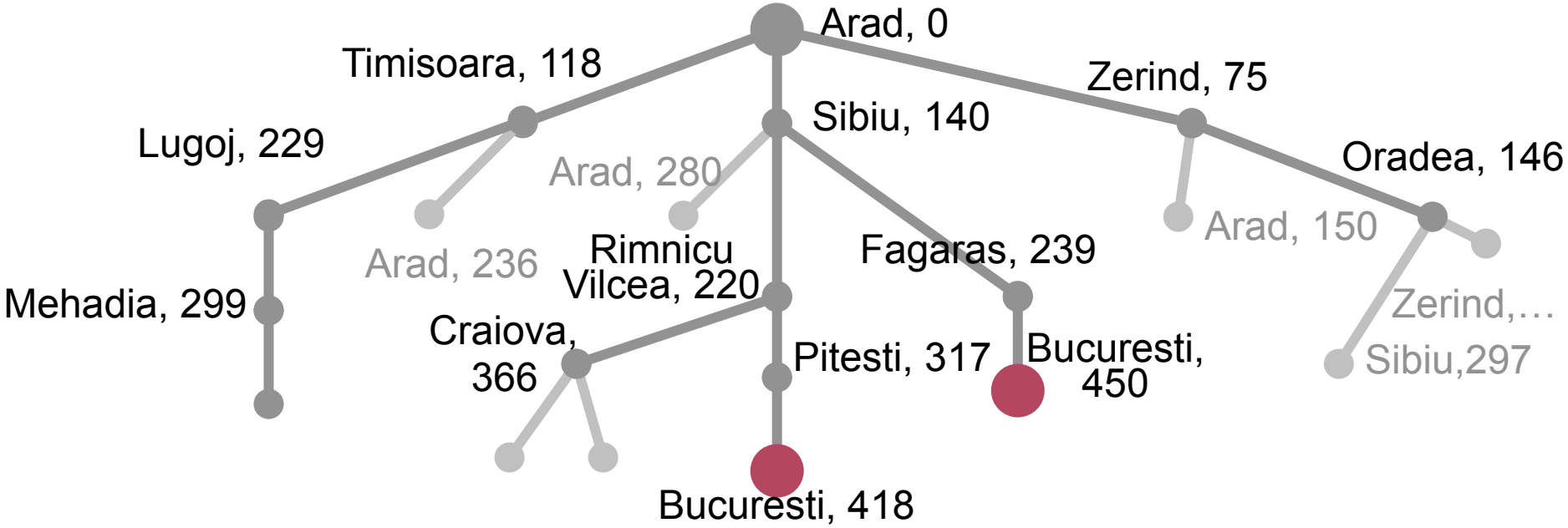
Finde Weg vom Start zum Ziel mit minimalen Pfadkosten!

Reiseproblem, Fortsetzung



Standard-Suche nach Kosten (*uniform-cost*)

- Geh vor wie bei Breitensuche,
- aber bewerte Knoten durch ihre Pfadkosten ab Wurzel $g(n)$
- sortiere in Suchfront nach Knotenwerten (billigste vor)
- ende erst, wenn ein Zielknoten expandiert werden müsste



Eigenschaften der *uniform-cost* Suche

- Aktionskosten mindestens $\varepsilon > 0$!
- Breitensuche ist Spezialfall von *uniform-cost*
  (bei Einheitskosten)

  - ☺ Vollständig
  - ☺ Optimal
  - ☹ Speicherbedarf: $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$
    (für Kosten C^* des optimalen Pfads)
  - ☹ Zeitbedarf: $O(b^{(1+\lfloor C^*/\varepsilon \rfloor)})$

Billige Schritte in die falsche Richtung erhöhen die Suchkosten:
$1+\lfloor C^*/\varepsilon \rfloor >> d$