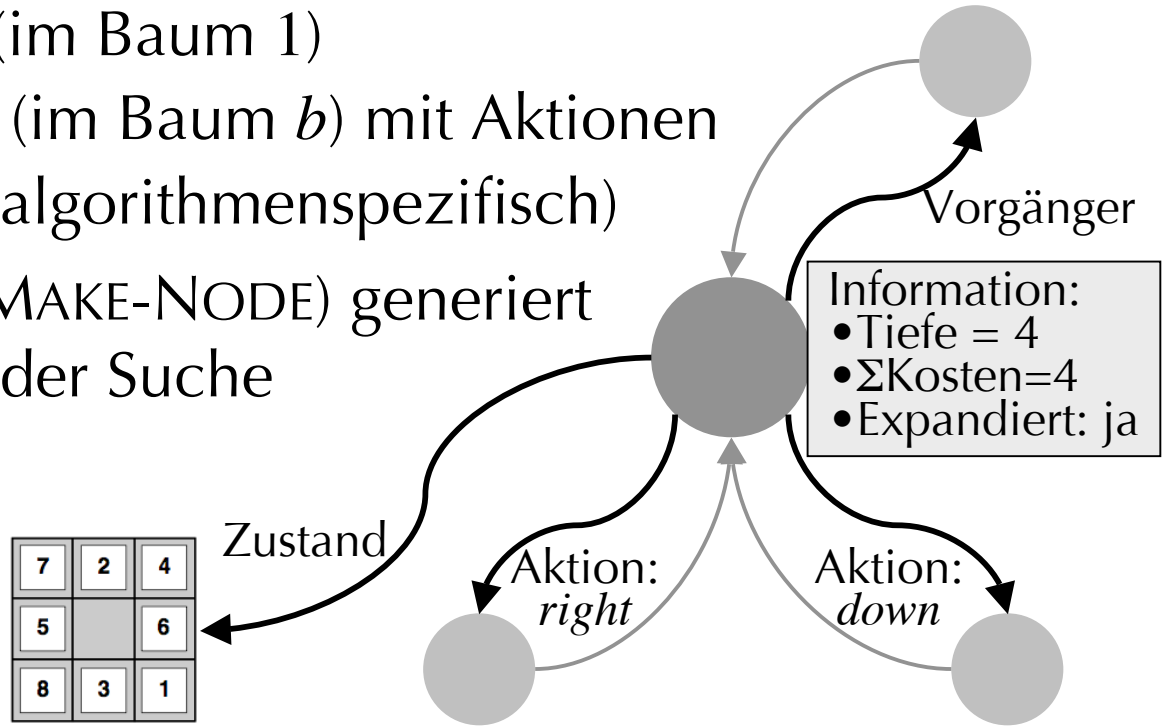
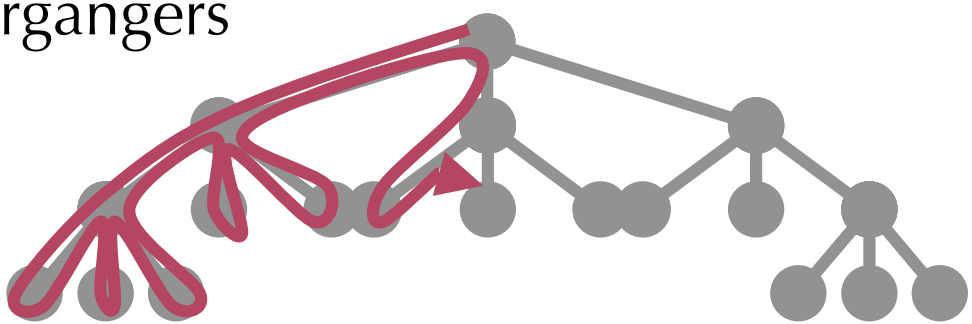
Zustände und Knoten

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



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: $O(m)$!

- 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 $O(b^l)$

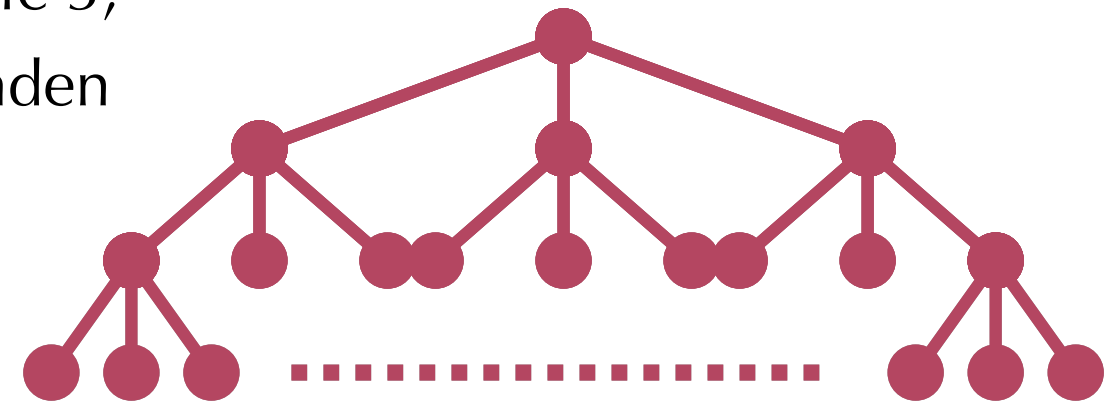
☹ Findet keine Lösung in Tiefen $d>l$

☺ Terminert sicher bei endlichem Verzweigungsfaktor

Natürlich auch bei Breitensuche anwendbar!

Var. III: Iterierte beschränkte Tiefensuche

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



☺ Vollständig

☺ Optimal bei konstanten Aktionskosten

☺ Speicherbedarf: *O(bd)*

… aber ist das nicht 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: $\sum_{i=0}^{d} b^i = \frac{b^{d+1}-1}{b-1}$

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

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

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

Eigenschaften der Iterierten Tiefensuche

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

Iterierte Tiefensuche oder Varianten ist oft die Methode der Wahl!

Gleiche Zustände nur einmal bearbeiten

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Registriere bereits besuchte Zustände

Wann GRAPH-SEARCH?

- Als Variante aller Suchmethoden verwendbar
- bei Berücksichtigung von Kosten immer *billigsten* Knoten im Graphen belassen! (hier verwendet: erstes Auftreten)
- erhöhte Kosten durch Verwaltung der *closed-Menge*
  - effizient z.B. mit Hashing
- sinnvoll, wenn viele inverse Aktionen im Zustandsraum oder viele kommutative Aktionen
- Einsparung exponenziell in der Zahl von Doubletten
- Speicher-Komplexität hängt ab von Größe des *Zustands*raums!

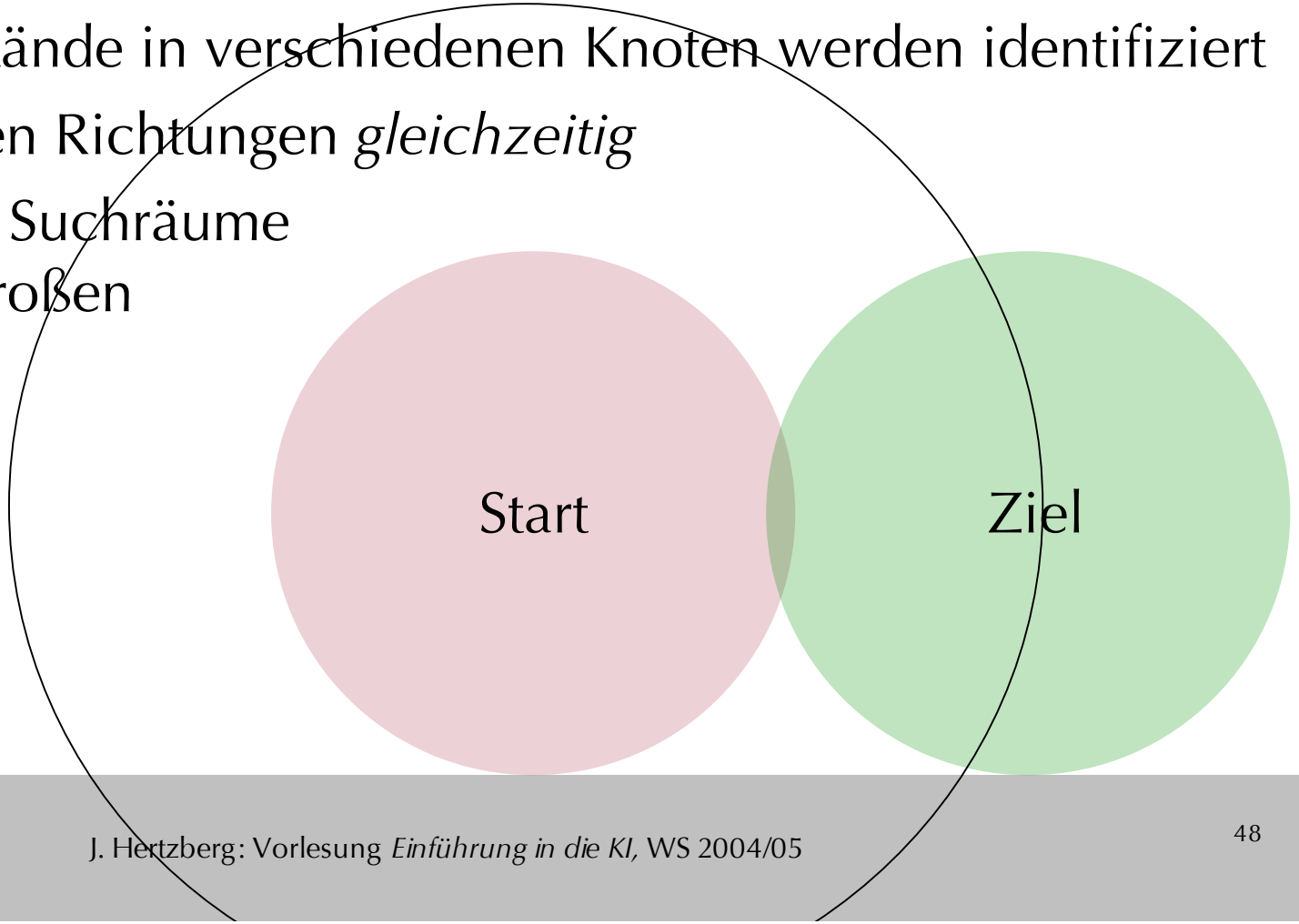
Bidirektionale Suche

Vorausgesetzt,

- Aktionen „rückwärts“ angewandt sind eindeutig
- der Zielzustand ist eindeutig (bzw. endlich)
- gleiche Zustände in verschiedenen Knoten werden identifiziert

suche aus beiden Richtungen *gleichzeitig*

⇒ Zwei kleine Suchräume
statt eines großen



Eigenschaften der bidirektionalen Suche

Bei Breitensuche in beiden Richtungen:

- ☺ Vollständig
- ☺ Optimal bei konstanten Aktionskosten
- 😐 Speicherbedarf: $O(b^{d/2})$
- 😐 Zeitbedarf: $O(b^{d/2})$

Für die Praxis ist der Speicherbedarf zu hoch!

Aktionskosten

Voraussetzung bisher:

Alle Aktionen verursachen gleiche Kosten bei Ausführung

⇒ **Tiefe** eines Knotens im Suchbaum
entspricht „Herstellungskosten“ des entsprechenden Zustands

Nun:

Aktionen verursachen unterschiedliche Kosten

⇒ **Pfadkosten** eines Knotens im Suchbaum
entspricht „Herstellungskosten“ des entsprechenden Zustands

Beispiele

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

Beispielproblem II: Das Reiseproblem

Zustand: Merkmal $in(x)$ für Ort x

Startzustand: $in(Arad)$

Zielzustand: $in(Bucuresti)$

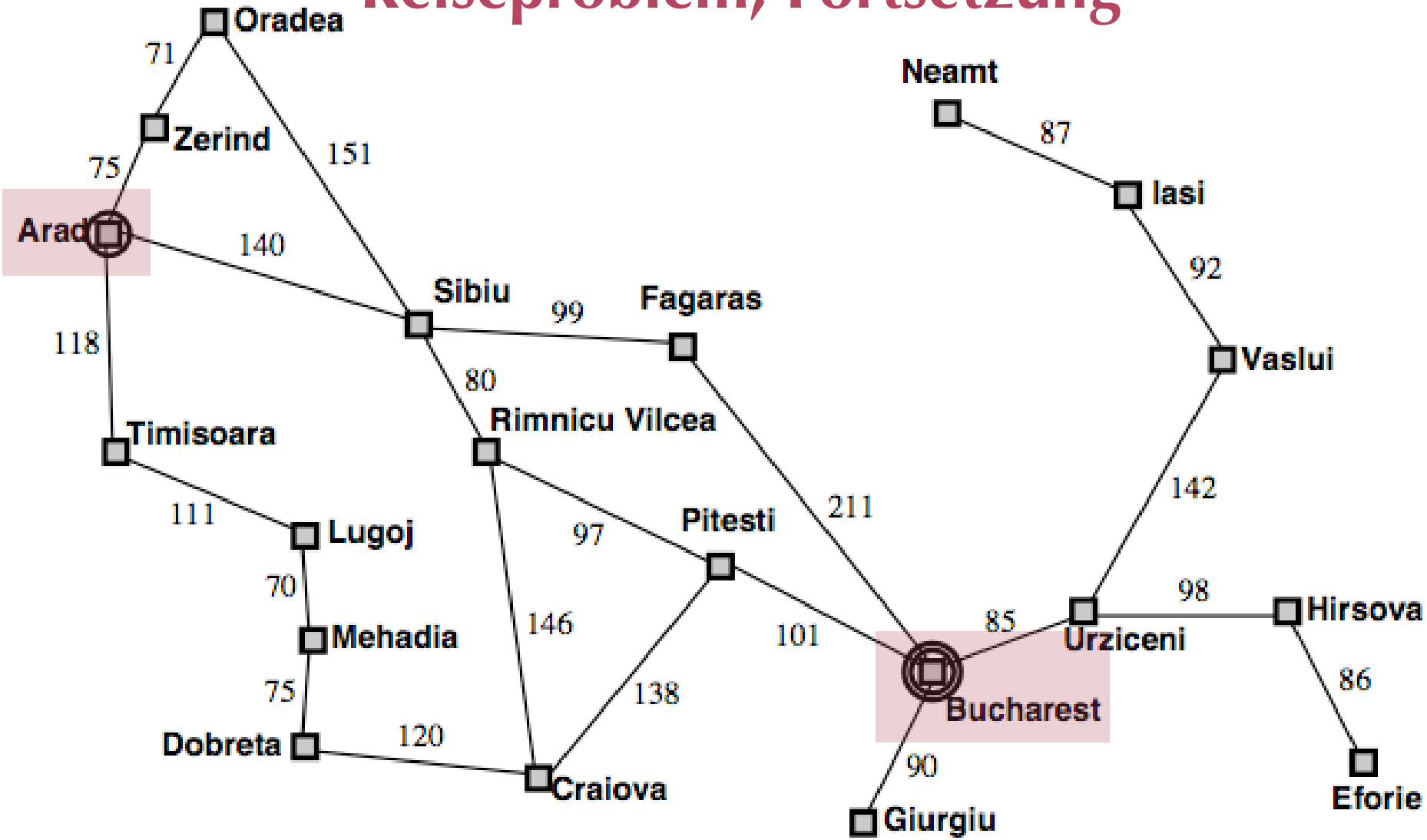
Aktionen: $go(x,y)$: fahre von Ort x nach Ort y
Wirkung: vorher galt $in(x)$; nachher gilt $in(y)$

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

Nebenbedingung

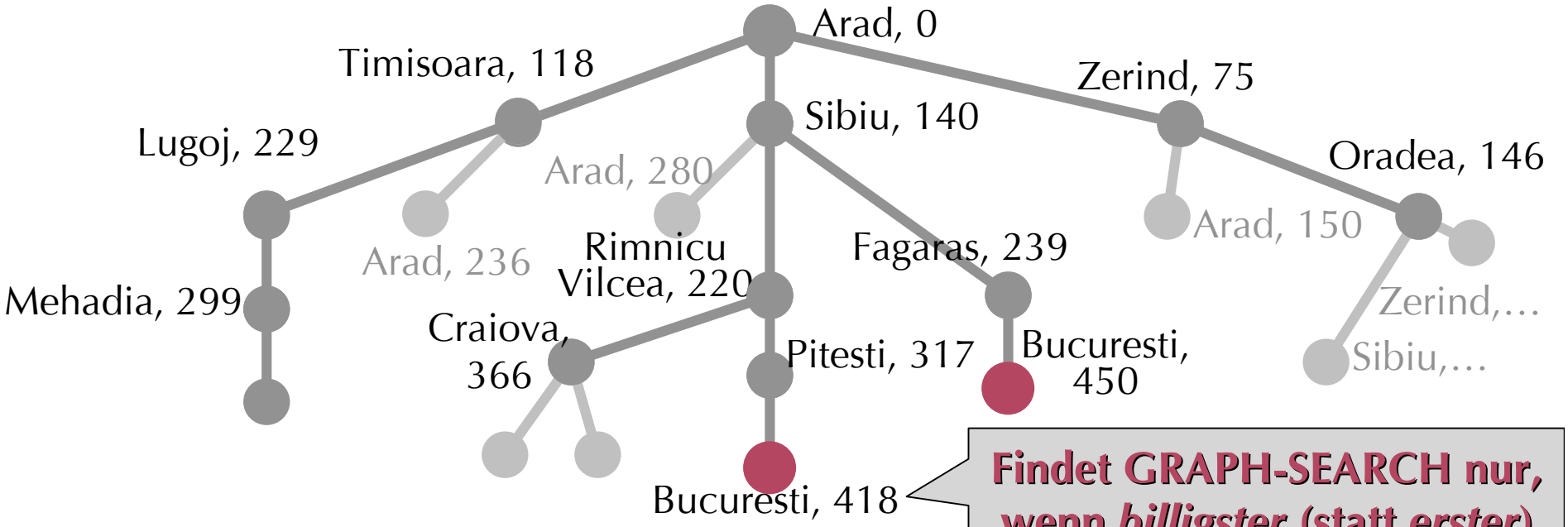
Finde Weg vom Start zum Ziel mit minimalen Pfadkosten!

Reiseproblem, Fortsetzung



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

- Geh vor wie bei Breitensuche,
- aber bewerte Knoten durch ihre Pfadkosten ab Wurzel ***g(n)***
- sortiere bei INSERTALL 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 (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:
$1+\lfloor C^*/\varepsilon \rfloor >> d$

Suche ist …

… eine der „schwachen Methoden“ (*weak methods*) der KI

… damit eine allgemeine Problemlösungsmethode/-metapher

- Deduktion wäre eine andere

… für konkrete Probleme durch spezifische Algorithmen zu ersetzen (wenn man welche kennt)