Künstliche Intelligenz

Problemlösung durch Suchen

Dr.-Ing. Stefan Lüdtke

Universität Leipzig

Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI)

Problemlösung durch Suchen

Problemlösende Agenten

- Ein *problemlösender Agent* ist ein Spezialfall des zielbasierten Agenten
- Sucht nach Aktionsfolgen, die einen wünschenswerten Zustand erreichen.
- Varianten der Suche:
  - *nichtinfomierte* Suche: Keine Information über den „Abstand“ zwischen wünschenswertem Zustand und anderen Situationen.
  - *informierte* Suche: Abstandsinformation vorhanden.
- Was bedeutet hierbei „Problem“ und „Lösung“?

Problemlösender Agent

Algorithmus am Beispiel

Wir sind in Arad, Rumänien. Was wollen wir tun?

- 1. Schritt: *Zielformulierung*:
  - Möglichst braun werden? Möglichst viel Spaß haben (ohne Kater ...)? Am nächsten Morgen in Bukarest sein (weil wir einen Flug gebucht haben)?
  - Welches Ziel der Agent wählt, hängt im Allgemeinen von der Nutzenfunktion ab.
  - Im Allgemeinen ist das Ziel definiert durch eine Menge von Weltzuständen. (Z. B. alle möglichen Welten, in denen wir in Bukarest sind.)
- 2. Schritt: *Problemformulierung*:
  - Auswahl der relevanten Aktionen in Abhängigkeit von Weltzustand und Ziel
  - Auswahl der relevanten Zustandsaspekte (z. B. „Ort“; aber nicht „Bräunungsgrad“)

Problemlösender Agent

Algorithmus am Beispiel

- 3. Schritt: *Suche einer Lösung*, gegeben Aktionen, Zustandsmenge und Ziel.
  - Diese Lösung ist eine Folge von Aktionen, die den aktuellen Zustand in einen Zustand überführt, der im Ziel liegt.
  - Von Arad führen Straßen nach Sibiu, nach Timisoara und nach Zerind. Welche nehmen wir?
    - Wenn wir keine Karte haben und uns in Rumänien nicht auskennen, ist das beste, was wir tun können, einen zufälligen Weg wählen.
    - Mit einer Karte können wir dagegen überlegen, in welcher Situation wir wären, wenn wir nach Sibiu fahren würden: d.h., welche Aktionen wir von Sibiu aus durchführen könnten.
    - Möglicherweise würden wir über einen solchen Mechanismus schließlich einen Weg nach Bukarest finden.
    - Die Fähigkeit zu einer solchen *hypothetischen* Aktionsausführung – also zur *Simulation* der Wirkung einer Aktionsfolge – ist die fundamentale Fähigkeit eines problemlösenden Agenten.
- 4. Schritt: schrittweise *Ausführung* der Aktionsfolge.

Problemlösender Agent

Funktionaler Aufbau

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*)
returns: an action
inputs: *percept*, a percept
static: *seq*, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation

state ← UPDATE-STATE(*state*, *percept*)
if *seq* is empty **then**
goal ← FORMULATE-GOAL(*state*) ▷ Step 1
problem ← FORMULATE-PROBLEM(*state*, *goal*) ▷ Step 2
seq ← SEARCH(*problem*) ▷ Step 3
action ← FIRST(*seq*) ▷ Step 4
seq ← REST(*seq*)
return *action*

Problemlösender Agent

Implizite Annahmen

- Die Umgebung ist *statisch*.
  Keine Aktualisierung von seq, während die Aktionen abgearbeitet werden.
- Die Umgebung ist *observabel*.
  Genauer: zumindest der initiale Zustand ist soweit bekannt, wie es für die Bestimmung der Aktionsfolge erforderlich ist.
- Die Schritte werden *diskret* ausgeführt.
- Aktionen sind *deterministisch*, soweit es die Ausführbarkeit und die Ergebnisse von Folgeaktionen betrifft.
- Wir haben in diesem Fall ein *Single-State*-Problem: Der Agent weiß stets, in welchem Zustand er sich befindet. Die Lösung ist eine Aktionsfolge.

Folgend betrachten wir FORMULATE-PROBLEM und insbesondere die Realisierung von SEARCH.

Formulate-Problem

Wohldefinierte Probleme

Ein wohldefiniertes Problem besteht aus den folgenden Komponenten:

- Ein *Initialzustand* x_0. Zum Beispiel: $In(Arad)$.
- Eine Beschreibung der möglichen *Aktionen* des Agenten. Dabei ist eine Aktion eine Funktion, die einen Zustand auf einen Nachfolgerzustand abbildet (später: Auf eine Wahrscheinlichkeitsverteilung von Nachfolgerzuständen)
- Initialzustand und Nachfolgerfunktion definieren implizit den *Zustandsraum* X des Problems – die Menge aller Zustände, die von x_0 aus erreichbar sind.
- Ein *Pfad* in X ist eine Folge von Zuständen $\langle x_0, \ldots, x_n \rangle$ die durch eine Folge von Aktionen $\langle a_1, \ldots, a_n \rangle$ verbunden wird, so dass $x_i = a_i(x_{i-1})$.

Formulate-Problem

Wohldefinierte Probleme (Fortsetzung)

- Ein *Zieltest*, der festlegt, ob ein bestimmter Zustand $x \in X$ ein Ziel ist. Der Zieltest definiert eine Menge $G \subseteq X$, die Zielzustände. Beispielsweise $\{In(Bukarest)\}$.
- Eine *Kostenfunktion*, die für jeden Pfad eine Zahl liefert.
  Die Wahl der Kostenfunktion repräsentiert die Nutzenfunktion des Agenten.
- Für einen Pfad $(\langle x_0, \ldots, x_n \rangle, \langle a_1, \ldots, a_n \rangle)$ sind die *Schrittkosten* von x_{i-1} nach x_i gegeben durch eine Funktion $c(x_{i-1}, a_i)$.
  Oft sind die Pfadkosten gegeben durch die Summe der Schrittkosten.
  Wir nehmen an, dass Schrittkosten nicht negativ sind.
- Die *Lösung* eines so definierten Problems ist gegeben durch einen Pfad der vom Initialzustand zu einem Zustand führt, der den Zieltest erfüllt.
- Die *optimale Lösung* ist die Lösung mit den geringsten Pfadkosten.

Suchprobleme: Formales Modell

Ein Suchproblem ist ein Tupel (S, A, G, σ, c), wobei

- $S = \{s_1, s_2, \dots\}$ ist der Zustandsraum (nicht notwendigerweise endlich)
- $A = \{a_1, a_2, \dots\}$ mit $a : S \to S$ ist die Aktionsmenge
- $G \subseteq S$ sind die Zielzustände
- $\sigma \in S$ ist der Initialzustand
- $c : S \times A \to \mathbb{R}$ ist die Kostenfunktion[1]

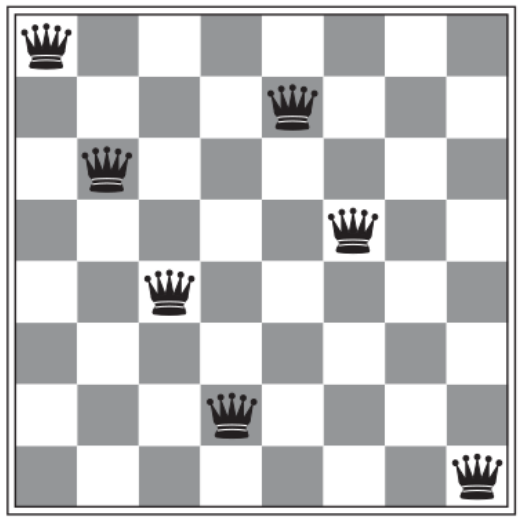
Eine Lösung eines Suchproblems ist eine Sequenz von Aktionen $\langle a_{(1)}, \dots, a_{(n)} \rangle$, sodass $(a_{(1)} \circ \cdots \circ a_{(n)})(\sigma) \in G$

[1] Später schauen wir uns nichtdeterministische Aktionen an, dann ist es sinnvoll, die Kostenfunktion als $c : S \times A \times S \to \mathbb{R}$ zu definieren.

Wahl des Zustandsraums

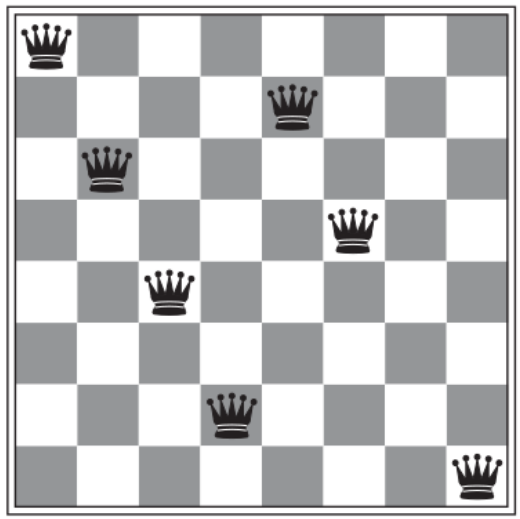
- Die Wahl eines guten Zustandsraums ist *entscheidend* für eine effiziente Problemlösung
- Der Zustandsraum sollte von allen unwichtigen Eigenschaften der Welt *abstrahieren*.
- D.h., ein Zustand $x \in X$ repräsentiert viele möglichen Zustände der realen Welt.
  Beispiel: $In(Arad)$
  - Es könnte in Arad regnen oder die Sonne scheinen.
  - Das Auto könnte rot oder grün sein.
  - Der Fahrer trägt einen Hut oder nicht ...
- Auch die Aktionen müssen von unwichtigen Eigenschaften abstrahieren (wie schnell wird eine Strecke gefahren? An welchen Parkplätzen angehalten? ...)
- Es sollte weniger abstrakte Aktionen als reale Aktionen geben.
- Wirksames Beispiel: 8-Damen-Problem

8-Damen-Problem



- Zustände? Jede Anordnung von 0 bis 8 Damen auf dem Brett
- Initialzustand? Leeres Brett
- Aktionen? Setze Dame auf leeres Feld
- Nachfolgerfunktion? Gebe Brett mit zusätzlicher Dame zurück
- Zieltest? 8 Damen, keine bedroht
- $64!/56! = 1.8 * 10^{14}$ mögliche Sequenzen

8-Damen-Problem



- Zustände? Anordnung von $0 \leq n \leq 8$ Damen, eine pro Spalte in den linken n Spalten, sodass sich keine bedroht
- Initialzustand? Leeres Brett
- Aktionen? Setze Dame auf nächste leere Spalte, sodass sie nicht bedroht ist
- Nachfolgerfunktion? Gebe Brett mit zusätzlicher Dame zurück
- Zieltest? 8 Damen, keine bedroht
- 2057 mögliche Sequenzen

Suchstrategien: Exploration

- Zustände und Aktionen bilden einen gerichteten Graphen (Knoten = Zustände, Kanten = Aktionen)
- Suchstrategien versuchen, in diesem Graphen Pfade von einem Zustand (Initialzustand) zu einem anderen Zustand zu finden, der das Ziel erfüllt.
- Im Allgemeinen sind die Knotenmenge S (i) beim Start nicht bekannt und/oder (b) so groß, dass sie nicht sinnvoll gehandhabt werden kann
- Deshalb: Explorative Verfahren
  - Anfänglich nur A und σ bekannt
  - *Erzeuge* systematisch durch Anwendung der Aktionen neue Zustände
  - Bis Ziel erreicht ist ("forward chaining")
  - Auch denkbar: "backward chaining": Exploriere von einem Zielknoten, bis Startknoten gefunden

Suchstrategien: Informiertheit

Sei eine Menge von bisher entdeckten Knoten, D , gegeben – d.h., eine Menge von Zuständen, die von aus erreichbar sind. Aber D enthalte noch keinen Zielzustand. Wie geht man nun weiter vor?

- Wähle einen Zustand $s \in D$ und wähle Aktion $a \in A$
- Erzeuge Folgezustand $s' = a(s)$
  Das erzeugen aller möglichen Folgezustände von s heißt auch "Expandieren von s"
- Prüfe, ob $s' \in G$. Falls ja: Ende
- Sonst: $D \leftarrow D \cup \{s'\}$, und alles von vorn

Wie wählt man s und a so, dass man sich dem Ziel möglichst schnell nähert?

- **Uninformierte** Suchalgorithmen: Man hat keine Ahnung über den Abstand von s zum Ziel.
- **Informierte** Suchalgorithmen: Man hat eine (ungefähre) Vorstellung über den Abstand

Suchstrategien: Exploration

Wie sehen eigentlich die Graphen aus, die exploriert werden müssen?
Mögliche Formen:

- Endliche Bäume: es gibt einen eindeutigen Pfad zu jedem Knoten. Der erste Weg ist der einzige und beste. In endlicher Zeit kann der Graph vollständig durchsucht werden.
- Unendliche Bäume: Der Graph kann nicht mehr in endlicher Zeit durchsucht werden. Eine falsche Wahl kann die Tiefensuche in eine unendliche Sackgasse führen.
- Gerichtete azyklische endliche Graphen: Es gibt mehrere Wege zu einem Knoten, aber der Pfadbaum ist endlich.
  (Tiefensuche im Pfadbaum terminiert, nicht notwendigerweise mit dem besten Pfad)
- Gerichtete zyklische endliche Graphen: Es gibt mehrere Wege zu einem Knoten und der Pfadbaum ist unendlich.
  (Tiefensuche im Pfadbaum terminiert nicht mehr sicher)

Uninformierte Suchalgorithmen

Sie haben im Studium bisher drei Algorithmen kennengelernt, die explorativ arbeiten:

- Tiefensuche
- Breitensuche
- Dijkstra's Algorithmus

Keines dieser Verfahren ist ein informiertes Verfahren – keines verwendet Wissen über den Abstand eines gegebenen Zustands zum Zielzustand.

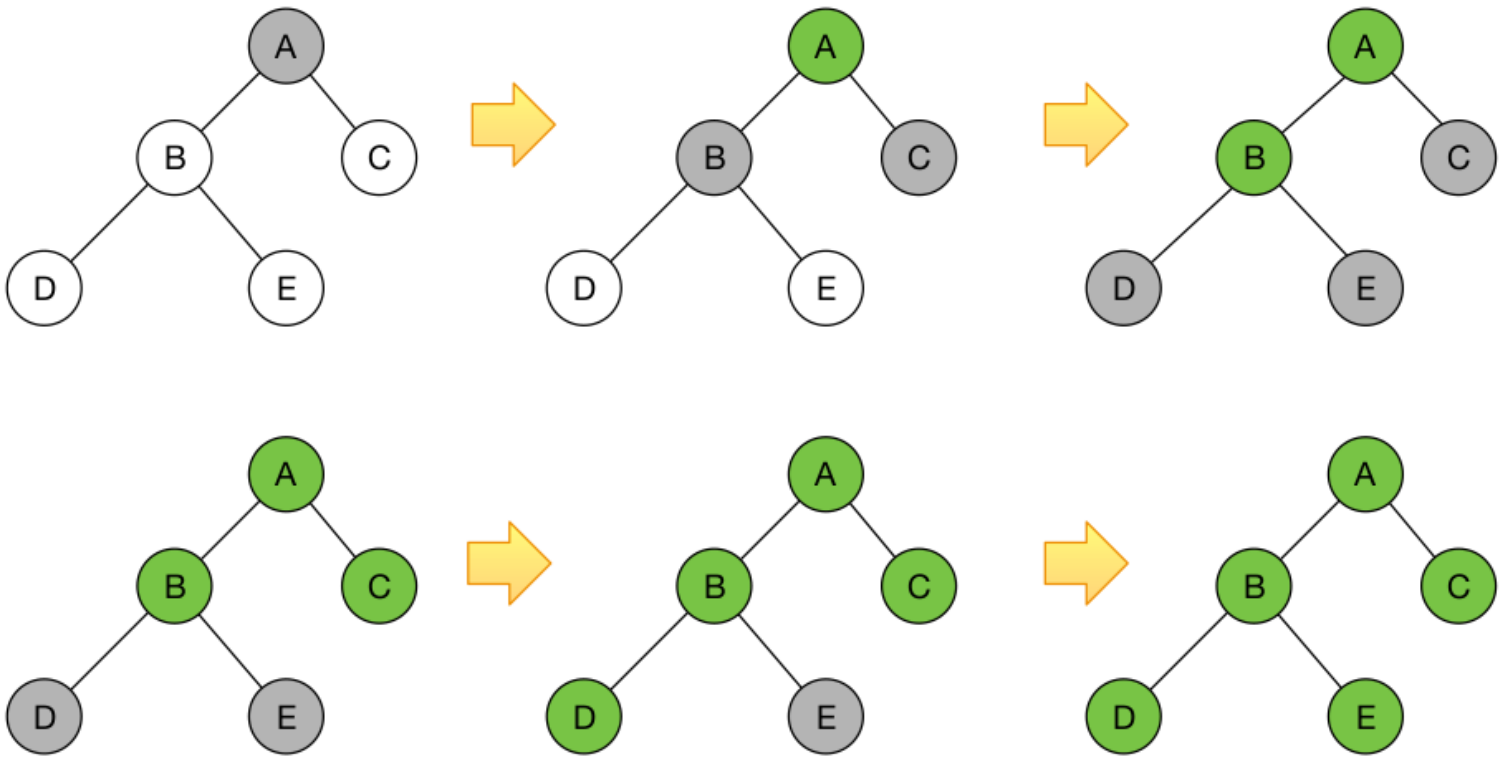
Der Unterschied zwischen den drei Verfahren liegt in der Verwaltung der neu entdeckten Knoten:

- Tiefensuche: Neu entdeckte Knoten werden in einem Stack (LIFO) gesammelt
- Breitensuche: Neu entdeckte Knoten werden in einer Queue (FIFO) gesammelt
- Dijkstra: Neu entdeckte Knoten werden in einem Priority-Queue gesammelt (Priorität = Kosten, mit dem der Knoten vom Start σ aus erreichbar ist)

Breitensuche

function BFS(σ,A,G)
  $Q \leftarrow$Mk-Queue
  Enqueue($Q, (\sigma, \langle\rangle, 0)$) ▷ Tripel: Zustand, Aktionsfolge, Kosten
  $D \leftarrow \{\}$ ▷ Bereits entdeckte Knoten
  while $\neg$ Empty(Q) **do**
    $(s, al, c) \leftarrow$ Dequeue(Q) ▷ Neuer expandierter Knoten
    if $s \in G$ **then**
      return (s, al, c)
    else if $s \notin D$ **then**
      $D \leftarrow D \cup \{s\}$ ▷ Als entdeckt markieren
      for each $(a, c') \in A$ **do**
        Enqueue($Q, (a(s), al \oplus \langle a\rangle, c + c')$)
  return Fail

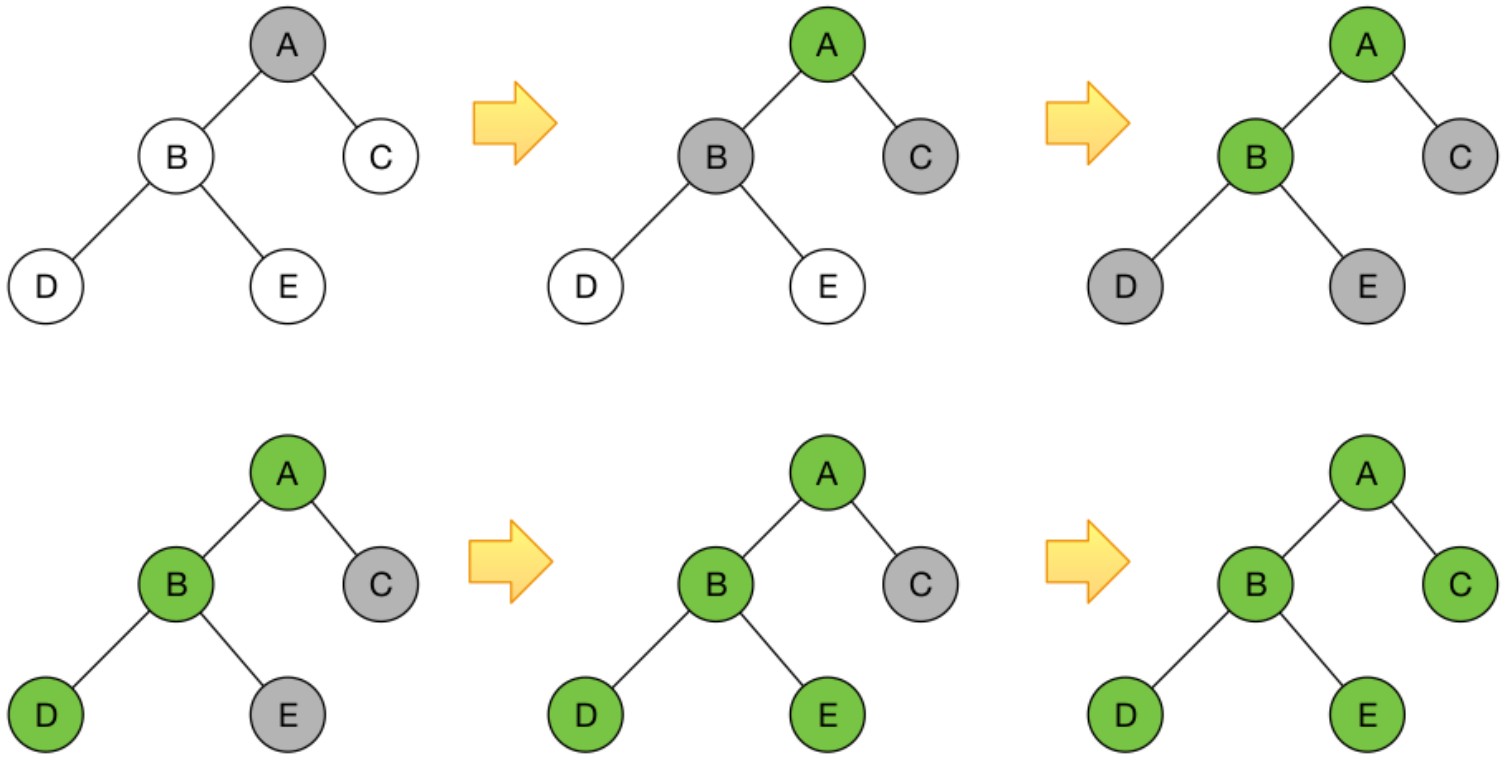
Breitensuche



Tiefensuche

function DFS(σ,A,G)
  $S \leftarrow$MK-STACK
  PUSH($S, (\sigma, \langle\rangle, 0)$) ▷ Tripel: Zustand, Aktionsfolge, Kosten
  $D \leftarrow \{\}$ ▷ Bereits entdeckte Knoten
  while $\neg$ EMPTY(S) **do**
    $(s, al, c) \leftarrow$ POP(S) ▷ Neuer expandierter Knoten
    if $s \in G$ **then**
      return (s, al, c)
    else if $s \notin D$ **then**
      $D \leftarrow D \cup \{s\}$ ▷ Als entdeckt markieren
      for each $(a, c') \in A$ **do**
        PUSH($S, (a(s), al \oplus \langle a \rangle, c + c')$)
  return FAIL

Wiederholung: Tiefensuche



Dijkstra

function DIJKSTRA(σ,A,G)
  $Q \leftarrow$MK-PRIORITY-QUEUE
  ENQUEUE($Q, (\sigma, \langle\rangle, 0)$) ▷ Tripel: Zustand, Aktionsfolge, Kosten
  $D \leftarrow \{\}$ ▷ Bereits entdeckte Knoten
  while $\neg$ EMPTY(S) **do**
    $(s, al, c) \leftarrow$ DEQUEUE-MIN(Q) ▷ Neuer expandierter Knoten
    if $s \in G$ **then**
      return (s, al, c)
    else if $s \notin D$ **then**
      $D \leftarrow D \cup \{s\}$ ▷ Als entdeckt markieren
      for each $(a, c') \in A$ **do**
        ENQUEUE($Q, (a(s), al \oplus \langle a \rangle, c + c')$)
  return FAIL

Probleme

- Unendlicher Graph: Tiefensuche führt nie zum Ziel ("unendliche falsche Abzweigung")
- Breitensuche liefert zwar immer ein (bei konstanten Aktionskosten optimales) Ergebnis, aber zu sehr hohen Speicherkosten
  - Wenn jeder Zustand b Nachfolger hat, und der Pfad zum Ziel n Schritte lang ist, benötigt Breitensuche Speicherplatz der Größe $\mathcal{O}(b^{n+1})$
  - Dijkstra benötigt $\mathcal{O}(b^{C^*/\epsilon})$ Speicher (C^*: Kosten der optimalen Lösung, ϵ: Minimale Kosten einer Aktion)
  - Testen, ob ein Zustand bereits erreicht wurde, kann bei vielen zu speichernden Zuständen sehr aufwendig werden

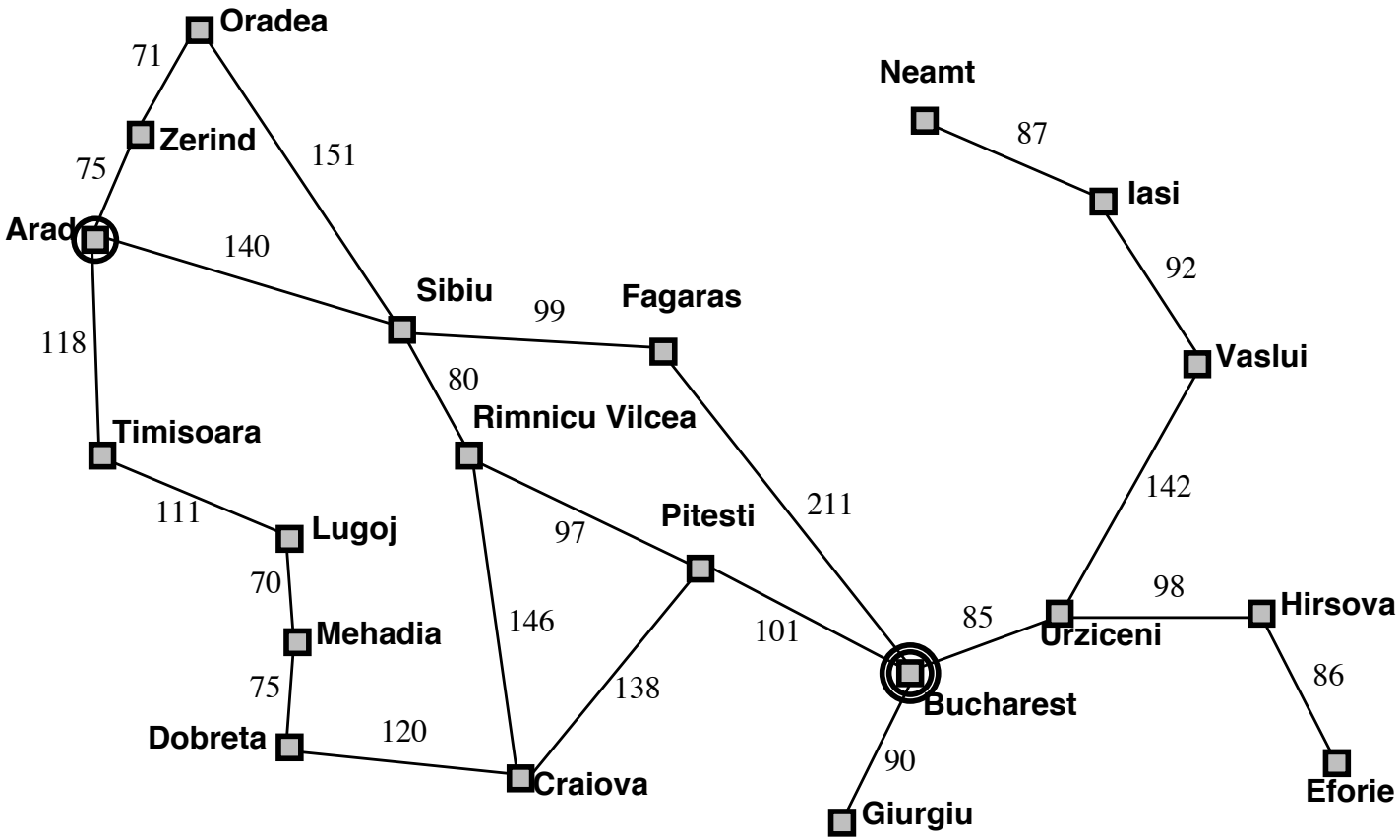
Suchstrategien

Tiefenbegrenzte Suche

- Falls man eine gute Schätzung für die maximale Anzahl von Lösungsschritten eines Suchproblems hat, lässt sich eine *tiefenbegrenzte Suche* nutzen.
  Damit lässt sich das Problem der Tiefensuche, manchmal in unendlichen Ästen zu verschwinden, wirksam lösen.
- Falls sich jedes Problem in einem Suchraum mit maximal d Schritten lösen lässt (und es keine kleinere Zahl mit dieser Eigenschaft gibt), heißt d der *Durchmesser* des Suchraums.
- Der Durchmesser des Suchraums ist damit ein guter Wert für die Tiefengrenze einer tiefenbegrenzten Tiefensuche.
- Für hinreichend einfache Probleme lässt sich der Durchmesser eines Suchbaums als längster Pfad des Spannbaums berechnen.

Suchstrategien

Tiefenbegrenzte Suche



Hier lässt sich beispielsweise jede Stadt von jeder anderen Stadt aus in neun Schritten erreichen. Damit ist der Durchmesser dieses Problems 9.

Tiefenbegrenzte Tiefensuche

function LIMITED-DFS(σ,A,G,d)
  $r \leftarrow$ fail
  $S \leftarrow$MK-STACK
  POP($S, (\sigma, \langle\rangle, 0, 0)$) ▷ Zustand, Aktionsfolge, Kosten, Tiefe
  $D \leftarrow \{\}$ ▷ Bereits entdeckte Knoten
  while $\neg$ EMPTY(S) **do**
    $(s, al, c, d') \leftarrow$ POP(S) ▷ Neuer expandierter Knoten
    if $s \in G$ **then**
      return (s, al, c, d')
    else if $d' > d$ **then** $r \leftarrow$ cutoff
    else if $s \notin D$ **then**
      $D \leftarrow D \cup \{s\}$ ▷ Als entdeckt markieren
      for each $(a, c') \in A$ **do**
        ENQUEUE($Q, (a(s), al \oplus \langle a\rangle, c + c', d' + 1)$)
  return FAIL
function ITERATIVE-DEEPENING-DFS(σ,A,G)
  for $d = 0, \ldots, \infty$ **do**
    $r \leftarrow$ LIMITED-DFS(σ,A,G,d)
    if $r \neq$ cutoff **then return** r

Tiefenbegrenzte Tiefensuche

- Vollständig und, bei konstanten Aktionskosten, optimal
- Wenn Lösung in Tiefe n gefunden wird: Speicherplatz $\mathcal{O}(b\,n)$
- Obwohl Zustände mehrfach besucht werden, ist die Zeitkomplexität nicht größer als bei Breitensuche
- Man kann bei der Breitensuche, beim Iterative Deepening und beim Dijkstra-Verfahren darauf verzichten, zu prüfen ob ein Knoten bereits besucht wurde
  - Alle drei Verfahren liefern den optimalen (kürzesten Pfad) – nur für die Tiefensuche muss das "Versacken" in Zyklen explizit ausgeschlossen werden.
  - Der Verzicht auf die Überprüfung kann zwar die mehrfache Expandierung eines Knotens bedeuten, aber andererseits erheblich Speicher und Rechenaufwand für die Verwaltung der "Entdeckungsliste" sparen.

Backtracking

- Betrachten wir folgende Zeilen der Tiefensuche:

  for each $(a, c') \in A$ **do**
  $\textsc{Push}(S, (a(s), al \oplus \langle a \rangle, c + c'))$

- Müssen wir wirklich alle Nachfolgezustände $a(s)$ erzeugen und auf den Stack packen? (Was, wenn Zustände groß/kompliziert sind, z.B. Schachbretter)
- Nein: Können auch einfach erste Aktion a ausführen, und uns merken, welche anderen Aktionen noch getestet werden müssen
- Wenn Aktionen $a_1, \ldots, a_n$ auf s anwendbar sind, erzeugt man zunächst $a_1(s)$, und sucht von dort aus weiter. Wenn dies nicht zum Erfolg führt, versucht man $a_2(s)$, danach $a_3(s)$ usw. bis $a_n(s)$. Wenn auch dies nicht zum Ziel führt geht man zum Vorgänger von s zurück, und versucht es von dort aus weiter. Dieses Verfahren ist auch als Backtracking bekannt.

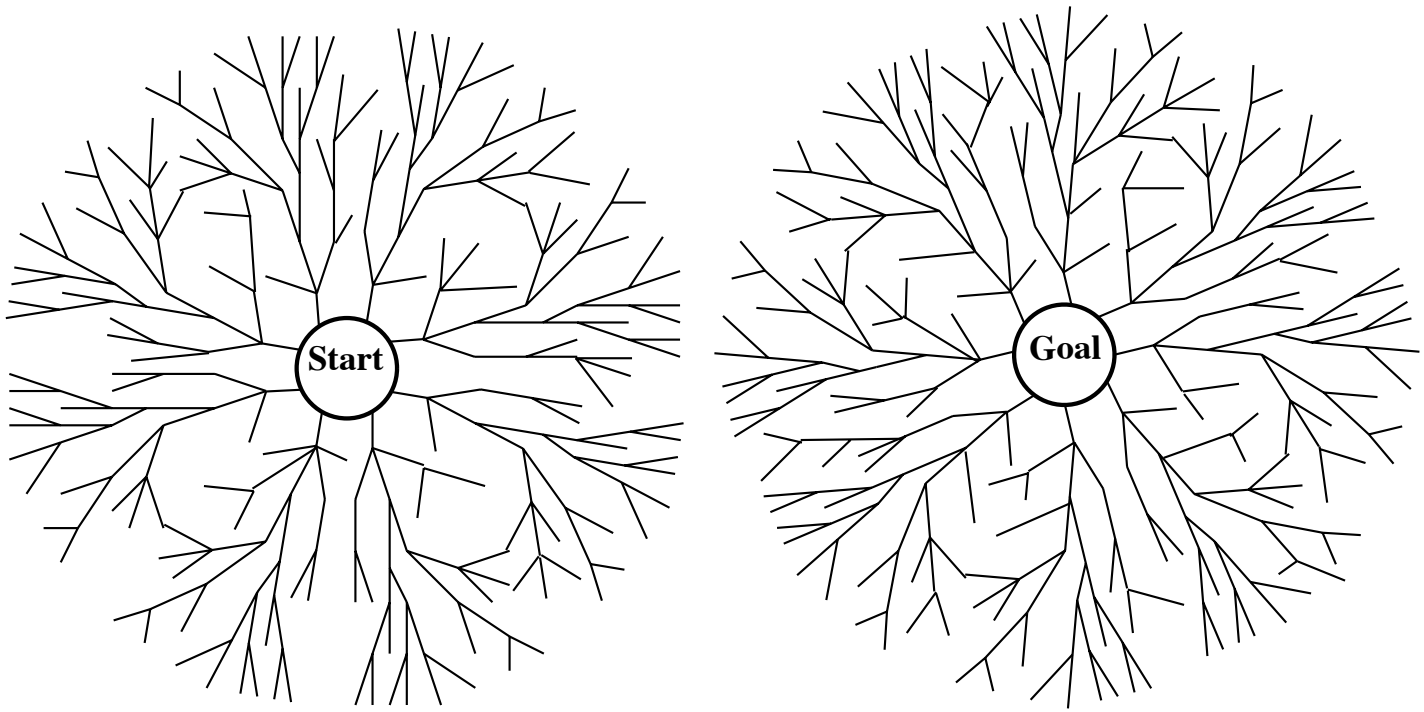
Backtracking

- Backtracking lässt sich sehr gut rekursiv implementieren:
- eine Funktion Backtrack(s) ist für die Expansion eines Knotens s verantwortlich.
- Backtrack(s) enthält eine Schleife, in der sequentiell alle Aktionen a_i für s getestet werden...
- und in der für jedes $a_i(s)$ die Tiefensuche durch den rekursiven Aufruf von Backtrack($a_i(s)$) weiter fortgeführt wird.

function BACKTRACK(s,A,G)
  if $s \in G$ **then**
    return (s, al, c)
  for each $(a, c') \in A$ **do**
    $(s', al, c) \leftarrow r \leftarrow$ BACKTRACK$(a(s), A, G)$
    if $r \neq$ FAIL **then return** $(s', al \oplus \langle a \rangle, c + c')$
  return FAIL

Suchstrategien

Bidirektionale Suche



Sofern Vorwärts- und Rückwärtssuche symmetrisch sind, erreicht man Suchzeiten von

$$O(2b^{d/2}) = O(b^{d/2})$$

z. B. für $b = 10$, $d = 6$ statt 1111111 nur 2222 Knoten.

Suchstrategien: Bidirektionale Suche

Probleme

- Operatoren nicht immer umkehrbar oder nur sehr schwer. (Berechnung der Vorgängerknoten)
- Man benötigt effiziente Verfahren, um zu testen, ob sich die Suchverfahren „getroffen“ haben.
- Welche Art der Suche wählt man für jede Richtung? (Im Bild Breitensuche)
- Bei Breitensuche: Speicherung von einem der Bäume zwecks Vergleich. (Speicher $O(b^{d/2})$)
- Nur plausibel bei *sehr wenig* Zielzuständen
  - Gegenbeispiel: Zielzustand „Schachmatt“ ...

Informierte Suche

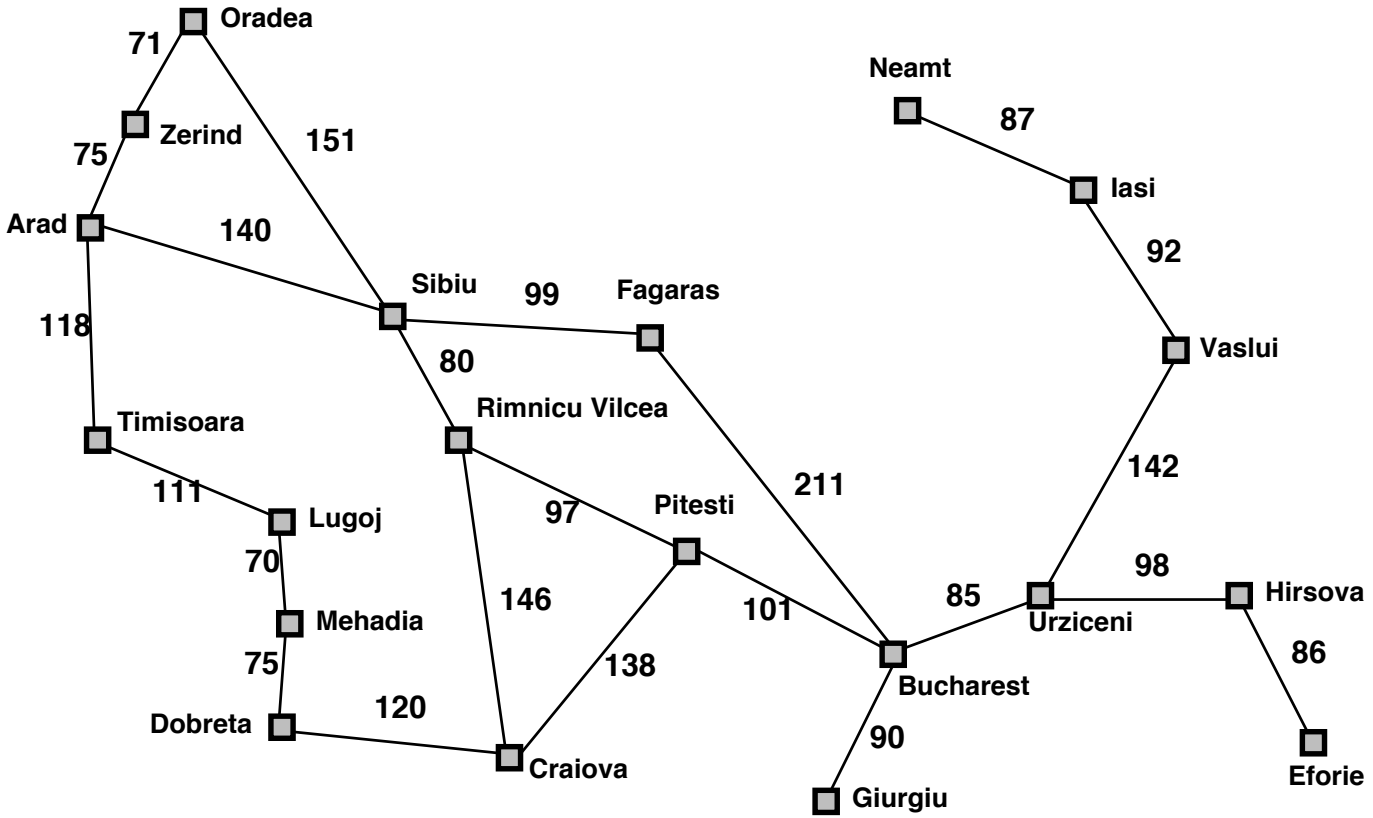
- In manchen Fällen kann man “schlauer” entscheiden, welcher Knoten als nächstes expandiert werden soll
- D.h. Knoten werden in einer Priority Queue gespeichert, wobei “aussichtsreichere” Knoten eine höhere Priorität haben
- Konkret: Sei $h : S \to \mathbb{R}$ eine *Heuristik*, die eine Abschätzung für die Distanz von s zum Ziel angibt
- Wir schauen uns zwei Fälle an:
  - greedy Suche
  - A^* Suche

Greedy Suche

Einfacher Fall: Priorität hängt ausschließlich von Heuristik ab

function GREEDY-SEARCH(σ,A,G)
  $Q \leftarrow$MK-PRIORITY-QUEUE
  ENQUEUE($Q, (\sigma, \langle\rangle, h(\sigma))$)
  $D \leftarrow \{\}$ ▷ Bereits entdeckte Knoten
  while $\neg$ EMPTY(S) **do**
    $(s, al, c) \leftarrow$ DEQUEUE-MIN(Q) ▷ Neuer expandierter Knoten
    if $s \in G$ **then**
      return (s, al, c)
    else if $s \notin D$ **then**
      $D \leftarrow D \cup \{s\}$ ▷ Als entdeckt markieren
      for each $(a, c') \in A$ **do**
        ENQUEUE($Q, (a(s), al \oplus \langle a\rangle, h(s))$)
  return FAIL

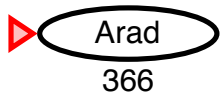
Beispiel: Rumänien



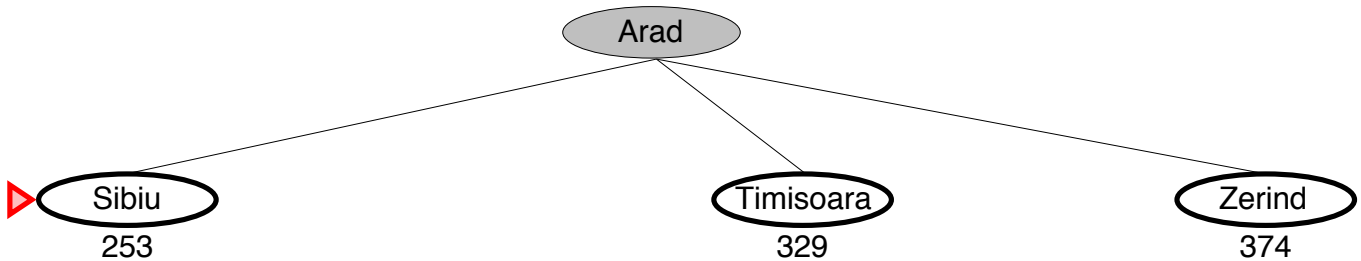
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

- Kosten: Distanzen zwischen Städten
- Heuristik: Luftlinien-Distanz nach Bukarest

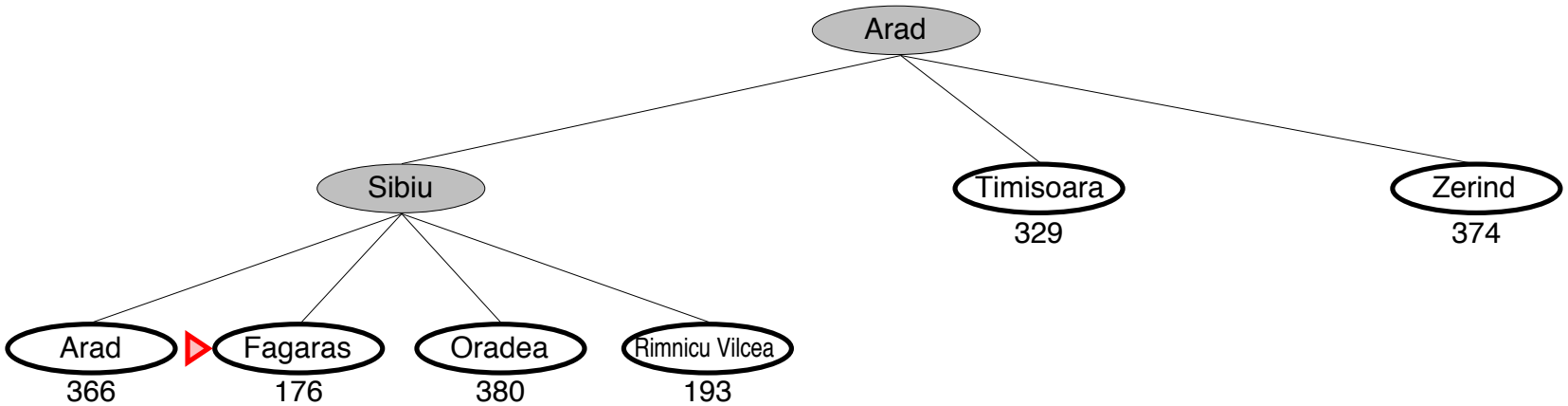
Beispiel: Greedy Suche



Beispiel: Greedy Suche



Beispiel: Greedy Suche



Eigenschaften der Greedy Suche

- Kann in Schleifen hängenbleiben, wenn naiv implementiert: Muss sich merken, welche Knoten schon besucht wurden
- Zeitkomplexität: $\mathcal{O}(b^m)$ – aber gute Heuristik kann das dramatisch verbessern ($m =$ maximale Tiefe des Suchbaums)
- Platzkomplexität: $\mathcal{O}(b^m)$ – muss alle Knoten im Speicher halten
- Nicht optimal (findet nicht unbedingt den kürzesten Weg)

A^* Suche

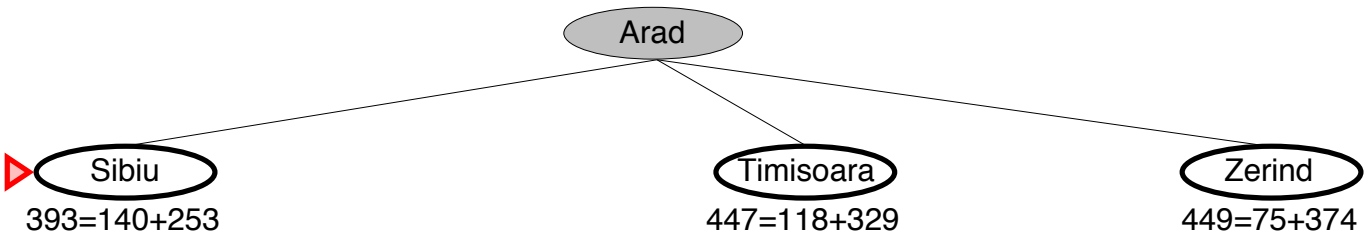
- Wie können wir Greedy-Suche verbessern?
- Idee: Vermeide Expansion von Pfaden, bei denen der bisher zurückgelegte Weg sehr teuer ist
- Funktion $f(s) = g(s) + h(s)$, wobei
  - $g(s)$: Bisherige Kosten des Pfades zu s
  - $g(s)$: Heuristik, Abschätzung der Kosten von s zum Ziel
  - $f(s)$: Abschätzung der gesamten Kosten vom Start zum Ziel über s
- Heuristik h heißt *zulässig*, wenn sie die tatsächlichen Kosten von s zum Ziel nie überschätzt
- Satz: A^* mit zulässiger Heuristik ist optimal.

A^* Suche

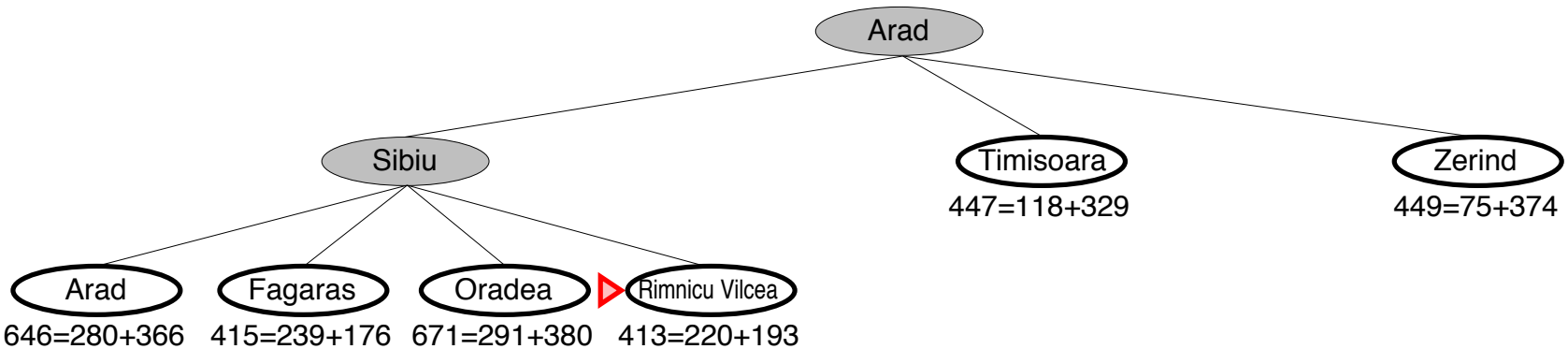
Verbessere Greedy Suche: Wähle den Knoten, für den die Abschätzung für den Weg insgesamt (Kosten bis zum Knoten + Heuristik vom Knoten bis zum Ziel) am geringsten ist

function A^*-SEARCH(σ,A,G)
  $Q \leftarrow$MK-PRIORITY-QUEUE
  ENQUEUE($Q, (\sigma, \langle\rangle, h(\sigma))$)
  $D \leftarrow \{\}$ ▷ Bereits entdeckte Knoten
  while $\neg$ EMPTY(S) **do**
    $(s, al, c) \leftarrow$ DEQUEUE-MIN(Q) ▷ Neuer expandierter Knoten
    if $s \in G$ **then**
      return (s, al, c)
    else if $s \notin D$ **then**
      $D \leftarrow D \cup \{s\}$ ▷ Als entdeckt markieren
      for each $(a, c') \in A$ **do**
        ENQUEUE($Q, (a(s), al \oplus \langle a \rangle, c + c' + h(s))$)
  return FAIL

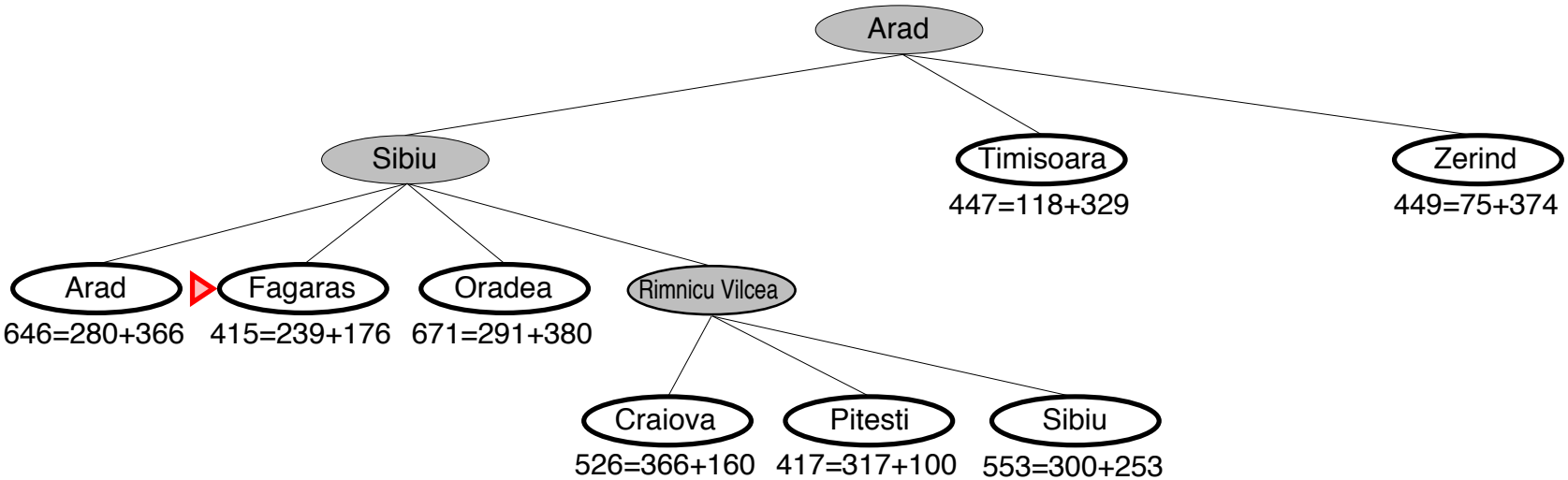
Beispiel: A^* Suche



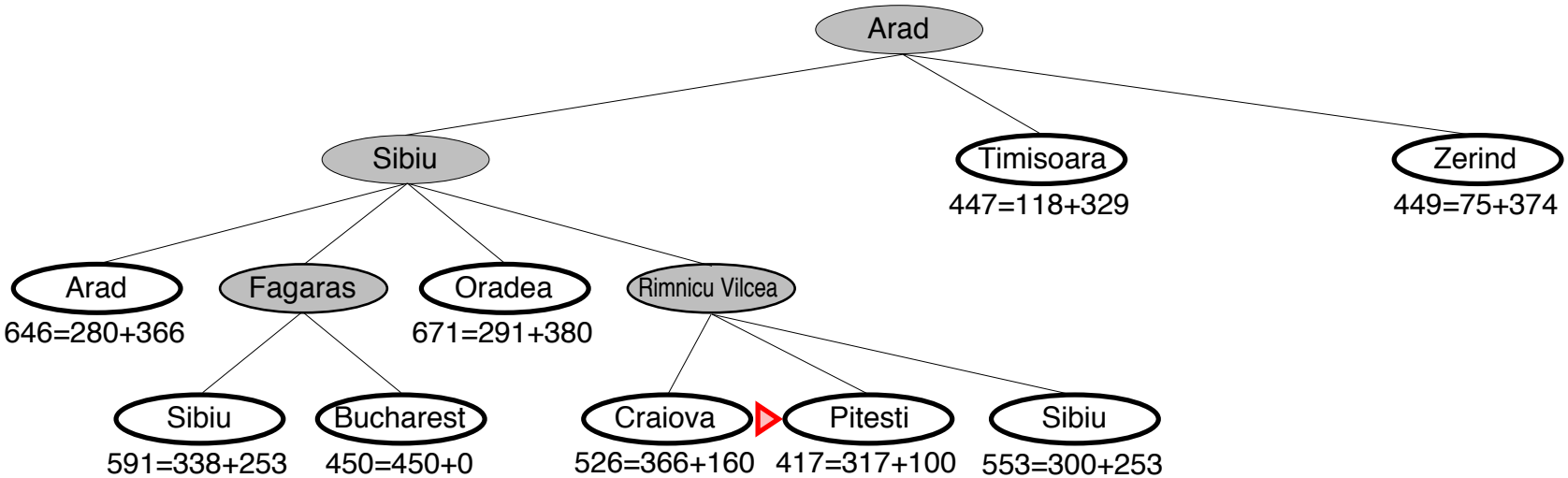
Beispiel: A^* Suche



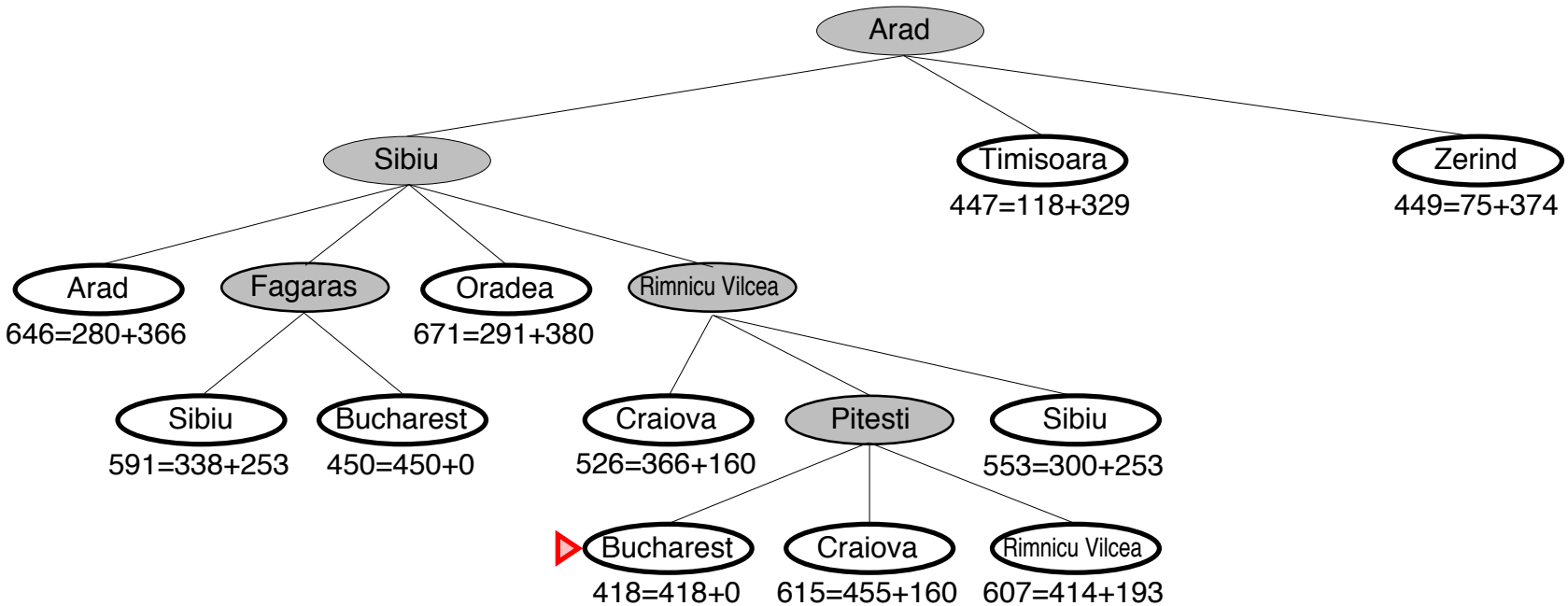
Beispiel: A^* Suche



Beispiel: A^* Suche

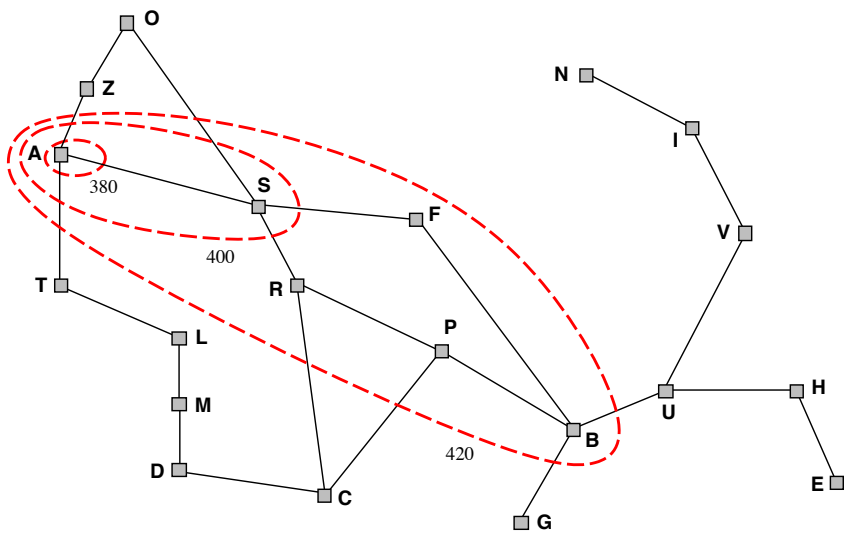


Beispiel: A^* Suche



Optimalität von A^*: Beweisidee

- Ein von A^* expandierter Knoten kann nie einen geringeren f-Wert haben als ein zuvor expandierter Knoten
- D.h. A^* erweitert nach und nach "f-Konturen": Kontur i enthält alle Knoten mit $f = f_i$, wobei $f_i < f_{i+1}$
- Seien C^* die Kosten des optimalen Pfads
- A^* expandiert alle Knoten mit $f(s) < C^*$
- A^* kann dann einige Knoten mit $f(s) = C^*$ expandieren, anschließend den Zielknoten
- Der so gefundene Pfad ist optimal



A^*: Eigenschaften

- Optimal: Findet garantiert einen kürzesten Weg zwischen Start und Ziel
- Zeitkomplexität
  - Seien h^* die tatsächlichen Kosten zum Ziel, und sei $\epsilon = (h^* - h)/h^*$ der *relative Fehler* der Heuristik h
  - Zeitkomplexität von A^*: $\mathcal{O}(b^{\epsilon n})$
- A^* ist *optimal effizient*: Für eine gegebene Heuristik kann es keinen Algorithmus geben, der einen optimalen Pfad in weniger Schritten findet
- Platzkomplexität $\mathcal{O}(b^{\epsilon n})$: Muss wieder alle Knoten im Speicher halten
- Insgesamt: Oftmals zu aufwendig, optimale Lösung zu finden
  - Approximativen Algorithmus verwenden, oder
  - A^*, aber nicht-zulässige (aber trotzdem nützliche) Heuristik verwenden

Zusammenfassung

- Problem, das wir in diesem Kapitel betrachtet haben:
  - Gegeben: Beobachtbare, statische Umgebung, Start- und Zielzustand
  - Gesucht: Aktionsfolge, um Start- in Zielzustand zu überführen
- Solche Probleme können als *Suchproblem* formalisiert werden
- Schwierigkeit, Lösung zu finden, kann wesentlich von Problemformalisierung abhängen (Abstraktion)
- Uninformierte Suchalgorithmen: Tiefensuche, Breitensuche, Dijkstra
- Informierte Suchalgorithmen (benutzen Heuristik, die "Aussichtsreichtum" eines Knotens beschreibt): Greedy Suche, A^*