Künstliche Intelligenz

Problemlösen als Suche – Teil I

Dr. Christian Meilicke
Research Group Data and Web Science
Universität Mannheim

Teile der Vorlesung basieren auf einem
Foliensatz von Prof. Dr. Heiner Stuckenschmidt

Rückblick

Erneuter Hinweis Seminar

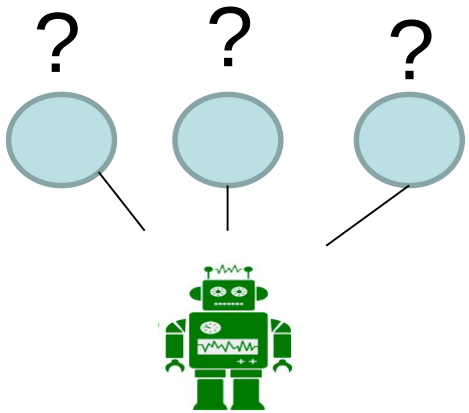
- Organisatorisches
- Was ist KI
  - Algorithmen die zu rationalem Verhalten in bestimmten Problemsituationen führen
- Geschichte der KI
  - Auf und ab, aktuell befinden wir uns in deutlichem Aufwärtstrend
  - Gestiegene Rechenleistung, große Mengen verfügbarer Daten, Wirtschaft investiert
- Agentenmetapher
  - Fokus auf zielbasiertem Agent
- 6 Dimensionen, um die Schwierigkeit von Problemen zu beschreiben
  - Beobachtbarkeit (partiell vs. vollständig)
  - Probablistisch vs. deterministisch
  - Sequentiell vs. episodisch
  - Statisch vs. dynamisch
  - Diskret vs. kontinuierlich
  - Ein Agent vs. mehrere Agenten

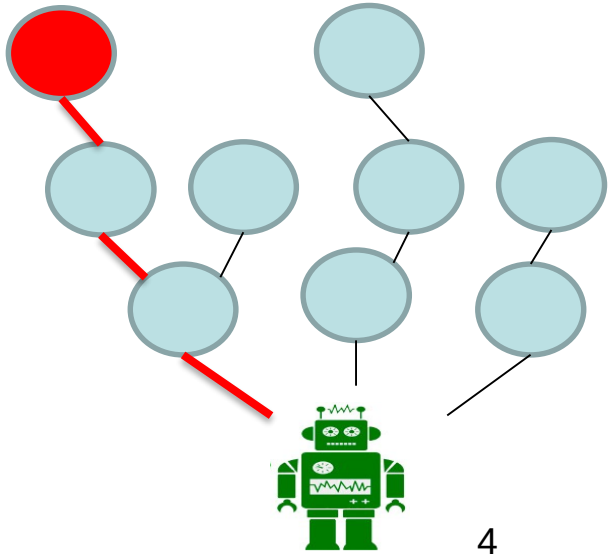
Inhalt Heute

- Definition Suchproblem / Zustandsraum
  - Beispiele
- Allgemeines Muster eines Algorithmus
- Qualitätskriterien
  - Vollständigkeit, Optimalität, Zeit und Speicherkomplexität
- Grundlegende Methoden
  - Breitensuche
  - Tiefensuche
- Varianten der Tiefensuche
  - Backtracking-Suche
  - Tiefenbegrenzte Suche
  - Iterative Tiefensuche

Suche im Zustandsraum

- Nutzen einer Handlungsalternative kann oft nicht direkt bestimmt werden
- Agenten müssen daher Sequenzen von Handlungen im Zustandsraum vergleichen und die beste aussuchen
  - Nur solche Agenten, die über eine Repräsentation der Umwelt verfügen, können überhaupt einen Zustandsraum aufbauen
- Die Konstruktion und der Vergleich solcher Handlungssequenzen wird **Suche** genannt
- Ist eine Handlungssequenz ausgesucht, wird die entsprechende Handlungssequenz (oder Teile davon) ausgeführt

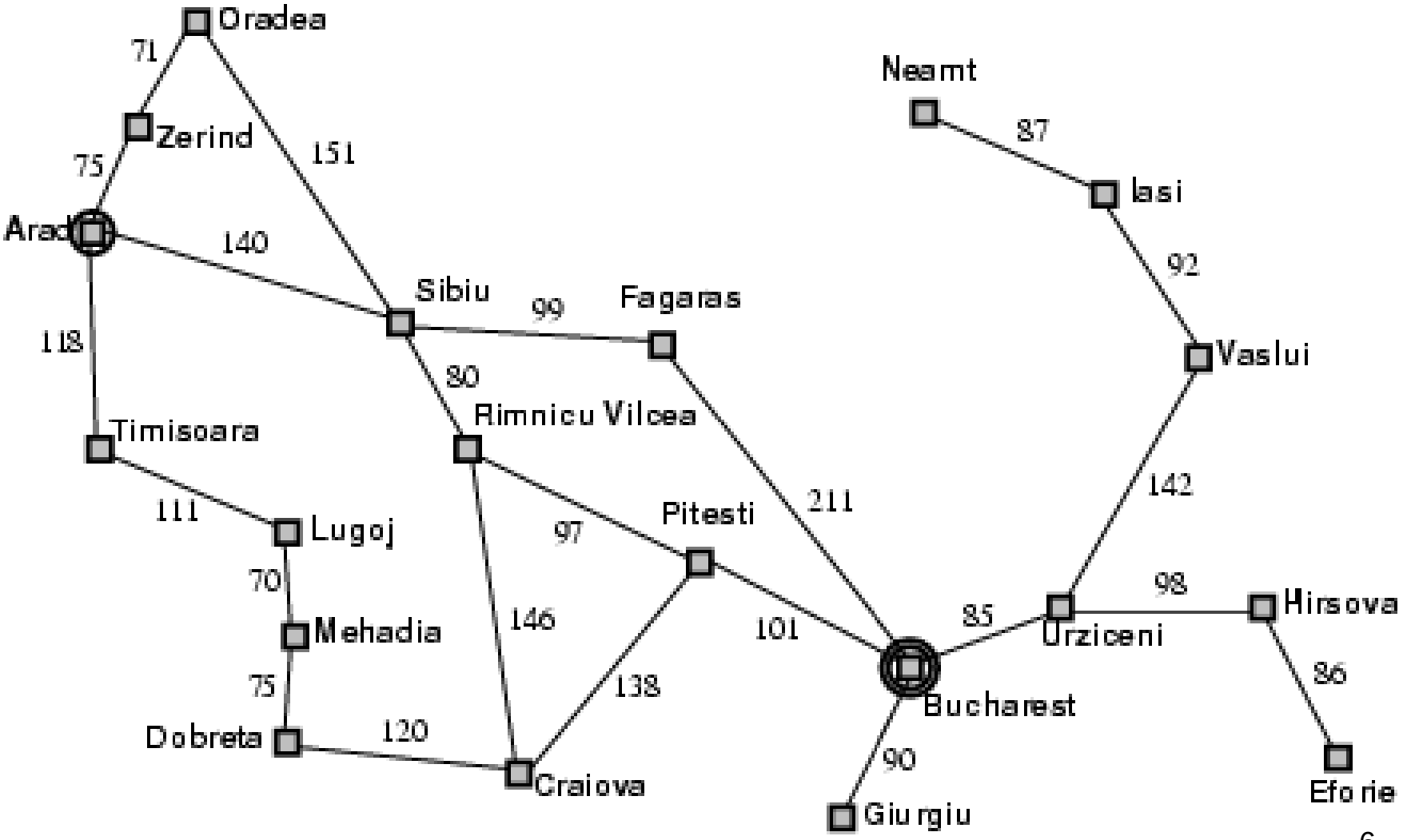




Definition Suchproblem

- Ein Suchproblem besteht aus:
  - einem **Ausgangszustand**: *In(Arad)*
  - möglichen **Aktionen** und deren **Wirkung**: *[Go(Sibiu),In(Sibiu)]* *(= Indirekte Repräsentation des Zustandsraums !)*
  - Einem **Zieltest**: *In(Bucharest) ?*
  - (OPTIONAL oder IMPLIZIT) Einer **Kostenfunktion** *c(In(Arad), Go(Sibiu), In(Sibiu)) = 140*
- Lösung eines Suchproblems ist eine Sequenz von Handlungen, die vom Ausgangs- in einen Zielzustand führt

Standard Problem: Routenplanung

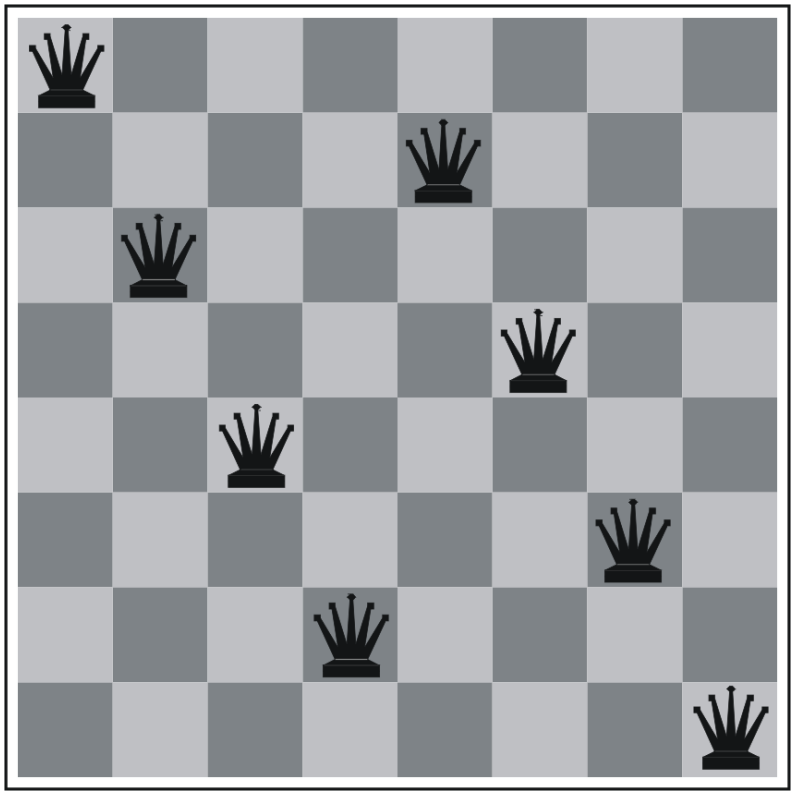


Andere Arten von Problemen

- Nicht immer leicht zu sehen, was Ausgangszustand, Zieltest, und mögliche Aktionen (mit Kostenfunktion) sind:

8-Damen-Problem

- Das 8 Damen-Problem:
  - Stelle 8 Damen so auf ein Schachbrett, dass keine die andere schlagen kann
  - Hier ist eine falsche Lösung:



- Nicht so einfach, oder ?
  - (es gibt 92 Lösungen...aber leider auch $3 * 10^{14}$ mögliche Zustände)
  - Macht nichts, Carl Friedrich Gauss hat auch nur 72 gefunden

8-Damen-Problem

- Klar ist, dass keine zwei Damen in derselben Spalte stehen dürfen, dies können wir für eine einfache Zustandsbeschreibung nutzen

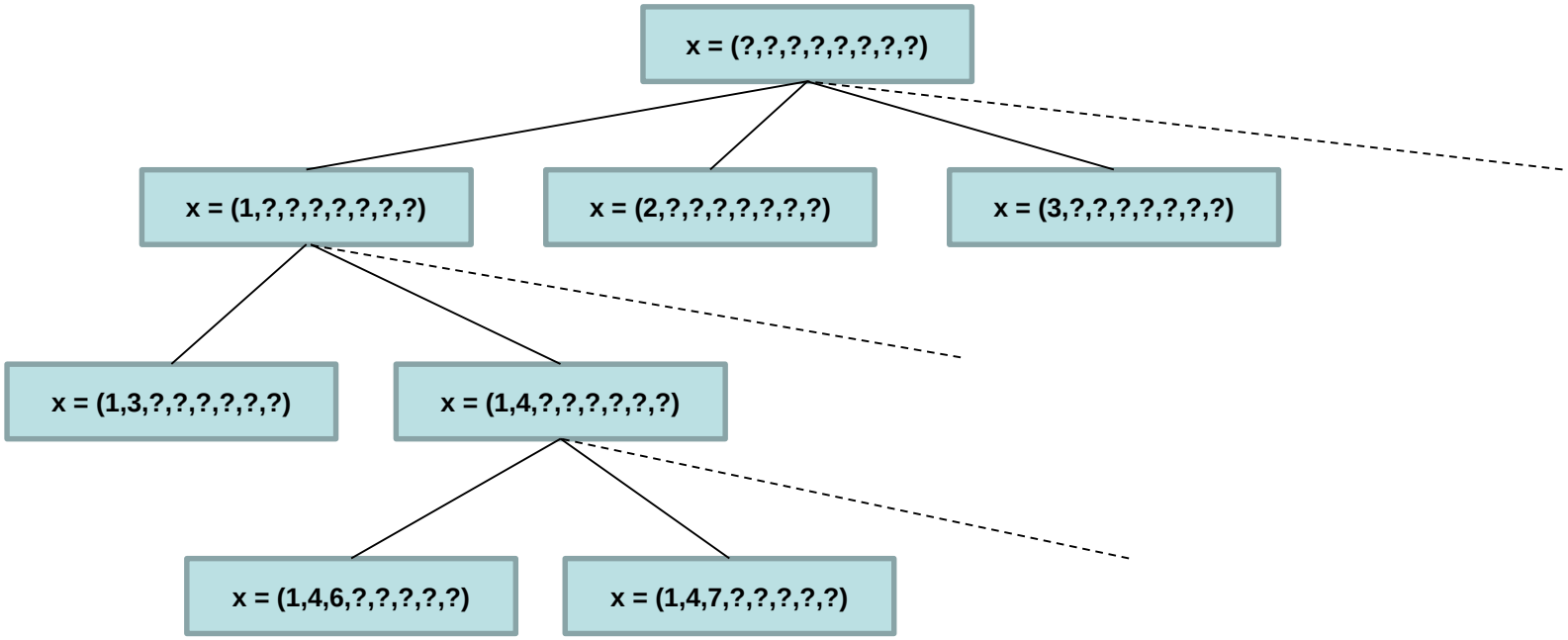
- Beispiel: Es befindet sich eine Dame in Spalte 1, Zeile 2 und eine Dame in Spalte 2, Zeile 4
  - Es sei x ein Zustand in der Suche, wobei x ein Vektor mit 8 Elementen ist
  - $x = (2, 4, ?, ?, ?, ?, ?, ?)$ wobei x_i den i-ten Eintrag in x bezeichnet

- Eine Aktion $put(s, z)$ ist das Setzen einer Dame in Zeile z der Spalte s, wobei der Zug nur erlaubt ist, wenn:
  - $x_s = ? \wedge x_{s-1} \neq ?$
  - $z \neq x_i$ für alle $i < s$
  - $z \neq x_i + (s - i)$ für alle $i < s$
  - $z \neq x_i - (s - i)$ für alle $i < s$
  - $1 \leq z \leq 8$

8-Damen-Problem

- Wirkung einer Aktion
  - $put\ (s, z) \mapsto\ x_s\ =\ z$
    (unter den auf der vorigen Folie erwähnten Nebenbedingungen)
- Startzustand
  - $x\ =\ (?, ?, ?, ?, ?, ?, ?, ?)$
- Zieltest
  - $x_i \neq ?$ für alle i mit $1\ \leq i\ \leq 8$
- Kostenfunktion
  - Gibt es bei diesem Problem nicht, alternativ: Kosten jeder Aktion = 1

8-Damen-Problem- Zustandsraum



Wieviele Blattknoten
hat der Suchbaum maximal?

Exkurs I: Komplexität

- Bei jedem Suchverfahren durchsuchen wir Teile dieses Suchraums
  - D.h wir bauen eines Suchbaums auf bzw. laufen diesen mit einer bestimmten Strategie ab
- Wie lange dauert es, wie wieviel Speicher benötigen wir?
- "Worst-Case" Komplexität:
  - Maximaler Zeit bzw. Speicherverbrauch in Abhängigkeit der Problemgrösse p
  - Erinnerung: O-Notation
    - O(f(p)): Es werden maximal c + a * f(p) Schritte/Speichereinheiten benötigt, z.B. $O(\lg(p)) < O(p) < O(\lg(p) * p) < O(p^2) < O(2^p)$

Exkurs II: Exponentielles Wachstum

- Vorige Folie: 2^p Allgemein: $c * a^p$
- Beispiel 1: Suchbaum mit Branchingfaktor 2, p = Ebenen

$= 1 * 2^p$

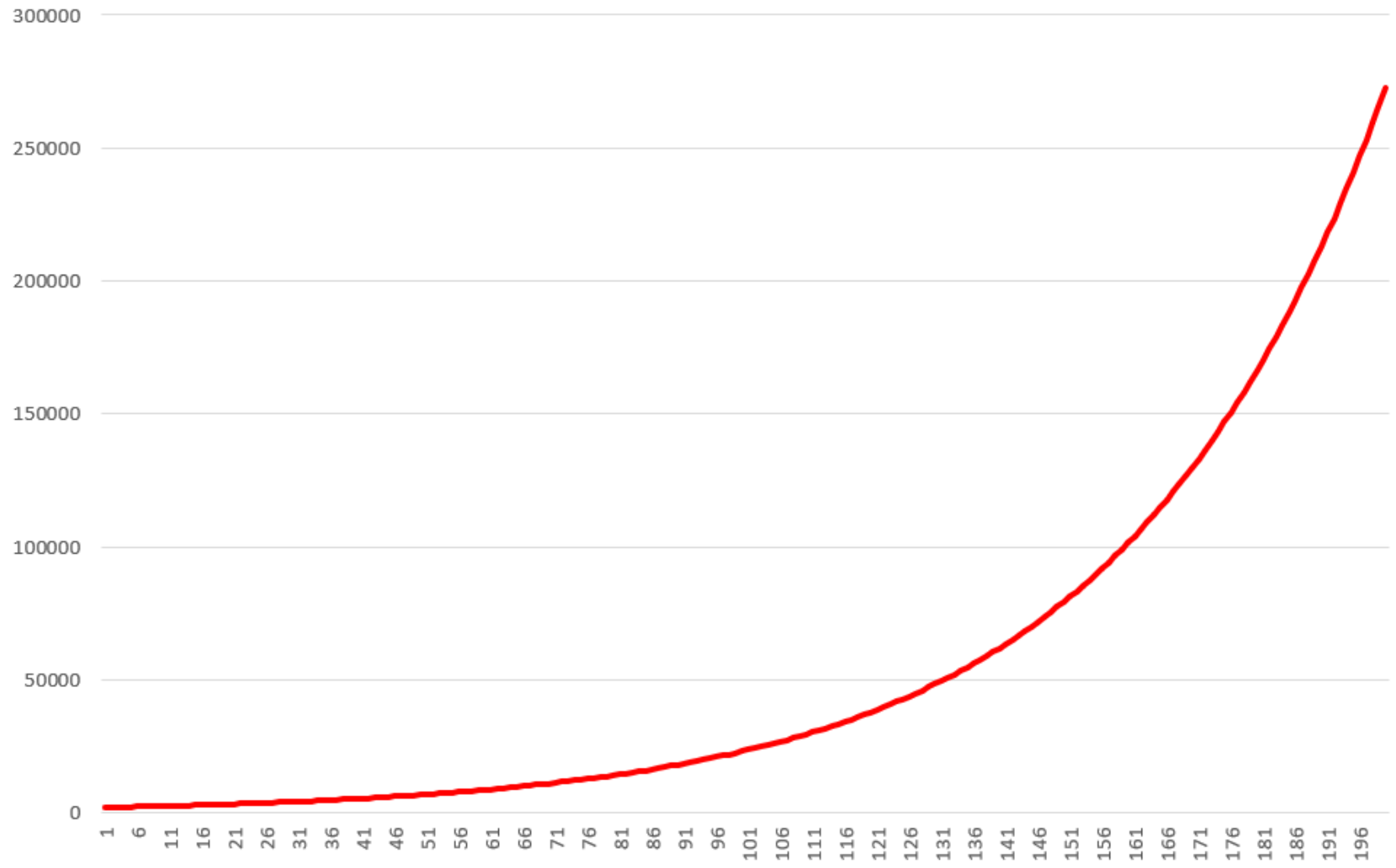
- Beispiel 2: Geldanlage 1000 Euro mit 3% Zinsen pro Jahr (p)

$= 1000 * 1.03^p$

- Beispiel 3: Pandemie mit 2000 Neuinfizierten täglich, jeden Tag steigt die Anzahl der täglichen Neuinfektionen um ca. 2.5%

$= 2000 * 1.025^p$

Exkurs II: Beispiel 3

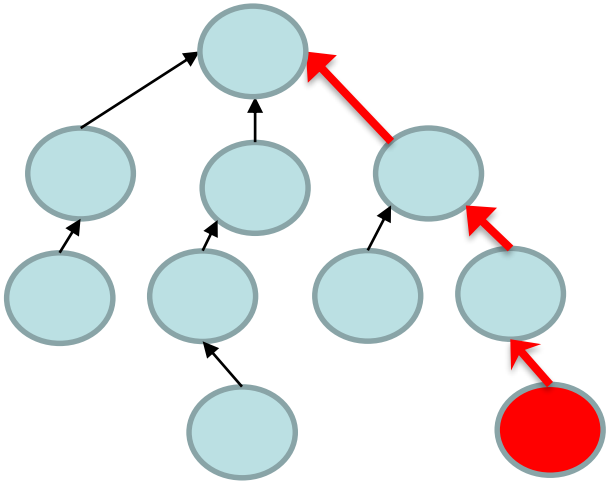


Analoge Ziele

- Suchbaum (KI Vorlesung 02+03)
  - Zielgerichtet durchlaufen, so dass vieles nie betrachtet wird
  - Vermeide große Teile des Baums zugleich im Speicher zu haben
- Spielbaum (KI Vorlesung 04+05)
  - Bestimmte Teile des Suchbaums abschneiden, um Branchingfaktor zu reduzieren (Pruning)
  - Auf vielversprechende Stellen konzentrieren
- Pandemie (die nächsten Monate/Jahre)
  - Isolation Infizierter um weitere Verbreitung einzudämmen (= Pruning)
  - Branchingfaktor durch allgemeine Verordnungen reduzieren (Ändern des gegebenen Problems / Ändern der Spielregeln)

Lösung als Sequenz von Handlungen

- In einer konkreten Situation interessiert nur der nächste Schritt
- Algorithmus endet aber am Zielzustand
- Konstruktion des Weges durch “Zeiger” (Referenzen) die z.B. innerhalb der `expand` Methode gesetzt werden
- Jeder expandierte Zustand erhält einen Zeiger auf den Zustand, aus dem er generiert wurde, d.h. jeder Zustand zeigt auf seinen Vorgänger im Baum
- Die Lösung (= der Weg) kann leicht ausgelesen werden, damit ist auch der nächste Schritt bekannt



Qualitätskriterien

- Vollständigkeit
  - findet das Verfahren immer (irgend)eine Lösung, wenn es eine gibt?
- Optimalität
  - findet das Verfahren die Lösung mit den geringsten Kosten, wenn das Problem eine Lösung hat?
- Zeit-Komplexität
  - Wie viel Zeit braucht das Verfahren im Verhältnis zur Größe des Problems?
- Speicher-Komplexität
  - Wieviel Speicherplatz belegt das Verfahren im Verhältnis zur Größe des Problems?

Parameter der Problemgrösse

- Tiefe der Lösung: **d** (depth)
- Maximale Tiefe des Suchbaums: **m** (m >> d) (maximal)
- Verzweigungsfaktor: **b** (branching factor)
- Kosten der Lösung: **c** (cost)
- Minimale Kosten pro Schritt: **e** (epsilon)

Suchstrategien

- Grundlegende Methoden
  - Breitensuche
  - Tiefensuche
- Varianten der Tiefensuche
  - Backtracking-Suche
  - Tiefenbegrenzte Suche
  - Iterative Tiefensuche
- Kosten-Sensitive Suche (auch Bestensuche)
  - Uniforme Kostensuche
  - Greedy Suche
  - A* Suche

Allgemeiner Algorithmus

```
List todo = [startState]
DO LOOP
   IF todo = []
      RETURN "Fail, no solution found"
   ELSE
      State s = selectState(todoList)
      IF isSolution(s)
         RETURN "Solution found"
      ELSE
         List expandedStates = expand(s)
         add(expandedStates, todo)
```

Ein Suchproblem besteht aus:

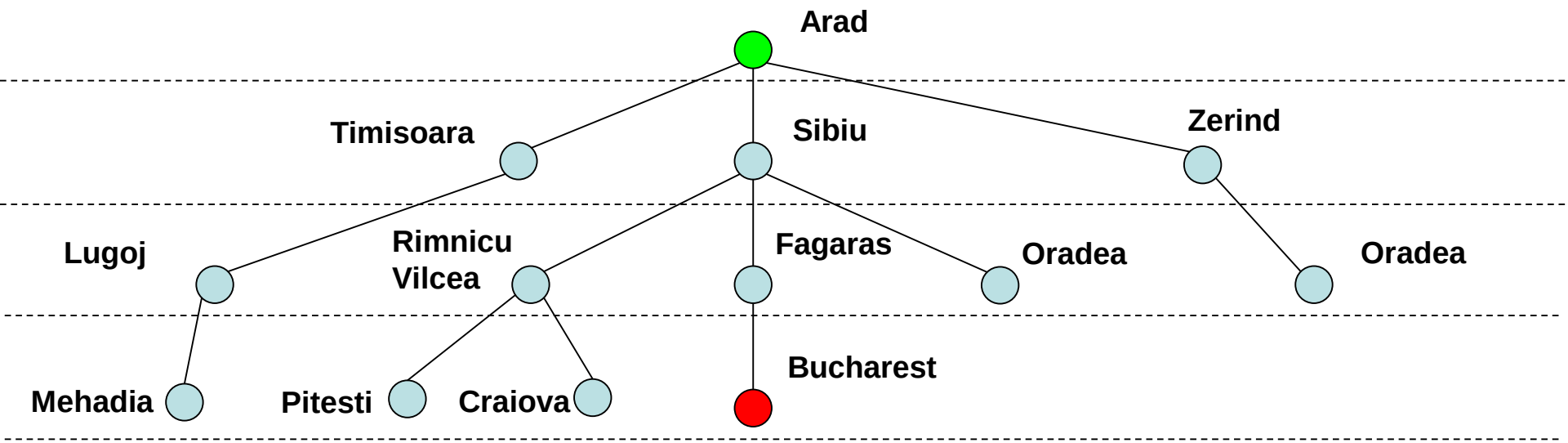
- einem Ausgangszustand
- möglichen Aktionen und deren Wirkung
- einem Zieltest
- einer Kostenfunktion (kann in selectState eine Rolle spielen)
- Suchverfahren unterscheiden sich in Bezug auf die Methode selectState

Die Lösung eines Suchproblems ist eine Sequenz von Handlungen, die vom Ausgangs- in einen Zielzustand führt.

Duplikat-Eliminierung kann in der add Funktion implementiert sein.

Breitensuche

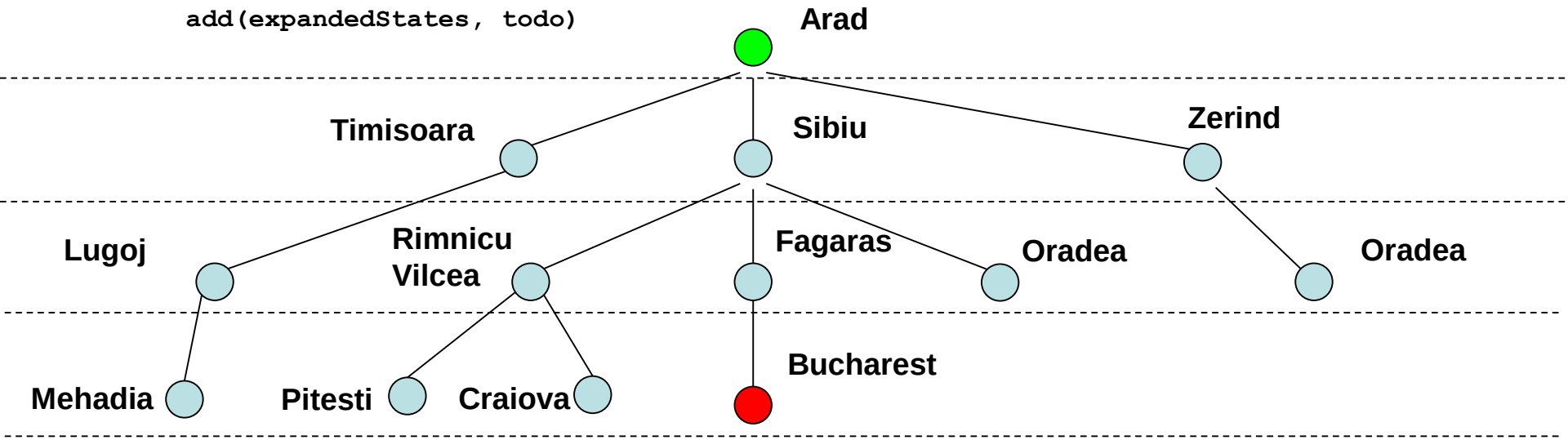
Todo Liste ist eine FIFO-Queue:
Eher eingefügte Knoten werden eher selektiert



Breitensuche

```
FiFoQueue todo = [startState]
DO LOOP
   IF todo = [] RETURN "Fail"
   ELSE
      State s = selectState(todoList)
      IF isSolution(s) RETURN "Solution found"
      ELSE
         List expandedStates = expand(s)
         add(expandedStates, todo)
```

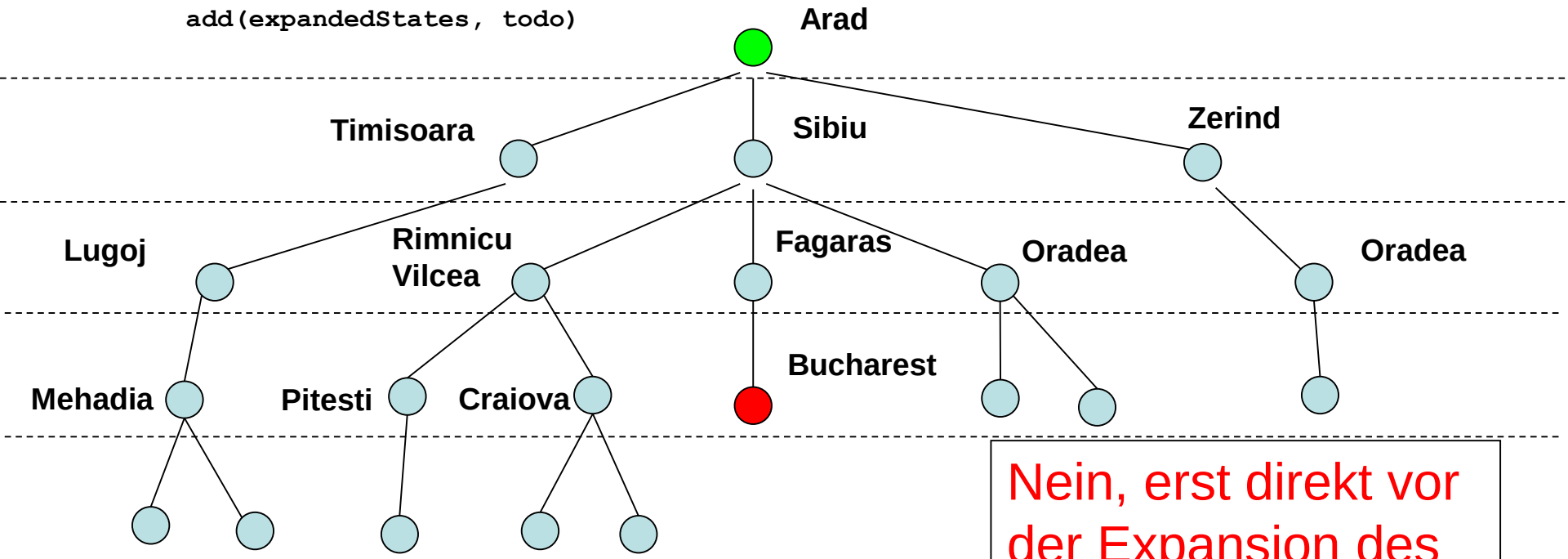
Achtung: Stoppt der Algorithmus wirklich bereits jetzt?



Breitensuche

```
FiFoQueue todo = [startState]
DO LOOP
   IF todo = [] RETURN "Fail"
   ELSE
      State s = selectState(todoList)
      IF isSolution(s) RETURN "Solution found"
      ELSE
         List expandedStates = expand(s)
         add(expandedStates, todo)
```

Achtung: Stoppt der Algorithmus wirklich bereits jetzt?



Nein, erst direkt vor der Expansion des Zielknotens!

Eigenschaften: Breitensuche

- Vollständigkeit:

  JA: der erste Lösungsknoten wird gefunden, wenn alle Knoten mit Tiefe von d oder kleiner expandiert wurden

- Optimalität:

  JA/NEIN: Nur optimal, wenn Tiefe die Kosten bestimmt. Dies ist der Fall wenn alle Schritte gleiche Kosten aufweisen.

- Zeitkomplexität:

  $O(b^d)$: Es müssen (fast) alle möglichen Knoten bis Tiefe d expandiert werden.

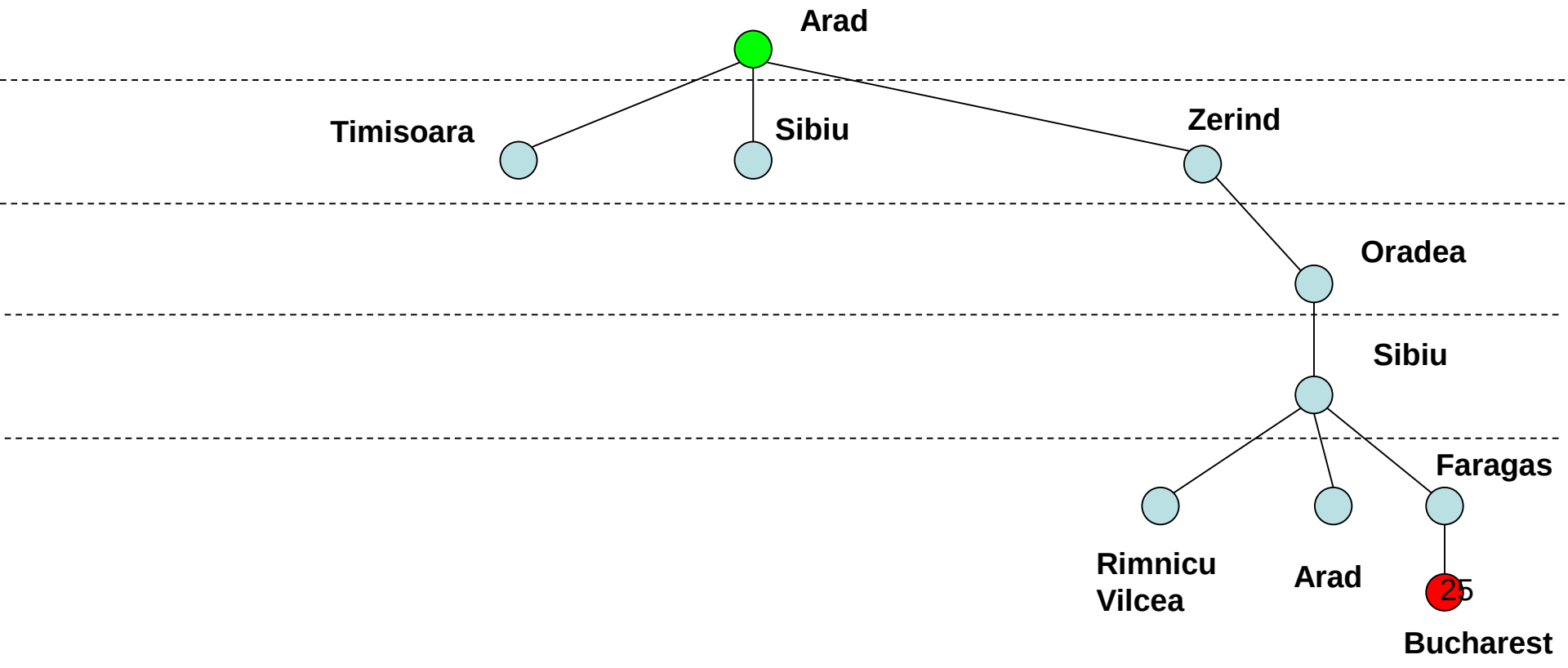
- Speicherkomplexität:

  $O(b^{d+1})$: Es müssen (fast) alle möglichen Knoten bis Tiefe d+1 gespeichert werden.

d = Tiefe der Lösung
m = Max.Tiefe des Suchbaums
b = Branching Faktor
c = Kosten der Lösung
e = Minimale Kosten pro Schritt

Tiefensuche

ToDoListe ist eine LIFO-Queue:
Zuletzt eingefügte Knoten werden eher selektiert



Eigenschaften: Tiefensuche

- Vollständigkeit:

  NEIN: Der Suchraum kann unendlich sein. Beginnt die Suche in einem unendlichen Zweig, werden Lösungen in anderen nicht gefunden

- Optimalität:

  NEIN: Der zuerst expandierte Zweig kann eine Lösung enthalten, die in einer größere Tiefe liegt als die optimale Lösung

- Zeitkomplexität:

  $O(b^m)$: Es müssen (fast) alle möglichen Knoten eines Zweigs expandiert werden, auch wenn keine Lösung enthalten ist.

- Speicherkomplexität:

  $O(b*m)$: Es muss immer nur ein Zweig gespeichert werden. Ist er vollständig durchsucht, kann er gelöscht werden.

d = Tiefe der Lösung
m = Max.Tiefe des Suchbaums
b = Branching Faktor
c = Kosten der Lösung
e = Minimale Kosten pro Schritt

Diskussion I

- Breitensuche (FIFO: first in, first out)
  - ist optimal und vollständig
  - Hat aber gravierende Komplexitätsprobleme
- Tiefensuche (LIFO: last in, first out)
  - reduziert den Speicherbedarf drastisch
  - gibt dafür jedoch Vollständigkeit und Optimalität auf
- In der Praxis wird meist Tiefensuche verwendet, da
  - für Breitensuche der Speicher meist nicht ausreicht
  - meist Lösungen mittels Tiefensuche gefunden werden
  - oft suboptimale Lösungen akzeptabel sind

Diskussion II

- Duplikateleminierung bei Tiefensuche führt zu ähnlicher Speicherkomplexität wie bei Breitensuche
  - Verhindert “Besuch einer Endlosschleife”

- Duplikateleminierung bei Breitensuche sinnvoll, wenn man auf verschiedenen Wegen denselben Zustand erreicht.
  - Insbesondere dann angemessen, wenn der Zustandsraum verhältnismäßig klein ist (Beispiel: Landkarte mit Städten)
  - Speichern jedes besuchten Zustandes zusammen mit der Information, auf welcher Ebende der Zustand liegt (später: wie teuer der Weg dorthin war)
  - Erzeugt man einen bereits besuchten Zustand $z' = z$ und z' hat höhere oder gleich hohe Kosten wie z, so kann z' eleminiert werden
  - Bei Kosten ≥ 0 eleminiert man so auch Kreise im Suchraum

Suchstrategien

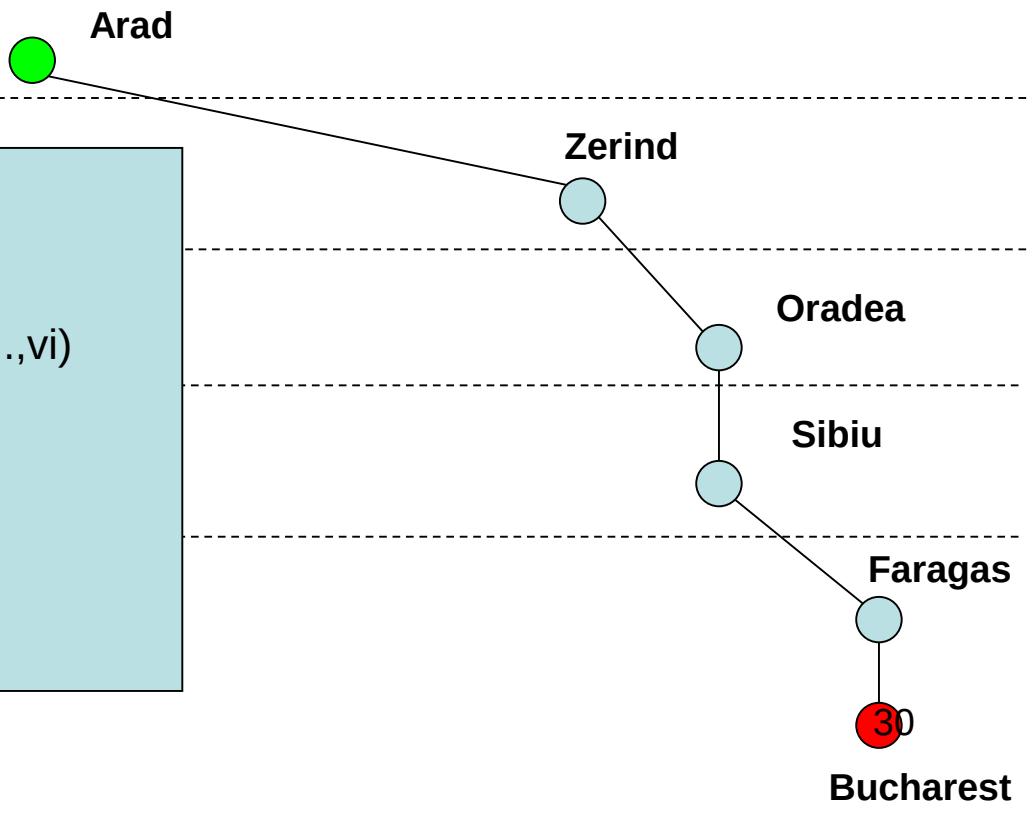
- Grundlegende Methoden
  - Breitensuche
  - Tiefensuche
- Varianten der Tiefensuche
  - Backtracking-Suche
  - Tiefenbegrenzte Suche
  - Iterative Tiefensuche
- Kosten-Sensitive Suche (auch Bestensuche)
  - Uniforme Kostensuche
  - Greedy Suche
  - A* Suche

Backtracking-Suche

Es wird immer nur der expandierte Knoten gespeichert,
Wird keine Lösung gefunden, wird der Suchpfad zurückverfolgt, um neue Randknoten zu finden.

=> Rekursive Implementierung

```
ALGORITHM backtrack([v1,...,vi])
      IF Solution(vi) THEN RETURN (v1,...,vi)
      WHILE hasNextChild(vi)
            v = nextChild(vi)
            r = backtrack([v1,...,vi,v])
            IF (r != ()) RETURN r
      END
RETURN ()
```



Eigenschaften: Backtracking

d = Tiefe der Lösung
m = Max.Tiefe des Suchbaums
b = Branching Faktor
c = Kosten der Lösung
e = Minimale Kosten pro Schritt

- Vollständigkeit:

  NEIN: Wie Tiefensuche

- Optimalität:

  NEIN: Wie Tiefensuche

- Zeitkomplexität:

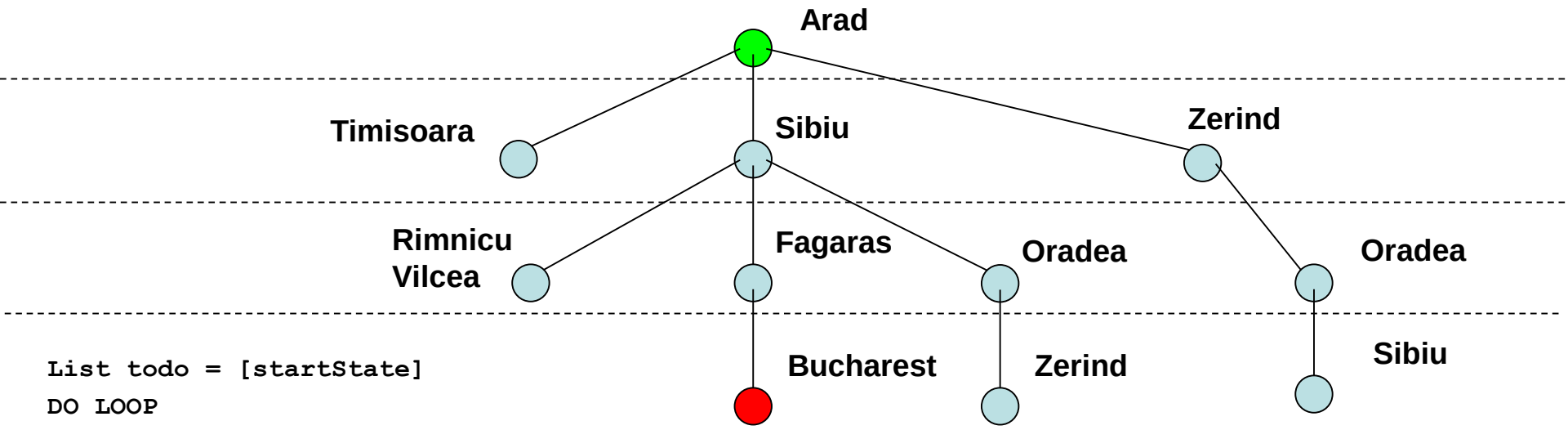
  $O(b^m)$: Wie Tiefensuche

- Speicherkomplexität:

  O(m): Es muss nur ein einziger Knoten sowie die bisherigen Schritte im aktuellen Zweig gespeichert werden

Tiefenbeschränkte Suche

Die maximal Suchtiefe m wird auf ein Limit (l << m) beschränkt, um eher aus aufwendigen Zweigen auszusteigen (Beispiel l = 3)




```
List todo = [startState]
DO LOOP
   IF todo = [] RETURN "Fail"
   ELSE
      State s = selectState(todoList)
      IF isSolution(s) RETURN "Solution found"
      ELSE IF level < maxLevel
         List expandedStates = expand(s)
         add(expandedStates, todo)
```

Eigenschaften: Tiefenbeschränkte Suche

- Vollständigkeit:

  NEIN: Wenn die Lösung ausserhalb der festgelegten Tiefe liegt, wird sie nicht gefunden.

- Optimalität:

  NEIN: Wie Tiefensuche

d = Tiefe der Lösung
m = Max.Tiefe des Suchbaums
b = Branching Faktor
c = Kosten der Lösung
e = Minimale Kosten pro Schritt

- Zeitkomplexität:

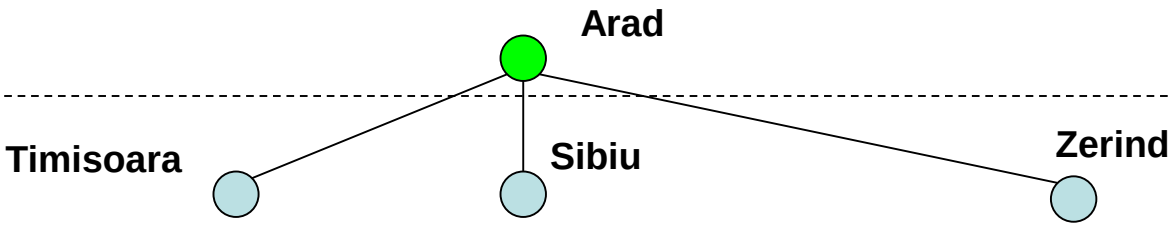
  $O(b^l)$: Wie Tiefensuche, allerdings mit Limit anstelle der maximalen Tiefe

- Speicherkomplexität:

  $O(b*l)$: Wie Tiefensuche, allerdings mit Limit anstelle der maximalen Tiefe

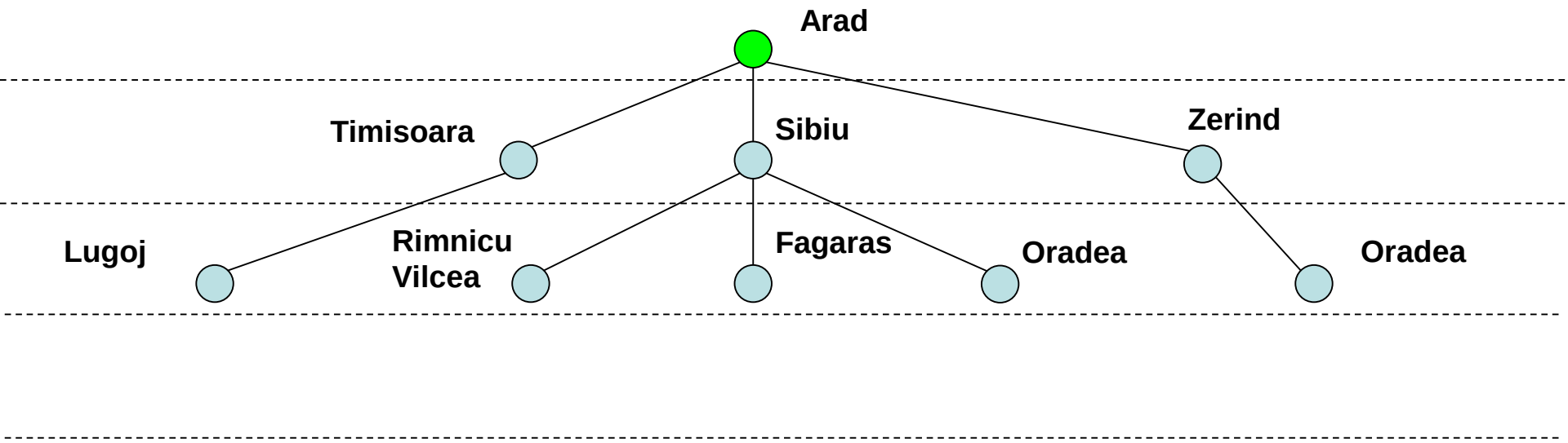
Iterative Tiefensuche

Wie Tiefenbeschränkte Suche. Man beginnt mit Limit 1
Und erhöht das Limit schrittweise um 1, wenn keine Lösung gefunden wurde.



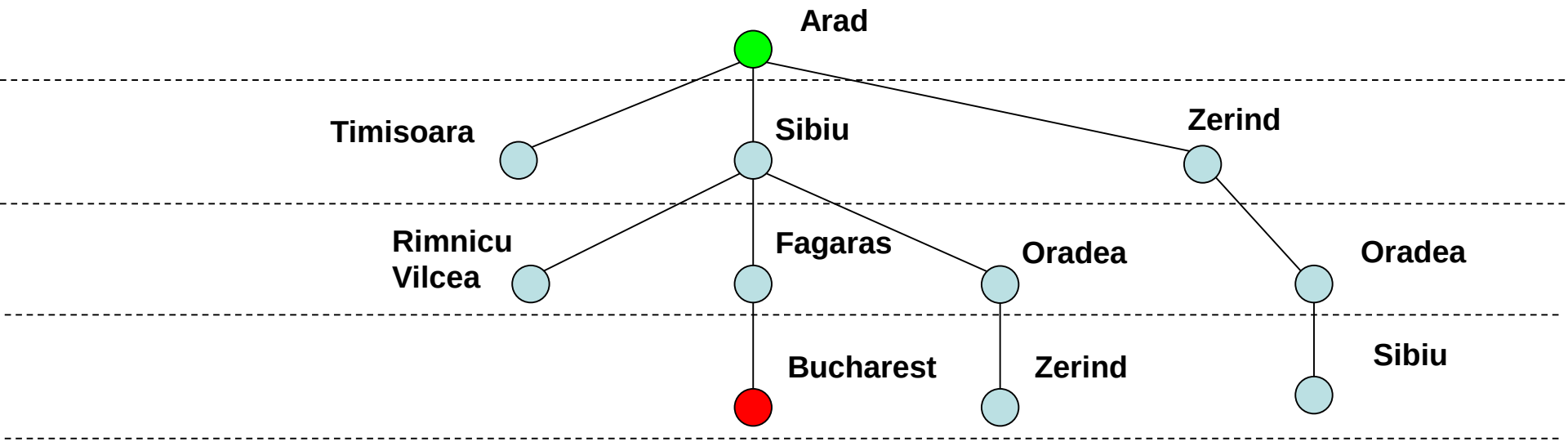
Iterative Tiefensuche

Wie Tiefenbeschränkte Suche. Man beginnt mit Limit 1
Und erhöht das Limit schrittweise um 1, wenn keine Lösung gefunden wurde.



Iterative Tiefensuche

Wie Tiefenbeschränkte Suche. Man beginnt mit Limit 1
Und erhöht das Limit (maxLevel) schrittweise um 1, wenn keine Lösung gefunden wurde.



Eigenschaften: Iterative Tiefensuche

- Vollständigkeit:

  JA: Eine Lösung, die in der Tiefe d liegt, wird gefunden, sobald das Limit l auf d angehoben wurde

- Optimalität:

  JA/NEIN: Wie Breitensuche

d = Tiefe der Lösung
m = Max.Tiefe des Suchbaums
b = Branching Faktor
c = Kosten der Lösung
e = Minimale Kosten pro Schritt

- Zeitkomplexität:

  $O(b^d)$: Alle Knoten der Tiefe n ($n \leq d$) werden d – (n-1) mal expandiert, dies bedeutet, dass in Tiefe n d – (n-1)*b^n Schritte benötigt werden, wobei das grösste n d entspricht

- Speicherkomplexität:

  O(b*d): Wie Tiefensuche, allerdings mit Lösungstiefe anstelle maximaler Tiefe

Eigenschaften: Iterative Tiefensuche

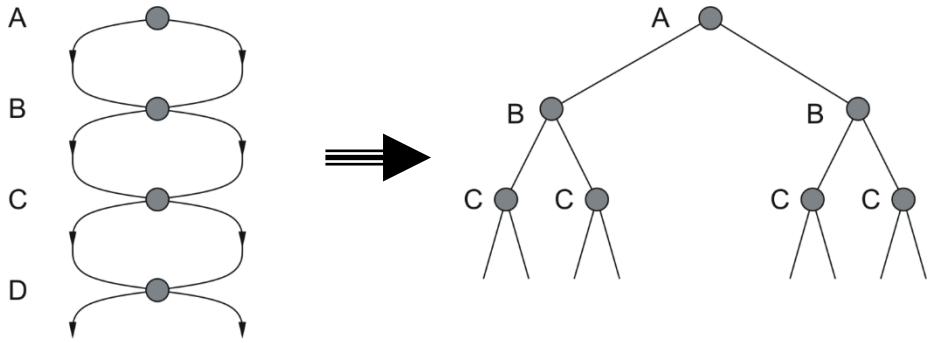
- Verbindet poistive Eigenschaften von Breiten- und Tiefensuche

- Bisher das klar beste Suchverfahren, wenn:
  - Konstante positive Kosten bzw. keine Aussagen über Kosten (= Lösungtiefe entspricht Qualität der Lösung)
  - Keine weitere Informationen über Restkosten verfügbar

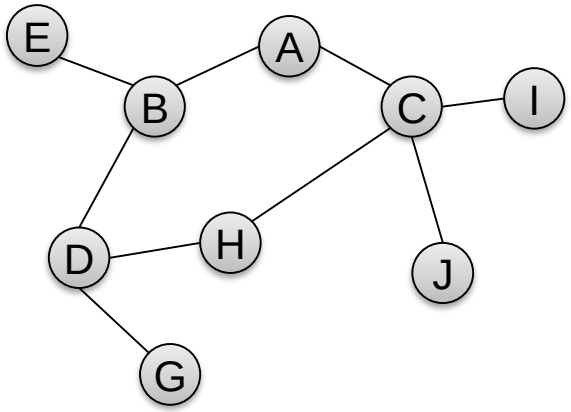
Kreise im Suchraum

- Kreise im Suchraum führen zu einer Explosion des Suchraums

Simples Beispiel



Komplexes Beispiel



Erkennung von Duplikaten

Frage: Wie implementiert man effizient die Erkennung von bereits besuchten Zuständen (Duplikaten) in JAVA.

Wissen aus Algorithmen und Datenstrukturen:

- Welche Datenstruktur kann man verwenden?
  - HashMap
- Welche Methoden muss man hiefür implementieren?
  - int hashCode(), boolean equals()

Fazit

- Backtracking
  - Ist eine Standardmethode zur Verbesserung der Tiefensuche, die mit anderen Varianten kombiniert werden kann
- Tiefenbeschränkte Suche
  - Ist sinnvoll, wenn die Lösungstiefe bekannt ist oder Resourcen knapp sind
- Iterative Tiefensuche
  - Verbindet Vorteile von Breiten- und Tiefensuche
- Die Iterative Tiefensuche ist die Methode der Wahl bei uninformierten Suchmethoden, wenn die Lösungstiefe unbekannt und der Suchraum groß ist.

Ausblick

- In der nächsten Vorlesung betrachten wir Suchverfahren die mit nicht-konstanten Kosten umgehen können
- Dabei wird auch die Abschätzung der zu erwartenden Restkosten relevant werden
  - D.h. das Suchverfahren beinhaltet eventuell eine Komponente um abzuschätzen welche Wege “schneller” zum Ziel führen.
  - Man nennt Verfahren mit einer solchen Komponente informierte Suchverfahren
- Danach wenden wir Suchverfahren auf Spiele an, um mit einfachen Mitteln eine spielstarke KI entwickeln zu können

Backup-Slide