Grundlagen der Künstlichen Intelligenz (2/6)
Grundlgende Suchverfahren

UNIVERSITÄT MANNHEIM
Fakultät für Wirtschaftsinformatik und Wirtschaftsmathematik

Dr. Christian Meilicke, Research Group Data and Web Science

Überblick

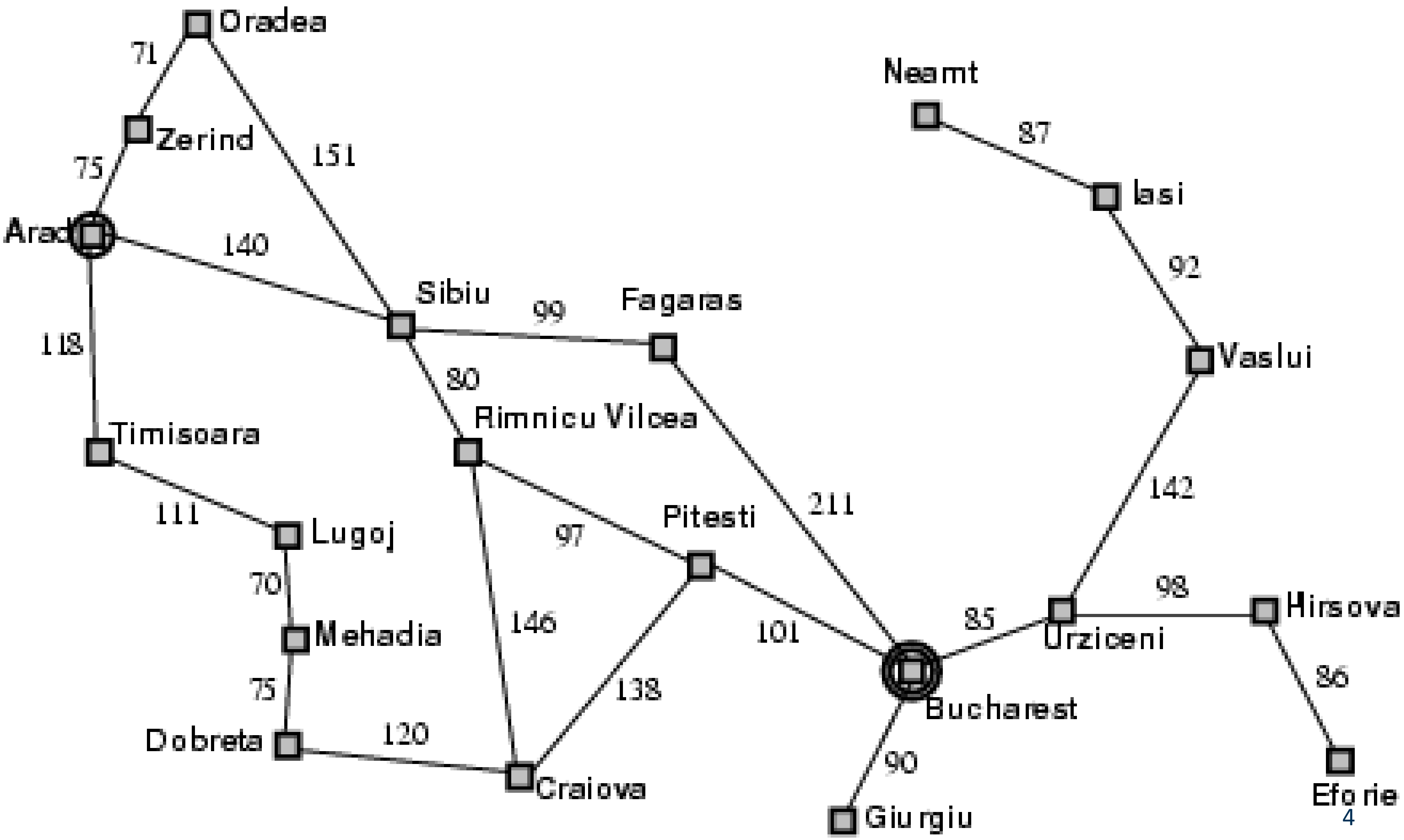
- Definition Suchproblem / Zustandsraum
  - Beispiele
- Qualitätskriterien
  - Vollständigkeit, Optimalität, Zeit und Speicherkomplexität
- Allgemeines Muster eines Algorithmus
- Grundlegende Methoden
  - Breitensuche
  - Tiefensuche
  - Iterative Tiefensuche

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

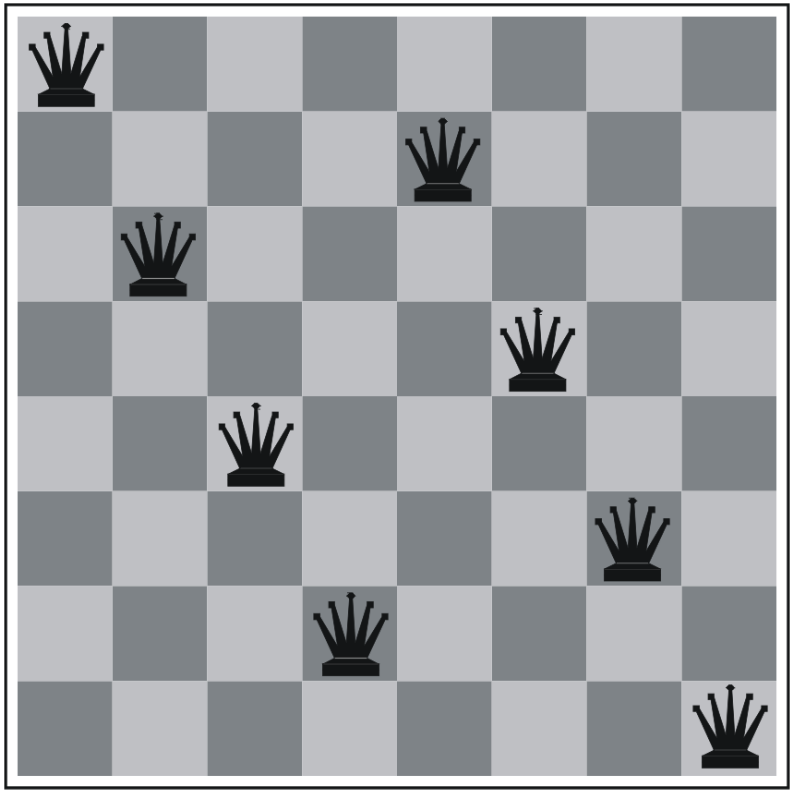
- Lösung eines Suchproblems ist eine Sequenz von Handlungen, die vom Ausgangs- in einen Zielzustand führt

Standard Problem: Routenplanung



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:
  - $x_s = ? \land x_{s-1} \neq ?$
  - $z \neq x_i$ 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

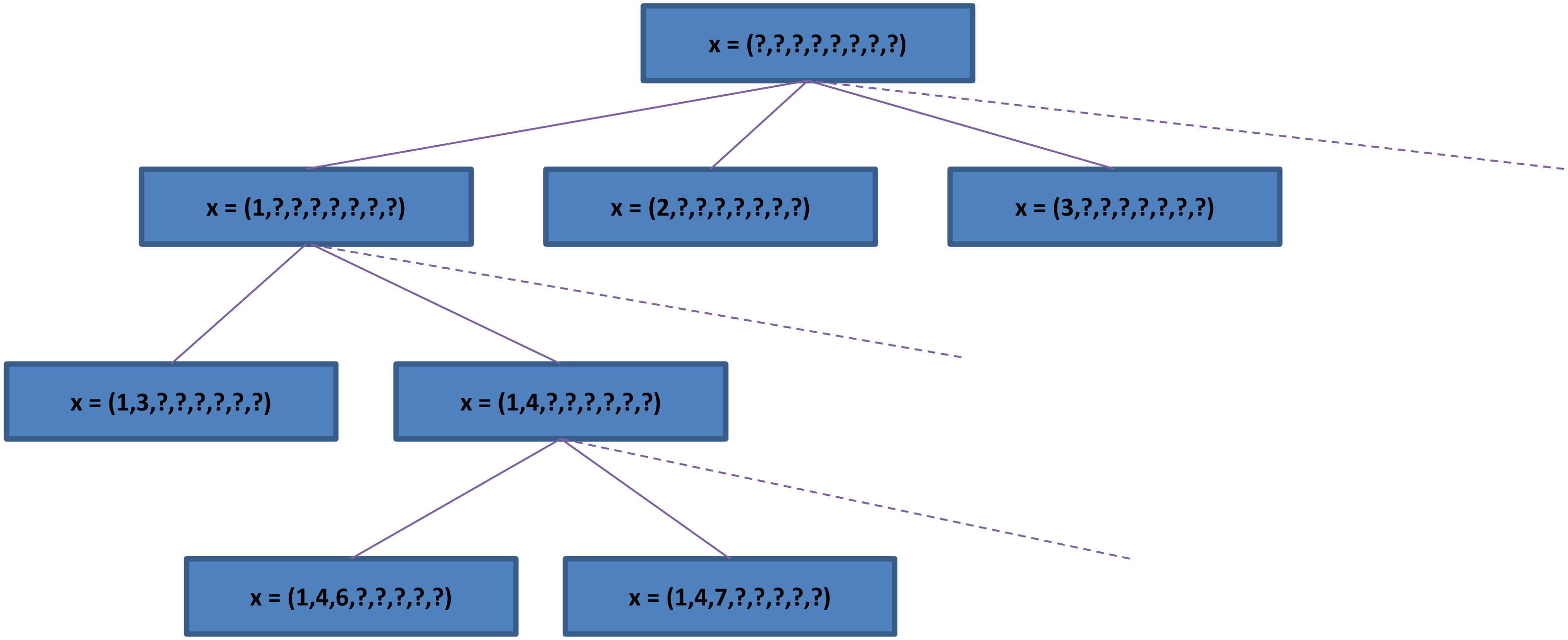
- Wirkung einer Aktion
  - $put\ (s, z) \mapsto\ x_s\ =\ z$
    (unter den auf der vorigen Folie erwähnten Nebenbedingungen)

- 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?

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?

Exkurs: Komplexität

- "Worst-Case" Komplexität:
  - Maximaler Zeit bzw. Speicherverbrauch in Abhängigkeit der Problemgrösse

- Erinnerung: O-Notation
  - O(f(p)): Es werden maximal c + a * f(p) Schritte/Speichereinheiten benötigt, z.B. $O(\lg(p)) < O(p) < O(\lg(p) * p) < O(p^2) < O(2^p)$

- Exponentielles Wachstum = Anzahl der Knoten in einem Suchbaums mit jeweils 10 Verzweigungen, d.h. $O(10^d)$ wenn d die Tiefe des Suchbaums ist

Parameter der Problemgrösse

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

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.

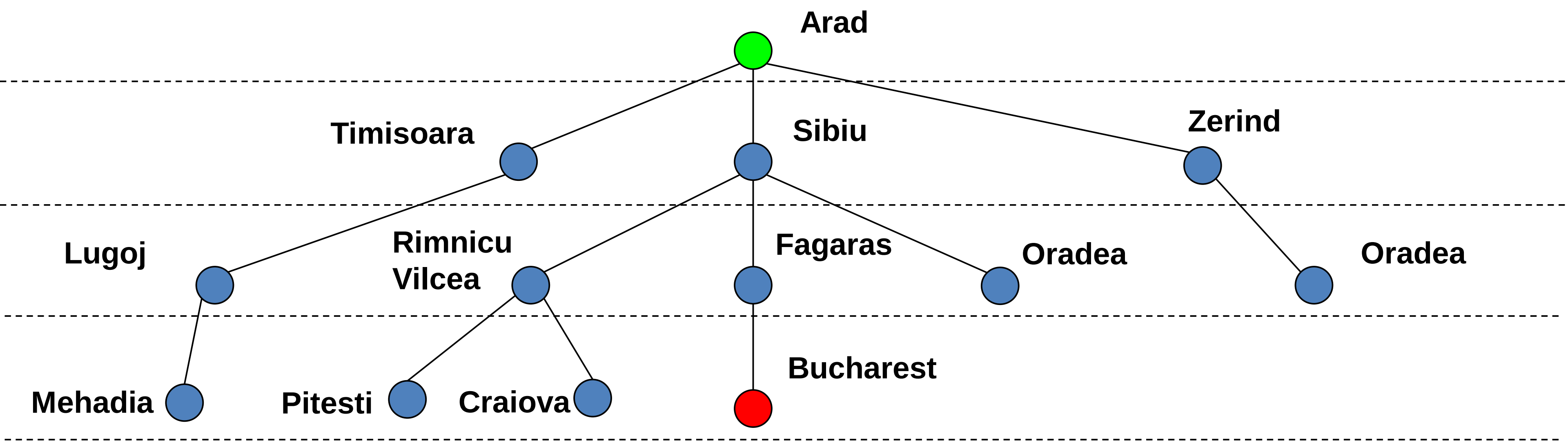
Suchstrategien

- Grundlegende Methoden
  - Breitensuche
  - Tiefensuche
  - Tiefenbeschränkte Suche und Iterative Tiefensuche

- Alle drei Verfahren lassen sich über den allgemeinen Algorithmus verstehen und implementieren

Breitensuche

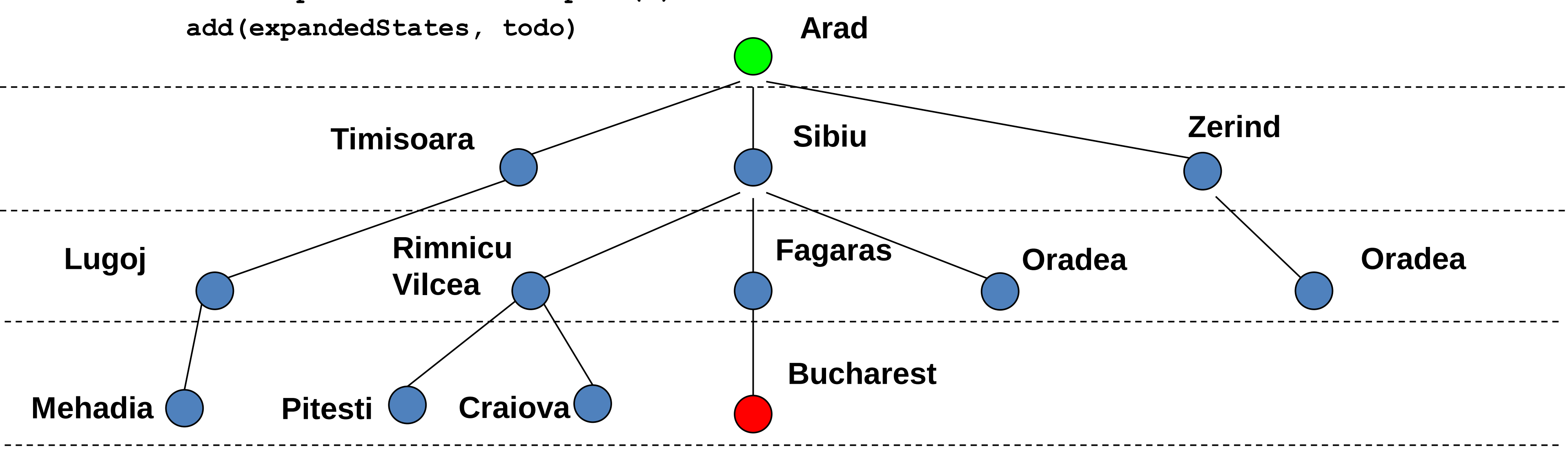
Todo Liste ist eine FIFO-Queue (first in, first out):
Eher eingefügte Knoten werden eher selektiert



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)
```

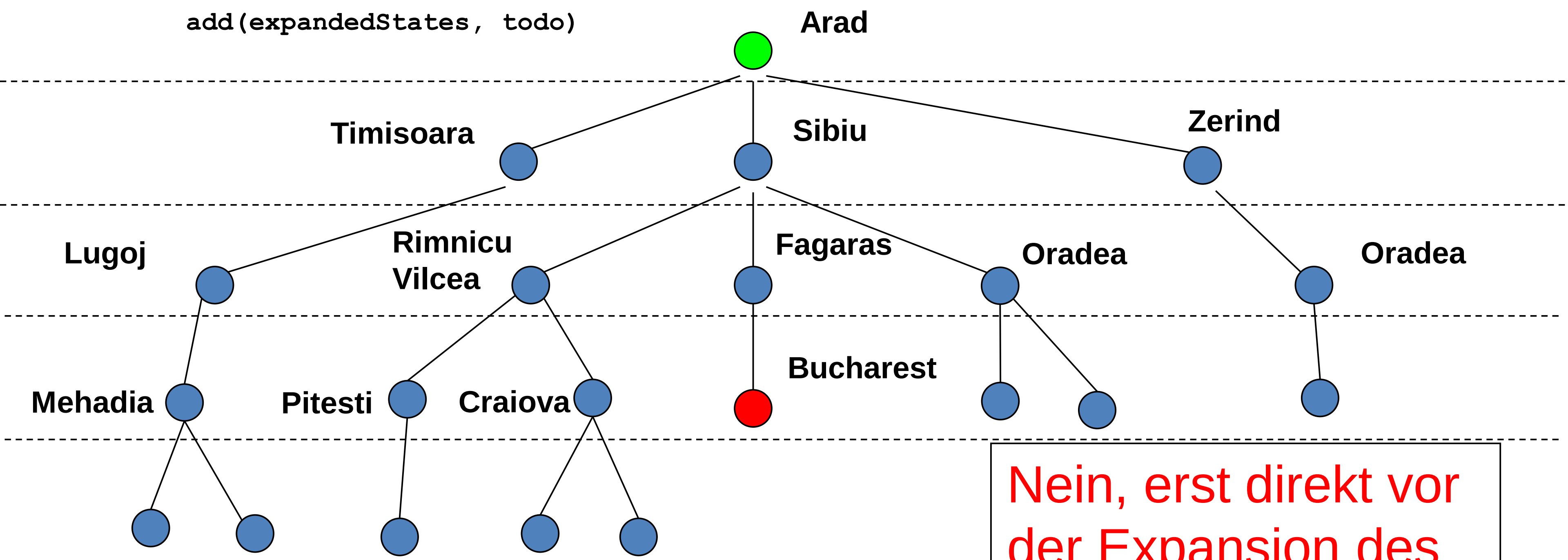
Achtung: Stoppt der Algorithmus wirklich bereits jetzt?



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.

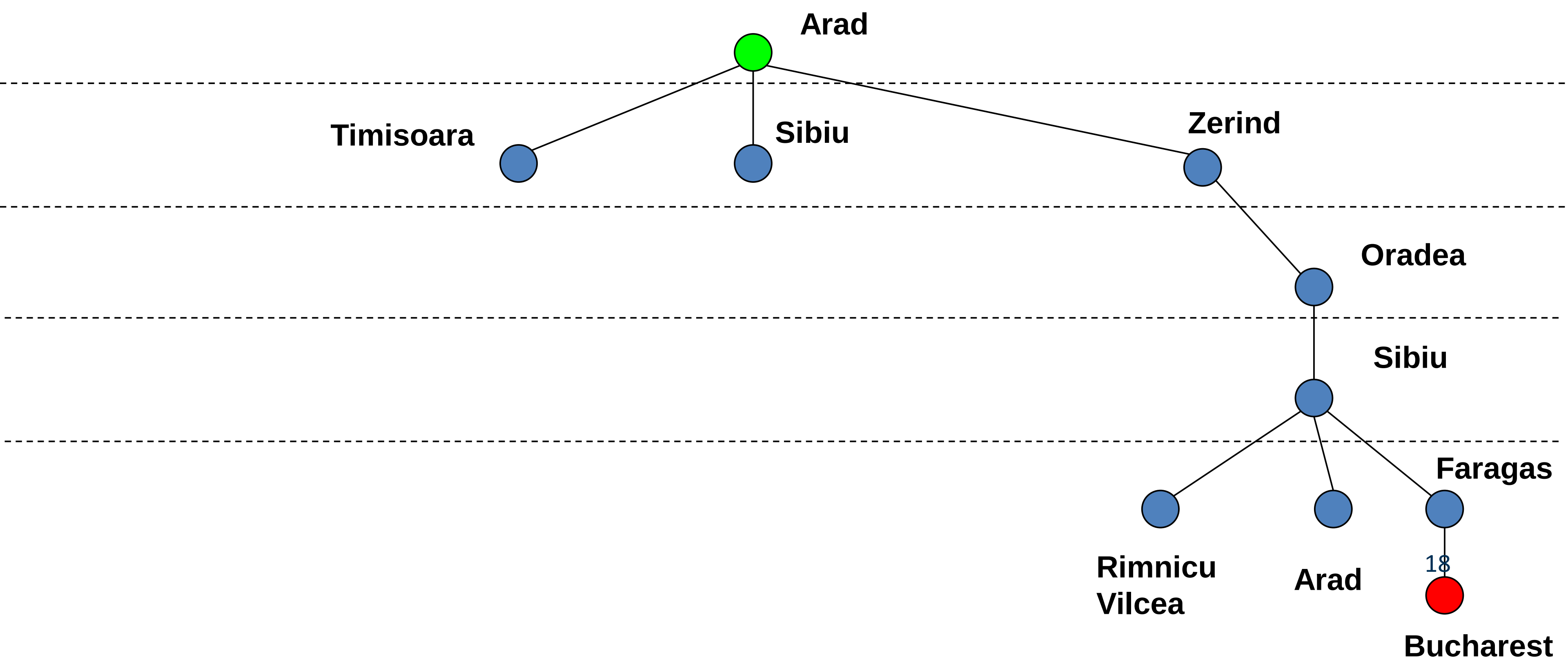
- Speicherkomplexität:

  $O(b^{d+1})$: 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 (last in, first out):
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

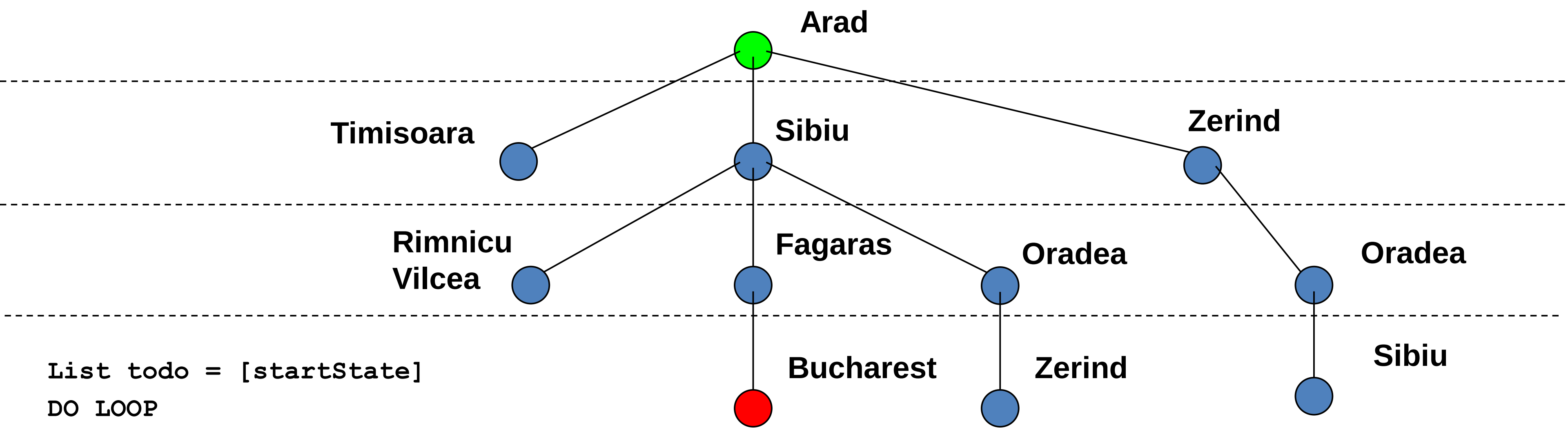
- Breitensuche (FIFO: first in, first out)
  - ist optimal und vollständig
  - Hat aber gravierende Komplexitätsprobleme

- Tiefensuche (LIFO: last in, first out)
  - reduziert den Speicherbedarf drastisch
  - gibt dafür jedoch Vollständigkeit und Optimalität auf

- Können wird die Vorteile von Breitensuche und Tiefensuche verbinden?

Tiefenbeschränkte Suche

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



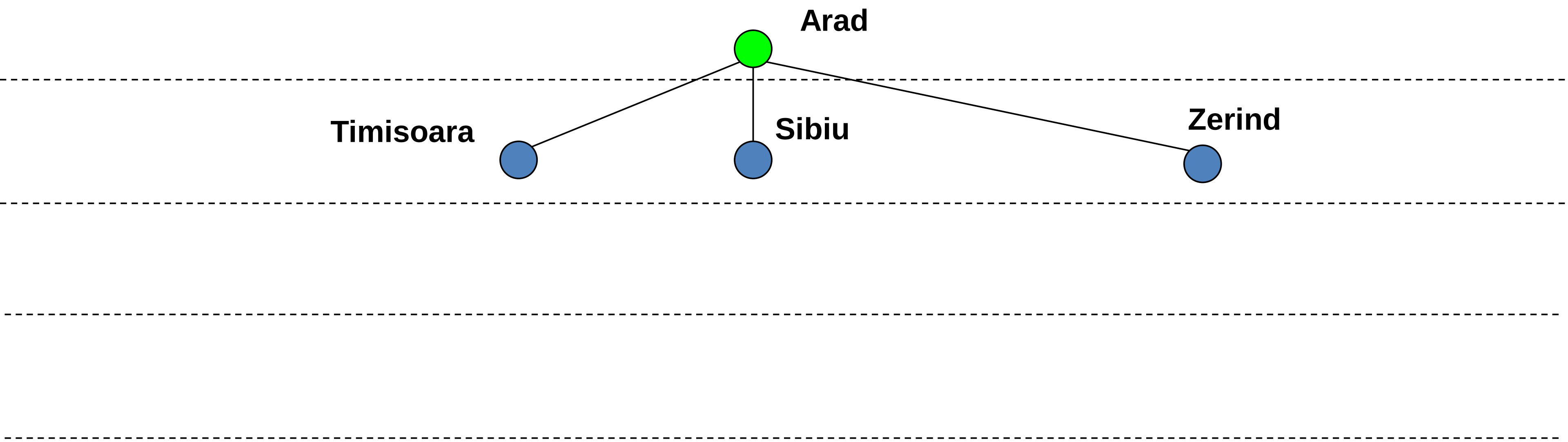

```
List todo = [startState]
DO LOOP
   IF todo = [] RETURN "Fail"
   ELSE
      State s = selectState(todoList)
      IF isSolution(s) RETURN "Solution found"
      ELSE IF level < maxLevel
         List expandedStates = expand(s)
         add(expandedStates, todo)
```

Tiefenbeschränkte Suche

Iterative Tiefensuche

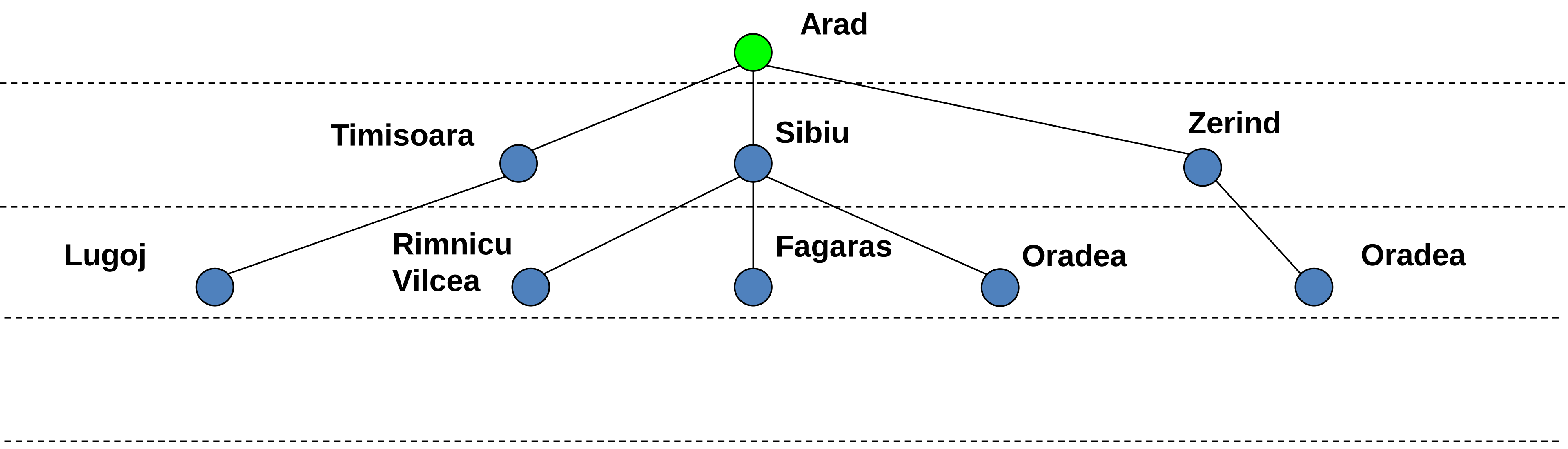
Iterative Tiefensuche

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



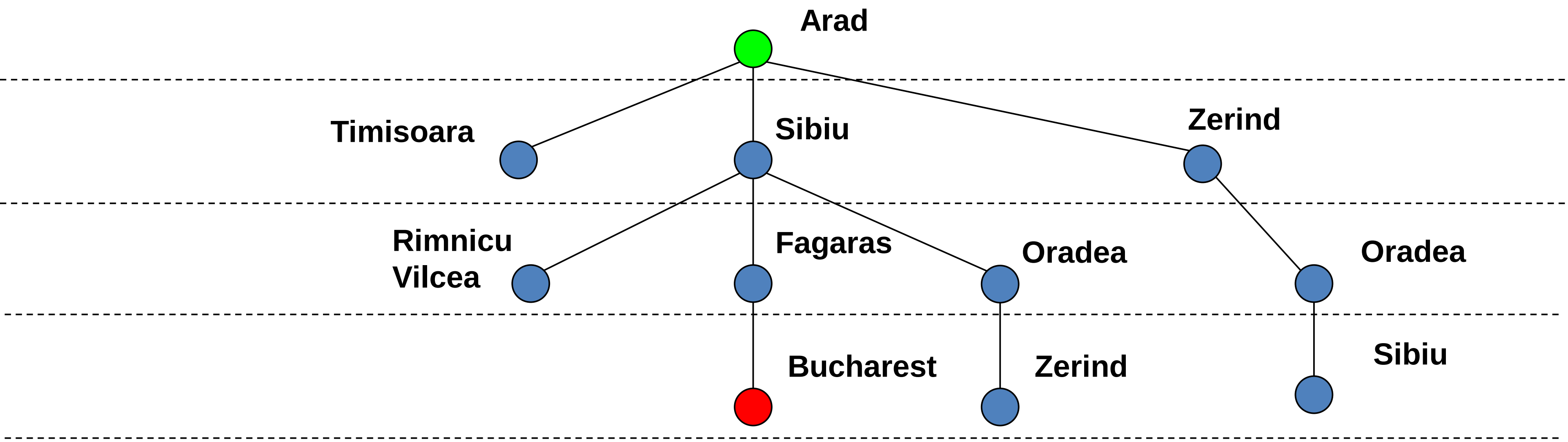
Iterative Tiefensuche

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 positive 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
- Aber: Einfache vollständige Suchverfahren werden immer Komplexitätprobleme bekommen (in der Laufzeit) bei
  - Hohem Branchingfaktor und/oder
  - Großer Suchtiefe

Ausblick

- Nach der Pause 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 (am Donnnerstag) wenden wir Suchverfahren auf Spiele an, um mit einfachen Mitteln eine spielstarke KI entwickeln zu können