Prof. Dr. Gerhard Weiss
Department of Knowledge Engineering
Maastricht University

Einführung in die Künstliche Intelligenz

— Übungsblatt 1 —

Übung 1-1(Formalisierung von Suchproblemen)

Allgemein kann ein Suchproblem formalisiert werden, indem folgende vier Komponenten angegeben werden:

(a) Der Startzustand der Problems (oder alternativ die Menge aller erlaubten Zustände des Problemraums).

(b) Die Nachfolgerfunktion, d.h. die Menge aller erlaubten Operatoren, die einen Übergang von einem Zustand in einen anderen Zustand bewirken.

(c) Test auf Erreichung des Zielzustandes, d.h. Prüfung, ob ein erreichter Zustand mit dem Zielzustand identisch ist.

(d) Pfadkostenfunktion, d.h. Angabe dazu, wie "teuer" ein erreichter Zustand sein soll bzw. wie sich die Kosten eines Zustandes errechnen sollen.

1-1-1. Formalisieren Sie das bekannte Suchproblem "8-Puzzle-Problem" (Abbildung 1-1), d.h. geben Sie zu diesem Problem die oben genannten vier Komponenten an. Bei diesem Problem geht es darum, 8 Ziffern von ihrer Startkonfiguration in eine Zielkonfiguration zu verschieben; gehen Sie davon aus, dass jedes Verschieben jeder Ziffer die gleichen Kosten verursacht.

1-1-2. Formalisieren Sie das bekannte Suchproblem "n-Türme-von-Hanoi-Problem" (Abbildung 1-2 illustriert dieses Problem für $n = 5$), d.h. geben Sie auch hierzu die oben genannten vier Komponenten an. Bei diesem Problem geht es darum, n Scheiben unterschiedlicher Größe von einer Ausgangsposition A auf eine Zielposition B zu bewegen, wobei die nur die Position C als "Zwischenlager" benutzt und niemals eine größere Scheibe auf eine kleinere gelegt werden darf.

Übung 1-2 (Breiten/Tiefensuche)

Implementieren Sie die einfachen Lösungsstrategien *Breadth-first* (d.h. Breitensuche) und *Depth-first* (Tiefensuche) und wenden Sie Ihre Breadth- und Depth-first Implementierungen auf die beiden Probleme "8-Puzzle" und "n-Türme-von-Hanoi" mit $n = 5$ an. Dabei soll insbesondere bei Depth-first-Suche eine maximale Suchtiefe vorgegeben werden.

Mit Hilfe Ihrer Implementierungen und der von Ihnen gewählten Instantiierungen sollen folgende Aufgabenstellungen bearbeitet werden:

(a) Begründen Sie Ihre Wahl der maximalen Suchtiefe bei Depth-first.

(b) Listen Sie für Ihre Breadth- und Depth-first Implementierungen die gewonnenen Lösungssequenzen übersichtlich auf. Vergleichen Sie möglichst genau Ihre beiden Implementierungen bezüglich deren Performanz (Laufzeit, Anzahl generierte Knoten, maximaler Platzbedarf, Anzahl der gefundenen Lösungen).

(c) Wie lautet der jeweils kürzeste gefundene Lösungspfad?

(d) Überlegen Sie sich "Strategien" zur Vermeidung von Mehrfach-Expansion desselben Knotens (an unterschiedlichen Positionen im Suchbaum). Eine solche "Vermeidungsstrategie" soll also verhindern, dass im Suchbaum zu einem Knoten mehrmals die Nachfolgerknoten generiert werden.

(e) Wieviele unnötige "Mehrfachexpansionen" von Knoten verhindert die von Ihnen vorgeschlagene "Vermeidungsstrategie"? Wieviel Mehraufwand – Zeit und Speicherplatz – verursacht diese "Vermeidungsstrategie"? Sie können (e) durch "reines Nachdenken" oder durch Gegenüberstellung von Testergebnissen für "Breadth/Depth-first mit versus ohne Vermeidungsstrategie" beantworten.

Zur Erinnerung: Bei *Breadth-first* werden zuächst alle Knoten der Ebene 1 generiert, dann alle Knoten der Ebene 2, usw. Bei *Depth-first* wird immer ein Knoten auf der jeweils untersten Ebene expandiert (falls mehrere Knoten auf der untersten Ebene sind dann zufällig auswählen oder eine Reihenfolge auf den Operatoren festlegen).

Übung 1-3 (Bidirektionale Suche)

Geben Sie einen Pseudocode für die bidirektionale Suche an. Nehmen Sie dabei an, dass sowohl vorwärts als auch rückwärts nach der Breadth-first search gesucht wird, wobei abwechselnd eine "Vorwärtsknoten" und eine "Rückwärtsknoten" expandiert wird (d.h. Vorwärts- und Rückwärtssuche sind verschränkt). Achten Sie darauf, dass nur soviele Vorwärts- und Rückwärtsknoten wie nötig verglichen werden. Sofern Sie keine bessere Idee haben, können Sie auch das Pseudocode-Fragment aus Abbildung 1-3 als Ausgangspunkt verwenden.

Hinweis: Bei *bidirektionaler Suche* wird in beide Richtungen gesucht, d.h. vom Startzustand zum Zielzustand ("Vorwärtssuche") und umgekehrt ("Rückwärtssuche"); der "Trick" besteht darin, die Knoten, die in beide Suchrichtungen generiert werden, geeignet abzugleichen.

Übung 1-4 (Uniform-Cost Strategie)

Neben Breiten- und Tiefensuche ist die "Suche abhängig von tatsächlichen Kosten" (engl. uniform-cost search oder lowest-cost-first search) eine häufig verwendete uninformierte Suchstrategie. Bei dieser Strategie werden mit jedem Knoten n die Kosten $g(n)$,

$$g(n) = \text{"tatsächliche Kosten von Startknoten zu Knoten } n\text{"} \quad ,$$

assoziiert und es wird jeweils der Knoten expandiert, der bislang die wenigsten Kosten verursacht hat. Vorausgesetzt wird dabei, dass die Pfadkosten monoton wachsen; es gilt also

$$g(SUCCESSOR(n)) \geq g(n) \quad \forall \;\; n \quad .$$

Wie einfach zu sehen ist entspricht eine uniform-cost search mit $g(n) = \text{Tiefe}(n)$ gerade der Breitensuche.

1-4-1. Welches Problem ergibt sich bei dieser Strategie (ebenso wie bei anderen kostenorientierten Strategien), falls Kanten im Suchbaum mit negativen Kosten – und damit sozusagen mit "Gewinn" oder "Belohnung" – bewertet sein können?

1-4-2. Wird dieses Problem gelöst, falls eine negative untere Schranke c angegeben werden kann, die garantiert von keiner Kantenbewertung unterschritten wird?

1-4-3. Welche Rolle spielt die Vermeidung von Mehrfachexpansionen von Knoten bei uniform-cost search?

1-4-4. Implementieren Sie die uniform-cost Strategie und wenden Sie diese Strategie auf das "Landkartenproblem" (Abbildung 1-4) an. Wie Sie bereits von Übung 1 wissen geht es bei diesem Problem darum, den kürzesten Weg von Arad nach Bucharest zu finden. Verwenden Sie bei diesem Problem die tatsächlichen Entfernungen für die Kostenfunktion g. Wie lauten die gefundenen Lösungen? Wieviele Knoten umfassen die Suchbäume?

Bearbeitungen in PDF- *und* Source-Format (World, LaTeX, etc.)
bis zur nächsten Vorlesung an: gerhard.weiss@maastrichtuniversity.nl

---

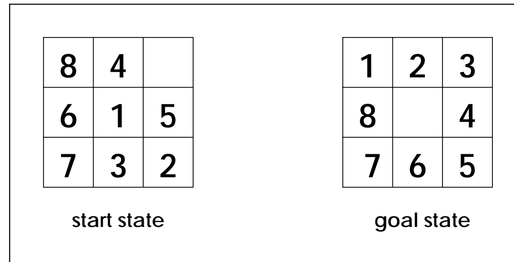

Abbildung 1-1: 8-Puzzle.

Abbildung 1-2: Towers of Hanoi.

```
function BIDIRECTIONAL SEARCH (in: Problem, out: Knoten)
    forward-queue =  Queue mit Anfangszustand des Vorwaertsproblems
    backward-queue = Queue mit Anfangszustand des Rueckwaertsproblems
    [hier ggf. weitere Datenstrukturen usw definieren]
    loop until both queues are empty do
         [innerhalb der Schleife auf Loesung pruefen und
          die Schleife ggf. vorzeitig abbrechen]
         Knoten = remove-front(forward-queue)
         expand(Knoten) entsprechend der Nachfolgeroperatoren
         add(Nachfolger von Knoten,forward-queue)
         ...
end function
```

Abbildung 1-3: Pseudocode-Fragment für bidirektionale Suche.

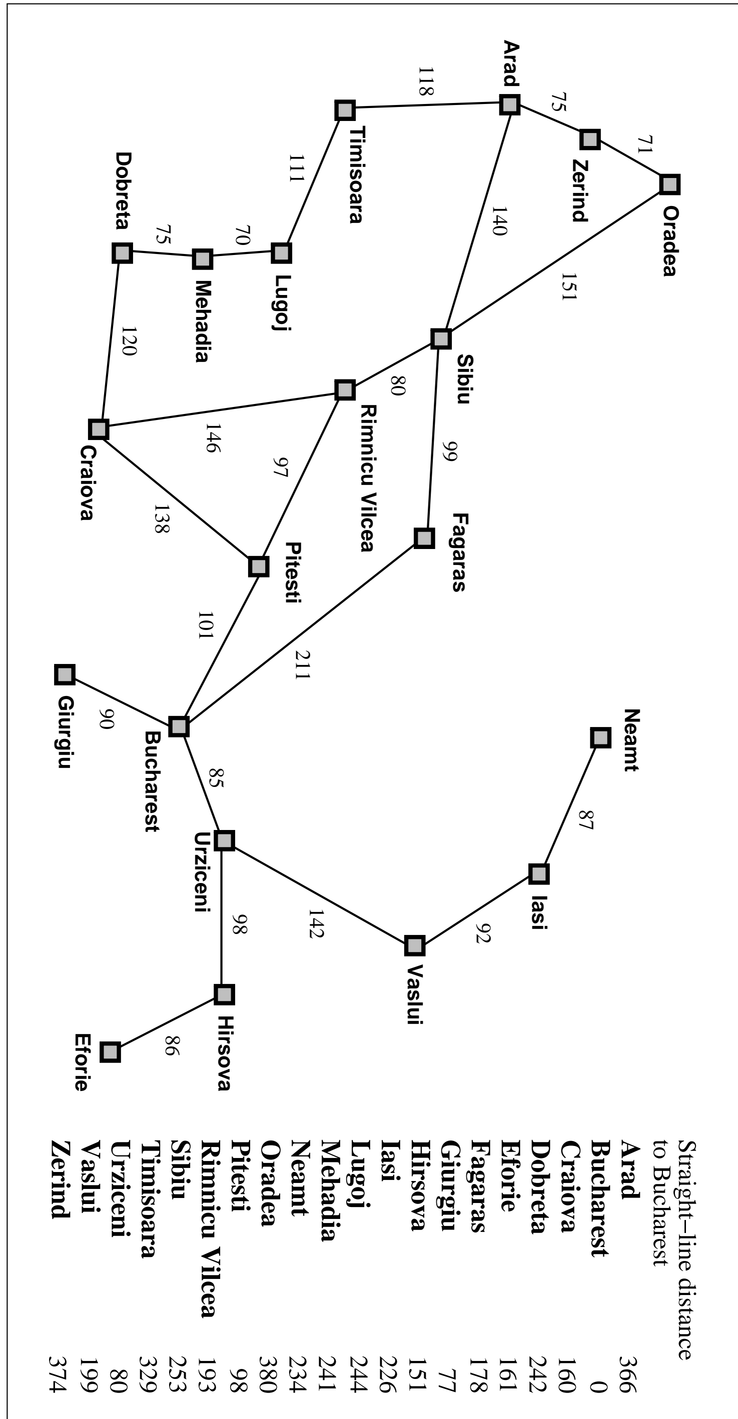

ABBILDUNG 1-4: Landkarte von Rumänien (aus [Russell & Norvig]).