Grundlagen der Künstlichen Intelligenz (CS 205)

Prof. Dr. M. Helmert
G. Röger
Frühjahrssemester 2013

Universität Basel
Fachbereich Informatik

Übungsblatt 1

Abgabe: 18. März 2013

Bei diesem Übungsblatt handelt es sich gleich einmal um eine Programmieraufgabe. Sie dürfen Programmieraufgaben in den Sprachen C, C++, Java oder Python sowie weiteren Programmiersprachen nach Absprache bearbeiten.
Wir erwarten, dass Sie Ihre Implementierung selbständig, das heisst ohne Anwendung von fremdem Code z.B. aus dem Internet erstellen. Die Standardbibliothek Ihrer gewählten Programmiersprache dürfen Sie selbstverständlich ohne Einschränkung verwenden.
Bei technischen Schwierigkeiten und Verständnisproblemen helfen wir gerne weiter. Bitte wenden Sie sich dazu *mit genügend zeitlichem Abstand zum Abgabetermin* an Gabi Röger oder Lukas Beck.

Aufgabe 1.1 (5+1+3+1 Punkte)
(Auf Seite 2 dieses Übungsblatts finden Sie ein paar Tipps zur Lösung dieser Aufgabe.)

(a) Implementieren Sie *Breitensuche* zur Lösung des Blocksworld-Problems (siehe Beispiel 1 auf Folien zu klassischen Suchproblemen).

Auf der Vorlesungsseite finden Sie Beispieleingabedateien, die verschiedene Blocksworldinstanzen kodieren. Es gibt dort auch ein Beispiel für eine mögliche Lösung. Das Eingabeformat ist darauf ausgelegt, leicht zu parsen zu sein. Das genaue Ein- und Ausgabeformat ist in der README-Datei spezifiziert.

(b) Testen Sie Ihre Implementierung auf den Beispiel-Probleminstanzen von der Webseite. Setzen Sie sich hierzu ein Zeitlimit von 10 Minuten und ein Speicherlimit von 2 GB. (Wenn Ihr Computer über 2 GB oder weniger RAM verfügt, setzen Sie sich ein Speicherlimit von "physisches RAM minus 256 MB".) Zeit- und Speicherlimit können Sie sich z.B. unter Linux mit `ulimit -t 600` bzw. `ulimit -v 2097152` für die aktuelle Shell-Sitzung setzen. Welche Probleminstanzen können Sie innerhalb dieser Schranken lösen? Berichten Sie die Laufzeit der drei gelösten Instanzen, die am längsten gebraucht haben. Berichten Sie weiterhin, ob die nichtgelösten Instanzen am Zeit- oder am Speicherlimit scheiterten. Falls am Speicherlimit, wie lange sind sie zuvor gelaufen?

(c) Betrachten Sie nun eine modifizierte Blocksworldversion, bei der alle Aktionen, die einen Block direkt von einem Block zu einem andern bewegen (also ohne den Tisch zu involvieren), Kosten 3 haben. Alle anderen Aktionen haben weiterhin Kosten 1.

Implementieren Sie *uniforme Kostensuche* zur Lösung des modifizierten Blocksworldproblems. Wie in der Vorlesung beschrieben, muss die Open-Liste des Suchalgorithmus hierzu eine Prioritätswarteschlange implementieren (in C++ z.B. `std::priority_queue`).

(d) Geben Sie eine möglichst kleine Beispielinstanz (im gleichen Eingabeformat wie bei (a)) an, die von der Breitensuche suboptimal, von der uniformen Kostensuche jedoch optimal gelöst wird.

Reichen Sie bitte den Code inklusive Hinweisen zu Kompilierung und Aufruf bei *courses* ein (sofern nicht selbsterklärend).

Die Übungsblätter dürfen in Gruppen von zwei Studierenden bearbeitet werden. Bitte schreiben Sie beide Namen auf Ihre Lösung.

Tipps und Hinweise

Falls Sie Schwierigkeiten mit der Aufgabenstellung haben, hier ein paar Tipps. Beachten Sie aber, dass man die Aufgabe auch anders lösen kann, d.h. Sie können auch eine ganz andere Implementierung wählen (solange es sich um eine Breitensuche bzw. uniforme Kostensuche handelt).

- Wählen Sie eine geeignete Repräsentation der Zustände. Eine Möglichkeit ist ein int-Array, das an der i-ten Position Eintrag j enthält, wenn der i-te Block auf dem j-ten Block liegt und -1, falls der i-te Block auf dem Tisch liegt.
- Um die anwendbaren Aktionen schnell bestimmen zu können, kann es sinnvoll sein, sich in jedem Zustand auch noch die Menge der freien Blöcke zu merken, also der Blöcke, auf denen kein anderer Block liegt.
- Eine Implementierung, die relativ nah an den Algorithmen der Vorlesung liegt, erhält man, wenn man z.B. eine Klasse `BlocksworldInstance` schreibt, die im Konstruktor die Eingabedatei liest und ansonsten das Blackbox-Interface von Folie 21 aus Kapitel 3 implementiert.
- Wählen Sie geeignete Datenstrukturen. Zum Beispiel wird die Openlist-Funktionalität bei der Breitensuche sehr gut durch eine verkettete Liste oder ein Deque implementiert. Andererseits testen wir zur Duplikatelimination auch, ob ein Zustand in der Openlist ist. Wie teuer ist das bei den genannten Datenstrukturen? Können Sie die Duplikatelimination auch geschickter machen? (Dies hat nur Auswirkungen auf die Effizienz Ihrer Implementierung.)
- Beachten Sie, dass wir für jeden *Zustand* höchstens einen *Suchknoten* im Speicher halten wollen. Evtl. benötigen Sie für die gewünschte Funktionalität geeignete Implementierungen der Vergleichs- und Hashmethoden in der Zustands- und Knotenklasse.