Shakey Projekt in Prolog

Einleitung

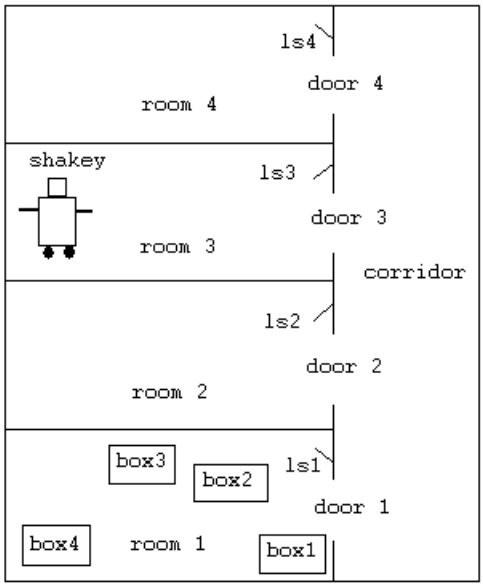
Was ist Shakey?

Shakey war der erste mobile Roboter, der seine eigenen Aktionen planen konnte. Seine Entwicklung kombinierte Robotik, Bildverarbeitung und Natural language processing.

Er wurde im Labor für künstliche Intelligenz des Stanford Research Institutes im Zeitraum 1966 bis 1972 unter der Leitung von Charles Rosen entwickelt. Die Programmierung erfolgte größtenteils in LISP.

Quelle: https://de.wikipedia.org/wiki/Shakey_(Roboter)

Shakeys Umgebung



Quelle: http://www.idi.ntnu.no/emner/tdt4136/EXERCISES/shakey.gif

Diese Skizze stellt die Umgebung von Shakey dar. Sie ist in sich abgeschlossen und besteht aus mehreren Räumen, welche durch einen Korridor miteinander verbunden sind. Zwischen Raum und Korridor befindet sich jeweils eine Tür, welche offen oder geschlossen sein kann. Dazu können weitere Objekte (beispielsweise Boxen) in den Räumen verteilt sein.

Shakey ist in der Lage, mit seiner Umgebung zu interagieren, also beispielsweise von Raum 1 in Raum 4 zu gehen oder eine Box von einem Raum in den anderen zu verschieben usw…

Ziel

Ziel dieses Projektes ist es, die Welt von Shakey in der logischen Programmiersprache Prolog abzubilden und Aktionen zu definieren, welche es erlauben, diese Welt zu verändern.

Es muss möglich sein, dass man nur einen Startzustand und einen Zielzustand kennt und daraus einen möglichst effizienten Weg vom Startzustand zum Zielzustand findet und diesen ausgibt.

Als Beispiel folgender Start- und Zielzustand (Pseudocode):

- Startzustand → [room(room1), room(room2), connected(room1, room2), in(shakey, room1)]
- Zielzustand → [room(room1), room(room2), connected(room1, room2), in(shakey, room2)]

Somit sind zwei verbundene Räume gegeben und Shakey soll nun von Raum 1 nach Raum 2 gehen.

Umsetzung

Ausführung mit fixen Zuständen

Die Einstiegsklausel ist run/2 mit der Start- und Zielzustandsliste als Argumente.

Da Shakey ein Roboter ist, welcher sich hauptsächlich aufs Verschieben im Raum und aufs Bewegen von Boxen konzentriert, wird davon ausgegangen, dass es für Shakey unmöglich ist, die gegebene Raumanordnung in irgendeiner Weise zu verändern. Somit gibt es also Fakten, welche in jedem Start- und Zielzustand identisch sind. Damit diese nicht jedes Mal vom Aufrufer der run/2 Klausel mitgegeben werden müssen, sind die im Programm fix definiert und werden in der run/2 Klausel hinzugeladen. Folgende fixe Zustände sind gegeben:

- room(room1)
- room(room2)
- room(room3)
- room(corridor)
- door(door1)
- door(door2)
- door(door3)
- connected(room1, corridor)
- connected(room2, corridor)
- connected(room3, corridor)
- connected(room1, door1)
- connected(room2, door2)
- connected(room3, door3)
- connected(corridor, door1)
- connected(corridor, door2)
- connected(corridor, door3)

STRIPS-Actions

Damit Shakey mit seiner Umgebung interagieren kann, benötigt es entsprechende Aktionen. Eine STRIPS-Aktion hat folgende Eigenschaften:

- Name (mit Argumenten)
- Liste aller Zustände, die gegeben sein müssen, damit die Aktion ausgeführt werden kann (Preconditions)
- Liste aller Zustände, welche nach der Ausführung der Aktion wahr sind (AddList)
- Liste aller Zustände, welche nach der Ausführung der Aktion nicht mehr wahr sind (DeleteList)

Folgende Aktionen wurden implementiert:

Name	**Preconditions**	**AddList**	**DeleteList**
move(From, To)	room(From), room(To), in(shakey, From), connected(To, From), connected(From, Door), connected(To, Door), status(Door, open)	in(shakey, To)	in(shakey, From)
openDoor(DoorName, RoomName)	door(DoorName), room(RoomName), connected(RoomName,	status(DoorName, open)	status(DoorName, closed)

	DoorName), status(DoorName, closed), in(shakey, RoomName)		
closeDoor(DoorName, RoomName)	door(DoorName), room(RoomName), connected(RoomName, DoorName), status(DoorName, open), in(shakey, RoomName)	status(DoorName, closed)	status(DoorName, open)
grab(ObjectName, RoomName)	object(ObjectName), room(RoomName), in(ObjectName, RoomName), in(shakey, RoomName), in(nothing, shakeysHand)	in(ObjectName, shakeysHand)	in(ObjectName, RoomName), in(nothing, shakeysHand)
put(ObjectName, RoomName)	object(ObjectName), room(RoomName), in(ObjectName, shakeysHand), in(shakey, RoomName)	in(ObjectName, RoomName), in(nothing, shakeysHand)	in(ObjectName, shakeysHand)

Aus den implementierten Aktionen ergeben sich auch die möglichen Zustände, welche vom Aufrufer von run/2 mitgegeben werden können oder müssen (Je nachdem, welche Ziele verfolgt werden):

- - in(RobotName / ObjectName, RoomName)
- - status(DoorName, StatusName)
- - object(ObjectName)

Für den Aufrufer gibt es die Zusammenfassung der fixen und variablen Zustände als Konsolenausgabe mit shakey_help/0.

STRIPS-Planer

Nachdem in run/2 die fixen Zustände nachgeladen wurden, wird direkt solve/4 aufgerufen. Solve/4 ist das Herz des Programms. Der rekursive STRIPS-Planer sucht sich so lange die nächste Aktion aus, welche den momentanen Zustand verändert bis der Zielzustand gefunden wurde. Solve/4 entspricht einer Vorwärts-Tiefensuche. Folgende Schritte werden rekursiv durchgeführt:

1. Aus der Liste der Aktionen die nächste Aktion laden
2. Überprüfen, ob alle Preconditions dieser Aktion auf den aktuellen Zustand zutreffen
3. Überprüfen, ob genau diese Aktion (mit Parameter) schon mal durchgeführt worden war (Vergleich mit ActionList), um unnötige Schleifen auszuschliessen
4. DeleteList auf den aktuellen Zustand anwenden
5. AddList auf den aktuellen Zustand anwenden
6. Überprüfen, ob genau dieser Zustand schon mal entstanden ist, um unnötige Aktionen auszuschliessen
7. Rekursiver Aufruf mit aktualisiertem Zustand

Dazu wird vor jedem Aufruf überprüft, ob der aktuelle Zustand dem Zielzustand entspricht, was das erfolgreiche Beenden des Programms zur Folge hat (Ausgabe der gefundenen Aktionen in der Konsole).

Diese Lösung findet nicht immer den schnellsten Weg. Dazu müsste man den STRIPS-Planer noch so erweitern, dass er nach einer gefundenen Lösung immer die kürzeste behält und danach weitersucht, ob es noch kürzere gibt. Eine solche Lösung wäre aber dann viel Rechenintensiver als die Jetzige.

Hürden

Reihenfolge der Aktionen

Da die implementierte Tiefensuche der Reihe nach, jede mögliche Aktion durchforstet und bei einer gefundenen Lösung beendet, können unterschiedliche Lösungen entstehen, wenn man die Reihenfolge der Aktionen untereinander vertauscht. Das ist auch besonders wichtig, wenn man im STRIPS-Planer nicht explizit überprüfen würde, ob die Aktion schon mal ausgeführt worden war, denn dann käme man in eine endlose Schleife rein.

Zwei move Aktionen

Für Shakey ist es durchaus möglich, dass er zuerst vom Raum 1 in den Korridor muss, also *move(room1, corridor)* und dann wieder zurück *move(corridor, room1)*.

In der Aktion move gibt es die Precondition *connected(From, To)*. Die oben genannten fixen Zustände beinhalten aber nur den Fakt *connected(room1, corridor)*. Somit stimmt die Precondition der move Aktion wenn Shakey vom Raum 1 in den Korridor will, aber nicht beim Zurückgehen.

Da die STRIPS-Aktionen nur positive literale beinhalten dürfen (also keine oder Verknüpfungen), wurde als Workaround einfach eine zweite move Aktion definiert, welche den zweiten Fall miteinschliesst.

Jedoch ist es vom Programmatischen Aspekt her schöner, wenn man mehr Zustände definiert und dafür weniger Aktionen hat. Deshalb wurden schliesslich die fixen Zustände um folgende literale erweitert:

- connected(corridor, room1)
- connected(corridor, room2)
- connected(corridor, room3)
- connected(door1, room1)
- connected(door2, room2)
- connected(door3, room3)
- connected(door1, corridor)
- connected(door2, corridor)
- connected(door3, corridor)

Allerdings stellten wir fest, dass diese Lösung von der Performance her schlechter ist, als wenn man zwei move Aktionen definiert. Der Testfall 10 (siehe Tests) benötigt durchschnittlich 70 Sekunden anstelle von 15 Sekunden.

Typensicherheit

Um noch mehr unnötige Aktionen von Anfang an zu vermeiden, wurden auch Typen eingebaut, welche in den Preconditions überprüft werden. Es gibt folgende drei Typen: room, door und object. Somit kann es nicht vorkommen, dass Shakey versucht sich selber aufzuheben.

Subset/2

Die eingebaute Subset-Klausel von SWI-Prolog überprüft nur, ob eine Liste in einer anderen Liste vorhanden ist, aber sie kann nicht durch Resolution alle möglichen Subsets generieren, was in unserem STRIPS-Planer benötigt wird. Deshalb wurde diese Funktion in unserem Programm ergänzt (siehe subset/2).

Tests

Damit das Programm während der Entwicklungsphase immer wieder getestet werden konnte, wurden zehn verschiedene Testfälle implementiert. Diese reichen von «gehe vom Raum 1 in den Korridor» bis zu «Nimm eine Box in Raum 1, gehe in den Raum 3, stelle dort die Box ab, gehe in den Raum 2, nimm dort eine andere Box, gehe wieder in Raum 1 und stelle die Box dort ab währenddem alle Türen immer geschlossen sind und auch wieder geschlossen sein müssen».

Ein Testfall hat folgende Eigenschaften:

- Nummer
- Text, was der Testfall machen soll
- Startzustand
- Endzustand

Dazu wurde auch noch einen Testrunner implementiert (test/1), welche alle Tests in der gegebenen Liste (Testnummern) ausführt und jeweils die Laufzeit ausgibt. Um alle Tests hintereinander auszuführen, reicht es runAllTests/0 auszuführen.