Projektarbeit Artificial Intelligence

Berner Fachhochschule Departement Technik und Informatik

Frühlingssemester 2021

Dozenten: Jürgen Eckerle

Manuel Gasser und Julian Haldimann

8. Juni 2021

1 Abstract

Die Aufgabe war es dem Agenten einen möglichen Weg zum Schatz zu zeigen. Der Schatz ist in einem der verschlossenen Räume versteckt. Der Ort des Schatzes ist jedoch bekannt. Lediglich der Weg zum Ziel ist ein Rätsel. Am Ende sollte eine mögliche Route zum Schatz in einer Liste abgebildet werden. Die Route beinhaltet verschiedene Schlüssel, welche genau in dieser Reihenfolge augelesen werden müssen um ans Ziel zu kommen. Falls keine Route vorhanden ist, wird einfach false zurückgegeben.

Damit wir überhaupt mit Prolog unser Projekt umsetzen konnten, mussten wir die Modellierung der Räume und der Schlüssel vornehmen. Ohne diese Definition können keine Abfragen gemacht werden. Bei der Thematik mit dem Raum und dessen Inhalt, haben wir uns überlegt für jede Kombination einen separaten Eintrag zu machen.

Dasselbe haben wir für verschachelte Räume gemacht. Jeder Raum der einen oder mehrere Räume enthält hat mindestens eine Definition erhalten. Damit wir von einem Startraum ausgehen konnten, haben wir einen solchen definiert. Dieser hat in unserem Fall die Raumnummer 0. Am Ende der Modellierung haben wir noch definiert, dass sich die Schatzkiste im Raum 13 befindet.

Am Ende der Arbeit konnten wir eine Weg zum Schatz finden. Der kalkulierte Weg wurde jedoch noch in einer falschen Reihenfolge ausgegeben und musste noch richtig sortiert werden. Dazu haben wir einen einfachen Bubblesort verwendet. Sortiert wurde nach Anzahl Schritten, welche man braucht um an einen Schlüssel zu kommen.

Inhaltsverzeichnis

2 Einführung

2.1 Projektbeschrieb

Als Projektarbeit für das Fach Artificial Intelligence haben wir folgende Ausgangssituation erhalten:

In einem Labyrinth von verborgenen Räumen befindet sich irgendwo ein Schatz verborgen, den es zu finden gilt. Die Türe zu den Räume sich verschlossen und lassen sich nur über den passenden Schlüssel öffnen. Räume können verschachtelt sein, das heisst innerhalb eines Raumes können weitere Räumen liegen, die ebenfalls durch Türen verschlossen sind.

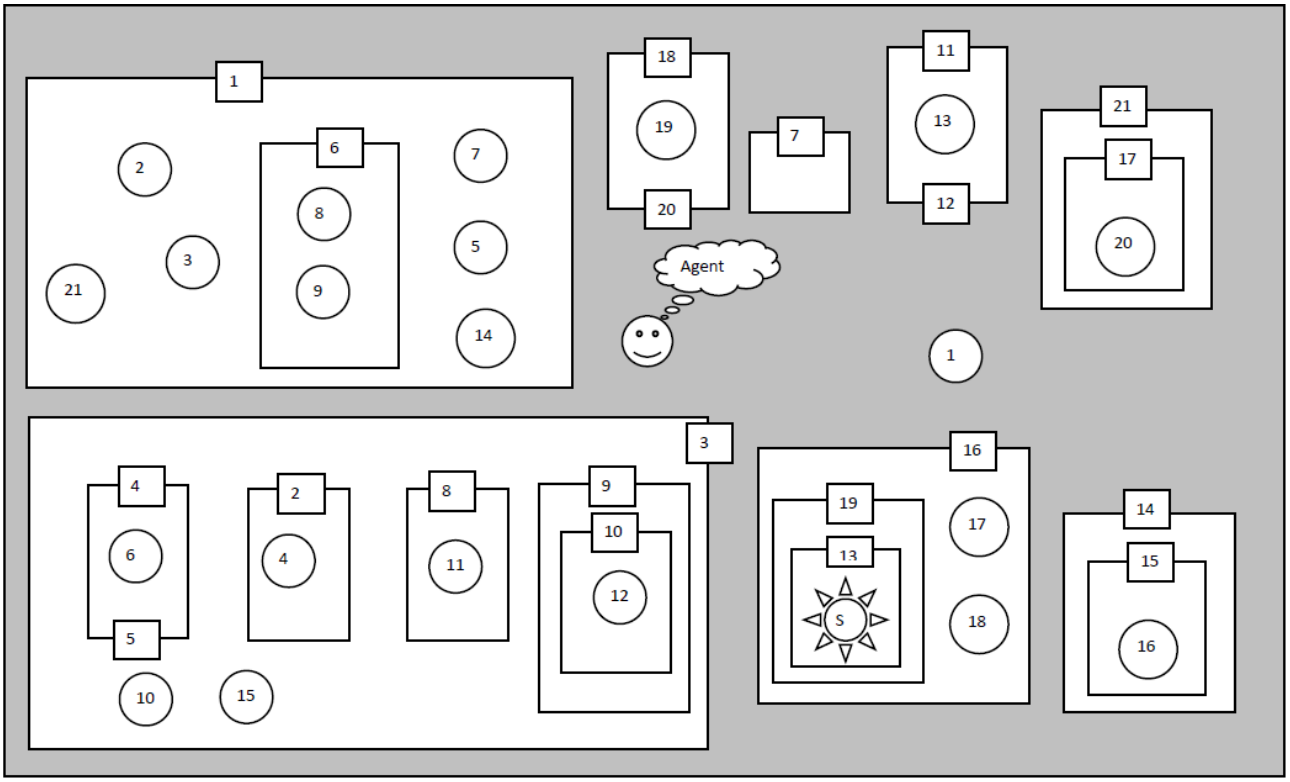

Abbildung 1: Ausgangslage

2.2 Aufgabenstellung

Folgende Aufgaben gibt es zu lösen in diesem Projekt:

- Schreiben Sie ein Prolog-Programm, das unsere obige Problemsituation modellieren unter Annahme, dass die vollständige Information gegeben ist. Es soll möglich sein, mittels Anfragen zu überprüfen, ob eine bestimmte Türe erreichbar ist oder ob ein bestimmter Raum betreten werden kann, usw.
- Modellieren Sie unter Verwendung von Listen die Möglichkeit einen Handlungsplan zu erhalten, der aufzeigt, wie die einzelnen Ziele erreicht werden können.

3 Vorgehen

In diesem Kapitel wollen wir Schritt für Schritt aufzeigen, wie wir beim Lösen der einzelnene Aufgaben vorgegenagen sind.

3.1 Modellieren der Ausgangslage

Bei der Modellierung der Ausgangslage, haben wir genau die Situation, wie sie auf dem Aufgabenblatt steht, in Prolog umgesetzt. Dabei haben wir für die Schlüssel und Tür Beziehung den Fact **roomkey(X,Y)** verwendet. Wobei **X** der Raum ist in welchem sich der Schlüssel **Y** befindet. So kann überprüft werden, ob eine Raum einen Schlüssel enthält.

Eine weitere wichtige Modellierung, welche zwingend notwendig war, ist die Verschachtelung der Räume. Wir müssen wissen, in welchem Raum sich welcher Unterraum befindet, damit wir an weitere Schlüssel kommen können. Deswegen haben wir den Fact **roomcontainsroom(X,Y)** erstellt. Bei diesem ist **X** der äussere und **Y** der innere Raum. Somit kann geprüft werden, ob ein Raum in einem anderen Raum vorhanden ist. Dies ist vorallem dann wichtig, wenn wir wissen wollen wo sich ein bestimmter Raum befindet.

Zu guter Letzt haben wir noch den Raum festgelegt, in welchem sich der Schatz befindet, welchen man schlussendlich finden muss. Für dies haben wir den Fact **roomcontainschest(X)**, wobei X der Raum ist wo sich der Schatz befindet.

3.2 Verwendete Rules

In diesem Kapitel beschreiben wir alle von uns verwendeten Rules. Dazu wird auch immer eine kleiner Codeausschnitt zu sehen sein.

3.2.1 Union

Bei der Union-Rule wird, wie der Name schon sagt, eine Vereinigung aus zwei Listen gemacht. Diese verwenden wir anschliessend in der **cangetchestfrom** Rule.

```
union([X|Y],Z,W) :- member(X,Z),  union(Y,Z,W).
union([X|Y],Z,[X|W]) :- \+ member(X,Z), union(Y,Z,W).
union([],Z,Z).
```

3.2.2 Member

Mit dieser Rule überprüfen wir, ob sich ein Element bereits in unserer Liste befindet.

```
member(X,[X|_]).
member(X,[_|TAIL]) :- member(X,TAIL).
```

3.2.3 ListAppend

Um bei einer Liste ein Element hinzuzufügen haben wir die **list_append()** Rule geschrieben. Damit können wir alle Schlüssel, welche nötig sind um zum Schatz zu kommen, in einer Liste anzeigen.

```
list_append(A,T,T) :- member(A,T),!.
list_append(A,T,X) :- append([A],T,X).
```

3.2.4 BubbleSort

Um die Schlüssel in der Reihenfolge, in welcher sie geholt werden müssen, anzeigen zu können sortieren wir die Liste der Schlüssel nach der Anzahl Schlüssel, welche benötigt werden um ihn zu erreichen. Wir uns für den einfachsten Sortieralgorithmus entschieden. Den Bubblesort haben wie folgt umgesetzt:

```
bubble_sort(List,Sorted):-b_sort(List,[],Sorted).
b_sort([],Acc,Acc).
b_sort([H|T],Acc,Sorted):-bubble(H,T,NT,Max),b_sort(NT,[Max|Acc],Sorted).

bubble(X,[],[],X).
bubble(X,[Y|T],[Y|NT],Max):-nth0(0,X,A), nth0(0,Y,B), A>B,bubble(X,T,NT,Max).
bubble(X,[Y|T],[X|NT],Max):-nth0(0,X,A), nth0(0,Y,B), A=<B,bubble(Y,T,NT,Max).
```

3.2.5 CanGetKeyFrom

Mit dieser Rule wollen wir feststellen, ob eine Schlüssel **K** von einem Raum **F** aus erreichbar ist. Wenn der Schlüssel erreicht werden kann, wird eine Liste **B** mit allen notwendigen Schlüsseln zurückgegeben. **L** ist am Anfang eine leere Liste, welche mit **list_append()** befüllt wird.

```
cangetkeyfrom(K,F,L,B):- (roomkey(F, K), list_append(K, L, B) ;
    (K > 0, list_append(K, L, A), roomkey(X, K), cangetkeyfrom(X,F,A,C),
    (roomcontainsroom(F,X), append([],C,B) ;
    (roomcontainsroom(Y,X), cangetkeyfrom(Y,F,C,B))))).
```

3.2.6 CanReachRoomFrom

Wie der Name bereits sagt, wollen wir herausfinden, ob ein Raum **R** von einem anderen Raum **F** aus erreicht werden kann. Wenn dies möglich ist, wird die Funktionen eine Liste **L** mit Schlüsseln zurückgeben. Ansonsten wird **false** als Resultat returniert.

```
canreachroomfrom(R,F,L):- cangetkeyfrom(R,F,[],A),
    (roomcontainsroom(F,R), append([],A,L) ;
    (roomcontainsroom(Y,R), cangetkeyfrom(Y,F,[],L))).
```

3.2.7 CanGetChestFrom

Mit dieser Rule stellen wir fest, ob wir den Schatz von einem bestimmten Raum **F** aus finden können. Wenn Prolog eine Lösung gefunden hat, wird eine Liste **L** mit benötigten Schlüsseln zurückgegeben. Diese Liste ist im Endeffekt nichts anderes als eine Schritt für Schritt Anleitung, wie der Schatz erreicht werden kann. Am Ende der Liste wird zusätzlich noch ein **S** angefügt, welches den Schatz symbolisiert.

```
cangetchestfrom(F,L):- roomcontainschest(X), cangetkeyfrom(X,F,[],A),
    canreachroomfrom(X,F,B), union(A,B,C), sortbykey(F,C,D), append(D,["S"],L).
```

3.2.8 LengthKey

Um unsere Reihenfolge der Schritte richtig zu bestimmen, haben wir eine Funktion geschrieben, welche ein Tupel aus dem Schlüssel **K** und der Anzahl Schlüssel **N**, um den Schlüssel zu erreichen, erstellt. So wissen wir genau wo in unserer Liste der entsprechende Schlüssel abgelegt werden muss.

```
lengthkey(K,O):- cangetkeyfrom(K,0,[],B), length(B,N), O=[N,K].
```

3.2.9 GetKeyOnly

Diese Rule wird verwendet um den Schlüssel aus dem vorher erstellten Tupel herauszuholen.

```
getkeyonly(LK,K):- nth0(1,LK,K).
```

3.2.10 SortByKey

Zu guter Letzt, wollen wir noch die Liste der Tupels sortieren, welche in den vorherigen Rule erstellt wurden. Dazu verwenden wir **maplist** und unseren Bubblesort Algorithmus.

```
sortbykey(F,L,O):- maplist(lengthkey, L, A),
    bubble_sort(A,S), maplist(getkeyonly, S, O).
```

4 Schlussfolgerung

4.1 Ergebnis

Wenn wir unsere Programm auführen erhalten wir folgende Ausgabe:

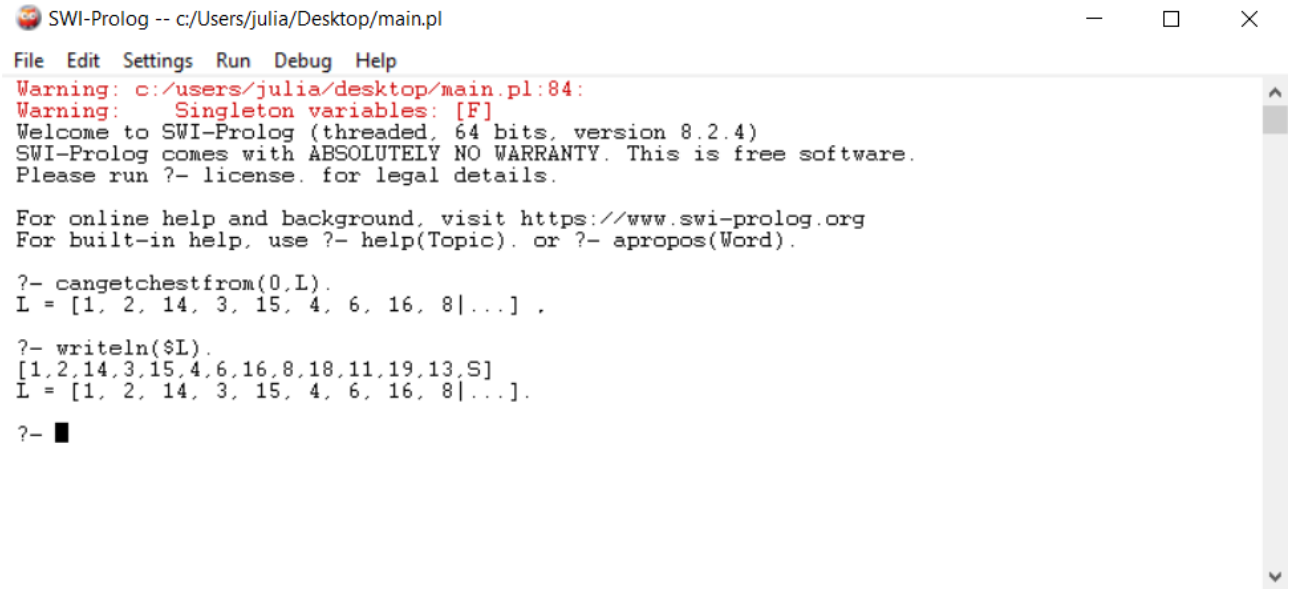

Abbildung 2: Output Prolog Keys and Doors

Wie auf der Abbildung zu erkennen ist, braucht der Agent genau 13 Schritte bis er beim Schatz angekommen ist. Dies ist nur eine von mehreren Lösungen, welche das Programm haben könnte.

4.2 Fazit

Es war interessant einmal mit einer Sprache zu programmieren, welche auf einem anderen Konzept basiert. Für uns war es etwas schwierig kein, da der Programaufbau auf Rules, Facts und Queries basiert. Dies ist ganz anders als wir es uns von Sprachen wie Java oder Python gewöhnt sind. Es hat deshalb ein wenig Zeit gebraucht um mit Prolog klar zu kommen. Die Sprache hat ihre Eigenheiten und muss zuerst gut kennengelernt werden, bevor man mit der Programmierung beginnt.

Abbildungsverzeichnis