



Projektarbeit Artificial Intelligence

Berner Fachhochschule Departement Technik und Informatik

Frühlingssemester 2021

Dozenten: Jürgen Eckerle

Manuel Gasser und Julian Haldimann

1. Juni 2021

1 Abstract

Inhaltsverzeichnis

1	Abstract	1
2	Einführung	2
2.1	Projektbeschreibung	2
2.2	Aufgabenstellung	2
3	Vorgehen	3
3.1	Modellieren der Ausgangslage	3
3.2	Verwendete Funktionen	3
3.2.1	Union	3
3.2.2	Member	3
3.2.3	ListAppend	4
3.2.4	BubbleSort	4
3.2.5	cangetkeyfrom	4
3.2.6	canreachroomfrom	4
3.2.7	cangetchestfrom	4
3.2.8	lengthkey	5
3.2.9	getkeyonly	5
3.2.10	sortbykey	5
4	Ergebnisse	6

2 Einführung

2.1 Projektbeschreibung

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 Türen zu den Räumen sind verschlossen und lassen sich nur über den passenden Schlüssel öffnen. Räume können verschachtelt sein, das heißt innerhalb eines Raumes können weitere Räume liegen, die ebenfalls durch Türen verschlossen sind.

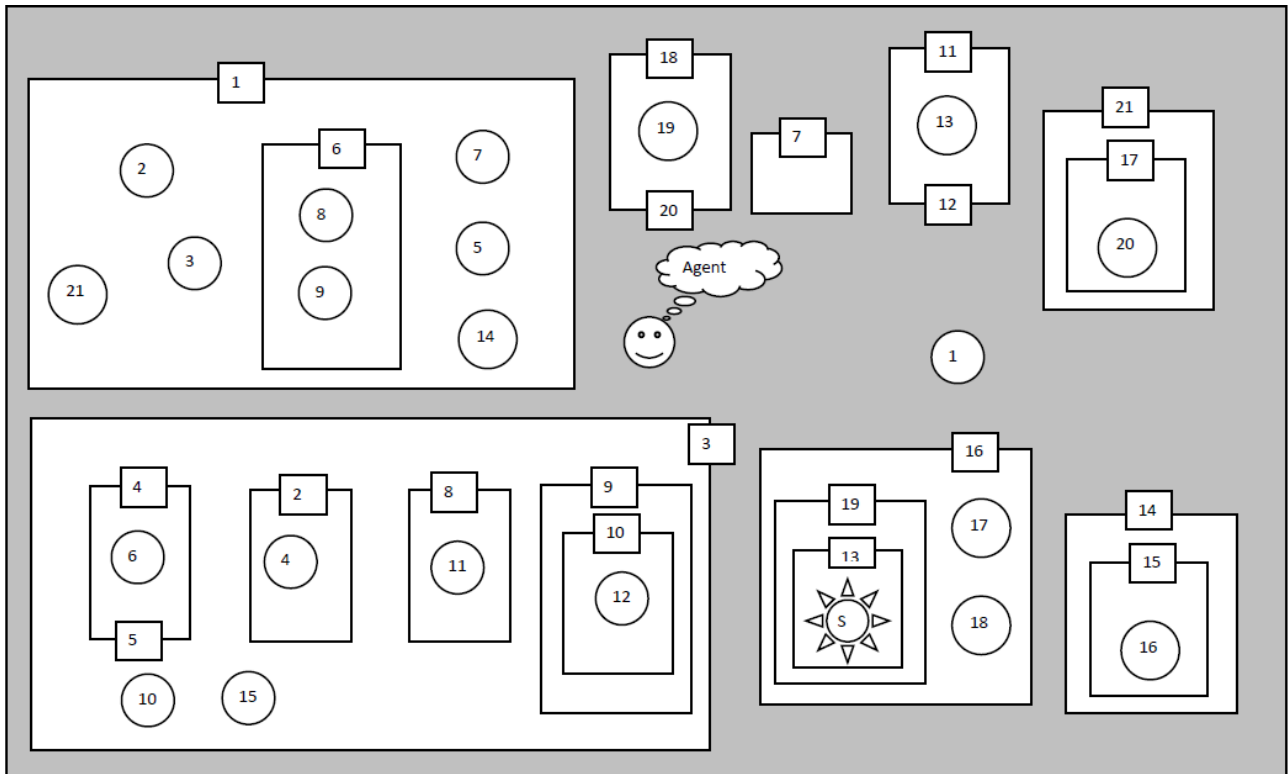


Abbildung 1: Ausgangslage

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 Tür 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 angeben wie wir beim Lösen der einzelnen Aufgaben vorgehen sind.

3.1 Modelieren 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 das Keyword **roomkey()** verwendet. Dabei wird in der ersten Position in der Klammer der Raum und an der zweiten Position der Schlüssel eingesetzt. So kann überprüft werden, ob ein 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 die Funktion **roomcontainsroom** erstellt, welche als erstes Argument den übergeordneten Raum nimmt und an der zweiten Stelle den Unterraum. 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. Diese Funktion wird auch nur verwendet um zu vergleichen, ob das Ziel erreicht wurde.

3.2 Verwendete Funktionen

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

3.2.1 Union

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

```
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 Funktion überprüfen wir ob ein Element bereits in unsere Liste vorhanden ist. Wenn es vorhanden ist, wird **True** zurück gegeben.

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

3.2.3 ListAppend

Um unsere Resultate zu speichern, haben wir uns eine Funktion zusammengebaut, welche Elemente einer Liste hinzufügt.

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

3.2.4 BubbleSort

Da wir unsere Liste noch nach Anzahl benötigter Schlüsser sortieren müssen, haben wir uns für den einfachsten Sortieralgorithmus entschieden. Den Bubblesort haben wir 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).
```

Bei unsere Lösung haben wir eine Liste verwendet, welche bei 0 und nicht bei 1 startet. Deswegen verwenden wir **nth0**.

3.2.5 cangetkeyfrom

Mit dieser Funktion wollen wir herausfinden ob eine Schlüssel K von einem Raum F erreichbar ist. Wenn der Schlüssel erreicht werden kann, wird eine Liste B mit allen notwendigen Schlüsseln zurückgegeben.

```
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

Diese Funktion ist zuständig herauszufinden, ob wir den Schatz von einem bestimmten Raum **F** aus bekommen 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 eine **S** angefügt. Dies symbolisiert den zu findenden Schatz.

```
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 um den Raum **N** zu erreichen erstellt. So wissen wir genau wo in unserer Liste der entsprechende Schlüssel abgelegt werden muss.

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

3.2.9 getkeyonly

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

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

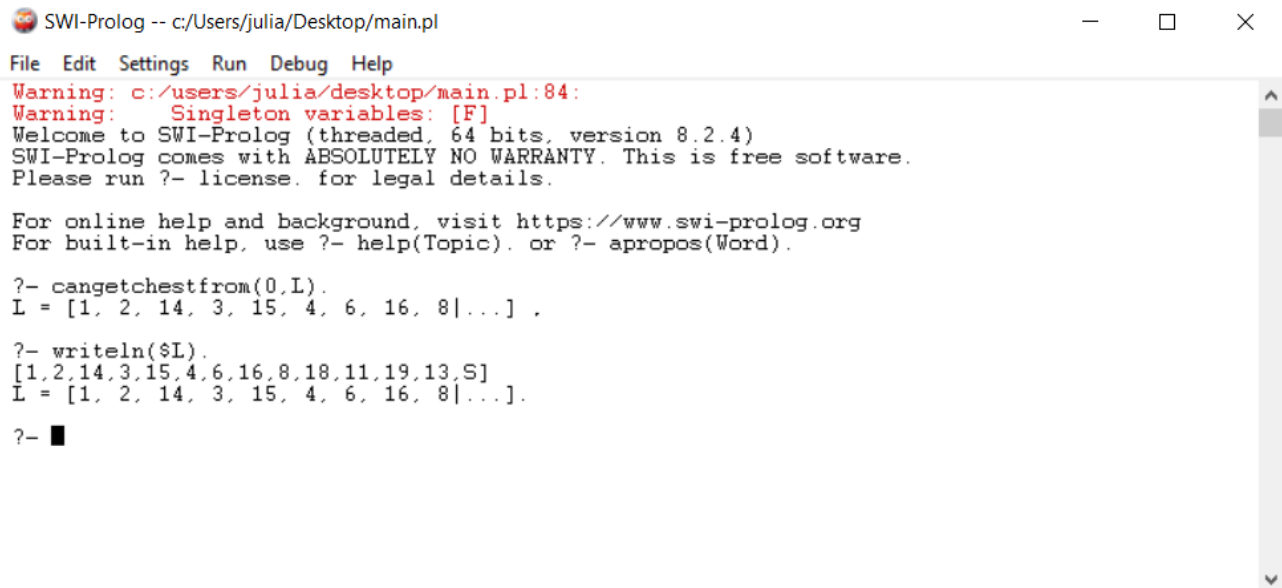
3.2.10 sortbykey

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

```
sortbykey(F,L,0):- maplist(lengthkey, L, A),  
                  bubble_sort(A,S), maplist(getkeyonly, S, 0).
```

4 Ergebnisse

Wenn wir unsere Programm ausführen erhalten wir folgende Ausgabe:



```
SWI-Prolog -- c:/Users/julia/Desktop/main.pl
File Edit Settings Run Debug Help
Warning: c:/users/julia/desktop/main.pl:84:
Warning: Singleton variables: [F]
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- cangetchestfrom(0,L).
L = [1, 2, 14, 3, 15, 4, 6, 16, 8|...] .

?- writeln($L).
[1,2,14,3,15,4,6,16,8,18,11,19,13,S]
L = [1, 2, 14, 3, 15, 4, 6, 16, 8|...] .

?-
```

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.

Tabellenverzeichnis

Abbildungsverzeichnis

1	Ausgangslage	2
2	Output Prolog Keys and Doors	6