

SS2010
BAI2-LBP Gruppe 1 Team 07
Lösung zu Aufgabe 3

R. C. Ladiges, D. Fast

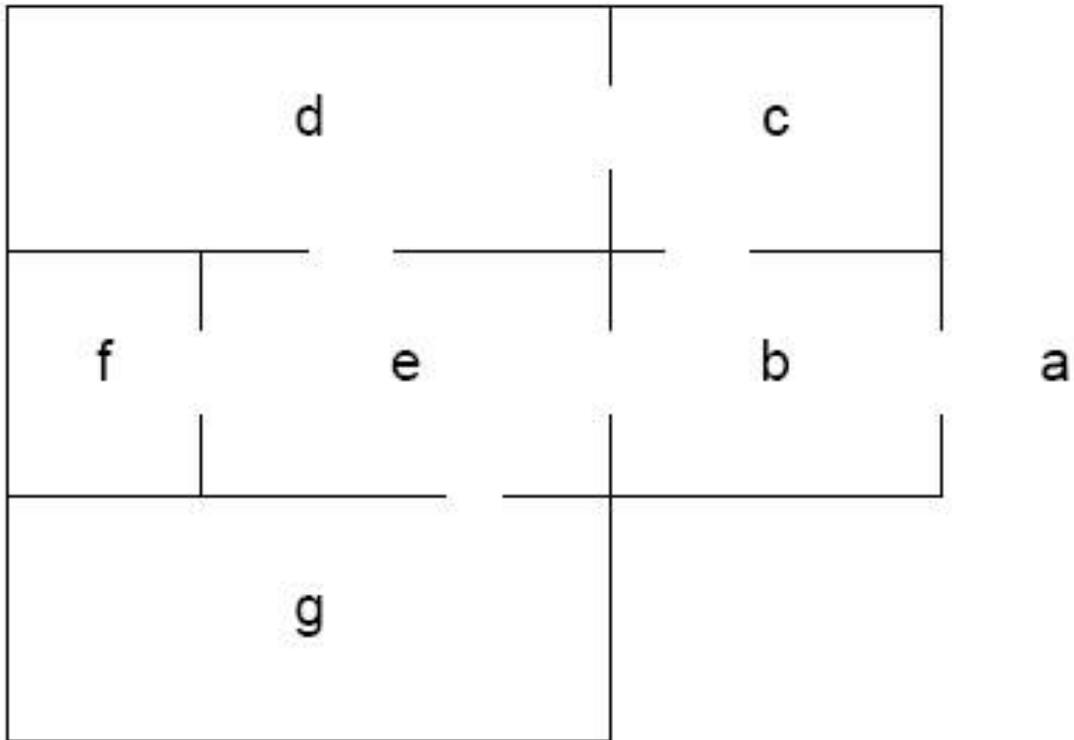
19. Mai 2010

Inhaltsverzeichnis

3 Aufgabe 3	3
3.1 Faktenbasis	4
3.1.1 Aufgabenstellung	4
3.1.2 Entwurf	4
3.1.3 Quelltext	4
3.2 Zieleigenschaft	5
3.2.1 Aufgabenstellung	5
3.2.2 Entwurf	5
3.2.3 Quelltext	5
3.3 Tiefensuche	5
3.3.1 Aufgabenstellung	5
3.3.2 Entwurf	5
3.3.3 Quelltext	6
3.4 Nicht im Kreis laufen	6
3.4.1 Aufgabenstellung	6
3.4.2 Entwurf	6
3.4.3 Quelltext	7
3.5 Kürzesten Weg finden	7
3.5.1 Aufgabenstellung	7
3.5.2 Entwurf	7
3.5.3 Quelltext	8
3.6 Breitensuche	9
3.6.1 Aufgabenstellung	9
3.6.2 Entwurf	9
3.6.3 Quelltext	10
3.7 Tiefen-/Breitensuche Vergleich	11
3.7.1 Aufgabenstellung	11
3.7.2 Entwurf	11
3.7.3 Quelltext	11

3 Aufgabe 3

In dieser Aufgabe soll ein Prgrogramm implementiert werden, dass den Weg in einem Labiryntn zu einem Raum mit einer bestimmten Eigenschaft findet. Folgendes Labyrinth ist vorgegeben:



Das erstellte Programm sollte jedoch auch mit anderen Labyrinthn arbeiten können! Beispiele: labyrinth_xs.pl, labyrinth_s.pl, labyrinth_xl.pl.

3.1 Faktenbasis

3.1.1 Aufgabenstellung

Implementieren Sie die Faktenbasis. Startpunkt ist in den Beispielen immer `a` wird im Allgemeinen jedoch über das Prädikat `start(Punkt)` vorgegeben. Damit die Basis austauschbar ist, verwenden Sie bitte folgende Darstellung: `tuer(a,b)`. Bedenken Sie, dass man Türen in zwei Richtungen beschreiten kann!

3.1.2 Entwurf

Wir betrachten das Labyrinth als einen **DG**(Directed Graph), in dem unsere Räume Knoten und unsere Türen Kanten sind. Diesen Graphen durchlaufen wir Knoten für Knoten und überprüfen bei jedem neuen Knoten den wir erreichen ob die Zieleigenschaft erfüllt ist, also ob wir in einem Raum sind, der eine Klausur enthält.

Zuerst erstellen wir Fakten für unsere Knoten als `raum/1`, die ein Literal ganz klar als Raum definiert (hierbei betrachten wir das Äußere eines Labyrinths auch als „Raum“). Außerdem brauchen wir Fakten für unsere Kanten als `tuer/2`, die sagen, dass zwischen zwei Räumen eine Tür existiert. Hierbei gibt `tuer/2` durch die Reihenfolge ganz klar eine Richtung vor, was nicht gewollt ist. Damit wir Türen durch beide Richtungen beschreiten können müssten wir nun für jede Raumkombination den Fakt zweimal definieren, oder wir erstellen uns dafür ein Prädikat `weg/2`. Dieses Prädikat besteht aus zwei Regeln, eine für den Hinweg und eine für den Rückweg einer Kante. Dadurch sparen wir im Vergleich zur doppelten Definition einer Raumkombination die Hälfte aller `tuer/2` Fakten.

Bei `labyrinth_xs.pl` brauchen wir so nur 7 statt 14, bei `labyrinth_s.pl` nur 9 statt 18, und bei `labyrinth_xl.pl` nur 25 statt 50 Türen zu definieren, was um einiges leichter zu erstellen und warten ist (weniger Fehlerquellen). Redundante Raumkombinationen bei Türen in gegebenen Labyrinthen müsste man auskommentieren, ansonsten werden Lösungen doppelt ausgegeben.

Zusätzlich benötigen wir noch ein Prädikat `start/1` (Fakt) welches einen Raum als Startraum festlegt.

3.1.3 Quelltext

```
raum(a). raum(b). raum(c). raum(d). raum(e). raum(f). raum(g).

tuer(a, b).
tuer(b, c).
tuer(b, e).
%tuer(c, b).
tuer(c, d).
%tuer(d, c).
tuer(d, e).
%tuer(e, b).
%tuer(e, d).
tuer(e, f).
tuer(e, g).
%tuer(f, e).
```

```

%tuer(g, e).

weg(X, Y):- raum(X), raum(Y), X\=Y, tuer(X, Y). %hinweg
weg(X, Y):- raum(X), raum(Y), X\=Y, tuer(Y, X). %rückweg

start(a).

```

3.2 Zieleigenschaft

3.2.1 Aufgabenstellung

Als Eigenschaft ist vorgegeben: `klausur(g)` ., d.h. in Raum `g` liegt eine verlorene LB-Klausur. Jedoch ist dies nicht als Ziel vorzugeben, sondern nach dem betreten eines Raumes zu prüfen!

3.2.2 Entwurf

Ein Prädikat `klausur/1` (Fakten) sagt das in einem bestimmten Raum eine Klausur liegt. Dies ist sowohl für die **Tiefen-** als auch **Breitensuche** wichtig, um zu wissen wann ein betretener Raum, samt Weg zu ihm, als Lösung ausgegeben werden soll, beim Durchlaufen des Labyrinth.

3.2.3 Quelltext

```

klausur(g).

```

3.3 Tiefensuche

3.3.1 Aufgabenstellung

Implementieren Sie nun als **Tiefensuche** ein Prädikat `suche_weg_ts/1`: `suche_weg_ts(Loesung)`, dass als Resultat eine Liste ausgibt (in der Variable `Loesung`), in der der Weg zur Klausur beschrieben wird.

3.3.2 Entwurf

Zur Umsetzung der **Tiefensuche** machen wir uns die Arbeitsweise von Prolog zu nutze. Welche von einem Knoten ausgehend, über alle ausgehenden Kanten, alle Nachfolgeknoten sequenziell durchläuft, bis es sein Ziel findet, mit jeweiligem **Backtracking**. Die Suche in einem Graphen würde also solange laufen, bis ein Knoten ohne Nachfolger gefunden wird, von welchem aus, es dann zurück zum vorigen Knoten gehen würde um die Suche fortzusetzen.

Das Prädikat `suche_weg_ts/1` sucht sich als Erstes den Startknoten und übergibt ihn als Zustand ¹ an ein Unterprädikat `finde_ts/2`. Dieses Prädikat bekommt als Eingabe einen Zustand und gibt nach und nach alle Wege zu einem Raum, in dem die Zieleigenschaft erfüllt ist, aus.

Für die Umsetzung des `finde_ts/2` Prädikats haben wir nur zwei Fälle.

¹Ein Zustand ist ein Tupel (zwei-elementige-Liste) aus dem Raum in der sich der Zustand befindet, und einer Liste aus vorherigen Räumen. Beispiel: `start(a)` führt zum Zustand `[a, []]`, von diesem weitergegangen sind wir in Zustand `[b, [a]]` und von diesem weitergegangen in den Zuständen `[a, [a,b]]`, `[c, [a,b]]` und `[e, [a,b]]`.

1. Eine Regel (Rekursionsabbruch) die erkennt ob der Raum in dem sich der übergebene Zustand befindet die Zieleigenschaft hat. In diesem Fall wird der Zustand zu einem Weg² und wird ausgegeben.
2. Erfüllt der Raum des übergebenen Zustandes nicht die Zieleigenschaft, werden nach und nach über das Prädikat `weg/2` alle Nachfolgeräume des Raumes ermittelt. Der übergebene Zustand wird zu einem Weg, welcher verknüpft mit dem jeweiligen Nachfolgeraum zu einem neuen Zustand wird, den wir in die Rekursion geben.

Diese Umsetzung des Prädikats würde aber nicht richtig funktionieren, da der mehrmalige Besuch eines Raumes möglich ist. Somit würde das Prädikat gegebenenfalls immer im Kreis laufen bei bestimmten Räumen und Verbindungen. Zum Beispiel bei dem vorgegebenen Labyrinth könnte es von Raum B zu C zu D zu E und von Raum E wieder in den Raum B gelangen, und dann immer so weiter.

Dadurch ist diese Implementation des Prädikates nur auf Labyrinth anwendbar die wie ein **DAG**(Directed Acyclic Graph) aufgebaut sind. Um diese Implementation zu testen kann man die zweite Regel für den Rückweg des Prädikates `weg/2` auskommentieren. Sofern man die Türen in der „richtigen“³ Reihenfolge definiert hat.

3.3.3 Quelltext

```
suche_weg_ts(Loesung):-
    start(Start),
    finde_ts([Start,[], Loesung).

finde_ts([Raum,Bisher], Weg):-
    klausur(Raum),
    append(Bisher, [Raum], Weg).
finde_ts([Raum,Bisher], Weg):-
    weg(Raum, Nachfolger),
    append(Bisher, [Raum], BisherNeu),
    finde_ts([Nachfolger,BisherNeu], Weg).
```

3.4 Nicht im Kreis laufen

3.4.1 Aufgabenstellung

Vermeiden Sie ein im Kreis laufen!

3.4.2 Entwurf

Um das im Kreis laufen zu vermeiden, bei dem es sonst in einem **DG** zu einen **Stack Overflow** kommt, muss lediglich der zweite Fall (beim Rekursionsaufruf) von `finde_ts/2` so erweitert werden, dass Nachfolgeräume die wir bereits besucht haben nicht betrachtet werden. Dies können wir dadurch tun, dass bei diesem Fall die Nachfolgeräume die uns `weg/2` liefert nicht

²Wie bekommen wir aus einem Zustand einen Weg? Ganz einfach, in dem man mit `append/3` den Raum, in dem der Zustand ist, an seine bisherigen Räume anhängt. Beispiel: Der Weg des Zustandes `[c, [a,b]]` ist `[a,b,c]`.

³Die „richtigen“ Reihenfolgen für das gegebene Labyrinth, damit es ein **DAG** ist, lauten: (A,B), (B,C), (B,E), (C,D), (D,E), (E,F) und (E,G).

member/2 der bisher besuchten Räume des übergeben Zustands sein dürfen.

Das wir schon besuchte Räume nicht erneut betrachten müssen liegt daran, dass wir alle Wege von diesem Raum aus ja schon betrachtet haben (oder noch mit **Backtracking** betrachten werden).

Dadurch funktioniert die Suche nun in allen **DG** und ist nicht mehr auf **DAG** beschränkt (weshalb die zweite Regel des weg/2 Prädikats nicht länger ausgeklammert sein darf).

Durch unser finde_ts/2 ist es uns nun auch möglich manuell Anfragen an das Labyrinth zu stellen um von anderen Räumen als A einen Weg zum Ziel zu finden. Z.B. vom Raum C aus zum Ziel:

```
?- finde_ts([c,[],], OUT).
OUT = [c, d, e, g] ;
OUT = [c, b, e, g] ;
false.
```

3.4.3 Quelltext

```
suche_weg_ts(Loesung):-
    start(Start),
    finde_ts([Start,[],], Loesung).

finde_ts([Raum,Bisher], Weg):-
    klausur(Raum),
    append(Bisher, [Raum], Weg).
finde_ts([Raum,Bisher], Weg):-
    weg(Raum, Nachfolger),
    not(member(Nachfolger, Bisher)), %neu
    append(Bisher, [Raum], BisherNeu),
    finde_ts([Nachfolger,BisherNeu], Weg).
```

3.5 Kürzesten Weg finden

3.5.1 Aufgabenstellung

Erweitern Sie Ihr Programm um das Prädikat kuerzester_weg/1: kuerzester_weg(Loesung), der unter Verwendung des Prädikates suche_weg/1 alle möglichen Wege findet und davon den (oder die) kürzesten Wege auswählt. Das Prädikat findall/3 darf dabei nur auf das Prädikat suche_weg_ts/1 angewendet werden.

3.5.2 Entwurf

Wenn wir das Prädikat findall/3 auf suche_weg_ts/1 anwenden, erhalten wir eine Liste, welche alle Wege⁴ zu einem Raum mit einer Klausur enthält (bei denen kein Raum doppelt betreten wurde).

Auf diese Liste wenden wir ein eigenes Unterprädikat diekleinsten/2 an, welches uns, nach

⁴Ein Weg ist hier eine Liste von Räumen. Die Liste von Links nach Rechts gelesen ergibt die Raum-Reihenfolge zum Klausur-Raum.

und nach als verschiedene Lösungen, die kürzesten Wege (gleiche kleinste Anzahl Räume) ausgibt, was wir einfach nur noch nach Außen weitergeben.

In `diekleinsten/2` bestimmen wir zuerst mit dem eigenem Unterprädikat `kleinsteanzahl/2` die Anzahl $\in \mathbb{N}$ der Räume der kürzesten Wege. Diese Zahl geben wir zusammen mit unserer Liste an ein eigenes Unterprädikat `alle_mit_anzahl/3` weiter, welche uns alle Wege in der Liste mit gegebener Anzahl nach und nach ausgibt.

Um die Anzahl der Räume eines Weges zu ermitteln, brauchen wir ein Prädikat `anzahl/2`, das rekursiv die Anzahl Elemente in einer Liste zählt.

1. Rekursionsabbruch ist die leere Liste mit Anzahl 0.
2. Beim Rekursionsschritt ist die Anzahl 1, für jeden Kopf, auf den rekursiven Aufruf des Rests zu addieren.

Das Prädikat `kleinsteanzahl/2` liefert uns die Anzahl der Räume der kürzesten Wege.

1. Bei einem einzelnen Weg (Rekursionsabbruch) berechnen wir die `anzahl/2` dessen.
2. Bei mehreren Wegen (Rekursionsschritt) berechnen wir die `anzahl/2` des Kopfes und rekursiv die `kleinsteanzahl/2` des Rests (Rekursionsaufruf). Für die beiden Anzahlen haben wir eine Fallunterscheidung, welche die jeweils kleinere Anzahl mit der Ausgabevariable unifiziert.

Fehlt noch das Prädikat `alle_mit_anzahl/3`, das sich über das Prädikat `member/2` alle Wege ausgeben lässt, und nur die nach Außen gibt, welche die angegebene Anzahl haben.

3.5.3 Quelltext

```
kuerzester_weg(Loesung):-
    findall(Tmp, suche_weg_ts2(Tmp), ListeAllerWege),
    diekleinsten(ListeAllerWege, Loesung).

diekleinsten(List, Out):-
    kleinsteanzahl(List, N),
    alle_mit_anzahl(List, N, Out).

kleinsteanzahl([X], N):-!, anzahl(X, N).
kleinsteanzahl([K|R], Out):-
    anzahl(K, T1),
    kleinsteanzahl(R, T2),
    ( (T2>=T1, Out=T1) ; (T1>T2, Out=T2) ),
    !.

alle_mit_anzahl(Liste, Anzahl, Out):-
    member(Out, Liste),
    anzahl(Out, Anzahl).

anzahl([], 0).
anzahl([_|R], AnzahlOut):-
    anzahl(R, AnzahlRest),
    AnzahlOut is AnzahlRest + 1.
```

3.6 Breitensuche

3.6.1 Aufgabenstellung

Implementieren Sie nun als **Breitensuche** ein Prädikat `suche_weg_bs/1`: `suche_weg_bs(Loesung)`, das als Resultat eine Liste ausgibt (in der Variable `Loesung`), in der der Weg zur Klausur beschrieben wird. Als vordefiniertes Prädikat darf `append`, `findall`, `last`, `member` und `not` verwendet werden!

3.6.2 Entwurf

Die Idee hinter unserer Lösung ist ein Labyrinth als eine Art **PDA**(Pushdown automaton) zu betrachten, der in verschiedenen Zuständen (Räumen), mit dazugehörigen Stacks (Liste der bisher besuchten Räume), gleichzeitig sein kann.

Mit einer Art Übergangsfunktion `schritt/2`, die eine Liste aller Zustände mit dazugehörigen Stacks, in denen wir uns befinden, bekommt, ermitteln wir alle Räume mit denen wir bei einem bestimmten Zustand eine Verbindung haben.

Die Räume mit denen wir eine Verbindung haben bereinigen wir, indem wir die Räume entfernen die wir bei dem Zustand bereits besucht haben (Betrachtung des Stacks als ganzes mit `member/2` und nicht nur des obersten Elements).

Aus allen Räumen, die nach der Bereinigung übrig sind, bilden wir neue Zustände, indem wir ihnen einen Stack zuordnen, der zusätzlich den bestimmten Raum des vorigen Zustandes enthält.

Sofern wir nicht im Kreis laufen, was dadurch garantiert ist, dass wir Räume in denen wir noch nicht waren nicht betrachten, terminieren alle Zustände nach einer endlichen Anzahl Schritten. Nach jedem Schritt betrachten wir alle Zustände, in denen wir uns nun befinden, mit `member/2`. Ist einer der Zustände in einem Raum mit einer Klausur, geben wir seinen Weg als Lösung aus. Alle akzeptierenden Zustände die nach einem gleichem Schritt gefunden werden, haben einen gleich kurzen Weg hinter sich.

Nachdem wir alle akzeptierenden Zustände als Lösungswege ausgegeben haben, betrachten wir die nicht akzeptierenden Zustände. Davon betrachten wir nur einen (welchen ist egal, er soll nur folgendes auslösen:) und führen rekursiv auf alle Zustände (gleichzeitig) die Suche erneut aus.

Vorige Beschreibung sollte nur sehr grob die Vorgehensweise umschreiben, um das folgende etwas verständlicher zu machen:

Unser Prädikat `suche_weg_bs/1` ermittelt den Startraum, und gibt diesen als einzigen Zustand in eine Liste für unser `finde_bs/2` Prädikat.

Das Prädikat `finde_bs/2` findet zu einer eingegebenen Liste⁵ von Zuständen, alle Zustände und später (Rekursion) alle Folgezustände, die das Zielprädikat erfüllen.

1. Unser Rekursionsabbruch ist, wenn wir keine Zustände mehr haben (leere Liste). In diesem Fall schließen wir das finden von weiteren Lösungen aus (Cut), und lassen den Prädikatszweig sterben (fail).
2. Mit `member/2` geben wir alle Zustände der Liste (einzelnd, nach und nach) als Weg aus, die in einem Raum sind, für den `klausur/1` Wahr ist.

⁵im Gegensatz zur **Tiefensuche**, wo wir nur einen Zustand betrachten.

3. Schließlich bleibt für das Prädikat `findebs/2` noch der Fall für den rekursiven Aufruf. Dazu führen wir einen `schritt/2` auf unsere Liste von Zuständen aus, und geben das Ergebnis davon in die Rekursion.

`schritt/2` liefert uns alle Nachfolgezustände einer Liste von Zuständen.

1. Wenn wir keine Zustände reinbekommen (Eingabeliste leer), können wir auch keinen Schritt auf diese Zustände durchführen. Folglich ist die Ausgabeliste leer (Rekursionsabbruch).
2. In allen anderen Fällen geht das Prädikat (nach und nach) alle Kopfzustände der Eingabeliste durch (Rekursion) und verfährt rekursiv mit dem Restzuständen (Rekursionsaufruf). Bei dem Kopfzustand sucht er zuerst mit `findall/3` auf `weg/2` ALLE Nachfolgeräume (Einfache Liste von Räumen). Die Nachfolger bereinigt⁶ er dann mit dem Unterprädikat `entfBisherige/3`. Aus allen bereinigten Nachfolgern machen wir nun Zustände (dies sind alle Nachfolgezustände des Kopfzustands), indem wir sie mit dem Weg unseres Kopfzustandes per Unterprädikat `addBisherige/3` verbinden. Alle Nachfolgezustände des Kopfzustandes mit allen Nachfolgezuständen der Restzustände per `append/3` verbunden ergeben die bereinigten Nachfolgezustände aller Zustände der Eingabeliste.

Anstatt sich zwei Prädikate `entfBisherige/3` und `addBisherige/3` zu erstellen, geht es auch mit unserem bereits in Aufgabe 1.6 erstellten Prädikat `kombiniere/3`, was zum gleichem Resultat führt.

Hierbei wäre `InfoListe` die Liste aller Nachfolgeräume des Kopfzustandes, `Daten` der Weg des Kopfzustandes und die Ausgabe `Kombination` alle Nachfolgezustände des Kopfzustandes.

Das interessante am Prädikat `schritt/2` ist, dass alle Zustände, durch die Betrachtung mit den vorher besuchten Räumen, in einer endlichen Anzahl Schritten garantiert enden. Heißt, dass wenn wir nur Zustände haben, bei denen alle Nachfolger schon vorkamen, liefert `schritt/2` eine leere Liste, und `finde_bs/2` endet dadurch endgültig.

3.6.3 Quelltext

```

suche_weg_bs(Out):-                %wie TS
    start(Start),                  %wie TS
    finde_bs([[Start,[]]], Out).    %ähnl TS

finde_bs([], _):-
    !,
    fail.

finde_bs(Wege, Out):-              %ähnl TS
    member([Ziel,Bisher], Wege),   %ähnl TS
    klausur(Ziel),                 %wie TS
    append(Bisher, [Ziel], Out).   %wie TS

finde_bs(Wege, Out):-
    schritt(Wege, NeueWege),
    finde_bs(NeueWege, Out).       %ähnl TS

```

⁶Bereinigen meint das Entfernen aller Nachfolgeräume in denen wir, bei dem Zustand, schon waren.

```

schritt([], []):- !.
schritt([[Zustand,Bisher]|Rest], AlleNFZ):-
    findall(T1, weg(Zustand, T1), KopfNFZ),
    append(Bisher, [Zustand], Weg),    %wie TS
    kombiniere(KopfNFZ, Weg, KopfBerNFZ),
    schritt(Rest, RestNFZ),
    append(KopfBerNFZ, RestNFZ, AlleNFZ),
    !.

kombiniere([], _, []):-!.
kombiniere([K|R], D, Out):-
    member(K, D),                %ähnl TS
    kombiniere(R, D, Out),       %ähnl TS
    !.                             %ähnl TS
kombiniere([K|R], D, Out):-
    kombiniere(R, D, Tmp),
    Out=[[K,D]|Tmp],            %ähnl TS
    !.

```

3.7 Tiefen-/Breitensuche Vergleich

3.7.1 Aufgabenstellung

Vergleichen Sie beide Lösungen: nehmen Sie im Kommentar Bezug der Breitensuche-Lösung zu der Tiefensuche-Lösung vor. Benötigt man bei der Breitensuche auch ein eigenes Prädikat, um den kürzesten Weg zu finden?

3.7.2 Entwurf

Wir benötigen bei der **Breitensuche** kein Prädikat⁷ um den kürzesten Weg zu finden, weil der Vorteil der **Breitensuche** ist, dass er erst die kurzen Wege und dann die längeren Wege, in einer festen Ordnung, ausgibt (weil sich durch jeden Schritt die Raumanzahl eines Weges um eins erhöht). Dadurch ist die erste Lösung die uns die **Breitensuche** ausgibt schon gleich (einer) der kleinste(n) Weg(e) zu einem Raum in dem eine Klausur liegt.

Statt dem extra Prädikat könnten wir stattdessen Anfragen mit einem Cut⁸ stellen die das gleiche tun.

Zur Lösung: Kommentare beim Quelltext der **Breitensuche**, die zeigen welche Teile des **Breitensuch**-Algorithmus (und dessen Unterprädikate) äquivalent und/oder ähnlich (auf die neue Datenstruktur bezogen) zu Teilen des **Tiefensuch**-Algorithmus sind.

3.7.3 Quelltext

Siehe Kommentare zum Quelltext von [3.6].

„wie TS“ meint, dass es das Gleiche ist bzw. Äquivalent ist.

„ähnl TS“ meint, dass es im Prinzip das Gleiche ist, auf die neue Datenstruktur bezogen.

⁷Man könnte dennoch eines schreiben, was bis auf den Aufruf von `suche_weg_ts/1` bei `findall/3` identisch ist.

⁸?- `suche_weg_bs(X), !.`

Informationen zur Signatur

	Unterzeichner	EMAILADDRESS=robin.ladiges@haw-hamburg.de, CN=Robin Christopher Ladiges
	Datum/Zeit	Sun Jun 27 00:00:29 CEST 2010
	Austeller-Zertifikat	CN=CAcert Class 3 Root, OU=http://www.CAcert.org, O=CAcert Inc.
	Serien-Nr.	44727
	Methode	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signatur)