Logisch programmeren 2012

Opdrachten Week 8

1 Fibonacci woorden

Een *Fibonacci woord* is een eindig rijtje over het twee-letter alfabet $\{1, 2\}$. De *rangorde* van een Fibonacci woord w is de som van de samenstellende cijfers. Bijvoorbeeld: de rangorde van het woord 1121 is $1 + 1 + 2 + 1 = 5$. We schrijven F_n voor de verzameling Fibonacci woorden van rang n.

De Fibonacci woorden ontlenen hun naam aan het feit dat het aantal woorden in F_n correspondeert met de Fibonacci reeks (1,1,2,3,5,8,...). Figuur 1 geeft de Fibonacci woorden voor rangorde $0 \leq n \leq 5$. Het symbool ε staat voor het lege rijtje — het enige rijtje van rangorde 0.

De bron voor dit binaire systeem is de klassieke Indische metriek. Het symbool 1 staat voor een korte lettergreep, het symbool 2 voor een lange. Een korte lettergreep is één maatslag lang, een lange twee. F_n geeft dus alle mogelijke ritmes voor n maatslagen.[1]

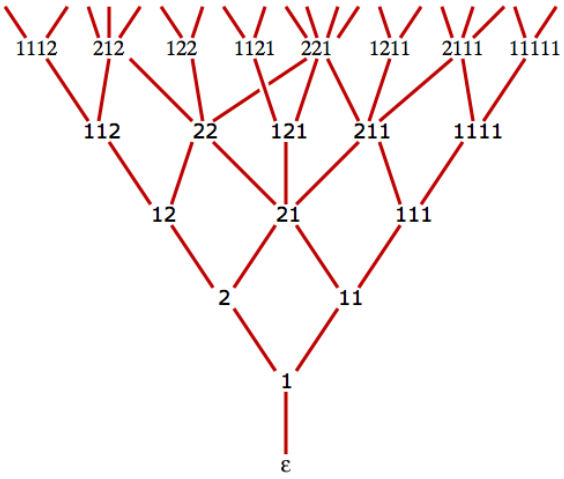
1.1 Opgave

In de opgaven die volgen schrijven we een Fibonacci woord als een rijtje van 1-en en 2-en. Definieer een predicaat `f/2` waarmee je F_n genereert: `f(N,Woord)` slaagt als `Woord` een Fibonacci woord is van rang `N`. Bijvoorbeeld:

```
?- f(3,L).
L = [1, 1, 1] ; L = [1, 2] ; L = [2, 1] ;
false.
```

Giet je definitie voor `f/2` in de vorm van een DCG voor een predicaat `f/4`. De eerste twee argumenten staan voor de rang, uitgedrukt als verschilgetal, de laatste twee voor het woord, uitgedrukt als verschillijst. Hieronder de wrapper; `d2s/3` (een oude bekende) zet een decimaal getal om in een verschilgetal.

[1] Rachel Wells Hall 'Math for poets and drummers' is een aardig achtergrondartikel.



Figuur 1: Diagram van F_n voor $0 \leq n \leq 5$

```
f(N,F) :- d2s(N,P,0), f(P,0,F,[]). % wrapper

% d2s/3: decimaal naar verschilgetal

d2s(N,s(X),Y) :- N>0, N1 is N-1, d2s(N1,X,Y).
d2s(0,X,X).
```

Denk erom dat de DCG pijl `-->` de verschillijst argumenten automatisch aanvult. Zorg ervoor dat je DCG niet linksrecursief is!

1.2 Opgave

Het *hoofd* van een Fibonacci woord is het langste prefix bestaande uit 2-en. In het volgende voorbeeld is het hoofd onderstreept: 2212.

Gegeven een Fibonacci woord w bereken je de *opvolgers* van w in de tralie van Figuur 1 aan de hand van een van de volgende operaties: (i) voeg links een 1 toe aan w; (ii) vervang het eerste voorkomen van 1 in w door een 2; (iii) voeg een 1 in waar dat kan tussen twee 2's in het hoofd van w of onmiddellijk na de laatste 2 van het hoofd.

De *voorgangers* van w bereken je aan de hand van een van de volgende operaties: (i) verwijder het eerste voorkomen van 1 in w; (ii) vervang een voorkomen van 2 in het hoofd van w door een 1.

Definieer een predicaat `f_desc/2` voor de voorganger-opvolger relatie. Gegeven een Fibonacci woord `F` slaagt `f_desc(F,F1)` als `F1` een opvolger is van `F`; gegeven een Fibonacci woord `F1` slaagt `f_desc(F,F1)` als `F` een voorganger is van `F`. Test je definitie aan de hand van Fig 1.

2 Zoeken: breadth first

Hieronder het generieke *breadth first* zoekalgoritme, zoals besproken in het hoorcollege. De predicaten voor het manipuleren van een wachtrij (*queue*) vind je in de Week 8 startcode.

```
bf(Start, Final, Path) :-  % wrapper
    empty_queue(Empty),
    bf(Start, Final, Empty, [], Path).

bf(S, S, _, RevPath, Path) :-
    solution(S),
    reverse([S|RevPath], Path).

bf(S0, S, Queue0, Path0, Path) :-
    findall(n(S1,[S0|Path0]), s(S0,S1), Children),
    !,
    queue_append(Queue0, Children, Queue1),  % Children achteraan aansluiten
    queue_cons(n(S2, Path2), Queue, Queue1),   % voorste is nu aan de beurt
    bf(S2, S, Queue, Path2, Path).
```

2.1 Opgave

Vul de probleemspecifieke informatie in voor de Fibonacci woorden van de vorige opgaven. Definieer de algemene opvolgersrelatie `s/2` in termen van `f_desc/2`. Bouw de conditie in dat in `s(Knoop,Opvolger)` de rang van `Knoop` kleiner is dan de rang van de gezochte oplossing. De gewenste oplossing declareer je met het dynamische `solution/1`. Hieronder een testwrapper.

```
bf_test(Sol) :- retractall(solution(_)),
    assert(solution(Sol)),
    bf([],Sol,Path),
    write(Path),nl,
    fail.
bf_test(_).
```

En een voorbeeldaanroep.

```
?- f_test([2,2]).
[[],[1],[1,1],[2,1],[2,2]]
[[],[1],[2],[1,2],[2,2]]
[[],[1],[2],[2,1],[2,2]]
true.
```

2.2 Opgave

Het `bf/5` programma herberekent vaak de `s/2` relatie voor knopen die al eerder zijn bezocht. Dat kan je verhelpen met de memo techniek. Het predicaat `s_memo/2` hieronder berekent `s/2` en slaat oplossingen op als feiten voor een dynamisch predicaat `found/2`. `If->Then;Else` is het ingebouwde if-then-else construct.

```
s_memo(S,S1) :- s(S,S1),
    store_solution(S,S1).

store_solution(S,S1) :-
    found(S,S1)  % al gevonden
    -> true      % doe niets
    ;
    assert(found(S,S1)).
```

Pas je code aan om met `s_memo/2` te werken. Noem de nieuwe predicaten `bf_memo/3` en `bf_memo/5`. Bij het expanderen van een knoop `S` laat je `findall` de kinderen van `S` simpelweg opzoeken als er al feiten voor `found(S,_)` zijn. Als er nog geen feiten voor `found(S,_)` zijn, laat je `findall` de kinderen aanmaken met `s_memo/2`. Hieronder de testwrapper.

```
bf_memotest(Sol) :- retractall(found(_,_)),
    retractall(solution(_)),
    assert(solution(Sol)),
    bf_memo([],Sol,Path),
    write(Path),nl,
    fail.
bf_memotest(_).
```

3 Zoeken: best first

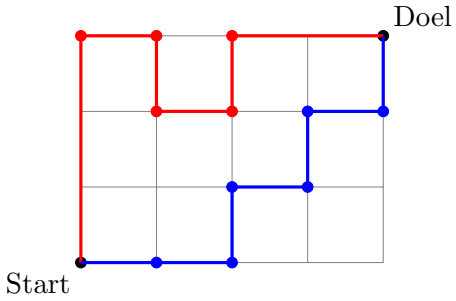
Hieronder het generieke A* best first algoritme, zoals je het vindt in de startcode voor Week 8.

```
bestf(Start, Goal, Cost, Path) :-
    empty_heap(Heap),
    bestf(Start, Goal, Heap, 0, Cost, [], Path).

bestf(S, S, _, Cost, Cost, RevPath, Path) :-
    solution(S),
    reverse([S|RevPath], Path).

bestf(S0, Goal, Heap0, G0, Cost, Path0, Path) :-
    findall(F-n(S1,G,[C,S0|Path0]), % F waarde is de prioriteit van S1
        (
        s_cost(S0,S1,C), % overgangen met een kostprijs C
        G is C+G0,
        h(S1,Goal,H),
        F is G+H
        ),
        Children),
    list_to_heap(Children,ChildrenHeap),
    merge_heaps(ChildrenHeap,Heap0,Heap1),
    get_from_heap(Heap1,_,n(S2, G2, Path2), Heap),
    bestf(S2, Goal, Heap, G2, Cost, Path2, Path).
```

Probleemspecifiek zijn `s_cost/3` en `h/3`.

3.1 Opdracht

In de graaf hieronder zie je twee paden (rood versus blauw) van Start naar Doel. Voor de Prolog codering van de graaf schrijven we knopen als coördinatenparen `X/Y`. Start is dan `0/0`, Doel is `4/3`. Verbindingen tussen knopen coderen we als feiten `s(From,To)`, bijvoorbeeld `s(0/0,0/3)` voor de eerste overgang op het rode pad. Geef de volledige set van `s/2` feiten (zes overgangen voor het rode pad; zeven voor het blauwe).



Voor gebruik met het best first algoritme willen we de overgangen van een kostprijs voorzien. Definieer daartoe een predicaat `manhattan/3` waarmee je de 'Manhattan distance' tussen twee punten uitrekent: de kortste route volgens het Manhattan stratenplan. Bijvoorbeeld:

```
?- manhattan(0/0,4/3,Dist).
Dist = 7.
```

Je kan de ingebouwde functie `abs/1` gebruiken om de absolute waarde van een getal te berekenen: `X is abs(-3)` slaagt met `X=3`.

Definieer vervolgens `s_cost/3` in termen van `s/2` en `manhattan/3`. `s_cost/3(S,S1,Cost)` slaagt als er een verbinding is tussen `S` en `S1` met de Manhattan afstand als `Cost`.

Rest nog de definitie voor de heuristische functie `h/3`, waarmee je de kortste afstand tussen een gegeven knoop en het doel inschat. Voor ons kortste pad zoekprobleem is een goede invulling voor `h/3` ... de Manhattan distance.

Schrijf de definitie voor `h/3` op en test de werking van `bestf/4` met de aanroep hieronder. Geef een korte commentaar waarin je de volgorde van de gevonden oplossingen toelicht.

```
?- bestf(0/0,4/3,Cost,Path),write((Cost,Path)),nl,fail.
```

Opdracht

Het `bf` algoritme kan je zien als een gedegenereerde vorm van best first zoeken waarbij de heuristische functie `h/3` triviaal is.

Pas de code voor `s_cost/3` en `h/3` op zulke manier aan dat het best first algoritme hierboven het pad met het kleinste aanstal *stappen* als eerste oplossing verkiest: het rode pad, in de voorbeeldgraaf.

□