Prolog. Gekwoteerde zitting.

Pieter Bonte

24/3/2023, 13:30 - 16:30

Er zijn drie vragen. Het symbool (*) duidt een (deel van een) vraag aan die je als iets moeilijker kan beschouwen en eventueel kan uitstellen voor het geval er op het einde tijd over is. Je kan de documentatie bij SWI-Prolog (`https://www.swi-prolog.org`) gebruiken, voorbeeldoplossingen, eigen oplossingen en de slides. Andere resources op het internet zijn niet toegelaten. E-mail en andere communicatie-toepassingen worden uitgeschakeld.

Als je klaar bent verwittig je de toezichter. Die zal je toelaten om je e-mail op te starten. Zend dan je antwoorden naar `pieter.bonte@kuleuven.be`

1 **Pad** (1/3)

Beschouw een vierkant bord met een startpunt en een eindpunt, zoals in Figuur 1 (a). Het startpunt zal altijd (1, 1) zijn. Het eindpunt wordt gegeven door een feit als *goto(1,4,final)*, en de grootte van het bord wordt gegeven door een feit size/1. Er zijn ook feiten voor goto/3 die je toelaten te verplaatsen over het bord, zoals in het volgende voorbeeld:

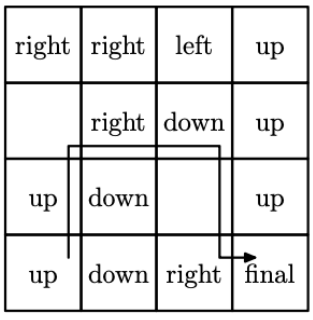
```
goto(1,1,up).
goto(1,3,right).
goto(3,3,down).
goto(3,2,left).
```

De bovenstaande feiten betekenen: als je positie (1, 1) is, moet je omhoog; als je positie (1, 3) is, ga dan naar rechts, enzovoort. Door deze goto/3-instructies te volgen, is het gemakkelijk om een pad te construeren van het beginpunt naar het eindpunt. Te gemakkelijk zelfs.

Gelukkig heeft een gemene trol wat informatie van het bord verwijderd: hij heeft enkele goto/3-feiten weggehaald, zodat het pad niet langer volledig weergegeven wordt. Toch was het niet zo gemeen, aangezien geen twee aangrenzende vakjes in het pad hun informatie verwijderd is, noch het eindpunt, en altijd twee buren

right	right	left	up
	right	down	up
up	down		up
up	down	right	final

(a) Given board



(b) Same board with path

Figuur 1: Voorbeeld

van een verwijderd vakje tot het juiste pad behoren (diagonaal rakende vakjes worden niet als buren beschouwd). Het is iets moeilijker geworden, maar je kunt het pad nog steeds gemakkelijk vinden. Een tweede, veel gemenere trol heeft echter informatie geïntroduceerd voor elk vakje dat niet op het pad ligt, en wel op zo'n manier dat als je deze informatie volgt, je ofwel in cirkels rondrent of van het bord valt. Figuur 1 (a) toont een verminkt bord en komt overeen met de volgende feiten:

```
goto(1,4,right). goto(2,4,right). goto(3,4,left).  goto(4,4,up).
goto(2,3,right). goto(3,3,down).  goto(4,3,up).
goto(1,2,up).    goto(2,2,down).  goto(4,2,up).
goto(1,1,up).    goto(2,1,down).  goto(3,1,right). goto(4,1,final).
size(4).
```

Gevraagd is

1. *(0.25/3)* Schrijf een predicaat *volgende(+Direction,+xOrig, +yOrig, -xNew, -yNew)* dat de volgende locatie op het bord berekent, op basis van de richting (up, down, left, right):

   ```
   ?- volgende(up, 1,1, XNew,YNew).
      XNew = 1,
      YNew = 2.

   ?- volgende(left, 3,3, XNew,YNew).
      XNew = 2,
      YNew = 3.
   ```

   Tip: *Hierbij hoef je geen rekening te houden met de goto/3 feiten.*

2. *(0.25/3)* Schrijf een predicaat *bad_move(+X,+Y,+Visited)* dat berekend als een volgende zet naar (X, Y) een geldige zet is. Een zet is geldig indien deze er voor zorgt dat je niet van het bord valt en je niet in cirkels loopt. Waarbij Visited de reeds bezochte locaties bevat.

   ```
   ?- bad_move(1,4,[]).
   false.

   ?- bad_move(0,1,[]).
   true .

   ?- bad_move(5,4,[]).
   true .

   ?- bad_move(1,4,[(1,1),(1,2),(1,3),(1,4)]).
   true .
   ```

   Waarbij we uitgaan van een bord zoals in het voorbeeld met *size(4)*.

3. (*) *(0.5/3)* Schrijf een predicaat *pad(-Path)* dat het juiste pad zoekt, beginnend bij (1,1), eindigend in het laatste punt (*final*) en geordend langs het pad. Gebruik hierbij predicaten *volgende/5* en *bad_move/3*.

   In het bovenstaande voorbeeld zouden we volgende oplossing krijgen:

   ```
   ?- path(Path).
   Path = [(1,1),(1,2),(1,3),(2,3),(3,3),(3,2),(3,1),(4,1)]
   ```

   Dit pad komt overeen met Figuur 1 (b).

Tip: *Indien er geen goto/3-feit aanwezig is, dienen alle vier de richtingen worden geprobeerd.*

2 Loops (1/3)

Je krijgt een gerichte graaf in de vorm van *arrow/2* feiten zoals in:

```
arrow(a,b).
arrow(b,c).
arrow(c,c).
arrow(a,d).
arrow(d,a).
```

Gevraagd wordt om alle minimale cycli in de graaf te zoeken. Een cyclus wordt weergegeven door een lijst met alle knooppunten in de cyclus in de volgorde waarin ze zijn gekoppeld door de bogen, en begint en eindigt met hetzelfde knooppunt. Een minimale cyclus bevat geen andere cyclus. Elke minimale cyclus van de graaf moet precies één keer worden gegeven. Voor het voorbeeld zou je programma een van de volgende lijsten als oplossing kunnen geven:

```
[[c,c],[a,d,a]]
[[c,c],[d,a,d]]
[[a,d,a],[c,c]]
[[d,a,d],[c,c]]
```

De volgorde in de lijst met cycli is irrelevant, en dat geldt ook voor het werkelijke startknooppunt van een cyclus.

Gevraagd is

1. *(0.33/3)* Schrijf een predicaat *delete_node(+ArrowList,+Node,-Results)* dat alle bogen verwijdert uit een lijst met bogen (*ArrowList*) voor een bepaald knooppunt (*Node*). Lees de **Tip** onderaan om te begrijpen waarom dit nodig is.

   ```
   ?- delete_node(AllArrows, a, R).
   AllArrows = [arrow(a, b), arrow(b, c), arrow(c, c), arrow(a, d), arrow(d, a)],
   R = [arrow(b, c), arrow(c, c)].
   ```

2. *(0.33/3)* Schrijf een predicaat *path(+Start,+Current,+ArrowList,+Visited,-Loop)* dat een cycle zoekt, startende van een gegeven knooppunt. Met:

   - *Start*: het startpunt van de cycle.
   - *Current*: het huidige knooppunt dat wordt bezocht. (Gebruikt voor recursie.)
   - *ArrowList*: de lijst van bogen in de graaf.
   - *Visited*: de lijst van reeds bezochte knooppunten.
   - *Loop*: lijst van knooppunten die een cycle vormen (indien een gevonden wordt).

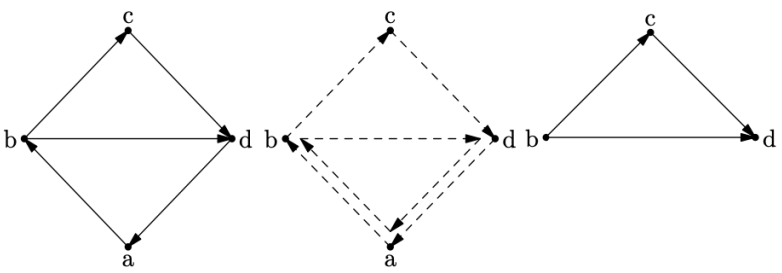
   ```
   ?- path(a,a,AllArrows,[],L).
   L = [d, a]
   ```

3. *(0.33/3)* Schrijf een predicaat *findloops(+ArrowList, +AccLoops, -AllLoops)* dat alle minimale cycli in de graaf zoekt. Gebruik hierbij de procedure die uitlegd staat in de **Tip** hieronder.

   ```
   ?-findloops(AllArrows,[],L).
   L = [[c, c], [a, d, a]].
   ```

   Je kan dit predicaat als volgt gebruiken:

   ```
   loops(Loops) :-
              findall(arrow(A,B),arrow(A,B),AllArrows),
              findloops(AllArrows,[],Loops) .
   ```



(a) Initial Graph

(b) Loops involving node **a**

(c) The graph after removing arrows from/to node **a**

Figuur 2: Voorbeeld

Tip: *Een minimale cyclus is een cyclus die niet meer dan één keer hetzelfde knooppunt bevat. Om alle minimale cycli te verzamelen, kunnen we daarom een willekeurig knooppunt selecteren en de minimale cycli verzamelen die bij dat knooppunt beginnen. Vervolgens verwijderen we het geselecteerde knooppunt uit de graaf en starten we de procedure opnieuw. Figuur 2(a) toont een graaf. Eerst wordt knooppunt* ***a*** *geselecteerd en worden de minimale cycli ervan geconstrueerd. Er zijn er twee en ze worden aangegeven in Figuur 2(b). Vervolgens worden de pijlen die knooppunt* ***a*** *betreffen verwijderd, wat leidt tot de graaf in Figuur 2(c). Op dat punt wordt de recursieve aanroep van findloops/3 in het programma uitgevoerd.*

3 Domino (1/3)

Bereken een manier om een gegeven set dominostenen te ordenen zodat ze een correcte dominoketting vormen. Dit betekent dat de stippen op één helft van een steen overeenkomen met de stippen op de aangrenzende helft van een naburige steen. Daarnaast moeten de stippen op de uiteinden van de ketting (de eerste en laatste steen) ook met elkaar overeenkomen.

Bijvoorbeeld, gegeven de stenen [2|1], [2|3] en [1|3], kan je een volgorde berekenen zoals:

1. [1|2] [2|3] [3|1]
2. [3|2] [2|1] [1|3]
3. [1|3] [3|2] [2|1]
4. enzovoort,

Let er op dat het eerste en laatste getal hetzelfde zijn. Voor de stenen [1|2], [4|1] en [2|3] is er geen geldige ketting mogelijk, want bijvoorbeeld [4|1] [1|2] [2|3] voldoet niet aan de voorwaarde dat het eerste en laatste getal gelijk moeten zijn (4 ≠3).

Gevraagd is

1. *(0.5/3)* Een predicaat *solve_domino(+Dominos,-Chain)* dat een geldige dominoketting (*Chain*) genereert voor de dominoblokken in *Dominos*.

```
?- solve_domino([(2,1), (2,3), (1,3)], Chain).
Chain = [(1, 3), (3, 2), (2, 1)] ;
Chain = [(3, 2), (2, 1), (1, 3)] ;
Chain = [(3, 1), (1, 2), (2, 3)] ;
false.

?- solve_domino([(1,2), (4,1), (2,3)], Chain).
false.
```

Tip: *Je kan het predicaat last/2 gebruiken om eenvoudig het laatste element van een lijst op te vragen.*

2. *(0.5/3)* Een predicaat *longest_domino_chain(+Dominos, -Chain)* dat de langst mogelijke ketting gegenereert gegeven een lijst van dominoblokken, maar waarbij niet alle dominoblokjes moeten gebruikt worden. Dezelfde regels als daarnet gelden:

   - Aangrenzende dominostenen moeten aan één kant overeenkomen.
   - De eerste en laatste dominosteen moeten ook overeenkomen (de ketting moet rond zijn).
   - Je mag nu ook enkele dominostenen uit de oplossing weglaten als dit leidt tot een langere geldige ketting.

   Bijvoorbeeld:

```
?- longest_domino_chain([(1,2), (2,3), (3,1),(4,5)], Chain).
Chain = [(1, 2), (2, 3), (3, 1)].

?longest_domino_chain([(2,1), (2,3), (3,4),(4,1),(5,2)], Chain).
Chain = [(3, 4), (4, 1), (1, 2), (2, 3)].
```

   Tip: *Je kan onderstaande code gebruiken om sublijsten van dominos te genereren.*

```
subset_own([], []).
subset_own([E|Tail], [E|NTail]):-
   subset_own(Tail, NTail).
subset_own([_|Tail], NTail):-
   subset_own(Tail, NTail).
```

   Bijvoorbeeld:

```
?- subset_own([(1,2),(2,3),(3,1)],R).
R = [(1, 2), (2, 3), (3, 1)] ;
R = [(1, 2), (2, 3)] ;
R = [(1, 2), (3, 1)] ;
R = [(1, 2)] ;
R = [(2, 3), (3, 1)] ;
R = [(2, 3)] ;
R = [(3, 1)] ;
R = [].
```