Homework #5

Deadline: Tuesday, 14 October 2014, 12:00 noon

Note: Questions 1 and 2 both require you to define a predicate called `goal/1`. As you should submit just a single file, this requires some extra care. Include the following line at the top of your file to avoid warnings about clauses for the same predicate not being together:

```
:- discontiguous(goal/1).
```

As you will use completely different representations of states for your two solutions, you should be able to ensure that the two programs do not interfere with each other. There will however be a very small negative effect on performance, which we can safely ignore here.

Question 1 (4 points)

We have previously seen how to write a computer program that can compete successfully in the *letters game* familiar from the famous television game show *Countdown* (also known as *Cijfers en Letters*). We now also want to crack the *numbers game.*

> In the numbers game you are given a target number (a positive integer), such as 297, as well as a list of six further numbers (all of them positive integers as well), such as 100, 75, 50, 25, 6, and 4. Your task is to construct an arithmetic expression from these six numbers that comes as close as possible to the target number. Here we will simplify the problem a little bit and only look for arithmetic expressions that evaluate to *exactly* the target number. The arithmetic operations you are allowed to use are addition, subtraction, multiplication, and division (although division is only allowed in case the result of the division in question is itself an integer). You do not need to use all the numbers. Each number may be used at most as many times as it occurs in the given list of six. One of several valid solutions for our example is $297 = 50 \cdot 6 - 75 / 25$.

Your ultimate task is to write a Prolog predicate called `solve/3` to compete in the numbers game. When given the target number in the first argument position and a list of six further numbers in the second argument position, it should instantiate the variable given in the third argument position with a suitable arithmetic expression. Examples:

```
?- solve(297, [100,75,50,25,6,4], Solution).
Solution = 50*6-75/25
Yes

?- solve(493, [1,3,10,7,6,4], Solution).
Solution = (1+6)*10*7+3
Yes

?- solve(505, [8,1,8,10,4,7], Solution).
Solution = (8+10)*4*7+1
Yes
```

Formulate the game as a search problem and apply a suitable (uninformed) search algorithm. You may follow the steps outlined below or you may implement your own solution—provided of course it is clearly documented.

(a) You first need to think of a good representation of states. Let's use terms of the form `Target:Terms`, where `Target` is the target number and `Terms` is a list of arithmetic expressions you are currently allowed to use (note that the colon `:` is simply a built-in infix operator that allows us to combine two terms, but that has not special meaning beyond that). In our first example, the initial state would simply be `297:[100,75,50,25,6,4]`, with the only arithmetic expressions available to us being the six numbers provided by the user. Also later states can be represented in this form, as we shall see next.

(b) Write a suitable `move/2` predicate. It should transform a given state into a new state by applying one of the four possible arithmetic operations. For example, if I apply the operation of addition to the first two numbers in state `297:[100,75,50,25,6,4]`, then I cannot use `100` and `75` anymore, but I can from now on use the expression `100+75`, so my new state becomes `297:[100+75,50,25,6,4]`. If I then apply the operation of division to `100+75` and `25`, my new state becomes `297:[(100+75)/25,50,6,4]`, and so forth. Keep in mind that division is only allowed in case the result is an integer. Also keep in mind that you cannot divide by zero.

(c) Write a suitable `goal/1` predicate. It should check whether the list of arithmetic terms in a given state includes a term that evaluates to the target number.

(d) Now implement `solve/3` as specified above. It should construct the initial state from the user input, apply a search algorithm to this initial state, and then return the solution found (rather than the full path to the goal state) to the user. For the search algorithm I recommend iterative deepening.

Finally (whether you followed the suggestions above or implemented your own approach), report the solutions found by your program for at least three different examples (show the queries and the results produced) and answer the following questions: What happens when you use other search algorithms than iterative deepening? Does breadth-first search work as well? Can you explain why? How about depth-first search? Does it make a difference whether or not you use the cycle-free version of depth-first search? Why? Can you explain why depth-first search tends to produce solutions involving more of the input numbers than iterative deepening (as in the following example)?

```
?- solve(297, [100,75,50,25,6,4], Solution).  % using iterative deepening
Solution = 50*6-75/25
Yes

?- solve(297, [100,75,50,25,6,4], Solution).  % using depth-first search
Solution = 50*6-((100+75)/25-4)
Yes
```

Question 2 (5 points)

Write a program that uses the A* algorithm to find the shortest route between two given cities in the Netherlands. This exercise leaves you a certain amount of freedom for how you want to write your program, but you should nevertheless abide by the following instructions:

(a) Your main predicate should be called `route/4`. The first two arguments should be the start and end points of the route; the third argument should be the route proposed by the system; and the fourth argument should be the distance travelled (in km):

```
?- route(amsterdam, maastricht, Route, Distance).
Route = [amsterdam, utrecht, hertogenbosch, eindhoven, maastricht]
Distance = 228
Yes
```

(b) Your database should include at least ten cities in the Netherlands. For each pair of cities that are directly connected by a major road, record the distance between the two cities when following that road.

(c) In addition, for each city record appropriate x/y-coordinates that can be used to compute the straight-line distance between any two cities (this number can be used for the heuristic function in the A* algorithm).

(d) Use the implementation of the A* algorithm exactly as given on the lecture slides, without making any changes.

In your submission, show what answers your program gives for the routes Breda-Haarlem and Amsterdam-Groningen (show the first three answers for each query). You are welcome to cooperate as far as the construction of the city database is concerned (it would be nice to be able to run tests on a large database).

Question 3 (1 point)

Let h_1, h_2 and h_3 be three admissible heuristic functions for A* for some specific search problem. Define a new heuristic function that (*i*) is also admissible and that (*ii*) will guide the search at least as well as the best of the three given functions alone. Justify your answer.