Logisch Programmeren en Zoektechnieken: Tentamen B **ANSWERS**
24 October 2014

Instructions: This exam paper consists of four questions. You may answer them in Dutch or in English. Only write in the answer booklets provided and mark each booklet with your name and student number. Give short and precise answers. Write clearly. The maximum time allowed is 120 minutes.

Question 1

(a) Translate the following Prolog program, which includes a query, into a set of formulas in first-order predicate logic: 10 marks

```
edge(a, b).
edge(b, c).
connected(X, Y) :- edge(X, Y).
connected(X, Y) :- edge(X, Z), connected(Z, Y).
accessible(X) :- edge(Y, X).
:- connected(a, X).
```

Answer:

$\{\ \mathit{edge}(a, b),$
$\mathit{edge}(b, c),$
$\forall x.\forall y.(\mathit{edge}(x, y) \rightarrow \mathit{connected}(x, y)),$
$\forall x.\forall y.\forall z.(\mathit{edge}(x, z) \wedge \mathit{connected}(z, y) \rightarrow \mathit{connected}(x, y))$
$\forall x.\forall y.(\mathit{edge}(y, x) \rightarrow \mathit{accessible}(x)),$
$\forall x.(\mathit{connected}(a, x) \rightarrow \bot)\ \}$

(b) In the context of propositional logic, define what a *Horn formula* is. Why are Horn formulas relevant to the study of logic programming? 5 marks

Answer:

Horn formulas are disjunctions of literals with at most one positive literal (saying that a Horn formula is a conjunction of such disjunctions is also an acceptable answer). Horn formulas are relevant to logic programming for two reasons: (a) basic Prolog rules, facts, and queries (without variables) all translate to (propositional) Horn formulas and (b) reasoning about Horn formulas is much more efficient than reasoning about general formulas.

(c) Explain the difference between *matching* in Prolog and *sound unification* in logic. 5 marks

Answer:

See Lecture Notes.

(d) For each of the following claims, state whether it is true or false. 5 marks

(i) The function f defined via $f(n) = n^2 + 2n + 1$ is in $O(n^2)$.

Answer:

True.

(*i*) The function f defined via $f(n) = n^2 + 2n + 1$ is in $O(n^3)$.

Answer:

True.

(*iii*) The function f defined via $f(n) = n^3 - n$ is in $O(n^2)$.

Answer:

False.

(*iv*) Every function in $O(2^n)$ grows faster than every function in $O(n^2)$.

Answer:

False.

(*v*) Every function in $O(n \log n)$ is also in $O(n^2)$.

Answer:

True.

Question 2

The purpose of this question is to implement and analyse a sorting algorithm known as *insertion-sort*. The algorithm works by recursively removing the first element from the unsorted input list and inserting that element at the appropriate position in the (sorted) output list (which is initially empty). Example:

```
Input:   28  44 19 17 30     Output:
Input:  [28] 44 19 17 30     Output:  28
Input:  [44] 19 17 30        Output:  28 44
Input:  [19] 17 30           Output:  19 28 44
Input:  [17] 30              Output:  17 19 28 44
Input:  [30]                 Output:  17 19 28 30 44
```

Answer the following questions:

(a) Write a Prolog predicate to insert a given element at the appropriate position in a given list that is already ordered with respect to a given ordering relation. Example: $\boxed{\text{10 marks}}$

```
?- insert(<, 30, [17, 19, 28, 44], List).
List = [17, 19, 28, 30, 44]
Yes
```

Make sure there are no incorrect alternative answers after enforced backtracking. You may assume that the following predicate, familiar from the course, is available. It can be used to to check whether two given terms `A` and `B` are ordered with respect to the ordering relation `Rel`:

```
check(Rel, A, B) :-
  Goal =.. [Rel,A,B],
  call(Goal).
```

Answer:

```
insert(_, X, [], [X]).
insert(Rel, X, [H|T], [X,H|T]) :- check(Rel, X, H), !.
insert(Rel, X, [H|T], [H|Result]) :- insert(Rel, X, T, Result).
```

(b) Now implement the *insertion-sort* algorithm. Examples: 10 marks

```
?- insertsort(<, [28, 44, 19, 17, 30], List).
List = [17, 19, 28, 30, 44]
Yes

?- insertsort(@<, [lion, zebra, elephant, tiger, gnu], List).
List = [elephant, gnu, lion, tiger, zebra]
Yes
```

Answer:

```
insertsort(Rel, List, Result) :- insertsort(Rel, List, [], Result).

insertsort(_, [], List, List).

insertsort(Rel, [H|T], List, Result) :-
  insert(Rel, H, List, IntermediateResult),
  insertsort(Rel, T, IntermediateResult, Result).
```

(c) Describe a special class of input lists for which *insertion-sort* requires a number of comparison operations that is quadratic in the length of the input list. Briefly justify your answer. 5 marks

Answer:
In case the input list is already ordered, *insertion-sort* requires a quadratic number of comparison operations. To see this, consider that in the kth round there are $k-1$ comparison operations, because the kth number to be inserted will end up at the very end of the $k-1$ numbers already sorted. The exact number of comparison operations thus is $0+1+2+\cdots+(n-1) = \frac{n(n-1)}{2} \in O(n^2)$.

Question 3

(a) For an uninformed search algorithm, define what it means to be *complete* and define what it means to be *optimal*. Write one sentence for each of these two definitions. 5 marks

Answer:

See lecture slides.

(b) Consider a search problem with 1000 possible states for which every state has at most 10 possible follow-up states. Is *cycle-free depth-first search* complete for this problem? Is it optimal? Briefly justify your answers. 5 marks

Answer:
It is complete: as cycles on a branch are excluded and as the number of branches is finite, every node will get visited eventually. It is not optimal: for example, there may be two solutions and the longer one of them may be on the leftmost branch.

(c) Consider the implementation of the *magic search* algorithm shown below. As with all the uninformed search algorithms discussed in class, it assumes that the predicates `move/2` and `goal/1` are provided by the application developer. Give a short high-level explanation of how magic search works. Then describe briefly what each of the clauses making up the program does (one sentence per clause is sufficient). 10 marks

Hint: The predicate `subroutine/4` is an implementation of one of the search algorithms discussed in class.

Answer:

This algorithm is like iterative deepening, except that in every round the depth bound is increased by 5 rather than just by 1.

The predicate `subroutine/4` is an implementation of depth-bounded depth-first search (almost exactly as shown in class). The first clause of this predicate is the base case: if the current state is a goal state, we return the reversal of the list of nodes visited on the branch leading to this goal node. The second clause is the recursive rule: if the depth bound has not yet been reached (`N > 0`), we choose a follow-up node (using `move/2`), check whether it has not been visited on the current branch before (using the negation of `member/2`), and if so continue searching recursively from the expanded branch with the depth bound lowered by 1.

The first clause of the main predicate is the user interface: initiate search with depth bound 0. The second clause is the base case: use depth-bounded depth-first search with the current bound. If this does not succeed, the final clause is used: increment the depth bound by 5 and recursively call magic search with this increased bound.

Note that besides the increment of 5 rather than 1 in each round, another difference to iterative deepening as presented in class is that magic search will also return solution paths that are shorter than the current depth bound. It is not required to note this difference to obtain full marks.

(d) For each of the following three search algorithms, state one advantage that *magic search*, as implemented below, has over that search algorithm: 5 marks

- cycle-free depth-first search

  Answer:

  Magic search will sometimes return shorter solutions and never longer ones (though it still is not optimal).

- breadth-first search

  Answer:

  Magic search has lower space complexity (linear *vs.* exponential).

- iterative deepening

  Answer:

  Magic search can be expected to run faster than iterative deepening for problems with solution paths that are not very short, as fewer rounds are required to reach the required depth (but note that in terms of abstract worst-case time complexity, the two algorithms are the same).

Magic Search:

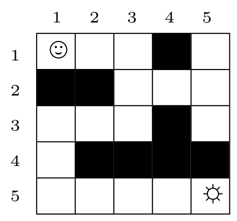
```
magicsearch(State, Path) :-
  magicsearch(0, State, Path).
```

```
subroutine(_, Seen, State, Path) :-
  goal(State),
magicsearch(N, State, Path) :-
  subroutine(N, [State], State, Path).

magicsearch(N, State, Path) :-
  N1 is N + 5,
  magicsearch(N1, State, Path).
```

```
  reverse(Seen, Path).

subroutine(N, Seen, State, Path) :-
  N > 0,
  move(State, Next),
  \+ member(Next, Seen),
  N1 is N - 1,
  subroutine(N1, [Next|Seen], Next, Path).
```

Question 4

This question is about using A* to find a path in a labyrinth modelled as an $n \times n$-grid of white and black cells. Suppose a robot starts in the upper lefthand corner and wants to move to the lower righthand corner. It can move horizontally and vertically (but not diagonally), and it can only move via white cells. In Prolog, we can represent such a grid as a list of lists of the atoms `w` (for *white*) and `b` (for *black*). We store the grid given to us using the predicate `grid/1`. Here is an example (for $n = 5$):



```
grid([ [w, w, w, b, w],
       [b ,b, w, w, w],
       [w, w, w, b, w],
       [w, b, b, b, b],
       [w, w, w, w, w] ]).
```

Your task is to model this problem using the state-space representation introduced in class, so that we can find the shortest path using our implementation of A*:

```
?- solve_astar(1/1, Path/NumberOfMoves).
Path = [1/1, 2/1, 3/1, 3/2, 3/3, 2/3, 1/3, 1/4, 1/5, 2/5, 3/5, 4/5, 5/5],
NumberOfMoves = 12
Yes
```

Your program should work for different (square) grids of any size n. But you may assume that cell `1/1` (the upper lefthand corner) is always white. Answer the following questions:

(a) We represent cells as terms of the form `X/Y` and we represent the current state as the cell at which the robot currently is located. Implement a predicate `move/3` to—upon enforced backtracking—return all legal follow-up states for a given state, together with the cost of that move. This cost should always be 1. Examples: 10 marks

```
?- move(1/3, NextState, Cost).      ?- move(5/1, NextState, Cost).
NextState = 2/3, Cost = 1 ;         NextState = 5/2, Cost = 1 ;
NextState = 1/4, Cost = 1 ;         No
No
```

Hint: Start by implementing a predicate `white/1` to check whether a given cell `X/Y` is white. You may want to use the built-in predicate `nth1/3`, which allows you to retrieve the element placed at a given position in a given list.

Answer:
```
white(X/Y) :- grid(Grid), nth1(Y, Grid, Row), nth1(X, Row, w).
move(X/Y, NewX/Y, 1) :- NewX is X - 1, white(NewX/Y). % move left
move(X/Y, NewX/Y, 1) :- NewX is X + 1, white(NewX/Y). % move right
move(X/Y, X/NewY, 1) :- NewY is Y - 1, white(X/NewY). % move up
move(X/Y, X/NewY, 1) :- NewY is Y + 1, white(X/NewY). % move down
```

(b) Implement a predicate `goal/1` that succeeds if the state given corresponds to the cell in the lower righthand corner. Keep in mind that you first have to find out how large the grid is (it need not always have size 5). Examples (for the above grid): 5 marks

```
?- goal(5/5).      ?- goal(4/4).      ?- goal(2/3).
Yes                No                 No
```

Answer:
```
gridsize(Size) :- grid(Grid), length(Grid, Size).
goal(Size/Size) :- gridsize(Size).
```

(c) Define what it means for a heuristic function for A* to be *admissible*. Then propose a suitable (non-trivial) heuristic function for our problem that is admissible. Finally, implement your heuristic function as a predicate `estimate/2`. 10 marks

Answer:

A heuristic function is called *admissible* if it never overestimates the cost of reaching a goal node from the current node.

A good admissible heuristic function for this problem is the number of moves required to go from the current cell to the lower righthand corner under the assumption that there are no black cells. We can compute this number as the sum of (a) the difference between the grid width and the current x-coordinate and (b) the difference between the grid height and the current y-coordinate.

```
estimate(X/Y, Cost) :- gridsize(Size), Cost is (Size - X) + (Size - Y).
```

The heuristic function computing the straight-line distance from the current cell to the lower righthand corner is also admissible, but not as sharp and thus leads to a less efficient search algorithm. But it would also constitute an acceptable answer.