Logisch Programmeren en Zoektechnieken: Tentamen B **ANSWERS**
24 October 2013

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

The purpose of this exercise is to implement and analyse the sorting algorithm known as *selection sort*. We restrict ourselves to the case of sorting a list of numbers in ascending order. The algorithm works by repeatedly extracting the smallest element from a list of numbers and moving that minimal element to the front of the list to be sorted.

(a) Implement a predicate `min/2` to return the minimal element in a given list of numbers. Example: [5 marks]

```
?- min([7, 23, 3, 17, 43], Min).
Min = 3
Yes
```

Answer:

```
min([X], X).

min([X,Y|Rest], Min) :-
  X =< Y,
  min([X|Rest], Min).

min([X,Y|Rest], Min) :-
  Y < X,
  min([Y|Rest], Min).
```

(b) Implement a predicate `extract/3` to extract the minimal element from a given list of numbers and to also return the list of remaining numbers. Example: [5 marks]

```
?- extract([7, 23, 3, 17, 43], Min, Rest).
Min = 3,
Rest = [7, 23, 17, 43]
Yes
```

Answer:

```
extract(List, Min, Rest) :-
  min(List, Min),
  select(Min, List, Rest).
```

(c) Implement a predicate `selsort/2` to sort a given list of numbers in ascending order. Your predicate should extract the smallest element from a given list and then insert that element at the front of the list obtained by sorting the remaining list. Example: [10 marks]

```
?- selsort([7, 23, 3, 17, 43], List).
List = [3, 7, 17, 23, 43]
Yes
```

Answer:

```
selsort([], []).

selsort(List, [Min|SortedRest]) :-
  extract(List, Min, Rest),
  selsort(Rest, SortedRest).
```

(d) What is the time complexity of *selection sort*? State your answer both in words and using the Big-O notation. Justify your answer. [5 marks]

Answer:

Selection sort has quadratic complexity: $O(n^2)$. Let n be the length of the list to be sorted. The algorithm proceeds in n rounds (one for each element). In round k we have to find the minimum of a list of $n-k+1$ elements, which involves $n-k$ comparisons. Hence, in total we have to make $\sum_{k=1}^{n} n-k = \frac{n(n-1)}{2} \in O(n^2)$ comparisons. Note that the implementation given above is not optimal in the sense that we go through the list twice, once to find the minimum element and once to extract it. This increases complexity by a factor of 2, i.e., it is still in $O(n^2)$.

Question 2

(a) Give a brief overview of the *state-space representation* for search problems as introduced in class (you do not need to cover cost functions). Your answer should include a brief discussion of the Prolog predicates a developer following this approach would have to implement. Give first a general overview of the approach and then sketch how to use it to represent a specific search problem of your choice. [10 marks]

Answer:

See lecture slides. A complete answer should mention states, the initial state, goal states, and moves between states. It should also mention the Prolog predicates `goal/1` and `move/2`. An example for a specific search problem that can be represented using this approach is the *blocks world* discussed in class.

(b) Define what it means for a function f to be in $O(n^3)$. Support your definition with (at least) two examples, one of them a function that is in $O(n^3)$ and one of them a function that is not. [5 marks]

Answer:

See lecture slides for the definition. $20 \cdot n^3 + 17$ is an example for a function in $O(n^3)$; n^4 is an example for a function that is not in $O(n^3)$.

(c) Briefly summarise what you know about the time and space complexity of depth-first search, breadth-first search, and iterative deepening. [10 marks]

Answer:

See lecture slides.

Question 3

(a) Explain, in your own words, why we have a guarantee that the A* algorithm will return an *optimal* answer whenever the heuristic function used is *admissible*. [5 marks]

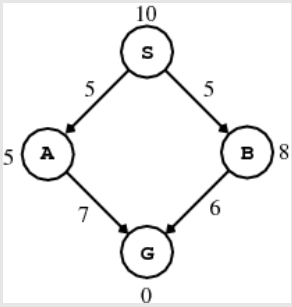
Answer:

See lecture slides.

(b) Give a concrete example that shows that A* with a heuristic function that is not admissible may find a suboptimal solution before finding any optimal solutions. [5 marks]

Answer:

In the following example, the heuristic function overestimates the cost of reaching the goal state `G` from node `B`. As a consequence of using this non-admissible heuristic function, A* would find the suboptimal path `S-A-G` first.



(c) In the Prolog implementation shown in class, the predicate `get_best/2` implements the search strategy of A* by selecting a path minimising the sum of the current cost and the current estimate: [5 marks]

```
get_best([Path], Path) :- !.

get_best([Path1/Cost1/Est1,_/Cost2/Est2|Paths], BestPath) :-
  Cost1 + Est1 =< Cost2 + Est2, !,
  get_best([Path1/Cost1/Est1|Paths], BestPath).

get_best([_|Paths], BestPath) :-
  get_best(Paths, BestPath).
```

It has been argued that changing the implementation of `get_best/2` is all that is required to implement any other best-first search algorithm. Explain how to change the code to implement *greedy best-first search*.

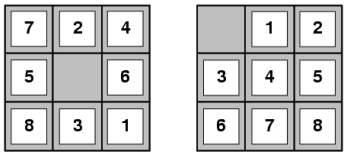
Answer:

In greedy best-first search, a path ending in a node with minimal estimated cost to reach a goal state is explored next. The only change required is to replace the second line of the second clause of `get_best/2` with the following line:

```
Est1 =< Est2, !,
```

To avoid syntax warnings (about singleton variables), `Cost1` and `Cost2` should also be replaced with anonymous variables.

(d) Recall the *eight-puzzle* briefly discussed in class: 10 marks



The goal is to move the tiles so as to reach the configuration shown on the right. The number of misplaced tiles is a possible heuristic function we can use when modelling this puzzle as a search problem. Write a Prolog predicate `misplaced/2` to compute, for a given configuration of the puzzle, the number of fields on which it differs from the goal configuration (i.e., this will be a number between 0 and 9). Assume that a configuration of the puzzle is represented as a list of three lists, with `x` representing the field without a tile. For example, the configuration shown on the left above is represented as `[[7,2,4], [5,x,6], [8,3,1]]`. Examples:

```
?- misplaced([[7,2,4], [5,x,6], [8,3,1]], Number).
Number = 9
Yes

?- misplaced([[4,3,2], [x,1,5], [6,7,8]], Number).
Number = 4
Yes
```

Answer:

```
misplaced(Puzzle, N) :-
  flatten(Puzzle, List),
  diff(List, [x,1,2,3,4,5,6,7,8], N).

diff([], [], 0).

diff([Head|Tail1], [Head|Tail2], N) :- !,
  diff(Tail1, Tail2, N).

diff([_|Tail1], [_|Tail2], N1) :-
  diff(Tail1, Tail2, N),
  N1 is N + 1.
```

Question 4

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

Answer:
A good solution should briefly say what matching and unification are, and it should mention the occurs-check.

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

```
app(e, X, X).

app(f(W,X), Y, f(W,Z)) :- app(X, Y, Z).

sel(X, Y, Z) :- app(U, f(X,W), Y), app(U, W, Z).

?- sel(b, f(a,f(b,f(c,e))), X).
```

Answer:

$$\{ \quad \forall x.app(e,x,x),$$
$$\forall w.\forall x.\forall y.\forall z.(app(x,y,z) \rightarrow app(f(w,x),y,f(w,z))),$$
$$\forall u.\forall w.\forall x.\forall y.\forall z.(app(u,f(x,w),y) \wedge app(u,w,z) \rightarrow sel(x,y,z),$$
$$\forall x.(sel(b,f(a,f(b,f(c,e))),x) \rightarrow \bot) \qquad \}$$

(c) Suppose the following operators representing the usual connectives of propositional logic have been made: 10 marks

```
:- op(100, fy, neg),
   op(200, yfx, and),
   op(300, yfx, or).
```

Write a Prolog predicate `cnf/1` to check whether a given formula of propositional logic is in conjunctive normal form (CNF). Examples:

```
?- cnf((p or q) and (neg p or neg q) and neg r).
Yes

?- cnf((p or neg neg q) and r).
No
```

Answer:

```
literal(A) :- atom(A).
literal(neg A) :- atom(A).

disj_of_lits(A) :- literal(A).
disj_of_lits(A or B) :- disj_of_lits(A), disj_of_lits(B).

cnf(A) :- disj_of_lits(A).
cnf(A and B) :- cnf(A), cnf(B).
```