

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 legibly. The maximum time allowed is 120 minutes.

Question 1

- (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 user following this approach would have to implement.

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`.

- (b) Recall the implementation of the basic depth-first search algorithm given in class:

10 marks

```
solve_depthfirst(Node, [Node|Path]) :-  
    depthfirst(Node, Path).
```

```
depthfirst(Node, []) :-  
    goal(Node).
```

```
depthfirst(Node, [NextNode|Path]) :-  
    move(Node, NextNode),  
    depthfirst(NextNode, Path).
```

Give a Prolog program defining a search space and an initial state such that the above algorithm would loop and never return an answer, even though an answer does exist (and even though that answer could be found by one of the other search algorithms introduced in class).

Answer:

```
move(a, a).  
move(a, b).  
goal(b).  
  
?- solve_depthfirst(a, Path).  
ERROR: Out of local stack
```

- (c) Define what it means for a function $f : \mathbb{N} \rightarrow \mathbb{N}$ to be in $O(n^2)$.

5 marks

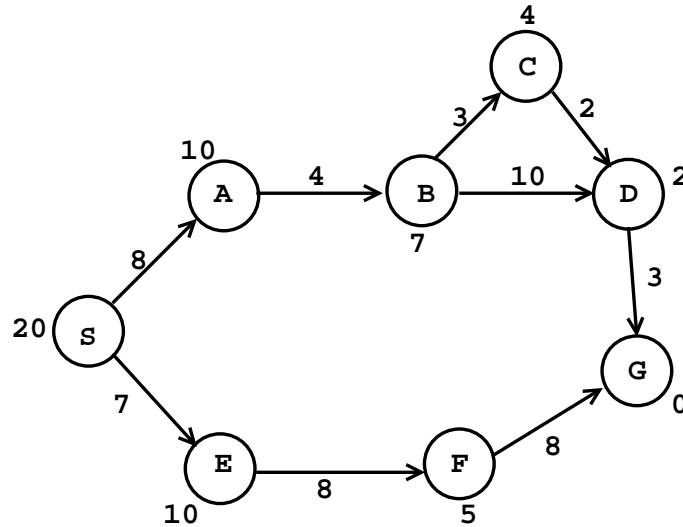
Answer:

$f(n) \in O(n^2)$ if and only if there exist a $c \in \mathbb{R}^+$ and an $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot n^2$ for all $n \geq n_0$.

Question 2

- (a) Consider the search space given by the following graph. S is the initial state and G is the goal state. The numbers labelling the arcs between nodes define the cost of moving from one node to the next, and the numbers next to the nodes define a heuristic function.

10 marks



In which order would the A* algorithm explore this search space? For each step of the algorithm, show which paths are currently kept in memory. Label each path with both its associated cost and the estimated cost of reaching the goal node from its end node.

Answer:		
(0)	S	0+20
(1)	S-A	8+10
	S-E	7+10 ⇐
(2)	S-A	8+10 ⇐
	S-E-F	15+5
(3)	S-A-B	12+7 ⇐
	S-E-F	15+5
	S-A-B-C	15+4 ⇐
(4)	S-A-B-D	22+2
	S-E-F	15+5
	S-A-B-C-D	17+2 ⇐
	S-A-B-D	22+2
(5)	S-E-F	15+5
	S-A-B-C-D-G	20+0 ✓
	S-A-B-D	22+2
(6)	S-E-F	15+5

- (b) In the Prolog implementation shown in class, the predicate `get_best/2` implements

10 marks

the search strategy of A* by selecting a path minimising the sum of the current cost and the current estimate:

```
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.

- (c) 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.

Question 3

The purpose of this question is to implement and analyse a sorting algorithm known as *merge-sort*. The algorithm works by recursively applying the following steps:

- (1) Split the input list into two lists of roughly equal length.
- (2) Recursively apply *merge-sort* to each of the two sublists.
- (3) Merge the two sorted sublists to obtain the full sorted list.

Answer the following questions:

- (a) Write a Prolog predicate to split a given list into two sublists of roughly equal length. Example:

5 marks

```
?- split([1,2,3,4,5,6,7,8,9], ListA, ListB).  
ListA = [1, 2, 3, 4],  
ListB = [5, 6, 7, 8, 9]  
Yes
```

Answer:

```
split(List, List1, List2) :-  
    length(List, Length),  
    Length1 is Length // 2,  
    length(List1, Length1),  
    append(List1, List2, List).
```

- (b) Write a Prolog predicate to merge two sorted lists. Example:

10 marks

```
?- merge(<, [1,3,5,7,9], [2,4,6,8], MergedList).  
MergedList = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
Yes
```

That is, the result should be the list we would obtain if we were to first append the two input lists and then sort the resulting list. But please note that this is just an explanation of the specification of `merge/4`; you will not need `append/3` and you should not use any sorting algorithms for answering this question.

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 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:

```
merge(_, [], List, List).  
  
merge(_, List, [], List).  
  
merge(Rel, [H1|T1], [H2|T2], [H1|Rest]) :-  
    check(Rel, H1, H2), !,  
    merge(Rel, T1, [H2|T2], Rest).  
  
merge(Rel, [H1|T1], [H2|T2], [H2|Rest]) :-  
    merge(Rel, [H1|T1], T2, Rest).
```

- (c) Now implement the *merge-sort* algorithm. Example:

5 marks

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

Answer:

```
mergesort(_, [], []) :- !.
```

```
mergesort(_, [X], [X]) :- !.
```

```
mergesort(Rel, List, SortedList) :-  
    split(List, List1, List2),  
    mergesort(Rel, List1, SortedList1),  
    mergesort(Rel, List2, SortedList2),  
    merge(Rel, SortedList1, SortedList2, SortedList).
```

- (d) What is the worst-case time complexity of *merge-sort*? Use the Big-O Notation to formally state your answer and give a brief justification.

5 marks

Answer:

The complexity of *merge-sort* is in $O(n \log n)$, where n is the length of the list to be sorted. Justification: Merging two lists of roughly length m each requires $O(m)$ comparisons. So the merging operations carried out at each recursion level add up to $O(n)$. The recursion depth is logarithmic (number of times we can keep on cutting a list in half): $O(\log n)$. Altogether we get a complexity of $O(n \log n)$. (Observe that we get the same complexity whatever the initial ordering of the elements in the input list, so any case is the worst/best case.)

Question 4

- (a) Translate the following Prolog program (which includes a query) into a set of first-order logic formulas.

10 marks

```
parent(alice, bob).  
parent(bob, carla).  
female(alice).  
female(carla).  
ancestor(X, Y) :- parent(X, Y).  
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).  
ancestor(adam, _).  
:- ancestor(X, carla), female(X).
```

Answer:

```
{ parent(alice, bob),  
  parent(bob, carla),  
  female(alice),  
  female(carla),  
   $\forall x. \forall y. (\text{parent}(x, y) \rightarrow \text{ancestor}(x, y))$ ,  
   $\forall x. \forall y. \forall z. (\text{parent}(x, z) \wedge \text{ancestor}(z, y) \rightarrow \text{ancestor}(x, y))$ ,  
   $\forall x. \text{ancestor}(\text{adam}, x)$ ,  
   $\forall x. (\text{ancestor}(x, \text{carla}) \wedge \text{female}(x) \rightarrow \perp)$  }
```

- (b) Use the resolution method to prove the validity of the following statement:

10 marks

If $(P \vee Q \vee R)$ and $(Q \rightarrow R)$ are true, then so is $(\neg R \rightarrow P)$.

You may have to translate some formulas into the appropriate normal form first. Make sure that it is possible to follow your proof: label each line with a number and always indicate which lines you are using to derive a new formula.

Answer:

Translate the premises and the negation of the conclusion into clauses. Note that the latter gives rise to *two* clauses.

(1)	$\{P, Q, R\}$	given
(2)	$\{-Q, R\}$	given
(3)	$\{-R\}$	given
(4)	$\{-P\}$	given
(5)	$\{P, R\}$	from 1 and 2
(6)	$\{R\}$	from 4 and 5
(7)	\square	from 3 and 6

(c) For each of the following Prolog queries say whether they would succeed and, in case of success, give all answers that can be obtained via enforced backtracking:

5 marks

- `?- X =.. [alpha, beta, gamma, delta].`

Answer:

```
?- X =.. [alpha, beta, gamma, delta].
X = alpha(beta, gamma, delta) ;
No
```

- `?- L = [a,b,c,a,d], findall(X, append(_, [a,X|_], L), What).`

Answer:

```
?- L = [a,b,c,a,d], findall(X, append(_, [a,X|_], L), What).
L = [a, b, c, a, d] ;
What = [b, d]
No
```

- `?- L = [1:a,2:b,3:a,4:b], setof(X, member(X:Y,L), Z).`

Answer:

```
?- L = [1:a,2:b,3:a,4:b], setof(X, member(X:Y,L), Z).

L = [1:a, 2:b, 3:a, 4:b],
Y = a,
Z = [1, 3] ;

L = [1:a, 2:b, 3:a, 4:b],
Y = b,
Z = [2, 4] ;

No
```