EARIN – exam 1 – *solution hints*

*You are free to use notes and books. Total writing time is **90 minutes**.*

Task 1 (10)

Assume that the search space is organized in a tree form. Suggest a definition of the cost function and the heuristic function that guarantees that the sequence of nodes visited by the A* method would be identical to the sequence of nodes visited by the depth-first method. Justify your answer briefly.

Solution hints:

The depth-first search uses the LIFO queue to keep solutions, whereas A*, which is a kind of the best-first search, uses the priority queue. The priority is the sum of the cost function and the heuristic function. Therefore the sum of the cost and the heuristic function should assign values which result in such organization of the priority queue that it resembles the LIFO. All nodes added earlier should have a lower priority than nodes added sooner. For example, the heuristic function should be zero for all nodes and the cost function should be equal to a value which is increased by one each in each iteration of the best-first search algorithm, assuming that the cost function is maximized. Alternatively, the cost function value for a node V should be equal to the number of edges between the node V and the root of the search space.

Task 2 (10)
A spanning tree of a connected graph G=(V,E) is a connected graph T=(V,E') where E' is a subset of the set of edges E that cannot be reduced without loosing connectivity of T. Consider a weighted graph. We search for a spanning tree whose sum of weights is smallest. Assume that we use the A* algorithm to perform this task. Define the solution representation, the neighborhood relation, the cost function and the heuristic function. Justify that the heuristic function is admissible and monotone.

Solution hints:

Solutions are represented as vectors. Length of each vector equals the number of edges in the graph G. Each position of this vector may equal 1 (the corresponding edge is included into the solution), 0 (the edge is not contained by the solution) or ? (the decision has not been made yet), similarly as in the knapsack problem. Note that the spanning tree for a graph with *n* vertices will contain exactly *n*-1 edges.

Vectors representing solutions are organized in a form of a tree. Neighbors of a vector X are vectors which resulted either from changing a single position 0 or 1 into ? or from changing a ? into 0 or 1. The root of the tree is the vector composed of all ?'s, and leaves are composed of exactly *n*-1 1's.

The cost function assigned to the vector X equals the sum of weights of these edges whose corresponding positions equal 1.

The heuristic function equals sum of n-k-1 smallest weights, selected from edges whose corresponding positions are ?; k is the number of 1's in the vector.
It is admissible since when we add these *n*-*k*-1 missing edges, their sum of weights will never be smaller than the sum of *n*-*k*-1 smallest weights.
It is monotone since when we add a single edge, the cost will increase by its weight and the heuristic function will decrease either by the same weight or by a value which is greater than that.

Task 3 (10)

Assume the following definitions of predicates that define an undirected graph

```
direct (a, b).
direct (a, c).
direct (a, d).
direct (b, c).
direct (c, d).
direct (d, e).

connection (X, Y):-
   direct (X,Y),!.
connection (X, Y):-
   direct (Y,X).
```

Define the predicate `connected(List)`which is true when the `List` contains nodes that form a connected graph. For example,
it is **true** that `connected([a,b,c,d])` and `connected([a,c,d]),`
but it is **not true** that `connected([b,d])` or `connected ([a,b,e]).`

Solution hints:

The idea is to check if there exists a path between any pair of nodes from the set `List`. Moreover, this path must be composed of nodes from the `List` only. The PROLOG code may look like this:

Checks for a path between `X` and `Y` which is composed of elements from the list `L`:

```
inpath(X,X,L):-!.
inpath(X,Y,L):-
   connection(X,Y),!.
inpath(X,Y,L):-
   connection(X,Z),
   member(Z,L),
   inpath(Z,Y,L),!.
```

Checks if such path exists between `X` and any node from the list given as the second argument; the list `L` indicates which nodes may be contained by these paths:

```
con2(X,[Y],L):-
   inpath(X,Y,L).
con2(X,[H|T],L):-
   inpath(X,H,L),
   con2(X,T,L).
```

The final predicate:

```
connected([X]):-!.
connected([H|T]):-
   con2(H,[H|T],[H|T]).
```

Task 4 (10)

A prime number is a natural number which has no divisors other than 1 and itself. Define the predicate `prime` which is true when a number is prime.
For example,
it is **true** when `prime(3)` and `prime(19)`,
but it is **not true** when `prime(8)` or `prime(36)`.

Hint: you may find useful the function `mod(X,Y)` which returns the result of the modulo operation, i.e., it finds the remainder after division of `X` by `Y`.

Solution hints:

An example solution is based on checking all numbers greater that 2 and smaller than the upper integer bound of the square root of `X`. If any of these numbers is a divisor of `X` then `X` is not a prime number. The PROLOG code may look like this:

Checks for a divisor of `X` which is greater than or equal `Y` and smaller than `X`; stop if `X` becomes smaller than `Y*Y`:

```
ind(X,X):-!.
ind(X,Y):-
   X<Y*Y,!.
ind(X,Y):-
   Y<X,
   0 =\= mod(X,Y),
   Z is Y+1,
   ind(X,Z).
```

The final predicate:

```
prime(1):-!.
prime(2):-!.
prime(X):-
   X>2,ind(X,2).
```