**Problem 4.1** Heuristic Searches

**Answer:**

According to the problem description,

1. In Greedy and A\* search, both expand the node that minimizes an evaluation function . Their evaluation function is as follows,
   1. Greedy search,
   2. A\* search,
2. is the optimal cost of the shortest path from to goal state (*Bucharest*). Since is always bigger than the straight-line distance from to , . Therefore is admissible.
3. For each search, the order in which nodes are expanded are as follows,

**Greedy Search:**

1. Expanded Nodes: *empty*

Fringe:

1. Expanded Nodes: *Lugoj*

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

**A\* Search:**

1. Expanded Nodes: *empty*

Fringe:

1. Expanded Nodes: *Lugoj*

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

Fringe:

1. Expanded Nodes:

**Problem 4.2** Heuristics

**Answer:**

1. **Similarity:** Greedy search and A\* search both use an evaluation function for expanding the node that minimizes the evaluation function .

**Difference:** Though Greedy search and A\* search both expand the node that minimizes the evaluation function , but their evaluation function is different from each other which are as follows, Greedy search, , and A\* search, . Here, is the heuristic function and is the cheapest path cost.

1. is admissible, if , where is the optimal cost of the shortest path from to goal state. For a constant function , the evaluation function of A\* search, . So, the constant function is always less than the cost of the shortest path for A\* search, thus it is admissible. But it is a useless heuristic function since it doesn’t help to decide which nodes should be expanded next.

**Problem 4.5** Minimax Search in ProLog

**Answer:**

% Game state: number N of remaining matches, current player P=1 or P=-1

% contains(L, A): checks whether A is a element of List L or not

contains([H|T], A) :- not(H=A), contains(T, A).

contains([A|\_], A).

% possible moves in state(N, P) yielding successor state T

successor(state(N, P), T) :- N>0, N2 is N-1, P2 is -P, T=state(N2, P2).

successor(state(N, P), T) :- N>1, N2 is N-2, P2 is -P, T=state(N2, P2).

successor(state(N, P), T) :- N>2, N2 is N-3, P2 is -P, T=state(N2, P2).

% find list Ts of successor states of S using accumulator Acc

successors(S, Acc, Ts) :- successor(S, T), \+ contains(Acc, T), !, successors(S, [T|Acc], Ts).

successors(\_, Acc, Acc).

% minvalue(Ss, MinSofar, V): returns minimum value V of list of states Ss

% where MinSofar is accumulator for minimum value seen so far

minvalue([], MinSofar, MinSofar).

minvalue([S|Ss], MinSofar, V) :- value(S, V1), V1<MinSofar, minvalue(Ss, V1, V).

minvalue([S|Ss], MinSofar, V) :- value(S, V1), V1>=MinSofar, minvalue(Ss, MinSofar, V).

% maxvalue(Ss, MaxSofar, V): returns maximum value V of list of states Ss

% where MaxSofar is accumulator for maximum value seen so far

maxvalue([], MaxSofar, MaxSofar).

maxvalue([S|Ss], MaxSofar, V) :- value(S, V1), V1>=MaxSofar, maxvalue(Ss, V1, V).

maxvalue([S|Ss], MaxSofar, V) :- value(S, V1), V1<MaxSofar, maxvalue(Ss, MaxSofar, V).

% value(S, V): returns winner V (1 or -1) given the initial state S

% step 1: (P=1)'s turn, here choose successor with maximum value

% step 2: (P=-1)'s turn, here choose successor with minimum value

value(S, V) :- state(\_, P)=S, P = 1, successors(S, [], Ts), maxvalue(Ts, -1, V).

value(S, V) :- state(\_, P)=S, P = -1, successors(S, [], Ts), minvalue(Ts, 1, V).

%--------------------------- Sample Query ---------------------------

% contains([1,2,3,4,5], 3)

% contains([1,2,3,4,5], 6)

% successor(state(4,1), T)

% successor(state(4,-1), T)

% successors(state(4, 1), [], T)

% successors(state(3, -1), [], T)

% value(state(4, 1), Player)

% value(state(5, 1), Player)