Artificial Intelligence 1 2022
Assignment4: Search
– Given Nov. 21, Due Nov. 27 –

Problem 4.1 (Heuristic Searches) 0 pt

Consider the graph of Romanian cities with edges labeled with costs $c(m, n)$ of going from m to n. $c(m, n)$ is always bigger than the straight-line distance from m to n. $c(m, n)$ is infinite if there is no edge.

Our search algorithm keeps:

- a list E of expanded nodes n together with the cost $g(n)$ of the cheapest path to n found so far,
- a fringe F containing the unexpanded neighbors of expanded nodes.

We want to find a cheap path from Lugoj to Bucharest. Initially, E is empty, and F contains only Lugoj. We terminate if E contains Bucharest.

Expansion of a node n in F moves it from F to E and adds to F every neighbor of n that is not already in E or F. We obtain $g(n)$ by minimizing $g(e) + c(e, n)$ over expanded nodes e.

As a heuristic $h(n)$, we use the straight-line distance from n to Bucharest as given by the table in the lecture.

1. Explain how the following algorithms choose which node to expand next:
   - greedy search with heuristic h
   - A* search with path cost g and heuristic h
2. Explain what h^* is here and why h is admissible.
3. For each search, give the order in which nodes are expanded.
   (You only have to give the nodes to get the full score. But to get partial credit in case you're wrong, you may want to include for each step all nodes in the fringe and their cost.)

Solution:

1. Both searches expand the node n that minimizes a function. The functions are
   (a) greedy search: $h(n)$
   (b) A*: $g(n) + h(n)$
2. $h^*(n)$ is the cost of the shortest path from n to Bucharest (which we do not know unless we expand all nodes). Because $c(m, n)$ is always bigger than the straight-line distance, every path is longer than the straight-line distance between its end points. Thus $h(n) \leq h^*(n)$.
3. Greedy search: Lugoj (244), Mehadia (241), Drobeta (242), Craiova (160), Pitesti (100), Bucharest (0)
   A* search: Lugoj (0+244), Mehadia (70+241), Drobeta ((70+75)+242), Craiova (70+75+120)+160), Timisoara (111+329), Pitesti ((70+75+120+138)+100), Bucharest ((70+75+120+138+101)+0)

Problem 4.2 (Heuristics) 20 pt

Consider heuristic search with heuristic function h.

1. Briefly explain what is the same and what is different between A^* search and greedy search regarding the decision which node to expand next.
2. Is the constant function $h(n) = 0$ an admissible heuristic for A^* search?

Solution:

1. Both choose the node that minimizes a certain function. As that function, A^* uses the sum of path cost and heuristic whereas greedy only uses the heuristic.
2. Yes. (But it's a useless one.)

Problem 4.3 (Games for Adversarial Search) 30 pt

For each of the following games and properties, state whether the game has the property.

Properties:

A 2 players alternating moves

I discrete state space

C players have complete information about state

F finite number of move options per state

E deterministic successor states

T games guaranteed to terminate

U terminal state has zero-sum utility.

Games:

1. 2-player poker (until one player is bankrupted)
2. Backgammon
3. Wrestling (one 5 minute round)
4. Connect Four
5. Rock-Paper-Scissors (with a repeat to break ties)
6. Meta-Game (player 1's first move is to choose a game that satisfies all properties, subsequent moves play that game)

Submit a text file containing one line per game (in the order given above) such that each line contains the upper-case letters (without space) of all conditions that are violated.

Solution: The games violate the following restrictions:

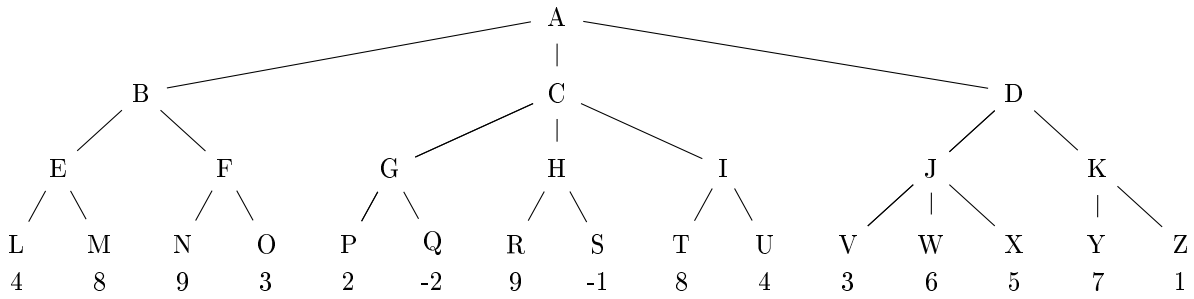
- 2-player poker: C, E, T
- Backgammon: E, T
- Wrestling: A, I, F
- Connect Four: none, solvable
- Rock-Paper-Scissors: A, T
- Meta-game: F

Games are solvable if they satisfy all properties.

Note: Because we have never made (and could not easily make) a formal definition of *game* that includes all possible games, it can be difficult to define the criteria in a way that allows checking them for any game. And whenever we do not have formal definitions, some answers may be unclear.

Problem 4.4 (Game Tree) 20 pt

Consider the following game tree. Assume it is the maximizing player's turn to move. The values at the leaves are the static evaluation function values of the states at each of those nodes.



1. Compute the minimax game value of nodes A, B, C, and D
2. Which move would be selected by Max?
3. List the nodes that the alpha-beta algorithm would prune (i.e., not visit). Assume children of a node are visited left-to-right.

Submit your solution as a text file containing the following:

1. Line 1: 4 numbers, separated by space, corresponding to the nodes in alphabetical order, e.g., "1 3 2 5" means A=1, B=3, C=2, D=5.
2. Line 2: The upper-case letter for the selected move.
3. Line 3: The upper-case letters of the pruned moves.

Solution:

1. B = 8, C = 2, D = 6, A = 8
2. B
3. O, H (and R and S), I (and T and U), K (and Y and Z)

Problem 4.5 (Minimax Search in ProLog) 30 pt

Consider the following game:

1. There is a pile of n matches in the middle.
2. Two players alternate taking away 1, 2, or 3 matches.
3. The winner is whoever takes the last match.

Solve this game (for all values of n) by implementing the minimax algorithm in Prolog. Specifically, implement exactly the following

- a Prolog predicate `value(S,P)` that holds if player `P` wins from initial state `S`,

- where the Prolog constructor `state(N,P)` represents the game state with `N` remaining matches and player `P` going next,
- where we represent players `P` using 1 for the starting player and -1 for the opponent.

Note: A partial solution will be explained in the tutorials, especially the use of `\+` for negation-as-failure and `!` for cut.

Solution:

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

% 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).

% membership in a list
contains([H|T],A) :- not(H=A), contains(T,A).
contains([A|_],A).

% find list Ts of succesor states of S using accumulator Acc
successors(S, Acc, Ts) :- successor(S,T), \+ contains(Acc,T), !, successors(S, [T|Acc], Ts).
successors(_, Acc, Acc).

% shown until here in the tutorials

% minvalue(Ss,Sofar,V) holds if V is the minimum value of list of states Ss
% Sofar is accumulator for minimum value seen so far

% end of list - return accumulator
minvalue([],Sofar,Sofar).
% next state has smaller value, replace accumulator and continue with rest
minvalue([S|Ss],Sofar,V) :- value(S,V1), V1<Sofar, minvalue(Ss,V1,V).
% next state has non-smaller value, keep accumulator and continue with rest
minvalue([S|Ss],Sofar,V) :- value(S,V1), V1>=Sofar, minvalue(Ss,Sofar,V).

% like minvalue
maxvalue([],Sofar,Sofar).
maxvalue([S|Ss],Sofar,V) :- value(S,V1), V1>=Sofar, maxvalue(Ss,V1,V).
maxvalue([S|Ss],Sofar,V) :- value(S,V1), V1<Sofar, maxvalue(Ss,Sofar,V).

% value(S,V) holds if state S has winner V (1 or -1)

% our turn (P=1): choose successor with maximal value
% we lose if no possible move (Ts=[], accumulator initialized to -1)
value(S,V) :- state(_,P)=S, P = 1, successors(S,[],Ts), maxvalue(Ts,-1,V).

% opponent's turn (P=-1): choose successor with minimal value
% we win if no possible move (Ts=[], accumulator initialized to 1)
value(S,V) :- state(_,P)=S, P = -1, successors(S,[],Ts), minvalue(Ts,1,V).
```
