CPSC 322
Introduction to Artificial Intelligence

November 12, 2004

Things...

Midterm 2 marks today

The final exam will be at noon on Friday, December 10, in MCML 166

Problem 1 (5 points): I've given up on getting WebCT figured out, so I'm going to post your scores on the web page so you can see them when you want to. Of course, I'll do that only while maintaining your privacy and only if you give me permission to do so. If you'd like me to make your scores accessible, please check the first box below and provide me with a secret identity code known only to you (4 to 8 characters) so that you can tell which scores are yours. If you do not want your scores to be published on the web, please check the second box below.

Problem 2 (30 points total): The peg puzzle consists of a small slab of wood with six holes drilled in it, all lined up in a row. It comes with two red pegs and two blue pegs and, as you might guess, the pegs are just the right size to fit in the holes.. Each of the holes may hold one peg. Initially the two blue pegs are in the left-most holes, and the two red pegs are in the right-most holes. Below is an idealized view of the start state of the peg puzzle from overhead.

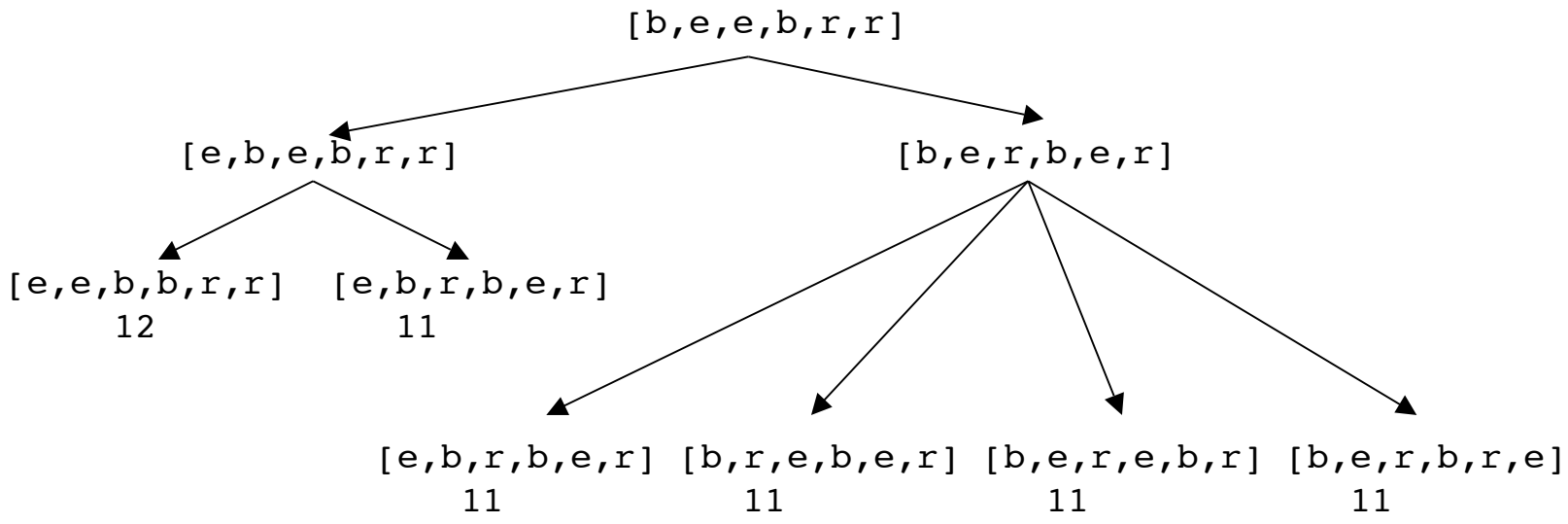
To solve this puzzle the blue pegs must end up in the right-most holes and the red pegs in the left-most holes by moving one peg at a time according to the following rules:

A peg may be moved into an adjacent empty hole.
A peg may jump over a single peg of another color into an empty hole.
The red pegs may only be moved to the left.
The blue pegs may only be moved to the right.

Note that the peg puzzle is a puzzle, not a two-player game with a winner and a loser. There is no requirement that moves alternate between colors. In other words, it is not the case that, for example, red moves first, then blue, then red, and so on.

2a. (5 points): Assume that you're going to construct a CILOG program to solve the peg puzzle. Select a representation scheme that's compatible with CILOG for representing one state in the peg puzzle domain. Show how you would represent the initial state. Answer: [b,b,e,e,r,r]

2b (10 points): Using the representation described in your answer to Problem 2a, show in detail the state space which consists of the state described below and all possible next states that could be generated in two moves.



2c (10 points): When your program finds that more than one guess could be made at any point in the game, it will need some way to select one guess over the others. In the space below, describe in English (not in CILOG) a function that your program could use to evaluate the "goodness," or nearness to the goal, of a given state. This function should "see" only the given state and the desired goal state, and return a numeric value that quantifies the "goodness" of the given state. Now go back to the tree you provided as an answer to Problem 2b and apply your function to the bottom-level nodes or states, then write the numeric value returned by your function below each of those bottom-level states on the previous page.

Answer: measure the distance of each peg from its goal position. The maximum value would be 16; the minimum value would be 0. This is a “Manhattan distance” metric in one dimension. See that answer to 2b for the actual values.

2d (5 points): Is the heuristic you provided for Problem 2c an admissible heuristic? If it's not, could you modify it so that it is admissible? Explain why or why not.

Answer: It’s admissible (or else with the application of some simple math, it could be made admissible).

Problem 3 (20 points total): Here is a description of the A* Algorithm (it's A* because we're assuming that the branching factor is finite, the arc costs are bounded above zero, and *h(n)* is admissible). The description below is slightly different than what was presented in class on a PowerPoint slide, so read the description carefully before you answer the questions:

```
Given a set of start nodes, a set of goal nodes,
and a graph (i.e., the nodes and arcs):

apply heuristic g(n)+h(n) to start nodes
make a "list" of the start nodes and their g(n)+h(n) values
   (let's call it the "frontier")
sort the frontier by g(n)+h(n) values

repeat
   if no nodes on the frontier then terminate with failure
   choose one node from the front of the frontier and remove it
   if the chosen node matches the goal node
      then terminate with success
      else put next nodes (neighbors) and their g(n)+h(n) values on
         the front of the frontier and sort frontier by g(n)+h(n) values
end repeat
```

An interesting feature of the A* Algorithm is that it can be converted to other search strategies by changing *g(n)* and *h(n)*. Now consider the following search strategies and recall how they behave:

a) depth-first search
b) breadth-first search
c) lowest-cost-first search
d) heuristic best-first search
e) heuristic depth-first search
f) iterative deepening search

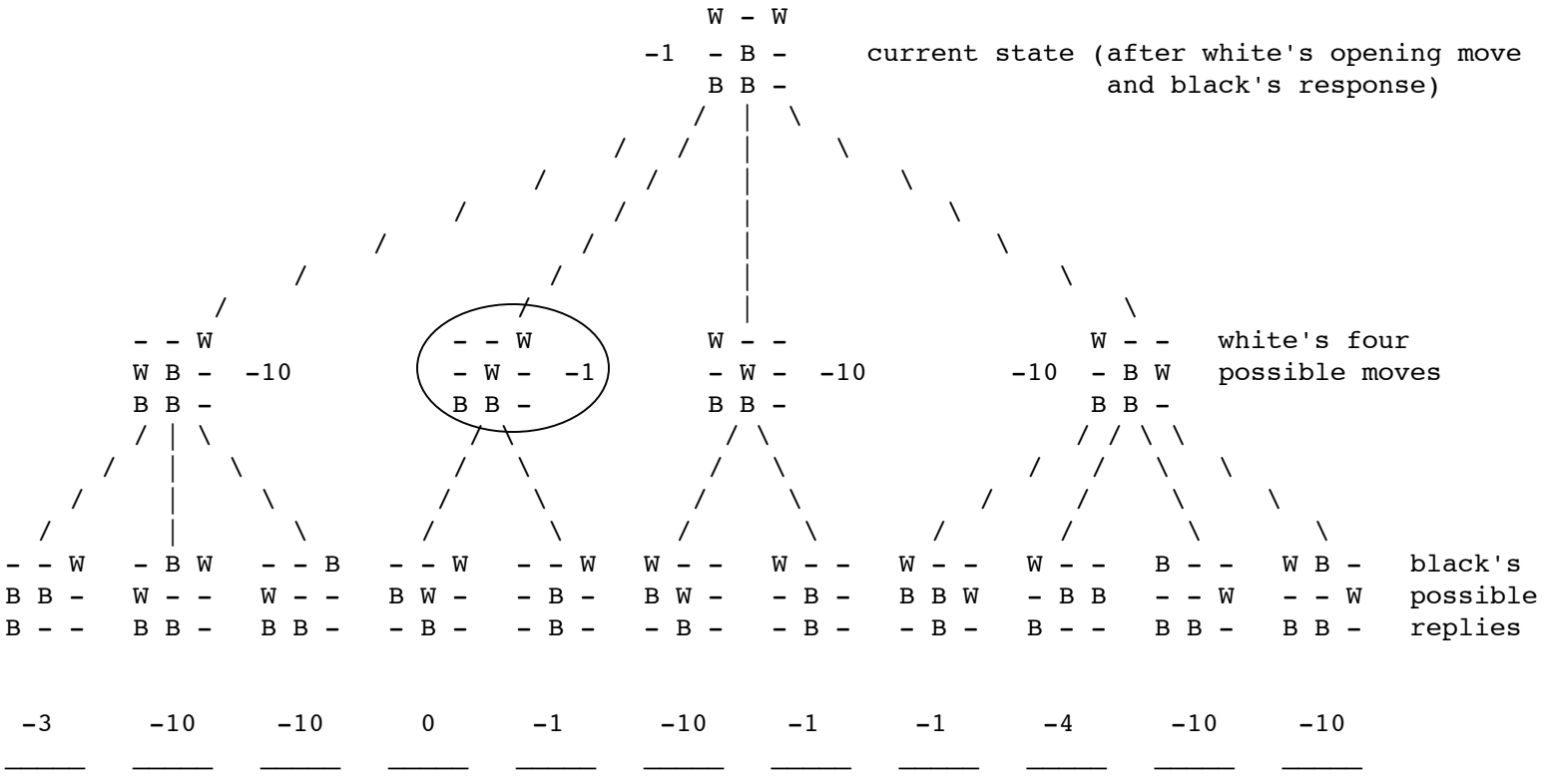
Of the search strategies listed above, which one does A* become if *g(n)* is ignored and only *h(n)* is considered? Write the letter associated with that search strategy here: _d__

Of the search strategies listed above, which one does A* become if *h(n)* is ignored and only *g(n)* is considered, *and* all the individual arc costs are the same? Write the letter associated with that search strategy here: ___b_____

Of the search strategies listed above, which one does A* become if *h(n)* is ignored and only *g(n)* is considered? Write the letter associated with that search strategy here: _c__

Of the search strategies listed above, which one does A* become if both *g(n)* and *h(n)* are ignored and no sorting is done (since there are no values to sort by)? Write the letter associated with that search strategy here: ____a_____

Problem 4 (15 points total): Below is part of the state space for a ripping good game of hexapawn. At the top is the current state of the game, which shows the result of white making an opening move by pushing a pawn forward to the center of the board, and black responding by taking white's pawn. The next level below that shows all four of white's possible responses, and below that you see the moves that black could make in response:



Pretend you're a minimaxing, hexapawn-playing CILOG program that is controlling the white pawns, and your static board evaluation function is as follows:

The function returns a +10 if the board is such that white wins. It returns a -10 if black wins. If neither side has won, the function returns the number of white pawns with clear paths in front of them minus the number of black pawns with clear paths in front of them PLUS the result of counting the number of white pawns on the board and subtracting the number of black pawns. "Clear path" means that a pawn has no other pawns of either color between it and the opposite end of the board as it moves straight ahead. "Clear path" does not consider what may be ahead of a pawn on a diagonal.

As a reminder, a pawn can move forward one space straight ahead to an empty space on any given turn, or it can move ahead one space on the diagonal to a space occupied by an opponent's pawn to capture that pawn. A player wins by capturing all the opponent's pawns, moving one pawn all the way to the opponent's end of the board, or putting the opponent in a position where he or she can't move on his or her turn.

4a (5 points): Apply this static evaluation function to each of the bottom-level boards. Write the value that your function would assign to each board below that board in the spaces provided in that diagram up there. (There are 11 values to compute.)

4b (10 points): On that same diagram, show which values would be propagated upward by your minimax component. Then clearly indicate which move should be made by white, according to the minimax strategy by drawing a circle around the board that represents the move.

5 (20 points): Assume the following relations written in CILOG:

```
fnx(A, [], [A]).
fnx([A, B], [[C, D]|E], [[A, B], [C, D]|E]) <- A>=C.
fnx([A, B], [[C, D]|E], [[C, D]|F]) <- A<C&fnx([A, B], E, F).
fny([], []).
fny([A|B], C) <- fny(B, D)&fnx(A, D, C).
```

Show what happens when CILOG is presented with the following query:

```
ask fny([[3,w],[1,x],[4,y],[2,z]],X).
```

Give the result, and explain how `fnx` and `fny` work.

Answer: X = [[4,y],[3,w],[2,z],[1,x]]
It's an insertion sort based on the first element of each 2-element list. It sorts by descending order. fny is the "sort" function; fnx is the "insert" function.

6 (10 points): When is depth-first search appropriate? When is breadth-first search appropriate? (Don't write something like "one is appropriate when the other isn't".)

Answer:

depth-first search:
When space is restricted.
When there are many solutions, perhaps with long paths.
When the order of nodes in the list of neighbors can allow you to tune the system so that solutions are found on the first try.
When you can easily determine when you are on the wrong path.

breadth-first search:
When space is not a problem.
When you want to find the solution with the fewest arcs.
When there may be few solutions, but with a short path length.
When there may be infinite paths.
When there's no extra (heuristic) knowledge to guide the search.