CS472 Foundations of Artificial Intelligence
Fall 2000
Assignment 2 Solutions

1. **Heuristic Search**

   (a) Monotonicity does imply admissiblity. Proof:
   Suppose that m is an ancestor of a goal node n. (If it is not, then the cost to get from m to n is infinite, and so $h(m)$ clearly underestimates the cost to the goal.) Let j = the cost to get from m to n. Then $g(n) = g(m) + j$. Since $h(n) = 0$, $f(n) = g(m) + j$. By monotonicity, $f(m) \leq f(n)$, so $f(m) \leq g(m) + j$. But $f(m) = g(m) + h(n)$, so $h(n) \leq j$, the actual cost to get from m to n. Thus h underestimates the actual cost to get from m to n. Since m was chosen arbitrarily, h is admissible.

   Grading: 1 point was deducted for forgetting that $h(goal) = 0$ by definition (see page 97 in the book). 2 points were deducted for applying pathmax to f as part of the proof (*i.e.*, to force f to be monotonic). Note that if h is inadmissible, then pathmax makes $f(goal) > g(goal)$. Thus, pathmax only makes sense if h is admissible in the first place.

   (b) We saw that the heuristic function, summing the Manhattan distances of each tile to its goal position is admissible and thus never overestimates. On the other side it clearly underestimates a lot. In fact it gives the correct evaluation if and only if we are no more than one step away from the solution. For upper levels in the search it just assumes that we can move tiles one over another... In order to tolerate this inconsistency, we can choose from many approaches, some of which are:

   - *To multiply $f(x)$ with an appropriate constant, which we think (on average) will reduce the difference between the heuristic and the actual cost.* (Note that this new heuristic overestimates at least when we have chosen a constant $c \geq 2$ and we are one step away from the goal. It will certainly overestimate at other nodes, especially if we choose a larger c, but this does not render it non-sensible);
   - *To use a logarithmic scale on the Manhattan distances.* This approach seems even more appropriate, as we can feel that tiles, which are away from their goal positions would require many other tiles to be moved in order to "clean-up" the path.

   Grading: 1 point was deducted for presenting a heuristic that doesn't make immediate intuitive sense and then failing to explain the intuition behind it.

   (c) Actually it is not as easy to construct such an example, as it seems at first sight. We can think of many different "bad" heuristics, but it is often the case that they just force a larger number of nodes to be expanded, failing to "escape" from the actual solution. Consider one example: $h(x) = 20$. Even this simple overestimating heuristic forces A^* to turn into breadth-first search, an thus always finds an optimal solution. One way to find such example is to pick an awful heuristic, for example $h(x) = random(20)$ (returns a random number between 0 and 20) and run A^* on many instances of the 8-puzzle, checking each time whether it had found a suboptimal solution.

   Grading: 1 point was deducted for showing that A^* heads down a suboptimal path and not noting that it will eventually backtrack from this path and explore an optimal path instead. 2 points were deducted for stating that A^* will go into an "infinite" loop, which is impossible unless you have an inconsistent (*i.e.*, $h(n) \neq h(n)$) heuristic.

2. **Static Evaluation Functions and Minimax**

   (a) Actually it is very useful to know the depth of the tree. In this case it is no more than 80. In reality, most games finish in 25-35 moves (depths 50-70). So we will consider 60 as an average depth of a game search tree. Now we have to estimate the branching factor. As capturing is obligatory, there are certain positions, where just one or two moves are legal (all of them capturing). Each player has exactly 12 pieces in the beginning of the game, thus he/she has at most 12 capturing moves (some captures can remove more than one piece of the opponent, but these situations are fairly rare). This observation (figuratively) reduces the depth of the tree to approximately 40. Theoretically, there are between 7 and 21 (even less) legal moves in the beginning of the game, while at the end the number of legal moves is roughly two times the number of pieces. So, the branching factor seems to depend a lot on the particular game, but we think 10 is a good average, taking into account the various conditions on the board throughout the game (other factors will also be accepted as correct). Now we have enough information to approximate the number of games to 10^{40}.

(b) Static evaluation functions for checkers tend to be complicated. This is partly due to the nondeterministic nature of the game. Generally most of them are a weighted linear combination of several independent attributes a_i of the configuration: $\sum_{i=1}^{n} w_i a_i$. A good source for such attributes is the appendix of olympiad.pdf (a very good paper on computers playing checkers, currently on the course web site). We are going to use a slightly simple function, which takes into account the difference in the count of pieces and the number of pieces which are immediately capturable (of course taken with the appropriate sign): $f(x) = \#black - \#white \pm \#capturable$.

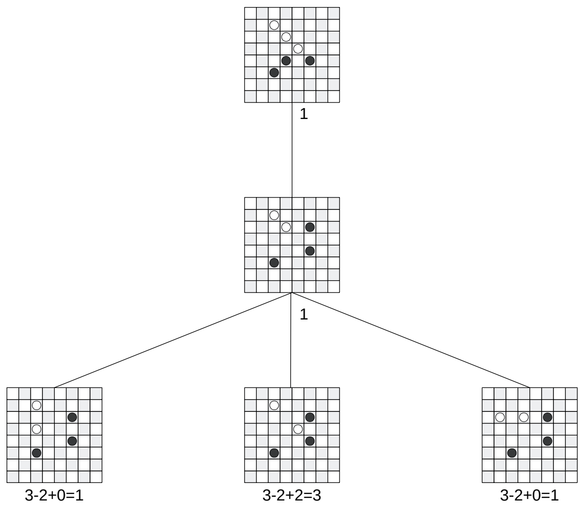

Figure 1: A 3-level (2 moves look-ahead) search tree, using the evaluation function from (b)

(c) Although there is only one legal move available for black, in figure 1 we see that our evaluation function does a pretty good job of estimating its value in this specific case. It sees that white has two equally good options available to it and one move (middle) that would give Max (black) a good counter move.

Grading: Most people did quite well on this question. For part (a) one point was deducted for failing to observe the reduction in the height of the tree. Other small deductions are due to "relevantness" and "correctness" of the assumptions. For part (b) several observations were considered, all valued at 1 point, e.g. mentioning many factors, relevance of the factors (like whose turn it is now) and so on. Very simple functions were accepted but speculation on a real-life function was required. For part (c) almost all mistakes were technical and again most of them were worth 1 point. Some students didn't take into account that Black has to capture (resulting in very big trees), other did 4-level expansion (compared to 3-level). A more major mistake was to make White move first.

3. $\alpha - \beta$ **Pruning**

(a) The correct move is to choose b_0 and the expected value of the move is 3. This can be obtained using the bare minimax procedure or as a consequence of (b).

(b) 8 leaves are evaluated. All leaves (and nodes) that are cut off are marked with dashed line on figure 2. The winning path is a_0, b_0, c_0, d_1 and either e_2 or e_3 at the end. The leaves, for which static values did not need to be computed are striked out.

(c) 15 leaves are evaluated. The only node which is cut off (and for which the static evaluation is not calculated) is the leaf e_0. The winning path is the same.

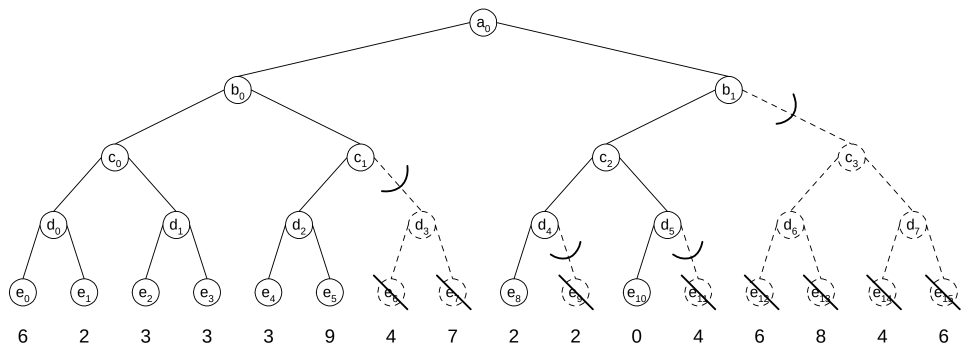

Figure 2: The $\alpha - \beta$ pruned tree.

4. **Three-Player Games**

   (a) See figure 3.

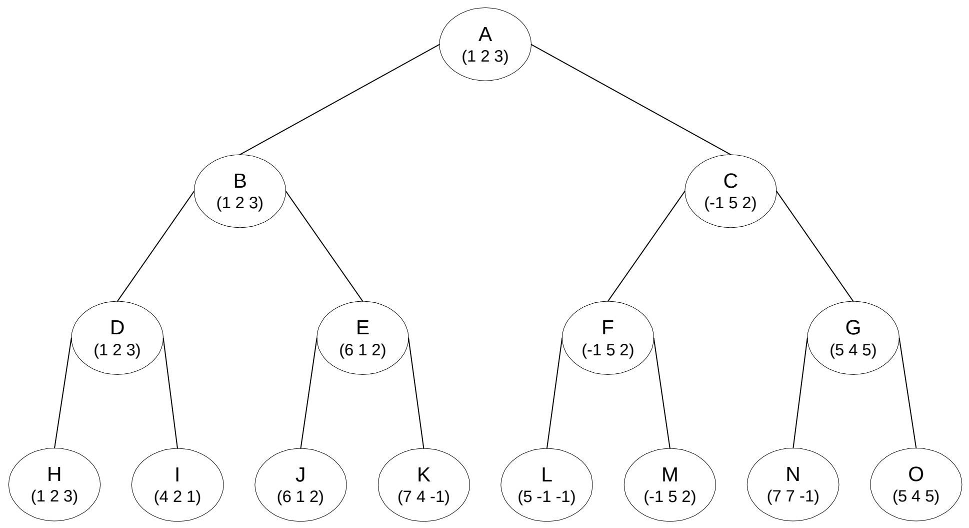

Figure 3: Completed three-player game tree.

   (b) If no alliances are allowed, the minimax algorithm turns into a kind of nMax algorithm, i.e. At each level, the player whose turn it is tries to maximize his/her own performance, without respect to the performance of the others. Here is an example pseudo code for evaluating node x, if it is player p's turn:

```
i. $max \leftarrow -\infty$
ii. for all childs $c$ of $x$ do if $c[p] > max$ then $\{max \leftarrow c[p];\ maxC \leftarrow c\}$
iii. return $maxC$
```

   (c) Alliances can make the outcome of the game vary very much. A player can start from choosing to maximize their partner's performance when they had already maximized theirs, or worse, can disregard themselves and optimize only the allied party. This can lead to fairly unpredictable results, including, but not limited to tricking the third player to lose, even if he was in a winning position. The volatile nature of the alliance itself is another factor which a player has to factor in while making a move, just in case that his alliance turns against him in the middle of the game.

Grading Most people did well on this question.

For part A, almost everyone got it correct except for one or two who did not maximise according to the specific player has the turn at each level.

For part B, the most common mistake is not specifying the term *maximizing each player* well enough. It is vital to show that state should be a triple, and at different level according to the turn of the game, select the child state which will maximize the value in the triplet for the current player. Another issue people missed out is the evaluation of terminal nodes which has to be treated differently from internal nodes.

For part C, the most common mistake is to describe how to incorporate the situation of alliances formation and termination into the current solution without realizing the additional complexities that has occured.

5. **Constraint Satisfaction Problems with Heuristics**

With a dictionary of more than 50000 words, we obviously can't apply the most-constrained-variable or least-constraining-value heuristics in the naive way. We can, however, "approximate" these heuristics. The most-constrained-variable heuristic selects the variable with the *fewest* possible values (after forward-checking, or some similar pre-processing step). Forward-checking gets too expensive when there are so many words to consider, however, so one way to see how many words are still viable for some variable would be to select k words at random, and count how many fit in the slot given the choices made so far. The value of k would need to be large enough for some of the slots to accomodate some of the sample words, or it the heuristic wouldn't provide any information. From there, depending on the usefulness of the most-constrained-variable heuristic, there would be performance trade-offs: by increasing k, the heuristic value would be more accurate, but each search step would take longer.

If you want to generate many crossword puzzles – I have one friend whose dream is to set up a program which generates a steady income for him by generating crossword puzzles every day and e-mailing them off to newspapers automatically – you could set up a database where the dictionary is clustered on different criteria, for example, all three to eight-letter words with the letter A, B, .. Z at each position. That would give 858 clusters, which could be computed and stored in linear time (with respect to the size of the dictionary). To find out how many words are still allowed for the slot "_A___", then, you could find the size that cluster in constant time. Deeper into the search tree, this technique could be combined with the approximation technique above to provide fairly accurate heuristic values.

The least-constraining-value heuristic chooses a value that rules out the smallest number of values in variables connected to the current variable by constraints. Similar techniques to those described for the most-constrained-variable heuristic can be used here. For each candidate word, an algorithm could quickly approximate the number of words left remaining for each of the intersecting slots, and could choose the one which imposes the fewest constraints, i.e., leaves the *most* candidate words in the intersecting slots.

There are definite trade-offs between the usefulness of a heuristic and the time required to compute it. How to best determine which heuristic to use for a real-world search problem is an open problem, and the topic of much current research.

Grading Many people had small problems with this question. Some assumed that the most constrained variable is the one with the largest number of constraints, but it's actually the one with the fewest values allowed. Counting constraints is a fairly weak approximation to the actual heuristics.

Doing explicit forward-checking (or worse, arc-consistency) is extremely expensive, given the size of the domain. The important thing to keep in mind is that the heuristics are are only heuristics, and that if it's very expensive to compute heuristic values, it's unlikely that the advantage to search provided by the heuristic. Linear overhead in the size of the dictionary is *not* and efficient way to implement a heuristic!

I gave full marks to anybody who gave a clear description of useful techniques for making both heuristics more effective/efficient. I gave 13 marks if the answer showed an understanding of the heuristics, but provided techniques that were either too weak. I gave 14 marks if somebody clearly described how to use "letter frequencies" (e.g., a word with a 'q' is not likely to have many values) but didn't describe the approximation or data structure techniques described above. I gave 10 marks to answers that applied the heuristics in the most obvious and naive way, i.e., very inefficiently. I also took off a mark or two if an answer showed a good understanding of one heuristic but not the other.