CS4725/CS6705 - Fall 2017

Lab Activity # 2

Name: SAMPLE SOLUTIONS

Student ID: ____________________

This lab activity will involve completing a small programming task and then answering a few questions.

Note: Your submissions (one Java file, submitted through Desire2Learn, and your responses to Questions 2 and 3, submitted on paper) are to be completed by 4:00 pm on **Wednesday, October 11, 2017**, but you are encouraged to finish as soon as possible.

---

1. This first exercise involves a short Java programming problem. The expectation is that you will have to write only about 10-15 lines of Java code in order to complete the task. However, you should spend some time testing your program to ensure that it works correctly.

   The goal is to fill in the missing lines in an implementation of the minimax algorithm with alpha-beta pruning.

   - On one of the lab machines, create a new directory for yourself and move to that directory. For example:

     ```
     mkdir AI-Lab2
     cd AI-Lab2
     ```

   - Copy the two Java files from the `/fcs/courses/cs4725/F17/minimax` directory into your directory.

     `cp /fcs/courses/cs4725/F17/minimax/* .` (Note the space and period at the end)

   - Note that the `alphaBetaSearch` algorithm is slightly different from what is in the textbook and our lecture slides, in that it returns the **value** of the root node instead of returning a choice of action.

- In the file `Minimax.java`, add the necessary code to the `maxValue` and `minValue` methods in the places indicated. For your reference, the relevant pseudocode algorithms from our lecture slides are presented below.

```
function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← -∞
    for each action in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(state,action), α, β))
        if v ≥ β  then return v        // ignore/prune all other actions
        α ← MAX(α, v)
return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each action in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(state,action), α, β))
        if v ≤ α  then return v        // ignore/prune all other actions
        β ← MIN(β, v)
return v
```

- Compile and run your program to test it.

  Note that the code is set up so that running `java Minimax` will test the program with a simple game tree that is provided, but running it with a parameter (*e.g.*, `java Minimax 1234`) will test it with a game tree that has random values in the leaf nodes (using the parameter as the seed for the random number generator).

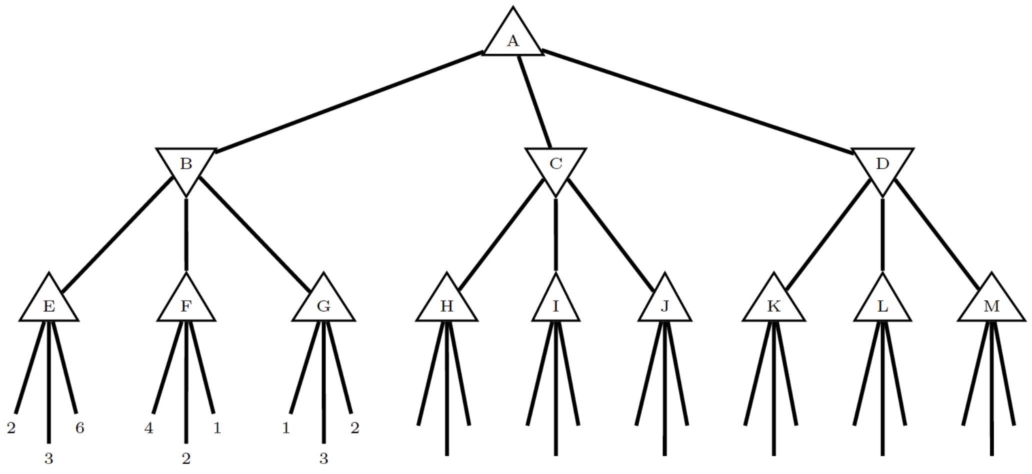
```
javac *.java     or     javac *.java
java Minimax            java Minimax 1234
```

  Note: You can also experiment with your own game trees by modifying the code in the `buildAlphaBetaTreeExample` method. (Please do not change the code in the `buildRandomAlphaBetaTreeExample` method.)

- In the `buildRandomAlphaBetaTreeExample` method, you can set verbose to true if you want to print the random numbers as they are generated. Please change this back to false before submitting your file, however.

- Submit **only** your `Minimax.java` file on Desire2Learn through `Assessments` → `Assignments` → `Lab #2`.

2. Consider the partially filled game tree pictured below.



- Fill in example values for the remaining terminal nodes so that the alpha-beta pruning algorithm would **not** be able to do any pruning at all. In other words, the algorithm would have to visit every single terminal node in order for us to be certain that it would choose the best action for MAX at node A.
- (Optional) Once you think you have found appropriate values for the terminal nodes, you can try entering your values in the `buildAlphaBetaTreeExample` method and then compile and run your program, to confirm that no pruning will take place.

[See the next page for solutions.]

Here is one possible set of values for the children of nodes H-M:

```
H: 2, 4, 6
I: 1, 3, 5
J: 0, 2, 4
K: 6, 8, 10
L: 5, 7, 9
M: 4, 6, 8
```

Another set of values would be:

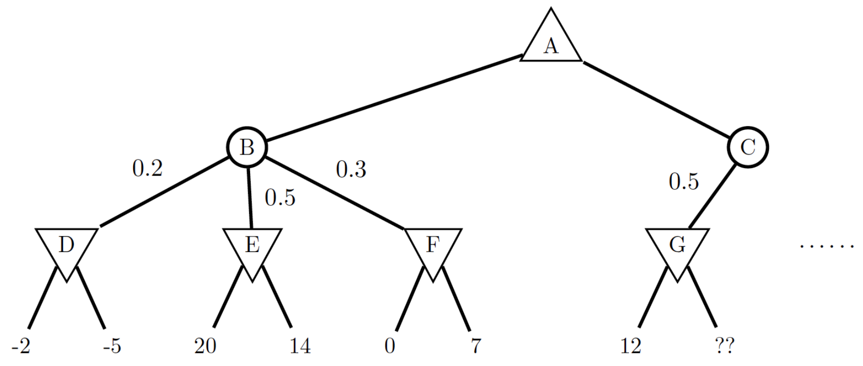
```
H: 6, 6, 6
I: 5, 5, 5
J: 4, 4, 4
K: 7, 7, 7
L: 6, 6, 6
M: 5, 5, 5
```

The key points are:

- At least one child of H must have a value greater than 3.
- At least one child of I must have a value greater than 3, **and** the first two children of I must be less than or equal to the largest child of H.
- At least one child of J must have a value greater than 3, **and** the first two children of J must be less than or equal to both the largest child of H and the largest child of I.
- At least one child of K must have a value greater than 3 and greater than C's value (which is the minimum of the max of H's children, the max of the I's children, and the max of J's children).
- At least one child of L must satisfy the same conditions as mentioned for K above. Also, the first two children of L must be less than or equal to the largest child of K.
- At least one child of M must satisfy the same conditions as mentioned for K above. Also, the first two children of M must be less than or equal to both the largest child of K and the largest child of L.

3. When we talked in class about pruning in game trees, we were focusing on trees that had only MAX and MIN nodes. This question considers the possibility of pruning in games that also have chance nodes.

   Consider the game tree below, which includes chance nodes B and C. Suppose that we know in advance that all terminal nodes will have values between -20 and 20, inclusive.



   We are exploring the nodes in this game tree in a depth-first order. Suppose that we have already explored the entire left side of the tree (node B and its descendants). On the right side, we have seen the first child of G and we know that 0.5 is the probability associated with the edge from C to G.

(a) When we explore the second child of node G, what **range** of values could it have that would allow us to conclude for certain that there is no need to explore any other children that node C might have?

Explain the reasoning behind your answer, showing any necessary calculations.

After exploring the left side of the tree, we know that:
$value(D) = min\{-2, -5\} = -5$
$value(E) = min\{20, 14\} = 14$
$value(F) = min\{0, 7\} = 0$
$value(B) = 0.2(-5) + 0.5(14) + 0.3(0) = -1 + 7 + 0 = 6$

To answer this question, we need to know what value the second child of G could have that would tell us for certain that the value of C could never be as high as the value of B, which is 6.

The maximum possible value that C could have would be if all of its other children had their maximum possible value, 20. In this case, the value of C would be $0.5(value(G)) + 0.5(20) = 0.5(value(G)) + 10$.

We can conclude that C has a value that is no larger than the value of B if

$0.5(val(G)) + 10 \leq 6 \rightarrow 0.5(val(G)) \leq -4 \rightarrow val(G) \leq \frac{-4}{0.5} \rightarrow val(G) \leq -8$

Therefore, if the second child of G has any value x, $-20 \leq x \leq -8$, we could conclude that there is no need to explore any other children of node C, as C would have a worse expected value than B for the MAX player.

(b) If the probability on the edge from C to G were changed from 0.5 to 0.4, how would that change your answer to part (a)?

If the probability were changed to 0.4, the relevant calculations would be

$0.4(val(G)) + 0.6(20) \leq 6 \rightarrow 0.4(val(G)) \leq 6 - 12 \rightarrow val(G) \leq \frac{-6}{0.4} \rightarrow val(G) \leq -15$

Therefore, if the second child of G has any value x, $-20 \leq x \leq -15$, we could conclude that there is no need to explore any other children of node C, as C would have a worse expected value than B for the MAX player.