Solution 2, CIS 471/571

Question 1 (20 points)

Prove the following assertion: For every game tree, the utility obtained by MAX using minimax decisions against a suboptimal MIN will be never be lower than the utility obtained playing against an optimal MIN.

Solution

Consider a MIN node whose children are terminal nodes. If MIN plays suboptimally, then the value of the node is greater than or equal to the value it would have if MIN played optimally. Hence, the value of the MAX node that is the MIN node’s parent can only be increased. This argument can be extended by a simple induction all the way to the root. If the suboptimal play by MIN is predictable, then one can do better than a minimax strategy. For example, if MIN always falls for a certain kind of trap and loses, then setting the trap guarantees a win even if there is actually a devastating response for MIN.

Question 2 (20 points)

Prove that alpha-beta pruning takes time $O(b^{\frac{m}{2}})$ with optimal move ordering, where $m$ is the maximum depth of the game tree and $b$ is the branch factor.

Solution

Even though we have a perfectly ordered list of children at each node, the algorithm will have to explore all the child nodes of the 1st player to find the best value. Thus will have a branching factor of $b$.

For the second player, it is enough to expand the first child. Because that value, being the best value, will make all values but the first value of the previous node pruned. Thus will have an effective branching factor of 1.

This means that we will be having two types of threads down the tree when optimal ordering is there.

Type A: will have number 1 in even levels and some other number $x$ ($1 < x < b$)
Type B: will have number 1 in odd levels and some other number $y$ ($1 < y < b$)
Given that we are at the ith level,

i. Number of type A threads: \(b^{i/2}\) (Note: we have to take the ceiling value here because we are counting threads that have 1 in even levels.)

ii. Number of type B threads: \(b^{i/2}\) (Note: we have to take the floor value here because we are counting threads that have 1 in odd levels.)

Now note that the special thread 111111....1 has been counted twice! (Both as type A and type B)

Thus if we have m levels the number of nodes visited will be;

\[
O(b^{m/2} + b^{m/2} - 1) [-1 to prevent 1111...1 thread being counted twice]
\]

Simplifying we get,

\[
O(b^{m/2} + b^{m/2} - 1) = O(2b^{m/2} - 1) = O(b^{m/2})
\]

Thus the time complexity is \(O(b^{m/2})\) when all the moves are optimal.

**Question 3 (60 points)**

Solution (Pseudo Codes):

The problem could be solved as a CSP problem with MRV and FC heuristics for faster result. We use backtracking to iterate through the domain choices.

- Define “P” as particular instance of problem to be solved
- Define update(c,P) as a function for updating the root for problem P (The problem could be nested and P could be part of a sub-problem)
- Define Domain \(D = \{0,1,...,8,9\}\) for each variables in the problem
- Define CSP(P,v) as a boolean variable to check whether the constraints are satisfied by the problem.

We apply the simple backtracking algorithm to our problem

- Select a root variable and assign it a value
  - If the assignment doesn’t break any constraints (e.g. no zero for left-most variables), we continue with the process else we reassign the root variable
  - If we have assigned all the variables and all constraints are satisfied, we have obtained the solution. We exit the backtracking function
Select the next variable to assign and assign it a value. Again, check for constraints and continue the process if no constraints are broken. Else reassign the variable.

A simple backtracking algorithm can get better result if we select the variable and value to be assigned it using some heuristics. We can apply some heuristics to the backtracking algorithm above to get faster results. The heuristics we use for selecting the variable to be assigned is given below:

- While selecting the variable to be assigned a value, chose the variable with least number of possible values. This means simply updating the domain for each variable in every step based on the assigned domain values and constraints: Minimum Remaining Values (MRV) [heuristic I]
- We could also chose variables that imposes the most constraints on the remaining variables. This could be done by selecting variable that appear in most number of constraints and is still unassigned. (Most Constraining Variable) [heuristic II]
- We select a variable with MRV first and only use heuristic II (MCV) to break a tie-break and on further tie-break use a random selection

We could also apply Forward Checking heuristic to stop backtracking on cases where the failure is imminent and thus reduce backtracking search path. The FC based heuristic we applied are:

- After selecting a variable, we pre-assign a value to that variable and then check whether the domain for any other variables is restricted or not. We select a value that provides maximum domain value options to other variables. This is simply updating the domain count for pre-assignment and then selecting the value which returns most number of domain options. Again, tie are broken randomly

Further steps for making the solution more optimal are possible [e.g. arc consistency and constraint propagation] but is left as exercise.

More details:

Assigned(var)
   Returns true if the variable is assigned

nextVariable()
   returns the next available variable based on MRV heuristic and MCV heuristic

Update(c,P)
   variable = nextVariable()
   if Assigned(variable) is False, then update the value of variable as c
CSP(\(P,v\))

- Check whether the first variable in addend/result is assigned zero value or not
  - if yes, return \(False\) (constraint is not satisfied)
- Check whether all variables are assigned or not
  - If \(True\), replace the variables with numbers and check the result. If the sum is equal to result, then return \(True\) (all constraints are satisfied) else return \(False\) (there are some constraints which fail)
  - If \(False\), then check for additional constraints within the incomplete assignment
    - If the current variables satisfy arc consistency return \(True\) else return \(False\)

*Note: We don't need to check for unary constraints within the CSP function as these constraints are used to constrain the domain of each variable*