Solution 2

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.

Consider a MIN node whose children are terminal nodes. If MIN plays sub optimally, 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 $O(b^{(m/2)})$ with optimal move ordering, where $m$ is the maximum depth of the game tree and $b$ is the branch factor.

The perfect order means that for the children of the Min node (beta), it is in ascending order, therefore only the left most child needs to be visited, the others can be pruned. The value of Min node will be determined by the left mode child. However, the algorithm needs to compare all values of Min nodes to get the value of Max, therefore the time complexity is $b$.

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 $1$.

We define a thread as a set of selections done by the algorithm. Eg: selected 2nd child at level 1, selected 5th child at level 2, selected 1st child at level 3. Will give the thread 251.
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 \( i \)th 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;

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

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)

Programming assignment:

Create a program that can solve a cryptarithmetic problem. Each letter must be assigned a unique digit from zero to nine, and no term is allowed to have leading zeros. (For example, F and E cannot be zero in the following:)

E I G H T
- F O U R
----------
F O U R

Solution (Pseudo Codes):

1. The problem could be solved as a CSP problem with backtracking and FC heuristics for faster result. We use backtracking to iterate through the domain choices. Design data structures to model constraints, variables, available domain for each variable. In detail, your data structures (hashmap and array are enough) should be able to track the state of them, because in the
process of backtracking, the states of them keep being updated by assigning a value to a variable or simply going back to previous state.

- 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 variable in the problem
- Define CSP(P,v) as a Boolean variable to check whether the constraints are satisfied by the problem.

\[\text{Assigned}(\text{var})\]

Returns true if the variable is assigned

\[\text{nextVariable}()\]

returns the next available variable based on MRV heuristic and MCV heuristic

\[\text{Update}(c,P)\]

\[\text{variable} = \text{nextVariable}()\]

if Assigned(\text{variable}) is False, then update the value of variable as c

\[\text{CSP}(P,v)\]

- 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

If we apply the simple backtracking algorithm to our problem. You should notice that the first letter of minuend is always 1.

- 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.

Backtracking algorithm can get better result if we select the variable and value to be assigned it using some heuristics. The heuristics we use for selecting the variable to be assigned is given below:

- Design two heuristic functions, MRV and LCV. Given current states of constraints, variables, and available domains for each variable, MRV implements “most constrained variable” and “most constraining variable” heuristics to choose next unassigned variable to try. Given a chosen variable, LCV implements “least constraining value” to choose a value for the variable. Both MRV and LCV will use the data structures you defined in part 1. For example, MRV loops over each unassigned variable, calculate the number of constraints related to it, sort calculate the number of constraints in descending order, then return the first one as most constrained variable. If there are ties after sorting by “most constrained variable”, use similar procedure to calculate “most constraining variable”, sort in descending order, and choose the first one. If there are still tie, you might use some arbitrary criterion to choose one to make your program consistent. Otherwise, randomly choosing one from ties will make your program inconsistent.

- Design backtracking procedure. The procedure should do following tasks: 1. Check if all variable has been assigned (concrete implement depends on your data structure). If so, one valid solution is found and return to previous state (previous recursion level). 2. Use MRV to generate a variable v to try. 3. Use LCV on v to determine the sequence for trying available values of v. 4. Choose a value and assign it to v. Then, test if the assignment is consistent with all available constraints. If so, recurs to next level. If not, try next possible value.

Above 3 core components should give you a guide on how to implement your solution, though there are still details for you to figure out. Designing good data structures for constraints, variables, and values are crucial, because your implementation of MRV, LCV, and all other facility function (e.g. check if an assignment is consistent with constraints) depend on it.

Following figure shows all the constraints, variables, and domains you need to model in the problem (It’s a different problem, but exactly same ideas.):
Finally, if you implement forward checking or arc consistency, apply them in the appropriate position to stop backtracking on cases where the failure is imminent. The forward checking heuristic can be implemented as following: 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 the most number of domain options. Again, ties are broken randomly.

We use three tests for grading:

Test1: EIGHT-FOUR = FOUR, 32 solutions

Test2: MONEY-SEND = MORE, 1 solutions

Test3: JUICY-AABB = CCDD, 0 solutions

So far, the best implementation achieves less 400 steps for the 3 test cases. Even without forward checking or arc consistency, the program should finish less than 1000 steps.