Q1. Worst-Case Backtracking

Consider solving the following CSP with standard backtracking search where we enforce arc consistency of all arcs before every variable assignment. Assume every variable in the CSP has a domain size $d > 1$.

(a) For each of the variable orderings, mark the variables for which backtracking search (with arc consistency checking) could end up considering more than one different value during the search.

(i) Ordering: $A, B, C, D, E, F$

Since we are enforcing arc consistency before every value assignment, we will only be guaranteed that we won’t need to backtrack when our remaining variables left to be assign form a tree structure (or a forest of trees). For the first ordering, after we assign $A$ and $B$, the nodes $C, D, E, F$, form a tree. For the second ordering, after we assign $B$, the nodes $A, C, D, E, F$ form a tree.

(ii) Ordering: $B, D, F, E, C, A$

(b) Now assume that an adversary gets to observe which variable ordering you are using, and after doing so, chooses to add one additional binary constraint between any pair of variables in the CSP to maximize the number of backtracking variables in the worst case. For each of the following variable orderings, select which additional binary constraint the adversary should add. Then, mark the variables for which backtracking search (with arc consistency checking) could end up considering more than one different value when solving the modified CSP.

(i) Ordering: $A, B, C, D, E, F$

The adversary should add the additional binary constraint:

When solving the modified CSP with this ordering, backtracking might occur at:

(ii) Ordering: $B, D, F, E, C, A$

The adversary should add the additional binary constraint:

When solving the modified CSP with this ordering, backtracking might occur at:
By adding the edge $CE$, now only after we assign $B, D, F$, the remaining nodes $A, C, E$ form a tree.
**Q2. Satisfying Search**

Consider a search problem \((S, A, Succ, s_0, G)\), where all actions have cost 1. \(S\) is the set of states, \(A(s)\) is the set of legal actions from a state \(s\), \(Succ(s, a)\) is the state reached after taking action \(a\) in state \(s\), \(s_0\) is the start state, and \(G(s)\) is true if and only if \(s\) is a goal state.

Suppose we have a search problem where we know that the solution cost is exactly \(k\), but we do not know the actual solution. The search problems has \(|S|\) states and a branching factor of \(b\).

(a) (i) Since the costs are all 1, we decide to run breadth-first tree search. Give the tightest bound on the worst-case running time of breadth-first tree search in terms of \(|S|\), \(b\), and \(k\).

The running time is \(O(b^k)\) This is the normal running time for BFS.

(ii) Unfortunately, we get an out of memory error when we try to use breadth first search. Which of the following algorithms is the best one to use instead?

- Depth First Search
- Depth First Search limited to depth \(k\)
- Iterative Deepening
- Uniform Cost Search

Firstly, notice that depth first search limited to depth \(k\) will find the solution, since we know that the solution is \(k\) moves long. Depth First search would explore paths of length larger than \(k\), which is useless computation. Iterative Deepening would first explore paths of length 1, then 2, and so on, which is useless computation. Uniform Cost Search is equivalent to BFS when the costs are 1, and so will probably also have an out of memory error.

Instead of running a search algorithm to find the solution, we can phrase this as a CSP:

Variables: \(X_0, X_1, X_2, \ldots X_k\)

Domain of each variable: \(S\), the set of all possible states

Constraints:

1. \(X_0\) is the start state, that is, \(X_0 = s_0\).
2. \(X_k\) must be a goal state, that is, \(G(X_k)\) has to be true.
3. For every \(0 \leq i < k\), \((X_i, X_{i+1})\) is an edge in the search graph, that is, there exists an action \(a \in A(X_i)\) such that \(X_{i+1} = Succ(X_i, a)\).

With these constraints, when we get a solution \((X_0 = s_0, X_1 = s_1, \ldots X_k = s_k)\), the solution to our original search problem is the path \(s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k\).

(b) This is a tree-structured CSP. Illustrate this by drawing the constraint graph for \(k = 3\) and providing a linearization order. (For \(k = 3\), the states should be named \(X_0, X_1, X_2, \text{ and } X_3\.)

\[
\begin{array}{cccc}
X_0 & X_1 & X_2 & X_3 \\
\end{array}
\]

Linearization Order: \(X_0, X_1, X_2, X_3\) or \(X_3, X_2, X_1, X_0\), although others are possible.
(e) We can solve this CSP using the tree-structured CSP algorithm. You can make the following assumptions:

1. For any state \( s \), computing \( G(s) \) takes \( O(1) \) time.
2. Checking consistency of a single arc \( F \rightarrow G \) takes \( O(fg) \) time, where \( f \) is the number of remaining values that \( F \) can take on and \( g \) is the number of remaining values that \( G \) can take on.

Remember that the search problem has a solution cost of exactly \( k \), \( |S| \) states, and a branching factor of \( b \).

(i) Give the tightest bound on the time taken to run the forward pass, in terms of \( |S| \), \( b \), and \( k \).

The running time for the forward pass is \( O(\frac{k|S|}{\text{---}}) \)

The forward pass assigns values to variables in order, and then enforces consistency of an arc so that the values for the next variable are all legal. When enforcing the consistency of \( X_{i+1} \rightarrow X_i \), \( X_i \) has already been assigned, so the time taken is \( O(|S| \cdot 1) = O(|S|) \). This is done for each of the \( k \) arcs, for a total of \( O(k|S|) \) time.

We would also accept \( O(kb) \), on the basis that after assigning a variable to \( X_i \), you only need to restrict \( X_{i+1} \) to the \( b \) possible values that follow \( X_i \).

(ii) Give the tightest bound on the time taken to run the backward pass, in terms of \( |S| \), \( b \), and \( k \).

The running time for the backward pass is \( O(\frac{\text{---}k|S|^2}{\text{---}}) \)

The backward pass simply enforces the consistency of \( k \) arcs, each of which takes \( O(|S|^2) \) time, for a total of \( O(k|S|^2) \) time.

(iii) Give the tightest bound on the time taken to run the forward pass, in terms of \( |S| \), \( b \), and \( k \).

The running time for the forward pass is \( O(\frac{\text{---}k|S|}{\text{---}}) \)

The forward pass assigns values to variables in order, and then enforces consistency of an arc so that the values for the next variable are all legal. When enforcing the consistency of \( X_{i+1} \rightarrow X_i \), \( X_i \) has already been assigned, so the time taken is \( O(|S| \cdot 1) = O(|S|) \). This is done for each of the \( k \) arcs, for a total of \( O(k|S|) \) time.

We would also accept \( O(kb) \), on the basis that after assigning a variable to \( X_i \), you only need to restrict \( X_{i+1} \) to the \( b \) possible values that follow \( X_i \).

(d) Suppose \( s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k \) is a solution to the search problem. Mark all of the following options that are guaranteed to be true after enforcing unary constraints and running arc consistency.

- The remaining values of \( X_i \) will be \( s_i \) and possibly other values.
- The remaining values of \( X_i \) will be \( s_i \) and nothing else.
- A solution can be found by setting each \( X_i \) to any of the remaining states in its domain.
- A solution can be found by executing the forward pass of the tree-structured CSP algorithm.
- None of the above

After enforcing unary constraints and running arc consistency, since this is a tree-structured CSP, we are guaranteed that all remaining values are part of some solution, but not necessarily all solutions. In addition, since arc consistency only eliminates impossible values, all the \( s_i \) values will still be present (since they are part of the solution \( s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k \)). Thus, \( X_i \) will have \( s_i \) and other values (corresponding to other solutions). However, we cannot set each \( X_i \) to any of the remaining states in its domain, if we set \( X_1 \) to a state \( s' \) and \( X_2 \) to \( s'' \), while we know there is some solution where \( X_1 = s' \), and some (possibly different) solution where \( X_2 = s'' \), we are not guaranteed that there is a solution where \( X_1 = s' \) and \( X_2 = s'' \).

The backward pass of the tree-structured CSP algorithm simply enforces consistency of some arcs. So, running full arc consistency will eliminate at least all the values that the backward pass would have eliminated. So, we can run the forward pass to find a solution.

(e) Suppose you have a heuristic \( h(s) \). You decide to add more constraints to your CSP (with the hope that it speeds up the solver by eliminating many states quickly). Mark all of the following options that are valid constraints that can be added to the CSP, under the assumption that \( h(s) \) is (a) any function (b) admissible and (c) consistent. Recall that the cost of every action is 1.
Either filled in or blank was accepted. Because of the ambiguity in question statement regarding acceptable constraints, this option was ignored.

Saying that there is some state that is a goal state - for example, the start state - this will allow us to make only as many transitions as necessary, and then all the remaining “extra” states can be set to the same state.

If we say

\[
\frac{(f)}{5}
\]

now suppose we only know that the solution will have \( \leq k \) moves. We do not need to find the optimal solution - we only need to find some solution of cost \( \leq k \). Mark all of the following options such that if you make single change described in that line it will correctly modify the CSP to find some solution of cost \( \leq k \). Remember, the CSP can only have unary and binary constraints.

- Remove the constraints “\((X_i, X_{i+1})\) is an edge in the search graph”. Instead, add the constraints “\((X_i, X_{i+1})\) is an edge in the search graph, OR \(X_i = X_{i+1}\)”.
- Remove the constraints “\((X_i, X_{i+1})\) is an edge in the search graph”. Instead, add the constraints “\((X_i, X_{i+1})\) is an edge in the search graph, AND \(X_i = X_{i+1}\)”.
- Remove the constraint “\(X_k\) is a goal state.” Instead, add the constraint “For every \(0 \leq i \leq k\), \(X_i\) is a goal state”.
- Remove the constraint “\(X_k\) is a goal state.” Instead, add the constraint “There is some \(i\), \(0 \leq i \leq k\), such that \(X_i\) is a goal state”.
- None of the above

If we say \((X_i, X_{i+1})\) is an edge AND \(X_i = X_{i+1}\), we are forcing all of the variables to be the same state \(s_0\) (since they all have to be equal to each other), so that cannot be right.

If we say \((X_i, X_{i+1})\) is an edge OR \(X_i = X_{i+1}\), then basically the constraint says that transitions have to be made, but we can also choose to not make a transition from \(i \rightarrow i + 1\) (in which case we would have \(X_i = X_{i+1}\)). This is exactly what we want - this will allow us to make only as many transitions as necessary, and then all the remaining “extra” states can be set to the previous state to satisfy the constraints.

It does not make sense to say that every state is a goal state - for example, the start state \(X_0 = s_0\) is usually not a goal state.

Saying that there is some \(i\) such that \(X_i\) is a goal state would work. It is not a unary or binary constraint (it is a constraint on all of the variables). Because of the ambiguity in question statement regarding acceptable constraints, this option was ignored. (Either filled in or blank was accepted.)