WA 2 Solutions

Om Khadka, U51801771
Collabs: N/A

Question 1: Correctness of Alpha-Beta Pruning (25 points)

Let s be the state of the game, and assume that the game tree has a finite number of vertices. Let v be the value produced by the minimax algorithm:

$$v = \texttt{Minimax}(s)$$

Let v' be the result of running Alpha-Beta Pruning on s with some initial values of α and β (where $-\infty \leq \alpha \leq \beta \leq +\infty$):

$$v' = \texttt{Alpha-Beta-Pruning}(s, \alpha, \beta)$$

Prove that the following statements are true:

- If $\alpha \leq v \leq \beta$ then $v' = v$
- If $v \leq \alpha$ then $v' \leq \alpha$
- If $v \geq \beta$ then $v' \geq \beta$

This means that if the true minimax value is between α and β, then Alpha-Beta pruning returns the correct value. However, if the tru minimax value if outside of this range, then Alpha-Beta pruning may return a different value. However, the incorrect value that Alpha-Beta pruning returns is bounded in the same manner that the true minimax value is (i.e. if the true minimax value is $\leq \alpha$ then the value produced by Alpha-Beta pruning is also $\leq \alpha$ and vice versa). Note that this implies that Alpha-Beta pruning will be correct with initial values of $(-\infty, +\infty)$ for (α, β).

Suppose a function, $P(s, \alpha, \beta)$ for a state s and an inital bounds α, β (with $\alpha \leq \beta$) with the following conditions:

Let $v = Minimax(s)$, and $v' = AB - Pruning(s, \alpha, \beta)$. Then:

1. If $\alpha \leq v \leq \beta$, then $v' = v$.
2. If $v \leq \alpha$, then $v' = \alpha$.
3. If $v \geq \beta$, then $v' \geq \beta$.

$P(s, \alpha, \beta)$ can be proven for all states of s via induction.

Base case: s is a terminal state.
In this case, both $Minimax(s)$ and $AB - Pruning(s, \alpha, \beta)$ simply returns the static evaluation of s, which can be shown as $u(s)$. Thus, $v = v' = u(s)$.

Assertion	**Explanation**
The case that $\alpha \leq v \leq \beta$	Base case definition.
$v' = u(s) = v$	Therefore, $v' = v$, satisfying **1**
The case that $v \leq \alpha$	Base case definition.
$v' = u(s) = v$	Since $v \leq \alpha$, it should follow that $v' \leq \alpha$, satisfying **2**
The case that $v \geq \beta$	Base case definition.
$v' = u(s) = v$	Since $v \geq \beta$, it follows that $v' = \beta$, satisfying **3**
$\therefore P(s, \alpha, \beta)$ holds for all terminal states s	All conditions are met.

Inductive case: s is a non-terminal state.
In this case, we can assume the inductive hypothesis of the following: that $P(s', \alpha', \beta')$ holds for every child state s' of s and for any bounds $\alpha' \leq \beta'$. We now prove that $P(s, \alpha, \beta)$ holds.
We can go at this no cases based on whether it's a maximizing or minimizing player's turn Automatic s.

The case that s is a MAX node.
Let the children of s be $s_1, s_2, \ldots, s_k$.
Let $v_i = Minimax_{s_i}$. By the minimax algorithm, $v = \max_i v_i$.
The AB-algo for a MAX node with initalize a variable *best_val* to $-\inf$ and then iterates through the children, updating *best_val* and α. It is important to note that it may prune a child s_j if the current *best_val* $\geq \beta$.

Let v' be the ultimate value returned by $AB - Pruning(s, \alpha, \beta)$.
By tracing the process, it can be shown that after process each child s_i, the current value *val* (which is $v'_i = AB - Pruning(s, \alpha, \beta)$) and the tentative *best_val* satisfies the inductive claim relative to the PARTIAL minimax value of the children processed at that point:

Assertion	**Explanation**
Let $V_{\text{sofar}} = \max v_1, \ldots, v_{i-1}$.	This is the true minimax value of the best child found so far.
Let b be the *best_val* AFTER processing children $s_1, \ldots, s_{i-1}$.	This is the AB value of hte best child so far.
b satisfies the following: **1.1** If $\alpha \leq V_{\text{sofar}} \leq \beta$, then $b = V_{\text{sofar}}$ **2.1** If $V_{\text{sofar}} \leq \alpha$, then $b \leq \alpha$ **3.1** If $V_{\text{sofar}} \geq \beta$, then $b \geq \beta$	The inductive claim for the loop. This will be proven by induction on the number of children processed. The base case (0 children) has $V_{\text{sofar}} = -\infty$, $b = -\infty$, which trivially satisfies **2.1**. The step follows from the main inductive hypothesis applied to child s_i.
Process child s_i. Call $v'_i = AB\text{-}Pruning(s_i, \alpha, \beta)$.	By the main inductive hypothesis of $P(s_i, \alpha, \beta)$, the value v'_i satisfies all three properties **1, 2, 3** with respect to v_i.
The algo updates *best_val* to $\max(b, v'_i)$.	

Consider the 3 global cases for the final value $v = \max_i v_i$.

The subcase that $\alpha \leq v \leq \beta$. Note that we're trying to prove that $v' = v$:

Assertion	**Explanation**
Since $v \leq \beta$, for all children $i, v_i \leq \beta$.	Because v is the maximum of all v_i.
For any child i w/ $v_i \leq \alpha$, inductive hypothesis **2** gives $v'_i \leq \alpha$.	These childre ncan't increase *best_val* above α if $b \leq \alpha$, or it would be correctly valued if $b > \alpha$.
Let s_m be a child where $v_m = v$. Since $\alpha \leq v_m \leq \beta$, by the inductive hypothesis **1**, this gives $v'_m = v_m = v$.	The true best child will be eval'ed correctly.
When processing s_m, $best\_val \geq v$.	Because $v'_m = v$.
For any child processed AFTER s_m, the pruning condition $best\_val \leq \beta$ will be FALSE.	Because $best\_val = v$ and $v \leq \beta$, and the condition for pruning is $bets\_val \geq \beta$. Since v will end up being $v \leq \beta$, the condition is only true if $best\_val > \beta$, which can't be. Thus, no pruning occurs AFTER finding the true max.
All children are evaluated. For any child i, $v_i \leq v$.	By definition of v.
The inductive hypothesis ensures that for any child i with $v_i \leq \alpha$, $v'_i \leq \alpha$, and for any child with $\alpha \leq v_i \leq \beta, v'_i = v_i$.	And thus, the max over all v'_i is v.
$\therefore v' = \max_i v'_i = v$.	This satisfies **1.1**.

The subcase that $v \leq \alpha$ (Proving now that $v' \leq \alpha$):

Assertion	**Explanation**
For every child $i, v_i \leq v \leq \alpha$.	
By the inductive hypothesis **2** being applied to each child, $s_i, v_i' \leq \alpha$.	
The algo's *best_val* is the maximum of these v_i'.	
$\therefore v' = \max_i v_i' \leq \alpha$.	Condition **2.1** is satisfied.

The subcase that $v \geq \beta$ ($v' \geq \beta$):

Assertion	**Explanation**
There exists at least one child s_m with $v_m = v \geq \beta$.	The true maximum value must be at least β.
Consider the first such child s_m that is evaluated by the algorithm.	We examine when this maximum value is encountered.
For all children before s_m, $V_{\text{sofar}} < \beta$.	If $V_{\text{sofar}} \geq \beta$, pruning would have occurred earlier.
When we call $AB\text{-}Pruning(s_m, \alpha, \beta)$, since $v_m \geq \beta$, by inductive hypothesis **3**, we get $v_m' \geq \beta$.	The pruned algorithm preserves the lower bound.
The algorithm then updates *best_val* to $\max(b, v_m') \geq \beta$.	The best value becomes at least β.
This immediately causes the remaining children to be pruned.	Because the pruning condition for a MAX node is *best_val* $\geq \beta$.
The algorithm returns $v' =$ *best_val* $\geq \beta$.	Condition **3.1** is satisfied.

The case that s is a MIN node

The proof for this is symmetric to that of the MAX node, with the roles of α, β being reversed and using min instead of max.

Thus, this proves that $P(s, \alpha, \beta)$ holds for the base (terminal) case and all other possible cases, and thusly proves the original statement.

Question 2: Oracle-Advised Two-Player Zero-Sum Adversarial-Search (25 points)

In deterministic adversarial search, we have to expand a game tree where nodes are either MAX (our turn to go) or MIN (our opponent's turn to go). This leads us to the Minimax algorithm, where we have to account for our opponents agency, and single-player solutions are not sophisticated enough to account for this agency. Imagine we have access to an oracle $O(s)$. This oracle takes a state as input and returns our opponent's optimal move (for them) if they were allowed to go in that state. This is called an oracle because it is never wrong, whenever we use the oracle it returns our opponent's optimal action 100% of the time.

Assume the game is deterministic. Design an algorithm to turn this two-player game into a single player game that calculates the optimal move we should make and prove its correctness.

An algorithm that works as the question above would look something like this:

```
function OracleMax(state s):
if s is a terminal state:
    return (utility(s), None)

bestValue = − inf
bestAction = None

for each action a that's available to MAX in s:
    s' = result(s, a) // The state after MAX's move a
    s" = result(s', O(s')) // The state after MIN's move (already optimize from oracle)
    (futureValue, None) = OracleMax(s")

    if futureValue > bestValue:
        bestValue = futureValue
    bestAction = a

return (bestValue, bestAction)
```

We can prove this algorithm via induction. Our claim to solve is the following:
For any deterministic, zero-sum game state s, the value v' and action a' returned by an Oracle-Advised adversarial search (we will call this OracleMax(s)) is equal to the standard minimax value and action for s.

Let d be the depth of the game tree from state s, where depth counts the num. of MAX's turns remaining. This definition aligns with the num. of moves the maximizing player has left, which aligns naturally with the recursive step of the algorithm.

BASE CASE: $d = 0$ (s is a terminal node):

Assertion	**Explanation**
$\text{OracleMax}(s) = (\text{util}(s), \text{None})$	By the algorithm's defintion, it returns the utility for a terminal state.
$\text{Minimax}(s) = \text{util}(s)$	By the definition of the Minimax algorithm, its output of a terminal state will also be its utility.
$\therefore \text{OracleMax}(s) \equiv \text{Minimax}(s)$ for $d = 0$.	The outputs are the same, thus proving the base case.

For the **Inductive Step**, we will assert the following **Inductive Hypothesis:**
Assume for all states s_k with depth $k \leq n$, $\text{OracleMax}(s_k) \equiv \text{Minimax}(s_k)$.
We are assuming that the algorithm will work properly for all shallower trees. The proof for this is below.

Inductive Step: For a state s with depth $n + 1$:

Assertion	**Explanation**
$A =$ the set of actions available to MAX in s.	Consider the root node of a tree of depth $n+1$.
For a given action $a \in A$: $s' = \text{result}(s, a)$, $a_{min} = O(s')$, and $s'' = \text{result}(s', a_{min})$	From evaluating each possible first move, we get the following: The state after MAX takes action a; The oracle-advised move of MIN from state s'; Annd the state arrived at after MAX's move a and MIN's response.
The depth of the tree from $s'' \leq n$.	MAX has just moved, and then MIN has also moved. The num. of MAX's turns left just decremented by AT LEAST 1.
By the inductive hypothesis, $\text{OracleMax}(s'') \equiv \text{Minimax}(s'')$.	The subtree rooted at s'' is shallower now, so our algorithm is working correctly.
Let $v_a = \text{value}(\text{OracleMax}(s''))$.	This is the value the algorithm should propagate back for taking in action a.

We now consider the Minimax calculation for the same action a:

Assertion	**Explanation**
$\text{Minimax}(s') = \min_{a' \in Amin} \text{value}(\text{Minimax}(\text{result}(s', a')))$.	From state s' (a MIN node), minimax selects the minimum value of the resulting states.
Since $a_{min} = O(s')$ is MIN's optimal move, $\text{Minimax}(s') = \text{Minimax}(s'')$.	The oracle guarantees a_{min} is the move that minimizes the value. Thus, the minimax value of s' is the same as the minimax value of s''.
$\therefore v_a = \text{Minimax}(s'') = \text{Minimax}(s')$.	Thusly, the value v_a determined by OracleMax for action a is the same what was computed from Minimax from state s'.
Core step of OracleMax(s) is: value $= \max_{a \in A} v_a = \max_{a \in A} \text{Minimax}(s')$.	Oracle algorithm chooses the max over the values of v_a.
Core step of Minimax(s) is: value $= \max_{a \in A} \text{Minimax}(s')$.	The minimax algorithm at a MAX node chooses the maximum of the minimax values of the successor states (which are the MIN nodes).
$\therefore$ value(OracleMax(s)) = value(Minimax(s)) action(OracleMax(s)) = action(Minimax(s)).	Since both algorithms are evaluating the same set of actions and assigning the same values Minimax(s') to each action a, they'll select the same maximizing action and compute the same value for the root node s.

This is of course assuming that we're going from a MAX node to a MIN node. When going from a MIN node to a MAX node, this proof is the same as above by symmetry. For a MIN node s, the algorithm OracleMin(s) that minimized over the values obtained after the oracle $O_{max}(s')$ provides MAX's optimal response, will correctly compute the minimax value $\min_a \max_{a'} V(\text{result}(\text{result}(s, a), a'))$, which is the minimax value of s.

The inductive proof is identical with the roles of max and min swapped. Thusly, this proves the original statement that this Orcacle-Advised algorithm can be reduced into a minimax search problem.

Question 3: Optimizing the AC3-algorithm (25 points)

The AC3 algorithm, whenever *any* value is deleted from the domain of variable X_i, puts *every* arc (X_k, X_i), even if each value of X_k is consistent with several remaining values of X_i. What if we stored, for every arc (X_k, X_i), the number of remaining values of X_i that are consistent with each value of X_k. Derive a way to update these numbers efficiently, and arrive at the conclusion that with this modification, arc consistency can be enforced with total runtime $O(n^2d^2)$.

The revised function would look something like this. Note that the main change in this function is within the **ReviseMod** function and how the queue works:

```
BOOLEAN ReviseMod((Xi, Xj)):
    revised = false
    for each value a in Domain[Xi]:
        // The val 'a' works only if it has AT LEAST one support in Xj
        if Counter[(Xi, Xj), a] == 0:
            remove a from Domain[Xi]
            revised = true
    return revised

// Main Algo
queue = all arcs (Xi, Xj) in the CSP
while queue is not empty:
    pop an arc (Xk, Xi) from queue
    if ReviseMod((Xk, Xi)):
        if Domain[Xk] is empty: return false
        for each neighbor Xj of Xk where Xj != Xi:
            push arc (Xj, Xk) onto queue
return true
```

The following data structures are used in this function:

Assertion	**Explanation**
Domain$[X_i]$	The set of values for variable X_i; Standard representation.
Counter$[(X_k, X_i), a]$	Int for arc (X_k, X_i), value $a \in$ Domain$[X_k]$; Stores the num of supports for val a of X_k in the domain of X_i.
On Initalization: For every arc (X_k, X_i), for every $a \in$ Domain$[X_k]$: Counter$[(X_k, X_i), a] = \|b \in$ Domain$[X_i]\|(a, b)$satisfies$C_{ki}\|$.	This is essentially saying that we precompute all support counts. This will take $O(ed^2)$ time, where e is the num of constraints (or arcs).

The main difference in this function is the new step added for **ReviseMod**. When a value v is deleted from Domain$[X_i]$ during ReviseMod$[(X_k, X_i)]$, the function will now update the counters for all arcs pointing to X_i.

Assertion	**Explanation**
Value v is deleted from Domain$[X_i]$.	This is the event that triggers for updates.
FOR EACH neighbor X_m of X_i (essentially arc (X_m, X_i)).	We're now letting all neighbors X_m know that a val in X_i's domain is gone.
FOR EACH VAL $a \in$ Domain$[X_m]$:	Now checking every valeu in the neighbor's domain.
IF (a, v) is consistent: Counter$[(X_m, X_i), a] - = 1$.	If value v was a support for a, then decrement counter.
IF Counter$[(X_m, X_i), a] == 0$, then add arc (X_m, X_i) to queue.	This is the new optimization. The arc is only added if a val lost its LAST support, making it so that revision is now necessary.

This update is done inside the **ReviseMod** function right after removing a value a (which is essentially v for its neighbors).
For complexity analysis, the proof is below:

Assertion	**Explanation**
Total num of arcs, e, is $O(n^2)$.	CSP with n vars will have $O(n^2)$ arcs.
Init cost is $O(e * d^2) = O(n^2 d^2)$.	For each arc, we check each $d \times d$ val pair.
Each value deletion is processed AT MOST once. Thus, possible value deletions is $O(nd)$.	A value, once deleted, is never added back. Thus, there'll be n vars, each with AT MOST d vals.

Cost per deletion:

Assertion	**Explanation**
Whenever val v is deleted from X_i:	
For each neighbor X_m : $O(n)$:	Var X_i can onyl have up to $O(n)$ neighbors.
For each val $a \in$ Domain$[X_m]$: $O(d)$:	Iterate over all values in the neighbor's domain.
Check consistency of (a, v) : $O(1)$:	Assume a precomputed constraint table to allow for constant-time checking.
Total cost : $O(nd) \times O(n) \times O(d) \times O(1) = O(n^2 d^2)$:	Num of deletions X Neighbors X Neighbors X Consistency check

Queue Ops:

Assertion	**Explanation**
Note: An arc (X_k, X_i) is added ONLY if a counter reaches 0.	
Max arcs in queue: $O(e \times d) = O(n^2 d)$:	For each of e arcs, each of the d vals in X_k can cause 1 insertion.
ReviseMod cost per arc : $O(d)$:	Max arc insertion X Cost per revision.
Total cost: $O(n^2 d^2) + O(n^2 d^2) = O(n^2 d^2)$:	Init cost + Deletion update cost + Queue procession cost.

Thus, this shows how by maintaining support counts and only revising an arc (X_m, X_i) when a val in X_m loses its last support in X_i, it is possible to enforce arc consistency with the same worst-case time complexity as the original function of $O(n^2 d^2)$. This makes this AC-3 function practically way more efficent by avoiding unnecessary arc revisions.

Question 4: CSP Reduction (25 points)

Prove that any n-ary constraint can be converted into a set of binary constraints. Therefore, show that all CSPs can be converted into binary CSPs (and therefore we only need to worry about designing algorithms to process binary CSPs).

Suppose a CSP $P = (X, D, C)$ where:
- $X = X_1, X_2, \ldots, X_n$ is a set of vars.
- $D = D_1, D_2, \ldots, D_n$ is their domains.
- C is a set of constrains, some of which are n-ary.

We now first start by making a new binary CSP $P' = (X', D', C')$:

Assertion	**Explanation**
Init: $X' = X, D' = D, C' =$ {all binary constraints from C}	Basically just make a copy of original CSP.
FOR EACH n-ary constraint $C_j \in C$ involving vars $X_{j1}, X_{j2}, \ldots, X_{jk}$: Create a synthetic variable Z_j and add to X'.	Process each non-binary constaint, and make a new var representing the compound assignment.
Domain: $D'(Z_j) = D_{j1} \times D_{j2} \times \ldots \times D_{jk}$.	All possible tuples satisying the OG domains.
Replace C_j with k binary constraints:	Decompose n-ary constraint into binary ones.
For $i = 1$ to k: Add constraint $R_{ji}(Z_j, X_{ji}) = \{(z, x) \mid z[i] = x\}$	Each ensures that the original var matches corresponding component of synthetic var.
All constrains in C' are now binary.	By our construction.

We now prove how this new CSP is equivalent to the original CSP:

The claim is that every solution to P corresponds to exactly one solution to P' and vice versa

The case of Forward direction: (A is a solution to P)

Assertion	**Explanation**
For each n-ary constraint C_j on $X_{j1}, \ldots, X_{jk}$: Let $z_j = (A(X_{j1}), \ldots, A(X_{jk}))$	Construct value for synthetic var.
$z_j \in D'(Z_j)$	By definition, since A satisfies domain constraints.
For each $i, (z_j, A(X_{ji})) \in R_{ji}$ since $z_j[i] = A(X_{ji})$, All original binary constraints in C remain satisfied.	All binary constraints are satisfied, and it is unchanged in construction.
$\therefore A' = A \cup \{Z_j = z_j\}$ is a solution to P'.	Extended assignment satisfies all constraints.

The case of Backwards direction: (A' is a solution to P')

Assertion	**Explanation**
Restrict A' to original vars $X : A = A'\|\|x$	Discard the synthetic vars.
For each original n-ary constraint C_j: Let $z_j = A'(Z_j) = (v_1, v_2, \ldots, v_k)$	Get tuple from synthetic variable.
For each original n-ary constraints R_{ji}: $A'(X_{ji}) = v_i = z_j[i]$	OG vars match components.
$\therefore A(X_{j1}, \ldots, X_{jk}) = (v_1, \ldots, v_k) = z_j$	Original assignment matches synthetic value.

From this information, we can now deduce that:

Assertion	**Explanation**
Since z_j was valid in P', it satisfies the original constraint C_j in P	The synthetic domain encodes valid tuples.
All binary constraints from P remain satisfied in P'	Preserved in construction.
$\therefore A$ is a solution to P	

This thus proves that any n-ary constraint can be converted into a set of binary constraints via the proof above. Since all CSPs can be converted into a set of binary CSPs, it is only necessary to make an algo for binary CSPs.

Extra Credit: Creating Crossword Puzzles (25 points)

Consider the problem of building a crossword puzzle (**creating** the crossword puzzle, **not** solving it). A crossword puzzle is defined as fitting words into a rectangular grid. The grid itself is given as part of the problem and specifies which squares are blank (and may be filled), and which are shaded (and therefore unavailable). You have at your disposal a list of words (i.e. a dictionary), and the task is to fill in the blank squares by using any subset of the list. Formulate this problem as:

a A general search problem. Choose an appropriate search algorithm and specify a heuristic function. Is it better to fill in blanks one letter at a time or one word at a time?

b A CSP. Should the variables be words or letters?

c Which formulation is better? Why?

A: General search problem

The first thing we'd need to figure out is the state representation. We can have it as the following
- The partially filled crossword, with some letters filled and other blank.
- The set of words remining in the dictionary.

The inital state is the empty crossword (all squares are empty), with a full dictionary.

The goal state is a crossword such that:
- Every square is filled.
- Every sequence of consecutive horiztonal and vertical blank squares makes a word in the dictionary.
- No word is used from the dictionary more than once (from my memory, crossword don't typically reuse words).

The for heuristic function, it could look something like this:
$h(s) =$ the number of unfilled word slots + (the number of conflicting intersections $\times$ 2)

The heuristic could simply just be the number of remaining blank squares, but that'd be weak. This improvement is now taking into account both the remaining number of unsolved words left, and the number of intersections that we've solved, putting a greater emphasis on solving these intersection (in my experience, the best way I solve words I don't know in crosswords is to try to solve other words intersecting that row. One may not know the word "orange" but if you're given " r ge", you'd have a better idea of what the word could be).

For the search algo itself, we would implement a backtracking search while checking the heuristic for word selection (something like AC-3 with heuristics).

Deciding between working with words or letters at a time, **words will be better in the case since:**
- This makes it such that the dictionary constraint is already satisfied at each step.
- Lowers branching factor compared to letter-by-letter checking.
- We can verify if an action is making meaningful process to the goal.
- Letter-by-letter checking would require also CONSTANTLY checking if partial words exist in the dictionary (not evne getting to also checking if that word is valid).

B: CSP

Vars: Each word slot in the crossword (horizontal and vertical).
Domains: For each var, domain will be the set of words from the dictionary that fits that sequence (for example, perhaps "pear" and "bear").
Constraints:
- The first constraint would be that no 2-word-var can have the same word value (like again crosswords don't reuse words).
- The second constraint would be that, for every intersection between a horizontal word H at position i and vertical word V at position j, the letter at $H[i]$ must be equal to the letter at $V[j]$.
  - $H[i], V[j] = \text{letters}, H[i] == V[j]$.

In a CSP, words would again be a better variable than letter because:
- The dictionary constraint would be implicilty encoded in the domains.
- Intersection constraints would naturally be binary constraints between word vars.
- Letter-by-letter formulation would require complex global contraints to make sure that each sequence is forming a valid dictionary word.

C: CSP would be better than a General Search Problem

The first reason is due to the question's way of representating states. The problem has variables (words slot), domains (possible words within those slots), and clear constraints (intersection must match and words have to be unique).

The next reason is due to CSP's constraint logic. It can efficently:
- Use arc consistency to prune domains where intersections conflict.
- Apply forward checking when assigning words.
- And use the heuristic to select the most constrained slots first.

In the search formulation, some word selections might lead to unsolvable intersection conflicts later. CSP constraint propagation checks these earlier, so these failures are avoided.

Also, crosswords naturally have a constraint graph where word slots are nodes and intersections are edges, which is pretty similar to the CSP representation.

Lastly, while both algorithms are exponential in their worst case, CSP will practically be much faster, as the var/val ordering and constraint propagation will usually perform better in this senario of such a concretely constrained hypothetical.

Essentially, think about what the constraints that need to be checked while solving a crossword puzzle: word uniqueness and intersection viability. CSPs are made to handle and check these constraints while

solving the problem too.