1 Discussion Questions

(a) What is the difference between Forward Checking and AC-3?

Forward checking and AC-3 both enforce arc consistency, but forward checking is more limited. Whenever a variable X is assigned, forward checking enforces arc consistency only for arcs that are pointing to X, which will reduce the domains of the neighboring variables in the constraint graph. Forward checking stops at this point, but AC-3 will continue to enforce arc consistency on neighboring arcs until there are no more variables whose domain can be reduced. As a result, FC ensures arc consistency of the assigned variable and its neighbors only, while AC-3 ensures arc consistency for the whole graph.

(b) Why does a tree-structured CSP take $O(nd^2)$ to solve?

If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time compared to general CSPs, where worst-case time is $O(d^n)$. To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a topological sort. Any tree with n nodes has $n-1$ arcs, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(nd^2)$.

(c) Why would one use the following heuristics for CSP?

(i) Minimum Remaining Values (MRV)

MRV: "Which variable should we assign next?"

- Fail fast
- We have to assign all variables at some point, so we might as well do hard stuff first (allowing us to prune the search tree faster/realize we need to backtrack)

(ii) Least Constraining Value (LCV)

LCV: "Which value should we try next?"

- We just want one solution.
- We don't try all combinations of value, so we should try ones that are likely to lead to a solution.

2 CSP: Air Traffic Control

We have five planes: A, B, C, D, and E and two runways: international and domestic. We would like to schedule a time slot and runway for each aircraft to either land or take off. We have four time slots: 1, 2, 3, 4 for each runway, during which we can schedule a landing or take off of a plane. We must find an assignment that meets the following constraints:

- Plane B has lost an engine and must land in time slot 1.
- Plane D can only arrive at the airport to land during or after time slot 3.
- Plane A is running low on fuel but can last until at most time slot 2.
- Plane D must land before plane C takes off, because some passengers must transfer from D to C.
- No two aircrafts can reserve the same time slot for the same runway.

(a) Complete the formulation of this problem as a CSP in terms of variables, domains, and constraints (both unary and binary). Constraints should be expressed implicitly using mathematical or logical notation rather than with words. Make sure to specify variables, domains, and constraints.

Variables: A, B, C, D, E for each plane.
Domains: a tuple (*runway type, time slot*) for runway type $\in$ {international, domestic} and time slot $\in \{1, 2, 3, 4\}$.
Constraints:

$$B[1] = 1$$

$$D[1] \geq 3$$

$$A[1] \leq 2$$

$$D[1] < C[1]$$

$$A \neq B \neq C \neq D \neq E$$

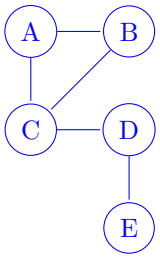
Note here we use B[1] to denote the second value of the tuple assigned to variable B. the time slot value, which is a number in $\{1, 2, 3, 4\}$

For the following parts, we add the following two constraints:

- Planes A, B, and C cater to international flights and can only use the international runway.
- Planes D and E cater to domestic flights and can only use the domestic runway.

(b) The addition of the two constraints above alters the CSP. Specifically, the domain does not need to include the runway type since this information is carried by the variable, and the binary constraints have changed. Determine the new domain and complete the constraint graph for this problem given the original constraints and the two added ones.

Variables: A, B, C, D, E for each plane.
Domain: $\{1, 2, 3, 4\}$
Constraint Graph:



Explanation of Constraints Graph: We can now encode the runway information into the identity of the variable, since each runway has more than enough time slots for the planes it serves. We represent the non-colliding time slot constraint as a binary constraint between the planes that use the same runways.

(c) What are the domains of the variables after enforcing arc consistency? Begin by enforcing unary constraints. (Cross out values that are no longer in the domain.)

Enforcing arc consistency with AC-3, we have the following domain as a result:

A	~~1~~	2	~~3~~	~~4~~
B	1	~~2~~	~~3~~	~~4~~
C	~~1~~	~~2~~	~~3~~	4
D	~~1~~	~~2~~	3	~~4~~
E	1	2	~~3~~	4

(explanation of process below)
Enforcing unary constraints (in an arbitrary order) first,

1. We cross out 2, 3, 4 from B's domain, adding arcs A → B and C → B to the queue.
2. We cross out 3, 4 from A's domain, adding arcs B → A and C → A to the queue.
3. We cross out 1, 2 from D's domain, adding arcs C → D and E → D to the queue.

Enforcing A → B, we cross out 1 from A's domain; add arcs B → A and C → A to the queue.
Enforcing C → B, we cross out 1 from C's domain; add arcs A → C, B → C, and D → C to the queue.
Enforcing B → A, no domain changes are necessary (all values remaining in B's domain have a consistent corresponding value in A's domain); no arcs are added.
Enforcing C → A, we cross out 2 from C's domain; add arcs A → C, B → C, and D → C to the queue.
Enforcing C → D, we cross out 3 from C's domain; add arcs A → C, B → C, and D → C to the queue.
Enforcing E → D, no domain changes are necessary.
Enforcing B → A, no domain changes are necessary.
Enforcing C → A, no domain changes are necessary.
Enforcing A → C, no domain changes are necessary.
Enforcing B → C, no domain changes are necessary.
Enforcing D → C, we cross out 4 from D's domain (there is no c in C's domain such that $c > 4$); add arcs C → D and E → D to the queue.
Enforcing A → C, no domain changes are necessary.
Enforcing B → C, no domain changes are necessary.
Enforcing D → C, no domain changes are necessary.
Enforcing C → D, no domain changes are necessary.
Enforcing E → D, we cross out 3 from E's domain; add arc D → E to the queue.
Enforcing D → E, no domain changes are necessary.
(phew!)

Note: For a general binary CSP, to enforce arc consistency before assigning any variables, you should add all arcs to the initial queue. For this problem, it can be easily seen that if there are no unary constraints, all the arcs will be consistent before any variable is assigned a value. As a result, we can start with the unary constraints and add arcs only for the related variables after enforcing the unary constraints.

(d) Arc-consistency can be rather expensive to enforce, and we believe that we can obtain faster solutions using only forward-checking on our variable assignments. Using the Minimum Remaining Values heuristic, perform backtracking search on the graph, breaking ties by picking lower values and characters first. List the (variable, assignment) pairs in the order they occur (including the assignments that are reverted upon reaching a dead end). Enforce unary constraints before starting the search.

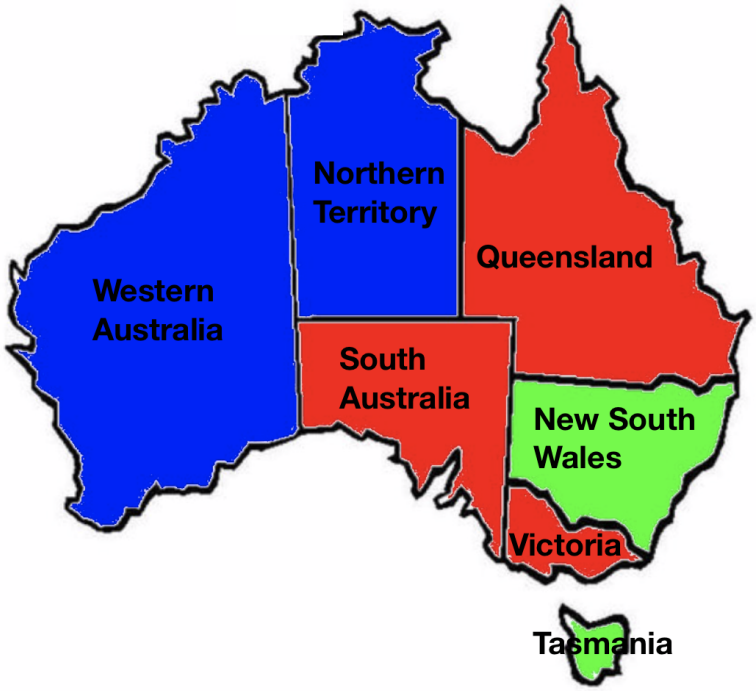
List of (variable, assignment) pairs:

(You don't have to use this table)

A	1	2	3	4
B	1	2	3	4
C	1	2	3	4
D	1	2	3	4
E	1	2	3	4

Answer: (B, 1), (A, 2), (C, 3), (C, 4), (D, 3), (E, 1)

3 Map Coloring with Local Search



Recall the various local search algorithms presented in lecture. Local search differs from previously discussed search methods in that it begins with a complete, potentially conflicting state and iteratively improves it by reassigning values. We will consider a simple map coloring problem, and will attempt to solve it with hill climbing.

(a) How is the map coloring problem defined (In other words, what are variables, domain and constraints of the problem)? How do you define states in this coloring problem?

- Variables: WA, NT, SA, Q, NSW, V, T (States in Australia)
- Domain: Green, Red, Blue
- Constraints: Adjacent countries can't have the same color assignment.
  e.g: Implicit: $WA \neq NT$
  Explicit: (WA, NT) $\in$ (red, blue), (red, green), (blue, red), (blue, green), (green, red), (green, blue)
- Problem state: a full coloring of the map (i.e., color assignments to all variables).

(b) Given a complete state (coloring), how could we define a neighboring state?

A neighboring state could be a full coloring of the graph with a different color assignment to only one variable.

(c) What could be a good heuristic be in this problem for local search? What is the initial value of this heuristic?

The heuristic could be the number of variable pairs that have conflicting colors. In the initial state, the following 3 pairs (WA-NT, Q-SA, SA-V) are conflicting, so the heuristic h = 3. (Note: there could be other possible heuristics for this problem.)

(d) Use hill climbing to find a solution based on the coloring provided in the graph.

Let h be our heuristic value.
In the original graph, we have 3 coloring conflicts as stated in (c). Depending on the search order, the assignment order might be different and the searched path lengths as well as coloring can also vary. We represent the coloring of states in a list with the following order: [WA, NT, Q, SA, NW, V, T]. Below are two examples of potential search paths.

- Step 1: h = 3. Conflicts are WA-NT, Q-SA, SA-V. We start with WA-NT. Coloring NT with Green would resolve the WA-NT conflict. Coloring: [B, G, R, R, G, R, G]
  Step 2: h = 2. Conflicts are Q-SA, SA-V. We can pick SA-Q pair and assign Blue to SA, which would resolve SA-Q and SA-V conflicts but will add coloring conflict for WA-SA pair. Still, it decreases the number of conflicts and is a better neighboring state. Coloring: [B, G, R, B, G, R, G]
  Step 3: h = 1. Conflict is just WA-SA pair. We can simply assign WA with Red to resolve this conflict, where we completed the search and found a solution to the problem. Coloring: [R, G, R, B, G, R, G]

We got pretty lucky in the search above and found a solution in 3 steps. However, local search may not always resolve conflicts optimally. Below is an example where it has to resolve the conflicts with more steps.

- Step 1: h = 3. Conflicts are WA-NT, Q-SA, SA-V. We start with WA-NT. Coloring WA with Green would resolve the WA-NT conflict. Coloring: [G, B, R, R, G, R, G]
  Step 2: h = 2. Conflicts are Q-SA, SA-V. We can resolve both of these conflicts by assigning Blue to SA, which will lead us to a better neighboring state with only one conflict: NT-SA. Coloring: [G, B, R, B, G, R, G]
  Step 3: h = 1. Conflicts are NT-SA. However, looking at all possible assignments to both of these two states, we see that no matter what color we assign to either one of them, we can't find a better neighboring state. Therefore the current iteration of hill climbing would end and we would restart from the initial state if we are applying random-restart hill climbing. An alternative is to use simulated annealing, which would allow us to sometimes move to states of higher heuristic value in order to escape local minima.

We see that in the second search, we need more steps to complete search. There are other possible search steps sequences depending on the choice on the order of conflicts to resolve, and the color assignment when resolving each conflict.

We can also use a genetic algorithm approach to find a solution to color the map. We could let the number of non-conflicting State pairs (in this case, WA-SA, NT-SA, Q-NT, SA-NW, Q-NW, NW-V) minus the number of conflicts in each state be our fitness function, and represent the variables in a list as above: [WA, NT, Q, SA, NW, V, T].

With a genetic algorithm, we first randomly select $2k$ full colorings of the graph with replacement (i.e. k pairs of parents), with probability proportional to their fitness function values. For each pair of parents, we would choose a cross over point $\in \{1...6\}$ to split the variable list in two halves and combine the color assignments with cross over to generate two offspring states.
(There are many other ways we could've implemented this crossover - this is just following the n-queens example from lecture.)

Then with a small probability, we could mutate the color assignments to the variables by one color, which means we would modify one of the values assigned to the variable list to a value in $\{Red, Green, Blue\}$. We would repeat the process until we find a good enough coloring that does not result in any conflicts within the map, or until it exceeds the max number of iterations.

(e) How is local search different from tree search?

Tree search keeps all unexplored alternatives on the frontier, where local search does not have a frontier but only keeps improving a single option until there's no better neighboring states.