Constraint Satisfaction

CSCI 2400 "Part 4"
Instructor: David Byrd

Disclaimer: Lecture notes can't and won't cover everything I say in class. You should attend class each day and use these for review or reinforcement.

7 Constraint Satisfaction Problems

7.1 Motivational example

Consider the game of Set, in which you must select cards that make a valid set of n cards according to some specific rules. The cards show symbols and have various attributes, for example: number of symbols, color of symbols, shape of symbols, and shading of symbols. Each attribute will have n possible values. For example, if you must choose three cards, there will be three variations of each attribute (three possible colors, etc).

The rule to follow is simple: n cards make a valid set if (and only if) for each attribute, the values on the cards are all the same, or the values are all different. For example, if you are selecting three cards, there will be three possible card colors. To follow the rule, either all three cards must be the same color (red, perhaps) or all must be different colors (red, green, blue). If two are the same but not the third (two red, one blue) then the rule is not followed, and the set is not valid.

Could you solve this problem with the previous search algorithms (uninformed or informed)? Yes, but not sensibly. Our basic search algorithms are goal-based with respect to state. We will define a state to be a selection of zero to n cards, and our actions will be to add a single card to the set. States which contain n cards and follow the "rule" for all attributes (color, shape, etc) will be goal states. A selection of zero cards will be the initial state.

What are the issues with this approach?

1. The search algorithm can only ask: "Is this a goal state?" It cannot tell if its current path leads to a goal, only if it has reached one already. In this problem, the goals are all at depth n (because we must select n cards before a state could possibly be a goal) and we have rules that *could* let us know early that our set cannot be valid. The search algorithm does not make use of this information.

2. The search algorithm is meant to find paths to goal nodes. There is no "path" in this problem; it is not sequential. That is, the selection of cards does not need to happen in any particular order. Only the final set matters. Thus search is solving a more complex problem that we do not care about.

3. Most search algorithms try to find the best path or best goal in some sense. There is no "best" here. All goals are equally good, so again the search will probably be doing extra work for no benefit.

Together, these issues mean that our search algorithm is basically performing a brute force approach: select three cards and see if that is a goal. If not, put the last card back and try again. Note that the search will have no idea *why* a set of cards is or is not a goal, or what is "wrong" or how to fix it. This search may not seem so bad with $n = 3$, but imagine we must pick $n = 100$ cards from ten thousand choices and each attribute (color, shading, etc) has one hundred variations. Brute force of all permutations of selecting one card at a time does not seem like a reasonable answer!

What if, instead of using search algorithms that treat the state as a "black box", we were to allow the algorithm to look within the state and inspect the values of each attribute? Then we could perform a sort of search, but one which could check all of the rules each time it performed an action, and backtrack immediately if a rule is broken!

7.2 Overview

We can define a Constraint Satisfaction Problem (CSP) as:

- X: a set of variables $\{x_1, x_2, \ldots, x_n\}$
- D: a set of domains for X, such that $x_1 \in D_1, x_2 \in D_2, \ldots, x_n \in D_n$
- C: a set of constraints, each one a pair $< scope, relation >$, where *scope* is the set of variables to which the constraint applies and $relation$ is the allowed value combinations for those variables.

For the Set example given above:

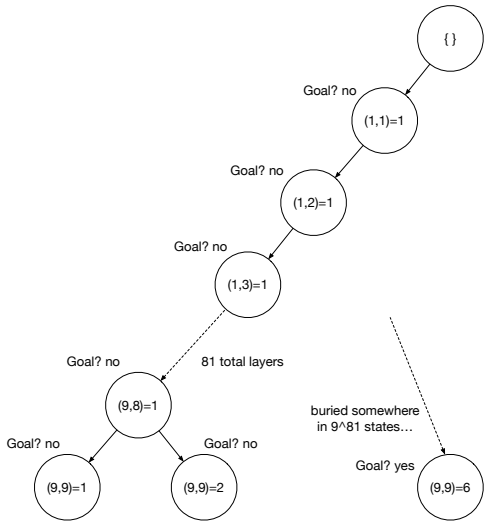
- $X = \{count1, color1, shape1, shading1, count2, color2, \ldots, shading3\}$
- $D = \big\{\{1,2,3\}, \{red, green, purple\}, \{oval, diamond, squiggle\},$
  $\{hollow, striped, filled\}, \{1,2,3\}, \{red, green, purple\}, \ldots,$
  $\{hollow, striped, filled\}\big\}$
- $C = \big\{ < \{color1, color2, color3\}, (color1 = color2 = color3) \vee$
  $(color1 \neq color2 \neq color3) >, \ldots \big\}$

The *goal* of a CSP is always the same: to find a non-sequential state configuration that does not violate any constraints. The CSP process is therefore one of assigning values to variables. A *consistent* assignment is one that violates no constraints. A *complete* assignment is one that assigns a value to all variables. A *solution* is any consistent, complete assignment.

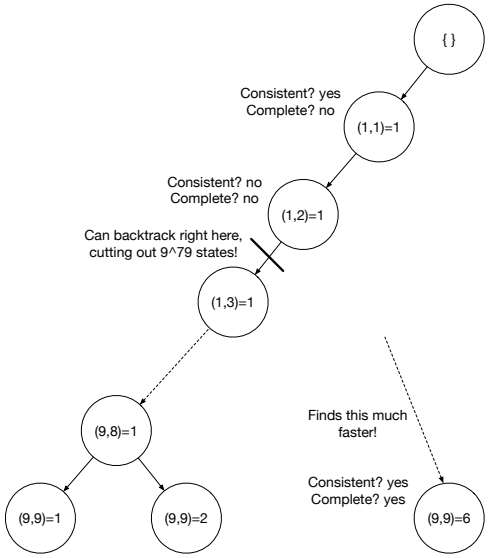
To test your understanding of CSP formulation, try to formally define the common puzzle game Sudoku as a CSP. You may wish to define a pseudocode function *satisfiesConstraints*$(x_1, \ldots, x_n)$ that returns true or false, rather than try to enumerate the full set of Sudoku constraints.

7.3 Why do constraints help the situation?

Recall the search tree we constructed in the previous section of the course. Without CSP, it is always possible that we will potentially examine *all* states:



However with CSP, we could check the constraints after each assignment of a value to a variable, and thus eliminate *many* states from consideration every time we take an action (make an assignment):



The CSP approach can generally help when you:

- Don't have a heuristic or performance metric, or there is no "cost" function.
- Don't exactly know what the goal states *are*, or can't know the solution until you see it.
- Do know what the solutions *don't* look like (i.e. rule-breaking partial solutions).
- Don't care what solution you receive; each is as good as any other.

7.4 Constraint Graphs

Consider a simplified state map of Australia (the Northern Territory is technically not a state, but we include it due to its large size):

A commonly-studied CSP is the "four color" problem, in which one attempts to color a map of the world using only four distinct colors, such that no two adjacent regions share the same color. We will discuss CSP in the same context, but using the simple map of Australia. How many colors will it require?

The problem of coloring Australia such that no two adjacent states are the same color can be expressed formally as:

- $X = \{WA, NT, SA, Q, NSW, V, T\}$
- $D_i = \{R, G, B\} \forall x_i \in X$
- $C = \{ < (WA, NT), \{(R,G), (G,R), (R,B), (B,R), (B,G), (G,B)\} >, \dots \}$

We have only specified one of the constraints required to implement the rule "all adjacent regions must be different colors". To formulate the problem in this way, we will have to enumerate all allowed combinations of values for all pairs of adjacent states. This is probably not how we want to spend our time.

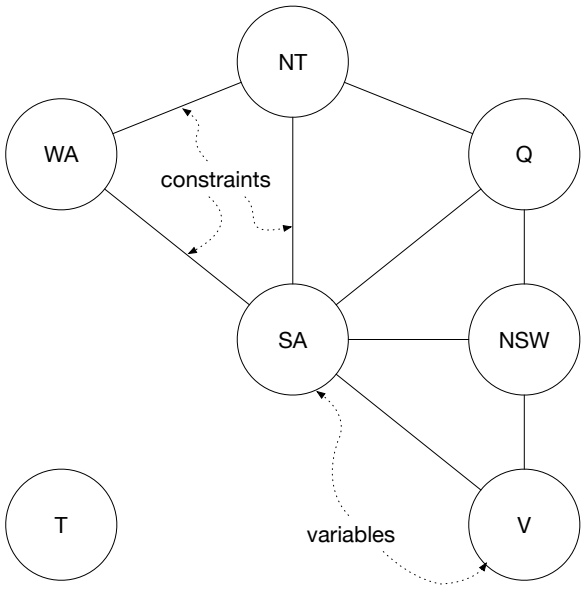
Instead, let us define a simple "language" and some operators to serve as shorthand. We can use WA to represent the *color* of Western Australia, and the operators $=$ and $\neq$ to mean that two state colors must, or must not, be the same. Now we can define C completely, without filling the page:

- $C = \{WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, Q \neq SA,$
  $Q \neq NSW, SA \neq NSW, SA \neq V, NSW \neq V\}$

That's better. Because this problem is small and straightforward, it is not hard to envision a solution ourselves. For example, we can take the very "central" state of South Australia to be our first color (maybe red), then go clockwise around the remaining states assigning alternating colors (green and blue). One such solution is: $\{WA = B, NT = G, Q = B, NSW =$

$G, V = B, SA = R, T = R\}$. Note that the color of Tasmania is never constrained, because it has no neighboring regions.

It is still not so easy to imagine what C is trying to tell us. For a large, complex problem, it would likely be impenetrable! Thus we commonly visualize C as a *constraint graph* instead, with the CSP variables as nodes, and the CSP constraints as edges (typically called *arcs* in this problem domain). Here's an example for the current problem:



Note how all of the constraints are represented. The edge connecting node WA to node NT represents the constraint $WA \neq NT$. Two useful properties are easily apparent from the graph that are not necessarily easy to spot in the constraint list: first, that T has no constraints at all; second, that SA has the most constraints. Finally, note that for this problem, we have *binary constraints* only. That is, constraints that describe allowable combinations of exactly two variables. *Unary* constraints are also possible, which simply reduce the domain for a single variable, as well as *n-ary* or higher order constraints, which may include more complex relationships among three or more variables.

8 Solving CSP with Search

Having decided that our previous search methods do not take advantage of the *structure* of Constraint Satisfaction Problems and therefore perform sub-optimally, we can consider augmenting those search methods to be more appropriate for CSP.

8.1 Backtracking Search

Using our usual formulation of the search problem, let us examine a tweaked version of Depth-First Search that *is* allowed to inspect state internals and therefore ask if a state (even a partial assignment) violates any constraints.

- S: all variables and their assigned values, in any combination including "empty"
- S_0: an empty set of assignments { }
- S_F: our goal is any state with all variables assigned and no constraints violated
- A: our permitted actions are to assign a value to an empty variable without violating any constraints

Thus the current state is the variable assignments made so far (initially none) and we take action by assigning one variable a value. The primary improvement over plain DFS is in the *action* definition. Our search function's *getSuccessors(s)* or equivalent capability is now able to return *only* those successor states (current assignment + one new variable assignment) that do not violate constraints. This means the search can *backtrack immediately* when an assignment is made that results in no further valid assignments being possible, rather than having to assign all remaining variables first, as would happen in plain DFS.

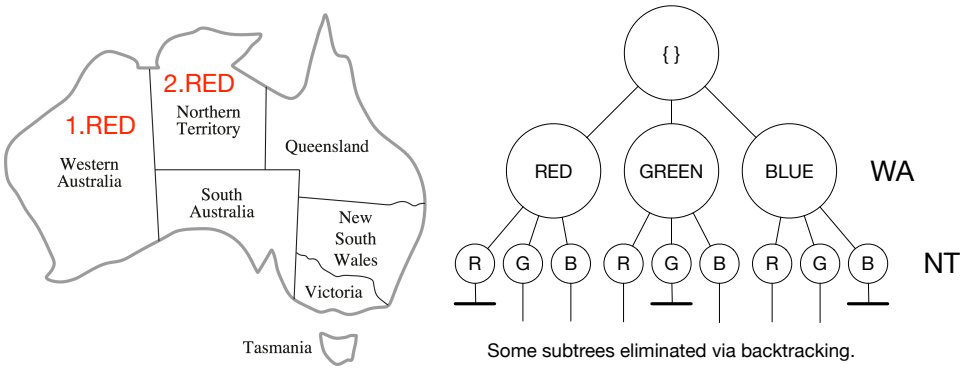
8.2 Variable Ordering

However, our backtracking search is still suboptimal. In a CSP, a requirement for any *solution* is that all variables must be assigned a value (i.e. a *complete* assignment). This means that in terms of our search tree, if there are n variables in the problem, all solutions will be at exactly depth n. Backtracking search, however, still spends quite a lot of time at depths above n, where there are no solutions possible.

In fact, if our domain $d_i \in D$ is the same for all variables $x_i \in X$, as in the Australia problem, there will be $(nd)^n$ nodes generated in the worst case, because the branching factor of our tree is nd (number of variables times choices per variable). That is a lot of nodes!

It seems we could improve this only by reducing the number of variables or the number of value choices for the variables, but either of those would change the problem we are trying to solve. What we *can* do is observe that our tree is trying all *permutations* of assigning values to variables. That is, it generates every possible *ordering* for the same assignment of values to variables. We do not care, however, about the order the assignments are made. It would be enough to try all *combinations* of variable-to-value assignments (i.e. without considering order).

We can do this by arbitrarily picking one variable per tree level (ahead of time) and only assigning that variable at that level. There is now only one ordering of each assignment combination, and this reduces our branching factor to d, the number of choices for the one variable at this level of the tree. In turn, this reduces the number of nodes generated in the worst case to d^n. This is still exponential, but at least not n^n exponential!

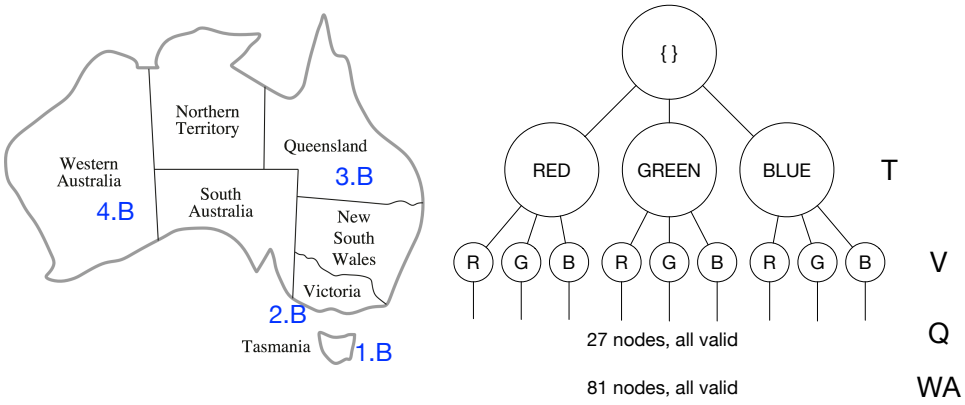


We can think of this as a depth-limited (by number of variables) DFS with one variable per level and variable assignments that propagate down the tree. We backtrack at any point that we have violated a constraint, or when there are no valid successor nodes that can be generated.

Why do we not use Breadth-First Search here? We typically think of it as finding "better" solutions, and certainly no worse than DFS. Remember that in a CSP, *all* solutions are at depth n exactly. BFS expands all nodes at depth 1, then all nodes at depth 2, eventually expanding all nodes at depth $n-1$ before it *ever* considers the first node at depth n. This means BFS must search a huge number of *guaranteed wrong* nodes before it ever explores the first node that *might* be correct. With DFS, it will go directly down and "bounce along" the bottom of the tree, backtracking one step only and then going down again. Thus DFS *could* get lucky and find a solution quickly, where BFS is guaranteed *not* to do so!

8.3 Variable Ordering Issues

We have seen that we can reduce the potential number of nodes in our search tree to d^n by selecting any arbitrary ordering of variables and assigning one to each level of the tree. Consider one such ordering:



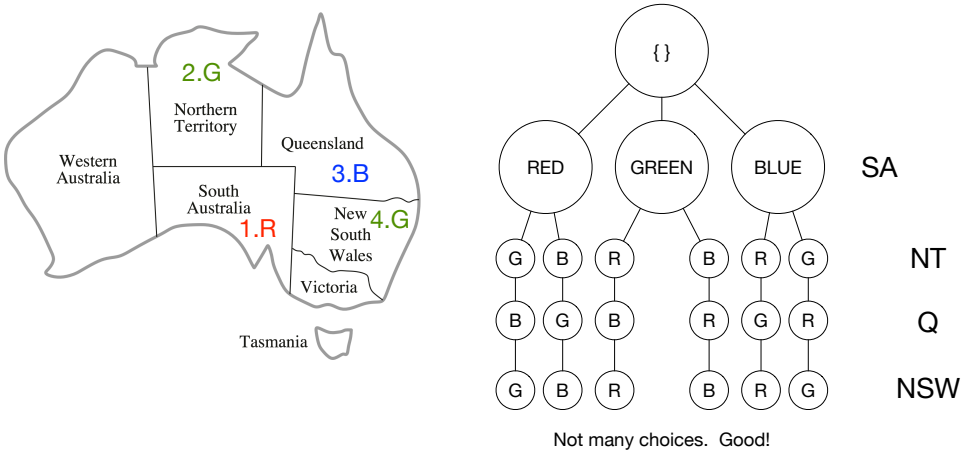
That was unlucky! Since we have chosen to order our variables such that there are no neighboring regions in the first four layers, *all* combinations of values that we assign are valid, and we will not backtrack until deep within the tree, eliminating few nodes from

consideration. We could improve the ordering by deliberately selecting regions that are adjacent to each other, so our constraints are more likely to be violated sooner, but that is a problem-specific approach that requires our (human) knowledge. We would like something the algorithm can do on its own, that generalizes to any CSP.

In general, the best way to assign variables is to select the one with *minimum remaining values*, called the **MRV heuristic**. That is, choose the variable that has the fewest legal options first, to try to force backtracking as early as possible. Failing early on is useful, because this eliminates the greatest number of nodes (in the pruned subtree)!

If there are multiple nodes equally selected by MRV, we may choose the variable with the *most unassigned neighbors* in the constraint graph. This is called the **degree heuristic**. Selecting the node with highest degree also helps us force backtracking, because this node will have the most constraints on its value.

An example of much better variable ordering:



For our first variable selection, all variables have equal domain sizes (remaining possible values), so MRV is a tie. Using the degree heuristic, South Australia is the clear winner, with a degree of five. For the next level, having assigned *some* value to SA (red in the example), all of its neighbors now have only two valid values left (blue and green). Using the degree heuristic again, NT, Q, and NSW all tie for the highest number of *unassigned* neighbors in the constraint graph (two each). We arbitrarily pick NT. For the third variable choice, WA now has only one valid value left (blue), but Q also has only one valid value (blue). Using the degree heuristic to break this tie, we select Q (one unassigned neighbor) over NT (no unassigned neighbors). At the fourth level, WA and NSW both have only one valid value remaining (blue and green, respectively). NSW has one neighbor and WA still has none, so we select NSW by the degree heuristic.

This is much better! We have followed the leftmost branch of the tree shown above, but note how restricted the choices have been. For our first selection, we had three choices. For the second, we had only two choices, and for the third and fourth the choice was predetermined (only one choice). In fact, you can see from the tree that *regardless* of our first two choices, the third and fourth will always have only one choice (the tree is not branching). If you continue the example on your own, you will find that the fifth and sixth selections also have only one choice each, regardless of prior choices.

By ordering the tree in this way, we have reduced the search space *dramatically*. We choose one of three colors for SA, then choose one of two colors for NT, then everything else is determined with no choices, save T, whose value does not actually matter (all choices are always valid).

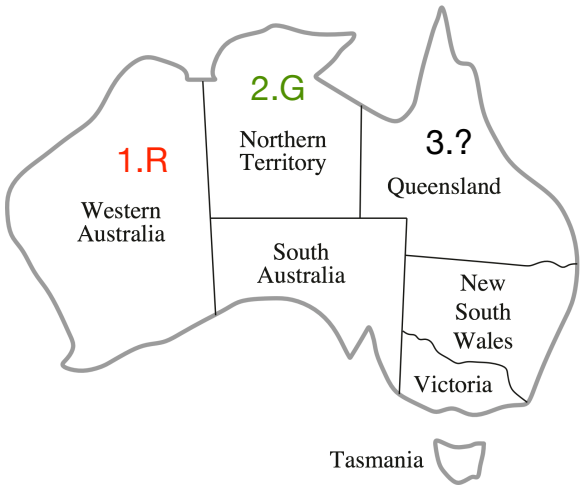
9 CSP Efficiency

There are four major approaches to improve search efficiency for a Constraint Satisfaction Problem: variable exploration order, value assignment order, early failure detection, and clever exploitation of the problem structure. We have covered the first in Section 3 already. Now we can examine the others.

9.1 Value Assignment Order

After having selected a single order in which to assign variables a value, we still face the choice of *which values* to try first. That is, among the successor states of a single search node, which should we expand first?

A common approach, which we will follow, is the *least constraining value* heuristic. This heuristic selects the successor node that leaves the *most* options available for other variables. In the map diagram below, assume we have assigned $WA = R, NT = G$ and selected Q as the next variable to assign. Which value should we explore first?



Clearly we must not select $Q = G$, as it will immediately violate a constraint. What about $Q = B$? This will leave *no* valid values in the domain for SA. So what about $Q = R$? This does leave one valid value for SA (blue), and at least one value for all other variables. Thus we select R as the least constraining value for Q.

This introduces an alternating duality for variable and value selection:

- When selecting a variable to assign, try to *fail early* in order to prune the search tree.

- When selecting a value to expand, try to *fail late* to keep your options open for finding a goal sooner within this subtree.

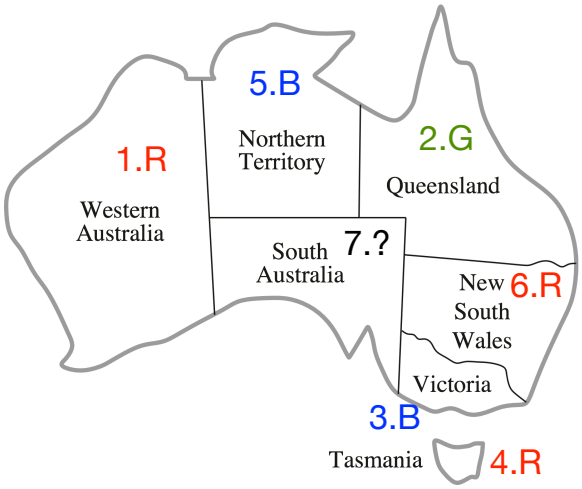
To put it differently, we hope to quickly eliminate subtrees that contain no goals (fail early). Having selected a subtree that may contain a goal, we would like to dive deep and find that goal as soon as possible (fail late). The least constraining value heuristic makes sense because all goals are equally valid and we need only find one of them. If we wish to generate *all* valid solutions, then our variable ordering heuristics are still helpful (prune zero-goal subtrees), but our value ordering heuristic is not.

9.2 Early failure detection by Forward Checking

The heuristics given above are meant to help guide the algorithm, but they are only *estimations* of the best choice to make at any given time. When our search algorithm inevitably enters a subtree that contains no goals, we would like to discover that error as quickly as possible. The earlier we can backtrack, the faster we will find a solution!

The first method we employ to that end is *forward checking*, which consists of three parts:

1. We must track all remaining possible values for *unassigned* variables.
2. We must propagate constraints down the tree to remove rule-breaking value assignments.
3. We must check for empty domains, which indicate a dead-end subtree (no goals).



In the example assignment above, I have again chosen the order of my variables poorly, such that I can reach the very bottom of the tree (last assignment) before discovering that South Australia has no valid values to assign. Only then can my algorithm backtrack. As humans, carefully examining our assignment choices, we can see that no solution is possible once we assign $WA = R, Q = G$, because NT will have to be assigned B, leaving nothing for SA. Could forward checking detect this failure prior to choice seven?

	WA	NT	Q	NSW	V	SA	T
	R, G, B	R, G, B	R, G, B	R, G, B	R, G, B	R, G, B	R, G, B
$WA \leftarrow R$	**Red**	G, B	R, G, B	R, G, B	R, G, B	G, B	R, G, B
$Q \leftarrow G$	**Red**	B	**Green**	R, B	R, G, B	B	R, G, B
$V \leftarrow B$	**Red**	B	**Green**	R	**Blue**	$\{\ \}$	R, G, B

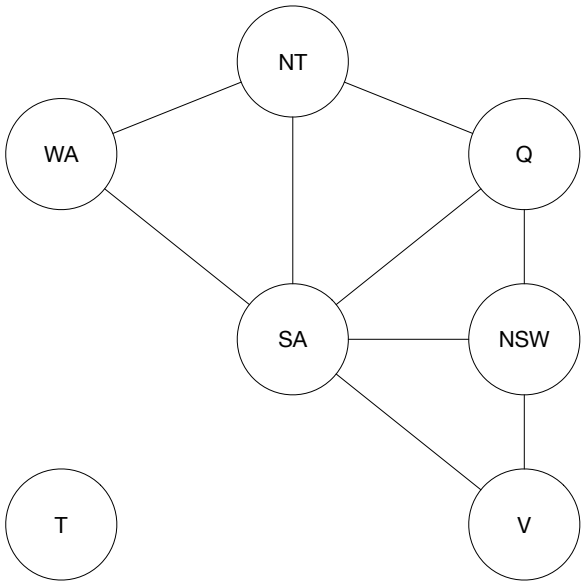
Table 1: Tracking remaining allowed values for each variable. **Bold** indicates a value assignment actually made.

Prior to assigning any values, the domain for all variables is $\{R, G, B\}$. Following the forward checking process listed above, when we assign red to Western Australia, we also eliminate directly conflicting choices in other variable domains by checking the constraints of which WA is part. This causes us to eliminate red from the domains of NT and SA, the neighboring regions to WA. We then assign green to Queensland, eliminating green from the neighboring regions: NT, SA, NSW. When we assign blue to Victoria, we eliminate blue from the neighboring domains V and SA. In step three of the forward checking algorithm, we now find that there is an unassigned, empty domain: SA. We can therefore stop here and backtrack our choices, as there will be *no* goal nodes in this subtree. We might not actually try to *assign* South Australia until the bottom level of the tree, four more choices from now, but we are now able to *detect* the failure after only three choices.

9.3 Early failure detection with MAC

Forward checking allows us to more quickly detect entry into a subtree that contains no goals. Our algorithm is still rather simplistic, however. When we make an assignment, we only eliminate values from other domains that *directly* contradict a constraint *involving* the variable we just assigned. It feels like we are leaving some performance on the table here, because a variable assignment could create a contradiction among *other* variables, not just the one currently being assigned. We would not detect that type of failure until further down the tree!

In order to detect failure even earlier, we can employ the *Maintain Arc Consistency* or **MAC** algorithm. (Remember, arc is just another term for an edge in a graph.) Refer to the constraint graph for this problem:



We say that arc $\overrightarrow{X_i, X_j}$ is *consistent* if for every value in D_i, there is a value in D_j that satisfies all constraints on X_i and X_j. (Note that this is unrelated to the idea of consistency in the A* search heuristic.) Conversely, the arc is inconsistent if some values of X_i do not allow for *any* legal value of X_j to be selected. We further say that a graph G is consistent if *all* arcs within it are consistent.

At the first step of our Australia problem, having made no variable assignments, is arc $\overrightarrow{WA, NT}$ consistent? If Western Australia is assigned red, then Northern Territory can be assigned green or blue without violating any constraints. If Western Australia is green, then NT can become red or blue, and if WA is blue, then NT can become red or green. For all values left in the WA domain, there is a corresponding valid value in the NT domain, so $\overrightarrow{WA, NT}$ is consistent.

Now consider the forward checking example again. (Look back at the table we created for it.) Using the MAC algorithm, we can detect the failure even one step sooner, at row three (choice two)! Notice that at this point, we have domains $NT = \{B\}, SA = \{B\}$, and there is a constraint $NT \neq SA$. There is no valid combination of values for these two variables remaining, so $\overrightarrow{NT, SA}$ cannot be made arc consistent without emptying an unassigned variable's domain, and we can backtrack out of this subtree even sooner.

We can formulate the MAC algorithm in this way:

Algorithm 1 Make Arc Consistent

```
function MAC(X_start)
    queue ← {all arcs X_j X_start}
    while queue not empty do
        (X_i, X_j) ← pop(queue)
        Make X_i arc consistent with X_j
        if size of D_i decreased then
            if ||D_i|| = 0 then
                return false
            end if
            for X_k in neighbors (X_i) − {X_j} do
                queue ← queue + {X_k, X_i}
            end for
        end if
    end while
    return true
end function
```

Note that the word *queue* is used here for historical reasons. The list of arcs to examine need not be a proper Queue, as exploration order is irrelevant.

MAC should be applied to determine if a CSP constraint graph can still be made arc consistent after each variable assignment is made. When MAC returns false, the algorithm may backtrack immediately, because the subtree will contain no solutions. It may detect failures earlier than forward checking, which only "notices" when a domain becomes empty, while MAC locates (non-empty) domains in which the constraints can no longer be satisfied.

A detailed walkthrough of the MAC algorithm on one step of the Australia problem was shown in class. I recommend walking through such an example on your own to ensure you fully understand how and why MAC works. Take a CSP, select an arbitrary initial assignment, and then step through the MAC algorithm to see how it determines if the graph can still be made consistent. Continue making assignments until you reach a point where MAC returns false.

In common practice, you will also see the **AC-3** (Arc Consistency 3) algorithm, so named because it was the third iteration of the algorithm. AC-3 works exactly the same as MAC, except the queue begins with *all* arcs in the graph. It is run *prior* to starting the search algorithm, as a preprocessing step, to eliminate impossible (already inconsistent) values from variable domains up front. This accelerates the search process by shrinking the domains to examine. It will not change the solution(s) that will be found for the CSP.

However, AC-3 is not always useful in preprocessing a constraint graph. Consider the Australia problem. Prior to the first assignment, all domains are $\{R, G, B\}$ and there is a valid combination of values for all constraints, so all arcs are consistent at the start, and AC-3 cannot modify any variable domains.

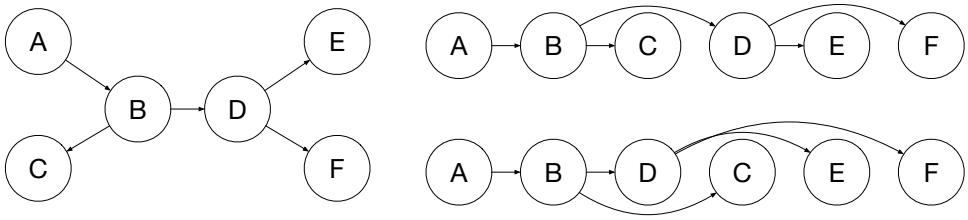
9.4 Exploiting Problem Structure

Have you noticed that in our Australia CSP, the variable T (representing Tasmania) connects to nothing in the constraint graph? This means its value is unconstrained within its domain, and therefore we have two *independent subproblems*. We can therefore solve the coloring problem for T completely separately from the coloring problem for $Australia - \{T\}$ and simply concatenate the results. While this is only slightly useful here, it would be a great help in the four-color Earth problem: independent subproblems with $4^m + 4^n + \ldots$ permutations, where $m, n, \ldots$ are the size of each contiguous landmass in regions, are much better than the original problem of $4^{m+n+\cdots}$ all merged together.

Independent subproblems are useful, but not common enough in natural practice to rely on as a primary performance enhancement. What if we could *make* independent subproblems within an arbitrary constraint graph?

Our best CSP algorithms, while heavily optimized to prune the search tree, fail early, and find a goal quickly, still run in exponential time in the worst case. Recall that a *tree* in graph theory is a graph in which any two nodes are connected by only one path. If our constraint graph *just so happened* to be a tree, would that improve things?

Given a tree data structure, we can always perform a *topological sort* because no tree node has two parents. (A topological sort is a one-dimensional listing of nodes from left to right, such that all child nodes appear after their parent nodes.) Example of tree-shaped CSP and two possible results of a topological sort:



Intuitively, the parent-child directed edges all point to the right in a proper topological sort. So did this help? Yes! During steps when we make the CSP consistent, we now have an order in which we can visit each node *exactly once* without backtracking, because any changes to a parent node's domain can only affect its children, which we are guaranteed not to have visited yet. (That is, with the topological sorting, our arc consistency algorithms are guaranteed only to cause further changes to the right of the current location, never to the left.) This allows the overall search time to become $O(n^2d^2)$, polynomial instead of exponential!

Of course, most constraint graphs are not tree-shaped. Even our "toy" Australia CSP does not have a tree-shaped constraint graph. Could we *make* it a tree? Yes, if we first ignore T as an independent subproblem, and then *cut* node SA from the graph, what is left will be a tree. Can we do that and still get the proper solution?

Let's try this:

1. Remove T from the problem and solve it separately. It has no constraints, so the value can be anything from its domain. We will choose solution: $\{T \leftarrow B\}$.

2. Cut SA from the remaining problem and solve it separately. As its own subproblem, it *currently* has no constraints, so the value can be anything from its domain. We will choose solution: $\{SA \leftarrow R\}$.

3. The SA subproblem *did* however have constraints connecting it to the remaining "main" problem, so we must honor those constraints. How can we modify the main problem to ensure none of these SA constraints are violated? By removing value R from the domains of each variable that had a binary constraint with SA!

4. Thus our main problem now has variables: $\{WA, NT, Q, NSW, V\}$ each with a domain of $\{G, B\}$. It is also a tree, meaning we can now solve this problem in polynomial time, and it is now *independent* of the SA and T subproblems, so no solution we generate will clash with those.

5. Finally, we can concatenate together the variable assignments from the three subproblems. That is our solution to the initial problem!

If this approach reminds you a little of `MergeSort`, you are on the right track. It is essentially a divide and conquer technique to solve CSPs more quickly:

1. Find the largest tree-based subproblem that you can. The remainder is a (probably) non-tree subproblem.

2. Solve the hopefully small non-tree subproblem. Then, remove the solution values from the tree subproblem according to any constraints that span the two subproblems (i.e. include variables from both subproblems).

3. Now solve the tree subproblem.

4. Finally, concatenate the two solutions together (a trivial "merge" step).

If you find at step two that the remaining non-tree subproblem is still too large, you can repeatedly cut out tree subproblems until you finally have a single non-tree subproblem of reasonable size, solve it, and then begin working your way back up the chain of tree subproblems that you split off, before finally merging the solutions.

Acknowledgements and thanks to Professors Mark Riedl and Jim Rehg of the Georgia Tech School of Interactive Computing.