Lecture 6 - Constraint Satisfaction Problems

A CSP is a computational problem that involves finding values for a set of **variables** that satisfies a set of **constraints**. It is defined by:

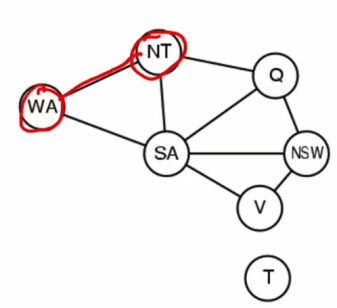
- A finite set of variables X_i
- Values from a domain, D_i
- A finite set of constraints C_j which limit the values that the variables can take.

Can example of this would be coloring a map. This is a well-known CSP. A state in this case would be defined by a set of variables, each of which having a value. The goal state would be a complete assignment of values to variables that do not violate the constraints. This would be "consistent assignment."

What are different types of constraints?

- Unary constraints: Involves a single variable;
- Binary constraints: Two states next to each other cannot have the same color;
- Global constraints: Involves an arbitrary number of variables. Example: alldiff constraint $\longrightarrow$ all the variables involved must have different values.

Constraint Graph: This is a graphical representation used to model CSPs.



In this graph, we can see that edges represent the existence of a constraint.

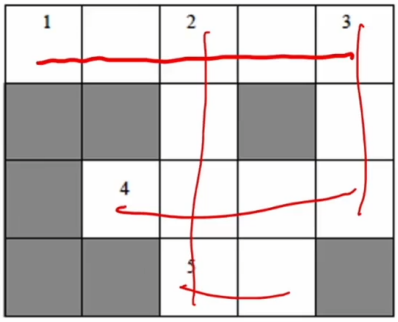
In real world applications, this could be an assignment problem (such as assigning professors to teach classes). Each course must be assigned to a professor, but a professor can teach multiple courses. So on and so forth. It could also be a timetabling problem, transportation scheduling, factory scheduling, and continuous domains. This is an application of **Linear Programming**.

CSPs can be expressed as a standard search problem. We already know what a state in CSPs are.

- Initial state: The empty assignment
- Result function: Assign a value to an unassigned variable that does not conflict with current assignment $\longrightarrow$ fail if no legal assignments
- Goal test: The current assignment is complete.

In each level of the search tree, we would be assigning one variable.

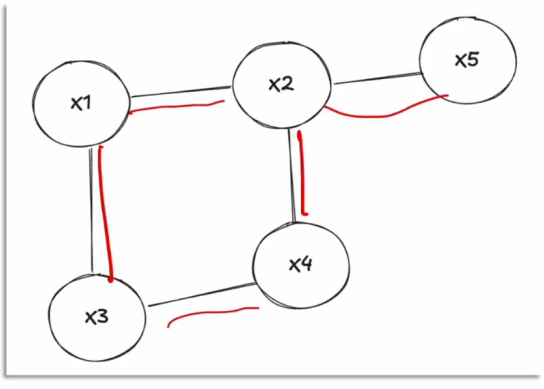
Notice that every solution appears at a depth n with n variables assigned. Therefore, this is basically the same thing as DFS. We call this **DFS with Backtracking Search**. Choose a value for one variable and backtrack when a variable has no legal assignments left.



Variables: X1, X2, X3, X4, X5
The position of the variables in the puzzle is labeled 1,2,3,4,5

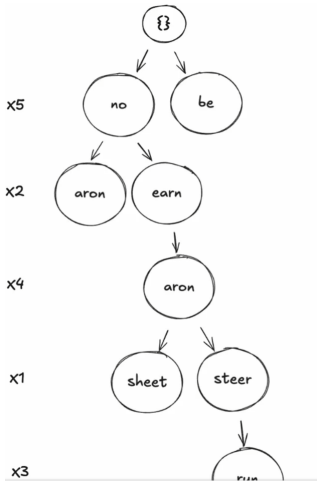
Domain values:
X1 can be assigned {hoses, laser, sheet, snail, steer}
X2 can be assigned {hike, aron, keet, earn, same}
X3 can be assigned {run, sun, let}
X4 can be assigned {hike, aron, keet}
X5 can be assigned {no, be}

The constraint graph for this can be represented as follows:



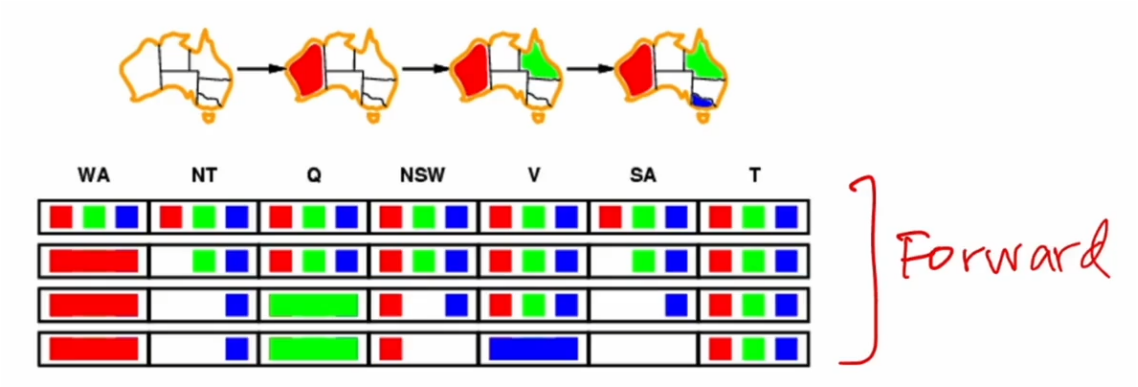
We can see that the word X_1 can affect both X_2 and X_3, so we add an edge between them. So on and so forth. What can we do such that we don't have to backtrack too much?

Minimum Remaining Values (MRV) heuristic: This is a general heuristic (not problem specific). In the above case, the MRV is for X_5, which can only take two variables. Let's start with that instead. This is relatively straight forward.



As you can see, in this case, we had to backtrack a lot less. So this is better. In fact, we only had to backtrack twice. In the alphabetical order, we had to backtrack 4 times. The process of keeping

track of the remaining legal values is called **Forward Checking**. The idea is to terminate the search when any variable has no legal values left.

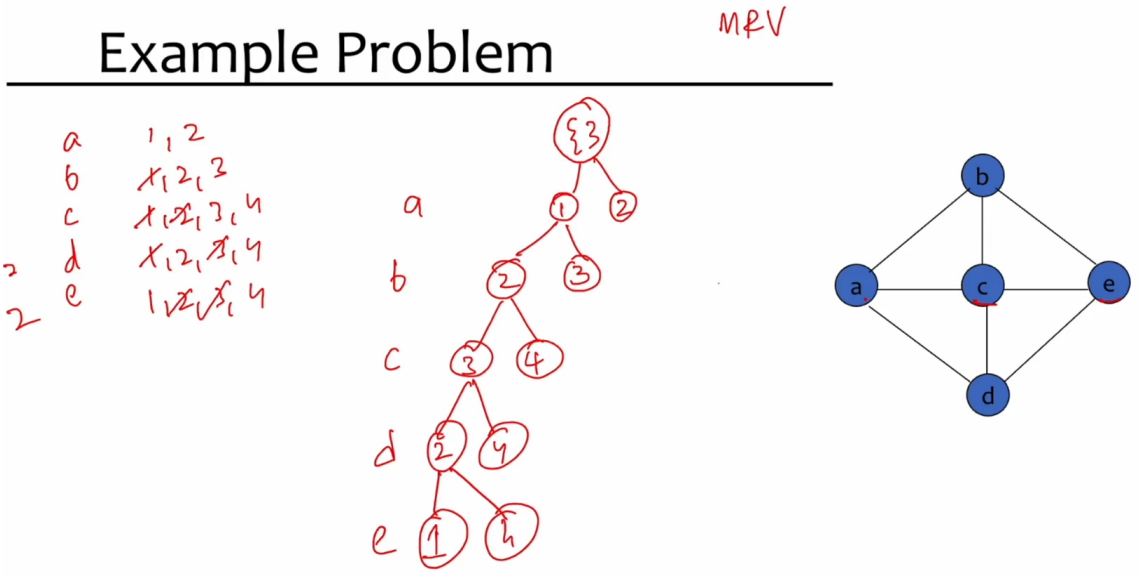


This is usually done in conjunction with MRV. However, it can be done independently as well. When no legal values are left for a variable, we backtrack.

Let's go back to the n-queens problem, and apply forward checking to it. Given a random initial stte, we can solve the n-queens problem in almost constant time for some arbitrary n.

Degree Heuristic: Choose the variable with the most constraints on remaining variables (Most constraining variable).

In the example he showed in class, we applied MRV first. If there is a tie, that's when we apply the degree heuristic.



The variables a and b can only be assigned $\{1, 2\}$ and $\{1, 2, 3\}$, respectively. Nodes that are connected by an edge cannot have the same value assigned.

Arc Consistency: This is applied in CSPs involving binary constraints. We would list all the binary constraints between variables X_i and X_j: $X_i \longrightarrow X_j$. The objective is to go through each arc and make them consistent. The arc $X_i \longrightarrow X_j$ is consistent if for every value of X_i, there is some allowed value for X_j. If it is not consistent, then we remove the value from X_i that makes the arc inconsistent. We also need to add new arcs based on assigned values.

If X_i loses a value, then arcs $X_k \longrightarrow X_i$ also need to be checked for consistency again, where X_k are neighbours of X_i in the binary constraint graph. The arc consistency algorithm is called AC-3. The algorithm is given below:

```
Algorithm AC-3(CSP):
    Initialize queue Q with all arcs (Xi, Xj) where Xi and Xj neighbours
    while Q is not empty:
        remove arc (Xi, Xj) from Q
        if REVISE(CSP, Xi, Xj) is True:
            If domain(Xi) is empty:
                return False ## The CSP is inconsistent
            for each neighbour Xk of Xi (excluding Xj):
                Add(Xk, Xi) to Q
    return True ## The CSP is arc consistent

Procedure REVISE(CSP, Xi, Xj):
    set Revised = False
    for each value x in domain Xi:
        if no value y in domain Xj satisfies the constraint between Xi and Xj:
            remove x from domain Xi
            set Revised = True
    return Revised
```

The idea is sort of similar to forward checking. If, after applying AC-3, the domain of any variable is empty, then CSP is arc inconsistent and has no solution. If all variables have a single value in their domain, then CSP is arc consistent and has a unique solution. Similarly, if there are multiple values, then the CSP is arc consistent but can have multiple solutions.

After AC-3, **backtracking search** is used to find the solution.