Study Notes for AI (CS 440/520)
Lecture 6: Constraint Satisfaction Problems

Corresponding Book Chapters: 6.1-6.2-6.3-6.4
Note: These notes provide only a short summary and some highlights of the material covered in the corresponding lecture based on notes collected from students. Make sure you check the corresponding chapters. Please report any mistakes in or any other issues with the notes to the instructor.

1 Constraint Satisfaction Problem

The search problems considered so far correspond to a wide variety of problems that can have many different formulations. Constraint Satisfaction Problems are a subset of all search problems which conform to a standard, structured and very simple representation. The advantage of `CSP`s is that they have very general purpose heuristics which can be used for all `CSP` problems.

`CSP`s are defined by two sets of parameters: **variables**(X) and **constraints**(C). Each variable has a nonempty domain of possible values. If an assignment has no variables violating any of the constraints then the assignment is considered **consistent**. If every variable is used in an assignment then the assignment is considered **complete**. A solution to a `CSP` must be both a consistent and complete assignment.

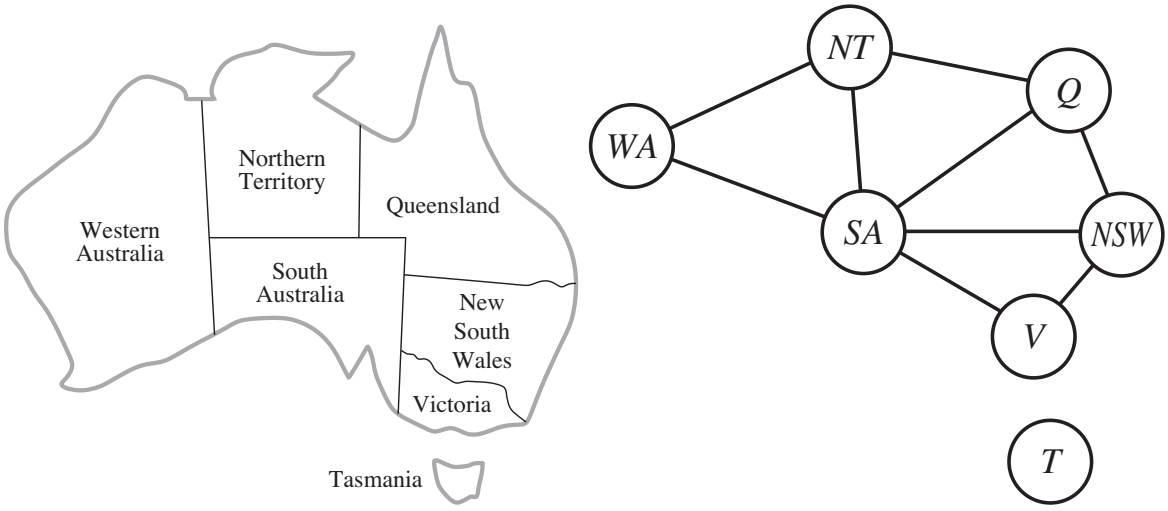

Figure 1: An example `CSP` challenge: Coloring the map of Australia.

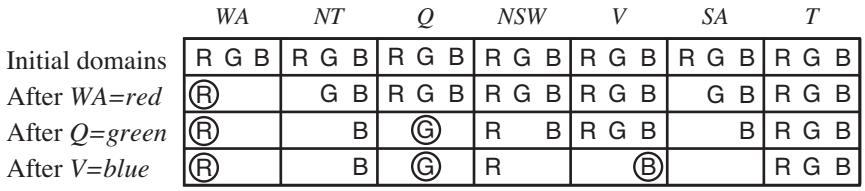

Figure 2: Considering assignments of values to variables for the map coloring problem.

Figure 1 provides an example of a CSP. The goal of this problem is to color each section of Australia so that no neighboring regions have the same color. So, for this problem our variables are the states of Australia. The domain for each variable is the set of possible colors: red, green, or blue. The constraints show what states can't have the same color because the two states touch. For example, Western Australia can't have the same color as the Northern Territory and South Australia.

An easy way to understand the relationships between the variables and constraints is to create a graph as shown in figure 1. Each node represents a variable and each edge represents a constraint. Notice how the WA node has constraints with NT and SA just as discussed. In Figure 2 one can see that when WA is assigned the color red the constraints on WA's neighbors cause the color red to be removed from each of their domains.

So how do we go about solving CSPs. Since they are a subset of search problems we can employ search algorithms. Since there is no competing agent, an adversarial search won't help much here. However, informed, uninformed and local search techniques are applicable. CSPs have the property that they can be formulated in two different ways, either by following an incremental formulation or a complete one. Each formulation lends itself to a solution with a different search algorithm.

1.1 Incremental Formulation

Initial state: The empty assignment in which all variables are unassigned
Successor function: A value is assigned to a variable, assuming it doesn't violate any constraints. The approach retains consistent or valid assignments throughout the problem.
Goal test: Is the assignment complete?
Path cost: 1 for every step.

An example of the incremental formulation is the 8-queens problem, when we place each queen one at a time to the left most column while making sure it's placement doesn't cause any existing queen to be attacked. The incremental formulation lends itself to a solution by classical search (uninformed and informed).

1.2 Complete-State Formulation

Initial state: A random complete state which may or may not satisfy all constraints.
Successor function: Change the value of a variable. So in this case the assignment is complete throughout the entire search process.
Goal test: Is the assignment consistent?
Path cost: Does not apply.

The complete state formulation lends itself to a solution with a local search approach. For the problem of 8-queens, the complete state formulation specifies that all the queens are placed on the board. A local search algorithm can then locally move queens until a configuration with no conflicts is found.

1.3 Domains and examples of CSPs - Complexity

There are different types of CSPs with different domains. There are **discrete and finite domains** such as map coloring problems and the 8-queens puzzle, **boolean CSPs** such as satisfiability problems, **discrete and infinite domains** such as scheduling over the set of integers (e.g., all the days after today), and **continuous domains** such as scheduling over continuous time or linear programming problems.

Additional examples are: crossword puzzles, Sudoku, cryptography problems, and many classical NP-complete problems such as: clique problems, vertex-cover, traveling salesman, subset-sum, and hamiltonian-cycle. The common feature of all these problems is that they are NP-complete. Consequently, we do not expect that we will find an algorithm to solve these problems in polynomial time in the worst case. Nevertheless, by properly designing the algorithms we can still improve the performance of the search procedure in many practical cases. The following discussion will focus on techniques for designing practically efficient algorithms for `CSP`s.

2 Classical Search for `CSP`s: Backtracking Search

An important characteristic of `CSP`s is that the order with which variables are assigned doesn't matter. The final assignment is all that matters. Take for example the Australia coloring problem, the order with which the colors are entered has no meaning. Consequently, for a `CSP` where we have d values in the domain and n variables, the number of leaf nodes will be d^n. This means that there are d^n complete assignments (exponential complexity in the number of variables as expected, since `CSP`s are NP-complete problems).

The fact that the depth of the tree is limited implies that the most appropriate technique from classical search is the depth-first algorithm (or more appropriately its variation: depth-limited). This is due to the fact that depth-first search has the best space complexity and once the depth is limited, it is also complete. When depth-first search is applied on `CSP`s, it is usually referred to as backtracking search. That is because the search algorithm does not always have to reach the leaf nodes of the tree. As soon as it discovers that it has reached an inconsistent assignment, the algorithms stop searching down this branch of the tree and backtracks to the parent node from where it can evaluate alternative assignments.

Figure 3 gives an example of a search tree for the map coloring problem of Australia. At the first level WA is assigned red, green and blue. Each creates a branch where at the next level NT is assigned. If you look at the leaf node with WA = Red, NT = Green, Q = Blue, if the next variable is SA then this branch would find that none of the three colors can be assigned to SA so there is no valid assignment for this branch. If one looks back at figure 2, one can see that SA's domain has no valid values after V is assigned. So in both cases the search will simply backtrack to the parent and try the other branches of the parent node. If the search reaches a depth n for n variables then we know we have a valid complete assignment and therefore a solution.

2.1 Designing Heuristics for Backtracking Search

In order to improve the performance of the basic backtracking algorithm we can employ heuristics. An important advantage of the common structure of constraint satisfaction problems is that we can design general-purpose heuristics that are applicable to all of these problems. Below we present a set of questions that arise in the operation of the basic backtracking algorithm. Heuristics can be used to answer these questions in an attempt to improve the performance of the search procedure:

1. Which variable should be assigned next? Although the order of assigning variables does not effect the existence of a solution, it does effect the performance of the search algorithm. Similarly, what value should be first assigned to a variable?
2. What are the implications of assigning one variable for the remaining unassigned variables? Notice in Figure 2 how assigning one variable can limit the domains of multiple variables based on the constraints.
3. If a path fails can we avoid the same failure in subsequent paths? This is an issue because the same reason for causing a failure in one branch of the tree can also appear in multiple branches.

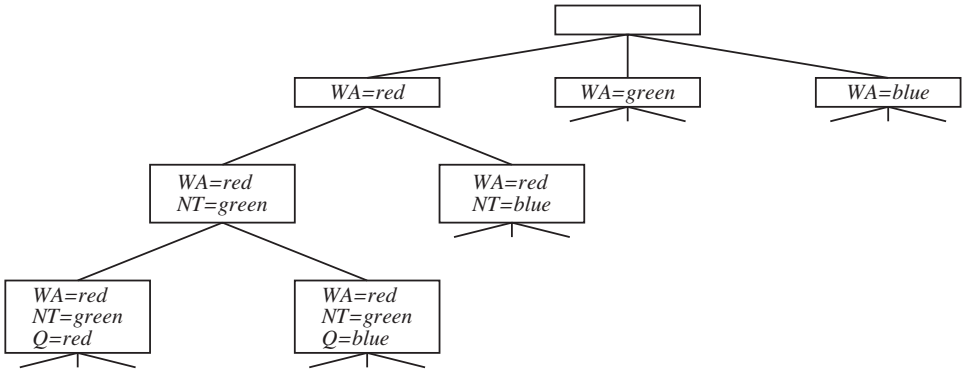

Figure 3: Backtracking Search tree for the Color Australia `CSP`.

2.2 Variable and Value Ordering

Consider Figures 4 and 5. After assigning "WA = red" (Western Australia) and "NT = green" (Northern Territory), the obvious choice for the next variable to assign is Southern Australia. This is the most constrained variable, there is only one value that is still valid: "SA = blue". The above principle leads to a heuristic for backtracking called: **Minimum Remaining Values** (`MRV`) heuristic. It specifies an order for choosing which variable to assign next. Variables that are most constrained have higher priority, since this will minimize the amount of branching out and searching necessary to find a solution.

The `MRV` heuristic, however, does not specify which variable to choose first. One way to address this issue is to use the **degree heuristic**: select the variable that is involved in the highest number of constraints. This heuristic aims to quickly constraint the highest number of variables so as to again minimize the amount of searching necessary. The degree heuristic can also be used as a tie-breaker when the MRV heuristic is not sufficient to distinguish between two variables.

The above two heuristics define a variable order. The backtracking algorithm must also select the order of values to test for a variable. An appropriate heuristic, in this case, is the **Least Constraining Value** heuristic: it prefers the value that rules out the fewest choices for the neighboring variables. For example, consider Figure 5. After assigning "WA = red" and "NT = green" and if selecting values for variable "Q" then the value "red" is preferable. This value of "Q" does allow for "SA" to be blue. Assigning "Q = blue" would lead to unnecessary and failed attempts to find a consistent assignment for the remaining variables.

2.3 Propagating Information Through Constraints: Forward Checking

The second series of heuristics attempts to use the constraints so as to quickly invalidate values for the remaining variables that will lead to conflicts. One such heuristic is the **Forward Checking** approach. The idea is when variable X is assigned to go to variable Y, where Y is a neighbor of X in the constraint graph and delete all the possible values for Y that are inconsistent with X's choice. Figure 6 provides an example. When "WA" is assigned the value "R" then the domains of all the neighboring variables are updated (the value "R" is removed from their domain). After assigning "WA=red", "Q=green" and "V=blue" forward checking directly reveals that this partial assignment cannot lead to a solution (no valid values remain for variable "SA").

The above example, however, reveals that we should actually detect the inconsistency one step earlier. Notice that after assigning "WA=red" and "Q=green" that the only remaining value for variables "NT" and "SA" is blue. If we check the constraint graph in Figure 4, these two variables are neighbors and consequently the assignment "WA=red" and "Q=green" already leads to inconsistency. An approach that detects early on inconsistencies of this

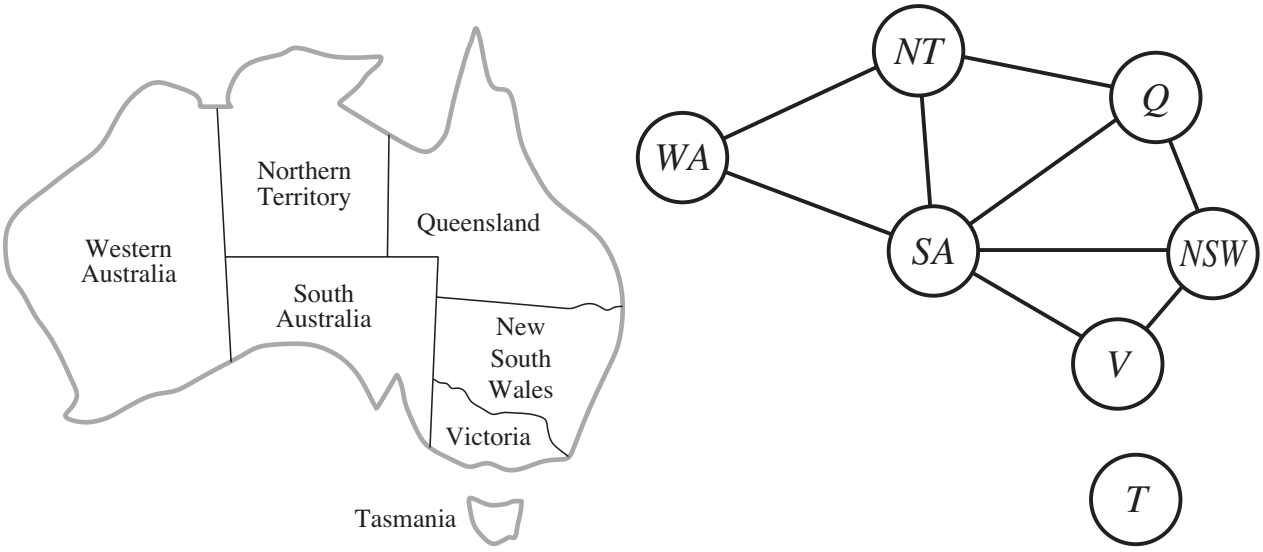

Figure 4: An example of a CSP: coloring the various regions of Australia so that no two neighboring regions have the same color. CSPs can be abstracted by graphs, like the one to the right for the Australia coloring problem, where nodes correspond to variables and edges correspond to constraints.

type is called **arc consistency**. The name arc corresponds to an edge on the constraint graph. Arc consistency specifies that:

Given the domain of neighboring variables X and Y, the arc between the two is consistent if $\forall$ values x of X, $\exists$ some value y of Y that is consistent with x.

2.4 Intelligent Backtracking

Assume the partial assignment "Q = R, NSW = G, V = B, T = R". Notice that changing the value of variable "T" (Tasmania in the map of Australia) does not affect at all the consistency properties of the assignment. When we detect the inconsistency, then we have to backtrack to a variable further back that "T" along the search tree. The **backjumping** heuristic attempts to detect the variable that caused the inconsistency and the one variable that the backtracking algorithm should backtrack to.

To achieve this objective the heuristic approach maintains a conflict set every time that an inconsistency is detected. For example, given the above assignment, and when the algorithm attempts to assign a value for variable "SA" a failure will occur because the domain of the variable is now empty. Then the algorithm will create the conflict set "Q,NSW,V", because these are the variables related to the detected inconsistency at "SA". The algorithm will backtrack to one of the variables in the conflict set and not to the "T" variable.

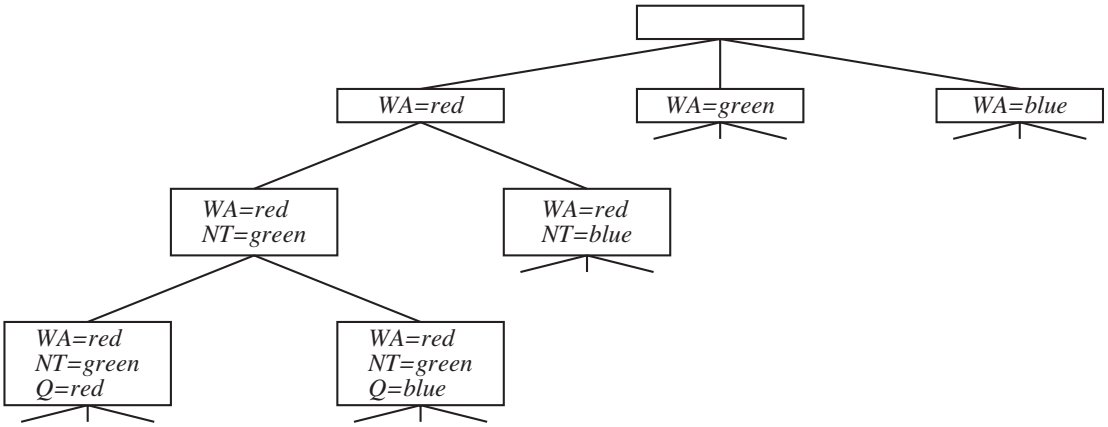

Figure 5: A partial search tree for the Australia coloring problem.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Figure 6: An example of the Forward Checking heuristic.