

Constraint satisfaction problems (CSP)

In the previous set of notes, we discussed the application of local search algorithms to problems such TSP, 8-queens and SAT. In those problems, we were interested in finding a solution, but we didn't care how we get to a good state. Greedy hill-climbing search is an effective algorithm for these problems, but for a subset of them, we will be able to devise more efficient search algorithms, ones that rely on our making explicit more of the structure in the problem.

Recall that by incorporating a heuristic function, we converted uniform cost search into a much better algorithm, A^* . The heuristic function was a way to convey information about the problem to the algorithm, so that A^* could reason about what nodes it could skip expanding, and still guarantee finding an optimal solution. That reasoning process was an example of **inference**.

In today's lecture, we'll describe a formalism that makes more of the structure of certain search problems explicit to the search algorithm, so that it can reason more "deeply" into the problem and perform deeper inference regarding the space of solutions. Specifically, we will describe **constraint satisfaction problems (CSP)**, where the goal is to find assignments to a set of variables, so that the assignments satisfy a certain set of constraints. The CSP problem specification will allow us to do inference to prune off huge portions of the search space which would have been explored by simpler algorithms such as blind search. In some cases, it will also help us to choose the most promising areas of the state space to explore first.

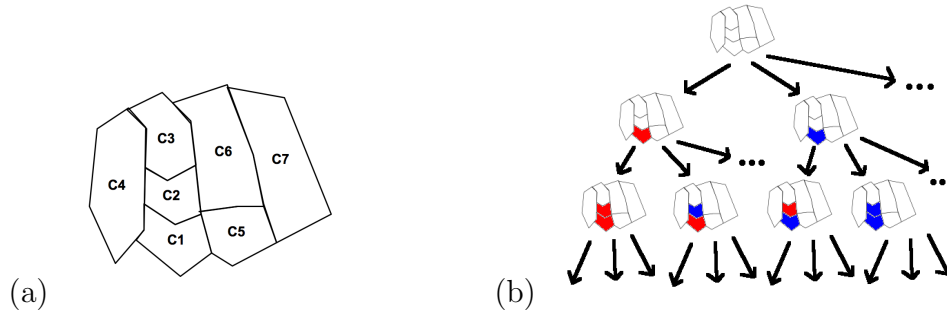


Figure 1: (a) A map which we want to color using only three colors, with no two adjacent countries sharing the same color. (b) A portion of the search tree if we attempt to solve the problem using naive depth-first search. Note that the search continues down the left hand side even though the color constraint is already violated after just two steps.

1 Problem Definition

First, a motivating example. Consider the problem of coloring the map in Figure 1 with no more than three colors, such that no two adjacent countries share the same color. How might we solve this problem *naively* using depth-first search? We define a variable C_i for each region, which can take values in the set $\{\text{red, green, blue}\}$. The states will be partial assignments of values (colors) to the variables. The operator takes the next uncolored region, and colors it. The goal test takes a full assignment of colors to all of the countries and checks if all pairs of adjacent countries have different colors.

What happens when we apply naive depth first search (DFS) using this formalism? Part of the search tree is shown in Figure 1. The algorithm first proceeds down the left-hand side of the search tree, assuming the first operator is to color the next country red. Therefore, it immediately colors countries 1 and 2 red. This seems silly to us, because we know there can't possibly be a satisfying solution where both of these countries are red. But remember that for general search, the goal test is simply a black box which takes a state and tells us if it is a goal. It can't tell us whether or not a state might lead to a goal later on, and so after having colored countries 1 and 2 red, it will waste a large amount of time exploring all possible combinations of colors for countries 3, 4, \dots , 7, not realizing that there's no combination of colors for them that could lead to a solution in this leftmost portion of the tree where countries 1 and 2 have already been colored red.

In order to make this sort of information available to the algorithm, we

must make the constraints explicit, or **declarative**. This will allow us to construct algorithms which can prune large portions of the search space as soon as inconsistencies are introduced.

A **constraint satisfaction problem** (CSP) comprises the following:

- A set $V = \{v_1, \dots, v_n\}$ of variables. In our example, V is the set of countries.
- A finite domain $D = \{d_1, \dots, d_m\}$ of possible values for each variable. (When the domain is different for different variables, we will use D_i to denote the domain of variable V_i .) In our example, D is the set $\{\text{red, green, blue}\}$.
- A set C of constraints. Each constraint defines some restriction on the possible values a subset of the variables may jointly take. Each constraint is defined by specifying a subset $V' \subseteq V$ of variables, and the set of legal values (tuples) for the variables in V' . For example, the constraint that neighboring countries have different colors can be represented as:

$$\{(\text{red, green}), (\text{red, blue}), (\text{green, red}), \\ (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}$$

Our goal is to find a set of domain values to assign to all the variables V_i , so that all the constraints in C are satisfied simultaneously. Let us look now at some additional examples of CSPs.

1.1 8 queens problem

Recall that, in chess, a queen can attack any piece which lies on the same row, column, or diagonal. We want to place 8 queens on a chess board such that no one queen can attack any other queen. An example solution to the 8 queens problem is shown in Figure 2.

There are many ways to formulate the problem of finding a solution to the 8 queens problem as a CSP. Here is a fairly good one, which uses the insight that there has to be exactly one queen per column:

- **Variables:** V_i for $i = 1, \dots, 8$, representing the row number of the queen in the i^{th} column.
- **Domain:** $\{1, \dots, 8\}$

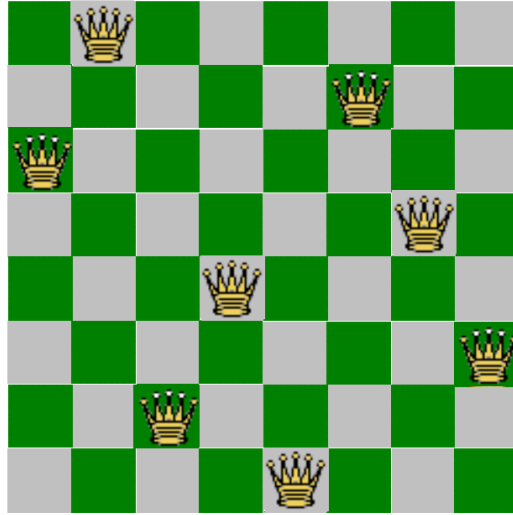


Figure 2: An example of a solution to the 8 queens puzzle.

- **Constraints:** For each V_i and V_j , the constraint that they cannot attack each other. For each V_i and V_j , (r_i, r_j) is a legal pair of values if:

$$r_i \neq r_j \quad \text{for } i \neq j$$

$$|r_i - r_j| \neq |i - j| \quad \text{for } i \neq j$$

The first line above ensures that the queens in columns i and j aren't in the same row ($r_i \neq r_j$). The second line above ensures that the queens in columns i and j aren't along the same diagonal. (This is perhaps an unfortunately cryptic way of writing it, but it is basically the constraint that the difference in x -coordinate $|r_i - r_j|$ isn't the same as the difference in y -coordinate $|i - j|$ —in other words, that they aren't on the same diagonal.)

1.2 Scheduling Ph.D. visits

Here's our third example. It's visiting weekend for Stanford Ph.D. admits, and there are n admitted students who want to meet with all n professors. All these meetings must happen in the n available time slots; further, each person can be in only one meeting at a time, so if a student and a professor are meeting at some time, then neither one of them can have any other meeting at the same time. We can represent this as a CSP:

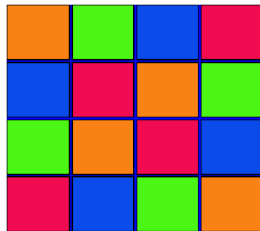


Figure 3: An example of a completed Latin square.

- **Variables:** V_{ij} for each student i and professor j , giving the time slot in which they meet.
- **Domain:** $\{1, \dots, n\}$
- **Constraints:** For all $i, j, i' \neq i$, and $j' \neq j$, we have $V_{ij} \neq V_{i'j}$ and $V_{ij} \neq V_{ij'}$. In other words, no student or professor can be in more than one meeting at a time.

We can also represent this problem pictorially. Imagine a matrix where the rows and columns represent students and professors respectively, and the value in each cell represents the meeting time. If we use a different color to represent each of the n possible timeslots, the problem is to find a way to color the cells such that each color appears only once in each row and column. This is also known as the **Latin square** problem, and an example solution is shown in Figure 3. Often, we are interested in solving a harder problem: given an assignment of colors to a subset of the cells, is it possible to find a coloring of the remaining cells which gives a Latin square?

It turns out that, more generally, most scheduling problems can be formulated as CSPs.¹

2 Inference algorithms

Now that we've defined the constraint satisfaction problem, how do we actually solve it? Actually, we are not going to present individual monolithic

¹For example, in case our Ph.D. visit weekend scenario seems contrived, note that n students, each of whom wants to meet a different subset of m professors, in a total of t timeslots (and with each professor being available for a different subset of the timeslots), is also a CSP, and this is what actually happens at Stanford's Ph.D. visit weekend. This more general formulation doesn't lead to the nice Latin square solutions, though.

algorithms for solving CSPs. Rather, we will present three **inference procedures** of increasing sophistication, and then we will discuss a variety of heuristics which further help in the search process. Our inference procedures will allow us to rule out large sections of the search tree by proving it could not possibly contain a solution. These inference procedures and heuristics will be embedded in the overall search algorithm, and can be mixed and matched in various ways depending on the task at hand.

Recall our description previously of a depth-first search process over a state space of partial assignments to variables. Our operators in the search space will take one unassigned variable, and assign a value to it. This depth-first search algorithm will serve as our overall template. Now, however, we will check at each node which assignments could possibly lead to solutions. This will allow us to select more promising assignments for each variable, or to backtrack earlier when it is clear no assignment will work.

2.1 Consistency checking

When we discussed the map coloring example, we saw that naive depth-first search wasted time considering an entire section of the search tree even though it had already assigned two neighboring countries the same color. The most basic inference algorithm, **consistency checking**, simply rules out this case. Specifically, we say that a partial assignment is **consistent** if it does not already violate any of the constraints in C .² In consistency checking, we simply don't assign any value v_i to V_i which leads to an inconsistent partial assignment. For instance, in map coloring, we don't even consider coloring C_2 red if we have already colored C_1 red. As shown in Figure 4, this often allows us to backtrack early.

2.2 Forward checking

A somewhat more powerful inference algorithm is called **forward checking**. We don't have to wait until we assign a value v_j to variable V_j to check which assignments are consistent. Rather, each time we instantiate a variable V_i , we can propagate all of its constraints forward. For each variable V_j which has not been instantiated, we remove from its domain all values which conflict with V_i . An example is shown in Figure 5. The key advantage of forward

²Technically, this definition only works for binary constraints (constraints between two variables). For N -ary constraints with $N > 2$, an assignment to a subset of the variables in a constraint C is consistent if there exists some tuple in C which does not contradict that assignment.

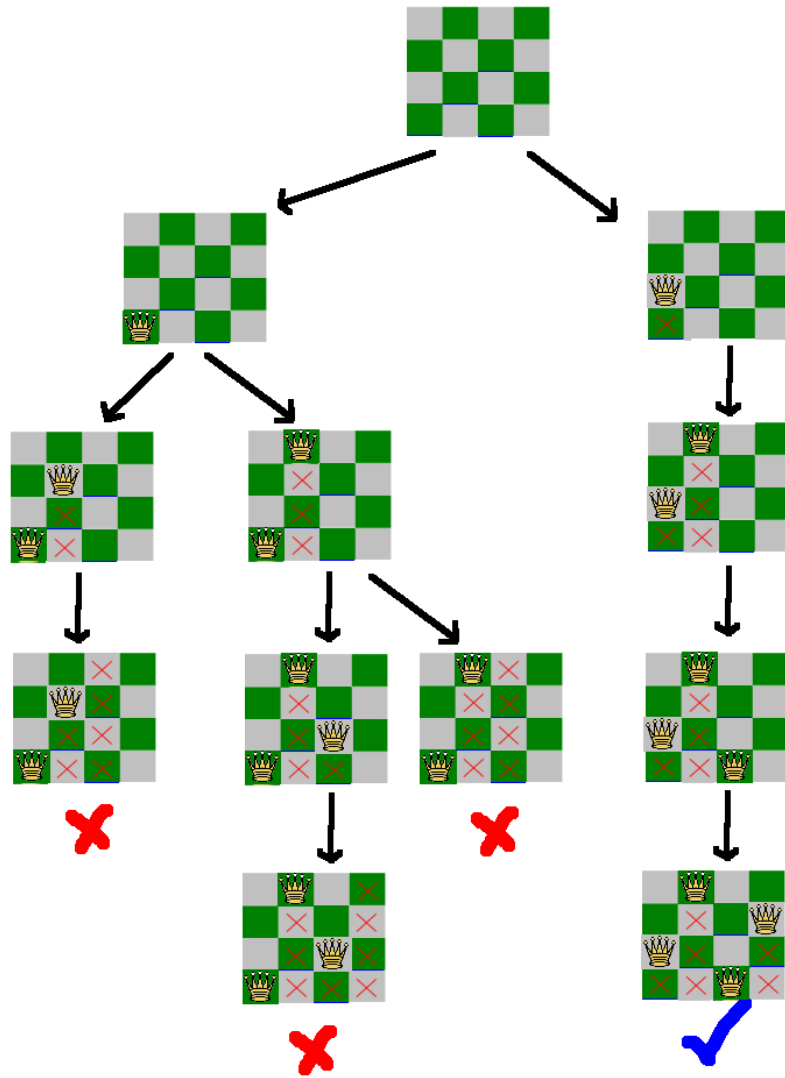


Figure 4: The complete search tree for solving the 4 queens problem using consistency checking.

checking over consistency checking is that it allows us to detect early on when a given variable has no possible consistent assignments, and therefore backtrack. A case where this makes a difference is demonstrated in Figure 6. When we only use consistency checking, we have to wait until the offending variable V_6 is actually instantiated.

2.3 Arc consistency and constraint propagation

The most powerful inference algorithm we will discuss here, **constraint propagation**, actually considers pairs of uninstantiated variables and rules out inconsistent pairings. The goal is to eliminate values from domains until all of the domains become arc consistent, a term we define shortly. This often rules out significantly more possibilities than forward checking, but at a much larger computational cost. For simplicity, let's restrict our attention to binary constraints (those involving only two variables).

For intuition, consider a pair of variables V_i and V_j , with domains D_i and D_j . (These are not the original domains from the problem definition, but rather the domains with some values already removed by constraint propagation.) Suppose we try all of the assignments $v_j \in D_j$ for V_j , and all of them turn out to be inconsistent with some specific assignment v_i to V_i . We can then rule out assigning v_i to V_i , since it is inconsistent with all remaining values for V_j .

Definition. Let c be a constraint over two variables V_i and V_j . A value v_i for V_i is **c -consistent** with D_j if there is some possible assignment $v_j \in D_j$ to V_j which does not violate c (more formally, $(v_i, v_j) \in c$). A domain D_i is **c -consistent** with a domain D_j if all values $v_i \in D_i$ are c -consistent with D_j . If D_i is c -consistent with D_j , we also say that D_i is **arc consistent** with D_j .³

The basic step in our algorithm is **constraint propagation**. Given a constraint c over variables V_i and V_j , we **propagate** c from j to i by eliminating from D_i all values v_i that are not c -consistent with D_j . After this step, D_i will be c -consistent with D_j .

The goal of arc consistency checking is to ensure that all pairs of domains are arc-consistent. To ensure arc consistency, on each pass over the variables, we perform constraint propagation on all pairs V_i, V_j of variables for $i \neq j$. Note that our definitions of c -consistency and constraint propagation

³For general CSPs, two domains D_i and D_j can be included in multiple constraints, and in this case, c -consistency and arc consistency are different. Here, however, we restrict ourselves to the case of binary constraints, so we treat the two as the same.

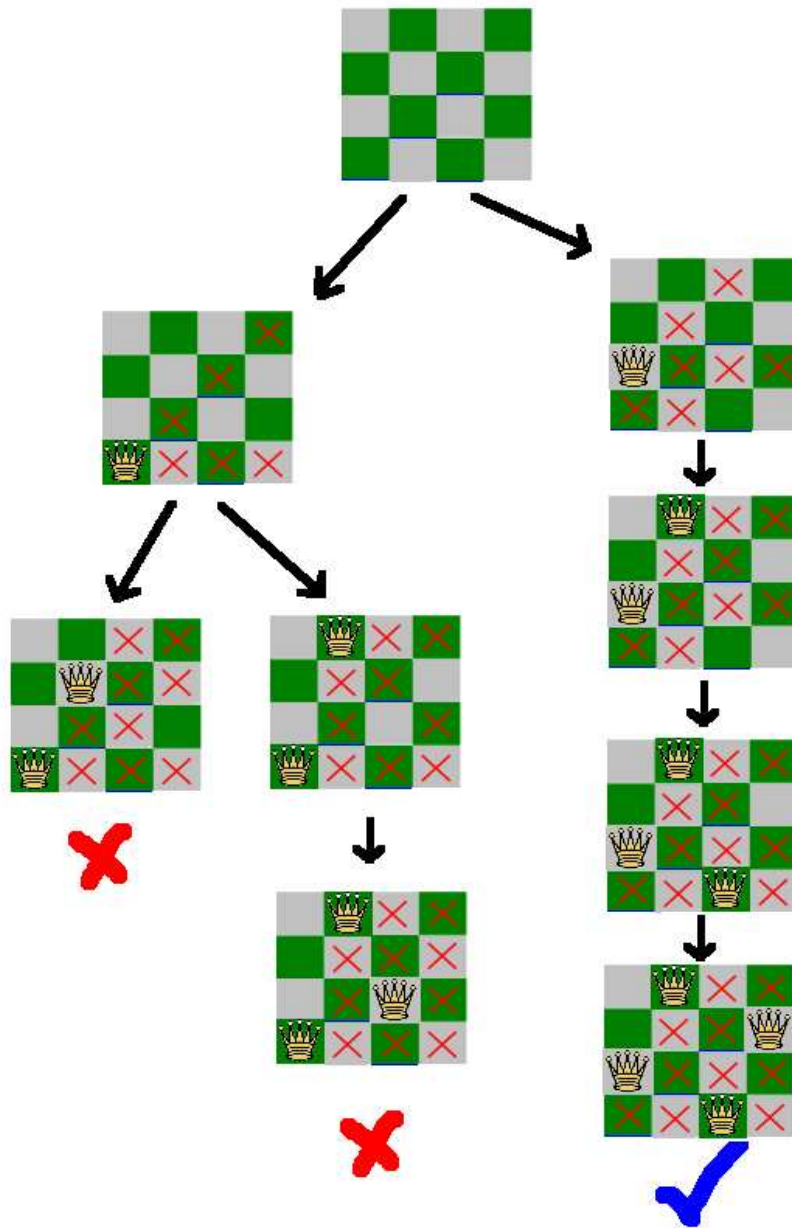


Figure 5: The complete search tree for solving the 4 queens problem using forward checking.

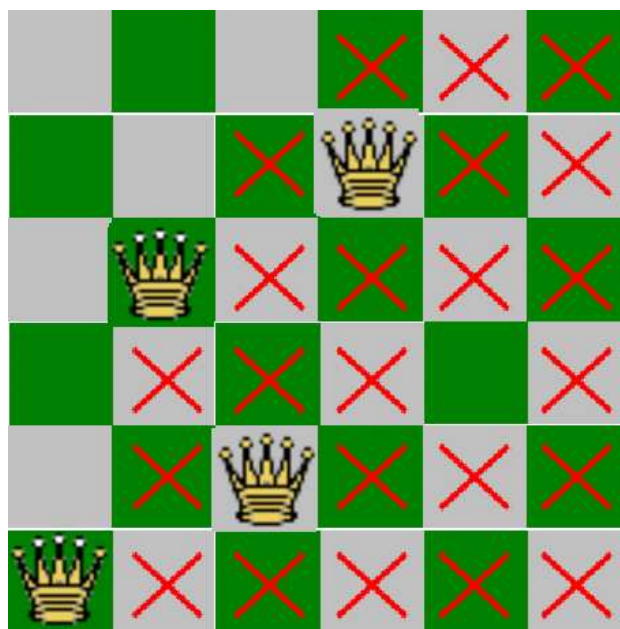


Figure 6: An example of a search node in the 6 queens puzzle where forward checking saves a lot of time relative to consistency checking. As soon as we instantiate the fourth queen, we detect that the sixth queen has no moves left, and we can backtrack. Using consistency checking, we would have to wait until we instantiated the sixth queen, thereby going two levels deeper into the search tree.

are not symmetric, so we must perform constraint propagation separately on (V_i, V_j) and (V_j, V_i) . Because removing a value from one domain might make it possible to remove other values from other domains, we must keep repeating the above procedure until the graph is arc consistent. (Make sure you understand why this procedure will eventually terminate and guarantee arc consistency.)

In our description of the algorithm thus far, we talked about iterating over all pairs of variables V_i, V_j . We described the procedure this way since it made it conceptually simpler to describe, but repeatedly iterating over all pairs of variables would be unnecessarily inefficient, and real implementations of arc consistency typically use somewhat more efficient ways to organize the ordering of the constraint propagation steps and keep track of what constraints need to be propagated next. (For example, one such algorithm is **AC-3**, which is described in the textbook.)

Suppose we run arc consistency to convergence, until no more domain values can be eliminated. What happens when we're done propagating constraints at a particular point in the search tree? There are three possibilities:

- For some variable V_i , we have $D_i = \emptyset$. We know then the problem is inconsistent, so we can backtrack.⁴
- For all variables V_i , we have $|D_i| = 1$. I.e., every D_i is a singleton set. In this case, we are done—we've found a solution, and can simply assign all remaining variables to the value in their corresponding domains. (Why are we guaranteed that this won't violate any additional constraints?)
- D_i has multiple remaining values. In this case, we have to keep searching. I.e., we instantiate some value for D_i , and keep searching.

In practice, full arc consistency is often too expensive to run deep in the inner loop of a search algorithm, so we usually make a compromise between effective inference and computational efficiency. For instance, one common choice is to check arc consistency only at the very highest levels of the search tree, and then use only forward checking for the rest of the search. Alternatively, we might check arc consistency only for a fixed subset of the constraints. This is the usual tradeoff between expensive inference to allow

⁴Note that these domain variables D_i (in both the cases of forward checking and constraint propagation) depend on what node of the search tree we are in. In particular, when our DFS procedure backtracks and “unassigns” some variable, we need some sort of data structure to keep track of which values were previously removed from a given domain D_i and need to be added them back as part of the backtracking step.

us to search fewer nodes, vs. using cheaper inference but more brute force to search more nodes.

3 Heuristics

In the last section, we outlined three inference algorithms to use inside the search. Even with these algorithms, however, as part of DFS we have to make two arbitrary decisions:

- Which variable to instantiate next.
- For a given variable, which values in the domain to try first.

For both of these, we have assumed a naive approach, where we arbitrarily numbered our variables V_1, \dots, V_n and the domain values d_1, \dots, d_n , and we checked them in those orders. It turns out we can do significantly better using two simple heuristics. These heuristics can be used with any of the three inference algorithms presented above.

3.1 Most constrained variable

First, we tackle the question of which variable to instantiate next. Intuitively, if we are exploring the wrong part of the search tree, we want to find out as soon as possible. One way to do this is by using the **most constrained variable (MCV)** heuristic (also called the **minimum remaining values (MRV)** heuristic), in which we start by instantiating the variable with the fewest remaining values left in its domain. The fewer values we have left, the fewer we have to try before we reach a contradiction. As a special case of this heuristic, if a variable only has one value left in its domain, we may as well go ahead and assign that variable.

3.2 Least-constraining value

Once we've chosen a variable to instantiate, in what order do we try the different values? In this case, the order in which we choose values won't affect how long it takes to find a contradiction. If there is a contradiction, we will have to try all of the values no matter what. On the other hand, if we are in the correct part of the search tree, we want to find a correct solution as soon as possible. This suggests choosing the value which would remove the fewest values from other variables' domains, or the **least constraining**

value (LCV), since this leaves more of our options open, so that there's hopefully more likely for there to be a solution to be found still.

Note a key difference between these two heuristics in terms of how we treat them when backtracking. Under the LCV heuristic, if we color Country 2 green and it doesn't work, it makes sense to backtrack and try red. In contrast, choosing variable ordering is a **non-backtrack step**. E.g., if we choose to color Country 2 first and find that that doesn't work—none of the values work with our current assignment—then it doesn't make sense to go back and try coloring Country 4 first.