




# Artificial Intelligence Programming

## *Constraint Satisfaction*

Chris Brooks

Department of Computer Science  
University of San Francisco



# Constraint Satisfaction

- So far, we've focused on using search to find the *best* solution.
- In many cases, you just need to find a solution that satisfies some criteria.
- These criteria are called constraints.
- A problem in which we want to find any solution that satisfies our constraints is a *constraint satisfaction problem*
- Constraints provide us with additional knowledge about the problem that we can exploit.
  - We can also consider optimizing constrained problems.

# Examples

---

- Toy Problems
  - Map coloring, N-queens, cryptarithmic
- Real life problems
  - Scheduling, register allocation, resource allocation

# Formalizing a CSP

- We can model a CSP as a set of variables  $\{x_1, x_2, \dots, x_n\}$
- Each variable as a domain of possible values  
 $D_1, D_2, \dots, D_n$
- We also have a set of constraints  $C_1, C_2, \dots$ 
  - Unary constraints:  $x < 10, y \bmod 2 == 0$ , etc
  - Binary constraints:  $x < y, x + y < 50$ , no two adjacent squares can be the same color, etc
  - N-ary constraints:  $x_1 + x_2 + \dots + x_n = 75$ , weight of chassis plus engine plus body  $< 3000$  lbs, etc.
- An assignment of values to variables that satisfies all constraints is called a *consistent* solution.
- We might might also have an objective function  $y = f(x_1, \dots, x_n)$  that lets us compare solutions.

# Approaches

---

- If the domain of all variables is continuous (i.e. real numbers) and constraints are all linear functions, we can use *linear programming* to solve the problem.
  - Express the problem as a system of equations
- In other cases, we can use *dynamic programming*.
- Dynamic programming is a form of search.
- In the most general case, we can express a CSP as a search problem.

# Solving CSPs with search

- We'll begin with an initial state: no values assigned to  $x_1, \dots, x_n$
- An action assigns values to variables.
- A goal is an assignment to each variable such that all constraints are met.
- The successor function returns all possible single assignments such that constraints are still met.
  - Notice that our solution for this sort of problem is the goal state, as opposed to a path through the state space.

# Constraint graph

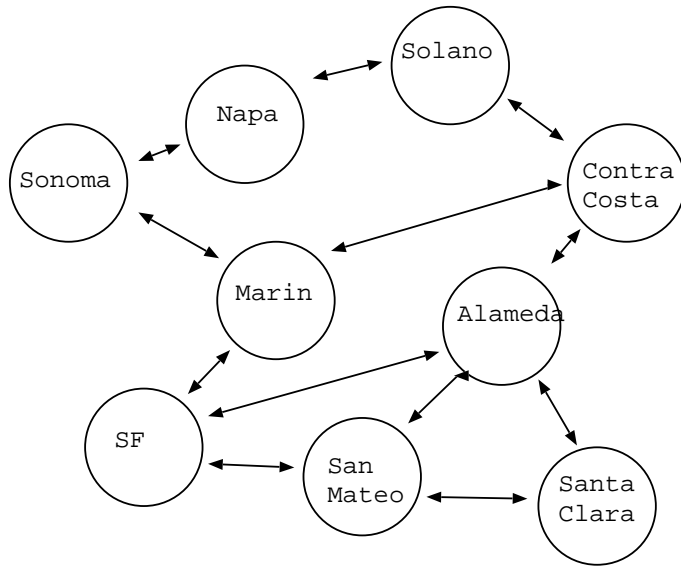
- Often, the most difficult part of solving a CSP is formulating the problem.
- It is often useful to visualize the CSP as a *constraint graph*
- Nodes represent variables, edges represent binary constraints.
  - For n-ary constraints, we must add nodes that represent combinations of values.

# Example - Coloring a Bay Area Map



- Can we color this map using only four colors (R, Y, B, G), so that no adjacent counties have the same color?

# Example - Coloring a Bay Area Map



- Can we color this map using only four colors (R,Y,B,G), so that no adjacent counties have the same color?

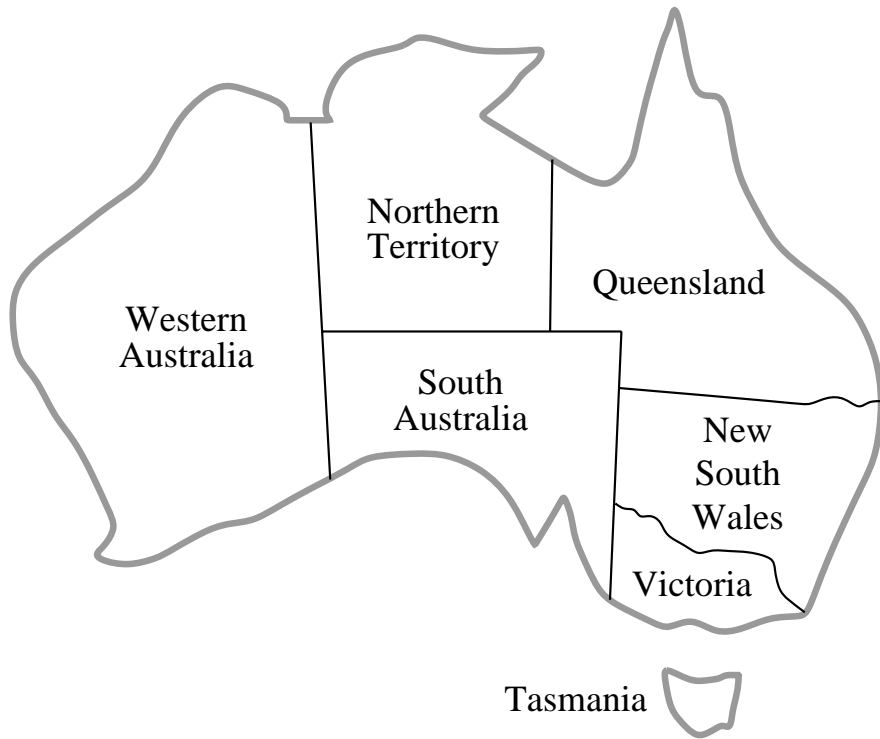
# Example - Coloring a Bay Area Map

- Initially, pick a color for SF. (Red)
- Our successor function will return all possible colorings that don't violate the constraint.
  - Marin = Blue, Solano = Red, Alameda = Yellow, Napa = Yellow, etc
- We can be more clever about how to approach this problem
  - CSPs are commutative; it doesn't matter which order we assign colors to counties.
  - Therefore, we should consider one assignment at a time.
- For the moment, let's start by always assigning a color to the county with the smallest domain.

# Example - Coloring a Bay Area Map

- SF = Red.
- SF = Red, Marin = Blue
- SF = Red, Marin = Blue, Alameda = Yellow
- SF = Red, Marin = Blue, Alameda = Yellow, CC = Green
- etc.
- In this case, we can find a consistent four-coloring,
- What about a 3-coloring?

# Example - Australia



- Three-coloring the map of Australia (Red, Green, Blue)
- Let's draw the search tree
- Initially, we color  $Q=Red$ .
- What are possible successors?

# Example - Australia

- Neighbors of Q have a domain of Green, Blue - smallest.
- Choose a neighbor and select a color that satisfies all constraints.
- NSW = Green.
- Now SA has a domain of size 1 (Blue)
- Coloring SA then fixes the choices for V and NT.

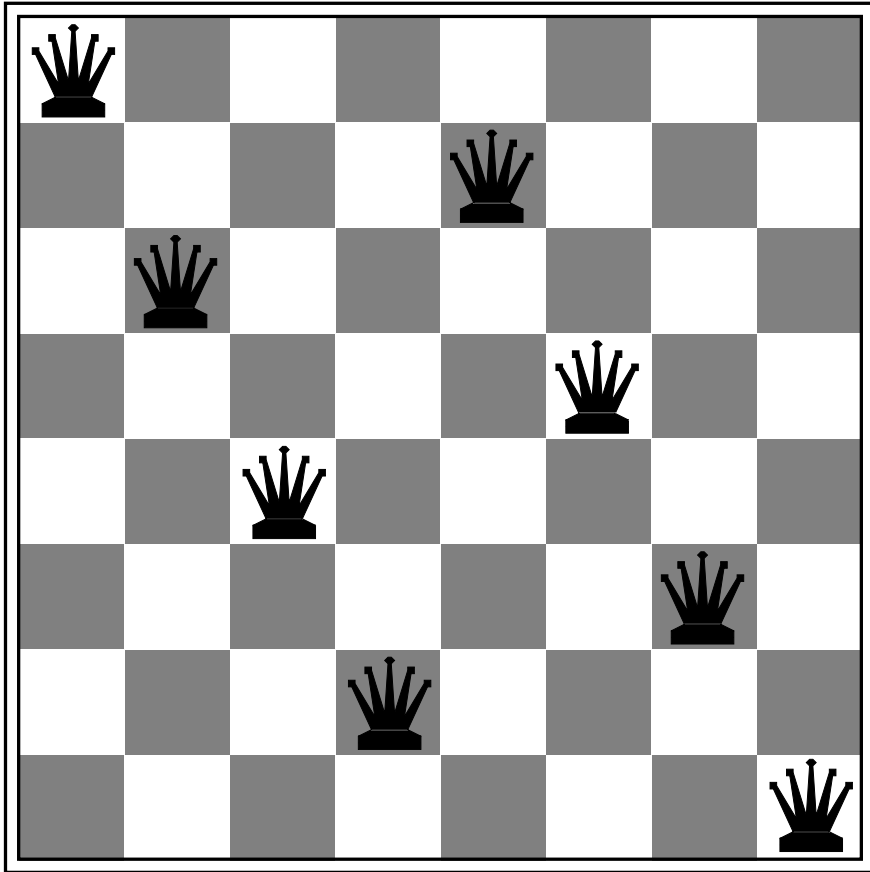
# Heuristics

- How do we pick which country to color next?
- How do we choose what color to give it?
- Intuition: always try to make decisions that leave as much flexibility as possible.
- Most constrained variable: pick the variable (country) that has the fewest possible choices.
- Least constraining value: pick the value (color) that has the least effect on possible values for other variables.

# Backtracking

- In the previous example, we were fortunate.
  - We never made a bad choice.
  - What if we had colored  $Q = \text{red}$ ,  $NSW = \text{green}$ ,  $V = \text{blue}$ ?
- Usually, when solving a CSP, there are times when you have to 'undo' a bad choice.
- This is called *backtracking*.

# Example - 8-queens



- Problem: place each queen on the board so that no queen is attacking another.
- We can reduce this to: What row should each queen be in?

# Chronological Backtracking

- The easiest approach is to use depth-first search.
- If we reach a point where we can't place a queen, back up and undo the most recent placement.
- If that placement can't be changed, undo its predecessor.
- Always undo assignments in reverse order of when they were done.
- This is called *chronological backtracking*.

# Chronological Backtracking

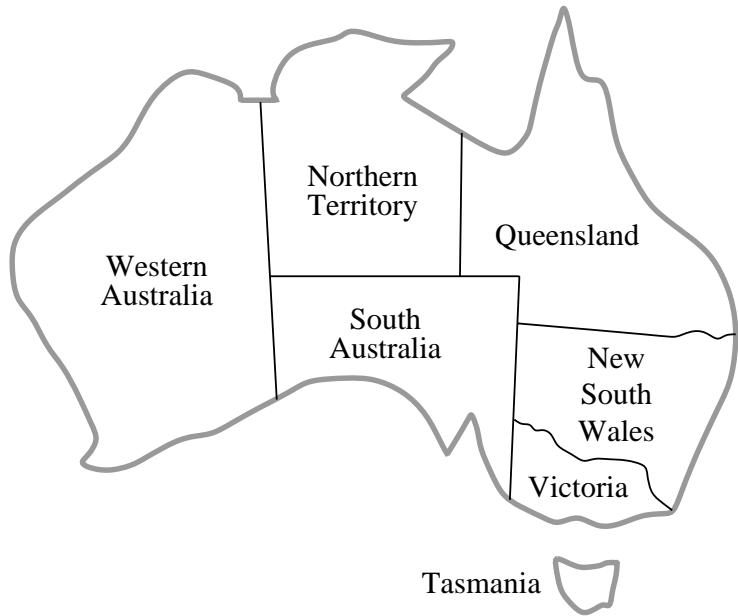
a1(0)	○			
a2(0)				○
a3(0)		○		
a4(1)			○	

- Problem: we make a bad decision early on that isn't noticed until later.
- In this 4-queens problem, there is no solution that has the first queen in the top row.
- We'll spend a lot of time trying different combinations for queens 2 and 3, even though there is no possible solution that can be reached.
- How can we better identify which decision is causing a constraint violation?

# Backjumping

- What we'd like to do is identify those variables that are causing a problem and change them directly.
- To do this, when we reach a variable for which we cannot find a consistent value, we look for all variables that it shares a constraint with.
- We call these variables the conflict set.
- We then 'unroll' our search and undo the most recently-set variable in the conflict set.

# Example - Australia



- Let's say our search algorithm did the following coloring:
  - Q: Red, NSW: Green, V: Blue, T: Red
- There is no consistent color for SA.
- Backtracking will try all other colors for T, which cannot possibly help.
- The conflict set for SA is  $\{Q, NSW, V\}$ .
- We backjump and undo V. Once V: Red, we can color SA.

# CSP summary

---

- Many interesting real-world problems can be formulated as CSPs.
  - Scheduling, resource allocation, design, etc.
- CSPs can be solved using DFS
  - Successor chooses an unassigned variable and gives it a value
- Problem structure can be exploited to guide search
  - Heuristics, intelligent backtracking.

# Branch-and-Bound Search

- We can also use heuristics to prune portions of a search tree.
- In many problems, we need to search for an optimal solution in a depth-first manner.
  - Laying out components on a chip, finding the fastest schedule, TSP
- How can we identify regions of the search space that can't possibly lead to an optimal solution?

# Applying heuristics

- If we have an admissible heuristic, we can identify branches that cannot lead to a solution.
- Search down the left-most branch of the search tree and find the first candidate solution.
- As we search down the remaining branches, we apply our heuristic at each step.
- If we reach a point where the f-cost of an incomplete solution is higher than the previous best solution, we can discard the branch (assuming we want a minimal solution)
  - Since  $h$  underestimates, the true solution cost will be at least as high as  $f$ .
  - If  $f$  is already worse than what we've seen, there's no point in expanding this branch.

# Branch-and-Bound Search

- This approach is used extensively in game-playing search to eliminate branches that can't lead to a winning strategy.
- Challenge: Branch-and-bound is most effective when very good solutions are found early in the search.
  - Why is this?
- This leads to the development of heuristics for selecting the order in which nodes are placed on the stack in DFS.
- Most promising node is at the top.

# Summary

- Heuristics can be used not only to guide search, but also to reduce the size of the search space.
- In backjumping, they can be used to 're-order' the tree.
- In branch-and-bound search, they can be used to prune pieces of the search space.
- The more knowledge you have about a problem, the more aggressive you can be about solving it.
  - Caveats:
    - Your knowledge must be correct.
    - Your solution will be less applicable if the problem changes, or when applied to another problem.