Artificial Intelligence Programming

Constraint Satisfaction Programming

Cindi Thompson

Department of Computer Science

University of San Francisco

Outline

- Constraint Satisfaction Problems - Definition
- Constraints and Graphs
- Backtracking search
- Conflicts and consistency checking

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, cryptarithmetic
- Real life problems
  - Scheduling, register allocation, resource allocation

Formalizing a CSP

Standard search problem

- A *state* is a set of variables $\{x_1, x_2, ..., x_n\}$ with assigned values
- Each variable has a domain of possible values $D_1, D_2, ..., D_n$
- We also have a set of constraints $C_1, C_2, ..., C_m$
- The *goal test* is the set of constraints specifying allowable combinations of values for subsets of variables

Allows useful general purpose algorithms with more power than standard search algorithms

Formalization, Continued

Constraint examples

- Unary constraints: $x < 10, y \mod 2 == 0$, etc
- Binary constraints: $x < y, x + y < 50$, no two adjacent squares can be the same color, etc
- N-ary constraints: $x_1 + x_2 + ... + x_n = 75$, 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 $y = f(x_1, ..., x_n)$ that lets us compare solutions.

Example: Map Coloring

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D_i = \{$ red, green, blue $\}$
- Constraints: Adjacent regions must have different colors
- e.g., $WA \neq NT$, or (WA,NT) in {(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)}

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 $x_1, ..., x_n$
- 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.

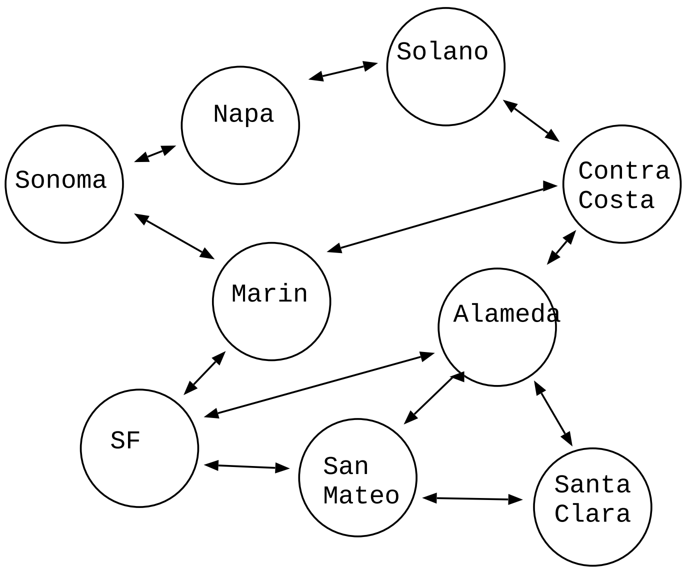
Constraint graph

- 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 = B, Marin = Y, Marin = G, Solano = R, Solano = Y, ..., Alameda = Y, Napa = Y, 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: variable (country) that has the fewest possible choices.
- Most constraining variable: variable with the most constraints on remaining variables.
- Least constraining *value*: value (color) that has the least effect on possible values for other variables.

Most constrained variable

- Most constrained variable:

  choose the variable with the fewest legal values

- a.k.a. minimum remaining values (MRV) heuristic

Most constraining variable

Tie-breaker among most constrained variables

- Choose the variable with the most constraints on remaining variables
- You're going to have to assign it eventually!

Least constraining Value

Given a variable, choose the least constraining value

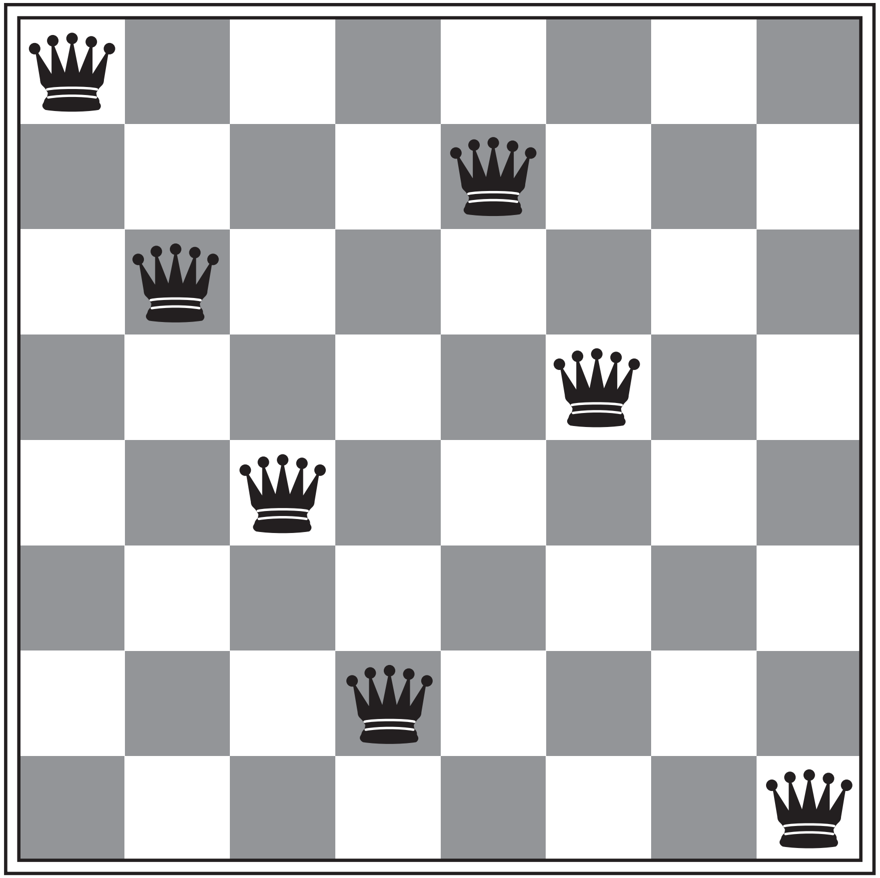
- Value that rules out the fewest values in the remaining variables

Combining these heuristics makes 1000-queens feasible

Backtracking

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

Example - 8-queens

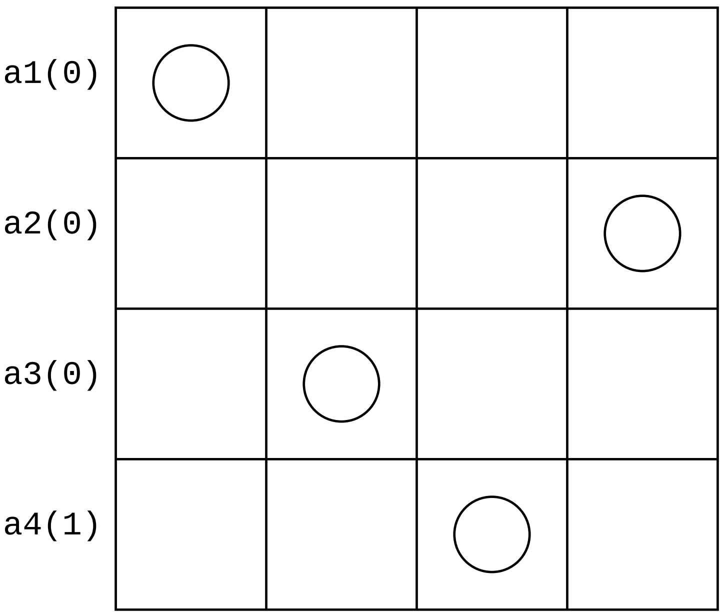


- Problem: place each queen on the board so that no queen is attacking another.
- We can reduce this to: What row should each queen be in?

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 asignments in reverse order of when they were done.
- This is called *chronological backtracking*.

Chronological Backtracking



- 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.

Varieties of CSPs

Discrete variables

- finite domains:
  - n variables, domain size d: $O(dn)$ complete assignments
  - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
- infinite domains:
  - integers, strings, etc.
  - e.g., job scheduling, variables are start/end days for each job need a constraint language, e.g., StartJob1 + 5 $\leq$ StartJob3

  Continuous variables

  - e.g., start/end times for Hubble Space Telescope observations

Looking ahead

- Backtracking is useful, but only lets us undo mistakes.
- Lookahead allows us to examine a partial solution and restrict the search space.
- Idea: can we look at a partial solution and determine whether it could lead to a complete solution?
- Can we look at a partial solution and determine what values unassigned variables can take on?

Forward checking

Simplest way to do this: Forward checking

Idea

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

Forward checking

Spelled out:

- Assume we have n variables $x_1, \ldots, x_n$
- We are expanding the ith variable.
- When we consider a value d for x_i, we examine each variable x_{i+1} through x_n
- For each examined variable, we ask whether there is any value it could take on that would lead to a consistent solution.
- If the answer is no, we don't bother to expand x_i

Flaws in forward checking

- Propagates information from assigned to unassigned variables,
- but doesn't provide early detection for all failures
- Examines states one by one and sees each has consistent constraints

Constraint propagation techniques repeatedly enforce constraints locally

Arc Consistency

Extends forward checking to look at pairs of variables

- $X \to Y$ is consistent iff for every value x of X there is some allowed y

Use the constraint graph edges

Arc Consistency

- $X \rightarrow Y$ is consistent iff for every value x of X there is some allowed y

Use the constraint graph edges

- If X loses a value, neighbors of X need to be rechecked
- Can be run as a preprocessor or after each assignment

AC-3 Pseudocode

```
v = the variables in our problem.
d[v] is the list of values in the domain of each v

for vertex in v :
   neighbors = all vertices in v that share a constraint with vertex
   for n in neighbors :
      for value in d[v] :
         if there is no value in d[n] consistent with value:
            remove value from d[v]
            if d[v] is empty, return failure
repeat until d[v] does not change for any v
```

n-ary Consistency

Arc consistency works well, but can still miss failure states involving constraints over three variables.

- Building a car: weight of frame, engine, and chassis cannot exceed 2000 lbs.

We can extend arc consistency to deal with n-ary constraints for an increase in running time.

Local Search

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - i.e., hill-climb with h(n) = total number of violated constraints

Example: 4-Queens

- States: 4 queens in 4 columns (4^4=256 states)
- Actions: move queen in column
- Evaluation: h(n) = number of attacks

Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)

CSP summary

- CSPs are a special kind of problem:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values
- Many interesting real-world problems can be formulated as CSPs.
- CSPs can be solved using DFS, one var assigned per action
- Problem structure can be exploited to guide search
  - Value-selection Heuristics, Variable ordering
  - Intelligent backtracking
  - Constraint propagation to detect inconsistencies