Constraint Satisfaction Problems
Constraint satisfaction basically boils down to graph coloring.
Graph coloring is NP complete, so any solution is *worst case* exponential time.
This is a valid 4-coloring...we're just not using the 4\textsuperscript{th} color!
Note map coloring can be defined as graph coloring. Each state in map is represented as a node in the graph, with edges between nodes if the states are adjacent.

All map coloring can be represented as graph coloring, but not all graph coloring can be represented as map coloring. This is because maps are planar graphs.

Example: Map-Coloring

- **Variables** WA, NT, Q, NSW, V, SA, T
- **Domains** $D_i = \{\text{red, green, blue}\}$
- **Constraints**: adjacent regions colored differently
Example: Map-Coloring

- **Solutions** are complete and consistent assignments, e.g.,
  - WA = red, NT = green, Q = red, NSW = green,
  - V = red, SA = blue, T = green

Complete: every state is colored. Consistent: it’s a valid soln.
Binary CSP can usually be expressed as a graph, because each edge has 2 endpoints, so each edge can be one constraint.
CSP Definition

- CSP = \{V, D, C\}
- Variables: \(V = \{V_1, \ldots, V_n\}\)
- Domain: The set of \(d\) values that each variable can take
- Constraints: \(C = \{C_1, \ldots, C_k\}\)
- Each constraint consists of a tuple of variables and a list of values that the tuple is allowed to take for this problem
  - Example:
    \([(V_2, V_3), \{(R, B), (R, G), (B, R), (B, G), (G, R), (G, B)\}]]
- Constraints are usually defined implicitly
  - Example: \(V_i \neq V_j\) for every edge \((i, j)\)

Example \([(V_2, V_3), \{(R, B), \ldots\}]]\): the possible values the tuple \((V_2, V_3)\) can take

The second example is an easier way to write the first.
Varieties of constraints

- **Unary** constraints involve a single variable,
  - e.g., SA ≠ green
- **Binary** constraints involve pairs of variables,
  - e.g., SA ≠ WA
- **Higher-order** constraints involve 3 or more variables

Higher order example: allDif(X,Y,Z)

allDif(X,Y,Z) means X, Y, and Z all must be different.
Binary CSP

- Each constraint is either unary or binary, i.e., refers to one variable or to two variables.
- It is possible to convert any n-ary CSP to a binary CSP.

Note you can convert any CSP to binary. Ex: diff(X,Y,Z) → diff(X,Y), diff(Y,Z), diff(Z,X)

When you convert to binary, # of constraints can blow up
N-Queens

Given one queen per column, find row for each queen, such that there no queen attacks another queen.

\[ Q_1 = 1 \quad Q_2 = 3 \]

i: column
Q_i: row
Example: N-Queens

- Variables: $Q_i$
- Domains: $D_i = \{1, 2, 3, 4\}$
- Constraints
  - $Q_i \neq Q_j$ (cannot be in the same row)
  - $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

- Valid values for $(Q_1, Q_2)$ are
  $(1, 3)$, $(1, 4)$, $(2, 4)$, $(3, 1)$, $(4, 1)$, $(4, 2)$
Cryptarithmetic

SEND

+MORE

MONEY
Example: Cryptarithmetic

- Variables
  \( D, E, M, N, O, R, S, Y \)

- Domains
  \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

- Constraints
  \( M \neq 0, S \neq 0 \) (unary constraints)
  \( Y = D + E \) OR \( Y = D + E - 10 \).
  \( D \neq E, D \neq M, D \neq N, \) etc.

\[
\begin{align*}
\text{SEND} + \text{MORE} & = \text{MONEY} \\
S & \implies \{6\} \\
E & \implies \{0, 1, 5\} \\
M & \implies \{0, 1, 6\} \\
D & \implies \{2, 6\}
\end{align*}
\]
Varieties of CSPs

• Discrete variables
  – finite domains:
    • $n$ variables, domain size $d \rightarrow O(d^n)$ 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
• Continuous variables
  – e.g., start/end times for Hubble Space Telescope observations
  – linear constraints solvable in polynomial time by linear programming
Example: Scheduling

- A set of N jobs, \( J_1, \ldots, J_n \).
- Each job \( j \) is composed of a sequence of operations \( O_{i,j}^1, \ldots, O_{i,j}^{L_j} \).
- Each operation may use resource \( R \), and has a specific duration in time.
- A resource must be used by a single operation at a time.
- All jobs must be completed by a due time.
- Problem: assign a start time to each job.
In addition to what we said before, need to wait for $O_1^i$ to finish before $O_2^i$ can start, etc.

- 4 jobs
- 4 resources
- 10 operations
$S_{ij}$: start time for operation $j$ of job $i$

$T_{ij}$: duration of operation $j$ of job $i$
Resource constraints

Operations (1,1), (2,1), and (3,2) share the same resource R1
How to use DFS? For example, you’d have a tree with root as start state, children as \( V_1=\text{red} \), \( V_1=\text{blue} \), \( V_1=\text{green} \), and then branch out on \( V_2 \), etc.

### CSP as a Standard Search Problem

- **State:** assignment to \( k \) variables with \( k+1, \ldots, N \) unassigned
- **Successor:** Assignment of a value to variable \( k+1 \), keeping the others unchanged
- **Start state:** \( (V_1=?, V_2=?, V_3=?, V_4=?, V_5=?, V_6=?) \)
- **Goal state:** All variables assigned with constraints satisfied
- **No concept of cost on transition; just a solution, no path**

Example state:

\( (V_1=\text{G}, V_2=\text{B}, V_3=?, V_4=?, V_5=?, V_6=?) \)
How many possible successors? Number of colors. You'll get to the bottom of the tree and have to backtrack because you made a dumb assignment.

You can check if V1 and V2 don't violate.

How do you pick which node to explore next? Choose the most constrained node. But we're not going to get that smart yet.
DFS Improvements

- Evaluate only value assignments that do not violate any constraints with the current assignments
- Don’t search branches that obviously cannot lead to a solution
- Predict valid assignments ahead
- Control order of variables and values
<table>
<thead>
<tr>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
</tr>
</thead>
</table>

Order of values:
(B,R,G)

<table>
<thead>
<tr>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>R</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>R</td>
<td>R</td>
<td>B</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>R</td>
<td>R</td>
<td>B</td>
<td>G</td>
<td>?</td>
</tr>
</tbody>
</table>
Backtracking DFS

Order of values: (B,R,G)

Don’t even consider that branch because $V_2 = B$ is inconsistent with the parent state.

Backtrack to the previous state because no valid assignment can be found for $V_6$. 
Backtracking DFS

- For every possible value $x$ in $D$:
  - If assigning $x$ to the next unassigned variable $V_{k+1}$ does not violate any constraint with the $k$ already assigned variables:
    - Set the variable $V_{k+1}$ to $x$
    - Evaluate the successors of the current state with this variable assignment
  - If no valid assignment is found: Backtrack to previous state
  - Stop as soon as a solution is found

This can be really slow.
Backtracking DFS Comments

• Additional computation: At each step, we need to evaluate the constraints associated with the current candidate assignment (variable, value).

• Uninformed search, we can improve by predicting:
  – What is the effect of assigning a variable on all of the other variables?
  – Which variable should be assigned next and in which order should the values be evaluated?
  – When a branch fails, how can we avoid repeating the same mistake?
This works slightly better than DFS.

---

### Forward Checking

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

<table>
<thead>
<tr>
<th></th>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_3$</th>
<th>$V_4$</th>
<th>$V_5$</th>
<th>$V_6$</th>
</tr>
</thead>
</table>

Warning: Different example with order $(R,B,G)$
Forward Checking

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

<table>
<thead>
<tr>
<th></th>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>O</td>
<td>X</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>?</td>
</tr>
</tbody>
</table>

Place X’s in places that can’t be red anymore. If V₁ is red, then V₂, V₄, and V₅ can’t be assigned red.
**Forward Checking**

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

<table>
<thead>
<tr>
<th></th>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>O</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>X</td>
<td>?</td>
<td>X</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

![Graph Diagram]
Forward Checking

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

<table>
<thead>
<tr>
<th></th>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_3$</th>
<th>$V_4$</th>
<th>$V_5$</th>
<th>$V_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>$O$</td>
<td>$O$</td>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>$O$</td>
<td></td>
<td>$?$</td>
<td>$X$</td>
<td>$?$</td>
<td></td>
</tr>
<tr>
<td>$G$</td>
<td></td>
<td></td>
<td>$?$</td>
<td>$?$</td>
<td>$?$</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

Graph with nodes $V_1$ to $V_6$ connected by lines.
Forward Checking

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

<table>
<thead>
<tr>
<th></th>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_3$</th>
<th>$V_4$</th>
<th>$V_5$</th>
<th>$V_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>$O$</td>
<td>$O$</td>
<td>$X$</td>
<td>$X$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>$O$</td>
<td>$O$</td>
<td>$X$</td>
<td>$X$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$G$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$?$</td>
<td>$?$</td>
</tr>
</tbody>
</table>
The X’s mean there are whole branches of the search tree that we don’t look at. We still need to backtrack, but this is much more efficient. This is an example of pruning a search tree (chopping off branches)
Constraint Propagation

- Forward checking does not detect all the inconsistencies, only those that can be detected by looking at the constraints which contain the current variable.

- Can we look ahead further?

<table>
<thead>
<tr>
<th></th>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_3$</th>
<th>$V_4$</th>
<th>$V_5$</th>
<th>$V_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$G$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

At this point, it is already obvious that this branch will not lead to a solution because there are no consistent values in the remaining domain for $V_5$ and $V_6$. 

Constraint Propagation, not “just” checking

- $V =$ variable being assigned at the current level
- Set variable $V$ to a value in $D(V)$
- For every variable $V'$ connected to $V$:
  - Remove the values in $D(V')$ that are inconsistent with the assigned variables
  - For every variable $V''$ connected to $V'$:
    - Remove the values in $D(V'')$ that are no longer possible candidates
    - And do this again with the variables connected to $V''$
      - ...until no more values can be discarded

This is even better.
Constraint Propagation, not “just” checking:

- $V$ = variable being assigned
- Value in $D(V)$ inconsistent with the assigned variables
- For every variable $V''$ connected to $V$:
  - Remove the values in $D(V'')$ that are inconsistent with the assigned variables
  - And do this again with the variables connected to $V''$
    - ...until no more values can be discarded

New: Constraint Propagation

Forward Checking as before
CP For Graph Coloring

Propagate \((node, color)\)

1. Remove color from the domain of all of the neighbors

2. For every neighbor \(N\):
   
   If \(D(N)\) was reduced to only one color after step 1 \((D(N) = \{c\})\):

   Propagate \((N,c)\)
After Propagate \((V_1, R)\):

<table>
<thead>
<tr>
<th></th>
<th>(V_1)</th>
<th>(V_2)</th>
<th>(V_3)</th>
<th>(V_4)</th>
<th>(V_5)</th>
<th>(V_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R)</td>
<td>O</td>
<td>X</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>?</td>
</tr>
</tbody>
</table>
Uh oh, V5 can only be one color. Take it. This is a key difference
Now that you set $V_5$, $V_4$ can now only be one color!
### After Propagate \((V_2, B)\):

<table>
<thead>
<tr>
<th></th>
<th>(V_1)</th>
<th>(V_2)</th>
<th>(V_3)</th>
<th>(V_4)</th>
<th>(V_5)</th>
<th>(V_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R</strong></td>
<td>O</td>
<td>(X)</td>
<td>(X)</td>
<td>(X)</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>O</td>
<td>(X)</td>
<td>?</td>
<td>(X)</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td><strong>G</strong></td>
<td>?</td>
<td>(X)</td>
<td>?</td>
<td>(X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: We get directly to a solution in **one step of CP** after setting \(V_2\) **without any additional search**

Some problems can even be solved by applying CP directly without search
Variable and Value Heuristics

So far we have selected the next variable and the next value by using a fixed order

1. Is there a better way to pick the next variable?
2. Is there a better way to select the next value to assign to the current variable?
CSP Heuristics: Variable Ordering I

- **Most Constraining Variable**
  - Selecting a variable which contributes to the largest number of constraints will have the largest effect on the other variables
  - Equivalent to finding the variable that is connected to the largest number of variables in the constraint graph.

<table>
<thead>
<tr>
<th>V_1</th>
<th>V_2</th>
<th>V_3</th>
<th>V_4</th>
<th>V_5</th>
<th>V_6</th>
<th>V_7</th>
</tr>
</thead>
</table>

Setting variable V_5 affects 4 variables

Setting variable V_2 (or V_3, V_4) affects fewer variables
Fail-first is good, because you don’t waste your time looking at branches of the search tree that can’t possibly work.
Pick green for V₃ as it adds no new constraints (We already know that V₄ can’t be green!)
Conclusion – Generic CSP Solution

• Repeat until all variables have been assigned:
  • Apply a consistency enforcement procedure
    – Forward checking
    – Constraint propagation
  • If no solutions left:
    – Backtrack to a previous variable
  • Else
    – select the next variable to be assigned
      • Using variable ordering heuristic
    – Select a value to try for this variable
      • Using value ordering heuristic