Canonical Example: Graph Coloring

- Consider $N$ nodes in a graph
- Assign values $V_1, ..., V_N$ to each of the $N$ nodes
- The values are taken in $\{R, G, B\}$
- Constraints: If there is an edge between $i$ and $j$, then $V_i$ must be different from $V_j$
Canonical Example: Graph Coloring

CSP Definition

- **CSP** = \{V, D, C\}
- **Variables**: V = \{V_1, ..., V_n\}
  - Example: The values of the nodes in the graph
- **Domain**: The set of d values that each variable can take
  - Example: \(D = \{R, G, B\}\)
- **Constraints**: C = \{C_1, ..., C_p\}
  - 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 → A function is defined to test if a tuple of variables satisfies the constraint
    - Example: \(V_i \neq V_j\) for every edge \((i, j)\)
Binary CSP

- Variable $V$ and $V'$ are connected if they appear in a constraint.
- Neighbors of $V = \{v \in V \mid (v, v') \in E\}$ are variables that are connected to $V$.
- The domain of $V$, $D(V)$, is the set of candidate values for variable $V$.
- $D_i = D(V_i)$

- Constraint graph for binary CSP problem:
  - Nodes are variables
  - Links represent the constraints
  - Same as our canonical graph-coloring problem

N-Queens

![N-Queens Diagram]

$Q_1 = 1 \quad Q_2 = 3$
Example: N-Queens

- Variables: $Q_i$
- Domains: $D_i = \{1, 2, 3, 4\}$
- Constraints
  - $Q_i \neq Q_j$ (cannot be in 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
```
Search Space

Example state:

$$(V_1=G, V_2=B, V_3=\?, V_4=\?, V_5=\?, V_6=\?)$$

- **State**: assignment to $k$ variables with $k+1,...,N$ unassigned
- **Successor**: The successor of a state is obtained by assigning 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 → We just want to find a solution, we don’t worry how we get there

```
<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>
</tr>
</thead>
</table>

<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>
</tr>
</thead>
</table>

<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>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
```

Really dumb assignment

9d
Depth First Search

- Recursively:
  - For every possible value in \( D \):
    - Set the next unassigned variable in the successor to that value
    - Evaluate the successor of the current state with this variable assignment
    - Stop as soon as a solution is found

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
Outline

- Definitions
- Standard search
- Improvements
  - Backtracking
  - Forward checking
  - Constraint propagation
- Heuristics:
  - Variable ordering
  - Value ordering
- Examples
- Tree-structured CSP
- Local search for CSP problems

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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>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

- 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

Order of values: \((B,R,G)\)

Backtrack to the previous state because no valid assignment can be found for $V_6$
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?

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>❓</td>
<td>❓</td>
<td>❓</td>
<td>❓</td>
<td>❓</td>
<td>❓</td>
</tr>
<tr>
<td>B</td>
<td>❓</td>
<td>❓</td>
<td>❓</td>
<td>❓</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>

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>
**Forward Checking**

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

<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>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>?</td>
<td>X</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

![Diagram of Forward Checking](image)

---

**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>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

![Diagram of Forward Checking](image)
Forward Checking
- Keep track of remaining legal values for unassigned variables
- Backtrack when any variable has no legal values

<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>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>O</td>
<td>O</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>O</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are no valid assignments left for V_6, we need to backtrack.

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>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></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>G</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

- \( V \) = variable being assigned at the current level of the search
- 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

New: Constraint Propagation

Forward Checking as before

- 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