CSP slides are drawn from or inspired by a multitude of sources including:

Faheem Bacchus
Sheila McIlraith
Andrew Moore
Hojjat Ghaderi
Craig Boutillier
Jurgen Strum
Shaul Markovitch
Constraint Satisfaction Problems (CSPs)

• Chapter 6
  – 6.1: Formalism
  – 6.2: Constraint Propagation
  – 6.3: Backtracking Search for CSP
  – 6.4 is about local search which is a very useful idea but we won’t cover it in class.
• **Uninformed search problems**
  
  – use *problem-specific* state representations and heuristics;
  
  – are generally concerned about determining *paths* from the current state to goal states;
  
  – view states as black boxes with *no internal structures*.

• **Constraint Satisfaction Problems (CSPs)**
  
  – care less about paths and more about final (goal) *configurations*;
  
  – take advantage of a *general state representation*.
  
  – the uniform state representation allows design of *more efficient algorithms*.

• Techniques for solving CSPs have many practical applications in industry.
Constraint Satisfaction Problems (CSPs) – Intuition

• Represent **states** as **vectors of feature values**.\(^1\)
  
  – A set of \( k \) variables (known as **features**).
  
  – **Each variable** has a **domain** of different values.
  
  – A **state** is specified by an **assignment of values** to all variables.
  
  – A **partial state** is specified by an assignment of a value to some of the variables.

• A **goal** is specified as **conditions** on the vector of feature values.

• **Solving a CSP**: find a set of values for the features (**variables**) so that the values **satisfy** the specified conditions (**constraints**).

\[ \begin{align*}
W_1, \ W_2, \ W_3 \\
\text{Dom}[W_1] = \text{Dom}[W_2] &= \{10, 15, 20\} \\
W_1 > W_2 &\quad W_2 = W_3 \\
W_1 > W_3
\end{align*} \]

\(^1\) Feature vectors provide a general state representation that is useful in many other areas of AI, particularly Machine Learning, Reasoning under Uncertainty, and Computer Vision.
Example: Sudoku
Example: Sudoku

• Each **variable** represent a cell.

• **Domain**: a **single value** for cells already filled in; the set \{1, ..., 9\} for empty cells.

• **State**: any completed board given by specifying the value in each cell.

• **Partial State**: some incomplete filling out of the board.

• **Constrains**: The variables that form
  
  - a column must be distinct;
  
  - a row must be distinct;
  
  - a sub-square must be distinct.
A **CSP** consists of

- A set of **variables** $V_1, \ldots, V_n$;
- A (finite) **domain** of possible values $Dom[V_i]$ for each variable $V_i$;
- A set of **constraints** $C_1, \ldots, C_m$.

Each variable $V_i$ can be assigned any value from its domain:

$$V_i = d \quad \text{where} \quad d \in Dom[V_i]$$

Each constraint $C$

- Has a set of variables it operates over, called its **scope**.
  
  **Example:** The scope of $C(V_1, V_2, V_4)$ is $\{V_1, V_2, V_4\}$

- Given an assignment to variables the $C$ returns
  
  **True** if the assignment satisfies the constraint;
  
  **False** if the assignment falsifies the constraint.
Solution to a CSP: An assignment of a value to all of the variables such that every constraint is satisfied.

A CSP is unsatisfiable if no solution exists.
Types of Constraints

- **Unary** Constraints (over one variable)
  
  \[ C(X) : X = 2; \]
  \[ C(Y) : Y > 5 \]

- **Binary** Constraints (over two variables)
  
  \[ C(X, Y) : X + Y < 6 \]

- **Higher-order** constraints: over 3 or more variables.
  
  \[ ALL – Diff(V_1, \ldots, V_n) : V_1 \neq V_2, V_1 \neq V_3, \ldots, V_2 \neq V_1, \ldots, V_n \neq V_1, \ldots, V_n \neq V_{n-1}. \]

\[ ^2 \text{Later, we will see that this collection of binary constraints has less pruning power than } ALL – Diff, \text{ so } ALL – Diff \text{ appears in many CSP problems.} \]
• We can specify the constraints with a table

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V4</th>
<th>C(V1, V2, V4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>False</td>
</tr>
</tbody>
</table>

\[ C(1, 1, 1) = \text{False} \]  
\[ C(2, 1, 1) = \text{True} \]

• Often we can specify the constraint more compactly with an expression.

\[ C(V_1, V_2, V_4): (V_1 = V_2 + V_4) \]
Example: Sudoku

- **Variables**: $V_{11}, V_{12}, \ldots, V_{21}, V_{22}, \ldots, V_{91}, \ldots, V_{99}$

- **Domains**: $Dom[V_{ij}] = \{1, 2, \ldots, 9\}$ for empty cells
  
  $Dom[V_{ij}] = \{k\}$, where $k$ is a fixed value, for filled cells.
Example: Sudoku

• **Constraints:**
  
  - **Row constraints:**
    
    $All - Diff(V_{11}, V_{12}, V_{13}, ..., V_{19})$
    
    $All - Diff(V_{21}, V_{22}, V_{23}, ..., V_{29})$
    
    ...
    
    $All - Diff(V_{91}, V_{12}, V_{13}, ..., V_{99})$

- Sudoku grid:

```
1 2 6 4 3 7 9 5 8
8 9 5 6 2 1 4 7 3
3 7 4 9 8 5 1 2 6
4 5 7 1 9 3 8 6 2
9 8 3 2 4 6 5 1 7
6 1 2 5 7 8 3 9 4
2 6 9 3 1 4 7 8 5
5 4 8 7 6 9 2 3 1
7 3 1 8 5 2 6 4 9
```
Example: Sudoku

- **Constraints:**

  - **Row constraints:**
    
    \[
    \text{All} - \text{Diff}(V_{11}, V_{12}, V_{13}, ..., V_{19})
    \]
    
    \[
    \text{All} - \text{Diff}(V_{21}, V_{22}, V_{23}, ..., V_{29})
    \]
    
    ...\[
    \text{All} - \text{Diff}(V_{91}, V_{12}, V_{13}, ..., V_{99})
    \]

  - **Column Constraints:**
    
    \[
    \text{All} - \text{Diff}(V_{11}, V_{21}, V_{31}, ..., V_{91})
    \]
    
    \[
    \text{All} - \text{Diff}(V_{12}, V_{22}, V_{32}, ..., V_{92})
    \]
    
    ...\[
    \text{All} - \text{Diff}(V_{19}, V_{29}, V_{39}, ..., V_{99})
    \]
Example: Sudoku

- **Constraints:**

  - **Row constraints:**
    \[ \text{All} \rightarrow \text{Diff}(V_{11}, V_{12}, V_{13}, \ldots, V_{19}) \]
    \[ \text{All} \rightarrow \text{Diff}(V_{21}, V_{22}, V_{23}, \ldots, V_{29}) \]
    ...
    \[ \text{All} \rightarrow \text{Diff}(V_{91}, V_{12}, V_{13}, \ldots, V_{99}) \]

  - **Column Constraints:**
    \[ \text{All} \rightarrow \text{Diff}(V_{11}, V_{21}, V_{31}, \ldots, V_{91}) \]
    \[ \text{All} \rightarrow \text{Diff}(V_{12}, V_{22}, V_{32}, \ldots, V_{92}) \]
    ...
    \[ \text{All} \rightarrow \text{Diff}(V_{19}, V_{29}, V_{39}, \ldots, V_{99}) \]

  - **Sub-Square Constraints:**
    \[ \text{All} \rightarrow \text{Diff}(V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33}), \]
    ...
    \[ \text{All} \rightarrow \text{Diff}(V_{77}, V_{78}, V_{79}, \ldots, V_{97}, V_{98}, V_{99}) \]
**Problem Statement:** Place $N$ Queens on an $N \times N$ chess board so that no Queen can attack any other Queen.

$Q_1 = 1$

$Q_2 = 18$

$Q_3 = 21$
Example: N-Queens

Problem formulation:

- **Variables:** \( N \) variables, each representing a queen

- **Domains:** \( N^2 \) values for each variable, representing the position of a queen on the chessboard

Number of possible configurations: \( (N^2)^N \)

For 8-queens:

\[ (64)^8 = 281,474,976,710,656 \]

Possible configurations
Is there a better way to represent the N-queens problem? We know we cannot place two queens in a single row.

**Problem Statement:** Place $N$ Queens on an $N \times N$ chess board so that no Queen can attack any other Queen.

**Better Formulation:**

- **Variables:**
  - $N$ variables, one for each queen on each row
  - $Q_i$: $i$-th queen on row $i$

- **Domains:**
  - Value of $Q_i$ is the column the queen on row $i$ is placed.
  - Possible values: $\{1, 2, \ldots, N\}$

**Number of all possible configurations:** $N^N$

For 8-queens: $8^8 = 16,777,216$ configurations
Example: N-Queens

$Q_1 = 1 \quad Q_2 = 7 \quad Q_4 = 8$
Example: N-Queens

Constraints:

- Cannot put two Queens in same column: For all $i \neq j$, $Q_i \neq Q_j$
  
  or $\text{All-Diff}(Q_1, Q_2, \ldots, Q_N)$

- Diagonal constraints: for all $i \neq j$,
  
  $|Q_i - Q_j| \neq |i - j|$
Example: N-Queens
A CSP could be formulated as a search problem:

- **Initial State**: Empty assignment.
- **Successor Function**: Assigned values to an unassigned variable.
- **Goal Test**:
  1. The assignment is complete
  2. No constraints is violated.
CSP Backtracking Search - Intuition

CSPs do NOT require finding a path (to a goal). They only need the configuration of the goal state. CSPs are best solved by a specialized version search called Backtracking Search.

Key Intuitions:

• Searching through the space of partial assignments, rather than paths.

• Decide on a suitable value for one variable at a time. Order in which we assign the variables does not matter.

• If a constraint is falsified during the process of partial assignment, immediately reject the current partial assignment.
CSP Search Tree:

- **Root**: Empty Assignment.
- **Children** of a node: all possible value assignments for a particular unassigned variable.
- The tree stops descending if an assignment violates a constraint.
- **Goal Node**:  
  1. The assignment is complete  
  2. No constraints is violated.
Draw the CSP search tree for 4-Queens.
We will apply a **recursive** implementation:

- If all variables are set, print the solution and **terminate**.
- Otherwise:
  - Pick an unassigned variable $V$ and **assign** it a value.
  - **Test** the constraints corresponding with $V$ and all other variables of them are assigned.
  - If a constraint is **unsatisfied**, return (**backtrack**).
  - Otherwise, go one lever deeper by invoking a **recursive call**.
def BT(Level):
1. if all Variables assigned
2. PRINT Value of each Variable
3. EXIT or RETURN # EXIT for only one solution
   # RETURN for more solutions
4. V := PickUnassignedVariable()
5. Assigned[V] := TRUE
6. for d := each member of Domain(V) # the domain values of V
7. Value[V] := d
8. ConstraintsOK := TRUE
9. for each constraint C such that (i) V is a variable of C and
   (ii) all other variables of C are assigned:
10. if C is not satisfied by the set of current assignments:
11. ConstraintsOK := FALSE
12. if ConstraintsOk == TRUE:
13. BT(Level+1)
14. Assigned[V] := FALSE # UNDO as we have tried all of V’s values
15. RETURN