Overview

• Constraint satisfaction is a powerful problem-solving paradigm
  – Problem: set of variables to which we must assign values satisfying problem-specific constraints
  – Constraint programming, constraint satisfaction problems (CSPs), constraint logic programming...

• Algorithms for CSPs
  – Backtracking (systematic search)
  – Constraint propagation (k-consistency)
  – Variable and value ordering heuristics
  – Backjumping and dependency-directed backtracking
Motivating example: 8 Queens

Place 8 queens on a chess board such that none is attacking another.

Generate-and-test, with no redundancies → “only” $8^8$ combinations

$8^{*8}$ is 16,777,216
Motivating example: 8-Queens

After placing these two queens, it’s trivial to mark the squares we can no longer use.
What more do we need for 8 queens?

• Not just a successor function and goal test
• But also
  – a means to propagate constraints imposed by one queen on others
  – an early failure test
→ Explicit representation of constraints and constraint manipulation algorithms
Informal definition of CSP

• CSP (Constraint Satisfaction Problem), given
  (1) finite set of variables
  (2) each with domain of possible values (often finite)
  (3) set of constraints limiting values variables can take

• Solution: assignment of a value to each variable such that all constraints are satisfied

• Tasks: decide if a solution exists, find a solution, find all solutions, find “best solution” according to some metric (objective function)
Example: 8-Queens Problem

• Eight variables $X_i$, $i = 1..8$ where $X_i$ is the row number of queen in column $i$
• Domain for each variable $\{1,2,\ldots,8\}$
• Constraints are of the forms:
  – No queens on same row
    $X_i = k \implies X_j \neq k \text{ for } j = 1..8, j \neq i$
  – No queens on same diagonal
    $X_i = k_i, X_j = k_j \implies |i-j| \neq |k_i - k_j| \text{ for } j = 1..8, j \neq i$
Example: Task Scheduling

Examples of scheduling constraints:
• T1 must be done during T3
• T2 must be achieved before T1 starts
• T2 must overlap with T3
• T4 must start after T1 is complete
Example: Map coloring

Color this map using three colors (red, green, blue) such that no two adjacent regions have the same color.

Diagram:

```
  E
 /|
D  A
/ |
C  B
```
Map coloring

• Variables: A, B, C, D, E all of domain RGB
• Domains: RGB = {red, green, blue}
• Constraints: A ≠ B, A ≠ C, A ≠ E, A ≠ D, B ≠ C, C ≠ D, D ≠ E
• A solution: A = red, B = green, C = blue, D = green, E = blue
Brute Force methods

• Finding a solution by a brute force search is easy
  – Generate and test is a weak method
  – Just generate potential combinations and test each

• Potentially very inefficient
  – With n variables where each can have one of 3 values, there are $3^n$ possible solutions to check

• There are ~190 countries in the world, which we can color using four colors

• $4^{190}$ is a big number!
Example: SATisfiability

• Given a set of logic propositions containing variables, find an assignment of the variables to \{false, true\} that satisfies them

• For example, the two clauses:
  \[ \neg (A \lor B \lor \neg C) \land (\neg A \lor D) \]
  \[\neg (\text{equivalent to } (C \rightarrow A) \lor (B \land D \rightarrow A))\]
  are satisfied by
  \[ A = \text{false}, \ B = \text{true}, \ C = \text{false}, \ D = \text{false} \]

• **Satisfiability** is known to be **NP-complete**, so in worst case, solving CSP problems requires exponential time
Real-world problems

CSPs are a good match for many practical problems that arise in the real world

- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision

- Graph layout
- Network management
- Natural language processing
- Molecular biology / genomics
- VLSI design
Definition of a constraint network (CN)

A constraint network (CN) consists of

• Set of variables $X = \{x_1, x_2, \ldots, x_n\}$
  – with associate domains $\{d_1, d_2, \ldots, d_n\}$
  – domains are typically finite

• Set of constraints $\{c_1, c_2, \ldots, c_m\}$ where
  – each defines a predicate that is a relation over a particular subset of variables ($X$)
  – e.g., $C_i$ involves variables $\{X_{i1}, X_{i2}, \ldots, X_{ik}\}$ and defines the relation $R_i \subseteq D_{i1} \times D_{i2} \times \ldots \times D_{ik}$
Running example: coloring Australia

- Seven variables: \{WA, NT, SA, Q, NSW, V, T\}
- Each variable has same domain: \{red, green, blue\}
- No two adjacent variables can have same value:
  WA ≠ NT, WA ≠ SA, NT ≠ SA, NT ≠ Q, SA ≠ Q, SA ≠ NSW, SA ≠ V, Q ≠ NSW, NSW ≠ V
Unary & binary constraints most common

- Binary constraints

- Two variables are adjacent or neighbors if connected by an edge or an arc
- Possible to rewrite problems with higher-order constraints as ones with just binary constraints
Formal definition of a CN

• Instantiations
  – An instantiation of a subset of variables $S$ is an assignment of a value (in its domain) to each variable in $S$
  – An instantiation is legal iff it violates no constraints

• A solution is a legal instantiation of all variables in the network
Typical tasks for CSP

• Solution related tasks:
  – Does a solution exist?
  – Find one solution
  – Find all solutions
  – Given a metric on solutions, find best one
  – Given a partial instantiation, do any of above

• Transform the CN into an equivalent CN that is easier to solve
Binary CSP

• A binary CSP is a CSP where all constraints are binary or unary
• Any non-binary CSP can be converted into a binary CSP by introducing additional variables
• A binary CSP can be represented as a constraint graph, with a node for each variable and an arc between two nodes iff there’s a constraint involving them
  – Unary constraints appear as self-referential arcs
Running example: coloring Australia

- Seven variables: \{WA, NT, SA, Q, NSW, V, T\}
- Each variable has same domain: \{red, green, blue\}
- No two adjacent variables can have same value:
  
  WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q, SA \neq NSW, 
  SA \neq V, Q \neq NSW, NSW \neq V
A running example: coloring Australia

• Solutions: complete & consistent assignments
• Here is one of several solutions
• For generality, constraints can be expressed as relations, e.g., describe $\text{WA} \neq \text{NT}$ as 
  \{(\text{red},\text{green}), (\text{red},\text{blue}), (\text{green},\text{red}), (\text{green},\text{blue}), (\text{blue},\text{red}),(\text{blue},\text{green})\}$
Backtracking example
Backtracking example
Backtracking example
Backtracking example
Basic Backtracking Algorithm

CSP-BACKTRACKING(PartialAssignment a)
– If a is complete then return a
– X ← select an unassigned variable
– D ← select an ordering for the domain of X
– For each value v in D do
  If v is consistent with a then
    – Add (X= v) to a
    – result ← CSP-BACKTRACKING(a)
    – If result ≠ failure then return result
    – Remove (X= v) from a
  – Return failure

Start with CSP-BACKTRACKING({})

Note: this is depth first search; can solve n-queens problems for n ~ 25
Problems with backtracking

• Thrashing: keep repeating the same failed variable assignments
• Things that can help avoid this:
  – Consistency checking
  – Intelligent backtracking schemes
• Inefficiency: can explore areas of the search space that aren’t likely to succeed
  – Variable ordering can help
Improving backtracking efficiency

Here are some standard techniques to improve the efficiency of backtracking

– Can we detect inevitable failure early?
– Which variable should be assigned next?
– In what order should its values be tried?
Forward Checking

After variable $X$ is assigned to value $v$, examine each unassigned variable $Y$ connected to $X$ by a constraint and delete values from $Y$’s domain inconsistent with $v$.

Using forward checking and backward checking roughly doubles the size of N-queens problems that can be practically solved.
Forward checking

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

- Forward checking propagates info. from assigned to unassigned variables, but doesn't provide early detection for all failures.
- NT and SA cannot both be blue!
Definition: Arc consistency

- A constraint $C_{xy}$ is **arc consistent** wrt $x$ if for each value $v$ of $x$ there is an allowed value of $y$
- Similarly define $C_{xy}$ as arc consistent wrt $y$
- A binary CSP is arc consistent iff every constraint $C_{xy}$ is arc consistent wrt $x$ as well as $y$
- When a CSP is not arc consistent, we can make it arc consistent, e.g., by using AC3
  – Also called “enforcing arc consistency”
Arc Consistency Example 1

• Domains
  – $D_x = \{1, 2, 3\}$
  – $D_y = \{3, 4, 5, 6\}$

• Constraint
  – Note: for finite domains, we can represent a constraint as an enumeration of legal values
  – $C_{xy} = \{(1,3), (1,5), (3,3), (3,6)\}$

• $C_{xy}$ isn’t arc consistent wrt $x$ or $y$. By enforcing arc consistency, we get reduced domains
  – $D'_x = \{1, 3\}$
  – $D'_y = \{3, 5, 6\}$
Arc Consistency Example 2

- **Domains**
  - $D_x = \{1, 2, 3\}$
  - $D_y = \{1, 2, 3\}$

- **Constraint**
  - $C_{xy} = \lambda v1, v2: v1 < v2$

- $C_{xy}$ is not arc consistent wrt $x$, neither wrt $y$. By enforcing arc consistency, we get reduced domains
  - $D'_x = \{1, 2\}$
  - $D'_y = \{2, 3\}$
Arc consistency

• Simplest form of propagation makes each arc consistent
• $X \rightarrow Y$ is consistent iff for every value $x$ of $X$ there is some allowed $y$
Arc consistency

• Simplest form of propagation makes each arc consistent

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

If X loses a value, neighbors of X need to be rechecked
Arc consistency

- Arc consistency detects failure earlier than simple forward checking
- WA=red and Q=green is quickly recognized as a deadend, i.e. an impossible partial instantiation
- The arc consistency algorithm can be run as a preprocessor or after each assignment
General CP for Binary Constraints

Algorithm AC3

contradiction $\leftarrow$ false

$Q \leftarrow$ stack of all variables

while $Q$ is not empty and not contradiction do

$X \leftarrow$ UNSTACK($Q$)

For every variable $Y$ adjacent to $X$ do

If REMOVE-ARC-INCONSISTENCIES($X,Y$)

If domain($Y$) is non-empty then STACK($Y,Q$)

else return false
Complexity of AC3

- $e =$ number of constraints (edges)
- $d =$ number of values per variable
- Each variable is inserted in queue up to $d$ times
- REMOVE-ARC-INCONSISTENCY takes $O(d^2)$ time
- CP takes $O(ed^3)$ time
Improving backtracking efficiency

• Some standard techniques to improve the efficiency of backtracking
  – Can we detect inevitable failure early?
  – Which variable should be assigned next?
  – In what order should its values be tried?

• Combining constraint propagation with these heuristics makes 1000-queen puzzles feasible
Most constrained variable

• Most constrained variable: choose the variable with the fewest legal values
• a.k.a. minimum remaining values (MRV) heuristic
• After assigning a value to WA, both NT and SA have only two values in their domains – choose one of them rather than Q, NSW, V or T
Most constraining variable

• Tie-breaker among most constrained variables
• Choose variable involved in largest # of constraints on remaining variables

• After assigning SA to be blue, WA, NT, Q, NSW and V all have just two values left.
• WA and V have only one constraint on remaining variables and T none, so choose one of NT, Q & NSW
Least constraining value

• Given a variable, choose least constraining value:
  – the one that rules out the fewest values in the remaining variables

• Combining these heuristics makes 1000 queens feasible

• What’s an intuitive explanation for this?
Is AC3 Alone Sufficient?

Consider the four queens problem
Solving a CSP still requires search

• Search:
  – can find good solutions, but must examine non-solutions along the way

• Constraint Propagation:
  – can rule out non-solutions, but this is not the same as finding solutions

• Interweave constraint propagation & search:
  – perform constraint propagation at each search step
4-Queens Problem

X1 \{1,2,3,4\}

X2 \{1,2,3,4\}

X3 \{1,2,3,4\}

X4 \{1,2,3,4\}
4-Queens Problem

\[ X_1 \{1,2,3,4\} \]
\[ X_2 \{ , ,3,4\} \]
\[ X_3 \{ ,2, ,4\} \]
\[ X_4 \{ ,2,3, } \]
4-Queens Problem

X2=3 eliminates \{ X3=2, X3=3, X3=4 \}  
⇒ inconsistent!
**4-Queens Problem**

\[ X_2 = 4 \implies X_3 = 2, \text{ which eliminates } \{ X_4 = 2, X_4 = 3 \} \implies \text{inconsistent!} \]
4-Queens Problem

X1 can’t be 1, let’s try 2
4-Queens Problem

Can we eliminate any other values?
4-Queens Problem

\[
\begin{array}{cccc}
 X1 & \{2,3,4\} & & \\
 X2 & \{,4\} & & \\
 X3 & \{1,,\} & & \\
 X4 & \{1,3,4\} & & \\
\end{array}
\]
Arc constancy eliminates x₃=3 because it’s not consistent with X₂’s remaining values.
There is only one solution with $X_1=2$
Sudoku Example

How can we set this up as a CSP?
Sudoku

- Digit placement puzzle on 9x9 grid with unique answer
- Given an initial partially filled grid, fill remaining squares with a digit between 1 and 9
- Each column, row, and nine $3 \times 3$ sub-grids must contain all nine digits

Some initial configurations are easy to solve and some very difficult
def sudoku(initValue):
    p = Problem()
    # Define a variable for each cell: 11,12,13...21,22,23...98,99
    for i in range(1, 10):
        p.addVariables(range(i*10+1, i*10+10), range(1, 10))
    # Each row has different values
    for i in range(1, 10):
        p.addConstraint(AllDifferentConstraint(), range(i*10+1, i*10+10))
    # Each column has different values
    for i in range(1, 10):
        p.addConstraint(AllDifferentConstraint(), range(10+i, 100+i, 10))
    # Each 3x3 box has different values
    p.addConstraint(AllDifferentConstraint(), [11,12,13,21,22,23,31,32,33])
    p.addConstraint(AllDifferentConstraint(), [41,42,43,51,52,53,61,62,63])
    p.addConstraint(AllDifferentConstraint(), [71,72,73,81,82,83,91,92,93])
    p.addConstraint(AllDifferentConstraint(), [14,15,16,24,25,26,34,35,36])
    p.addConstraint(AllDifferentConstraint(), [44,45,46,54,55,56,64,65,66])
    p.addConstraint(AllDifferentConstraint(), [74,75,76,84,85,86,94,95,96])
    p.addConstraint(AllDifferentConstraint(), [17,18,19,27,28,29,37,38,39])
    p.addConstraint(AllDifferentConstraint(), [47,48,49,57,58,59,67,68,69])
    p.addConstraint(AllDifferentConstraint(), [77,78,79,87,88,89,97,98,99])
    # add unary constraints for cells with initial non-zero values
    for i in range(1, 10):
        for j in range(1, 10):
            value = initValue[i-1][j-1]
            if value:
                p.addConstraint(lambda var, val=value: var == val, (i*10+j,))
    return p.getSolution()

# Sample problems
easy = [
    [0,9,0,7,0,0,8,6,0],
    [0,3,1,0,5,0,2,0,0],
    [8,0,6,0,0,0,0,0,0],
    [0,0,7,0,5,0,0,0,6],
    [0,0,0,3,0,7,0,0,0],
    [5,0,0,1,0,7,0,0,0],
    [0,0,0,0,0,1,0,9,0],
    [0,2,0,6,0,0,5,0,0],
    [0,5,4,0,0,8,0,7,0]]

hard = [
    [0,0,3,0,0,0,4,0,0],
    [0,0,0,0,7,0,0,0,0],
    [5,0,0,4,0,6,0,0,2],
    [0,0,4,0,0,0,8,0,0],
    [0,9,0,0,3,0,0,2,0],
    [0,0,7,0,0,5,0,0,0],
    [6,0,0,5,0,2,0,0,1],
    [0,0,0,9,0,0,0,0,0],
    [0,0,9,0,0,3,0,0,0]]

very_hard = [
    [0,0,0,0,0,0,0,0,0],
    [0,0,9,0,6,0,3,0,0],
    [0,7,0,3,0,4,0,9,0],
    [0,0,7,2,0,8,6,0,0],
    [0,4,0,0,0,0,7,0,0],
    [0,0,2,1,0,6,5,0,0],
    [0,1,0,9,0,5,0,4,0],
    [0,0,8,0,2,0,7,0,0],
    [0,0,0,0,0,0,0,0,0]]
Local search for constraint problems

• Remember local search?
• There’s a version of local search for CSP problems
• Basic idea:
  – generate a random “solution”
  – Use metric of “number of conflicts”
  – Modifying solution by reassigning one variable at a time to decrease metric until solution found or no modification improves it
• Has all features and problems of local search
Min Conflict Example

- **States:** 4 Queens, 1 per column
- **Operators:** Move a queen in its column
- **Goal test:** No attacks
- **Evaluation metric:** Total number of attacks

How many conflicts does each state have?
Basic Local Search Algorithm

Assign a domain value $d_i$ to each variable $v_i$
while no solution & not stuck & not timed out:

- $\text{bestCost} \leftarrow \infty$; $\text{bestList} \leftarrow \emptyset$;
- for each variable $v_i$ | Cost(Value($v_i$) > 0
  for each domain value $d_i$ of $v_i$
    if Cost($d_i$) < $\text{bestCost}$
      $\text{bestCost} \leftarrow \text{Cost}(d_i)$; $\text{bestList} \leftarrow d_i$;
    else if Cost($d_i$) = $\text{bestCost}$
      $\text{bestList} \leftarrow \text{bestList} \cup d_i$
Take a randomly selected move from bestList
Eight Queens using Backtracking

Undo move for Queen 7 and so on...
Eight Queens using Local Search

Answer Found
Backtracking Performance

![Graph showing the relationship between the number of queens and time in seconds. The x-axis represents the number of queens, ranging from 0 to 32, and the y-axis represents time in seconds, ranging from 0 to 5000.]
Min Conflict Performance

• Performance depends on quality and informativeness of initial assignment; inversely related to distance to solution
• Min Conflict often has astounding performance
• For example, can solve arbitrary size (i.e., millions) N-Queens problems in constant time
• Appears to hold for arbitrary CSPs with the caveat...
Min Conflict Performance

Except in a certain critical range of the ratio constraints to variables.
Famous example: labeling line drawings

- **Waltz** labeling algorithm, earliest AI CSP application (1972)
  - Convex interior lines are labeled as +
  - Concave interior lines are labeled as –
  - Boundary lines are labeled as
- There are 208 labeling (most of which are impossible)
- Here are the 18 legal labeling:
Labeling line drawings II

• Here are some illegal labelings:
Labeling line drawings

Waltz labeling algorithm: propagate constraints repeatedly until a solution is found

A solution for one labeling problem

A labeling problem with no solution
Shadows add complexity

CSP was able to label scenes where some of the lines were caused by shadows.
Intelligent backtracking

- **Backjumping**: if $V_j$ fails, jump back to the variable $V_i$ with greatest $i$ such that the constraint $(V_i, V_j)$ fails (i.e., most recently instantiated variable in conflict with $V_i$)
- **Backchecking**: keep track of incompatible value assignments computed during backjumping
- **Backmarking**: keep track of which variables led to the incompatible variable assignments for improved backchecking
Challenges for constraint reasoning

• What if not all constraints can be satisfied?
  – Hard vs. soft constraints vs. preferences
  – Degree of constraint satisfaction
  – Cost of violating constraints

• What if constraints are of different forms?
  – Symbolic constraints
  – Numerical constraints [constraint solving]
  – Temporal constraints
  – Mixed constraints
Challenges for constraint reasoning

• What if constraints are represented intentionally?
  – Cost of evaluating constraints (time, memory, resources)

• What if constraints, variables, and/or values change over time?
  – Dynamic constraint networks
  – Temporal constraint networks
  – Constraint repair

• What if multiple agents or systems are involved in constraint satisfaction?
  – Distributed CSPs
  – Localization techniques