Artificial Intelligence
V05: Constraint satisfaction problems

Introduction to CSPs
CSP solving
Solving CSPs in practice

Based on material by Stuart Russell, UC Berkeley
Educational objectives

• **Remember** what makes CSP solving **more powerful** than pure search techniques

• **Explain** how CSPs are solved on the **algorithmic level** by **backtracking** using the **MRV / degree- / least constraining value heuristics** and **forward checking / constrained propagation**

• **Formulate** a suitable problem as a **CSP**

“In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.”

➔ Reading: AIMA, ch. 6
1. INTRODUCTION TO CSPS
Constraint satisfaction problems (CSPs)

Standard search problem
• State is a “black box” – any data structure that supports Goal Test, Eval, Successor

CSP
• State is defined by variables $X_i$ with values from domain $D_i$
• Goal Test is a set of constraints: allowable combinations of values for subsets of variables

→ Simple example of a formal representation language
→ Allows useful general-purpose algorithms with more power than standard search
Example: Map-coloring

Variables: $WA$, $NT$, $Q$, $NSW$, $V$, $SA$, $T$

Domains: $D_i = \{\text{red, green, blue}\}$

Constraints: adjacent regions must have different colors
- e.g. $WA \neq NT$ (if language allows this; otherwise $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), ...\}$)

Solutions: assignments satisfying all constraints
- e.g. $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}$

Binary CSPs (each constraint relates at most two variables) have a constraint graph. General-purpose CSP algorithms use the graph structure to speed up search: E.g., $T$ is an independent subproblem!
Varieties of CSPs

Discrete variables
• Finite domains of size $d \rightarrow O(d^n)$ complete assignments ($n$ is number of variables)
• Other finite domains (integers, strings, etc.)
  • e.g., job scheduling: variables are days (or integer-minutes) for each job
  • need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
  • linear constraints solvable, nonlinear undecidable

Continuous variables
• e.g., precise start/end times for Hubble Telescope observations
• linear constraints solvable in polynomial time by linear programming methods

Varieties of constraints
• **Unary** constraints: involve a single variable, e.g. $SA \neq green$
• **Binary** constraints involve variable pairs, e.g., $SA \neq WA$ (all constraints can be made binary)
• **Higher-order** constraints involve 3 or more variables, e.g. column constraints in Sudoku
• **Preferences** (soft) constraints, e.g. $red IS\_BETTER\_THAN green$
  \rightarrow often representable by a cost for each assignment: constrained optimization problems (COP)
Examples

Car assembly (job scheduling, simplified)
- Variables: \( Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect \)
- Domains: \( D_i = \{1,2,3, \ldots, 27\} \) (start time of tasks as integer, due to an overall runtime of 30 minutes)
- Constraints:
  (precedence constraints among tasks)
  - \( Axle_F + 10 \leq Wheel_{RF} \)
  - \( Axle_F + 10 \leq Wheel_{LF} \)
  - \( Axle_B + 10 \leq Wheel_{RB} \)
  - \( Axle_B + 10 \leq Wheel_{LB} \)
  - \( Axle_F + 10 \leq Axle_B \text{ or } Axle_B + 10 \leq Axle_F \)
  - \( \ldots \)

Cryptarithmetic (which letter represents which digit?)
- Variables: \( F, T, U, W, R, O, C_1, C_2, C_3 \)
- Domains: \( \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)
- Constraints:
  - all\( \text{diff}(F,T,U,W,R,O) \)
  - \( O + O = R + 10C_1 \)
  - \( C_1 + W + W = U + 10C_2 \)
  - \( C_2 + T + T = O + 10C_3 \)
  - \( C_3 = F \)
Real-world CSPs

- **Assignment** problems
e.g., who teaches what class
- **Timetabling** problems
e.g., which class is offered when and where?
- **Optimization** with spreadsheets
e.g., debugging (Abreu, Riboira & Wotawa, 2012)
- Other **scheduling** tasks
e.g., in transportation or factory workflow
- Other **layout** tasks
e.g., floor planning or hardware configuration

→ Notice that many real-world problems involve real-valued variables
Exercise: Formulating Sudoku as a CSP

see also P03

Sudoku puzzles are played on a 9x9 board and enjoyed by millions of people daily. The goal is to fill in each cell with a single digit, subject to several constraints:

• Each digit must be present in each row exactly once
• Each digit must be present in each column exactly once
• Each digit must be present in each box exactly once
  (the 9x9 board consists of 9 non-overlapping 3x3 boxes
  see thicker lines below)
• Each digit must be consistent with any digit already placed on the original board by the riddle issuer

Formulate the Sudoku riddle below as a CSP using pen & paper (i.e., decide on variables, domains and constraints)
2. CSP SOLVING
Standard search formulation
Seriously flawed, thus incremental

Let's start with the straightforward, dumb approach, then fix it
• **States** are defined by the **values assigned so far**
  • Initial state: the empty assignment \{\}
  • Successor function: assign a value to an unassigned variable without conflict with current assignment
    \(\rightarrow\) fail if no legal assignment (not fixable!)
  • Goal test: the current assignment is complete
• CSPs all have a common structure
  \(\rightarrow\) This is the same for all CSPs, **no domain-specific adaptations** (transition models etc.) needed! 😊

• Every solution appears at depth \(n\) (for \(n\) variables)
  \(\rightarrow\) use **depth-first search**
• Path is irrelevant, so can also use **complete-state formulation** (as with local search)
  \(\rightarrow\) i.e., *evolve one state* instead of creating new ones
• Branching factor \(b = (n - l)d\) at depth \(l\)
  \(\rightarrow\) hence \(n! d^n\) leaves! ☹️ ☹️ ☹️
Backtracking search

First improvement
• Variable assignments are commutative
e.g. \([WA = red, then NT = green]\) same as \([NT = green, then WA = red]\)
  ➔ Only need to consider assignments to a single variable at each node
  ➔ \(b = d\), thus there are \(d^n\) leaves

Backtracking search
• Using depth-first search with single-variable assignments for CSPs is called backtracking search
• It is the basic uninformed algorithm for CSPs
  ➔ Can solve \(n\)-queens for \(n = 25\)

Remember V04: simple heuristic solves 1’000’000-queens…
Backtracking search
Algorithm & suggested improvements

function Backtracking-Search(csp) returns solution/failure
   return Backtrack({}, csp)

function Backtrack(assignment, csp) returns solution/failure
   if assignment is complete then return assignment
   var ← Select-Unassigned-Variable(csp)
   for each value in Order-Domain-Values(var, assignment, csp) do
      if value is consistent with assignment then
         add {var = value} to assignment
         inferences ← Inference(csp, var, value)  #optional
         if inferences ≠ failure then  #optional
            add inferences to assignment
            result ← Backtrack(assignment, csp)
            if result ≠ failure then return result
            else remove {var = value} from assignment
         end if
      end if
   end for
   return failure

General-purpose methods can give huge gains in speed:
• **Which variable** should be assigned next?
• **In what order** should its **values** be tried?
• Can we **detect** inevitable **failure early**?
• Can we **take advantage** of **problem structure**?

⇒ can be achieved by implementing the **bold/italic** functions above
Which variable should be assigned next?

Ideas for Select-Unassigned-Variable (csp)

**Minimum remaining values (MRV):**
- Choose the variable with the **fewest legal values**
  \( \rightarrow \) **failing fast** prunes large portions of the tree
- Can work up to 1’000 times better than picking just the next (or a random) unassigned variable (very problem dependent)

**Degree heuristic**
- Choose the variable that adds **most constraints on remaining** variables
  \( \rightarrow \) In practice: Used as **tie-breaker** among MRV variables

---

All have equal opportunities (3)

Neighbors are restricted now (2)

SA can only be blue now

SA imposes constraints on all 5 neighbors

Several equal options from here (e.g., NT, Q, NSW have degree 2)
In what order should its values be tried?

Ideas for Order-Domain-Values (var, assignment, csp)

Least constraining value

- Given var, choose the value that rules out the fewest values in the remaining var
  → Combining this with the previous 2 heuristics makes 1’000-queens feasible (instead 25)
Can we detect inevitable failure early?

Ideas for Inference \((csp, var, value)\)

Forward checking

- Idea: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
Can we detect inevitable failure early?
Ideas for Inference(*csp*, *var*, *value*)

Forward checking
- Idea: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
Can we detect inevitable failure early?

Ideas for *Inference*(*csp, var, value*)

**Forward checking**
- Idea: Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
Can we detect inevitable failure early?
Ideas for Inference ($csp, \ var, \ value$)

Forward checking
• Idea: Keep track of remaining legal values for unassigned variables
  ➔ Terminate search when any variable has no legal values
Can we detect inevitable failure early? (contd.)

Ideas for Inference ($csp$, $var$, $value$)

Constraint propagation
- **Forward checking** propagates information from assigned variables only to immediate neighbours (i.e., fails to do so recursively after a change in some domain)
  - e.g., $NT$ and $SA$ cannot both be blue!

> Constraint propagation would repeatedly enforce constraints locally
Can we detect inevitable failure early? (contd.)

Ideas for Inference($csp$, $var$, $value$)

Arc
Can we detect inevitable failure early? (contd.)

Ideas for Inference($csp$, $var$, $value$)

Arc
Can we detect inevitable failure early? (contd.)

Ideas for Inference ($csp$, $var$, $value$)

Arc

- If $X$ loses a value, neighbors of $X$ need to be rechecked (→ see AC-3 algorithm in appendix)
Can we detect inevitable failure early? (contd.)

Ideas for Inference(*csp*, *var*, *value*)

Arc

- If $X$ loses a value, neighbors of $X$ need to be rechecked (→ see AC-3 algorithm in appendix)
Can we detect inevitable failure early? (contd.)

Ideas for Inference(csp, var, value)

Arc

- If $X$ loses a value, neighbors of $X$ need to be rechecked (→ see AC-3 algorithm in appendix)
Backtracking search
Revisiting suggested improvements

function Backtracking-Search(csp) returns solution/failure
    return Backtrack({}, csp)

function Backtrack(assignment, csp) returns solution/failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment then
            add \{var = value\} to assignment
            inferences ← Inference(csp, var, value) #optional
            if inferences ≠ failure then #optional
                add inferences to assignment
                result ← Backtrack(assignment, csp)
                if result ≠ failure then return result
            else remove \{var = value\} from assignment
    return failure

General-purpose methods can give huge gains in speed:
• **Which variable** next? MRV (fewest legal values), degree heuristic (most constraints on rest) on tie
• **What value first?** Least constraining value
• **How detect failure early?** Constraint propagation via arc consistency

• Can we **take advantage** of **problem structure**? \(\rightarrow\) next
Can we take advantage of problem structure?  
Exploiting structure in the constraint graph

Example
- Tasmania and mainland are independent **subproblems**, identifiable as **connected components** of constraint graph  
  ➔ can be solved individually, and solution combined

- Suppose each subproblem has $c$ variables (out of $n$ total)  
  ➔ Worst-case solution cost is $n/c \cdot d^c$ (linear in $n$)

- This is a **dramatic improvement**!
  - E.g., $n = 80$, $d = 2$, $c = 20$:  
    ➔ $2^{80} = 4$ billion years (at 10 million nodes/second)  
    ➔ $4 \cdot 2^{20} = 0.4$ seconds (at 10 million nodes/second)
Can we take advantage of problem structure?
Exploiting structure in the constraint graph (contd.)

Tree-structured CSPs
- A (constraint) graph is a **tree** if any 2 variables are **connected by only 1 path** (i.e., no loops)
- **Theorem:** If the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$
  - Also applies to logical and probabilistic reasoning
  - Important example of the relation between **syntactic restrictions** and the **complexity of reasoning**

Algorithm for tree-structured CSPs
- Do a **topological sort**: Choose a variable as root, then **order variables** from root to leaves such that every node's parent precedes it in the ordering
- Create **directed arc-consistency** by: For $j$ from $n$ down to 2, make $(\text{Parent}(X_j), X_j)$ arc consistent
- For $j$ from 1 to $n$, **assign** $X_j$ consistently with $\text{Parent}(X_j)$
3. SOLVING CSPS IN PRACTICE
Exploiting non-optimal structure

Nearly tree-structured CSPs

- Many real-world CSPs can be converted to tree-structured problems
  \( \Rightarrow \) then solved by divide & conquer
- …by choosing a cycle cutset: a set of variables that if removed make the graph a tree

\[ \text{Runtime is } O(d^c \cdot (n - c)d^2) \text{ (linear in } n) \]

...and subsequent cutset conditioning: instantiate (in all ways) the variables in the cutset, then prune choices from remaining variables in the tree

\( \Rightarrow \) Very fast for small cutset size \( c \): Runtime is \( O(d^c \cdot (n - c)d^2) \) (linear in \( n \))
Other advice

• **Exploiting structure in the values by breaking symmetry** reduces search space up to $d!$ (e.g., we have to give $WA$, $NT$, $SA$ 3 different colors, but have $3!$ options to do so
  → can be reduced by adding a symmetry-breaking constraint like $NT < SA < WA$)

• **Local search** (→ see V04) is very effective for CSPs
  → **Min-conflicts** heuristic very useful: start with random full assignment, subsequently change the variable that minimizes remaining conflicts
  → E.g., hill climbing search with min-conflicts solves $n$-queens in constant time with high probability (even for $n = 10'000'000$)

• **Constraint learning** (→ see appendix) is one of the most important techniques in modern CSP solvers
  (together with backtracking search, the MRV / degree- / least constraining value heuristics, and forward checking / arc consistency)

• **Trade-off** between the cost of enforcing consistency and the reduction in search time
  (some researchers favor pure forward checking, some full arc consistency after each assignment
  → full arc consistency pays off for harder CSPs)

• Comparing CSP algorithms is done empirically (no algorithm dominates on all CSPs)
Where’s the intelligence?
Man vs. machine

• If classical search is brute force…

• …CSP solving enhances it using the following powerful ingredients:
  • General-purpose heuristics
    (MRV etc. ➔ not problem- or domain specific!)
  • Inference over constraints
    (constraint propagation ➔ allows e.g. for intelligent backjumping)
  • Exploiting structure in the problem definition to vastly prune the search space
    (e.g. symmetric values, tree-like constraint graph ➔ implements a general divide & conquer approach)

• CSP solving thus can reduce the time complexity of some problems from exponential to linear, by acting more “clever”

• Human intelligence goes into stating the task as a CSP
Review

- 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**

- **Backtracking** = **depth-first search** with one variable assigned per node
  - **Variable ordering** and **value selection heuristics** help significantly
  - **Forward checking prevents** assignments that guarantee **later failure**
  - **Constraint propagation** (e.g., arc consistency) does **additional** work to constrain values and detect **inconsistencies**

- The CSP representation allows **analysis of problem structure**
  - **Tree-structured** CSPs can be solved in **linear time**
  - **Iterative min-conflicts** is usually **effective in practice**

- **Methods** can **handle** problems with **up to 100’000 variables**, and up to **1’000’000 constraints** in practice
APPENDIX
Arc consistency
AC-3 Algorithm

function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables (X, D, C)
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
    (X_i, X_j) \leftarrow Remove-First(queue)
    if Revise(csp, X_i, X_j) then
        if size of D_i = 0 then return false
        for each X_k in X_i.Neighbors – {X_j} do
            add(X_k, X_i) to queue
    return true

function Revise(csp, X_i, X_j) returns true iff we revise the domain of X_i
revised \leftarrow false
for each x in D_i do
    if no value y in D_j allows (x, y) to satisfy the constraint X_i and X_j then
        delete x from D_i
        revised \leftarrow true
return revised

- After applying AC-3, either every arc is consistent or some variable has an empty domain
  \Rightarrow CSP not solvable
- Time complexity: \(O(n^2 d^3)\) (can be reduced to \(O(n^2 d^2)\), but detecting all is NP-hard)
- Trivia: Name stems from this algorithm being the third one in the paper (Mackworth, 1977)
Can we detect inevitable failure early? (contd.)

Ideas for Inference\( (csp, \, \text{var}, \, \text{value}) \)

Constraint learning

- If \texttt{Backtrack()} fails on \( X_i \), it \textbf{backs up to the last variable} and tries another value
  \( \Rightarrow \) would be more intelligent to track back to one of the variables that caused \( D_i = {} \)
- Forward checking etc. already has this information
  \( \Rightarrow \) can be stored in a \textit{conflict set}
- Constraint learning \textbf{adds new constraints} on the fly for sets of assignments (so-called no-goods) that \textbf{repeatedly caused} \texttt{Backtrack()} to fail