ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Constraint Satisfaction Problems

Fundamentals of Artificial Intelligence

MSc in Applied Artificial Intelligence, 2023-24

Contents

- Topics included
  - Constraint Satisfaction Problems
  - Backtracking search and heuristics

- These slides were based essentially on the following bibliography:
  - Norvig, P, Russell, S. (2021). Artificial Intelligence: A Modern Approach, 4th Edition. Pearson, ISBN-13: 978-1292401133

Constraint Satisfaction Problems

Constraint satisfaction problems

- Domain-specific heuristics could estimate the cost of reaching the goal from a given state
    - For the search algorithm, each **state is atomic**, or indivisible— a black box with no internal structure
    - For each problem we need **domain-specific code** to describe the transitions between states.

- Problems can also be addressed using a **factored representation**
    - Each state is defined by a **set of variables** and it corresponding a value
    - A problem is solved when each variable has a value that satisfies all the constraints on the variable
    - A problem described this way is called a **constraint satisfaction problem** (CSP)

- CSP search algorithms take advantage of the structure of states and use **general heuristics** to enable the solution of complex problems
    - Eliminate large portions of the search space by pruning combinations that violate the constraints
    - The actions and transition model can be deduced from the problem description

Defining CSP

- A constraint satisfaction problem consists of three components, X,D, and C:
  - $\mathcal{X}$ is a set of variables, {X1, . . . ,Xn}.
  - ***D*** is a set of domains, {D1, . . . ,Dn}, one for each variable.
  - ***C*** is a set of constraints that specify allowable combinations of values.

- A **domain**, Di, consists of a set of allowable values, {v1, . . . ,vk}, for variable Xi. Different variables can have different domains of different sizes. Domain of a Boolean = {T, F}.
- Each **constraint** Cj consists of a pair «scope, rel», where *scope* is a tuple of variables that participate in the constraint and *rel* is a relation that defines the values that those variables can take on.
- A **relation** can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation
  - For example, if X1 and X2 both have the domain {1,2,3}, then the constraint saying that X1 must be greater than X2 can be written as «(X1,X2),{(3,1), (3,2), (2,1)}» or as «(X1,X2),X1 > X2».

Defining CSP (2)

- CSPs deal with assignments of values to variables, $\{X_i = V_i, X_j = V_j \ldots\}$.
  - An assignment that does not violate any constraints is called a **consistent or legal assignment**
  - A **partial assignment** is one that leaves some variables unassigned
  - A **complete assignment** is one in which every variable is assigned a value
  - A **partial solution** is a partial assignment that is consistent
  - A **solution** to a CSP is a consistent, complete assignment.
- Solving a CSP is an NP-complete (Non-deterministic Polynomial Time) problem in general
  - CSPs is a natural representation for many problems, so it is easy to formulate a problem as a CSP
  - CSP solvers are fast and efficient
  - A CSP solver can quickly prune large portions of the search space
  - For example, once we have chosen {SA=blue} in the Australia problem, all neighboring variables ≠ blue.
- In atomic state-space search we can only ask: is this specific state a goal? No? What about this one?
  - With CSPs, once we find out that a partial assignment violates a constraint, we can immediately discard further refinements of the partial assignment.

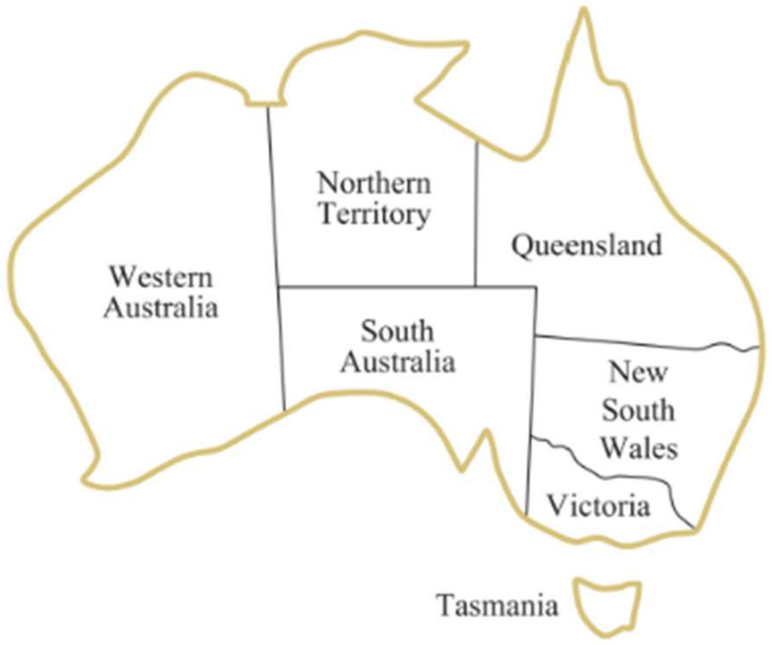
Example 01: Map coloring

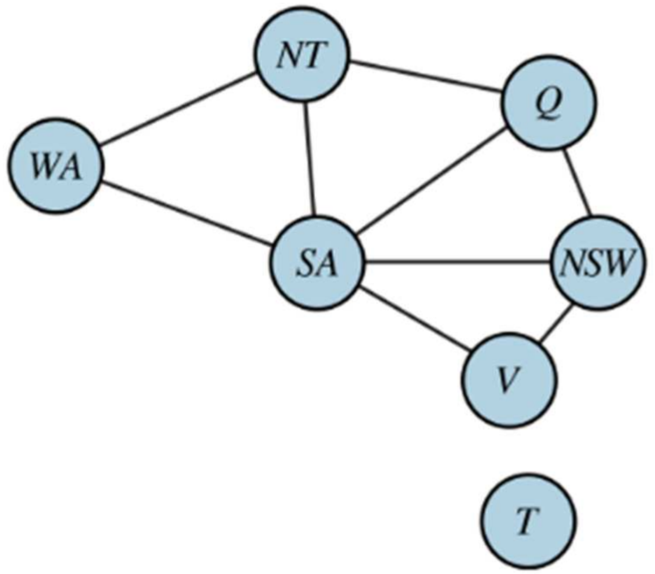
- The task is **to color each region of the Australia states and territories either red, green, or blue** in such a way that no two neighboring regions have the same color
  - Regions (variables): *X* = { WA, NT, Q, NSW, V, SA, T}

- The constraints require neighboring regions to have distinct colors
  - Constraints: C = {SA≠WA, SA≠NT, SA≠Q, SA≠NSW, SA≠V, WA≠NT, NT≠Q, Q≠NSW, NSW≠V}
  - SA≠WA is a shortcut for «(SA,WA),SA≠WA», where SA≠WA can be fully enumerated as {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}.
  - One solution to this problem: {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=red }

- It can be helpful to visualize a CSP as a **constraint graph**
  - the nodes of the graph correspond to variables of the problem
  - an edge connects any two variables that participate in a constraint.

Australian example





- Coloring this map can be viewed as a constraint satisfaction problem (CSP)
  - The goal is to assign colors to each region so that no neighboring regions have the same color
  - The map-coloring problem can be represented as a constraint graph (right figure)

Example 02: Job-shop scheduling

- CSP can help factories the daily scheduling of jobs that are subject to many constraints.
- We can model each job task as a variable, where the **value of each variable is the time** (minutes) that the task starts.
- **Constraints** can assert that one task must occur before another and takes a certain amount of time to complete
- We consider a small set of 15 tasks: install axles (2 tasks), affix all four wheels (4), tighten wheel nuts (4), affix hubcaps (4), and inspect the assembly.
- We can represent the tasks with 15 variables:

  X = {AxleF, AxleB ,WheelRF ,WheelLF ,WheelRB ,WheelLB , NutsRF, NutsLF, NutsRB, NutsLB, CapRF, CapLF ,CapRB ,CapLB, Inspect}.

- The axles takes 10 min to install before the wheels:

  $AxleF + 10 \le WheelRF$; $AxleF + 10 \le WheelLF$;
  $AxleB + 10 \le WheelRB$; $AxleB + 10 \le WheelLB$

- Next, we affix the wheel (1 minute x4), tighten the nuts (2 minutes x4), and finally attach the hubcap (1 minute, but not represented yet):

  $WheelRF + 1 \le NutsRF$; $NutsRF + 2 \le CapRF$;
  $WheelLF + 1 \le NutsLF$; $NutsLF + 2 \le CapLF$;
  $WheelRB + 1 \le NutsRB$; $NutsRB + 2 \le CapRB$;
  $WheelLB + 1 \le NutsLB$; $NutsLB + 2 \le CapLB$.

- Suppose we have four workers, but they the need the same unique tool to put the axle in place. We need a **disjunctive constraint**

  $(AxleF + 10 \le AxleB)$ *or* $(AxleB + 10 \le AxleF)$

- We also need to assert that the inspection comes last and takes 3 min. For every variable X!= Inspect

  $X + dX \le Inspect$

- The whole assembly is done in less then 30 minutes. So, the domain of all variables is

  $D_i = \{0,1,2,3, . . . ,30\}$.

Variations on the CSP formalism

- The simplest kind of CSP involves variables that have discrete, **finite domains**.
  - The 8-queens problem, like the 2 previous examples, are a finite-domain CSP, where ...
    - the variables Q1, . . . ,Q8 correspond to the queens in columns 1 to 8, and
    - the domain of each variable specifies the possible row numbers, $D_i = \{1,2,3,4,5,6,7,8\}$.
    - The constraints say that no two queens can be in the same row or diagonal.
- A discrete **domain can be infinite**, such as the set of integers, so we must use implicit constraints like $T1+d1 \leq T2$
- Special **solution algorithms exist for linear constraints** on integer variables
  - Are constraints, such as the one just given, in which each variable appears only in linear form
  - Linear programming problems is applied for continuous-domain CSPs using linear equalities or inequalities
  - Linear programming problems can be solved in time polynomial in the number of variables (P class)
  - Problems with different types of constraints and objective functions have also been studied
- It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables — the problem is unsolvable

Types of constraints

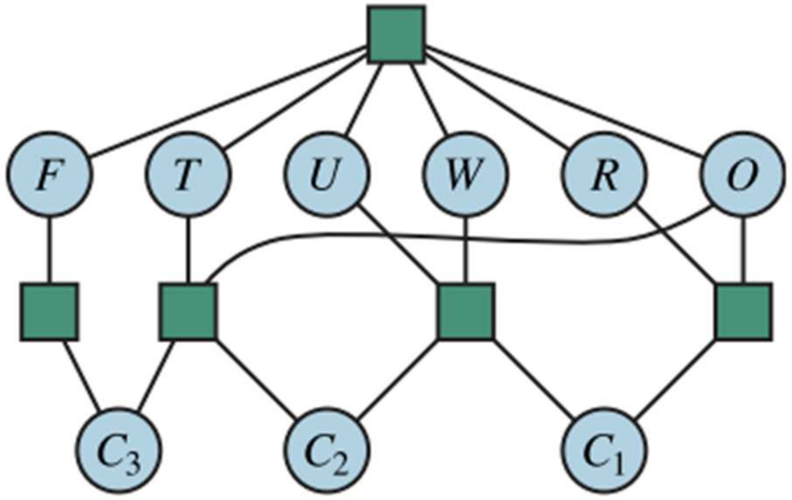
- The simplest type is the **unary constraint**, which restricts the value of a single variable
  - For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green; we can express that with the unary constraint «(SA),SA ≠ green».
  - The initial specification of the domain of a variable can also be seen as a unary constraint
- A **binary constraint** relates two variables
  - For example, SA ≠ NSW is a binary constraint.
  - A binary CSP is one with only unary and binary constraints;
- We can also define higher-order constraints. The **ternary constraint** *Between(X,Y,Z)*, for example, can be defined as «(X,Y,Z),X <Y < Z or X >Y > Z».
- A constraint involving an arbitrary number of variables is called a **global constraint**
  - need not involve all the variables in a problem
  - *Alldiff* says that all the variables in the constraint, e.g., *Alldiff(X,Y,Z)*, must have different values.

Constraint hypergraph

- A **hypergraph** consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints— constraints involving n variables.
- The top figure represents a **cryptarithmetic problem**
  - Each letter stands for a distinct digit with the aim is to find a substitution of digits for letters
  - The resulting sum must be arithmetically correct
  - No leading zeroes are allowed.

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

- The **constraint hypergraph** in the bottom figure, shows
  - The *Alldiff* constraint, the square at top,
  - The column addition constraints, 4 squares In the middle
  - The variables C1, C2, and C3 represent the carry digits for the three columns from right to left.



Preference constraints

- Many real-world CSPs include preference constraints indicating which solutions are preferred
  - E.g., in a university class-scheduling problem, it may allow preference constraints:
    Prof. A might prefer teaching in the morning, whereas Prof. B prefers teaching in the afternoon.
  - The optimal solutions must consider the preferences
- Preference constraints can often be encoded as costs on individual variable assignments
  - E.g., assigning an afternoon slot for Prof. A costs 2 points, whereas a morning slot costs 1.
  - So, CSPs with preferences can be solved with optimization search methods, either path-based or local.
- Linear programs are one class of **constrained optimization problems** (COPs)

Constraint Propagation: Inference in CSPs

- A CSP algorithm has two choices
  - It can generate successors by choosing a new variable assignment or
  - it can do a specific type of inference called constraint propagation
- **Constraint propagation** consists in using the constraints to **reduce the number of legal values for a variable**, which in turn can reduce the legal values for another variable, and so on
  - This will leave fewer choices to consider when we make the next choice of a variable assignment
  - May be intertwined with search, or it may be done as a preprocessing step, before search starts
  - Sometimes this preprocessing can solve the whole problem, so no search is required at all.
- The key idea is **local consistency**
  - Treats each variable as a node in a graph and each binary constraint as an edge
  - The process of <u>enforcing local consistency in each part of the graph can cause inconsistent values</u> that will be eliminated throughout the graph.
- There are different types of local consistency
  - node consistency, arc consistency and path consistency

Local consistency

- A single variable (a node) is **node-consistent** if all the values in its domain satisfy the variable's unary constraints.
  - A graph is node-consistent if every variable in the graph is node-consistent
  - At the start of the solving process all the unary constraints in a CSP can be eliminated by reducing the its domain
  - E.g., if South Australians dislike green, the variable SA starts with domain {red, blue}.
- A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints.
  - Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi,Xj)
  - E.g., the constraint «(X,Y),{(0,0), (1,1), (2,4), (3,9)}» reduces X's domain to {0,1,2,3} and Y's to {0,1,4,9}
  - After applying AC-3 algorithm, either every arc is arc-consistent, or some variable has an empty domain.
- **Path consistency** uses implicit constraints that are inferred by looking at triples of variables
  - E.g., a two-variable set {Xi,Xj} is path-consistent with respect to a variable Xm if, for every assignment {Xi = a, Xj = b} consistent with the constraints on {Xi,Xj}, there is an assignment to Xm that satisfies constraints on {Xi,Xm} and {Xm,Xj}.
  - The "path consistency" refers to the overall consistency of the path from Xi to Xj with Xm in the middle.
- A CSP is **k-consistent** if, for any set of k−1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any k*th* variable

The arc-consistency algorithm AC-3

- After applying AC-3, either
  - every arc is arc-consistent, or
  - some variable has an empty domain, indicating that the CSP cannot be solved.
- AC-3 then pops off an arbitrary arc (X_i,X_j) from the queue and makes X_i arc-consistent with respect to X_j.
- If this leaves D_i unchanged, the algorithm just moves on to the next arc
- But if this revises Di (makes the domain smaller), then we add to the queue all arcs (X_k,X_i) where X_k is a neighbor of X_i

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
    queue ← a queue of arcs, initially all the arcs in csp

    while queue is not empty do
        (Xi, Xj)←POP(queue)
        if REVISE(csp, Xi, Xj) then
            if size of Di = 0 then return false
            for each Xk in Xi.NEIGHBORS - {Xj} do
                add (Xk, Xi) to queue

    return true

function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
    revised←false
    for each x in Di do
        if no value y in Dj allows (x, y) to satisfy the constraint between Xi and Xj then
            delete x from Di
            revised ← true

    return revised
```

Global constraints

- A **global constraint** is one involving an arbitrary number of variables, but not necessarily all variables
    - Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far.
    - For example, the *Alldiff* constraint says that all the variables involved must have distinct values

- One simple form of **inconsistency detection for *Alldiff*** constraints works as follows:
    - if *m* variables are involved in the constraint, and if they have *n* possible distinct values altogether, and *m* > *n*, then the constraint cannot be satisfied
    - This leads to the following simple algorithm
        - First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables.
        - If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

Resource constraint

- Another important higher-order constraint is the **resource constraint** or the ***Atmost*** constraint.
- E.g., in a scheduling problem, let P1, . . . ,P4 denote the personnel assigned to each of four tasks.
    - The constraint that no more than 10 personnel are assigned is written as *Atmost(10,P1,P2,P3,P4)*.
    - Inconsistency is detected by checking the sum of the minimum values of the current domains;
    - If each variable has the domain {3,4,5,6}, the *Atmost* constraint cannot be satisfied. If each variable in our example has the domain {2,3,4,5,6}, the values 5 and 6 can be deleted from each domain.

- For large resource-limited problems with integer values, domains are represented by upper and lower bounds and are managed by **bounds propagation**.
    - For example, in an airline-scheduling Bounds propagation problem, let's suppose there are two flights, F1 and F2, for which the planes have capacities 165 and 385, respectively.
    - The initial domains for the passengers on flights F1 and F2 are then D1 = [0,165] and D2 = [0,385]
    - If the two flights together must carry 420 people: F1 +F2 = 420,
      then D1 = [35,165] and D2 = [255,385]

Sudoku game

- Introduced millions of people to constraint satisfaction problems
- In Sudoku puzzles for people resolve there is exactly one solution.
- A Sudoku board consists of **81 squares**, grouped into **9 boxes**, some of which are initially filled with digits from 1 to 9.
    - The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box, called a unit.
- A Sudoku puzzle can be considered a CSP with 81 variables
    - Variable names A1 through A9 stand for the top row (left to right),
    - The empty squares have the domain {1,2,3,4,5,6,7,8,9}
    - The prefilled squares have a domain consisting of a single value
    - There are **27 different Alldiff** constraints, one for each unit (row, column, and box of 9 squares):

      Alldiff(A1,A2,A3,A4,A5,A6,A7,A8,A9)

      Alldiff(B1,B2,B3,B4,B5,B6,B7,B8,B9)

      . . .

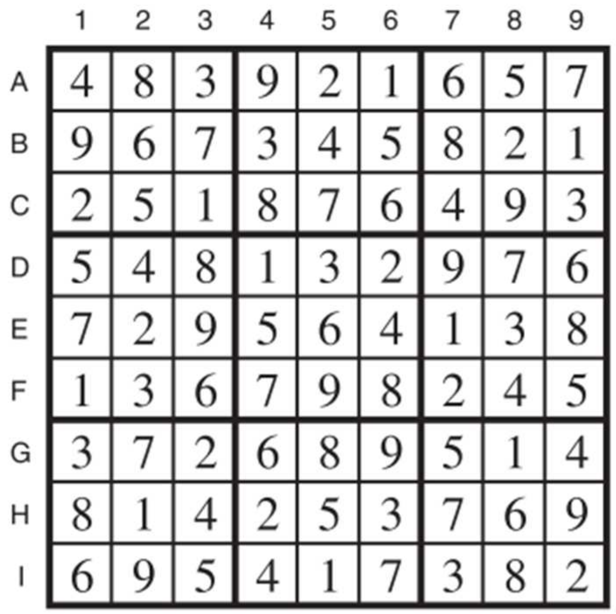
	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Alldiff(A1,B1,C1,D1,E1,F1,G1,H1, I1)
Alldiff(A2,B2,C2,D2,E2,F2,G2,H2, I2)
. . .
Alldiff(A1,A2,A3,B1,B2,B3,C1,C2,C3)
Alldiff(A4,A5,A6,B4,B5,B6,C4,C5,C6)
. . .

Sudoku game (2)

- We can apply the AC-3 algorithm directly if *Alldiff* constraints are expanded into binary constraints (such as A1 ≠ A2)
  - However, AC-3 works only for the easiest Sudoku puzzles
- All the strategies — arc consistency, path consistency, and so on — apply generally to all CSPs, not just to Sudoku problems
- Humans apply more complex inference strategies, combining the domain of multiple squares to reduce the domain of other squares or units
- This is the power of the CSP formalism

  For each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.



	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Backtracking search and heuristics

Backtracking Search for CSPs

- For searching for a solution of a CSP problem, we can use
  - **backtracking search** algorithms that work on partial assignments
  - **local search** algorithms over complete assignments, when we still have variables with multiple values
- Consider how a standard depth-limited search could solve CSPs (e.g., Australia map)
  - A state would be a partial assignment of one variable (e.g., WA = red)
  - An action would extend the assignment, adding more variables, e.g., NSW= *red* or SA = *blue*
- For a CSP with ***n* variables** of domain **size *d*** we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n.
  - The branching factor at the top level would be *n*d* because any of *d* values can be assigned to any of *n* variables.
  - At the next level, the branching factor is (n−1)d, and so on for n levels
  - The tree has **$n! * d^n$ leaves**, even though there are only d^n possible complete assignments
- A problem is **commutative** if the order of application of any given set of actions does not matter.
  - In CSPs, it makes no difference if we first assign NSW = red and then SA = blue, or the other way around
  - So, we need only consider a single variable at each node in the search tree and **number of leaves is d^n**

Backtracking Algorithm for CSPs

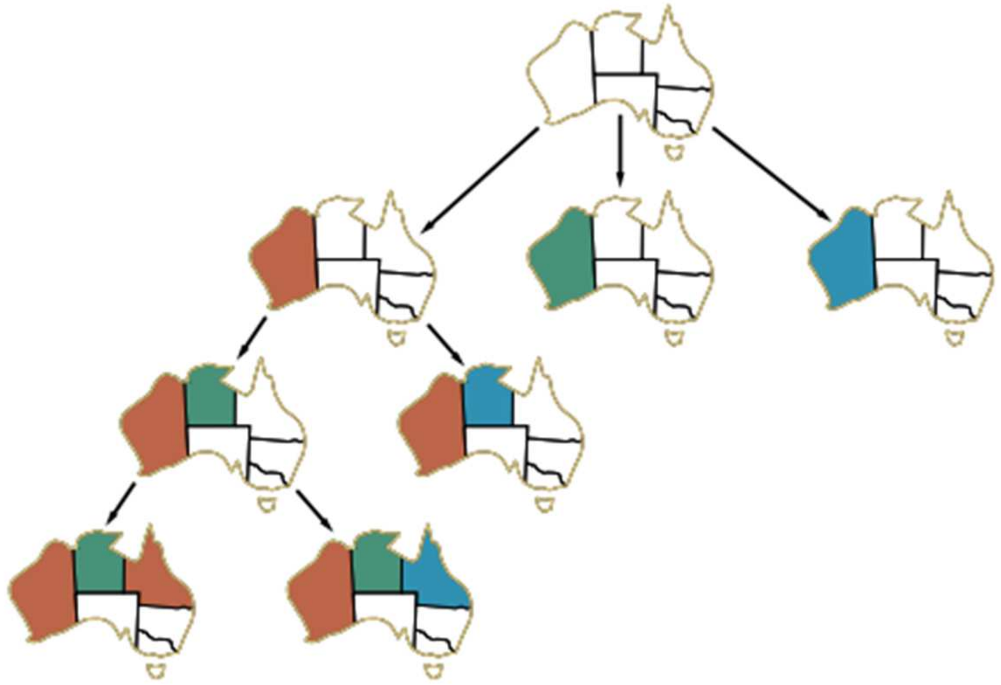
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, {})

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
            remove {var = value} from assignment
    return failure
```

Notes

- The INFERENCE function can optionally impose arc-, path-, or k-consistency, as desired
- If a value choice leads to failure, then value assignments are retracted and a new value is tried.

Search procedure for CSPs – Backtracking

- It repeatedly **chooses an unassigned variable**, and then
  - tries all values in the domain of that variable,
  - tries to extend each one into a solution via a recursive call.
- If the call succeeds, the solution is returned
  - if it fails, the assignment is restored to the previous state, and we try the next value.
- If no value works, then we return failure.
- Part of the search tree for the Australia problem is shown in the figure
- Notice that BACKTRACKING-SEARCH keeps only a single representation of a state (assignment)
  - alters that representation rather than creating new ones



Variable ordering

- Simple strategies for choosing the next unassigned-variable
  - The simplest strategy is **static ordering**: choose the variables in order, {X1,X2, . . .}.
  - The next simplest is to choose **randomly**.
  - Neither strategy, static ordering or randomly, is optimal.
- The **minimum-remaining-values** (**MRV**) consists of choosing the variable with the fewest “legal” values.
  - it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.
  - If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.
  - The MRV heuristic usually performs better than a random or static ordering, sometimes by orders of magnitude, although the results vary depending on the problem.
- The **degree heuristic** attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables
- The MRV heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker

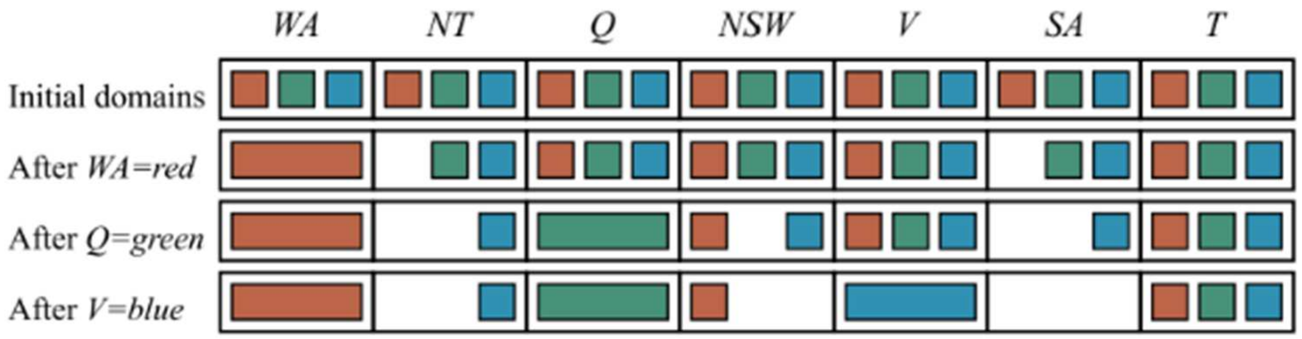
Value ordering

- The **least-constraining-value** (LCV) heuristic is effective for value selection ordering.
  - It prefers the value that excludes the fewest choices for the neighboring variables in the graph.
  - In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

- Why should variable selection be fail-first, but value selection be fail-last?
  - Concerning **variable selection**, every variable has to be assigned eventually, so by choosing the ones that are likely to fail first, we will on average have fewer successful assignments to backtrack over.
  - For **value ordering**, the trick is that we only need one solution
    - Therefore, it makes sense to look for the most likely values first
    - If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

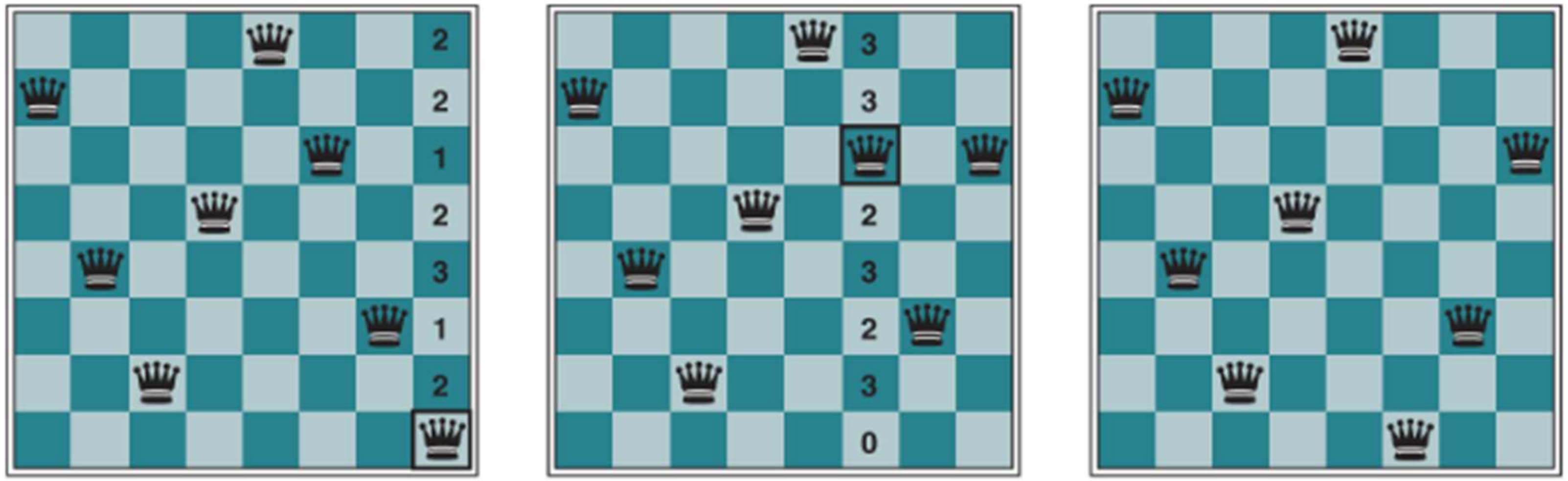
Interleaving search and inference

- **Inference** can reduce the domains of variables during the search
  - Every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables
- One of the simplest forms of inference is called **forward checking**
  - Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it.
  - For each unassigned variable Y that is connected to X by a constraint, delete from Y’s domain any value that is inconsistent with the value chosen for X.
  - Forward checking detects many inconsistencies, but doesn’t look ahead far enough
- **Maintaining Arc Consistency (MAC)** recursively propagates constraints when changes are made to the variable domains



Local Search for CSPs

- Uses a complete-state formulation where each state assigns a value to every variable, and the search changes the value of one variable at a time.
- Using the 8-queens problem, we start on the left with a complete assignment to the 8 variables
  - Typically, this will violate several constraints. We then randomly choose a conflicted variable, which turns out to be Q8, the rightmost column.
  - Using the **min-conflicts heuristic**, we will select the value that results in the minimum number of conflicts with other variables — the closer to the solution.



Min-conflicts

- Min-conflicts is surprisingly effective for many CSPs, including hard problems
  - On the n-queens problem, after the initial placement of queens, the run time of min-conflicts is independent of problem size. It solves even the million-queens problem in an average of 50 steps
  - It has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.
- It can be used in an online setting when the problem changes, e.g., a scheduling problem for an airline's weekly flights with thousands of flights, but bad weather at one airport can render the schedule infeasible. Local search can repair the schedule with a minimum number of changes.
- The landscape of a CSP under the min-conflicts heuristic usually has a series of plateaus
  - There may be millions of variable assignments that are only one conflict away from a solution.
  - **Tabu search** keeps a small list of recently visited states and forbids the algorithm to return to those states can be used for solve the plateau problem
  - **Simulated annealing** can also be used to escape from plateaus
  - **Constraint weighting** aims to concentrate the search on the important constraints

CSP Summary

- CSPs are a very specific kind of problems
  - The states are defined by the values of a fixed set of variables
  - Goal test is executed by the validation of the constraints on variable values

- Backtracking –> dept-first search with one variable assigned by node
  - Heuristics on variable ordering and value selection can help
- Forward checking prevents assignments that will result in later failure
- Constraint propagation, e.g., arc consistency, limit values and detects inconsistencies

- The CSP formulation allows to analyse the problem structure
- Tree-structured CSPs can be solved in linear time
- Iterative min-conflicts is usually effective in practice

Real-world Constraint Satisfaction Problems

These are the most common cases of CSPs

- Scheduling
  - How to efficiently and effectively schedule resources like personnel, equipment, and facilities.
  - The constraints in this domain specify the availability and capacity of each resource
  - The variables indicate the time slots or resources.
- Vehicle routing
  - How to minimizing travel time or distance by optimizing a fleet of vehicles’ routes
  - The constraints specify each vehicle’s capacity, delivery locations, and time windows
  - The variables indicate the routes taken by the vehicles.
- Assignment
  - How to optimally assign assignments or jobs to humans or machines
  - The variables stand in for the tasks
  - The constraints specify the knowledge, capacity, and workload of each person or machine.

Exercise 01

Consider the problem of placing k knights on an n×n chessboard such that no two knights are attacking each other, where k is given and k ≤ n2.

1. Choose a CSP formulation. In your formulation, what are the variables?
2. What are the possible values of each variable?
3. What sets of variables are constrained, and how?
4. Now consider the problem of putting *as many knights as possible* on the board without any attacks. Explain how to solve this with local search by defining appropriate ACTIONS and RESULT functions and an objective function.

CSP problem formulation

- Variables = each one of the k Knights = {K_1, K_2, ..., K_k}
- Domain for all K = the board cells {(1,1), (1,2), ..., (n,n)}
- Constraints
  - Knights have to be in different cells
    Alldiff(K1, K2, ..., Kk)
  - No knight can be on a cell attacked by other knight
    For each knight Ki, Alldiff(Ki, CellsAttacked)
- Local search
  - Actions(state) ➔ Move(Knight, cell)
  - Result (state, action) ➔ nr of pairs of knights attacking each other
  - Objective (state) ➔ True if KnightsAttacked = {}
- The approach is to start from n knights in the nxn board and, if we get a solution, increment knights by 1 until we find no solution.

Exercise 02

Consider the logic puzzle. Given the following facts, the questions to answer are :

- Where does the zebra live?
- In which house do they drink water?

Discuss different representations of this problem as a CSP.

Why would one prefer one representation over another?

In five houses, each with a different color, live five persons of different nationalities, each of whom prefers a different brand of candy, a different drink, and a different pet.

- *The Englishman lives in the red house.*
- *The Spaniard owns the dog.*
- *The Norwegian lives in the first house on the left.*
- *The green house is immediately to the right of the ivory house.*
- *The man who eats Hershey bars lives in the house next to the man with the fox.*
- *Kit Kats are eaten in the yellow house.*
- *The Norwegian lives next to the blue house.*
- *The Smarties eater owns snails.*
- *The Snickers eater drinks orange juice.*
- *The Ukrainian drinks tea.*
- *The Japanese eats Milky Ways.*
- *Kit Kats are eaten in a house next to the house where the horse is kept.*
- *Coffee is drunk in the green house.*
- *Milk is drunk in the middle house.*

Exercise 02 — Solution A

CSP problem formulation

- Variables = 5 persons living in 5 houses
  Each house has 5 attributes = {H1.n, H1.c , H1.b , H1.d, H1.p, H2.n, ..., H5.p}
- Domain = nationality (n) = {Englishman, Japanese, Norwegian, Spaniard, Ukrainian}
  house color (c) = {Blue, green, ivory, red, yellow}
  candy brand (b) ={Hershey, Kit Kats, Milky Ways, Smarties, Snickers}
  drink (d) = {coffee, milk, orange juice, tea, water}
  pet (p) = {dog, fox, horse, snails, zebra}
- Constraints (Global)

Persons have different attributes

Nationality (Pi.n)	➔ *Alldiff*(H1.n, H2.n , H3.n , H4.n , H5.n)
house color (Pi.c)	➔ *Alldiff*(H1.c, H2.c, H3.c , H4.c , H5.c)
candy brand (Pi.b)	➔ *Alldiff*(H1.b, H2.b , H3.b , H4.b , H5.b)
drink (Pi.d)	➔ *Alldiff*(H1.d, H2.d , H3.d , H4.d , H5.d)
pet (Pi.p)	➔ *Alldiff*(H1.p, H2.p , H3.p , H4.p , H5.p)

Exercise 02 — Solution B

CSP problem formulation

- Variables = One variable for each instance, 5 x 5 = 25 variables

  nationality variables = { Englishman, Japanese, Norwegian, Spaniard, Ukrainian }
  house_color variables = { Blue, green, ivory, red, yellow }
  candy brand variables = { Hershey, Kit Kats, Milky Ways, Smarties, Snickers }
  drink variables = { coffee, milk, orange juice, tea, water }
  pet variables = { dog, fox, horse, snails, zebra }

- Domain = All variable have the same domain = { 1, 2, 3, 4, 5 }

- Constraints (Global) — Persons have different attributes

Nationality	➔ *Alldiff*(Englishman, Japanese, Norwegian, Spaniard, Ukrainian)
house_color	➔ *Alldiff*((Blue, green, ivory, red, yellow)
candy brand	➔ *Alldiff*(Hershey, Kit Kats, Milky Ways, Smarties, Snickers)
drink (Pi.d)	➔ *Alldiff*(coffee, milk, orange juice, tea, water)
pet (Pi.p)	➔ *Alldiff*(dog, fox, horse, snails, zebra)

Exercise 02 Solution

Solution A

- Assignments

  {H1.n = Norwegian}

- Binary Constraints

  «(Hi.n, Hi.c), {Spaniard, dog}»,
  «(Hi.n, Hi.c), {Englishman, Red}»,
  «(Hi.c, Hi.c), green at_right ivory»,
  «(Hi.b, Hi.p), Hershey next_to fox»,
  «(Hi.c, Hi.b), {Yellow, KitKats}»,
  ...

-

  ...

Solution B

- Assignments

  { Norwegian = 1 }

- Binary Constraints

  «(Spaniard, dog), Spaniard = dog»,
  «(Englishman, Red), Englishman = Red»,
  «(green, ivory), green at_right ivory»,
  «(Hershey, fox), Hershey next_to fox»,
  «(Yellow, KitKats), Yellow = KitKats»,

  ...

-

  ...

Thank you!