ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Search in Complex Environments

Artificial Intelligence – *Inteligência Artificial*

Bachelor in Computer Systems Engineering , 2022-23

Licenciatura em Engenharia de Sistemas Informáticos

Contents

- Topics included
  - Local search
  - Nondeterministic Actions and Partially Observable Environments
  - Constraint Satisfaction Problems

- 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

Local Search

Search in complex environments

- The general approaches for problem solving, discussed in previous sections, address problems in fully observable, deterministic, static, known environments where the solution is a sequence of actions

- This section will introduce ...
  - the problem of finding a good state without worrying about the path to get there, covering both discrete and **continuous states**
  - how to use conditional plan and carry out different actions depending on what the agent observes in a **nondeterministic** world
  - how to keep track of the possible states the agent might be in **partial observable** world

Local search

- Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.
- These algorithms are not systematic — they might never explore a portion of the search space where a solution actually resides.

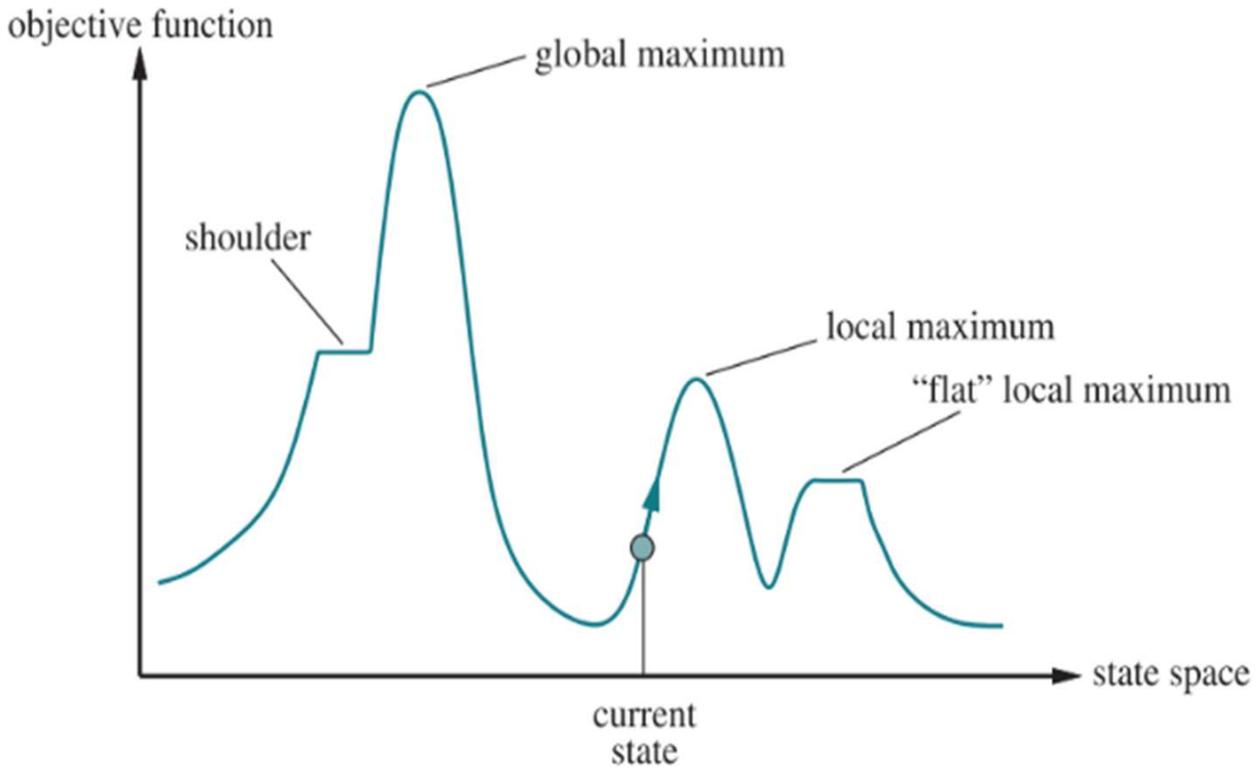
- However, local search algorithms have two key advantages
  - (1) they **use very little memory**; and
  - (2) they can **often find reasonable solutions** in large or infinite state spaces for which systematic algorithms are unsuitable.

- Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an objective function.

Optimization Problems

Consider the states of a problem laid out in a state-space landscape, as shown in Figure

- Each point (state) in the landscape has an “elevation,” defined by the value of the objective function.
- If elevation corresponds to an objective function, then the aim is to find the highest peak — a **global maximum** — and we call the process **hill climbing**.
- If elevation corresponds to cost, then the aim is to find the lowest valley — a **global minimum** —and we call it gradient descent.



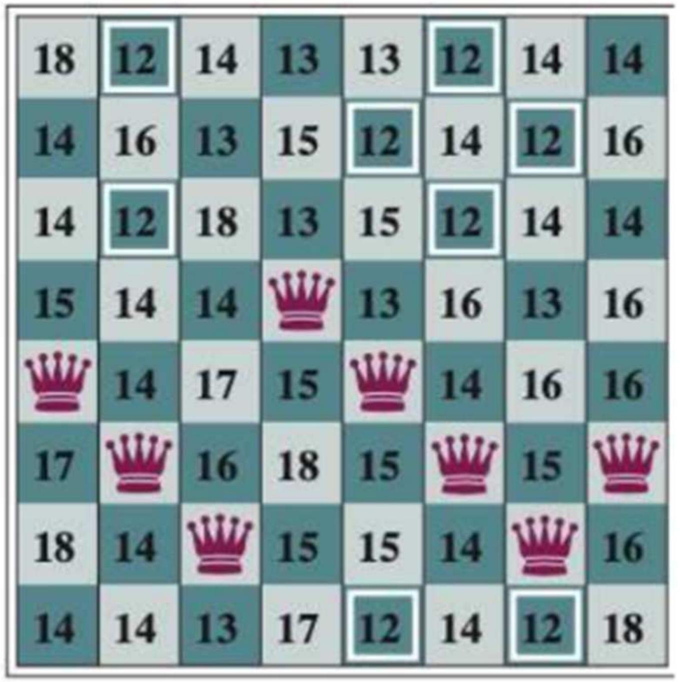
Iterative Improvement

- In many optimization problems, the path to the goal is irrelevant!
  - The **goal is itself the solution**!
  - State space = set of complete configurations!
- Local search algorithms are used for iterative improvement
  - Start as any (initial) solution of the problem and make changes to improve its quality

- **Hill-Climbing**
  - Pick a state randomly from the state space
  - Consider all neighbors of that state — we need to define a range
  - Pick the best neighbor
  - Repeat the process until there are no better neighbors
  - The current state is the solution

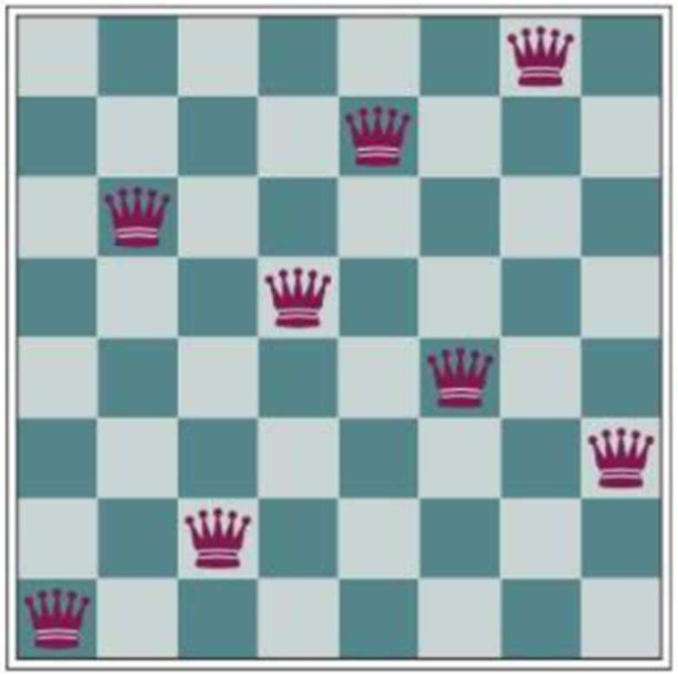
Hill-Climbing for the 8-queens problem

- Complete-state formulation — every state has all the components of a solution: 8 queens on the board, one per column
- The **initial state is chosen at random**, and
  - the successors of a state are all possible states generated by moving a single queen to another square in the same column
  - For each state there are 8×7 = 56 successors.
- The heuristic cost function *h* is **the number of pairs of queens that are attacking each other** — for solution states *s*, *h*(s) = 0
  - It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them
- board shows an 8-queens state with heuristic cost estimate ***h*=17**.
  - The value of *h* for each possible successor found by moving a queen within its column. There are 8 moves that are tied for best, with h=12.
  - The hill-climbing algorithm will pick one of these.



Hill-Climbing limitations

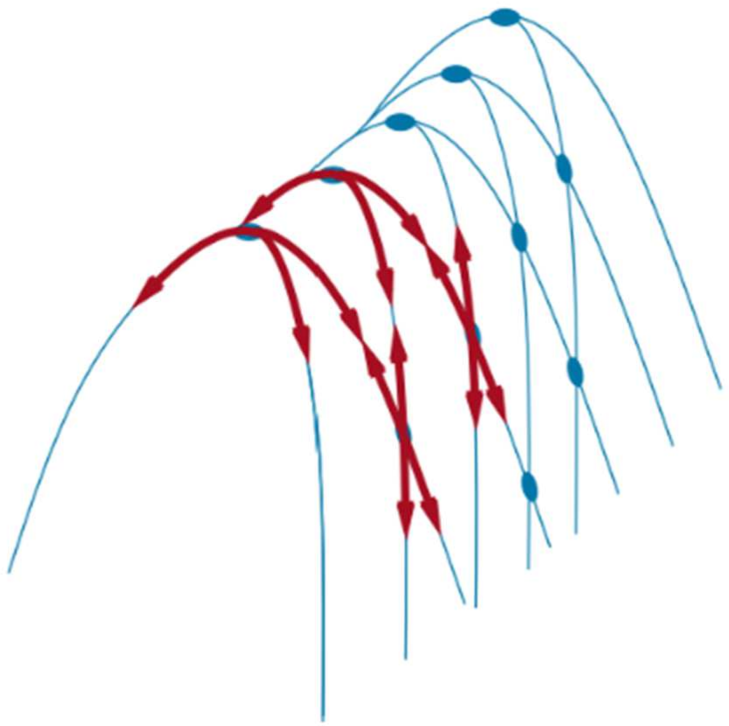
- **Local maxima**
  - A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
  - Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
  - The state in right board is a local maximum (i.e., a local minimum for the cost h). Every move of a single queen makes the situation worse.



- **Plateaus**
  - A plateau is a flat area of the state-space landscape. It can be a flat local Plateau maximum (no uphill exit exists) or a shoulder
  - A hill-climbing search can get lost wandering on the plateau.

Hill-Climbing limitations (2)

- **Ridges**
  - A ridge is shown in the right Figure
  - Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.



- In each limitation (3), the algorithm reaches a point at which no progress is being made.
  - Starting from a randomly generated 8-queens state, steepest-ascent hill climbing **gets stuck 86%** of the time, **solving only 14% of problem instances**.
  - On the other hand, it works quickly, taking **just 4 steps on average when it succeeds** and **3 when it gets stuck** — not bad for a state space with 88 ≈ 17 million states.

- There are many variants of hill-climbing to keep going when we reach a plateau — to allow a **sideways move** in the hope that the plateau is really a shoulder

Local beam search

- The local beam search algorithm keeps track of k states rather than just one.
    - It begins with **k randomly generated states**
    - At each step, all the successors of all k states are generated.
      If any one is a goal, the algorithm halts.
    - Otherwise, it selects the k best successors from the complete list and repeats.
- A local beam search with k states is different from than running k random restarts in parallel.
    - In a random-restart search, each search process runs independently of the others.
    - In a local beam search, useful information is passed among the parallel search threads.
    - The **algorithm quickly abandons unfruitful searches** and moves its resources to where the most progress is being made.
- Local beam can suffer from lack of diversity among the k states — they can become clustered in a small region of the state space, making the search little more than a k-times-slower version of hill climbing
- **Stochastic beam search**, Instead of choosing the top k successors, chooses successors with probability proportional to the successor's value, thus increasing diversity.

More Iterative Improvement Algorithms

- **Simulated Annealing**
  - Similar to Hill-Climbing Search but allows to explore worse neighbors
- **Tabu Search**
  - Explores the neighboring states but eliminates the worst ones (tabu neighbors)
  - maintains a tabu list of k previously visited states that cannot be revisited;
  - improves efficiency when searching graphs,
  - can allow the algorithm to escape from some local minima.
- **Ant Colony Optimization**
  - Several starting states (ant colony)
  - Determines the probability of a path being better from the number of "ants" passing through it
- **Particle Swarm Optimization**
  - Several starting states (swarm)
  - The neighborhood is explored, and the best solution and the best state are saved
  - The states are moved towards the best solution found so far
  - The speed of movement depends on the distances to the best solution and the best state and the position of the state
- **Genetic Algorithms**
  - Define a state as a chromosome
  - Generate solutions (chromosomes) from a population of initial states

Evolutionary algorithms

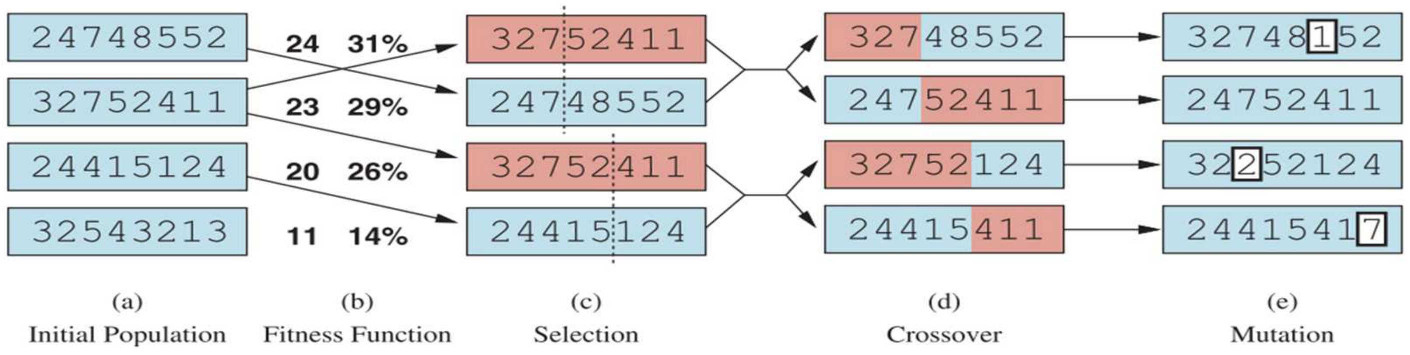
- Seen as variants of stochastic beam search motivated by the metaphor of natural selection:
  - there is a population of individuals (states), in which the fittest (highest value) individuals produce offspring (successor states) that populate the next generation, a process called recombination

- There are endless forms of **recombination** that vary on ...
  - The size of the population
  - The representation of each individual. In genetic algorithms, each individual is a string over a finite alphabet (often a Boolean string)
  - The mixing number, ρ, which is the number of parents that come together to form offspring. The most common case is $\rho = 2$: two parents combine their “genes” (parts of their representation) to form offspring. It is possible to have $\rho=1$ or $\rho > 2$
  - The selection process for the parents of the next generation: (a) select from all individuals according to their fitness score; (2) randomly select n individuals ($n > \rho$), and then select the ρ most fit ones as parents.

Evolutionary algorithms (2)

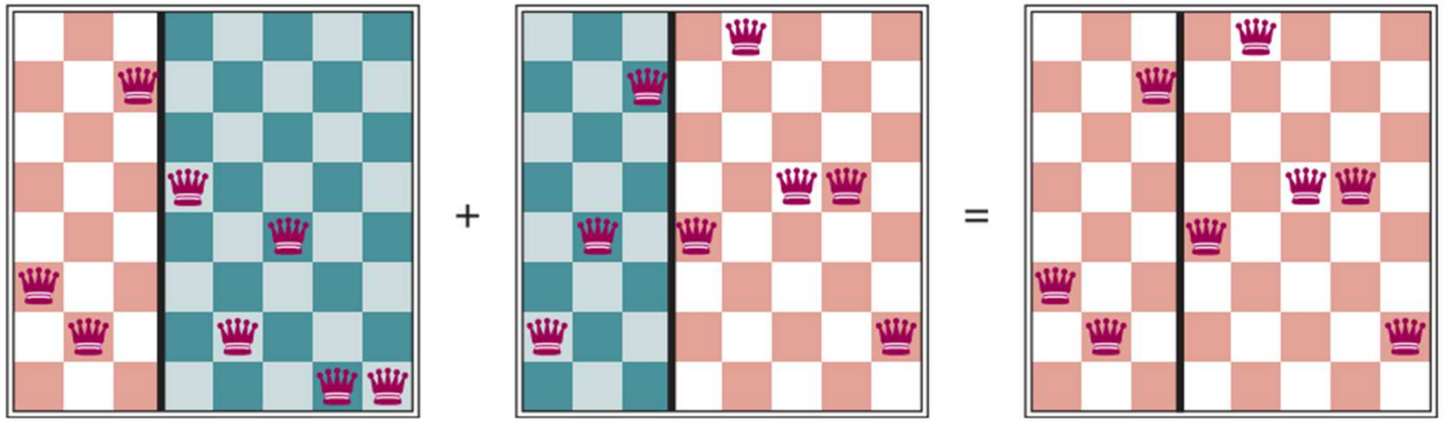
- More ways about how evolutionary algorithms use **recombination**...
  - The recombination procedure. One common approach (assuming ρ= 2), is to randomly select a crossover point to split each of the parent strings, and recombine the parts to form two children
  - The mutation rate, which determines how often offspring have random mutations to their representation. Once an offspring has been generated, every bit in its composition is flipped with probability equal to the mutation rate.
  - The makeup of the next generation. This can be just the newly formed offspring, or it can include a few top-scoring parents from the previous generation. The practice of culling, in which all individuals below a given threshold are discarded, can lead to a speedup

The 8-queens problem using evolutionary algorithms



- Above figure shows a population of four 8-digit strings, each representing a state of the 8queens puzzle: the *n*-th digit represents the row number of the queen in column *n*.
- In (b), each state is rated by the fitness function.
  - Higher fitness values are better, so for the 8-queens problem we use the number of nonattacking pairs of queens, which has a value of 8×7/2 = 28 for a solution
  - The values of the four states in (b) are 24, 23, 20, and 11. The fitness scores are then normalized to probabilities.
- In (c), two pairs of parents are selected, in accordance with the probabilities in (b)
  - Notice that one individual is selected twice and one not at all. For each selected pair, a crossover point (dotted line) is chosen randomly.
- In (d), we cross over the parent strings at the crossover points, yielding new offspring.
  - For example, the first child of the first pair gets the first three digits (327) from the first parent and the remaining digits (48552) from the second parent.

The 8-queens problem using evolutionary algorithms (2)

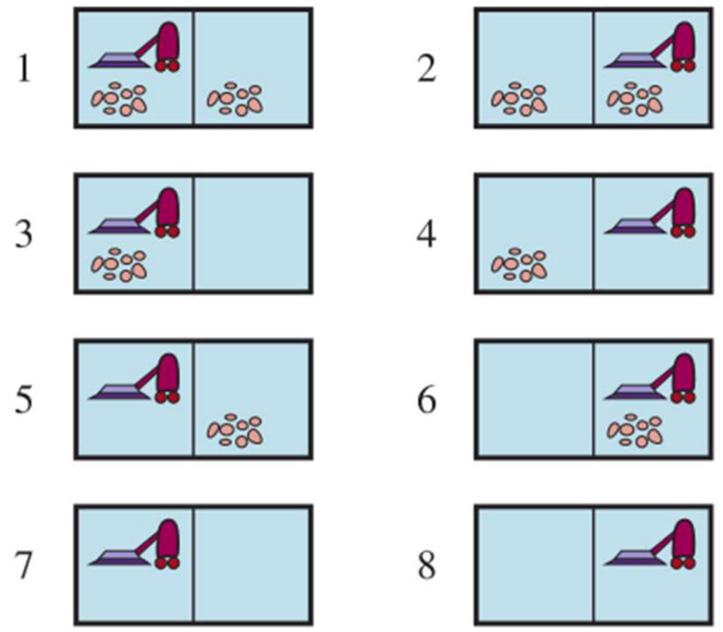


- The 8-queens states corresponding to the first two parents in Figures (c) and (d), shown in previous figure (positions correspond to the row: 1 on bottom).
  - The green/blue columns are lost in the crossover and the red columns are retained.
- Genetic algorithms are similar to stochastic beam search with the crossover operation
- This is advantageous if there are blocks that perform useful functions.
  - E.g., putting the first three queens in positions 2, 4, and 6, where they do not attack each other, may be a useful block that can be combined with other useful blocks
  - It can be shown mathematically that, if the blocks do not serve a purpose, then crossover conveys no advantage.
- Evolutionary algorithms are suitable for complex problems
  - Usually need some specific adaptations/improvements
  - But do not use when other method, like hill-climbing, works well

Nondeterministic Actions and Partially Observable Environments

Search with Nondeterministic Actions

- Previous search algorithms assumed the environment is fully observable, deterministic, and known.
- However ...
  - when it is **partially observable**, the agent doesn't know for sure what state it is in;
  - when it is **nondeterministic**, the agent doesn't know what state it transitions to.
- The agent should think "I'm either in state s1 or s3, and if I do action A I'll end up in state s2, s4 or s5."
- A set of physical states that the agent believes are possible is called a **belief state**
- The solution is no longer a sequence, but a **conditional plan**,
  - It specifies what to do depending on what percepts agent receives while executing the plan.



The eight possible states of the vacuum world; states 7 and 8 are goal states.

The erratic vacuum world

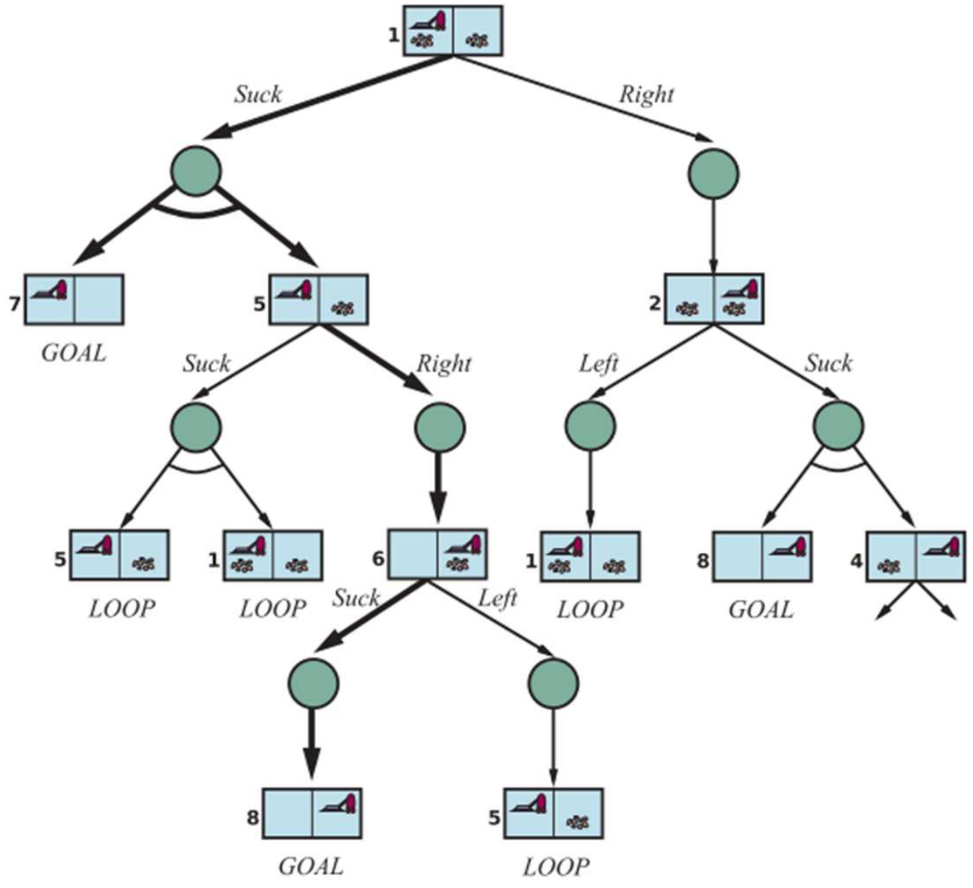
- The vacuum world example (previous slide figure)
  - there are three actions—Right, Left, and Suck
  - the goal is to clean up all the dirt (states 7 and 8).
- Now suppose we have a nondeterministic, **erratic vacuum world**, where the **Suck action** works as follows:
  - When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
  - When applied to a clean square the action sometimes deposits dirt on the carpet
- Instead of returning a single outcome state, the RESULTS function returns a set of possible outcome states.
- E.g., the Suck action in state 1 cleans up either just the current location, or both locations:

  RESULTS(1, Suck) = {5,7}
- If we start in state 1, no single sequence of actions solves the problem, but the following conditional plan does:

  [Suck, if State=5 then [Right,Suck] else [ ]]
- A conditional plan means that solutions are trees rather than sequences.
- In the if statement, the agent will be able to observe at runtime what the current state is
  - Alternatively, the agent could test the percept rather than the state.
- Many problems in the real world are contingency problems, because exact prediction of the future is impossible.

AND–OR search trees

- In a deterministic environment, the only branching is introduced by the agent’s own choices in each state: **I can do this action or that action**.
  - State nodes are **OR nodes** where some action must be chosen: Left or Right or Suck.
- In a nondeterministic environment, branching also results of the environment’s choice of outcome for each action: **AND nodes** (**circles**)
- E.g., the Suck action in state 1 results in the **And node belief state** {5,7}, so the agent would need to find a plan for state 5 and for state 7.
  - At the AND nodes, every Suck action outcome must be handled, as indicated by the arc linking the outgoing branches.



The first two levels of the search tree. The solution is shown in bold lines. it corresponds to the plan given in *[Suck, if State=5 then [Right,Suck] else [ ]]*

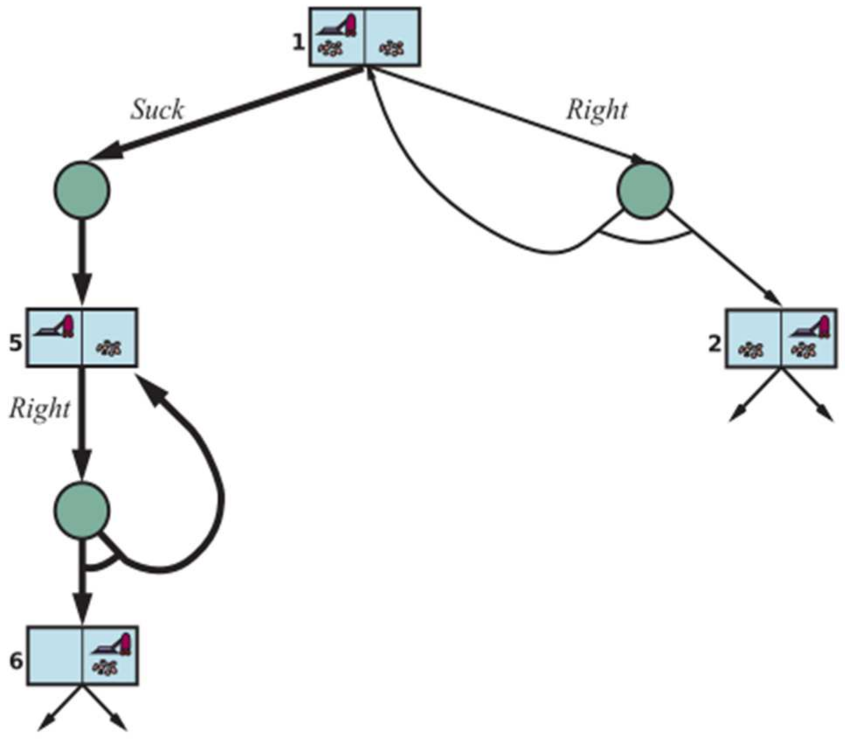
AND–OR search trees (2)

- A **solution for an AND–OR search problem is a subtree** of the complete search tree that
  - (1) has a goal node at every leaf
  - (2) specifies one action at each of its OR nodes, and
  - (3) includes every outcome branch at each of its AND nodes.
- One key aspect of the algorithm is the way in which it deals with cycles
  - e.g., if an action sometimes has no effect or if an unintended effect can be corrected
- If the current state is identical to a state on the path from the root, then it returns with failure
  - This doesn't mean that there is no solution from the current state;
  - it simply means that if there is a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded.
- It ensures that the algorithm finishes in every finite state space: a goal, a dead end, or a repeated state.
  - The algorithm does not check whether the current state is a repetition of a state on some other
- AND–OR graphs can be explored either breadth-first or best-first.
  - The concept of a heuristic function must be modified to estimate the cost of the solution
  - Next slide presents an analog of the A∗ algorithm for finding optimal solutions.

An algorithm for searching AND–OR graphs

An algorithm for searching AND–OR graphs generated by nondeterministic environments. A solution is a conditional plan that considers every nondeterministic outcome and makes a plan for each one.

```
function AND-OR-SEARCH (problem) returns a conditional plan, or failure
    return OR-SEARCH(problem, problem.INITIAL, [ ])

function OR-SEARCH (problem, state, path) returns a conditional plan, or failure
    if problem.IS-GOAL(state) then return the empty plan
    if IS-CYCLE(state, path) then return failure
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + [path])
        if plan ≠ failure then return [action] + [plan]
    return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
    for each $s_i$ in states do
        $plan_i$ ← OR-SEARCH(problem, $s_i$, path)
        if $plan_i$ = failure then return failure
return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else . . . if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```



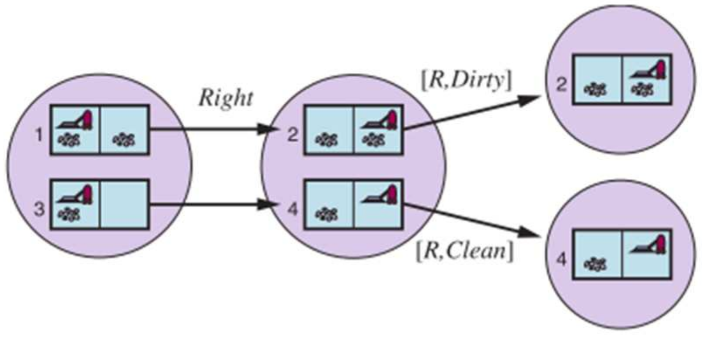
Cycle try, when action fails

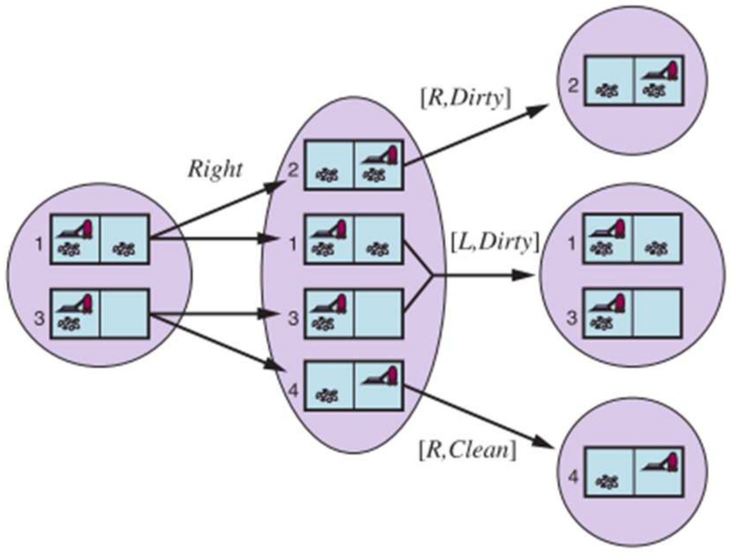
- Consider a **slippery vacuum world**, where the **movement actions sometimes fail**, leaving the agent in the same location.
  - E.g., moving Right in state 1 leads to the belief state {1,2}.
- There are no longer any acyclic solutions from state 1, and AND-OR-SEARCH would return with failure.
- Cyclic solution is to ***keep trying Right*** until it works using a
  - while construct: [Suck, while State=5 do Right, Suck] or
  - adding a label to some portion of the plan and referring to the label later: [Suck,L1 : Right, if State=5 then L1 else Suck]
- A cyclic plan is a solution if every leaf is a goal state and a leaf is reachable from every point in the plan.

- Which is the cause of the nondeterminism? It is an action execution failure that can be successfully if we try repeatedly, or it is due to some unobserved fact about the robot or environment?
- One way to solve is to **change the initial problem** formulation (fully observable, nondeterministic) to **partially observable, deterministic**, where the failure of the cyclic plan is attributed to an unobserved fact.

Searching in partially observable environments

- For a partially observable problem, is specified a PERCEPT(s) function that returns the percept received by the agent in a given state.
  - If sensing is nondeterministic, PERCEPTS function returns a set of possible percepts.
  - For fully observable problems, PERCEPT(s)=s for every state s, and for sensorless PERCEPT(s)=null.
- Consider a local-sensing vacuum world, in which the agent has
  - a **position sensor** that yields the percept L/R percepts, and
  - a **dirt sensor** that yields Dirty/Clean percepts
- With partial observability, it will usually be the case that **several states produce the same percept**;
  - E.g., state 3 will also produce [L, Dirty]. Hence, given this initial percept, the initial belief state will be {1,3}
- Top figure represents the **deterministic world** case
- bottom figure represents the **nondeterministic world** case

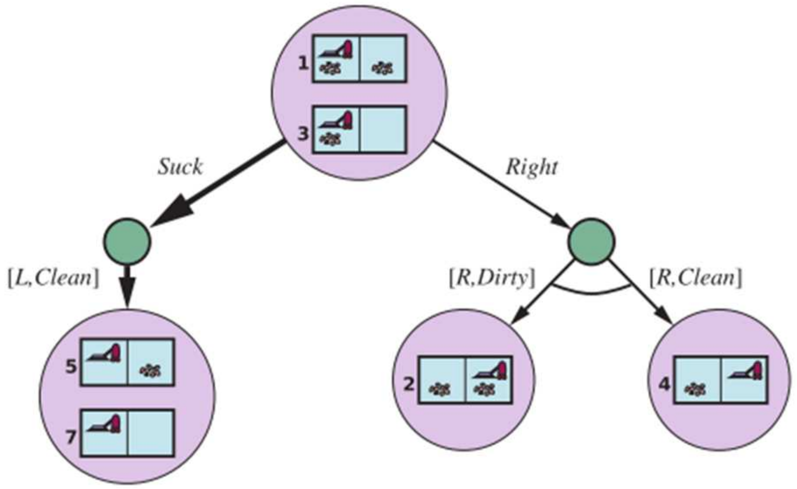




Solving partially observable problems

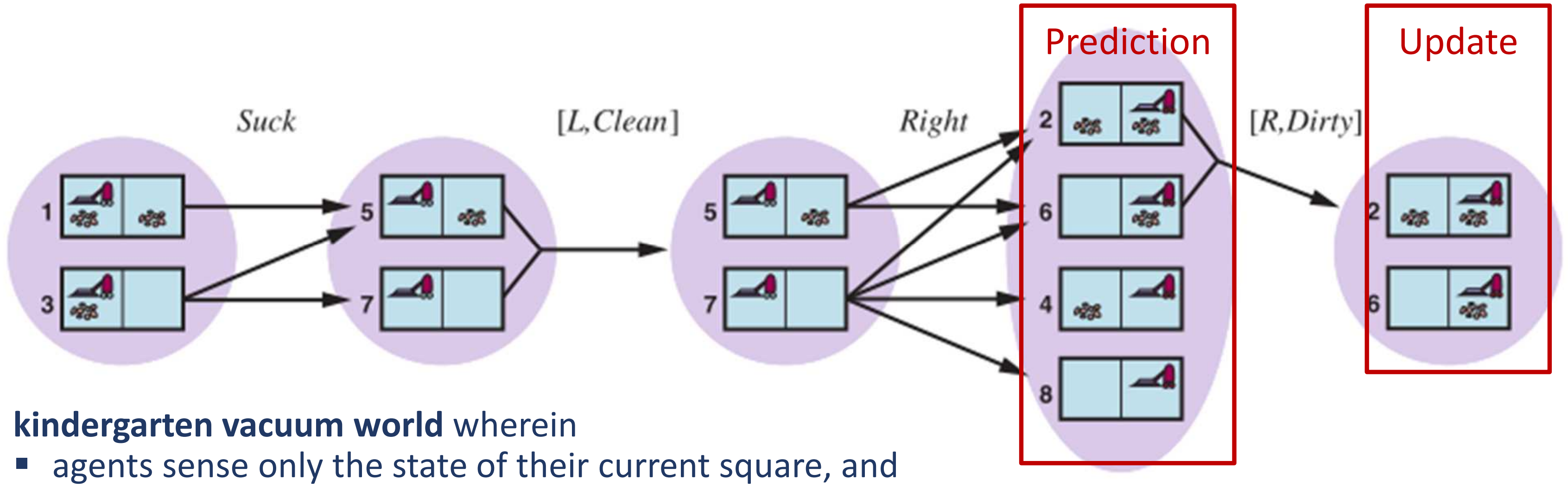
- The AND–OR search algorithm can be applied directly to derive a solution. The figure shows part of the search tree for the local-sensing vacuum world, assuming an initial percept [L, Dirty]
- The solution is the conditional plan

  [Suck,Right, if Rstate={6} then Suck else [ ]]
- It returned a **conditional plan** that tests the belief state
  - That's because we supplied a belief-state problem
  - In this environment the agent won't know the actual state.
- The AND–OR search algorithm treats belief states as black boxes
  - One can improve on this by checking for previously generated belief states that are subsets or supersets of the current state
- Figure shows the first level of the AND–OR search tree for a problem in the local-sensing vacuum world.
  - Suck is the first action in the solution



Solving partially observable problems – e.g. the kindergarten vacuum world

- There are two main differences between this agent and the one for fully observable deterministic environments.
  - First, the solution will be a conditional plan rather than a sequence; to execute an if–then–else expression, the agent will need to test the condition and execute the appropriate branch of the conditional.
  - Second, the agent will need to maintain its belief state as it performs actions and receives percepts
- The agent will do a process of prediction–observation–update
  - Bottom figure shows two **prediction–update** cycles of belief-state maintenance in the vacuum world with local sensing



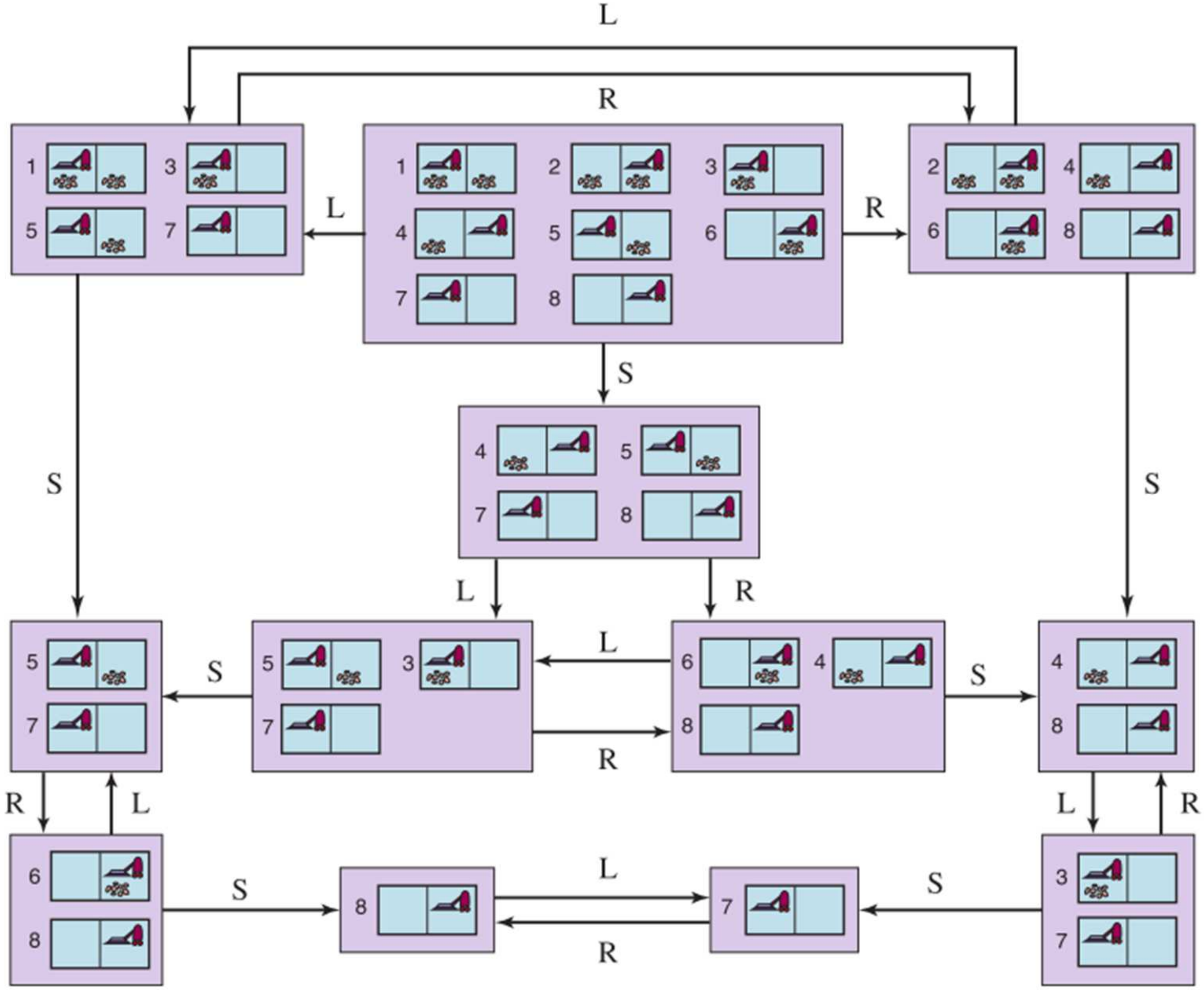
kindergarten vacuum world wherein
- agents sense only the state of their current square, and
- any square may become dirty at any time unless the agent is actively cleaning it at that moment.

Searching with no observation

- When the agent’s percepts provide no information at all, we have what is called a **sensorless problem**
  - Sensorless solutions are surprisingly common and useful
  - Sometimes a sensorless plan is better even when a conditional plan with sensing is available
- In the vacuum world, assume that the agent knows the geography of its world, but not its own location or the distribution of dirt
  - its initial belief state is {1,2,3,4,5,6,7,8}, and when it moves Right it will be in {2,4,6,8}
  - After [Right,Suck] it will end up in {4,8} —the agent has gained information without perceiving anything
  - Finally, after [Right,Suck,Left,Suck] the agent is guaranteed to reach the goal state 7
  - No matter what the start state. We say that the agent can **coerce** the world into state 7
- The **solution to a sensorless problem is a sequence of actions**, not a conditional plan (there is no perceiving).
- We **search in the space of belief states** rather than physical states.
  - In belief-state space, the problem is fully observable because the agent always knows its own belief state.
  - The solution (if any) for a sensorless problem is always a **sequence of actions**
  - This is true even if the environment is nondeterministic

Belief-state space for the deterministic, sensorless vacuum world

- Each rectangular box corresponds to a **single belief state**.
- At any given point, the agent has a belief state but **does not know which physical state** it is in.
- The initial belief state is the complete ignorance, in the case, the top center box.



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.

- However, we need to open the black box to address more problems by using a **factored representation** for each state: a **set of variables**, each of which has 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, {$X_i = V_i$, $X_j = V_j$ . . .}.
  - An assignment that does not violate any constraints is called a **consistent or legal assignment**
  - A **complete assignment** is one in which every variable is assigned a value
  - A **solution** to a CSP is a consistent, complete assignment.
  - A **partial assignment** is one that leaves some variables unassigned
  - A **partial solution** is a partial assignment that is consistent.
- 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.

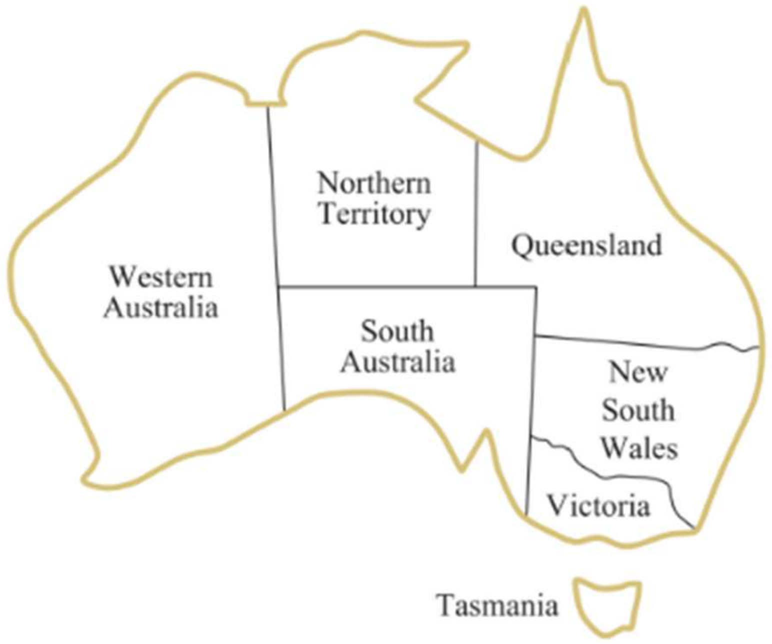
Example 01: Map coloring

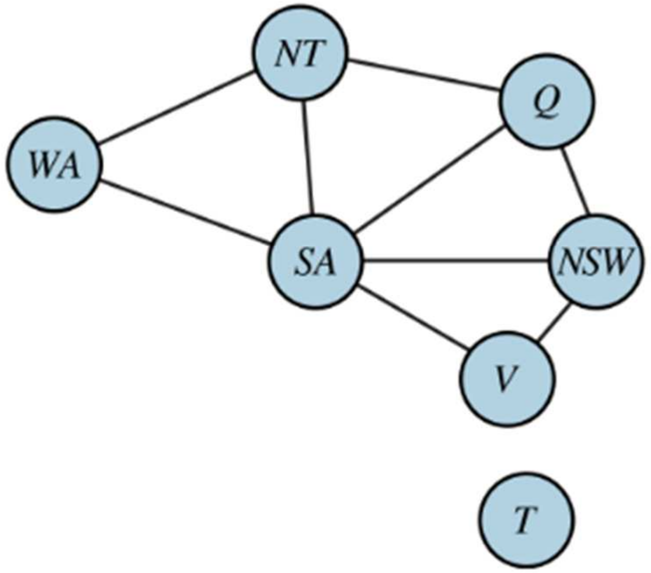
- The task is to color each region of the Australia states and territories either red, green, or blue in such a way that no two neighboring regions have the same color
  - Regions (variables): *X* = { WA, NT, Q, NSW, V, SA, T}

- 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 ≤ WheelRF; AxleF +10 ≤ WheelLF;
  AxleB +10 ≤ WheelRB; AxleB +10 ≤ 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 ≤ NutsRF; NutsRF +2 ≤ CapRF;
  WheelLF +1 ≤ NutsLF; NutsLF +2 ≤ CapLF;
  WheelRB +1 ≤ NutsRB; NutsRB +2 ≤ CapRB;
  WheelLB +1 ≤ NutsLB; NutsLB +2 ≤ 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 ≤ AxleB) or (AxleB +10 ≤ AxleF)

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

  X+dX ≤ Inspect

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

  Di = {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.
- In 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

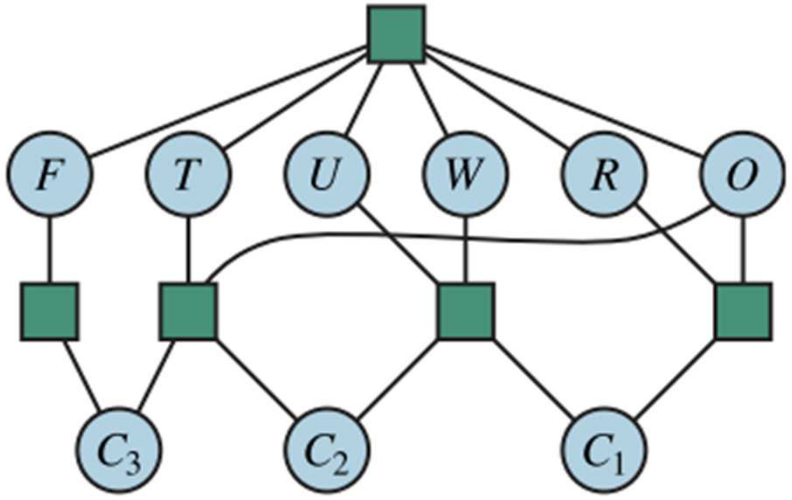
- The simplest type is the **unary constraint**, which restricts the value of a single variable
  - For example, in the map-coloring problem it could be the case that South Australians won’t tolerate the color green; we can express that with the unary constraint «(SA),SA ≠ green».
  - The initial specification 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. R might prefer teaching in the morning, whereas Prof. N 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. R costs 2 points against the overall objective function, whereas a morning slot costs 1.
    - With this formulation, 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 enforcing local consistency in each part of the graph can cause inconsistent values 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.
  - E.g., if South Australians dislike green, the variable SA starts with domain {red, blue},
  - We say that a graph is node-consistent if every variable in the graph is node-consistent.
  - It is easy to eliminate all the unary constraints in a CSP by reducing the its domain at the start of the solving process
- 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. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains.
  - Thus, 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]

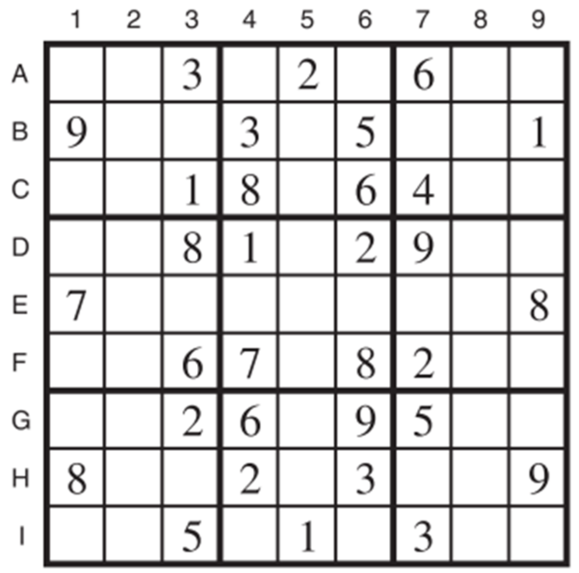
Sudoku game

- 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)

    . . .



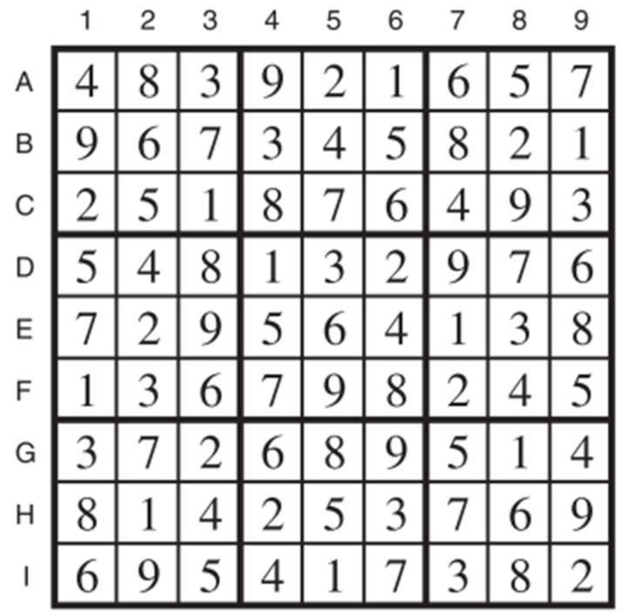
	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

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

  F***or 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 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

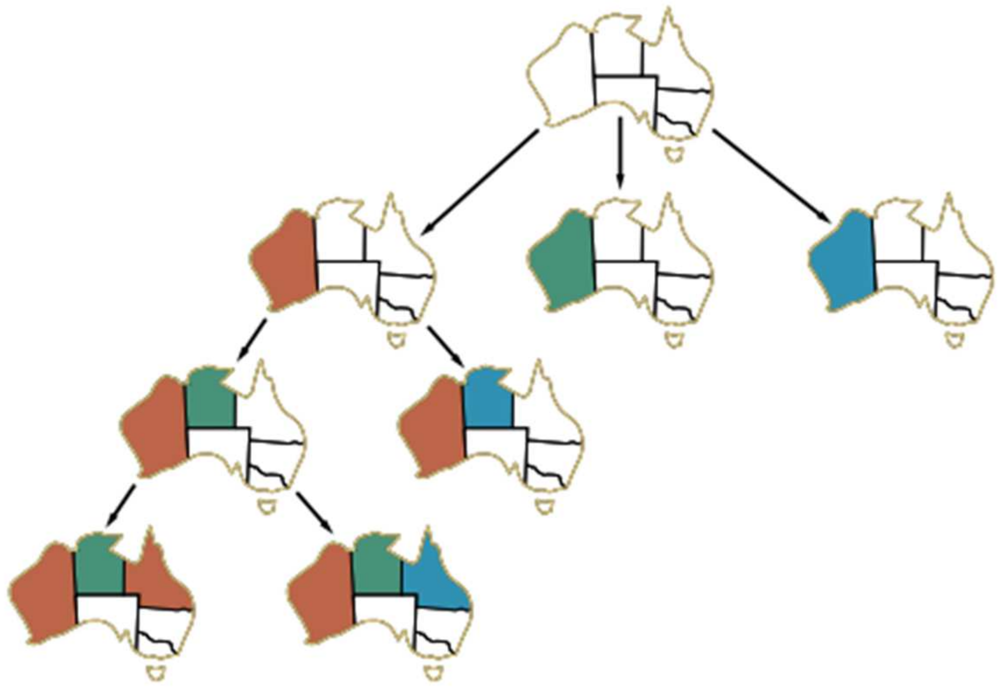
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, {})

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 <u>variable with the fewest "legal" values</u>.
  - 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** comes attempts to reduce the branching factor on future choices by selecting the variable that is <u>involved in the largest number of constraints</u> on other unassigned variables
- The MRV heuristic is usually a more powerful guide, but the degree heuristic can be useful as a <u>tie-breaker</u>

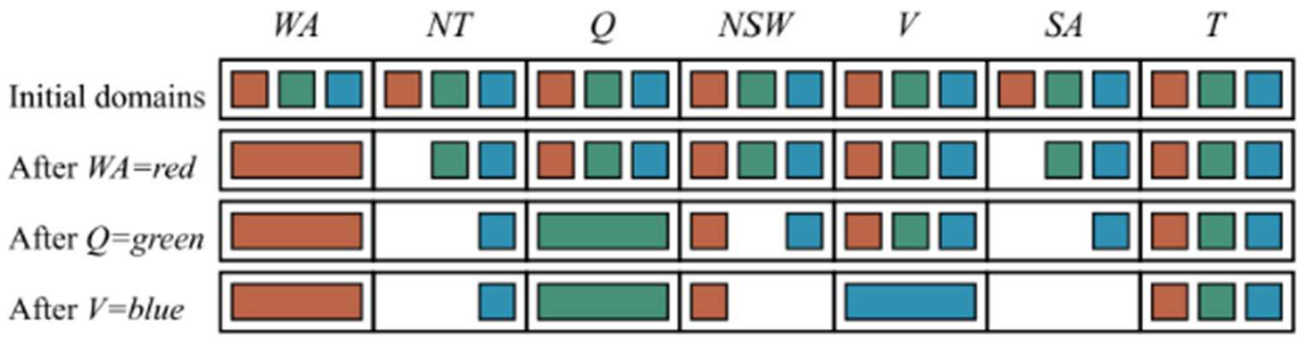
Value ordering

- The **least-constraining-value** 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?
  - 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.

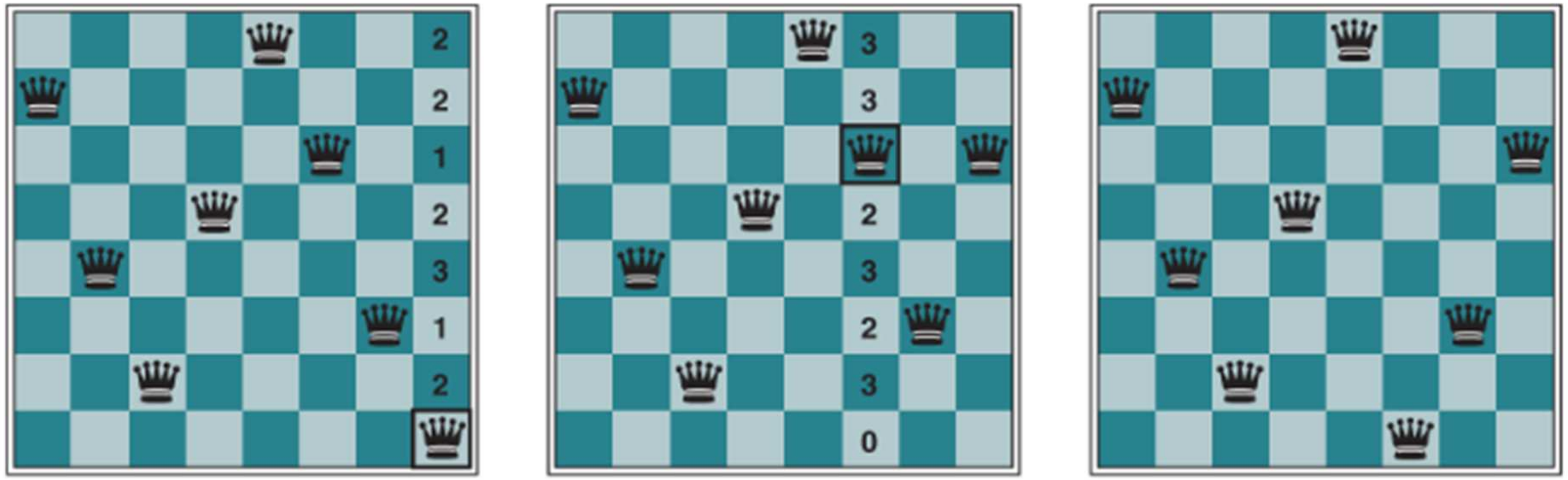
Interleaving search and inference

- **Inference** can reduce the domains of variables during the course of a 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 propagate constraints when changes are made to the variable domains



Local Search for CSPs

- Use 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.



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 = 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!