UNIVERSITÀ
DEGLI STUDI
DI PADOVA

NOTES OF

ARTIFICIAL INTELLIGENCE

SEARCH, CSP, DECISION MAKING, BAYESIAN NETWORKS, PLANNING

(Version 03/07/2021)

Edited by:
Stefano Ivancich

CONTENTS

This document was written by students with no intention of replacing university materials. It is a useful tool for the study of the subject but does not guarantee an equally exhaustive and complete preparation as the material recommended by the University.
The purpose of this document is to summarize the fundamental concepts of the notes taken during the lesson, rewritten, corrected and completed by referring to the to be used as a "practical and quick" manual to consult. There are no examples and detailed explanations, for these please refer to the cited texts and slides.

You can find flashcards of exams questions on: https://www.brainscape.com/p/3YF5P-LH-AUGGM

If you find errors, please report them here:
www.stefanoivancich.com
ivancich.stefano.1@gmail.com
The document will be updated as soon as possible.

1. Agents

1.1. Agents and environments

Agent: a system that perceive its environment through **sensors** and act upon it through **actuators**
Agent function: maps perception histories to actions $f: P^* \to A$
Agent program: runs on the physical architecture to produce f
Performance measure: An objective criterion for success of an agent's behavior
Autonomous agent: if its behavior is determined by its own experience (with ability to learn and adapt)
Rational agent: For each possible percept sequence, it should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.
Ideally: equip an agent with some prior knowledge and the ability to learn

Task environments: the "problems" for which artificial agents are "solutions"
Are specified by **PEAS** (Performance measure, Environment, Actuators, Sensors). Types:

- **Fully observable vs partially observable:** fully means the agents have access to the complete state of the environment at each point in time.
- **Deterministic vs stochastic:**
  - **Deterministic**: next state of the environment is completely determined by the current state and the action executed by the agent.
  - **Stochastic**: there is uncertainty on the next state and it is expressed with probabilities
  - **Non-deterministic:** there is uncertainty on the next state but no probabilities are available.
  - **Uncertain**: not fully observable and non-deterministic
- **Episodic vs sequential:**
  - **Episodic**: agent's experience is divided into atomic "episodes", Each episode consists of the agent perceiving and performing a single action. The choice of action in each episode depends only on the episode itself.
  - **Sequential:** a current decision may affect future decisions
- **Static vs dynamic:**
  - **Static**: environment is unchanged while an agent is deliberating
  - **Dynamic:** continuously asking the agent what it wants to do
  - **Semidynamic**: the environment itself does not change with the passage of time but the agent's performance score does.
- **Discrete vs continuous:** A finite number of distinct, clearly defined states, percepts and actions. Applies also to time.
- **Single agent vs multiagent.** Multiagent can be **competitive** or **cooperative**.
- **Known vs unknown:** depends on the knowledge of the agent or the designer of the agent of the rules governing the environment.
  In a known environment for each action there is an outcome (if deterministic) or a probability distribution over the possible outcomes (if stochastic).
  known and partially observable (for example, a card game)
  unknown and fully observable (a new videogame)

The environment type largely determines the agent design.
The real world is partially observable, stochastic, sequential, dynamic, continuous, multi-agent.

1.2. Agent types

Agent is completely specified by the agent function mapping percept sequences to actions.

Agent = architecture + program

Agent Architecture:

- feeds the percepts from the sensors to the program.
- runs the program.
- feeds the actions to the actuators.

Agent programs we design have the same skeleton.

- They take the current percept as input from the sensors.
- They return an action to the actuators.

Agent program ≠ Agent function

- Agent program: takes the current percept as input.
- Agent function: takes the entire percept history.
- If the agent’s actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

Aim of AI: design agent programs that implement the agent functions concisely.

The TABLE-DRIVEN-AGENT program

- invoked for each new percept
- retains the complete percept sequence in memory
- returns an action each time

```
function TABLE-DRIVEN-AGENT(percept) returns an action
persistent:
    • percepts, a sequence initially empty
    • table, a table of actions, indexed by percept sequences, initially
      fully specified
append percept to the end of percepts
action ← LOOKUP(percepts, table)
return action
```

Drawbacks:

- Invoked for each new percept
- Huge table (chess: 10150 entries)
- Takes a long time to build the table
- Even with learning, need a long time to learn the table entries.
- Who knows how to build it?!

Four basic types of agents in order of increasing generality:

- **Simple reflex agents:** select the action considering only the current percept.
- **Model-based reflex agents:** maintain internal state to track aspects of the world that are not evident in the current percept.
- **Goal-based agents:** act to achieve their goals.
- **Utility-based agents**: try to maximize their own expected “happiness”.

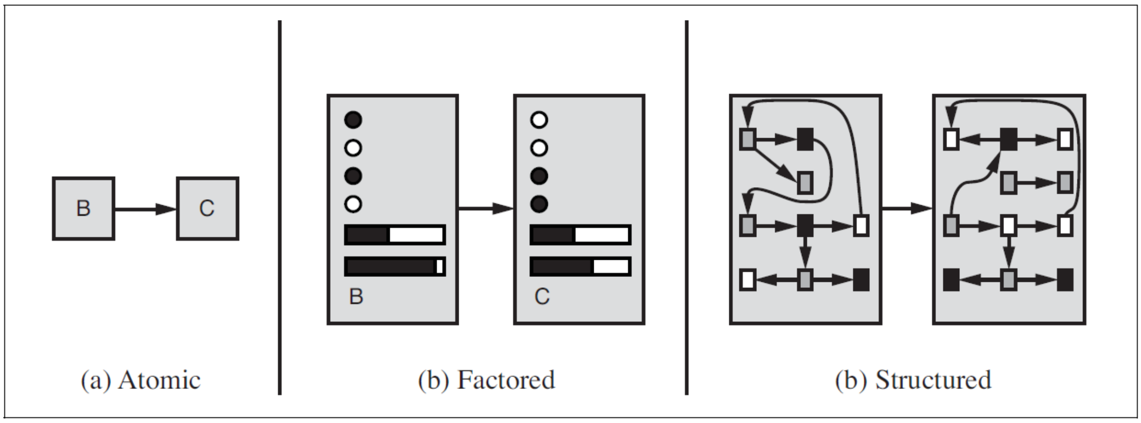
Agent programs consist of various components each can represent the environment in three ways:

- **Atomic representation:** Each state of the world is indivisible, it has no internal structure. Example: the algorithms underlying **search**.
- **Factored representation:** Each state contains a fixed set of variables (or attributes), Each variable can have a value. Two different factored states can share some variables.

Example: **constraint satisfaction** algorithms, planning

- **Structured representation:** Each state contains objects (with variables with values) and relations with other objects, Example: knowledge-based learning, natural language understanding.

A more expressive representation can capture at least as concisely everything a less expressive one can capture plus some more.



2.Solving problems by searching

2.1. Problem-solving agents

Intelligent agents are supposed to maximize their performance measure. Achieving this is sometimes simplified if the agent can select a goal and aim to satisfy it.

Agent: Problem-solving agents (is a Particular type of goal-based agent)
Representation: atomic (the states have no internal structure visible to the problem-solving algorithm)
Environment:

- observable
- known
- deterministic

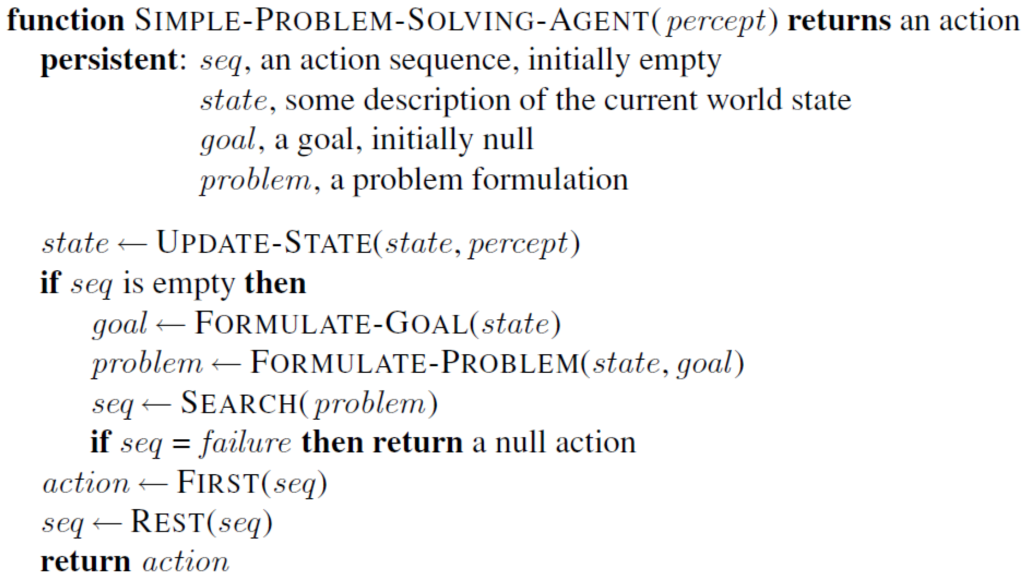
Solution: fixed sequence of actions
Search: process of looking for such a sequence
Search algorithm:

- input: problem
- output: an action sequence

The sequence in output can then be executed without worrying about the environment.

The agent execute repeatedly:

- Formulates a goal and a problem.
- Searches for a sequence of actions that would solve the problem.
- Executes the actions one at a time.



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

2.2. Problem formulation

A problem can be defined formally by:

- **Initial state:** of the agent
- **Actions available** to the agent
    - Given a state s, ACTIONS(s) returns the set of actions that can be executed in s
    - We say that each of these actions is applicable in s
- **Transition model:** description of what each action does.
    - RESULT(s,a) returns the state obtained from doing action a in state s
    - **Successor**: any state reachable from a given state by a single action
- **Goal test:** allows to check if a state is a goal.
- **Path cost:** numeric value associated to each path reflecting the desired performance measure. We assume path costs to be additive: sum of step costs.
  Step cost $c(x, a, y)$: for going from state x to state y by performing action a. Assumed to be ≥ 0.

Solution: a sequence of actions (path) leading from the initial state to a goal state.
Optimal solution: a solution with minimal path-cost.

Problem state space = (initial state, actions, transition model)
Is the set of all states reachable from the initial state by any sequence of actions.
can be depicted as a directed graph:

- nodes: states
- edges: actions
- path: sequence of states connected by a sequence of actions.

Abstraction: process of removing details.
Real world is absurdly complex so the state space must be abstracted for problem solving.

- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
- (Abstract) solution = set of real paths that are solutions in the real world

Valid abstraction: if we can expand any abstract solution into a solution in the more detailed world.

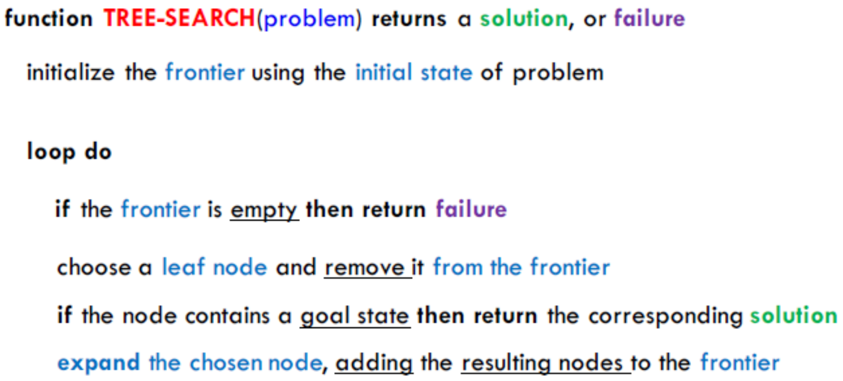
2.3. Uninformed search algorithms

They have no information about the problem other the definition.
Possible sequences of actions from initial state form a search tree:

- Root: initial state
- Nodes: states
- Branches: actions

The same state can appear multiple time.
Outgoing edges from a node corresponds to all possible actions available in the state represented by the node.

Tree search algorithm:

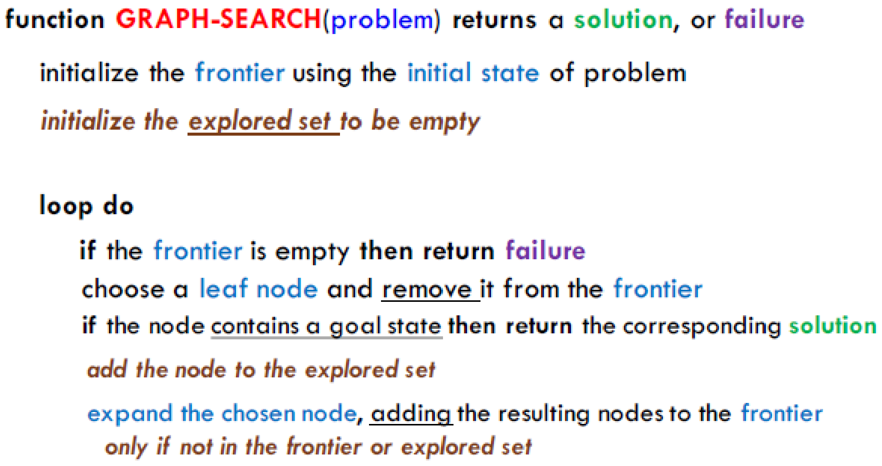


```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem

  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

Leaf node: a node with no children in the tree
Frontier: The set of all leaf nodes available for expansion at any given point
Redundant paths: whenever there is more than one way to get from one state to another.
Loopy paths: special case of redundant paths.

To avoid exploring redundant paths TREE-SEARCH algorithm is augmented with **explored set** that remembers every expanded node.
Graph search algorithm:



```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty

  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

2.3.1. Infrastructure for search algorithms

For each node n of the tree, we have a structure with:

- `n.STATE`: the state in the state space to which the node corresponds
- `n.PARENT`: the node in the search tree that generated this node
- `n.ACTION`: the action that was applied to the parent to generate the node
- `n.PATH-COST`: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

A state corresponds to a configuration of the world.
A node is a data structure used to represent the search tree.

Nodes are stored in a queue whose operations are: `isEmpty`, `pop`, `insert`.
Three common queue variants:

- FIFO queue: which pops the oldest element of the queue
- LIFO queue: which pops the newest element of the queue
- Priority queue: which pops the element of the queue with the highest priority according to some ordering function

Search strategy is defined by picking the order of node expansion
Performances are measured:

- **completeness:** does it always find a solution if one exists?
- **time complexity:** number of nodes generated/expanded
- **space complexity:** maximum number of nodes in memory
- **optimality**: does it always find a least-cost solution?

Time and space complexity are measured in terms of

- $\boldsymbol{b}$: branching factor of the search tree (i.e., maximum number of successors of any node)
- $\boldsymbol{d}$: depth of the least-cost solution
- $\boldsymbol{m}$: maximum depth of the state space

2.4. Uninformed Search strategies

Uninformed strategies use only the information available in the problem definition.

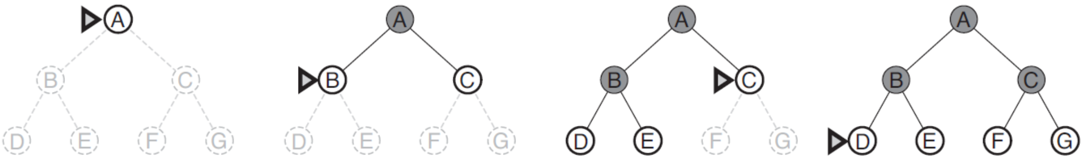
2.4.1. Breadth-first search

Expand shallowest unexpanded node.

Implementation: frontier is a **FIFO** queue, i.e., new successors go at end
Goal test is applied to each node when it is generated rather than when it is selected for expansion.

Properties:

- Completeness: only if b is finite
- Optimality: only if cost = 1 per step
- Time: $O\left(\boldsymbol{b^d}\right)$
- Space: $O\left(\boldsymbol{b^d}\right)$ keeps every generated node in memory



2.4.2. Uniform-cost search

Expand **least-cost** unexpanded node. Equivalent to breadth-first if step costs all equal.
Implementation:

- frontier = priority queue ordered by path cost $g(x)$
- goal test performed at expansion time

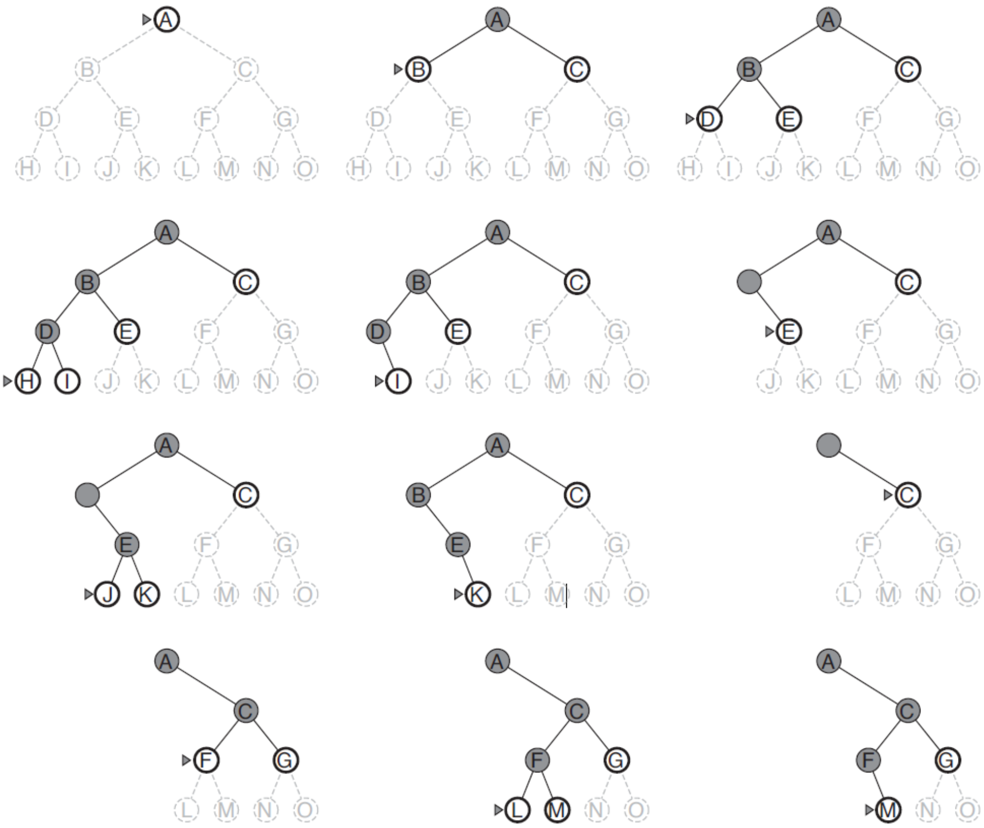
Properties:

- Completeness: if stepcost $\geq \epsilon$
- Optimality: Yes, nodes expanded in increasing order of $g(n)$
- Time, Space: $O\left(b^{\lfloor C^*/\epsilon \rfloor + 1}\right)$ where C^* is the cost of the optimal solution and each action costs at least ϵ

2.4.3. Depth-first search

Expand deepest unexpanded node.
Implementation: frontier = **LIFO** queue, i.e., put successors at front
Properties:

- Completeness: only in finite state spaces
- Optimality: No
- Time: $O(\boldsymbol{b^m})$, terrible if m is much larger than d
- Space: $O(\boldsymbol{bm})$



2.4.4. Depth-limited search

To avoid the problem of the illimited trees, we use depth-first search with depth limit L.

Properties:

- Completeness: only if $L \geq D$
- Optimality: No
- Time: $O(\boldsymbol{b^l})$
- Space: $O(\boldsymbol{bl})$

Can terminate with two kinds of failure: no solution exists or no solution within the depth limit.

Incomplete if $l < d$ (the shallowest goal is beyond the depth limit)

Non-optimal if $l > d$

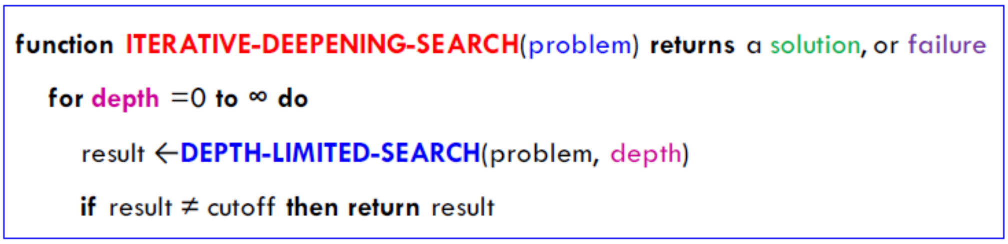
2.4.5. Iterative deepening search

It repeatedly applies depth-limited search with increasing limits

It terminates if the depth-limited search returns: solution, or failure, meaning that no solution exists.

Properties:

- Completeness: only if b is finite
- Optimality: only if step cost = 1
- Time: $O(\boldsymbol{b^d})$
- Space: $O(\boldsymbol{bd})$



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth =0 to ∞ do
      result ←DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Uninformed search summary:

Breadth-first search expands the shallowest nodes first

- Complete (if b finite)
- Optimal for unit step costs
- But exponential space complexity

Uniform-cost search expands the node with lowest path cost, $g(n)$

- Optimal for general step costs

Depth-first search expands the deepest unexpanded node first

- Neither complete nor optimal
- But linear space complexity
- Depth-limited search adds a depth bound

Iterative deepening search calls depth-first search with increasing depth limits until a goal is found

- Complete (if b finite)
- Optimal for unit step costs
- Time complexity comparable to breadth-first search
- Linear space complexity

2.5. Informed search algorithms

Use problem-specific knowledge to speed up the search process.

Best-first search: Node is chosen for expansion based on an **evaluation function $\boldsymbol{f(n)}$** that is a cost estimate, so is expanded the node with **lowest $\boldsymbol{f(n)}$** first.
A component of f is **Heuristic function $\boldsymbol{h(n)}$** = estimate of cost of the cheapest path from the state of the node n to a goal state.

- depends only on the state associated with n
- $h(n)$ is non negative
- $h(n) = 0$ at every goal state

Special cases of Best first search: greedy, A*

2.5.1. Greedy best-first search

Expands the node that appears to be closest to goal. $f(n) = h(n)$

Properties:

- Completeness: No, can be trapped in dead ends.
- Optimality: No
- Time: $O(\boldsymbol{b^m})$, but a good heuristic can give dramatic improvement
- Space: $O(\boldsymbol{b^m})$, keeps all nodes in memory

2.5.2. A* search

Avoid expanding paths that are already expensive.
$f(n) = g(n) + h(n)$

- $g(n)$ = path cost from the start node to node n
- $h(n)$ = estimated cost of the cheapest path from n to goal
- $f(n)$ = estimated cost of the cheapest solution through n

We expand the node with lowest $f(n)$

Properties:

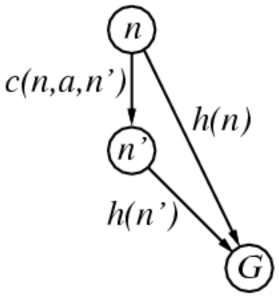
- Completeness: yes, unless there are infinitely many nodes with $f \leq f(G)$
- Optimality: Yes
- Time: $O(\boldsymbol{b^m})$
- Space: $O(\boldsymbol{b^m})$, keeps all nodes in memory

2.5.3. Heuristics

Admissible Heuristics: h is admissible if for every node n $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost form n to goal.
It never overestimates the cost to reach the goal.

If h is admissible, A* using TREE-SEARCH is optimal. (Proof lecture 7)

Consistent heuristics: h is consistent if for every node n, for every successor n' of n generated by an action a, $h(n) \leq c(n, a, n') + h(n')$



Consistency implies admissibility.

If h is consistent, A* using TREE-SEARCH is optimal.

h_2 **dominates** h_1 if $h_2(n) \geq h_1(n)$ for all node n

h_2 is better than h_1 for search: this means that h_1 will expand all the nodes h_2 expands and possibly more.

Relaxed problem: a problem with fewer restrictions on the actions.

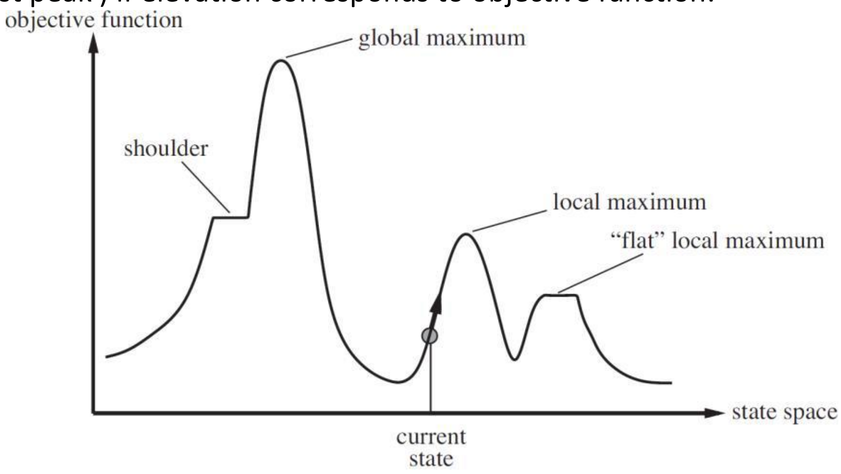
- A superset of actions are available from each state
- The graph of the relaxed problem is a supergraph of the original
- Any optimal solution of the original problem is also a solution of the relaxed problem.
- The cost of an optimal solution of the relaxed problem may be the same or lower of the cost of an optimal solution of the original problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.

2.6. Local search algorithms

In many optimization problems the path from the start node to the goal is irrelevant we care just about the goal state.
They keep a single node and move to adjacent nodes.
Useful for solving optimization problems, where the goal is to find the best state according to an objective function.

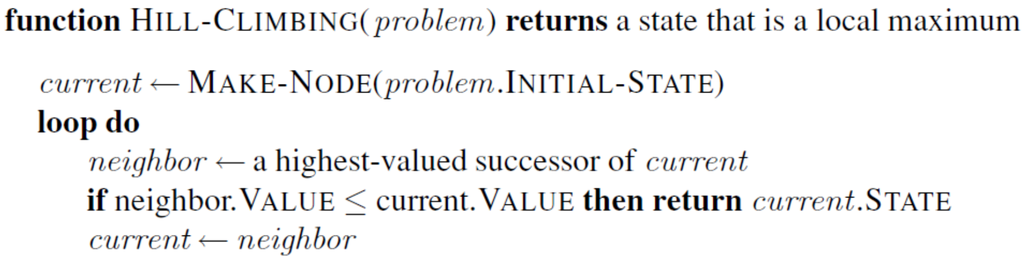
State space = set of "complete" configurations
Find configuration satisfying constraints.

Local search algorithms explore the state-space landscape.
State-space landscape: it has a location (defined by the state) and an elevation (defined by the value of heuristic cost function or objective function).
The aim is to find: a global minimum (lowest valley) if elevation corresponds to cost or a global maximum (highest peak) if elevation corresponds to objective function.



2.6.1. Hill-climbing search

Assume the elevation corresponds to the objective function.
Modifies the current state to try to improve it.
At each steps: Picks a neighbor with the highest value. (Usually chooses at random among neighbors with maximum value)
Terminates when it reaches a "peak" where no neighbor has a higher value.



```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Problem: often gets stuck in

- Local maxima: a peak that is higher than each of its neighboring states but lower than the global maximum.
- Plateau: a flat area of the state-space landscape. Flat local maximum, from which no progress is possible or a shoulder, from which progress is possible.

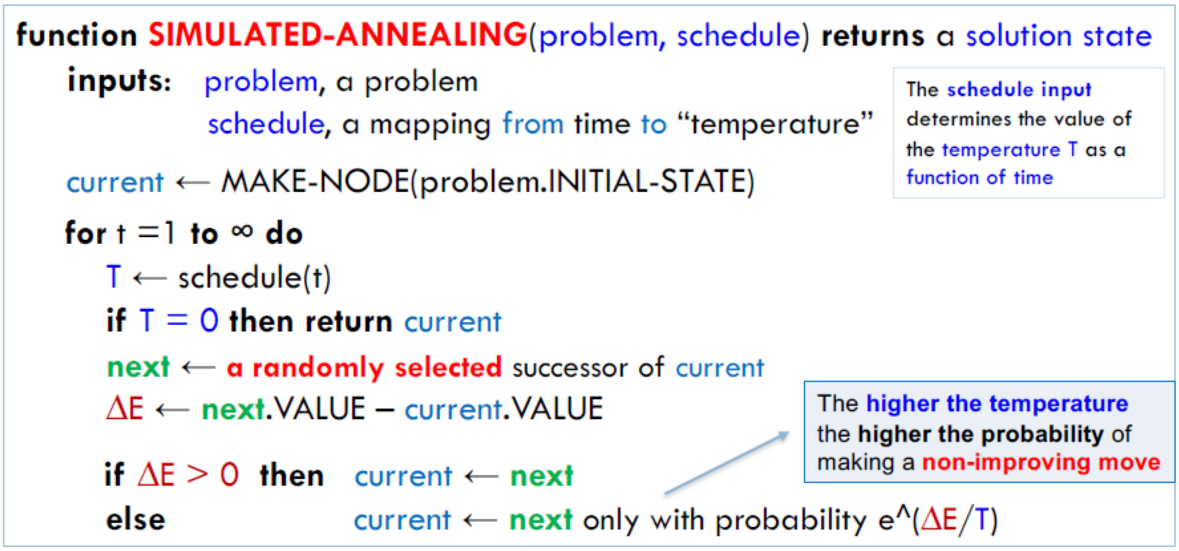
Variants:

- **Stochastic hill climbing:** chooses at random from the set of all improving neighbors
- **First-choice hill climbing:** jumps to the first improving neighbor found.
- **Random-restart hill climbing:** series of hill climbing runs until a goal is found. It will find a good solution very quickly.

Hill-climbing algorithm that never makes "downhill" moves toward states with lower value is incomplete because it can get stuck on a local maximum.
Purely random walk: moving to a successor chosen uniformly at random from the set of successors is complete but extremely inefficient.
Simulated annealing idea: to combine hill climbing with a random walk that yields both efficiency and completeness.

2.6.2. Simulated annealing search

Version of stochastic hill climbing where some downhill moves are allowed.
If the move improves the situation is always accepted, otherwise is accepted with some probability less than 1 that decrease exponentially over time.
Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency.



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    current ← MAKE-NODE(problem.INITIAL-STATE)
    for t =1 to ∞ do
        T ← schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.VALUE – current.VALUE
        if ΔE > 0  then  current ← next
        else             current ← next only with probability e^(ΔE/T)
```

The schedule input determines the value of the temperature T as a function of time

The higher the temperature the higher the probability of making a non-improving move

One can prove: If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.

2.6.3. Local beam search

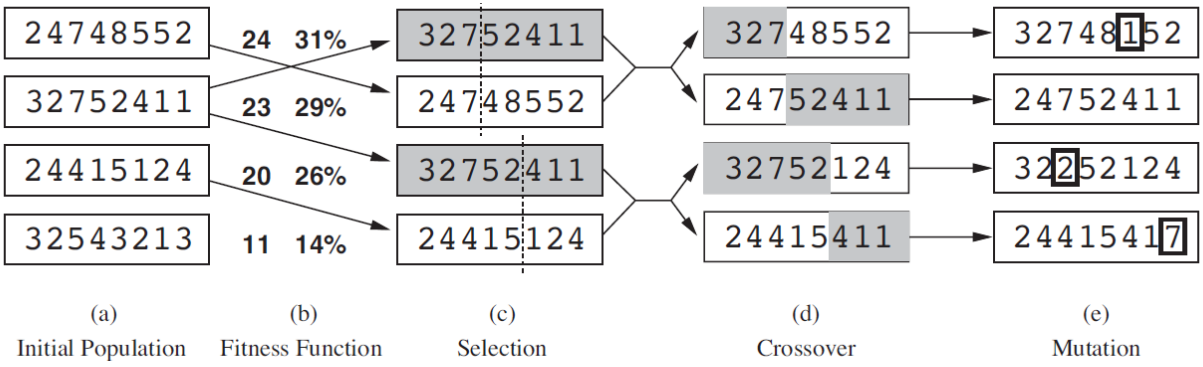
Keep track of k states rather than just one.

- Start with k randomly generated states.
- At each iteration all the successors of all k states are generated
- If any one is a goal state the algorithm stops, Else select the k best successors from the complete list and repeat (they could be all successors of the same node).

2.6.4. Genetic algorithms

Beam search variant, that is based on 5 steps:

- **Initial population:** Start with k randomly generated states (population). Every state is represented as a string over a finite alphabet (often a string of 0s and 1s or digits).
- **Fitness function:** Each state is associated to a value via an evaluation function. Returns higher values for better states.
- **Selection:** random choice of 2 pairs on the basis of the fitness function.
- **Crossover:** For each pair to be mated, a crossover point is chosen randomly from the positions in the string.
- **Mutation:** Each location is subject to random mutation with a small independent probability.



2.7. Stable matching problem

Multi agent system: collection of multiple intelligent agents which interact.
Ideally, we would like for the intelligent agents which we have designed so far to be able to communicate and interact with other agents.
We take inspiration from groups of humans: economics, game theory, and social choice.

Computational social choice: an interdisciplinary field at the interface of: Artificial intelligence, Economics, Voting theory, Game Theory, Social Choice.
We will see one problem in this area: matching problems, that are a mathematical abstraction of two-sided markets.

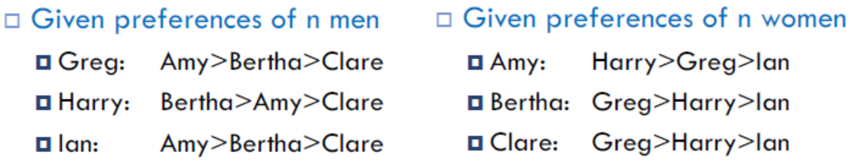
Stable Matching Problems

- Two sets of agents
- **Agents** of one set express preferences over agents of the other set
- **Goal**: to choose a matching among the agents of the two sets based on their preferences
  Matching: set of pairs (A_1, A_2), where A_1 comes from the first set and A_2 comes from the second set

Practical scenarios: Matching students with schools, Matching doctors with hospitals, Matching kidney donors and patients, Matching sailors to ships, Job hunting.

2.7.1. Stable Marriage formulation

Two sets of agents: men and women
Idealized model: Same number of men and women, All men totally order all women, and viceversa

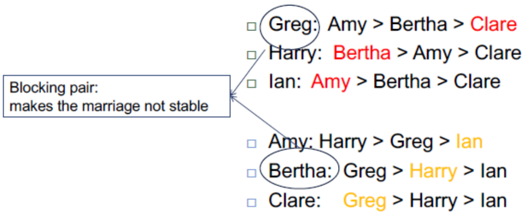


□ Given preferences of n men
- Greg: Amy>Bertha>Clare
- Harry: Bertha>Amy>Clare
- Ian: Amy>Bertha>Clare

□ Given preferences of n women
- Amy: Harry>Greg>Ian
- Bertha: Greg>Harry>Ian
- Clare: Greg>Harry>Ian

Marriage: is a one-to-one correspondence between men and women
Idealization: everyone marries at the same time
Blocking pair: pair (m, w), where m is a man and w is a woman such that:

- the marriage contains (m, w') and (m', w), but
- m prefers w to w'
- w prefers m to m'



Blocking pair:
makes the marriage not stable

□ Greg: Amy > Bertha > Clare
□ Harry: Bertha > Amy > Clare
□ Ian: Amy > Bertha > Clare

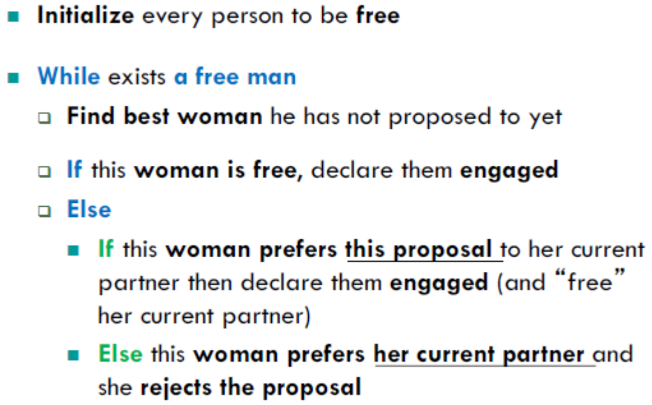
□ Amy: Harry > Greg > Ian
□ Bertha: Greg > Harry > Ian
□ Clare: Greg > Harry > Ian

Stable Marriage: a marriage with no pair (man, woman) not married to each other that would prefer to be together. OR marriage with no blocking pairs.
Idealization: assumes no cost in breaking a match

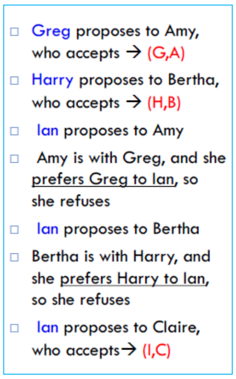
Given any stable marriage problem:

- There is at least one stable marriage.
- There may be many stable marriages especially in large AI domains.

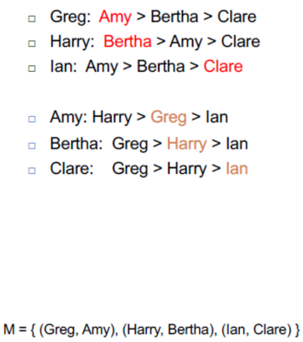
2.7.2. Gale Shapley algorithm



- **Initialize** every person to be **free**
- **While** exists **a free man**
  - **Find best woman** he has not proposed to yet
  - **If** this **woman is free,** declare them **engaged**
  - **Else**
    - **If** this **woman prefers this proposal** to her current partner then declare them **engaged** (and "free" her current partner)
    - **Else** this **woman prefers her current partner** and she **rejects the proposal**



- □ Greg proposes to Amy, who accepts → (G,A)
- □ Harry proposes to Bertha, who accepts → (H,B)
- □ Ian proposes to Amy
- □ Amy is with Greg, and she prefers Greg to Ian, so she refuses
- □ Ian proposes to Bertha
- □ Bertha is with Harry, and she prefers Harry to Ian, so she refuses
- □ Ian proposes to Claire, who accepts → (I,C)

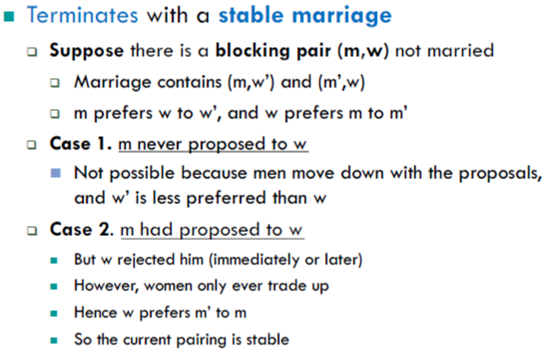


- □ Greg: Amy > Bertha > Clare
- □ Harry: Bertha > Amy > Clare
- □ Ian: Amy > Bertha > Clare
- □ Amy: Harry > Greg > Ian
- □ Bertha: Greg > Harry > Ian
- □ Clare: Greg > Harry > Ian

M = { (Greg, Amy), (Harry, Bertha), (Ian, Clare) }

Properties:

- Time $O(n^2)$: Each of n men can make at most n proposals.
- Terminates with everyone married.
- **Finds man optimal solution**: There is no stable matching in which any man does better.
- **Finds woman pessimal solution:** In all stable marriages, every woman does at least as well or better.
- **Terminates with a stable marriage**. Proof:



- Terminates with a **stable marriage**
  - **Suppose** there is a **blocking pair (m,w)** not married
    - Marriage contains (m,w') and (m',w)
    - m prefers w to w', and w prefers m to m'
  - **Case 1.** m never proposed to w
    - Not possible because men move down with the proposals, and w' is less preferred than w
  - **Case 2.** m had proposed to w
    - But w rejected him (immediately or later)
    - However, women only ever trade up
    - Hence w prefers m' to m
    - So the current pairing is stable

Other stable marriages: Other algorithms find "fairer" marriages.

Ex.: stable marriage which minimizes the maximum Regret. (regret of a man/woman = distance between his partner in the marriage and his most preferred woman/man)

2.7.3. Extensions

Extensions: Ties in preferences

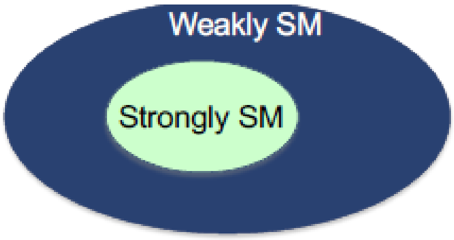
Preference orderings: total orders with ties

Stability

- weakly stable marriage: no un-matched couple such that each one strictly prefers the other to the current partner.
- strongly stable marriage: no un-matched couple such that one strictly prefers the other, and the other likes it as much or more as the current partner.

Existence

- Strongly stable marriage may not exist
  $O(n^4)$ algorithm for deciding existence
- Weakly stable marriage always exists
  Just break ties arbitrarily, Run GS, resulting marriage is weakly stable.
  Polynomial complexity



Extensions: Incomplete preferences

- Model **unacceptability of an option**
- More possible blocking pairs
- (m, w) blocking pair if
    - m and w are unmatched and do not find each other unacceptable, or
    - m, w both prefer each other to current partners, or
    - one of the two is matched but acceptable to the other and prefers the other who is unmatched
- GS algorithm Extends easily, Polynomial complexity
- The set of unmatched elements is the same in every stable marriage

Extensions: ties & incomplete prefs

Weakly stable marriages may have different sizes, Unlike with just ties where they are all complete Or with just incompleteness where the cardinality is fixed.

Finding weakly stable marriage of maximal cardinality is NP-hard. Even if only men declare ties Ties are of most of length two, The whole list is a tie.

Strategy proofness

GS is **strategy proof** (that is, non-manipulable) for men

- Assuming male optimal algorithm
- No man can do better than the male optimal solution

However, **women can profit from lying** (that is, women can obtain a better partner by expressing different preferences from the true ones)

- Assuming male optimal algorithm is run
- Assuming they know complete preference lists

GS can be manipulated: Every stable marriage procedure (that is, every procedure that returns a stable marriage) can be manipulated if preference lists can be incomplete.

3. Constraint satisfaction problems

In **standard search problems**, **states** are:

- Atomic ("black box" with no internal structure)
- Evaluated by domain-specific heuristics.
- Tested to see whether they are goal states.

In **CSPs**: a **factored representation** for each state

- **State** is defined by variables X_i with values from domain D_i
- **Goal** test is a set of constraints specifying allowable combinations of values for subsets of variables.
- **Solution**: one value for each variable that satisfies all the constraints

Allows useful general-purpose algorithms to solve complex problems

- with more power than standard search algorithms
- **general-purpose heuristics** rather than problem-specific heuristics

Algorithms for solving CSPs: eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

3.1. Problem Definition

CSP Definition:

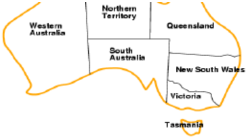
- Set of **variables** $X = \{X_1, \dots, X_n\}$
- Set of **domains** $D = \{D_1, \dots, D_n\}$
  - D_i consists of a set of allowable values for variable X_i.
  - In many cases the domain is assumed to be the same for all variables
- Set of **constraints** $C = \{c_i = (\text{scope}_i, \text{rel}_i) | i = 1, \dots, h\}$
  - scope_i: subset of X, the variables that are constrained by c_i
  - rel_i: is a relation and tells us which simultaneous assignments of values to variables in scope_i are allowed.

State: defined by an assignment of values to some or all of the variables.
Assignment can be:

- **Consistent**: it does not violate any constraints
- **Complete**: every variable is assigned
- **Partial**: only some of the variables are assigned

Solution: a consistent and complete assignment

Coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.



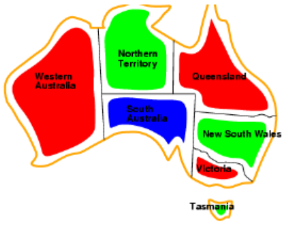
CSP formulation

□ Set of variables X = {WA, NT, Q, NSW, V, SA, T}

□ Domain of each variabile D_i = {red, green, blue}

□ Constraints: **adjacent** regions must have **different colors**
C = {SA ≠ WA, SA ≠ NT, SA ≠ Q, SA ≠ NSW, SA ≠ V,
WA ≠ NT, NT ≠ Q, Q ≠ NSW, NSW ≠ V }

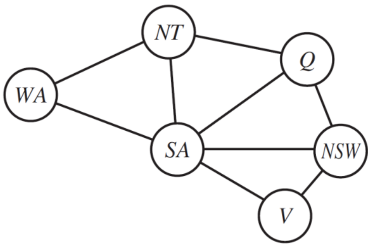
WA ≠ NT, or
(WA,NT) in
{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}



□ Solutions are complete and consistent assignments

e.g., {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green}

Constraint graph: nodes are variables, arcs are constraints



CSP solvers are faster than state-space searchers because the CSP solver can quickly eliminate large parts of the search space.

Types of variable:

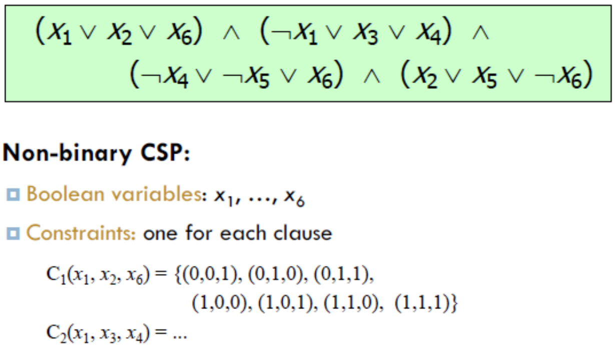
- **Discrete variables**
  - **Finite domains:** n variables, domain size d
    (e.g., variables WA, NT, Q, NSW, V, SA, T in the map coloring problem and each variable has the domain Di = {red, blue, green})
  - **Infinite domains:** integers, strings, etc.
    (e.g., job scheduling, variables are start/end days for each job n constraints: StartJob1 + 5 ≤ StartJob3)
- **Continuous variables:** common in the real world problems, studied in the field of operations research.

Types of constraints:

- **Unary**: involve a single variable
  e.g., SA ≠ green
- **Binary**: involve pairs of variables
  e.g., SA ≠ WA
- **Higher-order**: involve 3 or more variables
- **Global**: involve an arbitrary number of variables
  e.g., Alldiff, which says that all of the variables involved in the constraint must have different values.

Binary CSP: CSP where each constraint relates two Variables.
Any CSP can be converted into a CSP with only binary constraints.

Example of formulations:



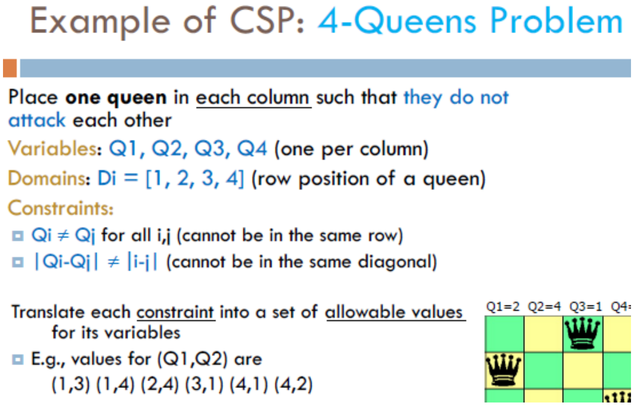
$$(x_1 \vee x_2 \vee x_6) \ \wedge \ (\neg x_1 \vee x_3 \vee x_4) \ \wedge$$
$$(\neg x_4 \vee \neg x_5 \vee x_6) \ \wedge \ (x_2 \vee x_5 \vee \neg x_6)$$

Non-binary CSP:

- Boolean variables: $x_1, \ldots, x_6$
- Constraints: one for each clause

$C_1(x_1, x_2, x_6) = \{(0,0,1), (0,1,0), (0,1,1),$
$(1,0,0), (1,0,1), (1,1,0), (1,1,1)\}$

$C_2(x_1, x_3, x_4) = \ldots$



Example of CSP: 4-Queens Problem

Place **one queen** in <u>each column</u> such that they do not attack each other

Variables: Q1, Q2, Q3, Q4 (one per column)

Domains: Di = [1, 2, 3, 4] (row position of a queen)

Constraints:

- Qi ≠ Qj for all i,j (cannot be in the same row)
- |Qi-Qj| ≠ |i-j| (cannot be in the same diagonal)

Translate each constraint into a set of allowable values for its variables

- E.g., values for (Q1,Q2) are
  (1,3) (1,4) (2,4) (3,1) (4,1) (4,2)

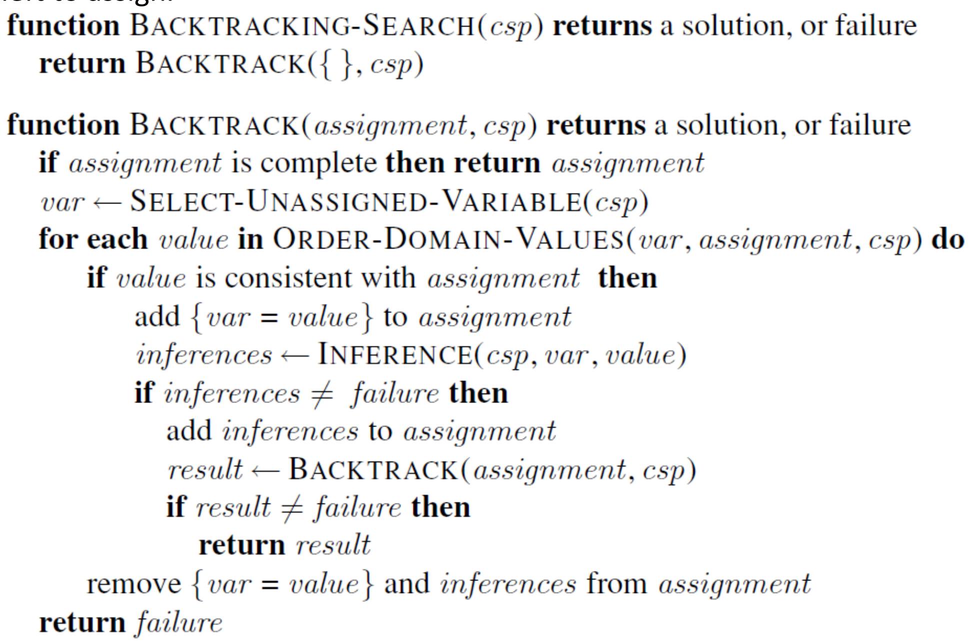
3.2. Backtracking search

Standard search formulation:

- **Initial state:** the empty assignment {}
- **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment. fail if no legal assignments
- **State:** partial assignment
- **Goal test:** the current assignment is complete

Every solution appears at depth n with n variables.
But if we use DFS, for a CSP with n variables of domain size d we are going to generate a tree with $\boldsymbol{n! * d^n}$ **leaves** even if only d^n possible complete assignments. **NOT feasible.**

In CSPs, **variable assignments are commutative**, i.e., {WA = red then NT = green } same as { NT = green then WA = red }
Thus, we need only consider a **single variable at each level** in the search tree, so the **number of leaves is $\boldsymbol{d^n}$**
Backtracking search: Depth-first search for CSPs with single variable assignments.
It chooses values for a not assigned variable at each level and backtracks when a variable has no legal values left to assign.



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

function BACKTRACK(assignment, csp) returns a solution, or 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)
        if inferences ≠ failure then
          add inferences to assignment
          result ← BACKTRACK(assignment, csp)
          if result ≠ failure then
            return result
     remove {var = value} and inferences from assignment
  return failure
```

Improving backtracking efficiency:

- Minimum remaining values **(MRV) heuristic:** chooses the variable with the fewest legal values. It will fail earlier.
- **Degree heuristic:** chooses the variable involved in the most constraints with unassigned variables. Reduce the branching factor.
- Given a variable, chooses the **least constraining value:** the one that rules out the fewest values in the remaining Variables. Succeed first.

The ordering of values does not matter if all solutions needed and no solution, because we have to consider every value.

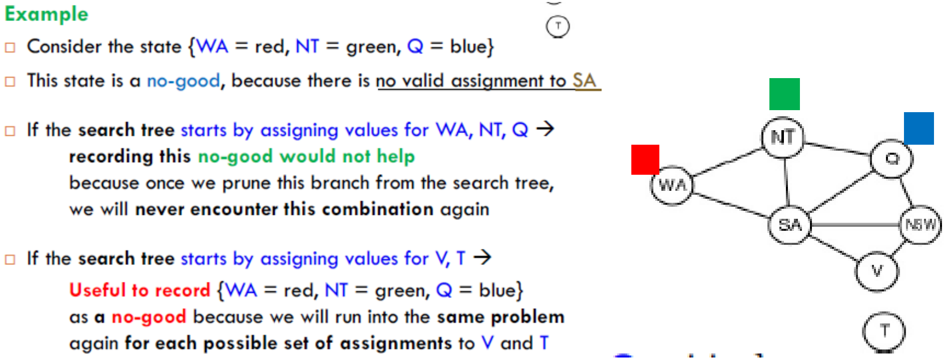
Chronological backtracking: Backtrack to the previous variable and try another value.

Backjumping: Backtrack to a variable that might fix the problem.
The set of these variables is called the **conflict set**.
Backtrack to the most recent variable in the conflict set.
Example:

- Assume variable ordering Q, NSW , V , T , SA, WA, NT
- The conflict set for SA is {Q, NSW, V}
- We jump over T and try a new value for V

No-good: To avoid redundant work (To avoid running into the same problem again)

- **Constraint learning:** finding a minimum set of variables from the conflict set that cause the problem. This set of variables with their corresponding values is called no-good.
- Record the no-good by adding a new constraint to the CSP



Example

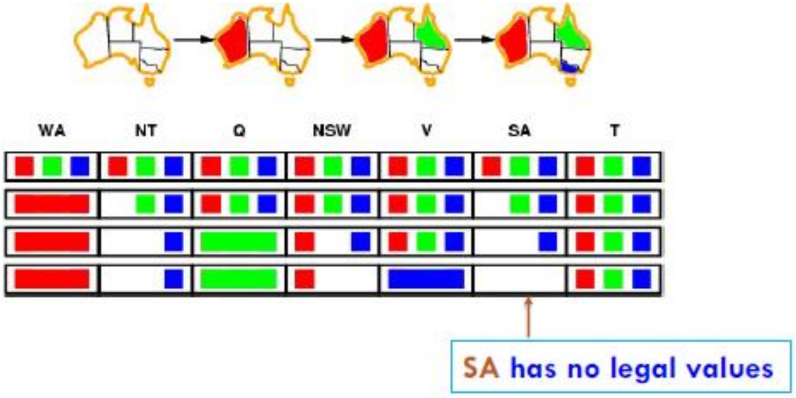
□ Consider the state {WA = red, NT = green, Q = blue}

□ This state is a no-good, because there is no valid assignment to SA

□ If the search tree starts by assigning values for WA, NT, Q →
recording this no-good would not help
because once we prune this branch from the search tree,
we will never encounter this combination again

□ If the search tree starts by assigning values for V, T →
Useful to record {WA = red, NT = green, Q = blue}
as a no-good because we will run into the same problem
again for each possible set of assignments to V and T

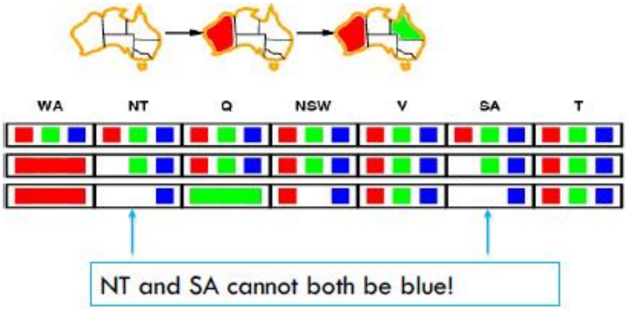
Forward checking: Keep track of remaining legal values for unassigned variables and Terminate search when a variable has no legal values.



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

SA has no legal values

Propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures, example:



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

NT and SA cannot both be blue!

3.3. Consistency techniques

They remove from domain variables values that cannot appear in the final solution.

Constraint propagation: Uses constraints to reduce the number of legal values for a variable, this can reduce the legal values for another variable.

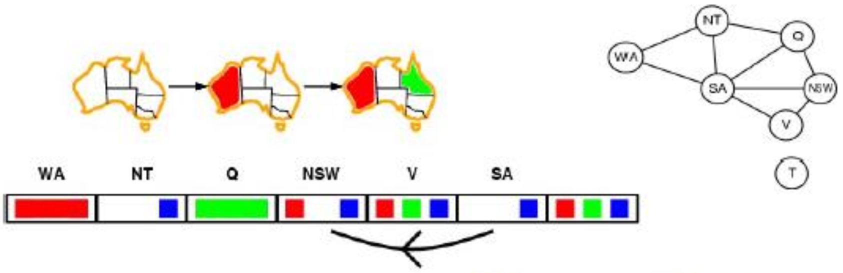
- May be interleaved with search.
- May be done as a preprocessing step, before search starts.
- Sometimes it can solve the whole problem, so no search is Required.
- By enforcing **local consistency** in each part of the constraint graph, so inconsistent values are eliminated in graph. There are different types of local consistency.

Node consistency: A variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints.
A CSP network is node-consistent if every variable in the network is node-consistent.

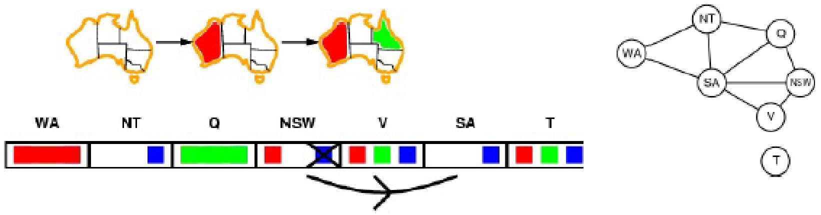
Unary constraints can be eliminated by running node consistency.
All n-ary constraints can be transformed into binary ones.
So, we will consider CSPs with only binary constraints.

Arc consistency: A variable is arc-consistent if every value in its domain satisfies the variable's binary constraints.
Formally:

- Assume there is a binary constraint between X_i and X_j,
- X_i is arc consistent with respect to X_j iff for every value x for X_i, there is some allowed y for X_j that satisfies the binary constraint between X_i and X_j



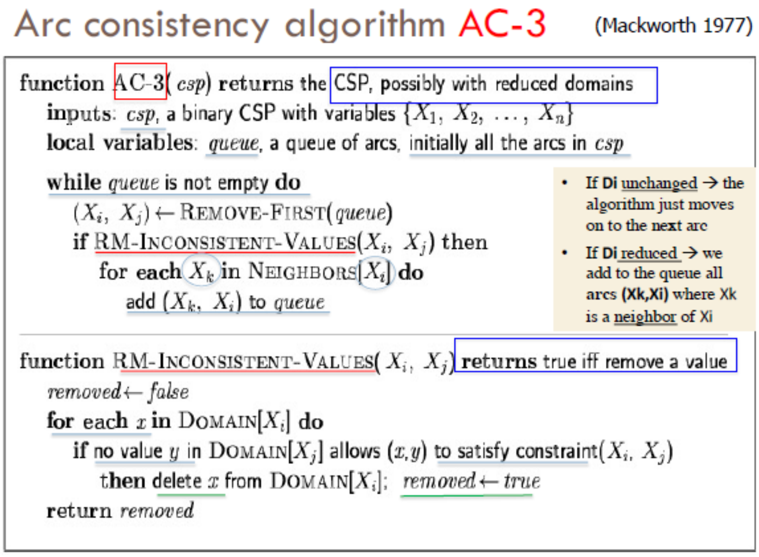
SA is **arc-consistent** with respect to **NSW**



- **NSW** is **not** arc-consistent with respect to **SA**
- **To make NSW arc-consistent** with **SA**, it is sufficient to **remove** the value ***blue*** from the domain of **NSW**

If a variable X loses a value, neighbors of X need to be rechecked.
Arc consistency detects failure earlier than forward checking.
Can be run as a preprocessor or after each assignment.

AC-3 Algorithm: maintains a queue of arcs. Initially contains all arcs of the CSP. Then one is arbitrarily extracted (X_i, X_j) so that X_i becomes arc consistent respect to X_j. If this doesn't change the domain D_i, the algorithm continues with the next arc of the queue, otherwise, it adds to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i. If D_i remains empty the problem has not solution, otherwise the algorithm continues to check the arcs until the queue is empty.



Arc consistency algorithm AC-3 (Mackworth 1977)

```
function AC-3( csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
    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 RM-INCONSISTENT-VALUES($X_i$, $X_j$) then
            for each $X_k$ in NEIGHBORS[$X_i$] do
                add ($X_k$, $X_i$) to queue

function RM-INCONSISTENT-VALUES( $X_i$, $X_j$) returns true iff remove a value
    removed $\leftarrow$ false
    for each $x$ in DOMAIN[$X_i$] do
        if no value $y$ in DOMAIN[$X_j$] allows $(x,y)$ to satisfy constraint($X_i$, $X_j$)
            then delete $x$ from DOMAIN[$X_i$]; removed $\leftarrow$ true
    return removed
```

- If Di unchanged → the algorithm just moves on to the next arc
- If Di reduced → we add to the queue all arcs (Xk,Xi) where Xk is a neighbor of Xi

For binary CSPs

- n: number of variables
- c: number of binary constraints (arcs)
- d: domain size

Time: $O(cd^3)$
Each arc (X_k, X_i) inserted in the queue only d times because X_i has at most d values to delete. Checking consistency of an arc can be done in $O(d^2)$ time. Thus, $O(cd^3)$ total worst-case time.

In general, AC prunes the search space => equivalent easier problem.

After applying AC-3:

- either every arc is arc-consistent or
- some variable has an empty domain, indicating that the CSP cannot be solved.

Path consistency (PC): A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if

- ∀ assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$
- ∃ assignment to X_m that satisfies:
    - the constraints on $\{X_i, X_m\}$ and
    - the constraints on $\{X_m, X_j\}$

This is called path consistency since it is like to consider a path from X_i to X_j with X_m in the middle. Path consistency does not guarantee that all the constraints among the variables on the path are satisfied. Only the constraints between the neighboring variables must be satisfied.

3.4. Local search

Assume an assignment is inconsistent.
Next assignment can be constructed in such a way that constraint violation is smaller.

- Only “small” changes (local steps) of the assignment are allowed.
- Next assignment should be “better” than previous.
- better = more constraints are satisfied.

Assignments are not necessarily generated systematically, **we lose completeness**, but we (hopefully) get better efficiency.

Search space: set of all complete variable assignments.
Set of solutions: subset of the search space. All complete assignments satisfying all the constraints.
Neighborhood relation: indicating what assignments can be reached by a search step given the current assignment during the search procedure.
Evaluation function: mapping each assignment to a real number representing “how far the assignment is from being a solution”.
Initialization function: returns an initial position given a possibility distribution over the assignments.
Step function: Given an assignment, its neighborhood, and the evaluation function. Returns the new assignment to be explored by the search.
Set of memory states (optional): holding information about the state of the search mechanism.
Termination criterion: stopping the search when satisfied.

Local search for CSPs:

- **Neighborhood of an assignment:** all assignments differing on one value of one variable.
- **Evaluation function:** mapping each assignment to the number of constraints it violates.
- **Initialization function:** returns an initial assignment chosen Randomly.
- **Termination criterion:** if a solution is found or if a given number of search steps is exceeded.

The different algorithms are characterized by the step function and use of memory.
Local search is used to eliminate violated constraints.
Heuristic for choosing a new value for a variable: value that results in the minimum number of conflicts with other variables.

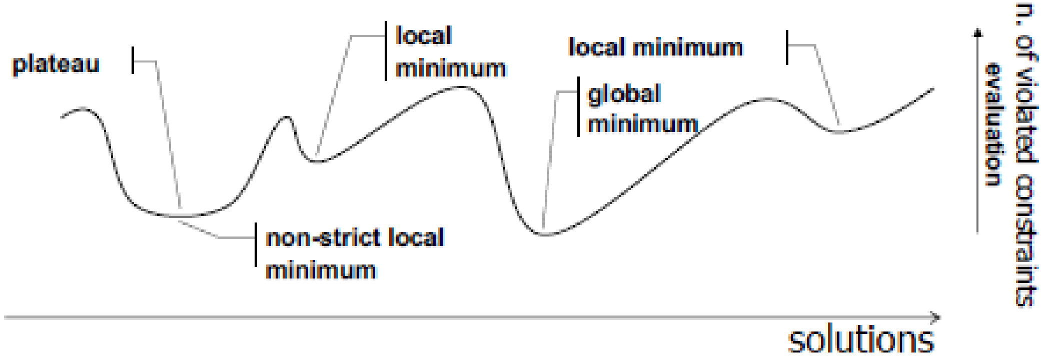
Min-Conflicts
Conflict set (of an assignment)**:** set of variables involved in some constraint that assignment is violating.
Min-conflict LS procedure:

- Starts at a randomly generated assignment.
- At each state of the search
  - Selects a variable from the current conflict set.
  - Selects a value for that variable that minimizes the number of violated constraints.
- If multiple choices: choose one randomly
  - neighbourhood = different values for the selected variable
  - neighbourhood size = (d-1)

The evaluation function can have:

- **local minimum**: a state that is not minimal and there is no state with better evaluation in its neighborhood

- **strict local minimum**: a state that is not minimal and there are only states with worse evaluation in its neighborhood.
- **global minimum:** the state with the best evaluation
- **plateau**: a set of neighboring states with the same evaluation



Escaping local minima: A local search procedure may get stuck in a local Minima.
Techniques for preventing stagnation:

- **Restart**
  - Re-initialize the search after MaxSteps (non-strictly improving) steps
  - New assignment chosen randomly
  - Can be combined both with hill-climbing and Minconflicts
  - It is effective if MaxSteps is chosen correctly and often it depends on the instance.
- Allowing non improving steps: **random walk**
  A new assignment from the neighborhood is selected randomly (e.g., the value is chosen randomly). Such technique can hardly find a solution, so it needs some guidance.
  Can be combined with the heuristic guiding the search via probability distribution:
  - p: probability of using the random walk (noise setting)
  - (1-p) : probability of using the heuristic guide.
  - Min-conflicts random walk
- Changing the neighborhood: **tabu search**
  - Being trapped in local minimum can be seen as cycling. To avoid cycles:
  - Remember already visited states and do not visit them again: memory consuming (too many states)
  - It is possible to remember just a few last states: Prevents “short” cycles.
  - **Tabu list** = a list of forbidden states. Has a fix length k (tabu tenure). “old” states are removed from the list when a new state is added. State included in the tabu list is forbidden (it is tabu)
- **Constraint weighting:** Can help concentrate the search on important constraints.
  Each constraint is given a numeric weight (initially all 1)
  At each step of the search:
  - We choose a variable/value pair to change with lowest total weight of all violated constraints.
  - Weights are then adjusted by incrementing the weight of each constraint violated by the current assignment.

3.5. Structure of the problems

Problem structure (constraint graph) can be used to find solutions quickly.
To deal with real world problems we decompose them into independent subproblems.
Independence can be obtained by finding connected components of the constraint graph.

- Each component corresponds to a subproblem CSP_i
- If assignment S_i is a solution of CSP_i then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$

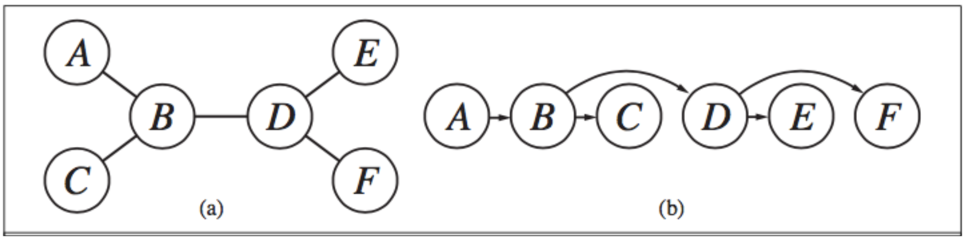
Decomposition is important because without it the work is $O(d^n)$ instead of $O(d^c n/c)$ linear.

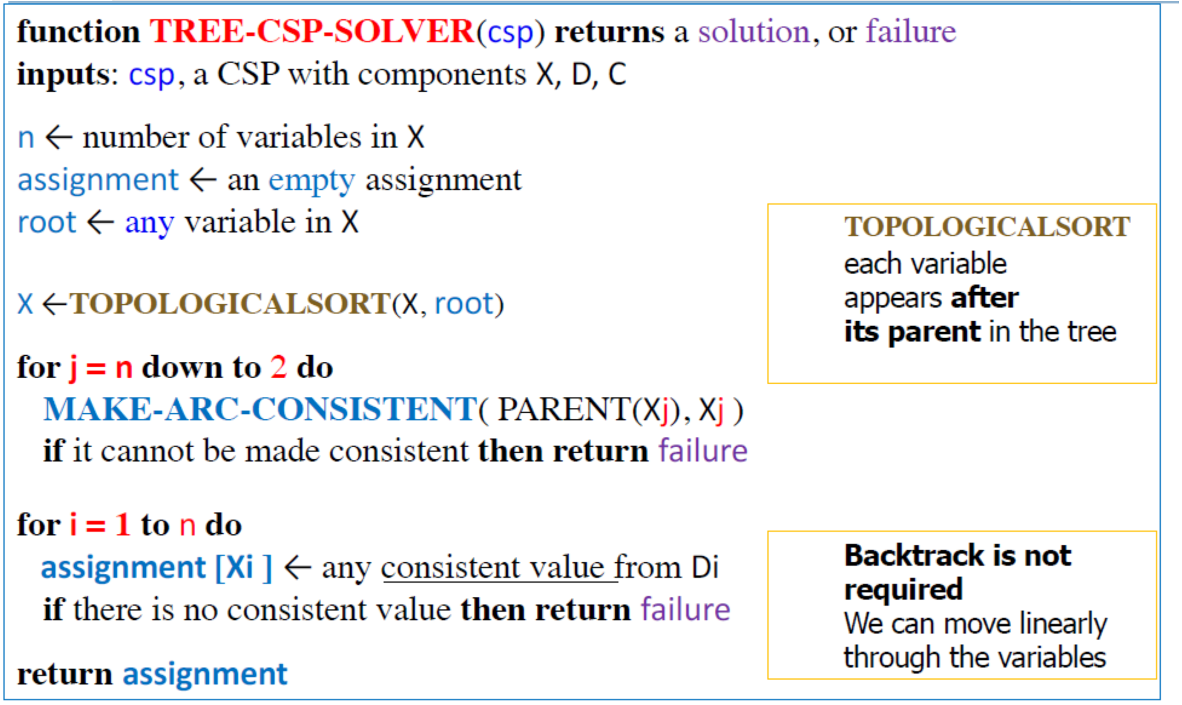
A constrained graph **is a tree** when any two variables are connected by only one path.
A CSP is defined to be **directed arc-consistent (DAC)** under an ordering of variables $X_1, X_2, \dots, X_n$ iff every X_i is arc-consistent with each X_j for $j > i$

Any **tree-structured CSP** can be solved in time linear in the number of variables.
If the CSP graph is a tree, it can be solved in $O(nd^2)$, n variables, with d values in each domain
To solve a tree-structured CSP:

- Pick any variable to be the root of the tree
- Choose an ordering of the variables such that each variable appears after its parent in the tree (topological sort)
- Make this graph directed arc-consistent ($O(nd^2)$,)
- Follow the list of variables starting from the root and choose any remaining value

DAC guarantees that for any value we choose for the parent, there will be a valid value left to choose for the child.





```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
inputs: csp, a CSP with components X, D, C

n ← number of variables in X
assignment ← an empty assignment
root ← any variable in X

X ←TOPOLOGICALSORT(X, root)

for j = n down to 2 do
  MAKE-ARC-CONSISTENT( PARENT(Xj), Xj )
  if it cannot be made consistent then return failure

for i = 1 to n do
  assignment [Xi ] ← any consistent value from Di
  if there is no consistent value then return failure

return assignment
```

TOPOLOGICALSORT
each variable appears **after its parent** in the tree

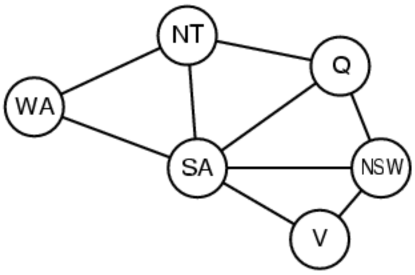
Backtrack is not required
We can move linearly through the variables

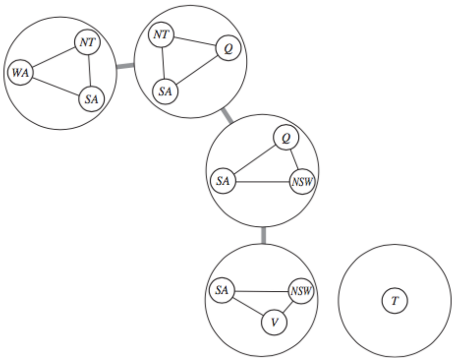
Amost tree-structured: Reduce the graph structure to a tree assigning values to some variables.
Cutset Conditioning:

- Choose a subset S of the CSPs variables such that the constraint graph becomes a tree after removal of S (S is called a cycle cutset)
- For each possible assignment of variables in S that satisfies all constraints on S:
    - Remove from the domains of the remaining variables any values that are inconsistent with the assignment for S
    - If the remaining CSP has a solution, return it together with the assignment for S

Tree decomposition: Decompose problem into a set of connected sub-problems, where two sub-problems are connected when they share a constraint.
Solve sub-problems independently and combine solutions.

- If a sub-problem has no solution the entire problem has no solution
- If we can solve all the subproblems we construct a global solution
    - Consider each sub-problem as new "mega-variable"
      Domain of each mega-variable: all the solutions to the sub-problem
    - Then, solve the constraints that connect the subproblems by using tree CSP solver to find an overall solution with identical values for the same Variable.



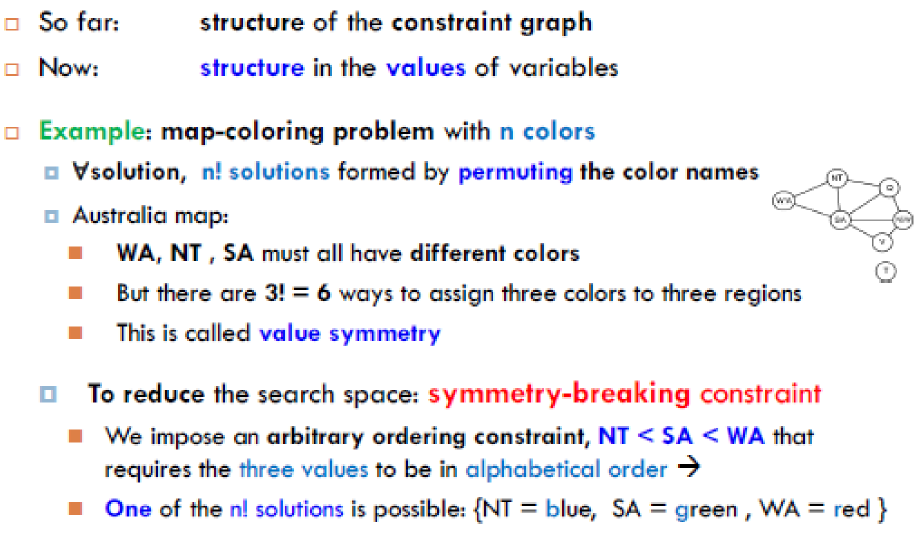


Tree decomposition must satisfy the following conditions:

- Every variable of the original CSP appears in at least one sub-problem.
- If two variables are connected by a constraint in the original CSP à they must appear with their constraint in at least one subproblem.
- If a variable appears in some subproblems it must have the same value in every subproblem.

A constraint graph allows for several tree decompositions.
Aim: to select decomposition with the subproblems as small as possible

Tree width of a tree decomposition = $s - 1$, where s is the size of largest sub-problem
Tree width of a graph w = the minimum tree width among all its tree decompositions.
If a graph has tree-width w and we know the corresponding tree decomposition, Then we can solve the problem in $O(nd^{w+1})$
CSPs with constraint graph with a bounded tree width can be solved in polynomial time.
Finding a tree decomposition with minimal tree width is NP-hard (but some heuristic methods work well in practice)



- So far: **structure of the constraint graph**
- Now: **structure in the values of variables**
- **Example: map-coloring problem** with **n colors**
  - **∀solution,** n! solutions formed by **permuting the color names**
  - Australia map:
    - **WA, NT , SA** must all have **different colors**
    - But there are **3! = 6** ways to assign three colors to three regions
    - This is called **value symmetry**
  - **To reduce** the search space: **symmetry-breaking** constraint
    - We impose an **arbitrary ordering constraint, NT < SA < WA** that requires the three values to be in alphabetical order →
    - **One** of the n! solutions is possible: {NT = blue, SA = green , WA = red }

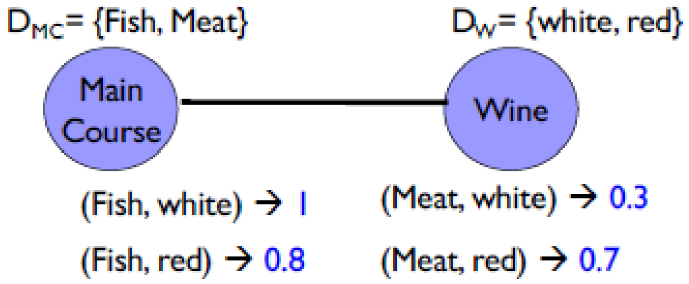
Summary of CSP:

- CSPs are a special kind of search problem:
  - states are value assignments.
  - goal test is defined by constraints.
- Backtracking = DFS with one variable assigned per node.
  Other intelligent backtracking techniques possible
- Variable/value ordering heuristics can help dramatically.
- Constraint propagation prunes the search space.
- Tree structure of CSP graph simplifies problem significantly.
- CSPs can also be solved using local search.

3.6. Soft Constraints satisfaction problems

Used to model **preferences**.

Soft constraint: a classical constraint, where each assignment of values to its variables has an associated preference value from a (totally or partially ordered) set called **c-semiring** (constraint-semiring). This set has two operations, which makes it similar to a semiring.



Soft CSP: is a set of soft constraints over a set of variables based on a specific c-semiring:

$$\langle \boldsymbol{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$$

Solution of a Soft CSP: complete assignment. One value for each variable.

Preference value of a solution (global evaluation): By combining (via ×) the preferences of the partial assignments given by the constraints.

General framework for soft CSPs: based on a c-semiring structure that is a tuple $\langle \boldsymbol{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$

- $\boldsymbol{A}$: set that specifies the preference values to be associated with each tuple, i.e., with each assignment of values of the variables.
- $+$, $\times$**:** two semiring operations
  - $+$ model constraint projection
  - $\times$ model constraint combination
- $\mathbf{0}$: worst preference value ($0 \in A$)
- $\mathbf{1}$: best preference values ($1 \in A$)

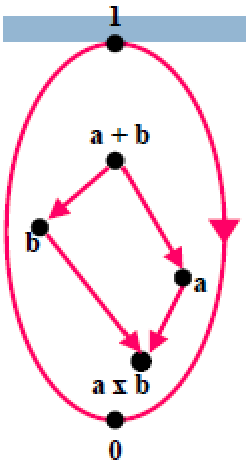
3.6.1. C-semiring framework for Soft CSPs

The c-semiring framework for soft CSPs:

- **Variables** $X = \{X_1, \dots, X_n\}$
- **Domains** $D = \{D(X_1), \dots, D(X_n)\}$
- **C-semiring** $S = \langle A, +, \times, 0, 1 \rangle$
  - A set of preferences
  - $+$ additive operator: induces the ordering $\leq_S$ over A defined as follows:

    $$\boldsymbol{a \leq_S b} \textbf{ iff } \boldsymbol{a + b = b}$$

    ($+$ is idempotent, commutative, associative, unit element 0)
  - $\times$ multiplicative operator: combines preferences.
    ($\times$ is commutative, associative, unit element 1, absorbing element 0)
  - $0,1 \in A$, they are bottom and top element.
  - $\times$ distributes over $+$
  - $+$ and $\times$ are monotone on $\leq_S$:
    - $a \leq_S b \Rightarrow a + c \leq_S b + c$
    - $a \leq_S b \Rightarrow a \times c \leq_S b \times c$
  - Intensive property: $\forall a, b \in A \quad a \times b \leq_S a$. It means that combining more constraints leads to a worse result.



  - $\langle A, \leq_S \rangle$ is a lattice:
    - $+$ is least upper bound operator.
    - $\times$ is greatest lower bound operator if x is idempotent.
  - When $\leq_S$ is a total order and $\times$ is idempotent: if $a \leq_S b$ then $a + b = b$ and $a \times b = a$ (+=max, ×=min)

Soft constraint: a pair $c = \langle f, \text{con} \rangle$ where:

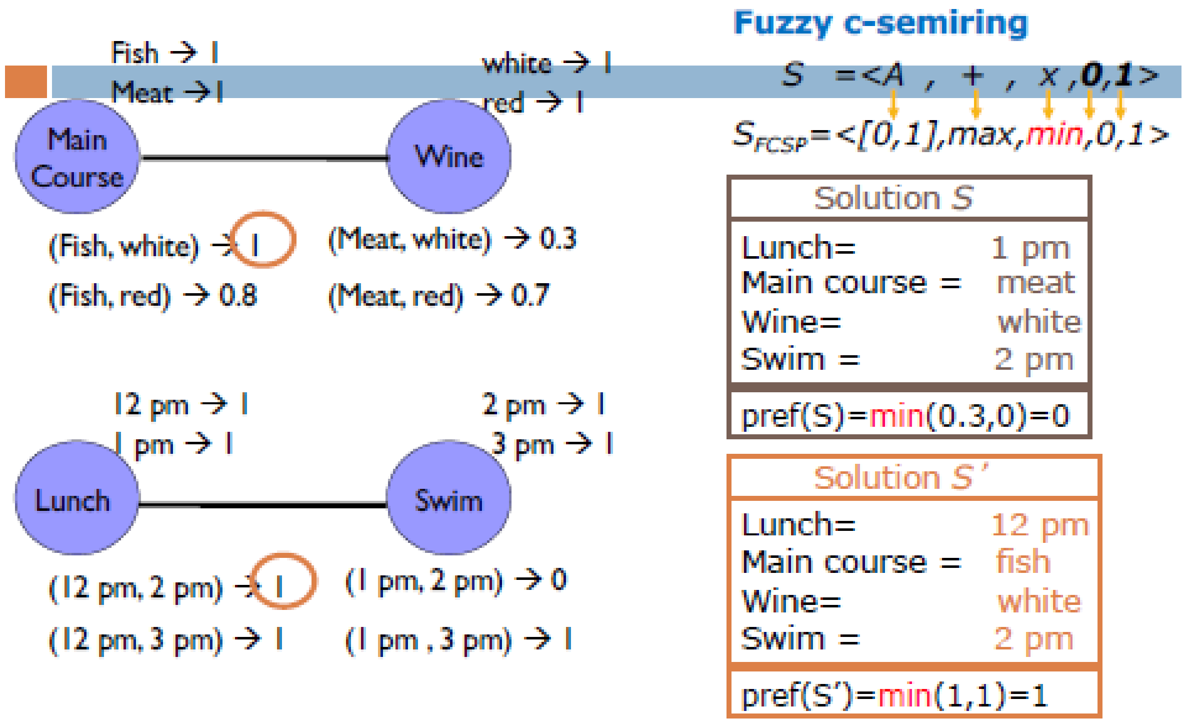
- Scope: $\text{con} = \{X_1^c, \dots, X_k^c\}$ subset of the set of variables X
- Preference function:
  - $f: D(X_1^c) \times \dots \times D(X_k^c) \to A$
  - $(v_1, \dots, v_k) \to p$ preference value
  - f is a function that associate to every assignment of values $(v_1, \dots, v_k)$ in con a specific preference value p taken from the set of preferences A

Hard constraint: a soft constraint where for each tuple $(v_1, \dots, v_k)$

- $f(v_1, \dots, v_k) = 1$ the tuple is allowed.
- $f(v_1, \dots, v_k) = 0$ the tuple is forbidden.

Fuzzy constraint satisfaction problems (FCSPs)

- **Fuzzy c-semiring**: $\langle A = [0,1], += \max, \times = \min, 0,1 \rangle$
- Preference values between 0 and 1
- Higher values denote better preferences.
  - 0 is the worst preference.
  - 1 is the best preference.
- Combination is taking the smallest value.
- optimization criterion = **maximize the minimum preference**



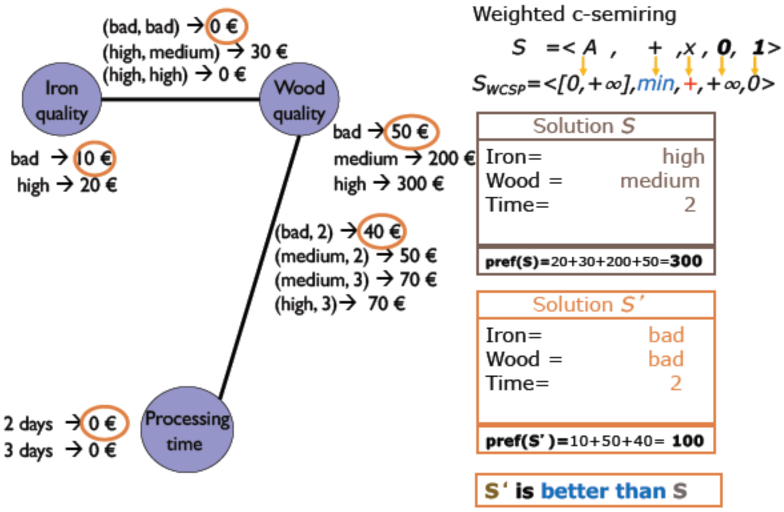
Solution S	
Lunch=	1 pm
Main course =	meat
Wine=	white
Swim =	2 pm
pref(S)=min(0.3,0)=0	

Solution S'	
Lunch=	12 pm
Main course =	fish
Wine=	white
Swim =	2 pm
pref(S')=min(1,1)=1	

S' is better than S

Weighted constraint satisfaction problems (WCSPs)

- **Weighted c-semiring**: $\langle [0, +\infty], += \max, \times = +, 0 = +\infty, 1 = 0 \rangle$
- Preference values are costs between 0 and $+\infty$
- Lower costs are better:
  - ○ $+\infty$ is the worst cost.
  - ○ 0 is the best cost.
- Combination is taking the sum of the costs.
- optimization criterion = **minimize the sum of costs**
- Useful in all settings where costs apply.



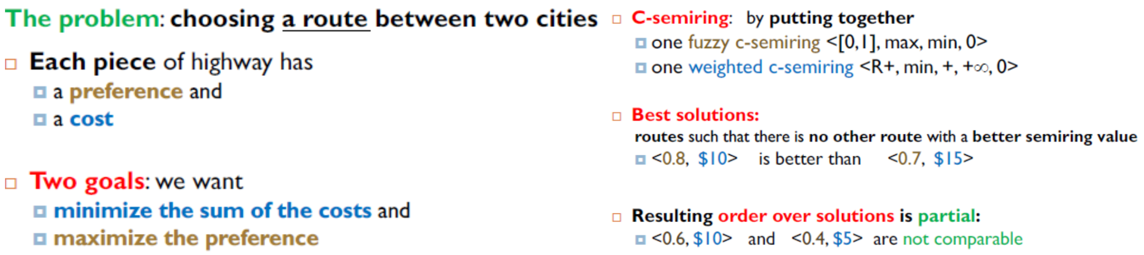
A soft CSP induces an **ordering over the solutions**, from the ordering $\leq_S$ of the c-semiring.

- If $\leq_S$ is a total order => total order over solutions (possibly with ties)
- If $\leq_S$ is a partial order => total or partial order over solutions (possibly with ties)

Any ordering can be obtained.

Instances of semiring-based soft constraints: Each instance is characterized by a c-semiring $\langle A, +, \times, 0, 1 \rangle$

- **Classical constraints:** $\langle \{0,1\}, OR, AND, 0, 1 \rangle$ Satisfy all constraints.
- **Fuzzy constraints:** $\langle [0,1], \max, \min, 0, 1 \rangle$ Maximize the minimum preference.
- **Weighted constraints:** $\langle \mathbb{R} \cup +\infty, \min, +, +\infty, 0 \rangle$ Minimize the sum of the costs.
- **Probabilistic constraints**: $\langle [0,1], \max, \times, 0, 1 \rangle$ Maximize the joint probability.
- **Multi-criteria problems:** One c-semiring for each criteria.
  - ○ Given n c-semirings $S_i = \langle A_i, +_i, \times_i, 0_i, 1_i \rangle$ we can build the c-semiring:
    $S = \langle \langle A_1, \ldots, A_n \rangle, +, \times, \langle 0_1, \ldots, 0_n \rangle, \langle 1_1, \ldots, 1_n \rangle \rangle$
    $+$ and $\times$ obtained by pointwise application of $+_i$ and $\times_i$ on each c-semiring.
  - ○ Each assignment is associated with a tuple of preference values $(p_1, \ldots, p_n)$
  - ○ The ordering over the solutions may be a partial order
  - ○ Cartesian product of c-semirings.



The problem: choosing a route between two cities

□ **Each piece** of highway has
- ◘ a **preference** and
- ◘ a **cost**

□ **Two goals**: we want
- ◘ **minimize the sum of the costs** and
- ◘ **maximize the preference**

□ **C-semiring**: by **putting together**
- ◘ one fuzzy c-semiring <[0,1], max, min, 0>
- ◘ one weighted c-semiring <R+, min, +, +∞, 0>

□ **Best solutions:**
routes such that there is **no other route** with a **better semiring value**
- ◘ <0.8, $10> is better than <0.7, $15>

□ **Resulting order over solutions is partial:**
- ◘ <0.6, $10> and <0.4, $5> are not comparable

3.6.2. Fundamental operations with soft constraints

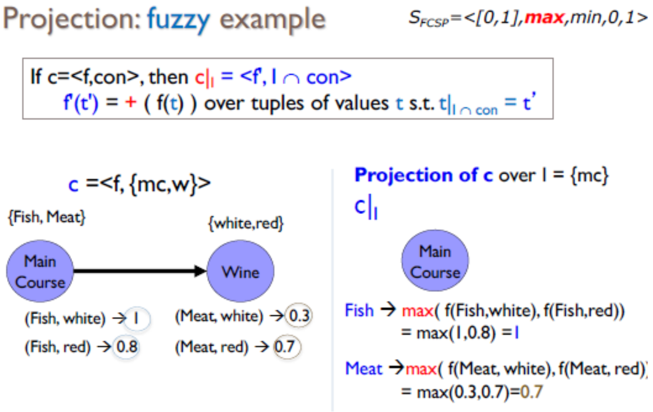
Projection: eliminate one or more variables from a constraint obtaining a new soft constraint preserving all the information on the remaining variables.
Given:

- a soft constraint $c = \langle f, \text{con} \rangle$
- a set $I \subset X$ of variables

the **projection of c over I** is the constraint $c|_I = \langle f', \text{con'} \rangle$ where:

- $\text{con'} = I \cap \text{con}$
- $f'(t') = +\big(f(t)\big)$ over tuples of values t s.t. $t|_{I \cap \text{con}} = t'$

  f' associates to each tuple of the remaining variables a preferences value which is the sum of the preference values associated by the original constraints to all the extension of this tuple over the eliminated variables.



Projection: fuzzy example

S_{FCSP}=<[0,1],max,min,0,1>

If c=<f,con>, then c|$_I$ = <f', I ∩ con>
f'(t') = + (f(t)) over tuples of values t s.t. t|$_{I \cap con}$ = t'

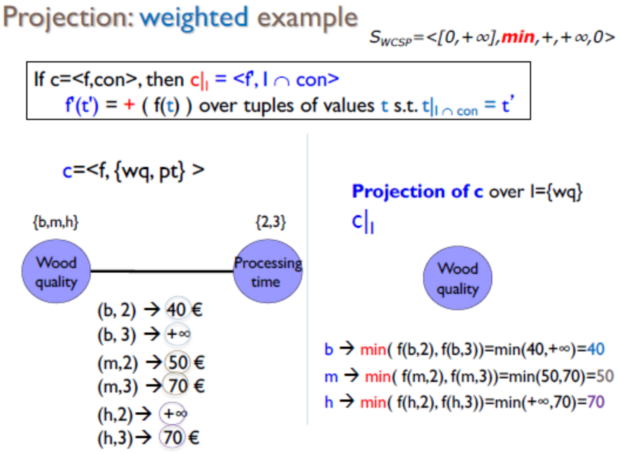
c =<f, {mc,w}>

(Fish, white) → 1
(Fish, red) → 0.8
(Meat, white) → 0.3
(Meat, red) → 0.7

Projection of c over I = {mc}
c|$_I$

Fish → max(f(Fish,white), f(Fish,red))
= max(1,0.8) =1

Meat → max(f(Meat, white), f(Meat, red)
= max(0.3,0.7)=0.7



Projection: weighted example

S_{WCSP}=<[0,+∞],min,+,+∞,0>

If c=<f,con>, then c|$_I$ = <f', I ∩ con>
f'(t') = + (f(t)) over tuples of values t s.t. t|$_{I \cap con}$ = t'

c=<f, {wq, pt} >

(b, 2) → 40 €
(b, 3) → +∞
(m,2) → 50 €
(m,3) → 70 €
(h,2)→ +∞
(h,3)→ 70 €

Projection of c over I={wq}
c|$_I$

b → min(f(b,2), f(b,3))=min(40,+∞)=40
m → min(f(m,2), f(m,3))=min(50,70)=50
h → min(f(h,2), f(h,3))=min(+∞,70)=70

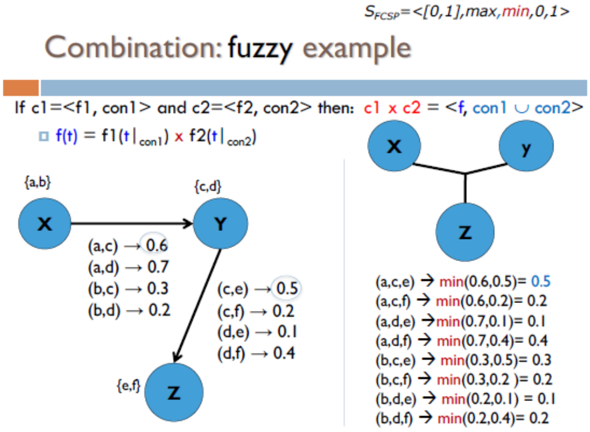
Combination: combine two or more soft constraints obtaining a new soft constraint "synthesizing" all the information of the original ones.
Given two soft constraints:

- $c_1 = \langle f_1, \text{con}_1 \rangle$
- $c_2 = \langle f_2, \text{con}_2 \rangle$

their **combination** is the constraint $c_1 \times c_2 = \langle f', \text{con'} \rangle$, where:

- $\text{con'} = \text{con}_1 \cup \text{con}_2$
- $f'(t') = f_1\big(t|_{\text{con}_1}\big) \times f_2\big(t|_{\text{con}_2}\big)$

  f' associates to each tuple of the domain values for these variables, a preference value that is obtained by multiplying the preference values associated by the original sub-constraints to the appropriate sub-tuples.



S_{FCSP}=<[0,1],max,min,0,1>

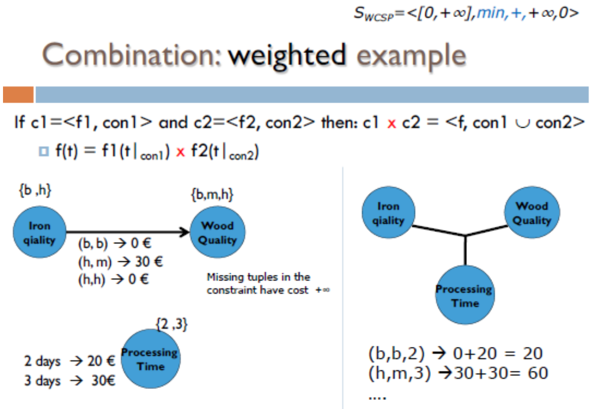
Combination: fuzzy example

If c1=<f1, con1> and c2=<f2, con2> then: c1 x c2 = <f, con1 ∪ con2>
□ f(t) = f1(t|$_{con1}$) x f2(t|$_{con2}$)

(a,c) → 0.6
(a,d) → 0.7
(b,c) → 0.3
(b,d) → 0.2

(c,e) → 0.5
(c,f) → 0.2
(d,e) → 0.1
(d,f) → 0.4

(a,c,e) → min(0.6,0.5)= 0.5
(a,c,f) → min(0.6,0.2)= 0.2
(a,d,e) →min(0.7,0.1)= 0.1
(a,d,f) → min(0.7,0.4)= 0.4
(b,c,e) → min(0.3,0.5)= 0.3
(b,c,f) → min(0.3,0.2)= 0.2
(b,d,e) → min(0.2,0.1) = 0.1
(b,d,f) → min(0.2,0.4)= 0.2



S_{WCSP}=<[0,+∞],min,+,+∞,0>

Combination: weighted example

If c1=<f1, con1> and c2=<f2, con2> then: c1 x c2 = <f, con1 ∪ con2>
□ f(t) = f1(t|$_{con1}$) x f2(t|$_{con2}$)

(b, b) → 0 €
(h, m) → 30 €
(h,h) → 0 €

Missing tuples in the constraint have cost +∞

2 days → 20 €
3 days → 30€

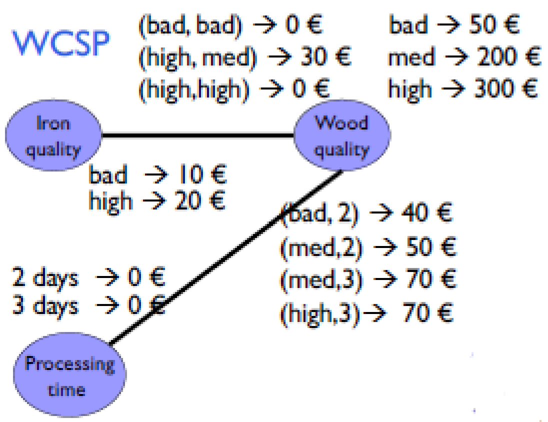
(b,b,2) → 0+20 = 20
(h,m,3) →30+30= 60
....

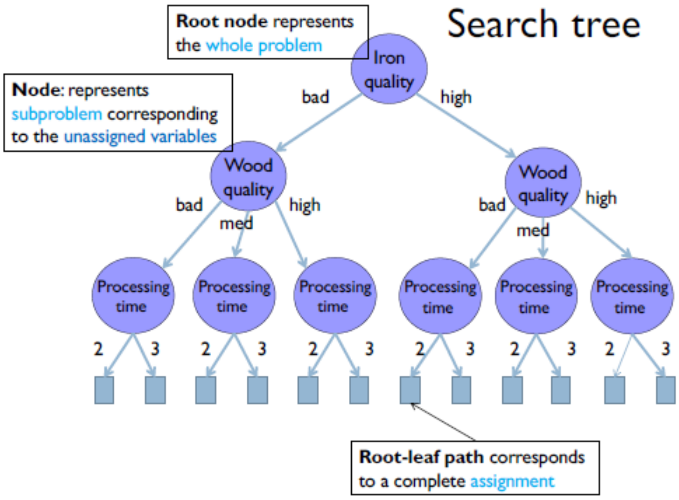
Find an optimal solution (Ex.: branch and bound + constraint propagation)
Is t an optimal solution? (we have to find the optimal preference level)
Is t better than t'? Linear in the number of constraints (compute the two pref. levels and compare them)

3.6.3. Systematic search: Branch and Bound

Visit each assignment that may be a solution, skip only assignments that are shown to be dominated by others.
Search tree to represent the space of all assignments.

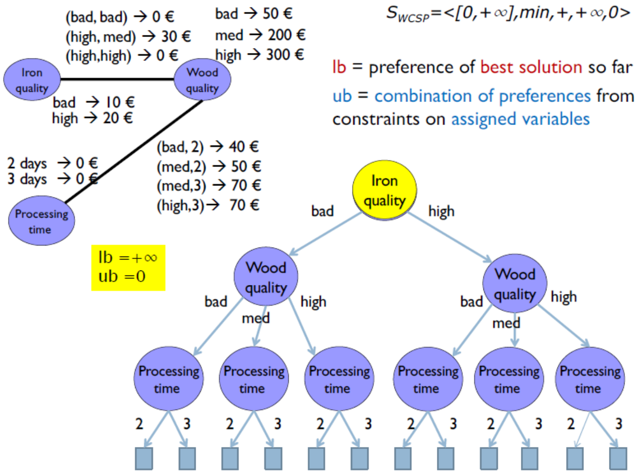
$$S_{WCSP}=<[0,+\infty],min,+,+\infty,0>$$

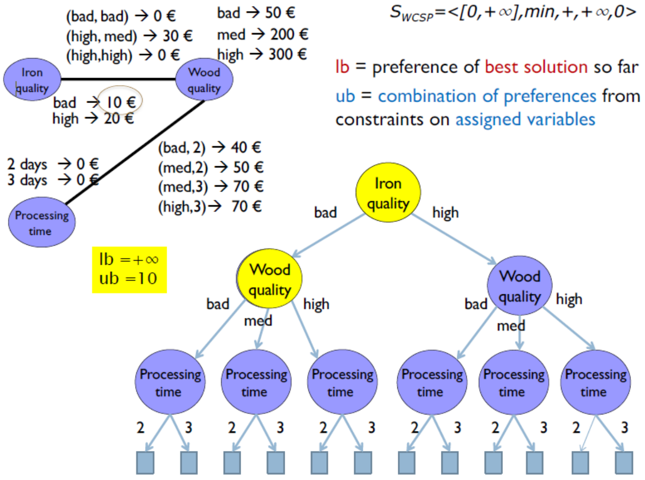


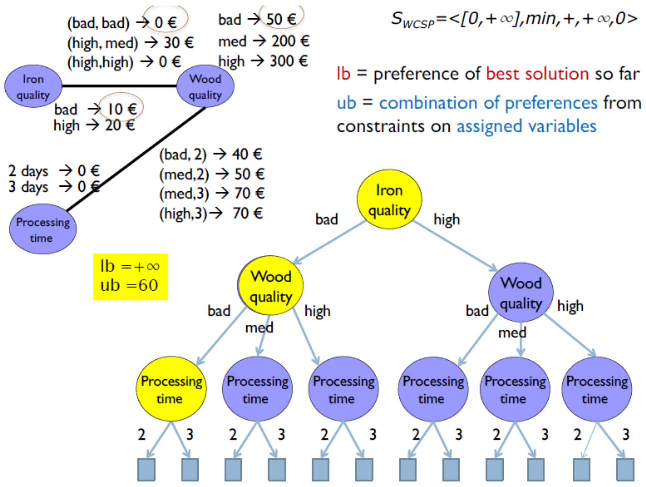


Search tree
Root node represents the whole problem
Node: represents subproblem corresponding to the unassigned variables
Iron quality
bad
high
Wood quality
bad
med
high
Processing time
2
3
Root-leaf path corresponds to a complete assignment

Lower bound (lb)= preference of best solution so far (0 of the semiring, at the beginning)
Upper bound (ub) for each node: upper bound to the preference of any assignment in the subtree rooted at the node. = combination (×) of preferences from constraints on assigned variables.
If ub is worst than lb => prune subtree



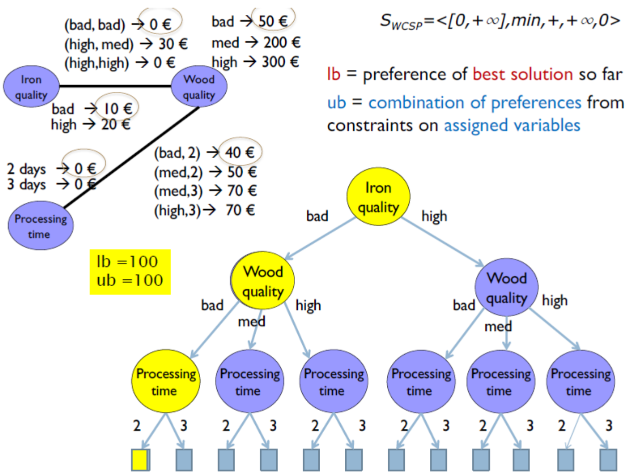




S_{WCSP}=<[0,+∞],min,+,+∞,0>

lb = preference of best solution so far

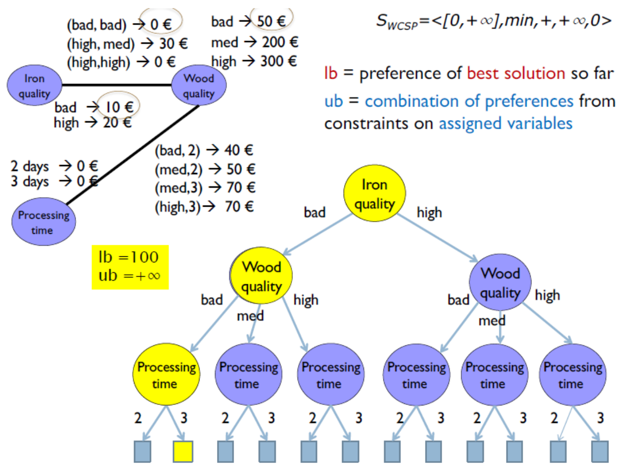
ub = combination of preferences from constraints on assigned variables



S_{WCSP}=<[0,+∞],min,+,+∞,0>

lb = preference of best solution so far

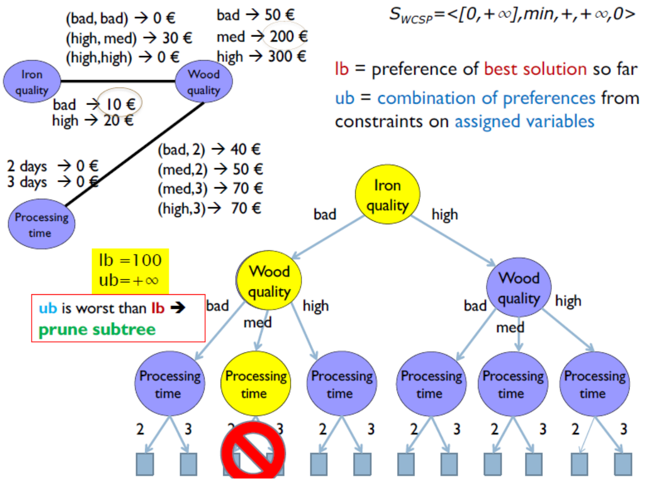
ub = combination of preferences from constraints on assigned variables



S_{WCSP}=<[0,+∞],min,+,+∞,0>

lb = preference of best solution so far

ub = combination of preferences from constraints on assigned variables



S_{WCSP}=<[0,+∞],min,+,+∞,0>

lb = preference of best solution so far

ub = combination of preferences from constraints on assigned variables

ub is worst than lb → prune subtree

Incomplete soft constraint problems:

- SCSPs with some missing preferences
- New notions of optimal solutions
- Idea: interleaving search and preference elicitation

4. Multi-agent decision making

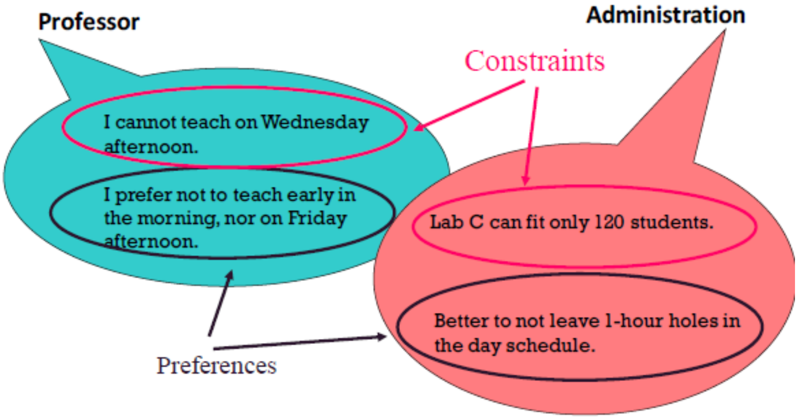
4.1. Preferences

Preferences:

- are **orderings** over possible options.
- can model levels of acceptance, or costs.
- are tolerant constraints.

Constraints are strict requirements that must be satisfied.

Constraints and preferences may be present in the same problem.


Professor
Administration
Constraints
I cannot teach on Wednesday afternoon.
I prefer not to teach early in the morning, nor on Friday afternoon.
Lab C can fit only 120 students.
Better to not leave 1-hour holes in the day schedule.
Preferences

Kind of preferences:

- **Unconditional:** *I prefer taking the bus*
- **Conditional**: *I prefer taking the bus if it's raining*
- Multi-agent: *I like blue, my husband likes green, what color for the new car?*
- **Quantitative**: Numbers or ordered set of objects. *My preference for ice cream is 0.8, and for cake is 0.6*
- **Qualitative**: Pairwise comparisons, *Ice cream is better than cake*

Ways to model preferences:

- **Soft constraints:** for modeling **quantitative** and **unconditional** preferences. (Ex., My preference for ice cream is 0.8, and for cake is 0.6)
- **CP-nets**: for modeling **qualitative** and **conditional** preferences. (Ex., Red wine is better than white wine if there is meat)

Collective decision making:

- Several agents
- Common set of possible decisions
- Each agent has its preferences over the possible decisions.
- **Goal**: to choose one of the decisions, based on the preferences of the agents
  or a set of decisions
  or a ranking over the decisions

To compute a collective decision:

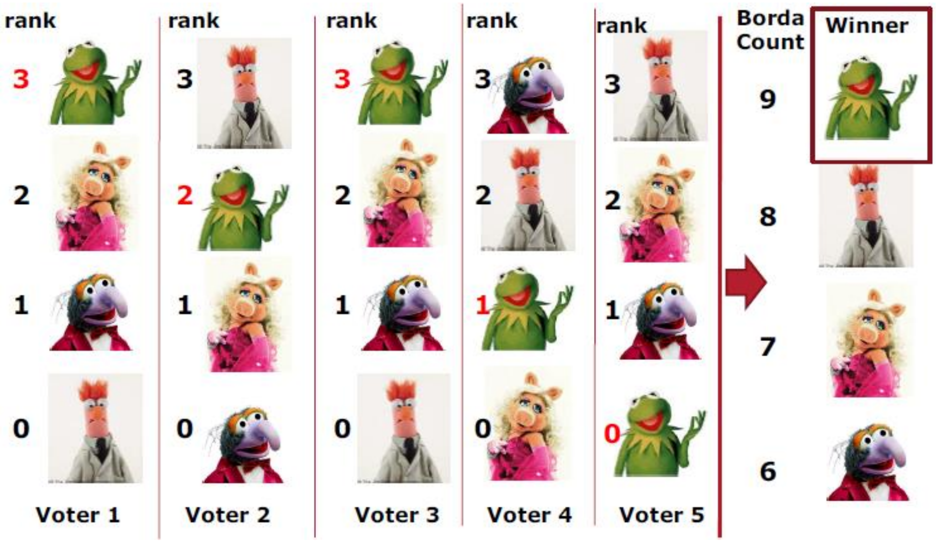
- Let the agents vote by expressing their preferences over the possible decisions.
- Aggregate the votes to get a single decision.

4.2. Voting theory (social choice)

- Agents = Voters
- Decisions = Candidates
- Preferences
- **Goal**: to choose one candidate (the winner), based on the voters' preferences. Also many candidates, or ranking over candidates
- **Voting Rules** (functions) to achieve the goal.

Voting rules:

- **Plurality:**
  - **Voting**: each voter provides the most preferred decision
  - **Selection**: the decision preferred by the largest number of voters
- **Majority**: like plurality, over 2 options
- **Approval:** (m options)
  - **Voting**: each voter approves any number of options
  - **Selection**: option with most votes
- **Borda:**
  - **Voting**: each voter provides a rank over all options
  - **Score** of an option: number of options that it dominates.
  - **Selection**: option with greatest sum of the scores



Desirable properties of voting rules:

- **Unanimity (efficiency):** If all voters have the same top choice, it is selected.
- **Non-dictatorship:** There is no dictator (Dictator: voter such that his top choice always wins, regardless of the votes of other voters)
- **Non-manipulability:** There is no incentive for agents to misrepresent the preferences.

Impossibility results:

- **Arrow's theorem:** Totally ordered preferences. It is impossible to find a voting rule with some desirable properties including **both unanimity and non-dictatoriality**
- **Gibbard-Sattherwaite's theorem:** Totally ordered preferences. it is impossible to have a reasonable voting rule that is **both non-dictatorial and non-manipulable**.

These impossibility results holds also when we allow partially ordered preferences.

4.3. Multi-agent systems

In multi-agent AI scenarios, we usually have:

- Incomparability
- Uncertainty, vagueness, preference elicitation
- Computational concerns
- Large set of decisions (candidates) w.r.t. number of agents (voters)
- Combinatorial structure for the set of decisions (candidates)

Incomparability:

- Preferences do not always induce a total order over the options.
- Preferences may induce a partial order where some options are incomparable.
- Some options are naturally incomparable.
  Eg., it may be easy to compare two apartments, but it may be difficult to compare an apartment and a house, thus we say they are incomparable.
- An agent may have several possibly conflicting preference criteria he wants to follow.
  Eg., a cheap and slow car is incomparable w.r.t. an expensive and fast car.
- Many AI formalisms to model preferences allow for partial orders (eg., soft constraints)

Uncertainty, vagueness:

- **Missing** preferences
  - Too costly to compute them
  - Privacy concerns
  - Ongoing elicitation process
- **Imprecise** preferences
  - Preferences coming from sensor data.
  - Too costly to compute the exact preference.
  - Estimates

Aim in AI: Find compact preference formalisms and solving techniques to model and solve problems with missing or imprecise preferences.

Preference elicitation

- Some preferences may be missing.
- Time consuming, costly, difficult, to elicit all preferences.
- We want to terminate preferences elicitation as soon as a winner fixed.

Computational concerns:

- We would like to avoid very costly ways to: Model agents' preferences, Compute the winner, Reason with the agents' preferences.
- On the other hand, we need a computational barrier against bad behaviors (such as manipulation)

Large set of candidates: In AI scenarios, usually

- the set of decisions (candidates) is much larger than
- the set of agents expressing preferences over the decisions.

Combinatorial structure for the set of decisions: need to Find compact preference formalism to express agents' preferences.

4.4. Formalism to model preferences

Soft constraint formalism (the c-semiring framework)

- The agent expresses his preferences over partial assignments of the decision variables.
- From these preferences the preference ordering over the solution space is generated

Soft constraints model quantitative and unconditional preferences but it is difficult to elicitate quantitative preferences from the user.

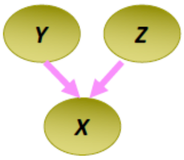
CP-net formalism: compactly represent **qualitative** and **conditional** preferences.

- Variables $\{X_1, \dots, X_n\}$ with domains

Independent variable: For each variable there is a total order over its values

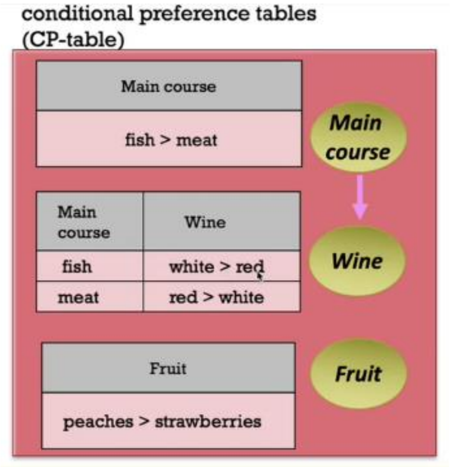
$$X = v_1 > X = v_2 > \cdots > X = v_k$$

Dependent variable: a total order for each combination of values of some other variables (conditional preference table, aka CP-table)



- $Y = a,\ Z = b: \quad X = v_1 > X = v_2 > \cdots > X = v_k$
- X depends on Y and Z (parents of X)
- Graphically: directed graph over $X_1, \dots, X_n$
- Possibly cyclic

CP-net induces an ordering over solutions.



conditional preference tables (CP-table)

Main course
fish > meat

Main course	Wine
fish	white > red
meat	red > white

Fruit
peaches > strawberries

Variables

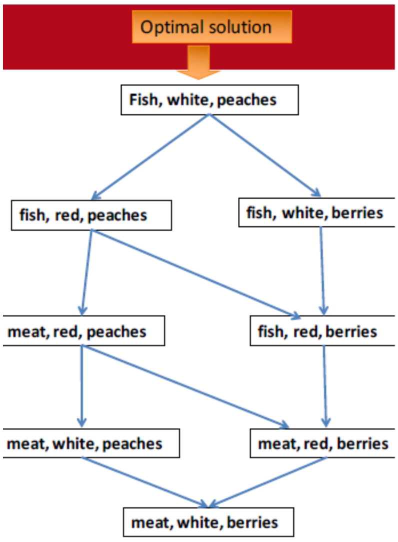
- **MainCourse**
- **Wine**
- **Fruit**

Domains

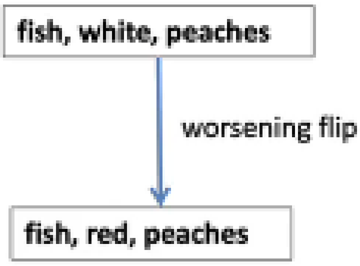
- D_MainCourse = {fish, meat}
- D_Wine = {white, red}
- D_{Fruit} ={peaches, strawberries}

MainCourse and **Fruit** are **independent variables**

Wine is a **dependent variabile**
The preference on the values of Wine is specified by a ***CP-table***



Worsening flip: changing the value of a variable in a way that is less preferred in some statement.



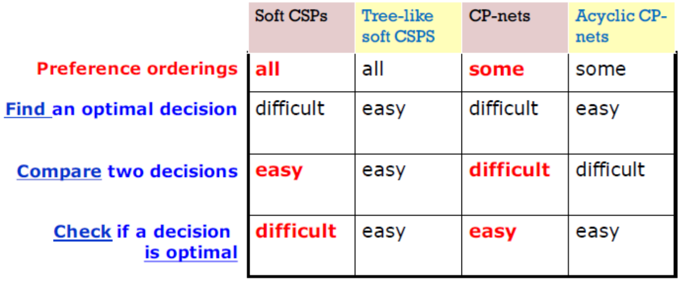
An outcome $\boldsymbol{O_1}$ **is preferred to** $\boldsymbol{O_2}$ iif there is a sequence of worsening flips from O_1 to O_2.
Optimal outcome: if no other outcomes is preferred.

Finding an optimal solution in acyclic CP-nets: forward sweep algorithm.

- First consider independent variables: Assign them their most preferred values.
- Then consider dependent variables, that directly depend on the assigned variables: Assign them their most preferred values that are consistent with the values previously assigned to their parents.
- And so on until we assign a value to all the variables.

Soft constraints vs CP-nets:

- Different expressive power
- Different computational complexity for reasoning with them

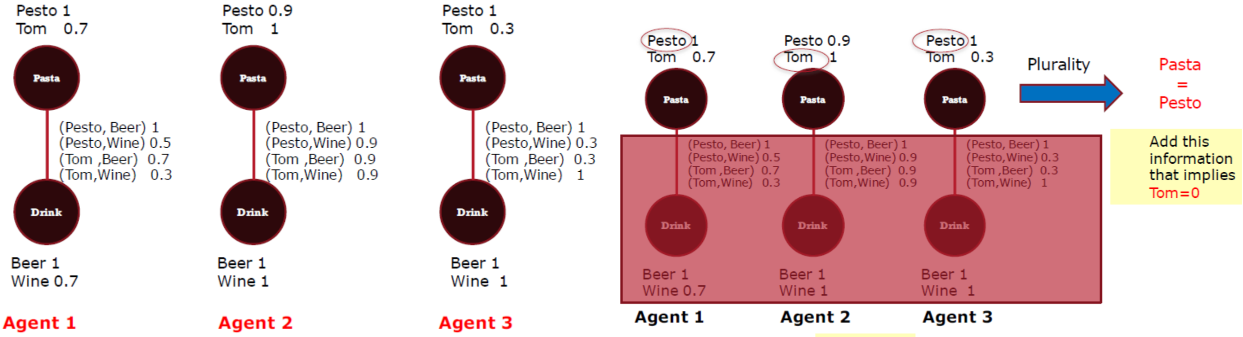


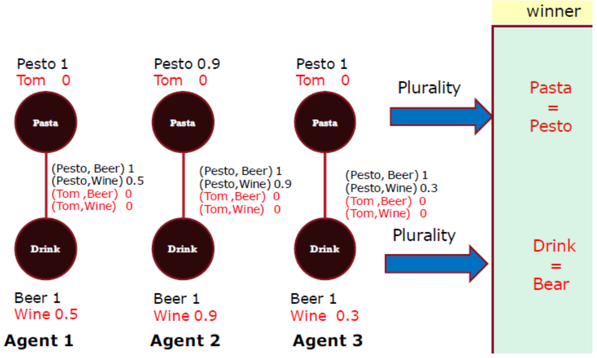
	Soft CSPs	Tree-like soft CSPS	CP-nets	Acyclic CP-nets
Preference orderings	all	all	some	some
Find an optimal decision	difficult	easy	difficult	easy
Compare two decisions	easy	easy	difficult	difficult
Check if a decision is optimal	difficult	easy	easy	easy

Sequential Voting: Several voters, Decisions are made by several issues.
Main idea: Vote separately on each issue, but do so sequentially.
This gives voters the opportunity to make their vote for one issue depending on the decisions on previous issues.

Apply **sequential voting** when agents express their preferences via **soft constraint problems**:
Assume the agents have: the same constraint graph, but different preference values.
For each variable:

- Compute an explicit profile over the variable domain.
- Apply a voting rule to this explicit profile => the rule will return a specific value that will be assigned to this variable
- Add the information about the selected variable value.





5. Bayesian network

Network models to reason under uncertainty according to the laws of probability theory.
Is a simple graphical notation to represent the dependencies among variables and for compact specification of any **full joint probability distribution**.

5.1. Syntax

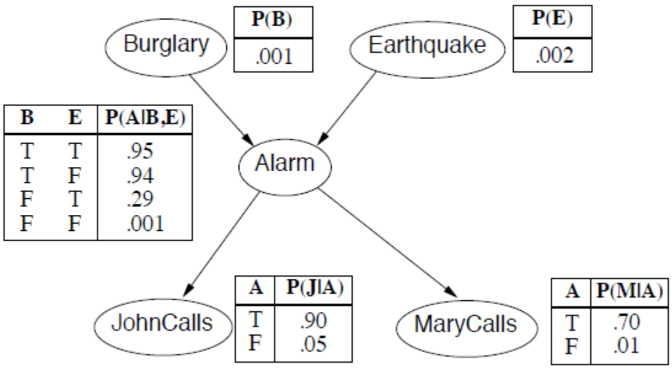
- Directed acyclic graph
- **set of nodes**, one per variable
- **set of oriented arcs.** Where $X \rightarrow Y$ means X directly influences Y. (X is parent of Y)
- For each node X_i, a conditional probability distribution given parents of X_i: $P\left(X_i \middle| \text{parents}(X_i)\right)$

The conditional distributions are represented as a **conditional probability table (CPT)** giving the probability distribution over X_i for each combination of parents values.

- Each row in a CPT contains the conditional probability of each node value for a conditioning case, that is, for each possible combination of values for the parent nodes.
- For Boolean variables, once you know that the probability of a true value is p, the probability of false must be $1-p$, so we often omit the second number.

Example:

- I'm at work, neighbor John calls to say my alarm is ringing, but neighbor Mary doesn't call. Sometimes it's set off by minor earthquakes. Is there a burglar?
- Variables: Burglary, Earthquake, Alarm, JohnCalls, MaryCalls
- Topology:



P(B)
.001

P(E)
.002

B	E	P(A|B,E)
T	T	.95
T	F	.94
F	T	.29
F	F	.001

A	P(J|A)
T	.90
F	.05

A	P(M|A)
T	.70
F	.01

Compactness: Bayesian network are more compact representation than the full joint distribution

- CPT for a Boolean variable X_i with k Boolean parents has 2^k rows for the combinations of parents values
- Each row requires one number p for X_i= true (since the number for X_i = false is $1-p$)
- Assume there are n Boolean variables
- If each variable has no more than k parents
- **Bayesian network** can be specified by at **most $n * 2^k$ numbers**
- The **full joint distribution** contains **2^n numbers**

Semantics: The full joint distribution is defined as the product of the local conditional distributions:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P\left(X_i \middle| \text{parents}(X_i)\right)$$

Constructing Bayesian networks:

- Choose an ordering of variables $X_1, \dots, X_n$
- For $i = 1$ to n:
  - Add X_i to the network
  - Select parents from $X_1, \dots, X_{i-1}$ such that $P\left(X_i \middle| \text{parents}(X_i)\right) = P(X_i|X_1, \dots, X_{i-1})$

This choice of parents guarantees:

$$\boldsymbol{P}(\boldsymbol{X_1}, \dots, \boldsymbol{X_n}) = \prod_{i=1}^{n} P(X_i|X_1, \dots, X_{i-1}) = \prod_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{P}\left(\boldsymbol{X_i} \middle| \textbf{parents}(\boldsymbol{X_i})\right)$$

Parents of node X_i should contain all those nodes in $X_1, \dots, X_{i-1}$ that directly influence X_i

5.2. Inference in Bayesian Networks

We want to compute the **posterior probability** distribution for a set of query variables given some observed event (**observed event** = an assignment of values to a set of **evidence variables**).
We assume **one query variable**.

- X denotes the **query variable.**
- E denotes the set of **evidence variables** $E_1, \dots, E_m$
- e is a particular observed event.
- Y denotes **hidden variables** $Y_1, \dots, Y_l$ (that are the nonevidence, nonquery variables)
- Complete set of variables: $X = \{X\} \cup E \cup Y$
- Typical query: compute posterior probability distribution $\boldsymbol{P}(\boldsymbol{X}|\boldsymbol{e}) = \alpha P(X, e) = \boldsymbol{\alpha} \sum_{\boldsymbol{y}} \boldsymbol{P}(\boldsymbol{X}, \boldsymbol{e}, \boldsymbol{y})$
  α: is a normalization constant.
  Sum on all possible values of hidden variables Y.

Example: compute the probability of intrusion (B) knowing that John called (j) and Mary called (m).
$P(B|j, m)$

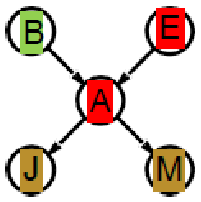
5.2.1. Exact inference by enumeration

Slightly intelligent way to sum out variables from the full joint distribution without actually constructing its explicit representation.
Query:

$$P(B|j, m) = \alpha P(B, j, m) = \alpha \sum_{e} \sum_{a} P(B, e, a, j, m)$$

Where B is a query variable, j and m are evidence variables, a and e are hidden variables.
Rewrite full joint entries using product of CPT entries.
For simplicity, we do this just for Burglary = true:

$$= \alpha \sum_{e} \sum_{a} P(b)P(e)p(a|b, e)P(j|a)P(m|a)$$

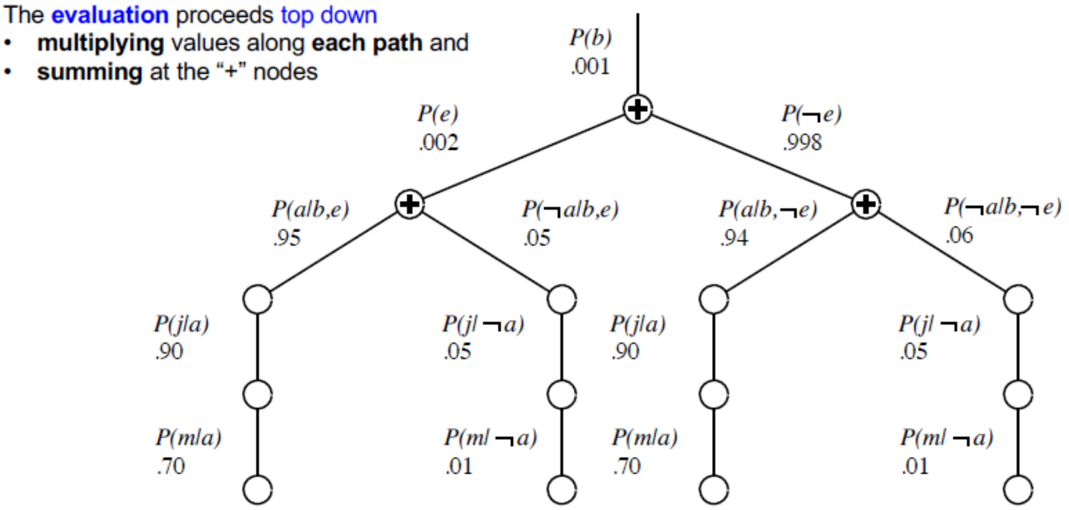


$P(e)$ is not dependent of a so can be taken out.

$$= \alpha P(b) \sum_{e} P(e) \sum_{a} P(a|b, e)P(j|a)P(m|a)$$

$O(2^n)$ time complexity for n Boolean variables

Evaluation tree



The **evaluation** proceeds top down
- **multiplying** values along **each path** and
- **summing** at the “+” nodes

Enumeration is inefficient, there are repeated computations (first and third subtree).

5.2.2. Exact inference by variable elimination

Enumeration algorithm can be improved by eliminating repeated calculations.

$$P(B|j,m) = \alpha P(B,j,m) = \alpha P(b) \sum_e P(e) \sum_a P(a|b,e) P(j|a) P(m|a)$$

We are going to **evaluate** the expression **from right to left** remembering the intermediate results.

$$\mathbf{P}(B\,|\,j,m) = \alpha\, \underbrace{\mathbf{P}(B)}_{f_1(B)}\, \Sigma_e \underbrace{P(e)}_{f_2(E)}\, \Sigma_a \underbrace{\mathbf{P}(a\,|\,B,e)}_{f_3(A,B,E)}\, \underbrace{P(j\,|\,a)}_{f_4(A)}\, \underbrace{P(m\,|\,a)}_{f_5(A)}$$

factors →

Each **factor** is a matrix indexed by the values of its argument variables.
Example:

- $f_4(A)$ correspond to $P(j|a)$ and $f_5(A)$ corresponds to $P(m|a)$, both depend just on A because J and M are fixed.

$$\mathbf{f}_4(A) = \begin{pmatrix} P(j\,|\,a) \\ P(j\,|\,\text{-}a) \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.05 \end{pmatrix} \qquad \mathbf{f}_5(A) = \begin{pmatrix} P(m\,|\,a) \\ P(m\,|\,\text{-}a) \end{pmatrix} = \begin{pmatrix} 0.70 \\ 0.01 \end{pmatrix}$$

A	P(J|A)
T	.90
F	.05

A	P(M|A)
T	.70
F	.01

- $f_3(A,B,E)$ is a 2x2x2 matrix
  - First element $P(a|b,e) = 0.95$
  - …
  - Last element $P(\neg a|\neg b, \neg e) = 1 - P(a|\neg b,\neg e) = 0.999$

B	E	P(A|B,E)
T	T	.95
T	F	.94
F	T	.29
F	F	.001

In terms of factors:

$$P(B|j,m) = \alpha f_1(B) \times \sum_e f_2(E) \times \sum_a f_3(A,B,E) \times f_4(A) \times f_5(A)$$

Evaluation:

- $$\mathbf{f}_6(B,E) = \Sigma_a\, \mathbf{f}_3(A,B,E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A)$$
  $$= (\mathbf{f}_3(a,B,E) \times \mathbf{f}_4(a) \times \mathbf{f}_5(a)) + (\mathbf{f}_3(\text{-}a,B,E) \times \mathbf{f}_4(\text{-}a) \times \mathbf{f}_5(\text{-}a))$$

$\mathbf{P}(B \mid j, m) = \alpha\, \mathbf{f}_1(B) \times \boxed{\Sigma_e\, \mathbf{f}_2(E) \times \mathbf{f}_6(B,E)}$

Sum out variable E from the product of $\mathbf{f}_2$ and $\mathbf{f}_6$

$$\mathbf{f}_7(B) = \Sigma_e\, \mathbf{f}_2(E) \times \mathbf{f}_6(B, E)$$

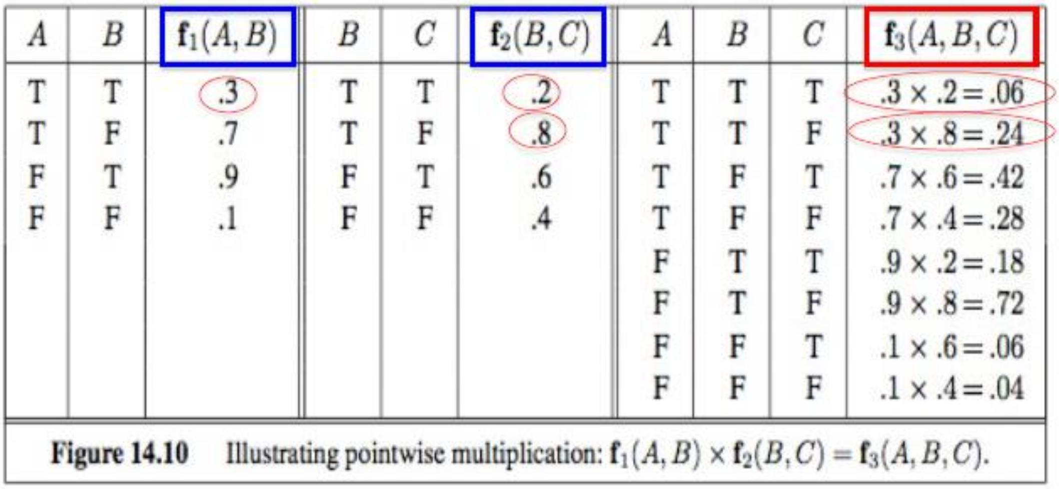
$$= \mathbf{f}_2(e) \times \mathbf{f}_6(B, e) + \mathbf{f}_2(-e) \times \mathbf{f}_6(B, -e)$$

- $\mathbf{P}(B \mid j, m) = \alpha\, \mathbf{f}_1(B) \times \mathbf{f}_7(B)$
- we need to take the **pointwise product** and normalize the result

Operations on factors:

- **Pointwise product:** two factors f_1 and f_2 yields a new factor f:
  - The variables of f are the union of the variables in f_1 and f_2
  - The elements of f are given by the product of the corresponding elements in the two factors.

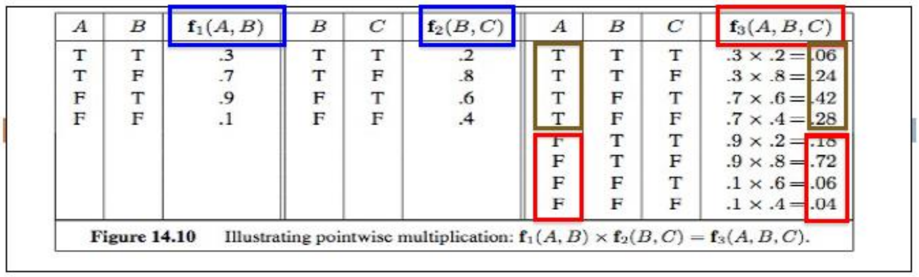
Example: $\mathbf{f}_1(A,B) \times \mathbf{f}_2(B, C) = \mathbf{f}_3(A, B, C)$



A	B	$\mathbf{f}_1(A,B)$	B	C	$\mathbf{f}_2(B,C)$	A	B	C	$\mathbf{f}_3(A,B,C)$
T	T	.3	T	T	.2	T	T	T	$.3 \times .2 = .06$
T	F	.7	T	F	.8	T	T	F	$.3 \times .8 = .24$
F	T	.9	F	T	.6	T	F	T	$.7 \times .6 = .42$
F	F	.1	F	F	.4	T	F	F	$.7 \times .4 = .28$
						F	T	T	$.9 \times .2 = .18$
						F	T	F	$.9 \times .8 = .72$
						F	F	T	$.1 \times .6 = .06$
						F	F	F	$.1 \times .4 = .04$

Figure 14.10 Illustrating pointwise multiplication: $\mathbf{f}_1(A,B) \times \mathbf{f}_2(B,C) = \mathbf{f}_3(A,B,C)$.

- **Summing out a variable** from a product of factors is done:
  - by adding up the submatrices formed
  - by fixing the variable to each of its values in turn



A	B	$\mathbf{f}_1(A,B)$	B	C	$\mathbf{f}_2(B,C)$	A	B	C	$\mathbf{f}_3(A,B,C)$
T	T	.3	T	T	.2	T	T	T	$.3 \times .2 = .06$
T	F	.7	T	F	.8	T	T	F	$.3 \times .8 = .24$
F	T	.9	F	T	.6	T	F	T	$.7 \times .6 = .42$
F	F	.1	F	F	.4	T	F	F	$.7 \times .4 = .28$
						F	T	T	$.9 \times .2 = .18$
						F	T	F	$.9 \times .8 = .72$
						F	F	T	$.1 \times .6 = .06$
						F	F	F	$.1 \times .4 = .04$

Figure 14.10 Illustrating pointwise multiplication: $\mathbf{f}_1(A,B) \times \mathbf{f}_2(B,C) = \mathbf{f}_3(A,B,C)$.

Example:

to sum out A from $\boxed{\mathbf{f}_3(A, B, C)}$:

$$\mathbf{f}(B,C) = \boldsymbol{\Sigma}_a\, \mathbf{f}_3(A, B, C) =$$

$$\mathbf{f}_3(a, B, C) + \mathbf{f}_3(\neg a, B, C)$$

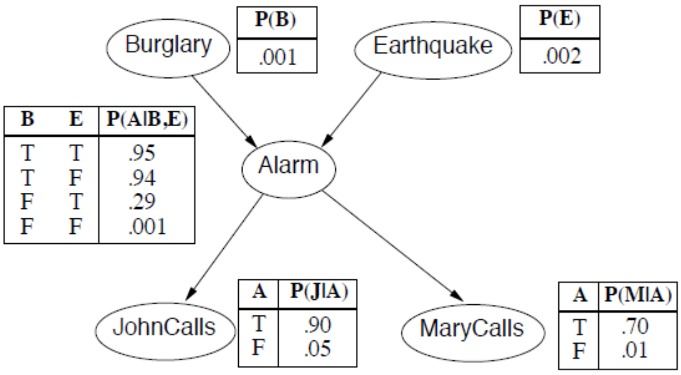
$$\begin{pmatrix} 0.06 & 0.24 \\ 0.42 & 0.28 \end{pmatrix} + \begin{pmatrix} 0.18 & 0.72 \\ 0.06 & 0.04 \end{pmatrix} = \begin{pmatrix} 0.24 & 0.96 \\ 0.48 & 0.32 \end{pmatrix}$$

5.2.3. Complexity of Exact Inference

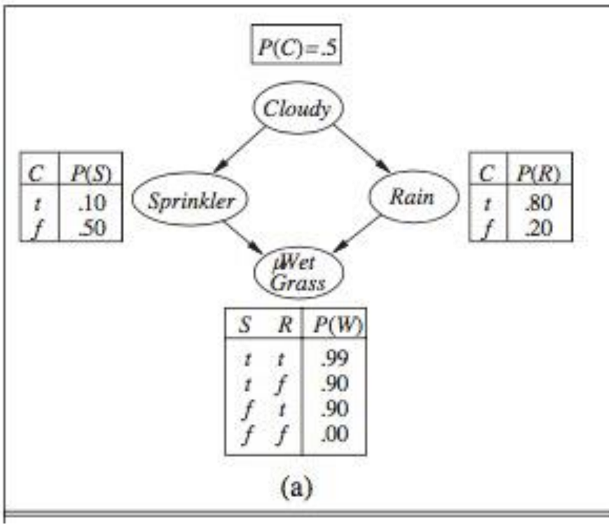
Complexity of exact inference depends on the type of network.

Singly connected networks (or polytree)

- Any two nodes are connected by at most one (undirected) path.
- Time and space complexity: **linear in the size of the network** (size = number of CPT entries)
- if the number of parents of each node is bounded by a constant =>**linear in the number of nodes**



Multiply connected networks: Exponential time and space complexity in the worst case, even when the number of parents per node is bounded.



In general exact inference is expensive. So approximate inference methods can be used.

Summary

- Bayesian networks (Topology + CPTs) = compact representation of joint distribution
- Exact inference: enumeration, variable elimination

6. Planning

Planning is about how an agent achieves its goals.

Assumptions:

- Agent's actions are deterministic. The agent can predict the consequences of its actions.
- No exogenous events beyond the control of the agent that change the state of the world.
- The world is fully observable.
- Time progresses discretely from one state to the next

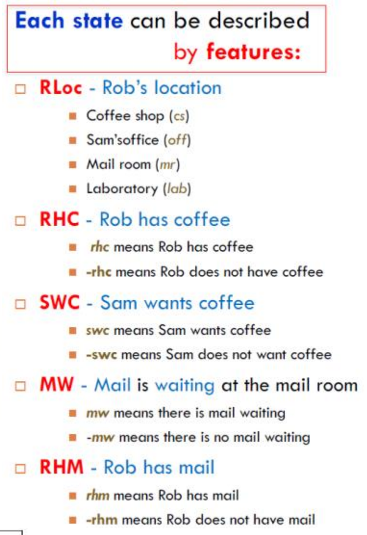
To reason about what to do, an agent must have:

- Goals
- A model of the world
- A model of the consequences of actions

Action:

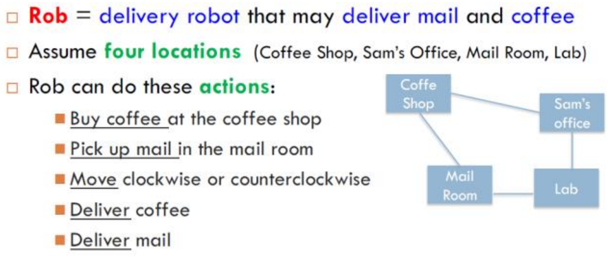
- **Partial** function from states to states.
  Partial means that not every action can be done in every state.
  Eg., a robot cannot carry out the action to pick up a particular object if it is nowhere near the object.
- **Precondition** of an action: specifies when the action can be done
- **Effect** of an action: specifies the resulting state

Example:

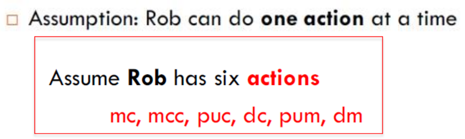


Each state can be described by **features:**

- □ **RLoc** - Rob's location
  - Coffee shop (*cs*)
  - Sam'soffice (*off*)
  - Mail room (*mr*)
  - Laboratory (*lab*)
- □ **RHC** - Rob has coffee
  - *rhc* means Rob has coffee
  - *-rhc* means Rob does not have coffee
- □ **SWC** - Sam wants coffee
  - *swc* means Sam wants coffee
  - *-swc* means Sam does not want coffee
- □ **MW** - Mail is waiting at the mail room
  - *mw* means there is mail waiting
  - *-mw* means there is no mail waiting
- □ **RHM** - Rob has mail
  - *rhm* means Rob has mail
  - *-rhm* means Rob does not have mail

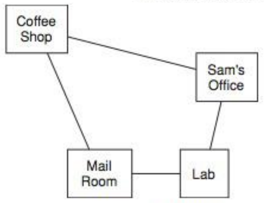


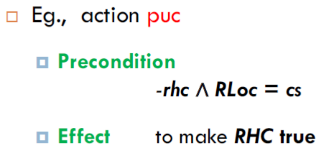
- □ **Rob** = delivery robot that may deliver mail and coffee
- □ Assume **four locations** (Coffee Shop, Sam's Office, Mail Room, Lab)
- □ Rob can do these **actions:**
  - Buy coffee at the coffee shop
  - Pick up mail in the mail room
  - Move clockwise or counterclockwise
  - Deliver coffee
  - Deliver mail



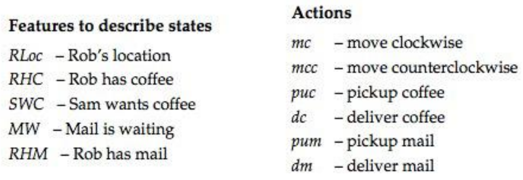
- □ Assumption: Rob can do **one action** at a time

Assume **Rob** has six **actions**
mc, mcc, puc, dc, pum, dm





- □ Eg., action puc
  - **Precondition**
    -rhc ∧ *RLoc* = *cs*
  - **Effect** to make *RHC* **true**



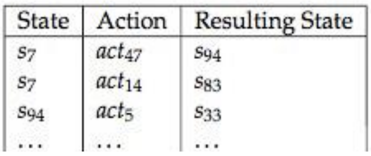
Features to describe states

RLoc – Rob's location
RHC – Rob has coffee
SWC – Sam wants coffee
MW – Mail is waiting
RHM – Rob has mail

Actions

mc	– move clockwise
mcc	– move counterclockwise
puc	– pickup coffee
dc	– deliver coffee
pum	– pickup mail
dm	– deliver mail

6.1. Representations



State	Action	Resulting State
s_7	act_{47}	s_{94}
s_7	act_{14}	s_{83}
s_{94}	act_5	s_{33}
...	...	...

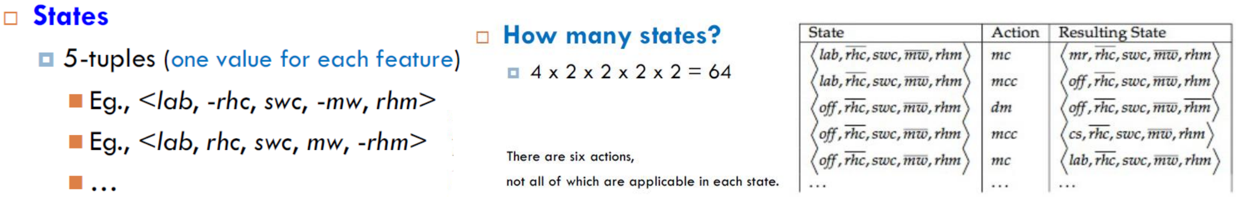
Explicit State-Space Representation (state-space graph**)**

- Explicitly enumerate all the states
- For each state specify all the actions possible in that state
- For each (state, action) specify the resulting state.

It's a Bad representation:

- Usually too many states to consider
- Small changes to the model implies large change to the representation
- Modeling another feature means changing the whole representation
  Eg., to model the level of power in the robot, so that it can recharge itself in the Lab every state has to change.

Example:



□ **States**

- □ 5-tuples (one value for each feature)
  - ■ Eg., <*lab*, -*rhc*, *swc*, -*mw*, *rhm*>
  - ■ Eg., <*lab*, *rhc*, *swc*, *mw*, -*rhm*>
  - ■ …

□ **How many states?**

- □ 4 x 2 x 2 x 2 x 2 = 64

There are six actions,
not all of which are applicable in each state.

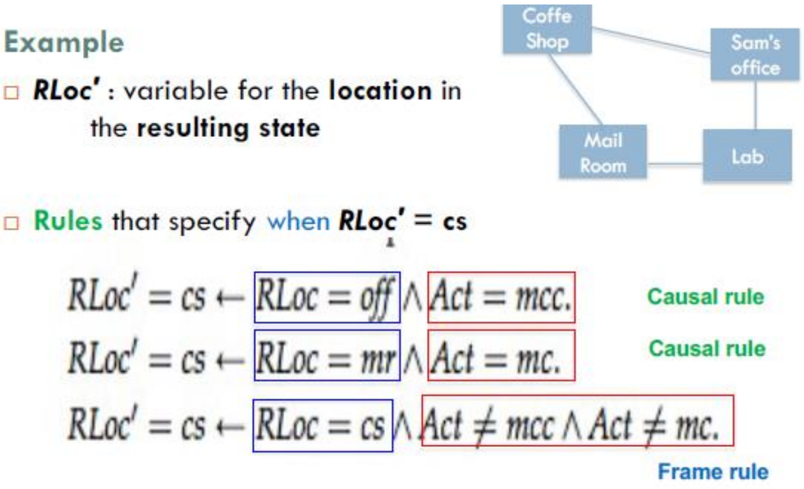
State	Action	Resulting State
$\langle lab, \overline{rhc}, swc, \overline{mw}, rhm \rangle$	mc	$\langle mr, \overline{rhc}, swc, \overline{mw}, rhm \rangle$
$\langle lab, \overline{rhc}, swc, \overline{mw}, rhm \rangle$	mcc	$\langle off, \overline{rhc}, swc, \overline{mw}, rhm \rangle$
$\langle off, \overline{rhc}, swc, \overline{mw}, rhm \rangle$	dm	$\langle off, \overline{rhc}, swc, \overline{mw}, \overline{rhm} \rangle$
$\langle off, \overline{rhc}, swc, \overline{mw}, rhm \rangle$	mcc	$\langle cs, \overline{rhc}, swc, \overline{mw}, rhm \rangle$
$\langle off, \overline{rhc}, swc, \overline{mw}, rhm \rangle$	mc	$\langle lab, \overline{rhc}, swc, \overline{mw}, rhm \rangle$
...	...	...

Feature-Based Representation of Actions

- It models the possible actions in a state in terms of values of the features of the state, and how feature values in the next state are affected by features values of the current state and action.
- Is Feature-centric: for each feature, there are rules that specify its value in the state resulting from an action.
- **Precondition** of an action: proposition that must be true to do the action.
- **Rules**: specify the value of each variable in the state resulting from an action
  Body of the rules:
  - ○ Features values in the previous state
  - ○ Action carried out.

  Rules have two forms:
  - ○ **Causal rule** specifies when a feature gets a new value.
  - ○ **Frame rule** specifies when a feature keeps its value.

Example:



Example

□ ***RLoc'*** : variable for the **location** in the **resulting state**

□ **Rules** that specify **when** ***RLoc'*** **= cs**

$$RLoc' = cs \leftarrow \boxed{RLoc = off} \wedge \boxed{Act = mcc.}$$

Causal rule

$$RLoc' = cs \leftarrow \boxed{RLoc = mr} \wedge \boxed{Act = mc.}$$

Causal rule

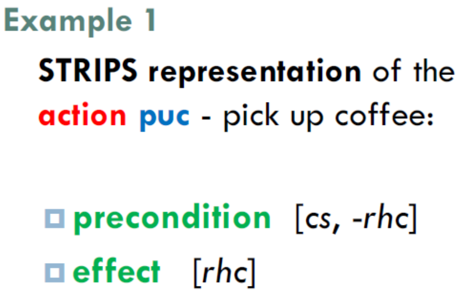
$$RLoc' = cs \leftarrow \boxed{RLoc = cs} \wedge \boxed{Act \neq mcc \wedge Act \neq mc.}$$

Frame rule

STRIPS representation:

- Is action-centric: for each action, specifies the effect of the action.
- **Features** are divided in:
    - **Primitive features**
    - **Derived features**

      Definite clauses determine the value of derived features from the values of the primitive features in any given state.
- STRIPS is **used to determine** values of **primitive features** in a state based on the Previous state and the Action taken by the agent.
- Idea: most things are not affected by a single action
- For each action, STRIPS models:
    - When the action is possible
    - What primitive features are affected by the action.
    - The effect of an action relies on the **STRIPS assumption:** All of the primitive features not mentioned in the description of the action stay unchanged.
- The **STRIPS representation for an action:**
    - **Precondition**: set of assignments of values to features that must be true for the action to occur.
    - **Effect**: set of resulting assignments of values to those primitive features that change as the result of the action.

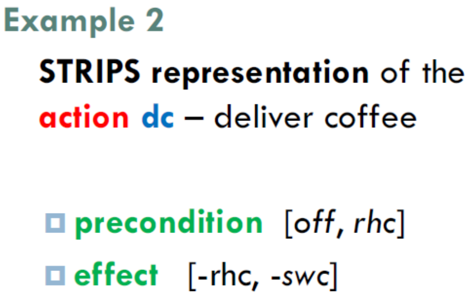


Example 1

STRIPS representation of the **action puc** - pick up coffee:

□ **precondition** [*cs*, -*rhc*]

□ **effect** [*rhc*]



Example 2

STRIPS representation of the **action dc** – deliver coffee

□ **precondition** [*off*, *rhc*]

□ **effect** [-rhc, -*swc*]

Feature-based vs STRIPS

- **Feature-based representation:**
    - **more verbose** than STRIPS representation
    - **more powerful** than STRIPS representation:
        - It can represent anything representable in STRIPS
        - Can specify conditional effects whereas STRIPS cannot represent these directly.

          Conditional effect = an effect of an action that depends on the value of other features
- **STRIPS** representation of a set of actions can be translated into the feature-based representation.

6.2. Planning problem

Assumption: fully observable and deterministic world

Initial states: a value for each feature for the initial time
2 Kinds of **goals**:

- **Achievement goal:** proposition that must be true in final state.
- **Maintenance goal:** proposition that must be true in every state through which the agent passes.

Other kinds of goals:

- **Transient goals** (that must be achieved somewhere in the plan but do not have to hold at the end)
- **Resource goals** (such as wanting to minimize energy used)

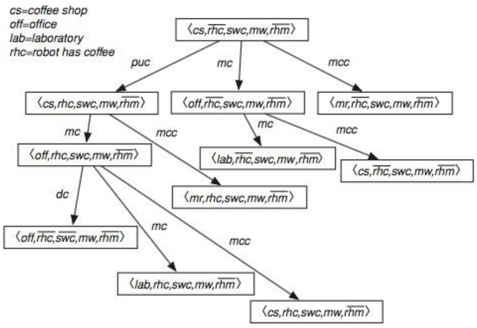
Deterministic plan: sequence of actions to achieve a goal from a given starting state.
Deterministic planner: a problem solver that can produce a plan.
Input of a planner:

- Initial world description
- Specification of the actions available to the agent (preconditions, effects)
- Goal description

Planning problem can be seen as a path planning problem in the state-space graph where:

- Nodes are states.
- Arcs correspond to actions from one state to another.
- ∀ state s, there is an arc for each action a such that:
  - precondition of a holds in state s
  - resulting state does not violate a maintenance goal.

Plan: is a path from the initial state to a state that satisfies the achievement goal.
Forward planner searches the state-space graph from the initial state looking for a state that satisfies a goal description. It can use any of the search strategies described in Chapter 2.
Example state-space graph:


cs=coffee shop
off=office
lab=laboratory
rhc=robot has coffee
$\langle cs,\overline{rhc},swc,mw,\overline{rhm}\rangle$
puc
mc
mcc
$\langle cs,rhc,swc,mw,\overline{rhm}\rangle$
$\langle off,\overline{rhc},swc,mw,\overline{rhm}\rangle$
$\langle mr,\overline{rhc},swc,mw,\overline{rhm}\rangle$
mc
mcc
mc
mcc
$\langle off,rhc,swc,mw,\overline{rhm}\rangle$
$\langle lab,\overline{rhc},swc,mw,\overline{rhm}\rangle$
$\langle cs,\overline{rhc},swc,mw,\overline{rhm}\rangle$
$\langle mr,rhc,swc,mw,\overline{rhm}\rangle$
dc
mc
mcc
$\langle off,\overline{rhc},\overline{swc},mw,\overline{rhm}\rangle$
$\langle lab,rhc,swc,mw,\overline{rhm}\rangle$
$\langle cs,rhc,swc,mw,\overline{rhm}\rangle$

Complexity of the search space:

- The complexity of the search space is defined by the forward branching factor of the graph
- The branching factor is the set of all possible actions at any state, which may be quite large
- When the domain becomes bigger, the branching factor increases and the search space explodes.
- This complexity may be reduced by finding good heuristics.

Planning as a CSP: It is possible to convert a planning problem to a constraint satisfaction problem (CSP). Thus one of the CSP methods can be used.

QUESTIONS FROM PAST EXAMS

You can find flashcards of those questions on: https://www.brainscape.com/p/3YF5P-LH-AUGGM

Agents

- Provide the definition of agent, agent function, and agent program.
- Indicare la differenza tra funzione agente e programma agente.

Search:

- Describe the simple problem-solving agent providing both the pseudocode and an informal description.
- Provide the formal definition of search problem (describe its components), solutions, and optimal solution in a search problem.
- Provide the formal definition of a solution in a search problem.
- Describe Breadth-first search and Depth-first search. What are the main differences between them?
- **EXERCISE**: Considerare uno spazio degli stati dove lo stato iniziale è il numero 1 e ogni stato k ha due successori: i numeri 2k e 2k+1. Disegnare la porzione di spazio degli stati per gli stati da 1 a 7. Supponiamo che lo stato obiettivo sia lo stato 5. Elencare i nodi nell'ordine in cui saranno visitati con **Breadth-first search, Depth-limited search with limit** 2, **Iterative deepening search**.
- Describe greedy best-First search.
- Describe algorithm A*.
- What is the difference between A* and the greedy algorithm?
- Dare la definizione di euristica consistente e di euristica ammissibile.
- **EXERCISE**: Si consideri il seguente grafo, dove S `e lo stato iniziale e G `e lo stato goal. Il costo di ciascun arco `e la sua etichetta. In tabella viene riportata la stima euristica della distanza di ciascun nodo dal nodo goal G. **Indicare l'ordine con cui la strategia di ricerca A*** espande i nodi alla ricerca di una soluzione e si mostri l'albero di ricerca espanso. In caso di più nodi figli, a parità di altre condizioni, si espanda per primo quello il cui nome precede alfabeticamente il nome degli altri figli. Mostrare la soluzione trovata. La funzione euristica h(n) presentata in questo esercizio `e ammissibile? È consistente? Motivare le risposte.
- Describe hill climbing search providing both the pseudocode and an informal description.
- Descrivere l'algoritmo hill climbing. Spiegare per quali ragioni spesso l'hill climbing rimane bloccato.
- Describe the local beam search algorithm.

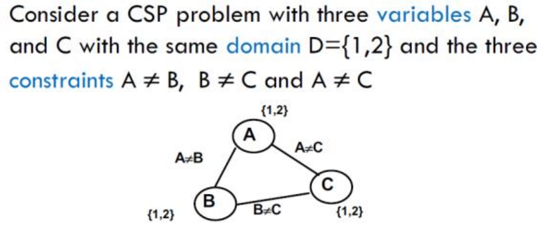
Stable Matching Problems:

- **EXERCISE**: applicare **algoritmo Gale Shapley**. Qual è il matching risultante? Quali sono le proprietà del matching ottenuto? Il matching {(R1; T2); (R2; T1); (R3; T3)} `e stabile? Motivare la risposta.
- **EXERCISE**: In this particular stable matching problem is there a man or woman that can profit from lying?
- What is a stable matching problem? What is a matching? What does it mean that a matching is stable?
- Dare la definizione di blocking pair.

- Describe Gale-Shapley algorithm (GS).

CSP:

- Provide the formal definition of a constraint satisfaction problem (CSP)
- Dare la definizione di soluzione di un CSP
- Che cosa `e un vincolo globale? Fare un esempio.
- Constraints involving more than three variables are more expressive than constraints involving only two variables? Motivate your answer.
- Descrivere l'euristica MRV (minimum remaining values) per i problemi di soddisfacimento di vincoli. Tale euristica viene anche chiamata euristica First-Fail.
- Descrivere l'euristica di grado (degree heuristic).
- Assume there is a binary constraint between the variables X and Y. What does it mean that X is arc consistent w.r.t. Y?
  ANS: It means that for each value x in the domain of X, there is some value y in the domain of Y that satisfies the constraint between X and Y.
- How can we enforce X to be arc-consistency w.r.t. Y?
  ANS: Remove all the values x in the domain of X for which there is no corresponding value y in the domain of Y that satisfies the constraint between X and Y.
- What are the possible outcomes of the arc consistency algorithm? Explain also how these outcomes are related to the set of the solutions.
  ANS:
  - At least one domain could be empty, in which case there is no solution
  - Each domain could have a single value, in which case there is a unique solution
  - Or some domains could have multiple values
- Consider the case where the arc consistency algorithm applied to a CSP terminates and some domains have multiple values. Is there guaranteed to be a solution? Justify the answer.
- Provide an example of a constraint satisfaction problem which is arc consistent but with no solution.



Consider a CSP problem with three variables A, B, and C with the same domain D={1,2} and the three constraints A ≠ B, B ≠ C and A ≠ C

- What does it mean that a two-variable set {Xi;Xj} in a CSP is path-consistent with respect to a third variable Xm.
- Indicare quando il grafo dei vincoli `e un albero.
- Dare la definizione formale di directed arc consistency.
- Che cosa `e la tree width di una tree decomposition?
- **EXERCISE**: Considerare un problema di colorazione della mappa… Disegnare il grafo dei vincoli e mostrare come cambiano i domini delle variabili quando applichiamo **Forward checking** a questo problema dopo aver assegnato alla variabile X1 il valore 1.
- **EXERCISE**: Define a map coloring problem with 5 variables where each variable can have values in D = red; blu; green. Apply **backtracking with forward checking** on this problem. **Apply arc consistency**.

Soft-CSP:

- Provide the formal definition of soft constraint.
- What is a soft constraint satisfaction problem (SCSP)? What is an optimal solution of an SCSP?
- **EXERCISE**: Si consideri un **weighted**…. Definire il vincolo weighted ottenuto combinando tutti i vincoli presenti in figura. Assumendo che il vincolo weighted ottenuto sia l'unico vincolo di un weighted CSP, indicare la soluzione ottima (o le soluzioni ottime). Giustificare la risposta.
- **EXERCISE**: Si consideri un **fuzzy CSP**… Disegnare il vincolo fuzzy ottenuto combinando tutti i vincoli presenti in figura mostrando tutti i passaggi. Assumendo che il vincolo ottenuto sia l'unico vincolo di un fuzzy CSP, indicare la soluzione ottima (o le soluzioni ottime) e il valore di preferenza di una soluzione ottima. Giustificare la risposta.

Multi agent decision making:

- Quale tipo di preferenze possono modellare i vincoli soft? Quale tipo di preferenze possono modellare le CP-net?
- Describe how Plurality rule works.
- Describe how Borda rule works. Is it possible to manipulate it?
- What are the main differences between soft constraint formalism and CP-nets?
- What is a conditional preference table in a CP-net?
- Descrivere l'algoritmo per trovare una soluzione ottima in una CP-net aciclica.

Bayessian networks:

- Describe the syntax and the semantics of a Bayesian network.
- What is a conditional probability table in a Bayesian network?
- What are the similarities and the differences between Bayesian networks and CP-nets?
- **EXERCISE**: Si consideri la rete bayesiana indicata in figura. Calcolare il valore di P(a; b;-c; d;-e).
- **EXERCISE**: Consider the BN shown in the figure below….. compute the probability P(d|b) by enumeration (you don't need to compute the normalization factor alpha).

Planning:

- What is a causal rule? What is a frame rule?
- The STRIPS representation for an action consists of what?
  ANS:
  - Preconditions - a set of assignments of values to variables that must be true for the action to occur
  - Effects - a set of resulting assignments of values to those variables that change as the result of the action
- What is the STRIPS assumption?
  ANS: All of the variables not mentioned in the description of an action stay unchanged when the action is carried out.

Other:

- What is the difference between supervised and unsupervised learning?
- What is k-fold cross validation?
- A Turing machine passes the Turing test if it can discriminate between a machine and a human through interaction? Motivate your answer.