# Fundamentals of Artificial Intelligence and Knowledge Representation (Module 1)

Last update: 17 December 2023

Academic Year 2023 – 2024
Alma Mater Studiorum · University of Bologna

# Contents

*Not required for the exam*

# 1 Introduction

## 1.1 AI systems classification

### 1.1.1 Intelligence classification

Intelligence is defined as the ability to perceive or infer information and to retain the knowledge for future use.

**Weak AI** aims to build a system that acts as an intelligent system. Weak AI

**Strong AI** aims to build a system that is actually intelligent. Strong AI

### 1.1.2 Capability classification

**General AI** systems able to solve any generalized task. General AI

**Narrow AI** systems able to solve a particular task. Narrow AI

### 1.1.3 AI approaches

**Symbolic AI (top-down)** Symbolic representation of knowledge, understandable by humans. Symbolic AI

**Connectionist approach (bottom up)** Neural networks. Knowledge is encoded and not understandable by humans. Connectionist approach

## 1.2 Symbolic AI

**Deductive reasoning** Conclude something given some premises (general to specific). It is unable to produce new knowledge. Deductive reasoning

**Example.** "All men are mortal" and "Socrates is a man" → "Socrates is mortal"

**Inductive reasoning** A conclusion is derived from an observation (specific to general). Produces new knowledge, but correctness is not guaranteed. Inductive reasoning

**Example.** "Several birds fly" → "All birds fly"

**Abduction reasoning** An explanation of the conclusion is found from known premises. Differently from inductive reasoning, it does not search for a general rule. Produces new knowledge, but correctness is not guaranteed. Abduction reasoning

**Example.** "Socrates is dead" (conclusion) and "All men are mortal" (knowledge) → "Socrates is a man"

**Reasoning by analogy** Principle of similarity (e.g. k-nearest-neighbor algorithm). Reasoning by analogy

**Example.** "Socrates loves philosophy" and Socrates resembles John → "John loves philosophy"

**Constraint reasoning and optimization** Constraints, probability, statistics. Constraint reasoning

*Not required for the exam*

## 1.3 Machine learning

### 1.3.1 Training approach

Supervised learning

**Supervised learning** Trained on labeled data (ground truth is known). Suitable for classification and regression tasks.

Unsupervised learning

**Unsupervised learning** Trained on unlabeled data (the system makes its own discoveries). Suitable for clustering and data mining.

Semi-supervised learning

**Semi-supervised learning** The system is first trained to synthesize data in an unsupervised manner, followed by a supervised phase.

Reinforcement learning

**Reinforcement learning** An agent learns by simulating actions in an environment with rewards and punishments depending on its choices.

### 1.3.2 Tasks

Classification

**Classification** Supervised task that, given the input variables $X$ and the output (discrete) categories $Y$, aims to approximate a mapping function $f : X \to Y$.

Regression

**Regression** Supervised task that, given the input variables $X$ and the output (continuous) variables $Y$, aims to approximate a mapping function $f : X \to Y$.

Clustering

**Clustering** Unsupervised task that aims to organize objects into groups.

### 1.3.3 Neural networks

Perceptron

A neuron (**perceptron**) computes a weighted sum of its inputs and passes the result to an activation function to produce the output.

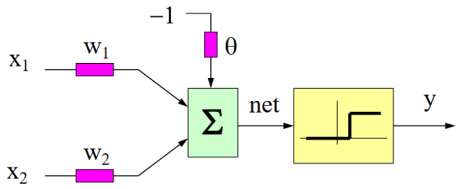



Figure 1.1: Representation of an artificial neuron

Feed-forward neural network

A **feed-forward neural network** is composed of multiple layers of neurons, each connected to the next one. The first layer is the input layer, while the last is the output layer. Intermediate layers are hidden layers.
The expressivity of a neural networks increases when more neurons are used:

**Single perceptron** Able to compute a linear separation.

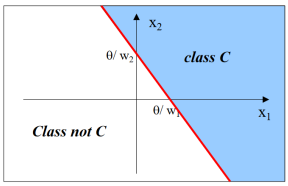



Figure 1.2: Separation performed by one perceptron

*Not required for the exam*

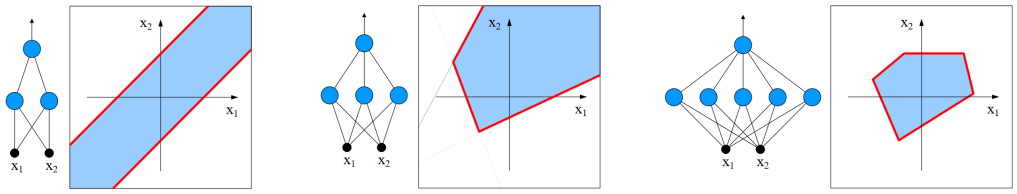



Figure 1.3: Separation performed by a three-layer network

**Three-layer network** Able to separate a convex region ($n_{\text{edges}} \leq n_{\text{hidden neurons}}$)

**Four-layer network** Able to separate regions of arbitrary shape.

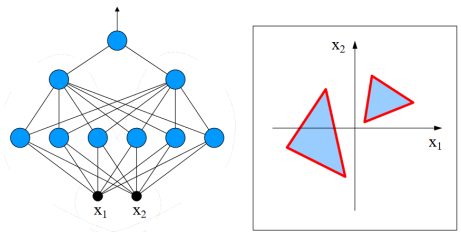



Figure 1.4: Separation performed by a four-layer network

**Theorem 1.3.1** (Universal approximation theorem)**.** A feed-forward network with one hidden layer and a finite number of neurons is able to approximate any continuous function with desired accuracy.

Universal approximation theorem

**Deep learning** Neural network with a large number of layers and neurons. The learning process is hierarchical: the network exploits simple features in the first layers and synthesis more complex concepts while advancing through the layers.

Deep learning

## 1.4 Automated planning

Given an initial state, a set of actions and a goal, **automated planning** aims to find a partially or totally ordered sequence of actions to achieve a goal.
An **automated planner** is an agent that operates in a given domain described by:

Automated planning

- Representation of the initial state
- Representation of a goal
- Formal description of the possible actions (preconditions and effects)

## 1.5 Swarm intelligence

Decentralized and self-organized systems that result in emergent behaviors.

Swarm intelligence

## 1.6 Decision support systems

**Knowledge based system** Use knowledge (and data) to support human decisions. Bottlenecked by knowledge acquisition.

Knowledge based system

*Not required for the exam*

Different levels of decision support exist:

**Descriptive analytics** Data are used to describe the system (e.g. dashboards, reports, ...). Human intervention is required.

Descriptive analytics

**Diagnostic analytics** Data are used to understand causes (e.g. fault diagnosis) Decisions are made by humans.

Diagnostic analytics

**Predictive analytics** Data are used to predict future evolutions of the system. Uses machine learning models or simulators (digital twins)

Predictive analytics

**Prescriptive analytics** Make decisions by finding the preferred scenario. Uses optimization systems, combinatorial solvers or logical solvers.

Prescriptive analytics

# 2 Search problems

## 2.1 Search strategies

**Solution space** Set of all the possible sequences of actions an agent may apply. Some of these lead to a solution. Solution space

**Search algorithm** Takes a problem as input and returns a sequence of actions that solves the problem (if exists). Search algorithm

### 2.1.1 Search tree

**Expansion** Starting from a state, apply a successor function and generate a new state. Expansion

**Search strategy** Choose which state to expand. Usually is implemented using a fringe that decides which is the next node to expand. Search strategy

**Search tree** Tree structure to represent the expansion of all states starting from a root (i.e. the representation of the solution space). Search tree

Nodes are states and branches are actions. A leaf can be a state to expand, a solution or a dead-end. Algorithm 1 describes a generic tree search algorithm.

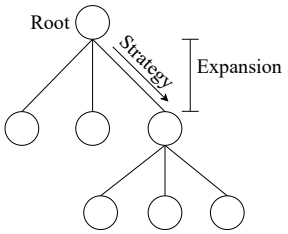



Figure 2.1: Search tree

Each node contains:

- The state
- The parent node
- The action that led to this node
- The depth of the node
- The cost of the path from the root to this node

### 2.1.2 Strategies

**Non-informed strategy** Domain knowledge not available. Usually does an exhaustive search. Non-informed strategy

**Informed strategy** Use domain knowledge by using heuristics. Informed strategy

**Algorithm 1** Tree search

```
def treeSearch(problem, fringe):
    fringe.push(problem.initial_state)
    # Get a node in the fringe and expand it if it is not a solution
    while fringe.notEmpty():
        node = fringe.pop()
        if problem.isGoal(node.state):
            return node.solution
        fringe.pushAll(expand(node, problem))
    return FAILURE

def expand(node, problem):
    successors = set()
    # List all neighboring nodes
    for action, result in problem.successor(node.state):
        s = new Node(
            parent=node, action=action, state=result, depth=node.dept+1,
            cost=node.cost + problem.pathCost(node, s, action)
        )
        successors.add(s)
    return successors
```

### 2.1.3 Evaluation

**Completeness** if the strategy is guaranteed to find a solution (when exists). — Completeness

**Time complexity** time needed to complete the search. — Time complexity

**Space complexity** memory needed to complete the search. — Space complexity

**Optimality** if the strategy finds the best solution (when more solutions are possible). — Optimality

## 2.2 Non-informed search

### 2.2.1 Breadth-first search (BFS)

Always expands the least deep node. The fringe is implemented as a queue (FIFO). — Breadth-first search

| | |
|---|---|
| **Completeness** | Yes |
| **Optimality** | Only with uniform cost (i.e. all edges have same cost) |
| **Time and space complexity** | $O(b^d)$, where the solution depth is $d$ and the branching factor is $b$ (i.e. each non-leaf node has $b$ children) |

The exponential space complexity makes BFS impractical for large problems.

### 2.2.2 Uniform-cost search

Same as BFS, but always expands the node with the lowest cumulative cost. — Uniform-cost search

| | |
|---|---|
| **Completeness** | Yes |
| **Optimality** | Yes |
| **Time and space complexity** | $O(b^d)$, with solution depth $d$ and branching factor $b$ |

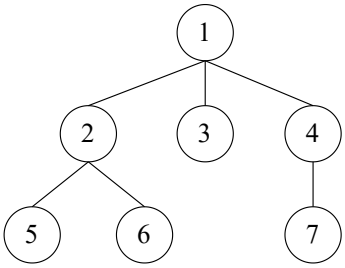



Figure 2.2: BFS visit order

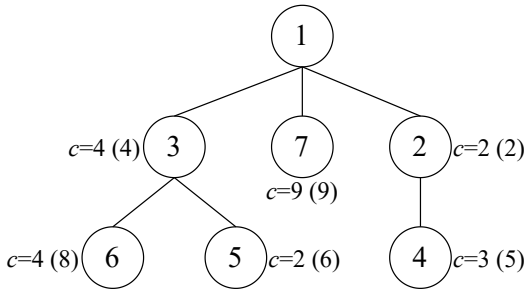



Figure 2.3: Uniform-cost search visit order. $(n)$ is the cumulative cost

### 2.2.3 Depth-first search (DFS)

Always expands the deepest node. The fringe is implemented as a stack (LIFO). Depth-first search

| | |
|---|---|
| **Completeness** | No (loops) |
| **Optimality** | No |
| **Time complexity** | $O(b^m)$, with maximum depth $m$ and branching factor $b$ |
| **Space complexity** | $O(b \cdot m)$, with maximum depth $m$ and branching factor $b$ |

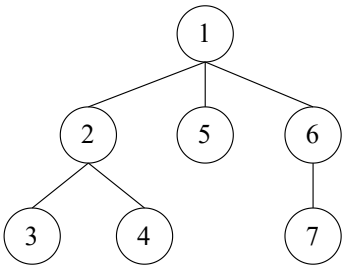



Figure 2.4: DFS visit order

### 2.2.4 Depth-limited search

Same as DFS, but introduces a maximum depth. A node at the maximum depth will not be explored further. Depth-limited search
This allows to avoid infinite branches (i.e. loops).

### 2.2.5 Iterative deepening

Iterative deepening

Runs a depth-limited search by trying all possible depth limits. It is important to note that each iteration is executed from scratch (i.e. a new execution of depth-limited search).

**Algorithm 2** Iterative deepening

```
def iterativeDeepening(G):
    for c in range(G.max_depth):
        sol = depthLimitedSearch(G, c)
        if sol is not FAILURE:
            return sol
    return FAILURE
```

Both advantages of DFS and BFS are combined.

| | |
|---|---|
| **Completeness** | Yes |
| **Optimality** | Only with uniform cost |
| **Time complexity** | $O(b^d)$, with solution depth $d$ and branching factor $b$ |
| **Space complexity** | $O(b \cdot d)$, with solution depth $d$ and branching factor $b$ |

## 2.3 Informed search

Informed search

Informed search uses evaluation functions (heuristics) to reduce the search space and estimate the effort needed to reach the final goal.

### 2.3.1 Best-first search

Best-first seacrh

Uses heuristics to compute the desirability of the nodes (i.e. how close they are to the goal). The fringe is ordered according the estimated scores.

Greedy search / Hill climbing

**Greedy search / Hill climbing** The heuristic only evaluates nodes individually and does not consider the path to the root (i.e. expands the node that currently seems closer to the goal).

| | |
|---|---|
| **Completeness** | No (loops) |
| **Optimality** | No |
| **Time and space complexity** | $O(b^d)$, with solution depth $d$ and branching factor $b$ |

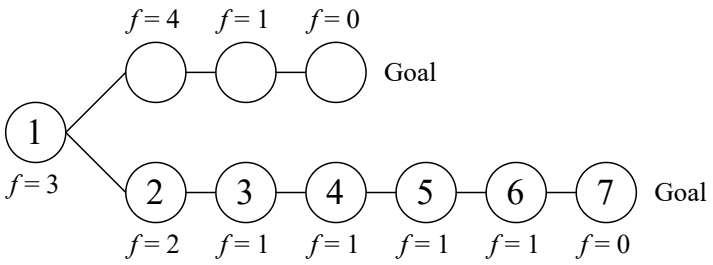



Figure 2.5: Hill climbing visit order

**A*** The heuristic also considers the cumulative cost needed to reach a node from the root. The score associated to a node $n$ is: A*

$$f(n) = g(n) + h'(n)$$

where $g$ is the depth of the node and $h'$ is the heuristic that computes the distance to the goal.

**Optimistic/Feasible heuristic** Given $t(n)$ that computes the true distance of a node $n$ to the goal. An heuristic $h'(n)$ is optimistic (i.e. feasible) if: Optimistic/Feasible heuristic

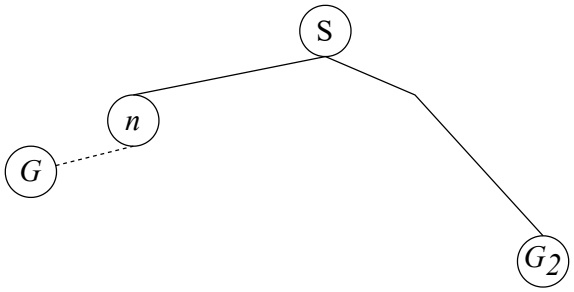
$$h'(n) \leq t(n)$$

In other words, $h'$ is optimistic if it always underestimates the distance to the goal.

**Theorem 2.3.1.** If the heuristic used by A* is optimistic ⇒ A* is optimal

*Proof.* Consider a scenario where the queue contains:

- A node $n$ whose child is the optimal solution
- A sub-optimal solution $G_2$




We want to prove that A* will always expand $n$.

Given an optimistic heuristic $f(n) = g(n) + h'(n)$ and the true distance of a node $n$ to the goal $t(n)$, we have that:

$$f(G_2) = g(G_2) + h'(G_2) = g(G_2), \text{ as } G_2 \text{ is a solution: } h'(G_2) = 0$$
$$f(G) = g(G) + h'(G) = g(G), \text{ as } G \text{ is a solution: } h'(G) = 0$$

Moreover, $g(G_2) > g(G)$ as $G_2$ is suboptimal. Therefore, $\boldsymbol{f(G_2) > f(G)}$.

Furthermore, as $h'$ is feasible, we have that:

$$\begin{aligned} h'(n) \leq t(n) &\iff g(n) + h'(n) \leq g(n) + t(n) = g(G) = f(G) \\ &\iff \boldsymbol{f(n) \leq f(G)} \end{aligned}$$

In the end, we have that $f(G_2) > f(G) \geq f(n)$. So we can conclude that A* will never expand $G_2$ as:

$$f(G_2) > f(n)$$

□

| | |
|---|---|
| **Completeness** | Yes |
| **Optimality** | Only if the heuristic is optimistic |
| **Time and space complexity** | $O(b^d)$, with solution depth $d$ and branching factor $b$ |

In general, it is better to use heuristics with large values (i.e. heuristics that don't underestimate too much).

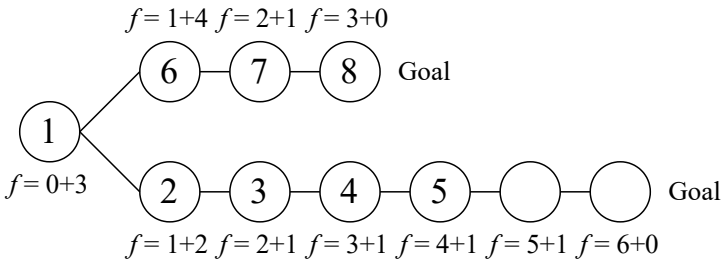



Figure 2.6: A* visit order

## 2.4 Graph search

Graph search

Differently from a tree search, searching in a graph requires to keep track of the explored nodes.

**Algorithm 3** Graph search

```
def graphSearch(problem, fringe):
    closed = set()
    fringe.push(problem.initial_state)
    # Get a node in the fringe and
    # expand it if it is not a solution and is not closed
    while fringe.notEmpty():
        node = fringe.pop()
        if problem.isGoal(node.state):
            return node.solution
        if node.state not in closed:
            closed.add(node.state)
            fringe.pushAll(expand(node, problem))
    return FAILURE
```

### 2.4.1 A* with graphs

A* with graphs

The algorithm keeps track of closed and open nodes. The heuristic $g(n)$ evaluates the minimum distance from the root to the node $n$.

Consistent heuristic (monotone)

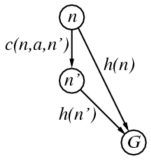
**Consistent heuristic (monotone)** An heuristic is consistent if for each $n$, for any successor $n'$ of $n$ (i.e. nodes reachable from $n$ by making an action) holds that:

$$\begin{cases} h(n) = 0 & \text{if the corresponding status is the goal} \\ h(n) \leq c(n,a,n') + h(n') & \text{otherwise} \end{cases}$$

where $c(n,a,n')$ is the cost to reach $n'$ from $n$ by taking the action $a$.

In other words, $f$ never decreases along a path. In fact:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$




**Theorem 2.4.1.** If $h$ is a consistent heuristic, A* on graphs is optimal.

# 3 Local search

**Local search** Starting from an initial state, iteratively improves it by making local moves in a neighborhood. Local search

Useful when the path to reach the solution is not important (i.e. no optimality).

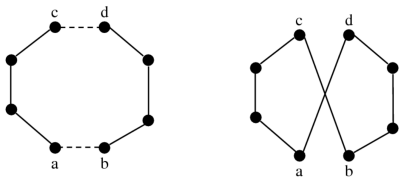
**Neighborhood** Given a set of states $\mathcal{S}$, a neighborhood is a function: Neighborhood

$$\mathcal{N} : S \to 2^{\mathcal{S}}$$

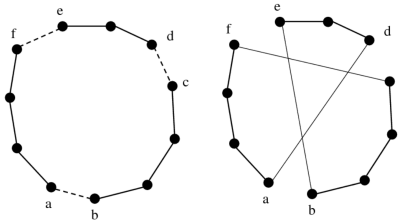
In other words, for each $s \in \mathcal{S}$, $\mathcal{N}(s) \subseteq \mathcal{S}$.

**Example** (Travelling salesman problem)**.** Problem: find an Hamiltonian tour of minimum cost in an undirected graph.

A possible neighborhood of a state applies the $k$-exchange that guarantees to maintain an Hamiltonian tour.




(a) 2-exchange




(b) 3-exchange

**Local optima** Given an evaluation function $f$, a local optima (maximization case) is a state $s$ such that:

$$\forall s' \in \mathcal{N}(s) : f(s) \geq f(s')$$

**Global optima** Given an evaluation function $f$, a global optima (maximization case) is a state $s_{\text{opt}}$ such that:

$$\forall s \in \mathcal{S} : f(s_{\text{opt}}) \geq f(s)$$

Note: a larger neighborhood usually allows to obtain better solutions.

**Plateau** Flat area of the evaluation function.

**Ridges** Higher area of the evaluation function that is not directly reachable.

## 3.1 Iterative improvement (hill climbing)

Algorithm that only performs moves that improve the current solution. Iterative improvement (hill climbing)
It does not keep track of the explored states (i.e. may return in a previously visited state) and stops after reaching a local optima.

**Algorithm 4** Iterative improvement

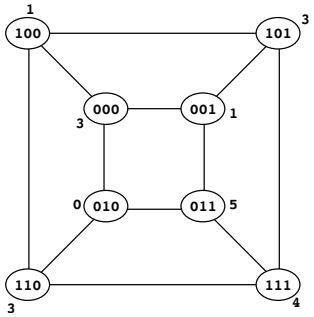
```
def iterativeImprovement(problem):
    s = problem.initial_state
    while noImprovement():
        s = bestOf(problem.neighborhood(s))
```

## 3.2 Meta heuristics

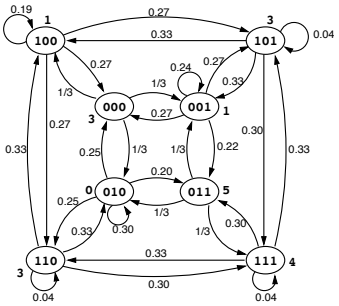
Methods that aim to improve the final solution. Can be seen as a search process over graphs: Meta heuristics

**Neighborhood graph** The search space topology.

**Search graph** The explored space.




(a) Neighborhood graph




(b) Search graph. Edges are probabilities

Meta heuristics aim to find a balance between:

**Intensification** Look for moves near the neighborhood. Intensification

**Diversification** Look for moves somewhere else. Diversification

Different termination criteria can be used with meta heuristics:

- Time constraints.
- Iterations limit.
- Absence of improving moves (stagnation).

### 3.2.1 Simulated annealing

Occasionally allow moves that worsen the current solution. The probability of this to happen is: Simulated annealing

$$\frac{e^{f(s)-f(s')}}{T}$$

where $s$ is the current state, $s'$ is the next move and $T$ is the temperature. The temperature is updated at each iteration and can be:

**Logarithmic** $T_{k+1} = \Gamma / \log(k + k_0)$

**Geometric** $T_{k+1} = \alpha T_k$, where $\alpha \in ]0, 1[$

**Non-monotonic** $T$ is alternatively decreased (intensification) and increased (diversification).

**Algorithm 5** Meta heuristics – Simulated annealing

```
def simulatedAnnealing(problem, T0):
    s = problem.initial_state
    T, k = T0, 0
    while not terminationConditions():
        s_next = randomOf(problem.neighborhood(s))
        if (problem.f(s_next) > problem.f(s) or
           downHillWithProbability(e^(problem.f(s) - problem.f(s_next)) / T)):
            s = s_next
        k += 1
        update(T, k)
```

### 3.2.2 Tabu search

Tabu search

Keep track of the last $n$ explored solutions in a tabu list and forbid them. Allows to escape from local optima and cycles.
Since keeping track of the visited solutions is inefficient, moves can be stored instead but, with this approach, some still not visited solutions may be cut off. **Aspiration criteria** can be used to allow forbidden moves in the tabu list to be evaluated.

**Algorithm 6** Meta heuristics – Tabu search

```
def tabuSearch(problem, T0):
    s = problem.initial_state
    tabu_list = [] # limited to n elements
    T, k = T0, 0
    while not terminationConditions():
        allowed_s = {s' ∈ N(s) : s' ∉ tabu_list or aspiration condition satisfied}
        s = bestOf(allowed_s)
        updateTabuListAndAspirationConditions()
        k += 1
        update(T, k)
```

### 3.2.3 Iterated local search

Iterated local search

Based on two steps:

**Subsidiary local search steps** Efficiently reach a local optima (intensification).

**Perturbation steps** Escape from a local optima (diversification).

In addition, an acceptance criterion controls the two steps.

**Algorithm 7** Meta heuristics – Iterated local search

```
def tabuSearch(problem):
    s = localSearch(problem.initial_state)
    while not terminationConditions():
        s_perturbation = perturbation(s, history)
        s_local = localSearch(s_perturbation)
        s = acceptanceCriterion(s, s_local, history)
```

### 3.2.4 Population based (genetic algorithm)



Population based meta heuristics are built on the following concepts:

**Adaptation** Organisms are suited to their environment.

**Inheritance** Offspring resemble their parents.

**Natural selection** Fit organisms have many offspring, others become extinct.

| Biology | Artificial intelligence |
|---|---|
| Individuals | Possible solution |
| Fitness | Quality |
| Environment | Problem |

Table 3.1: Biological evolution metaphors

The following terminology will be used:

**Population** Set of individuals (solutions).

**Genotypes** Individuals of a population.

**Genes** Units of chromosomes.

**Alleles** Domain of values of a gene.

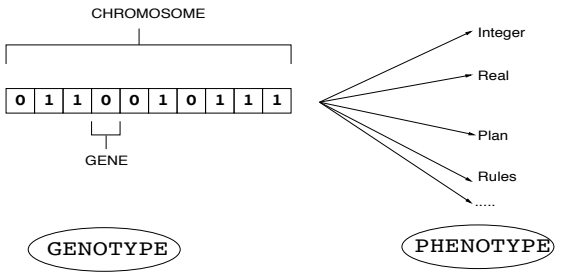
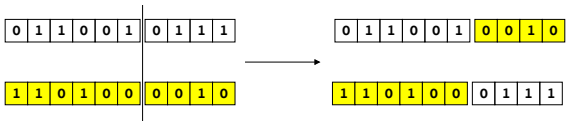


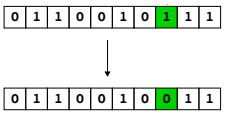

Figure 3.3

Genetic operators are:

**Recombination/Crossover** Cross-combination of two chromosomes.




**Mutation** Random modification of genes.




**Proportional selection** Probability of a individual to be chosen as parent of the next offspring. Depends on the fitness.

**Generational replacement** Create the new generation. Possibile approaches are:

- Completely replace the old generation with the new one.
- Keep the best $n$ individual from the new and old population.

**Example** (Real-valued genetic operators)**.** Solution $x \in [a, b]$ with $a, b \in \mathbb{R}$.

**Mutation** Random perturbation: $x \to x \pm \delta$, as long as $x \pm \delta \in [a, b]$.

**Crossover** Linear combination: $x = \lambda_1 y_1 + \lambda_2 y_2$, as long as $x \in [a, b]$.

**Example** (Permutation genetic operators)**.** Solution $x = (x_1, \ldots, x_n)$ is a permutation of $(1, \ldots, n)$.

**Mutation** Random exchange of two elements at index $i$ and $j$, with $i \neq j$.

**Crossover** Crossover avoiding repetitions.

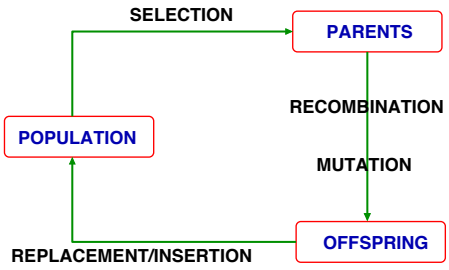



Figure 3.4: Evolutionary cycle

**Algorithm 8** Meta heuristics – Genetic algorithm

```
def geneticAlgorithm(problem):
    population = problem.initPopulation()
    evaluate(population)
    while not terminationConditions():
        offspring = []
        while not offspringComplete(offspring):
            p1, p2 = selectParents(population)
            new_individual = crossover(p1, p2)
            new_individual = mutation(new_individual)
            offspring.append(new_individual)
        population = offspring
        evaluate(population)
```

# 4 Swarm intelligence

**Swarm intelligence** Group of locally-interacting agents that shows an emergent behavior without a centralized control system. Swarm intelligence

A swarm intelligent system has the following features:

- Individuals are simple and have limited capabilities.
- Individuals are not aware of the global view.
- Individuals have local direct or indirect communication patterns.
- The computation is distributed and not centralized.
- The system works even if some individuals "break" (robustness).
- The system adapts to changes.

Agents interact between each other and obtain positive and negative feedbacks.

**Stigmergy** Form of indirect communication where an agent modifies the environment and the others react to it. Stigmergy

## 4.1 Ant colony optimization (ACO)

Ants release pheromones while walking from the nest to the food. They also tend to prefer paths marked with the highest pheromone concentration.

**Ant colony optimization** Probabilistic parametrized model that builds the solution incrementally. A problem is solved by making stochastic steps in a fully connected graph (construction graph) $G = (C, L)$ where: Ant colony optimization (ACO)

- The vertexes $C$ are the solution components.
- The edges $L$ are connections.
- The paths on $G$ are states.

Additional constraints may be added if needed.

**Example** (Travelling salesman)**.** The construction graph can be defined as:

- Nodes are cities.
- Edges are connections between cities.
- A solution is an Hamiltonian path in the graph.
- Constraints to avoid sub-cycles (i.e. avoid visiting a city multiple times).

**Pheromone** Value associated to each node and each edge to estimate the quality of the solution. Pheromone

**Heuristic values** Value associated to each node and each edge to represent the prior background knowledge. Heuristic values

### 4.1.1 ACO system

ACO system

**Transition rule** Ants build a path on the construction graph based on a transition rule that uses pheromones and heuristics. The probability of choosing the node $j$ starting from $i$ is parametrized on $\alpha$ (pheromones) and $\beta$ (heuristics), and is defined as:

$$p_{\alpha,\beta}(i,j) = \begin{cases} \frac{(\tau_{ij})^\alpha \cdot (\eta_{ij})^\beta}{\sum_{k \in \texttt{feasible\_nodes}} (\tau_{ik})^\alpha \cdot (\eta_{ik})^\beta} & \text{if j consistent} \\ 0 & \text{otherwise} \end{cases}$$

where $\tau_{ij}$ is the pheromone trail from $i$ to $j$ and $\eta_{ij} = \frac{1}{d_{ij}}$ is the heuristic ($d_{ij}$ is the distance).

**Pheromone update** After each step, the pheromone trail is updated depending on an evaporation factor $\rho \in [0, 1]$:

$$\tau_{ij} = (1-\rho)\tau_{ij} + \sum_{k=1}^{n_{\text{ants}}} \Delta\tau_{ij}^{(k)}$$

$\Delta\tau_{ij}^{(k)}$ of the $k$-th ant is defined as:

$$\Delta\tau_{ij}^{(k)} = \begin{cases} \frac{1}{L_k} & \text{if ant } k \text{ used the arch } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

where $L_k$ is the length of the path followed by the $k$-th ant.

For the best ant, the update also affects all the components on the path crossed by the ant.

**Daemon actions** Centralized actions performed on each solution built by the ants. These actions cannot be performed by single ants and are useful to improve the solution using global information. In practice, a local search can be applied to push the result towards a better solution.

**Algorithm 9** ACO system

```
def acoSystem(problem, α, β):
    initPheromones()
    while not terminationConditions():
        antBasedSolutionConstruction(α, β)
        pheromonesUpdate()
        daemonActions() # Optional
```

## 4.2 Artificial bee colony algorithm (ABC)

Artificial bee colony algorithm (ABC)

System where the position of nectar sources represents the solutions and the quantity of nectar sources represents the fitness of the solution. Artificial bees can be:

**Employed** Bee associated to a specific nectar source (intensification) (i.e. it represents a solution).

**Onlooker** Bee that is observing the employed bees and is choosing its nectar source.

**Scout** Bee that discovers new food sources (diversification).

The algorithm has the following phases:

**Initialization** The initial nectar source of each bee is determined randomly. Each solution (nectar source) is a vector $\mathbf{x}_m \in \mathbb{R}^n$ and each of its component is initialized constrained to a lower ($l_i$) and upper ($u_i$) bound:

$$\mathbf{x}_m\texttt{[}i\texttt{]} = l_i + \texttt{rand}(0,1) \cdot (u_i - l_i)$$

**Employed bees** Starting from their assigned nectar source, employed bees look in their neighborhood for a new food source with more fitness (more nectar). The fitness (for minimization problems) of a food source $\mathbf{x}_m$ is determined as:

$$\texttt{fit}(\mathbf{x}_m) = \begin{cases} \frac{1}{1+\texttt{obj}(\mathbf{x}_m)} & \text{if } \mathbf{x}_m \geq 0 \\ 1 + |\texttt{obj}(\mathbf{x}_m)| & \text{if } \mathbf{x}_m < 0 \end{cases}$$

where `obj` is the objective function.

**Onlooker bees** Onlooker bees stochastically choose their food source. Each food source $\mathbf{x}_m$ has a probability associated to it defined as:

$$p_m = \frac{\texttt{fit}(\mathbf{x}_m)}{\sum_{i=1}^{n_{\text{bees}}} \texttt{fit}(\mathbf{x}_i)}$$

This provides a positive feedback as more promising solutions have a higher probability to be chosen.

**Scout bees** Scout bees choose a nectar source randomly.

An employed bee that cannot improve its solution after a given number of attempts (gets fired and) becomes a scout (negative feedback).

**Algorithm 10** ABC

```
def abcAlgorithm(problem):
    initPhase()
    sol = None
    while not terminationConditions():
        employedBeesPhase()
        onlookerBeesPhase()
        scoutBeesPhase()
        sol = getCurrentBest()
```

## 4.3 Particle swarm optimization (PSO)

Particle swarm optimization (PSO)

In a bird flock, the movement of the individuals tend to:

- Follow the neighbors.
- Stay in the flock.
- Avoid collisions.

However, a model based on these rules does not have a common objective.
PSO introduces as common objective the search of food. Each individual that finds food can:

- Move away from the flock and reach the food.
- Stay in the flock.

Following the movement rules, the entire flock will gradually move towards promising areas.
Applied to optimization problems, the bird flock metaphor can be interpreted as:

**Bird** Agent that represents a possible solution that is progressively improved (exploration).

**Social interaction** Exploiting the knowledge of other agents to move towards a global solution (exploitation).

**Neighborhood** Individuals are affected by the actions of others close to them and are part of one or more sub-groups.

Note that sub-groups are not necessarily defined by physical proximity.

Given a cost function $f : \mathbb{R}^n \to \mathbb{R}$ to minimize (gradient is not known), PSO initializes a swarm of particles (agents) whose movement is guided by the best known position. Each particle is described by:

- Its position $\mathbf{x}_i \in \mathbb{R}^n$ in the search space.
- A velocity $\mathbf{v}_i \in \mathbb{R}^n$ that controls the movement of the particle.
- The best solution $\mathbf{p}_i$ it has found so far.

**Algorithm 11** PSO

```
def pso(f, n_particles, l, u, ω, φ_p, φ_g):
    particles = [Particle()] * n_particles
    gb = None # Global best
    for particle in particles:
        particle.value = randomUniform(l, u) # Search space bounds
        particle.vel = randomUniform(−|u − l|, |u − l|)
        particle.best = particle.value
        if f(particles.best) < f(gb): gb = particles.best

    while not terminationConditions():
        for particle in particles:
            r_p, r_g = randomUniform(0, 1), randomUniform(0, 1)
            x_i, p_i, v_i = particle.value, particle.best, particle.vel
            particle.vel = ω*v_i + φ_p*r_p*(p_i-x_i) + φ_g*r_g*(gb-x_i)
            particle.value = particle.value + particle.vel
            if f(particle.value) < f(particle.best):
                particle.best = particle.value
                if f(particle.best) < f(gb): gb = particle.best
```

# 5 Games

In this course, we are interested in two-player games with perfect knowledge (both players have the same information on the state of the game).

## 5.1 Minimax algorithm

Minimax algorithm

The minimax (min-max) algorithm allows to determine the optimal strategy for a player by building and propagating utility scores in a tree where each node represents a state of the game. It considers the player as the entity that maximizes (Max) its utility and the opponent as the entity that (optimally) minimizes (Min) the utility of the player.

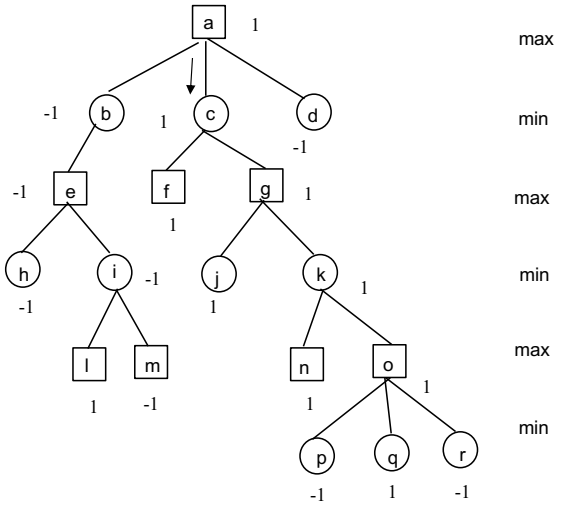



Figure 5.1: Example of game tree with propagated scores

In a game tree, each level represents the actions that a single player can do. In minimax, the levels where the player plays are the Max levels, while the levels of the opponent are the Min levels.
Given a node $n$ of a game tree, an iteration of the minimax algorithm can be described as follows:

**Expansion** Expansion of the sub-tree having $n$ as root by considering the possible moves starting from the state of $n$.

The expansion in height stops when a final state is reached or when some predetermined conditions are met (note that different branches may be expanded with different heights).

The expansion in width stops when all the possible moves have been considered or when some predetermined conditions are met.

**Evaluation** Each leaf is labeled with a score. For terminal states, a possible score

assignment is:

$$\texttt{utility(state)} = \begin{cases} +1 & \text{Win} \\ -1 & \text{Loss} \\ 0 & \text{Draw} \end{cases}$$

For non-terminal states, heuristics are required.

**Propagation** Starting from the parents of the leaves, the scores are propagated upwards by labeling the parents based on the children's score.

Given an unlabeled node $m$, if $m$ is at a MAX level, its label is the maximum of its children's score. Otherwise (MIN level), the label is the minimum of its children's score.

| | |
|---|---|
| **Completeness** | Yes, if the game tree is finite |
| **Optimality** | Yes, assuming an optimal opponent (otherwise, it may need more moves) |
| **Time complexity** | $O(b^m)$, with breadth $m$ and branching factor $b$ |
| **Space complexity** | $O(b \cdot m)$, with breadth $m$ and branching factor $b$ |

**Algorithm 12** Minimax

```
def minimax(node, max_depth, who_is_next):
    if node.isLeaf() or max_depth == 0:
        eval = evaluate(node)
    elif who_is_next == ME:
        eval = -∞
        for c in node.children:
            eval = max(eval, minimax(c, max_depth-1, OPPONENT))
    elif who_is_next == OPPONENT:
        eval = +∞
        for c in node.children:
            eval = min(eval, minimax(c, max_depth-1, ME))
    return eval
```

## 5.2 Alpha-beta cuts

Alpha-beta cuts

Alpha-beta cuts (pruning) allows to prune subtrees whose state will never be selected (when playing optimally). $\alpha$ represents the best choice found for MAX. $\beta$ represents the best choice found for MIN.
The best case for alpha-beta cuts is when the best nodes are evaluated first. In this scenario, the theoretical number of nodes to explore is decreased to $O(b^{d/2})$. In practice, the reduction is of order $O(\sqrt{b^d})$. In the average case of a random distribution, the reduction is of order $O(b^{3d/4})$.

**Algorithm 13** Minimax with alpha-beta cuts

```
def alphabeta(node, max_depth, who_is_next, α=-∞, β=+∞):
    if node.isLeaf() or max_depth == 0:
        eval = evaluate(node)
    elif who_is_next == ME:
        eval = -∞
        for c in node.children:
            eval = max(eval, alphabeta(c, max_depth-1, OPPONENT, α, β))
            α = max(eval, α)
            if eval >= β: break # cutoff
    elif who_is_next == OPPONENT:
        eval = +∞
        for c in node.children:
            eval = min(eval, alphabeta(c, max_depth-1, ME, α, β))
            β = min(eval, β)
            if eval <= α: break # cutoff
    return eval
```

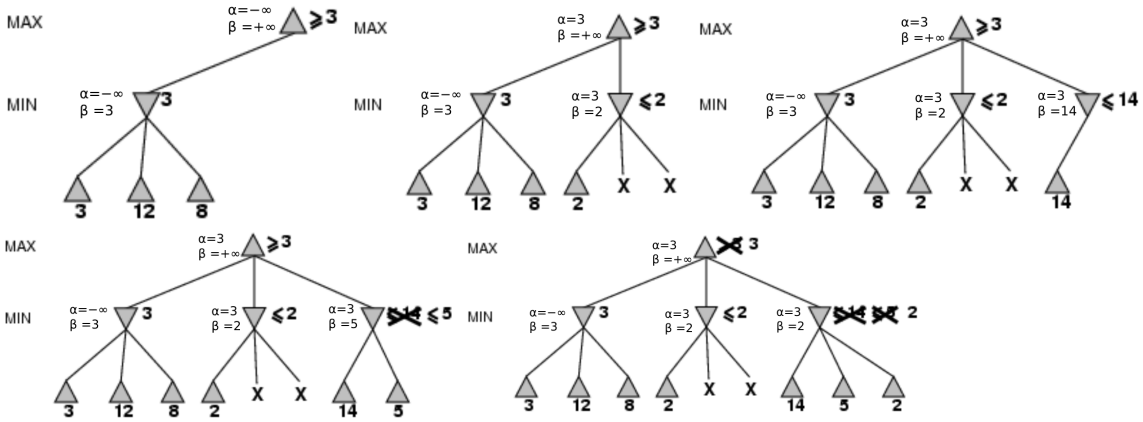



Figure 5.2: Algorithmic (left values) and intuitive (right value) application of alpha-beta

# 6 Automated planning

## 6.1 Definitions

**Automated planning** Given: Automated planning

- An initial state.
- A set of actions an agent can perform (operators).
- The goal to achieve.

Automated planning finds a partially or totally ordered set of actions that leads an agent from the initial state to the goal.

**Domain theory** Formal description of the executable actions. Each action has a name, pre-conditions and post-conditions. Domain theory

**Pre-conditions** Conditions that must hold for the action to be executable.

**Post-conditions** Effects of the action.

**Planner** Process to decide the actions that solve a planning problem. In this phase, actions are considered: Planner

**Non decomposable** An action is atomic (it starts and finishes). Actions interact with each other by reaching sub-goals.

**Reversible** Choices are backtrackable.

A planner can have the following properties:

**Correctness** The planner always finds a solution that leads from the initial state to the goal. Correct planner

**Completeness** The planner always finds a plan when it exists (planning is semi-decidable). Complete planner

**Execution** The execution is the implementation of a plan. In this phase, actions are: Execution

**Irreversible** An action that has been executed cannot (usually) be backtracked.

**Non deterministic** An action applied to the real world may have unexpected effects due to uncertainty.

**Generative planning** Offline planning that creates the entire plan before execution based on a snapshot of the current state of the world. It relies on the following assumptions: Generative planning

**Atomic time** Actions cannot be interrupted.

**Determinism** Actions are deterministic.

**Closed world** The initial state is fully known, what is not in the initial state is considered false (which is different from unknown).

**No interference** Only the execution of the plan changes the state of the world.

## 6.2 Linear planning

Linear planning

Formulates the planning problem as a search problem where:

- Nodes contain the state of the world.
- Edges represent possible actions.

Produces a totally ordered list of actions.
The direction of the search can be:

Forward search

**Forward** Starting from the initial state, the search terminates when a state containing a superset of the goal is reached.

Backward search

**Backward** Starting from the goal, the search terminates when a state containing a subset of the initial state is reached.

Goal regression is used to reduce the goal into sub-goals. Given a (sub-)goal $G$ and a rule (action) $R$ with delete-list (states that are false after the action) `d_list` and add-list (states that are true after the action) `a_list`, regression of $G$ through $R$ is defined as:

$$\texttt{regr}[G,\ R] = \texttt{true} \text{ if } G \in \texttt{a\_list} \text{ (i.e. regression possible)}$$
$$\texttt{regr}[G,\ R] = \texttt{false} \text{ if } G \in \texttt{d\_list} \text{ (i.e. regression not possible)}$$
$$\texttt{regr}[G,\ R] = G \text{ otherwise (i.e. } R \text{ does not influence } G)$$

**Example** (Moving blocks)**.** Given the action `UNSTACK(X, Y)` with:

$$\texttt{d\_list} = \{\texttt{handempty}, \texttt{on(X, Y)}, \texttt{clear(X)}\}$$
$$\texttt{a\_list} = \{\texttt{holding(X)}, \texttt{clear(Y)}\}$$

We have that:

$$\texttt{regr[holding(b), UNSTACK(b, Y)]} = \texttt{true}$$
$$\texttt{regr[handempty, UNSTACK(X, Y)]} = \texttt{false}$$
$$\texttt{regr[ontable(c), UNSTACK(X, Y)]} = \texttt{ontable(c)}$$
$$\texttt{regr[clear(c), UNSTACK(X, Y)]} = \begin{cases} \texttt{true} & \text{if Y=c} \\ \texttt{clear(c)} & \text{otherwise} \end{cases}$$

### 6.2.1 Deductive planning

Deductive planning

Formulates the planning problem using first order logic to represent states, goals and actions. Plans are generated as theorem proofs.

#### Green's formulation

Green's formulation

Green's formulation is based on **situation calculus**. To find a plan, the goal is negated and it is proven that it leads to an inconsistency.
The main concepts are:

**Situation** Properties (fluents) that hold in a given state `s`.

**Example** (Moving blocks)**.** To denote that `ontable(c)` holds in a state `s`, we use the axiom:

`ontable(c, s)`

The operator `do` allows to evolve the state such that:

$$\texttt{do(A, S)} = \texttt{S'}$$

`S'` is the new state obtained by applying the action `A` in the state `S`.

**Actions** Define the pre-condition and post-condition fluents of an action in the form:

$$\texttt{pre-conditions} \rightarrow \texttt{post-conditions}$$

Applying the equivalence $A \rightarrow B \equiv \neg A \vee B$, actions can be described by means of disjunctions.

**Example** (Moving blocks)**.** The action `STACK(X, Y)` has pre-conditions `holding(X)` and `clear(Y)`, and post-conditions `on(X, Y)`, `clear(X)` and `handfree`. Its representation in Green's formulation is:

$$\begin{aligned}\texttt{holding(X, S)} \wedge \texttt{clear(Y, S)} \rightarrow & \\ & \texttt{on(X, Y, do(STACK(X, Y), s))} \wedge \\ & \texttt{clear(X, do(STACK(X, Y), s))} \wedge \\ & \texttt{handfree(do(STACK(X, Y), s))}\end{aligned}$$

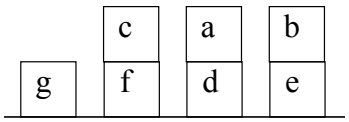
**Frame axioms** Besides the effects of actions, each state also have to define for all non-changing fluents their frame axioms. If the problem is complex, the number of frame axioms becomes unreasonable.

**Example** (Moving blocks)**.**

$$\texttt{on(U, V, S)} \wedge \texttt{diff(U, X)} \rightarrow \texttt{on(U, V, do(MOVE(X, Y, Z), S))}$$

**Example** (Moving blocks)**.** The initial state is described by the following axioms:

```
on(a, d, s0)        clear(c, s0)
on(b, e, s0)        clear(g, s0)
on(c, f, s0)         diff(a, b)
clear(a, s0)         diff(a, c)
clear(b, s0)        diff(a, d) ...
```




For simplicity, we only consider the action `MOVE(X, Y, Z)` that moves `X` from `Y` to `Z`. It is defined as:

$$\begin{aligned}\texttt{clear(X, S)}, \texttt{clear(Z, S)}, \texttt{on(X, Y, S)}, \texttt{diff(X, Z)} \rightarrow & \\ \texttt{clear(Y, do(MOVE(X, Y, Z), S))}, & \texttt{on(X, Z, do(MOVE(X, Y, Z), S))}\end{aligned}$$

This action can be translated into the following effect axioms:

$$\begin{aligned}\neg\texttt{clear(X, S)} \vee \neg\texttt{clear(Z, S)} \vee \neg\texttt{on(X, Y, S)} \vee \neg\texttt{diff(X, Z)} \vee & \\ \texttt{clear(Y, do(MOVE(X, Y, Z), S))} & \end{aligned}$$

$$\begin{aligned}\neg\texttt{clear(X, S)} \vee \neg\texttt{clear(Z, S)} \vee \neg\texttt{on(X, Y, S)} \vee \neg\texttt{diff(X, Z)} \vee & \\ \texttt{on(X, Z, do(MOVE(X, Y, Z), S))} & \end{aligned}$$

Given the goal `on(a, b, s1)`, we look for an action whose effects together with ¬`on(a, b, s1)` lead to an inconsistency. We decide to achieve this by using the action `MOVE(a, Y, b)`, therefore making the following substitutions:

$$\{\texttt{X}/\texttt{a}, \texttt{Z}/\texttt{b}, \texttt{s1}/\texttt{do(MOVE(a, Y, b), S)}\}$$

Using the disjunctive formulation (effect axioms), we need to show that the negated pre-conditions are false (therefore, making the action applicable):

| ¬`clear(a, S)` | ¬`clear(b, S)` | ¬`on(a, Y, S)` | ¬`diff(a, b)` |
|---|---|---|---|
| False with $\{\texttt{S}/\texttt{s0}\}$ | False with $\{\texttt{S}/\texttt{s0}\}$ | False with $\{\texttt{S}/\texttt{s0}, \texttt{Y}/\texttt{d}\}$ | False |

Therefore, the action `do(MOVE(a, d, b), s0)` defines the plan to reach the goal `on(a, b, s1)`.

### Kowalsky's formulation

Kowalsky's formulation avoids the frame axioms problem by using a set of fixed predicates: Kowalsky's formulation

`holds(rel, s/a)` Describes the relations `rel` that are true in a state `s` or after the execution of an action `a`.

`poss(s)` Indicates if a state `s` is possible.

`pact(a, s)` Indicates if an action `a` can be executed in a state `s`.

Actions can be described as:

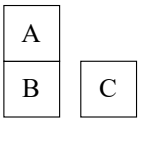
$$\texttt{poss(S)} \land \texttt{pact(A, S)} \rightarrow \texttt{poss(do(A, S))}$$

In the Kowalsky's formulation, each action requires a frame assertion (in Green's formulation, each state requires frame axioms).

**Example** (Moving blocks)**.** An initial state can be described by the following axioms:

```
holds(on(a, b), s0)
holds(ontable(b), s0)
holds(ontable(c), s0)
```

```
holds(clear(a), s0)
holds(clear(c), s0)
holds(handempty, s0)
poss(s0)
```




**Example** (Moving blocks)**.** The action `UNSTACK(X, Y)` has:

**Pre-conditions** `on(X, Y)`, `clear(X)` and `handempty`

**Effects**

**Add-list** `holding(X)` and `clear(Y)`

**Delete-list** `on(X, Y)`, `clear(X)` and `handempty`

Its description in Kowalsky's formulation is:

**Pre-conditions**

$$\texttt{holds(on(X, Y), S)}, \texttt{holds(clear(X), S)}, \texttt{holds(handempty, S)} \rightarrow \texttt{pact(UNSTACK(X, Y), S)}$$

**Effects** (use add-list)

$$\texttt{holds(holding(X), do(UNSTACK(X, Y), S))}$$

$$\texttt{holds(clear(Y), do(UNSTACK(X, Y), S))}$$

**Frame condition** (uses delete-list)

$$\texttt{holds(V, S)}, \texttt{V} \neq \texttt{on(X, Y)}, \texttt{V} \neq \texttt{clear(X)}, \texttt{V} \neq \texttt{handempty} \rightarrow \texttt{holds(V, do(UNSTACK(X, Y), S))}$$

### 6.2.2 STRIPS

STRIPS (Stanford Research Institute Problem Solver) is an ad-hoc algorithm for linear planning resolution. The elements of the problem are represented as: STRIPS

**State** represented with its true fluents.

**Goal** represented with its true fluents.

**Action** represented using three lists:

**Preconditions** Fluents that are required to be true in order to apply the action.

**Delete-list** Fluents that become false after the action.

**Add-list** Fluents that become true after the action.

Add-list and delete-list can be combined in an effect list with positive (add-list) and negative (delete-list) axioms.

**STRIPS assumption** Everything that is not in the add-list or delete-list is unchanged in the next state.

STRIPS uses two data structures:

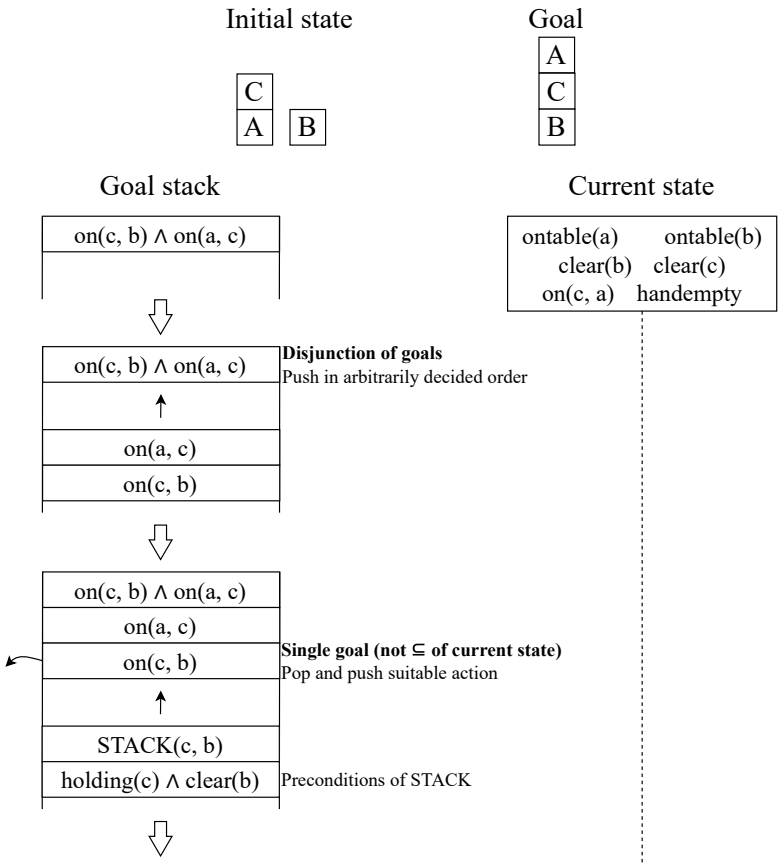
**Goal stack** Does a backward search to reach the initial state.

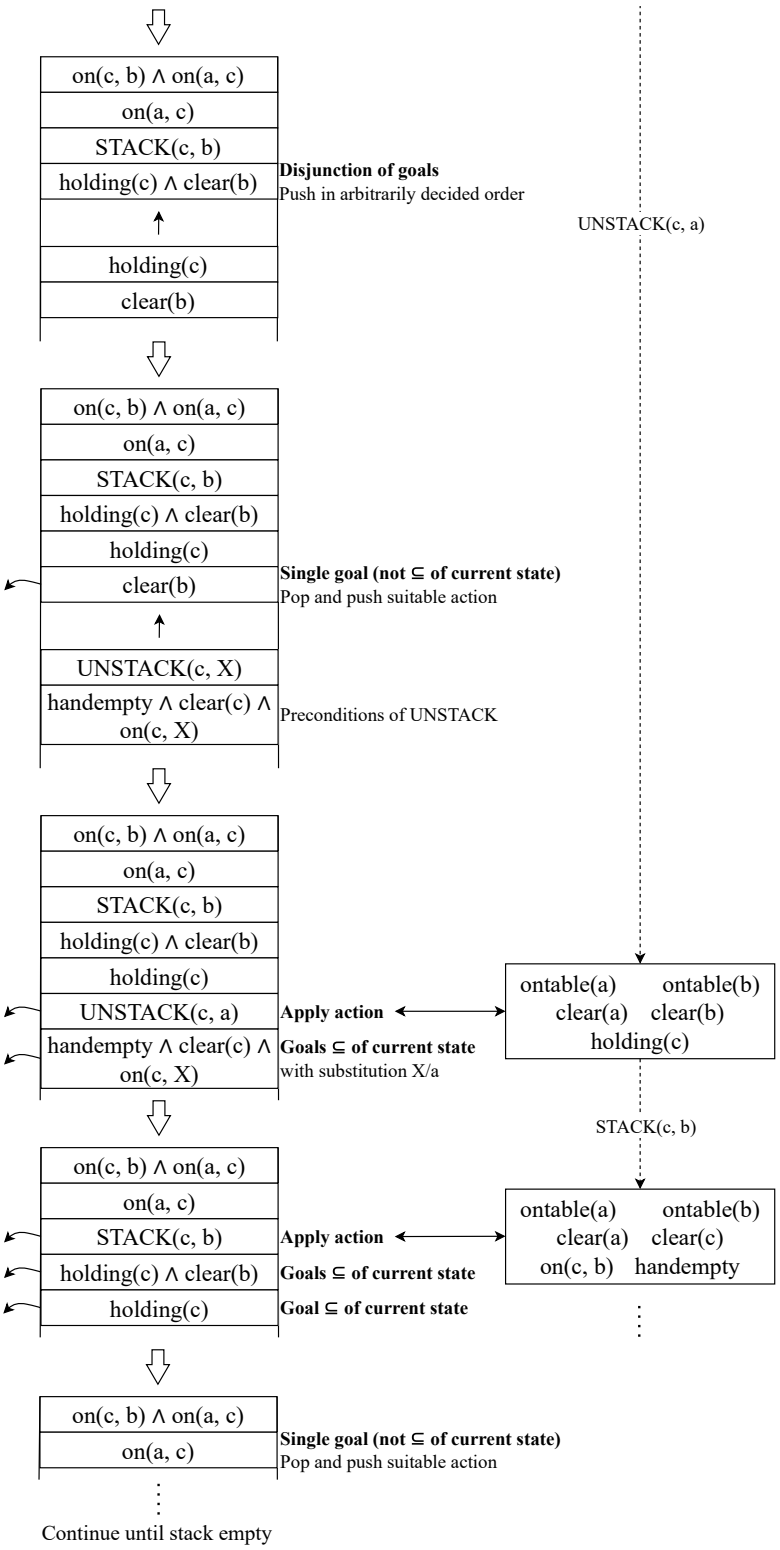
**Current state** Represents the forward application of the actions found using the goal stack.

**Algorithm 14** STRIPS

```
def strips(problem):
    goal_stack = Stack()
    current_state = State(problem.initial_state)
    goal_stack.push(problem.goal)
    plan = []
    while not goal_stack.empty():
        if (goal_stack.top() is a single/conjunction of goals and
            there is a substitution θ that makes it ⊆ current_state):
            A = goal_stack.pop()
            θ = find_substitution(A, current_state)
            goal_stack.apply_substitution(θ)
        elif goal_stack.top() is a single goal:
            R = rule with a ∈ R.add_list
            _ = goal_stack.pop() # Pop goal
            goal_stack.push(R)
            goal_stack.push(R.preconditions)
        elif goal_stack.top() is a conjunction of goals:
            for g in permutation(goal_stack.top()):
                goal_stack.push(g)
            # Note that there is no pop
        elif goal_stack.top() is an action:
            action = goal_stack.pop()
            current_state.apply(action)
            plan.append(action)
    return plan
```

**Example** (Moving blocks).






Since there are non-deterministic choices, the search space might become very large. Heuristics can be used to avoid this.

Sussman anomaly

Conjunction of goals are solved separately, but this can lead to the **Sussman anomaly** where a sub-goal destroys what another sub-goal has done. For this reason, when a conjunction is encountered, it is not immediately popped from the goal stack and is left as a final check.

## 6.3 Non-linear planning

### 6.3.1 Partial order planning

Non-linear planning

Non-linear planning finds a plan as a search problem in the space of plans (instead of states as in linear planning). Each node of the search tree is a partial plan. Edges represent plan refinement operations.
A non-linear plan is represented by:

**Actions**. Actions set

**Orderings** between actions. Orderings set

**Causal links** triplet $\langle S_i, S_j, c\rangle$ where $S_i$ and $S_j$ are actions and $c$ is a sub-goal. $c$ should be in the effects of $S_i$ and in the preconditions of $S_j$. Causal links

Causal links represent causal relations between actions (i.e. interaction between sub-goals): to execute $S_j$, the effect $c$ of $S_i$ is required first.

The initial plan is an empty plan with two fake actions `start` and `stop` with ordering `start` $<$ `stop`:

`start` has no preconditions and the effects match the initial state.

`stop` has no effects and the preconditions match the goal.

At each step, one of the following refinement operations can be applied until the goal is reached:

- Add an action to the set of actions.
- Add an ordering to the set of orderings.
- Add a causal link to the set of causal links.

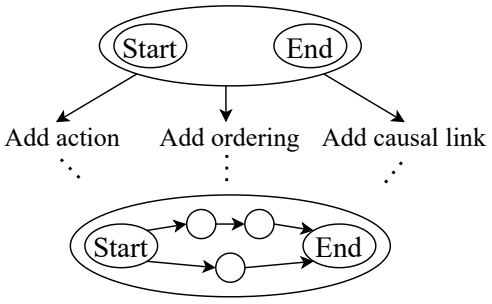



Figure 6.1: Example of search tree in non-linear planning

**Least commitment planning** Only strictly necessary restrictions (e.g. ordering) are imposed. Non-linear planning is a least commitment planning. Least commitment planning

**Linearization** At the end, the partially ordered actions should be linearized, respecting the ordering constraints, to obtain the final plan. Linearization

**Threat** An action $S_k$ is a threat to a causal link $\langle S_i, S_j, c\rangle$ if its effects cancel $c$. $S_k$ should not be executed in between $S_i$ and $S_j$. Threat

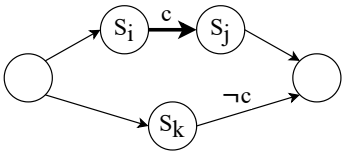



Figure 6.2: Example of threat. Causal links are represented using thick arrows.

Possible solutions to a threat $S_k$ to $\langle S_i, S_j, c\rangle$ are:

**Demotion** Add the ordering constraint $S_k < S_i$ (i.e. threat executed before). Demotion

**Promotion** Add the ordering constraint $S_k > S_j$ (i.e. threat executed after). Promotion

**Algorithm 15** Partial order planning (POP)

```
def pop(initial_state, goal, actions):
    plan = init_empty_plan(initial_state, goal)
    while not plan.isSolution():
        try:
            sn, c = selectSubgoal(plan)
            chooseOperator(plan, actions, sn, c)
            resolveThreats(plan)
        except PlanFailError:
            plan.backtrack()
    return plan

def selectSubgoal(plan):
    sn, c = random([sn, c in plan.steps if c in sn.unsolved_preconditions])
    return sn, c

def chooseOperator(plan, actions, sn, c):
    s = random([s in (actions + plan.steps) if c in s.effects])
    if s is None: raise(PlanFailError)
    plan.addCausalLink(⟨s, sn, c⟩)
    plan.addOrdering(s < sn)
    if s not in plan.steps:
        plan.addAction(s)
        plan.addOrdering(start < s < stop)

def resolveThreats(plan):
    for s_k, s_i, s_j in plan.threats():
        resolution = random([ DEMOTION, PROMOTION ])
        if resolution == DEMOTION:
            plan.addOrdering(s_k < s_i)
        elif resolution == PROMOTION:
            plan.addOrdering(s_k > s_j)
        if plan.isNotConsistent(): raise(PlanFailError)
```

**Example** (Purchasing schedule)**.** The initial state is:

`at(home), sells(hws, drill), sells(sm, milk), sells(sm, banana)`

where `hws` means "hardware store" and `sm` means "supermarket".

The goal is:

$$\texttt{at(home)}, \texttt{have(drill)}, \texttt{have(milk)}, \texttt{have(banana)}$$

The possible actions are:

`GO(X, Y)`

**Preconditions** `at(X)`

**Effects** `at(Y)`, $\neg$`at(X)`

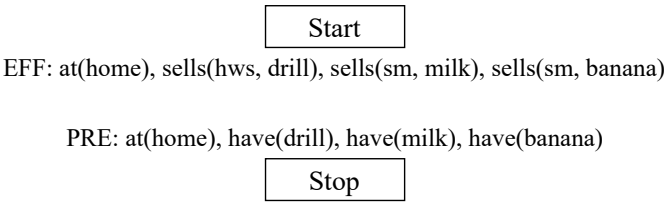
`BUY(S, Y)`

**Preconditions** `at(S)`, `sells(S, Y)`

**Effects** `have(Y)`

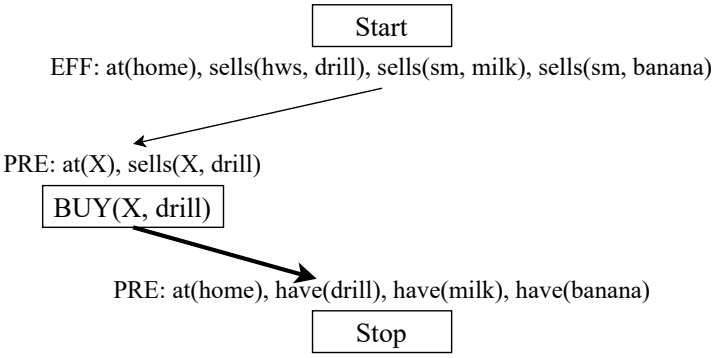
Partial order planning steps are:

1. Define the initial plan:

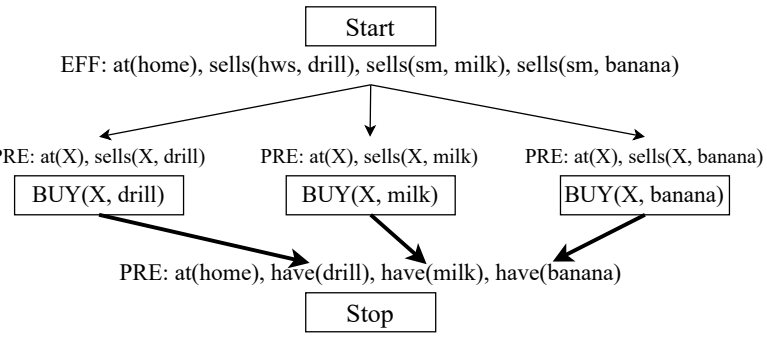



2. The loop of POP is:
   - Choose an action $a_i$ and one of its unsolved preconditions $c$.
   - Select an action $a_j$ with the precondition $c$ in its effects.
   - Add the ordering constraint `start` $< a_j <$ `stop`.
   - Add the causal link $\langle a_j, a_i, c\rangle$ (and ordering $a_j < a_i$).
   - Solve threats.

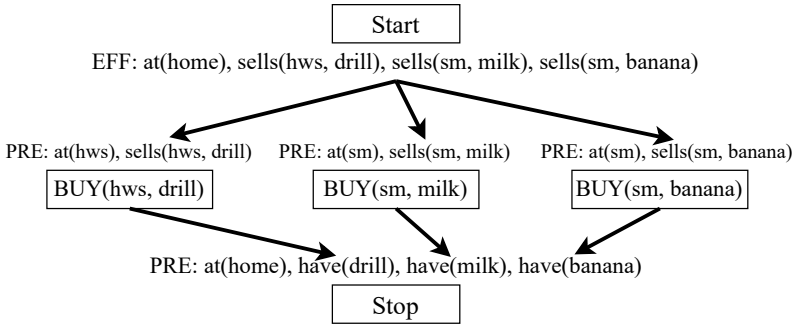
   We choose the action $a_i =$ `stop` and the precondition $c =$ `have(drill)`. We choose as action with $c$ in its effects $a_j =$ `BUY(X, drill)`. We therefore add to the plan the ordering `start` $<$ `BUY(X, drill)` $<$ `stop` and the causal link $\langle$`BUY(X, drill)`, `stop`, `have(drill)`$\rangle$:




3. Repeat the previous point for the preconditions `have(milk)` and `have(banana)`:



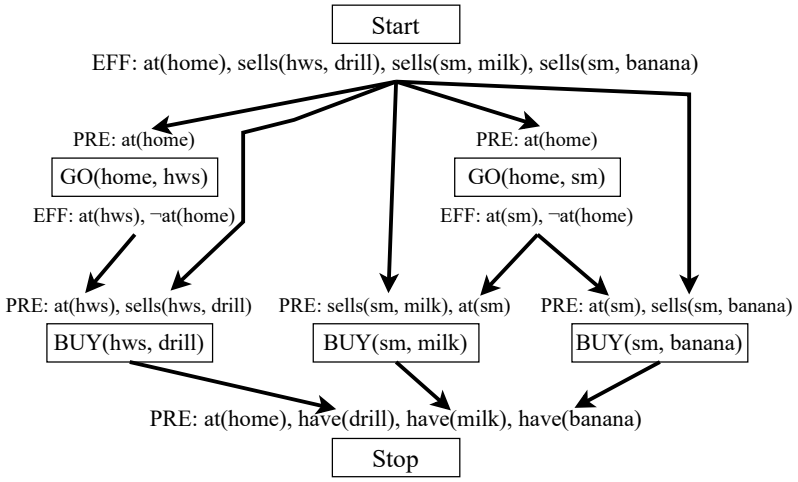
4. Now, we choose as action `BUY(X, drill)` and as unsolved precondition `sells(X, drill)`. This can be solved from the action `start` with effect `sells(hws, drill)`. We make the substitution `X/drill` and add $\langle$`start`, `BUY(hws, drill)`, `sells(hws, drill)`$\rangle$ to the causal links. The same process can be repeated for `BUY(X, milk)` and `BUY(X, banana)`:




5. Now, we choose as action `BUY(hws, drill)` and as unsolved precondition `at(hws)`. This can be solved using the action `GO(X, hws)`. We add $\langle$`GO(X, hws)`, `BUY(hws, drill)`, `at(hws)`$\rangle$ to the causal links.

   We continue by choosing as action `GO(X, hws)` and as unsolved precondition `at(X)`. This can be solved from `start` with effect `at(home)`. We therefore make the substitution `X/home` and add $\langle$`start`, `GO(home, hws)`, `at(home)`$\rangle$ to the causal links.

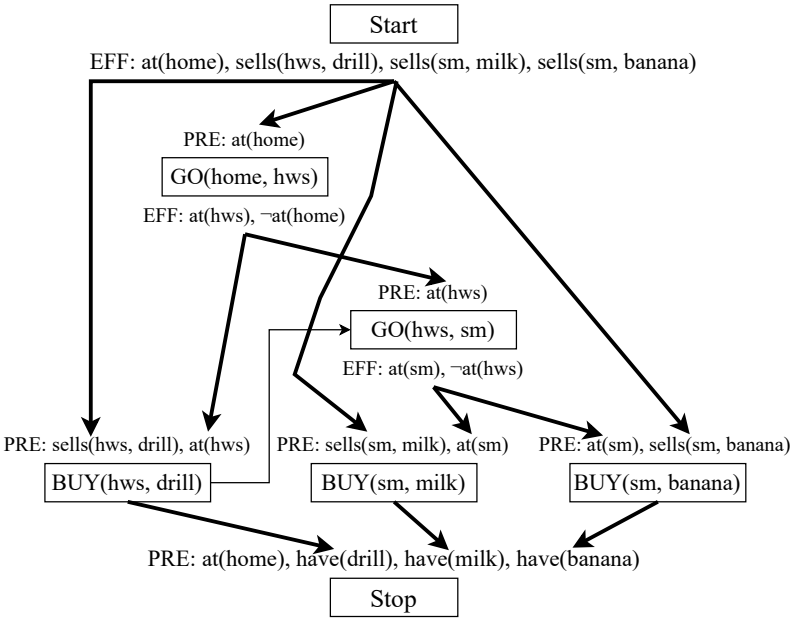
   The same process can be repeated for the `milk` and `banana` branch:




6. We have a threat between `GO(home, hws)` and `GO(home, sm)` as they both require the precondition `at(home)` and both have as effect ¬`at(home)`. It can be easily seen that neither promotion nor demotion solves the conflict. We are therefore forced to backtrack.

   We backtrack at the previous point, where we chose as action `GO(X, sm)` and as precondition `at(X)` (this step has been implicitly done in the previous point).

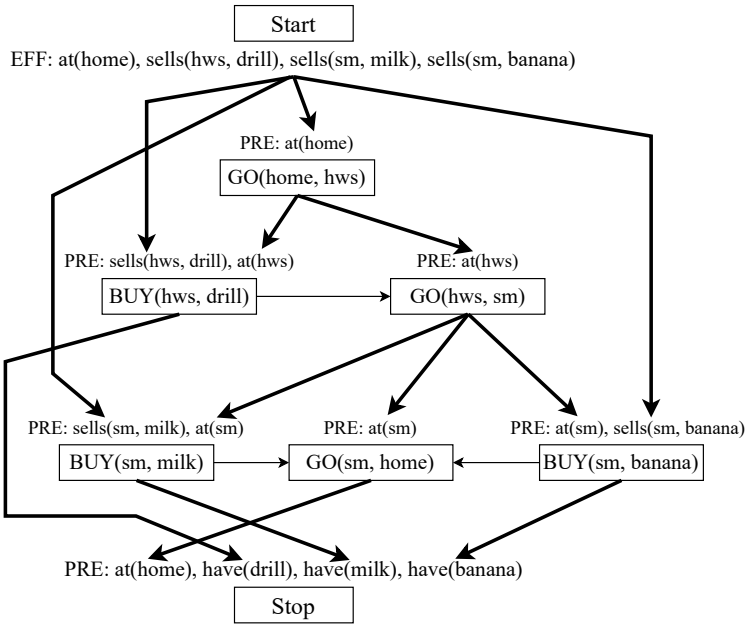
   - Instead of choosing the action `start`, we choose `GO(home, hws)` with the effect `at(hws)`. We therefore make the substitution `X/hws` and update the causal links.
   - We also resolve the threat `GO(hws, sm)` to `BUY(hws, drill)` (it removes the precondition `at(hws)`) by promoting `GO(hws, sm)` and adding the ordering constraint `BUY(hws, drill)` $<$ `GO(hws, sm)`:




7. Now, we choose as action `stop` and as precondition `at(home)`. We choose as action `GO(sm, home)` and update the causal links.

   Finally, we solve the threat `GO(sm, home)` to both `BUY(sm, milk)` and `BUY(sm, banana)` (it removes the required precondition `at(sm)`) by promoting `GO(sm, home)`. The newly added ordering constraints are `BUY(sm, milk)` $<$ `GO(sm, home)` and `BUY(sm, banana)` $<$ `GO(sm, home)`.

   The final plan is:




   By considering the ordering constraints, a linearization could be:

$$\texttt{GO(home, hws)} \rightarrow \texttt{BUY(hws, drill)} \rightarrow \texttt{GO(hws, sm)} \rightarrow \texttt{BUY(sm, milk)} \rightarrow \texttt{BUY(sm, banana)} \rightarrow \texttt{GO(sm, home)}$$

### 6.3.2 Modal truth criterion

Modal truth criterion

Modal truth criterion uses five plan refinement methods that ensures the completeness of the planner (POP is not complete).
MTC refinement methods are:

Establishment

**Establishment** The same standard operations of POP:

1. Insert a new action in the plan.
2. Add an ordering constraint.
3. Do a variable assignment.

Promotion

**Promotion** As in POP.

Demotion

**Demotion** As in POP.

White knight

**White knight** Insert an operator $S_n$ between $S_k$ and $S_j$, where $S_k$ threatens $S_j$, in such way that $S_n$ re-establishes the preconditions of $S_j$.

Separation

**Separation** Add a constraint to a variable to avoid that it unifies with an unwanted value.

## 6.4 Hierarchical planning

Hierarchical planning

Hierarchical planning allows to create a complex plan at different levels of abstraction. Different meta-level searches are executed to generate meta-level plans that are progressively refined.

### 6.4.1 ABSTRIPS

ABSTRIPS

In ABSTRIPS, a criticality value is assigned to each precondition based on the complexity of its achievement.
At each level, a plan is found assuming that the preconditions corresponding to lower levels of criticality are true (i.e. solve harder goals first). At the next level, the previously found plan and its preconditions are used as starting point in the goal stack.

**Algorithm 16** ABSTRIPS

```
def abstrips(problem, start_threshold, threshold_step):
    threshold = start_threshold
    plan = None
    while there is a precondition still not considered:
        true_preconds = problem.preconds[criticality < threshold]
        plan = strips(problem, true_preconds, starting_plan=plan)
        threshold -= threshold_step
    return plan
```

### 6.4.2 Macro-operators

Macro-operators

In macro-operators, two types of operators are defined:

**Atomic** Elementary operations that can be executed by an agent.

**Macro** Set of atomic operators. Before execution, this type of operator has to be decomposed.

**Precompiled decomposition** The decomposition is known and described along side the preconditions and effects of the operator.

**Planned decomposition** The planner has to synthesize the atomic operators that compose a macro operator.

Constraints are needed for a safe decomposition. Let $A$ be a macro with effect $X$ and $P$ its decomposition:

- $X$ must be the effect of at least an atomic action in $P$ and should be protected until the end of $P$.
- Each precondition of the actions in $P$ must be guaranteed by previous actions or be a precondition of $A$.
- $P$ must not threat any causal link.

Moreover, when a macro action $A$ is replaced with its decomposition $P$:

- For each $B$ such that $B < A$, impose the ordering $B <$ `First(`$P$`)`.
- For each $B$ such that $A < B$, impose the ordering `Last(`$P$`)` $< B$.
- Each causal link $\langle S, A, c\rangle$ is replaced with $\langle S, S_i, c\rangle$, where $S_i$ are actions in $P$ with precondition $c$ and do not have other atomic operators of the macro before.
- Each causal link $\langle A, S, c\rangle$ is replaced with $\langle S_i, A, c\rangle$, where $S_i$ are actions in $P$ with effect $c$ and do not have other atomic operators of the macro after.

**Algorithm 17** Hierarchical decomposition POP

```
def hdpop(initial_state, goal, actions, decomposition_methods):
    plan = init_empty_plan(initial_state, goal)
    while not plan.isSolution():
        try:
            if choice() == ESTABLISHMENT:
                sn, c = selectSubgoal(plan)
                chooseOperator(plan, actions, sn, c)
            else:
                macro = selectMacroStep(plan)
                chooseDecomposition(macro, decomposition_methods, plan)
            resolveThreats(plan)
        except PlanFailError:
            plan.backtrack()
    return plan
```

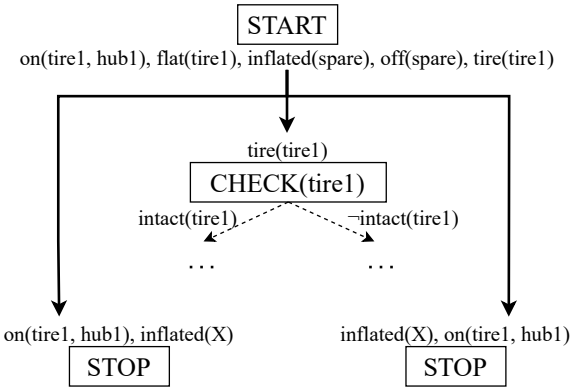
## 6.5 Conditional planning

Conditional planning

Conditional planning is based on the open world assumption where what is not in the initial state is unknown. It generates a different plan for each source of uncertainty and therefore has exponential complexity.

**Sensing action** Action with pre-conditions and post-conditions that allows to obtain unknown information.

**Example** (Inflate tire)**.**




When executing a sensing action (`CHECK(tire1)`), a copy of the goal is generated for each possible scenario.

## 6.6 Reactive planning

Reactive planners are on-line algorithms able to interact with the dynamicity the world.

### 6.6.1 Pure reactive systems

Pure reactive systems

Pure reactive planners have a knowledge base that describes the conditions for which an action has to be executed. The choice of the action is predictable. Therefore, this approach is not suited for domains that require reasoning.

### 6.6.2 Hybrid systems

Hybrid systems

Hybrid planners integrate the generative and reactive approach. The steps the algorithm does are:

- Generates a plan to achieve the goal.
- Checks the pre-conditions and post-conditions of the action it is going to execute.
- Backtracks the effects of an action in case of a failure.
- Corrects the plan if external events occur.

## 6.7 Graphplan

Graphplan

Graphplan is an off-line, least-commitment, closed-world planner that constructs a partially ordered set of actions through a planning graph based on time steps.
The planner is correct, complete, optimal and computationally efficient.

Actions

**Action** An action, as in STRIPS, has:

- Preconditions.
- Add list.
- Delete list.

`NO-OP` Action that does not change the state (to solve the frame problem). Can be seen as an action with the same proposition as precondition and add list.

**State** A state is represented by a set of propositions that are true in that time step. State

**Planning graph** Directed leveled graph where edges connect nodes of adjacent levels. Planning graph

There are two possible levels that are alternated during construction:

**Proposition level** Contains propositions that describe the state. Note that interfering propositions can appear.

**Action level** Contains all the possible actions that have as preconditions the propositions in the previous level. Note that interfering actions can appear.

The first level of the graph is a proposition level containing the initial state.

Edges can be:

**Precondition arcs** proposition → action.

**Add arcs** action → proposition.

**Delete arcs** action → proposition.

**Inconsistency** Actions and propositions in the same time step can be inconsistent. Possible causes are: Inconsistency

**Actions**

**Inconsistent effects** An action negates the effects of another one. Inconsistent effects

**Interference** An action deletes the preconditions of another one. Interference

**Competing needs** Two actions have mutually exclusive preconditions. Competing needs

**Domain dependent**

**Propositions** Propositions are inconsistent when they cannot appear together either because one negates the other or because they can be reached only through mutually exclusive paths. Inconsistent propositions

**Plan extraction** Once a proposition level containing the goal as non-mutually exclusive propositions has been reached, the algorithm can attempt to extract a plan. A valid plan has the following properties: Plan extraction

- Actions in the same time step do not interfere and can be executed in any order.
- Propositions at the same time step are non-mutually exclusive.
- The last step contains the goal as non-mutually exclusive propositions.

Even if the last step is a superset of the goal, planning may still fail. In this case, the algorithm has to continue generating levels.

**Memoization** At each step, if the goal is not satisfiable, the result is saved and when the same state is encountered in the future it will automatically fail. Memoization

**Theorem 6.7.1.** The following statements hold:

- If a valid plan exists, it can be found as a subgraph of the planning graph.
- In a planning graph, two actions in a time step are mutually exclusive if a valid plan containing both does not exist.

- In a planning graph, two propositions are mutually exclusive if they are inconsistent.

**Corollary 6.7.1.1.** Inconsistencies found during the planning graph construction prune paths in the search tree.

**Algorithm 18** Graphplan

```
def graphplan(initial_state, actions, goal):
    graph = PlanningGraph()
    graph.addPropositionLevel(initial_state)
    while True:
        if (goal in graph.lastPropositionLevel and
            not mutexPropositions(goal, graph.lastPropositionLevel)):
            plan = extractSolution(graph, goal)
            if plan is not FAIL: return plan
        graph.addActionLevel( selectActions(graph) )
        graph.addPropositionLevel( selectPreconditions(graph, actions) )

def selectActions(graph, actions):
    new_action_level = Level()
    for action in actions.unify(graph.lastPropositionLevel):
        preconds = action.preconditions
        if not mutexPropositions(preconds, graph.lastPropositionLevel):
            new_action_level.add(preconds, action)
    for proposition in graph.lastPropositionLevel:
        new_action_level.add(NO_OP(proposition))
    new_action_level.findInconsistencies()
    return new_action_level

def selectPreconditions(graph):
    new_props_level = Level()
    for action in graph.lastActionLevel:
        for prop in action.add_list:
            new_props_level.add(action, prop, "add")
        for prop in action.delete_list:
            new_props_level.add(action, prop, "delete")
    new_props_level.findInconsistencies()
    return new_props_level

def extractSolution(graph, goal):
    plan = Plan()
    actions = graph.lastActionLevel.getActionsWithEffect(goal)
    if mutexActions(actions, graph.lastActionLevel): return FAIL
    plan.addLevel(actions)
    graph = graph.popLastActionLevel()
    return plan.merge(extractSolution(graph, actions.preconditions))
```

**Example** (Moving objects with a cart)**.** Given the actions:

`MOVE(R, PosA, PosB)`

**Preconditions** `at(R, PosA)`, `hasFuel(R)`

**Add list** `at(R, PosB)`

**Delete list** `at(R, PosA)`, `hasFuel(R)`

`LOAD(Obj, Pos)`

**Preconds** `at(R, Pos)`, `at(Obj, Pos)`

**Add list** `in(R, Obj)`

**Delete list** `at(Obj, Pos)`

`UNLOAD(Obj, Pos)`

**Preconds** `in(R, Obj)`, `at(R, Pos)`

**Add list** `at(Obj, Pos)`

**Delete list** `in(R, Obj)`

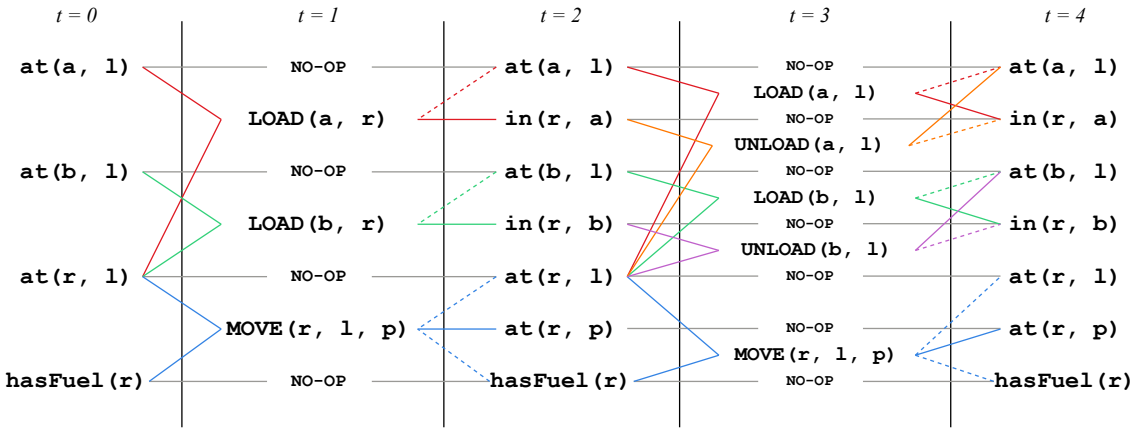
Given a scenario where:

- `r` is a cart.
- `a` and `b` are objects.
- `l` and `p` are locations.

and the initial state:

$$\texttt{at(a, l)} \cdot \texttt{at(b, l)} \cdot \texttt{at(r, l)} \cdot \texttt{hasFuel(r)}$$

The first four time steps of the planning graph are:




The inconsistencies at $t = 1$ are:

**Inconsistent effects**

$$\begin{cases}\texttt{NO-OP}\\ \texttt{LOAD(a, r)}\end{cases} \text{ for } \texttt{at(a, l)}, \quad \begin{cases}\texttt{NO-OP}\\ \texttt{LOAD(b, r)}\end{cases} \text{ for } \texttt{at(b, l)},$$

$$\begin{cases}\texttt{NO-OP}\\ \texttt{MOVE(r, l, p)}\end{cases} \text{ for } \texttt{at(r, l)}, \quad \begin{cases}\texttt{NO-OP}\\ \texttt{MOVE(r, l, p)}\end{cases} \text{ for } \texttt{hasFuel(r)}$$

**Interference**

$$\begin{cases}\texttt{MOVE(r, l, p)}\\ \texttt{LOAD(a, r)}\end{cases} \text{ for } \texttt{at(r, l)}, \quad \begin{cases}\texttt{MOVE(r, l, p)}\\ \texttt{LOAD(b, r)}\end{cases} \text{ for } \texttt{at(r, l)}$$

The inconsistencies of propositions at $t = 2$ are:

- Consequence of the add and delete list of each action:

$$\begin{cases}\texttt{at(a, l)}\\ \texttt{in(r, a)}\end{cases}, \begin{cases}\texttt{at(b, l)}\\ \texttt{in(r, b)}\end{cases}, \begin{cases}\texttt{at(r, l)}\\ \texttt{at(r, p)}\end{cases}, \begin{cases}\texttt{at(r, p)}\\ \texttt{hasFuel(r)}\end{cases}$$

- Consequence of the add list of interfering actions (mutual exclusion):
  $\begin{cases} \texttt{in(r, a)} \\ \texttt{at(r, p)} \end{cases}, \begin{cases} \texttt{in(r, b)} \\ \texttt{at(r, p)} \end{cases}$

Note that because of the mutually exclusive propositions at $t = 2$, at $t = 3$ the actions `UNLOAD(·, p)` cannot be performed.

### 6.7.1 Fast forward

Heuristic planner based on Graphplan, hill climbing and A*. Fast forward

**Heuristic** Given a problem $P$, the algorithm considers a relaxation $P^+$ where delete effects are ignored. $P^+$ is solved using Graphplan and the number of actions required to solve it is used as lower bound heuristic for $P$.

**Algorithm**

1. From a state $S$, examine the successors.
2. If there is a successor $S'$ better than $S$, move into it and return to point 1.
3. Otherwise, run a complete A* search.

# 7 Constraints satisfaction

**Constraints satisfaction problem (CSP)** Problem defined on a finite set of variables $(X_1, \ldots, X_n)$, each belonging to a domain $(D_1, \ldots, D_n)$ (notation: $X_i :: [d_{i,1}, \ldots, d_{i,m}]$ or $X_i \in [d_{i,1}, \ldots, d_{i,m}]$). Constraints satisfaction problem (CSP)

A constraint is a relationship between variables (e.g. $X < Y$). Formally, a constraint $c(X_{k_1}, \ldots, X_{k_m})$ is a subset of the cartesian product $D_{k_1} \times \cdots \times D_{k_m}$ of the admissible values of the variables.

A solution to a CSP is an admissible assignment of all the variables.

CSP can be solved as a search problem.

**A posteriori algorithms** Constraints are checked after the construction of the search tree. A posteriori algorithms

**Propagation algorithms** Once a variable has been assigned, constraints are propagated to the unassigned variables to prune part of the search space. Propagation algorithms

**Consistency techniques** Constraints are propagated to derive a simpler problem. Consistency techniques

## 7.1 A posteriori algorithms

### 7.1.1 Generate and test

The search tree is constructed depth-first. At level $i$, a value is assigned to the variable $X_i$. Constraints are only checked when a leaf has been reached. Generate and test
In other words, it computes all the permutations of the values of the variables.

### 7.1.2 Standard backtracking

The search tree is constructed depth-first. At level $i$, a value is assigned to the variable $X_i$ and constraints involving $X_1, \ldots, X_i$ are checked. In case of failure, the path is not further explored. Standard backtracking
A problem of this approach is that it requires to backtrack in case of failure and reassign all the variables in the worst case.

## 7.2 Propagation algorithms

### 7.2.1 Forward checking

The search tree is constructed depth-first. At level $i$, a value is assigned to the variable $X_i$ and constraints involving $X_i$ are propagated to restrict the domain of the remaining unassigned variables. If the domain of a variable becomes empty, the path is considered a failure and the algorithm backtracks. Forward checking

**Example.** Consider the variables:

$$X_1 :: [1, 2, 3] \qquad X_2 :: [1, 2, 3] \qquad X_3 :: [1, 2, 3]$$

and the constraints $X_1 < X_2 < X_3$.
We assign the variables in lexicographic order, at each step we have that:

1. $X_1 = 1 \qquad X_2 :: [\not{1}, 2, 3] \qquad X_3 :: [\not{1}, 2, 3]$
2. $X_1 = 1 \qquad X_2 = 2 \qquad X_3 :: [\not{1}, \not{2}, 3]$
3. $X_1 = 1 \qquad X_2 = 2 \qquad X_3 = 3$

In another scenario, assume that we choose $X_1 = 2$ as first step:

1. $X_1 = 2 \qquad X_2 :: [\not{1}, \not{2}, 3] \qquad X_3 :: [\not{1}, \not{2}, 3]$
2. $X_1 = 2 \qquad X_2 = 3 \qquad X_3 :: [\not{1}, \not{2}, \not{3}]$

As the domain of $X_3$ is empty, search on this branch fails and backtracking is required.

### 7.2.2 Look ahead

Partial look ahead (PLA)

**Partial look ahead (PLA)** When a variable $X_i$ has been assigned, it is propagated to unassigned variables (as in forward checking). Then, for each unassigned variable $X_h$, we check its values against the ones of the variables $X_{h+1}, \dots, X_n$ in front of it (note that the check is not done in both directions).

When checking the values of an unassigned variable $X_p :: D_p$ against $X_q :: D_q$, we maintain a value $x \in D_p$ only if there is at least a corresponding value $y \in D_q$ such that an assignment $X_p = x$ and $X_q = y$ does not violate the constraints. $y$ is called support of $x$.

Support

**Example.** Consider the variables and constraints:

$$X_1 :: [1, 2, 3] \quad X_2 :: [1, 2, 3] \quad X_3 :: [1, 2, 3] \qquad X_1 < X_2 < X_3$$

We assign the variables in lexicographic order. At each step we have that:

1. $X_1 = 1 \qquad X_2 :: [\not{1}, 2, \not{3}] \qquad X_3 :: [\not{1}, 2, 3]$
   Here, we assign $X_1 = 1$ and propagate to unassigned constraints. Then, we check the constraints of $(X_2, X_3)$: $X_2 = 3$ is not possible as it does not have a support in $X_3$.
2. $X_1 = 1 \qquad X_2 = 2 \qquad X_3 :: [\not{1}, \not{2}, 3]$
3. $X_1 = 1 \qquad X_2 = 2 \qquad X_3 = 3$

Full look ahead (FLA)

**Full look ahead (FLA)** Same as partial look ahead, but the checks on unassigned variables are bidirectional.

**Example.** Consider the variables and constraints:

$$X_1 :: [1, 2, 3] \quad X_2 :: [1, 2, 3] \quad X_3 :: [1, 2, 3] \qquad X_1 < X_2 < X_3$$

We assign the variables in lexicographic order. At each step we have that:

1. $X_1 = 1 \qquad X_2 :: [\not{1}, 2, \not{3}] \qquad X_3 :: [\not{1}, \not{2}, 3]$
   Here, we assign $X_1 = 1$ and propagate to unassigned constraints. Then, we check the constraints of:
   - $(X_2, X_3)$: $X_2 = 3$ is not possible as it does not have a support in $X_3$.

- $(X_3, X_2)$: $X_3 = 2$ is not possible as it does not have a support in $X_2$.

Note that checking in a different order might result in different (but correct) domains.

2. $X_1 = 1 \qquad X_2 = 2 \qquad X_3 = 3$

## 7.3 Search heuristics

When searching, we have two degrees of freedom:

1. Selection of the variable to assign.
2. Selection of the value to assign.
3. Strategy for propagation.

Heuristics can be used to make better selections (for the points 1. and 2.):

**Variable selection** Variable selection heuristics

**First-fail (minimum remaining values)** Choose the variable with the smallest domain. First-fail

**Most-constrained principle** Choose the variable that appears the most in the constraints. In this way, harder variables are ideally assigned first. Most-constrained principle

**Value selection** The choice of the value to assign to a variable does not have general rules. Value selection heuristics

Moreover, heuristics can be:

**Static** Computed before the search. Static heuristics

**Dynamic** Can be computed at each assignment. Dynamic heuristics

## 7.4 Consistency techniques

Consistency techniques reduce the original problem by removing domain values that are surely impossible. This class of methods can be applied statically before the search or after each assignment.

**Constraint graph** Graph where nodes are variables and arcs are constraints. Constraint graph

- There is a directed arc from $X_i$ to $X_j$ if there is a binary constraint $X_i \circ X_j$.
- There is a self-loop to $X_i$ if there is a unary constraint on $X_i$.

**Node consistency (level 1 consistency)** A node $X_i$ is consistent if all of its values in the domain satisfy the constraints. Node consistency

A graph is node consistent if all its nodes are consistent.

**Arc consistency (level 2 consistency)** An arc from $X_i :: D_i$ to $X_j :: D_j$ is consistent if each value of $D_i$ has a support in $D_j$ (as in look ahead). Arc consistency

A graph is arc consistent if all its arcs are consistent.

**AC-3 algorithm** A possible algorithm to achieve arc consistency. Uses a queue to store arcs to explore: AC-3 algorithm

1. At the beginning, all arcs are in the queue.
2. When evaluating an arc $(X_i, X_j)$, values in the domain of $X_i$ are removed if they don't have a support in the domain of $X_j$.
3. If the domain of $X_i$ has been modified, for all the neighbors $X_k$ of $X_i$, the arc $(X_k, X_i)$ is added to the queue.
4. Repeat until empty queue (quiescence).

**Example.** Note: when doing the exercises, it is sufficient to iterate until none of the nodes change, without using a queue.

Consider the variables and constraints:

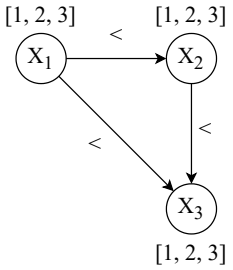
$$X_1 :: [1,2,3] \quad X_2 :: [1,2,3] \quad X_3 :: [1,2,3] \qquad X_1 < X_2 < X_3$$

The constraint graph and the iterations to achieve arc consistency are the following:

1. We check the following pairs:
   - $(X_1, X_2)$. Here, $X_1 :: [1,2,\cancel{3}]$.
   - $(X_2, X_1)$. Here, $X_2 :: [\cancel{1}, 2, 3]$.
   - $(X_2, X_3)$. Here, $X_2 :: [\cancel{1}, 2, \cancel{3}]$.
   - $(X_3, X_2)$. Here, $X_3 :: [\cancel{1}, \cancel{2}, 3]$.
   - $(X_1, X_3)$. Here, $X_1 :: [1,2,\cancel{3}]$ (no changes).
   - $(X_3, X_1)$. Here, $X_3 :: [\cancel{1}, \cancel{2}, 3]$ (no changes).

   The resulting domains are:

   $$X_1 :: [1,2,\cancel{3}] \quad X_2 :: [\cancel{1}, 2, \cancel{3}] \quad X_3 :: [\cancel{1}, \cancel{2}, 3]$$




2. We check the following pairs:
   - $(X_1, X_2)$. Here, $X_1 :: [1,\cancel{2},\cancel{3}]$.
   - Other pairs do nothing.

   The resulting domains are:

   $$X_1 :: [1,\cancel{2},\cancel{3}] \quad X_2 :: [\cancel{1}, 2, \cancel{3}] \quad X_3 :: [\cancel{1}, \cancel{2}, 3]$$

**Path consistency (level 3 consistency)** A path $(X_i, X_j, X_k)$ is path consistent if $X_i$ and $X_j$ are node consistent, $(X_i, X_j)$ is arc consistent and for every pair of values in the domains of $X_i$ and $X_j$, there is a support in the domain of $X_k$. Path consistency

**K-consistency** Generalization of arc/path consistency. If a problem with $n$ variables is $n$-consistent, the solution can be found without search. K-consistency

Usually it is not applicable as it has exponential complexity.

<end of course>