Fundamentals of Artificial Intelligence and Knowledge Representation (Module 1)

Last update: 17 November 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.

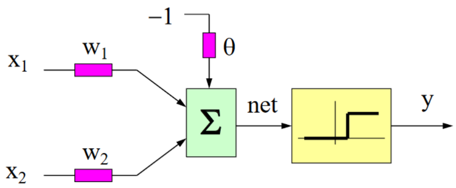


Figure 1.1: Representation of an artificial neuron

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.

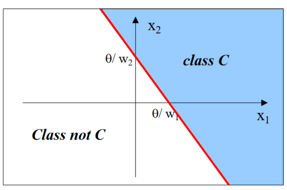


Figure 1.2: Separation performed by one perceptron

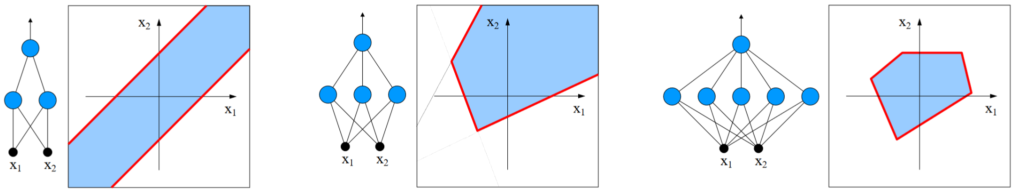


Figure 1.3: Separation performed by a three-layer network

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.

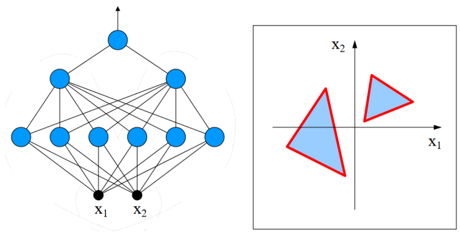


Figure 1.4: Separation performed by a four-layer network

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.

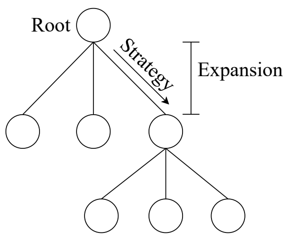


Figure 2.1: Search tree

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 less 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

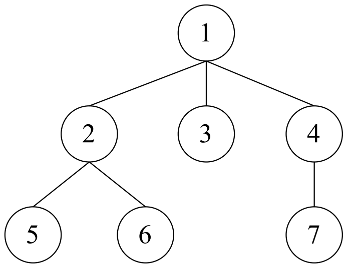


Figure 2.2: BFS visit order

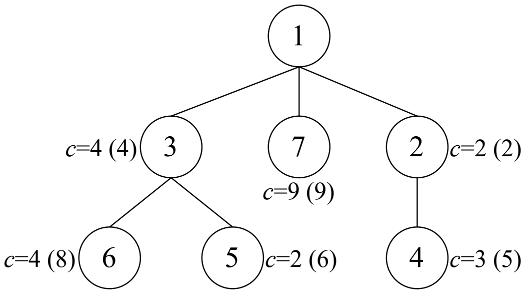


Figure 2.3: Uniform-cost search visit order. (n) is the cumulative cost

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

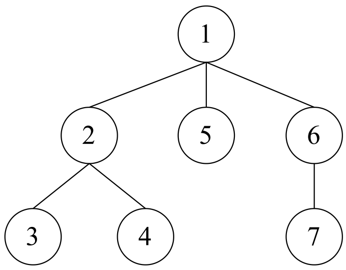


Figure 2.4: DFS visit order

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

Informed search

2.3.1 Best-first search

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.

Best-first search

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).

Greedy search / Hill climbing

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

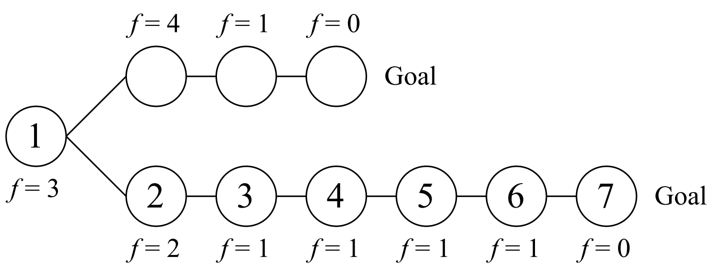


Figure 2.5: Hill climbing visit order

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

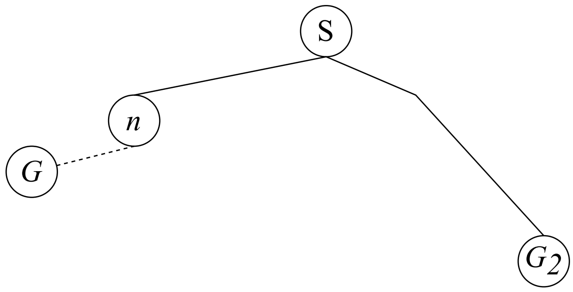
$$h'(n) \leq t(n)$$

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 $\Rightarrow$ 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 generally, it is better to use heuristics with large values (i.e. heuristics that don't underestimate too much).

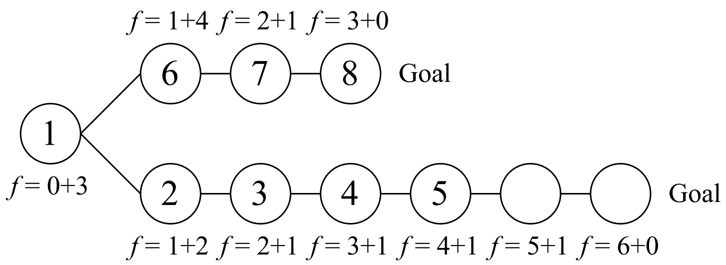


Figure 2.6: A* visit order

2.4 Graph search

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

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

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

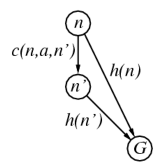
Consistent heuristic (monotone) An heuristic is consistent if for each n, for any successor n' of n (i.e. nodes reachable from n by making an action) holds that: Consistent heuristic (monotone)

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

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).

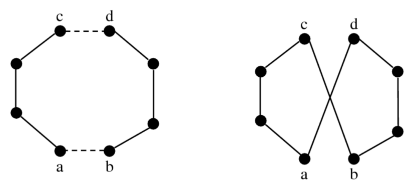
Neighborhood Given a set of states $\mathcal{S}$, a neighborhood is a function: Neighborhood

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

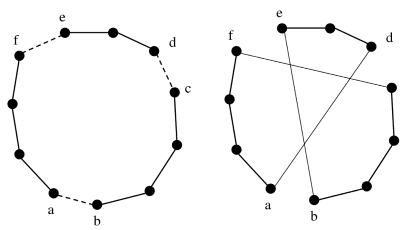
In other words, for each $s \in \mathcal{S}$, $\mathcal{N}(s) \subseteq \mathcal{S}$.

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_{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

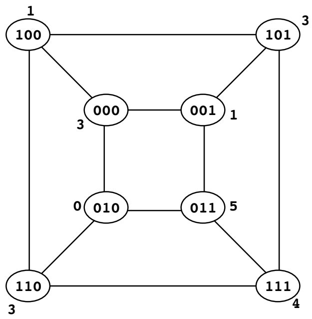
```python
def iterativeImprovement(problem):
    s = problem.initial_state
    while noImprovement():
        s = bestOf(problem.neighborhood(s))
```

3.2 Meta heuristics

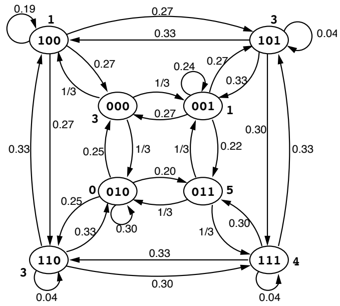
Methods that aim to improve the final solution. Can be seen as a search process over graphs: Meta heuristics

Neighborhood graph The search space topology.

Search graph The explored space.



(a) Neighborhood graph



(b) Search graph. Edges are probabilities

Meta heuristics finds 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

```python
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

```python
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

```python
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 (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.

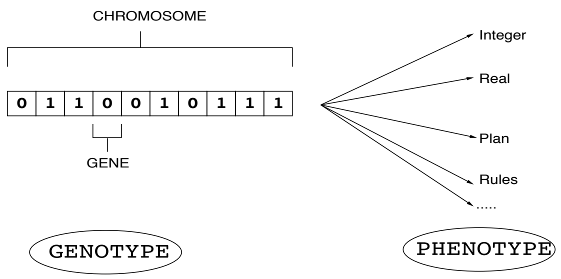
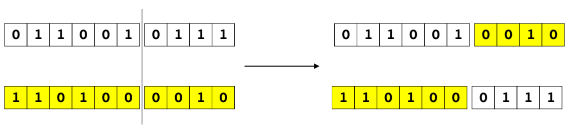


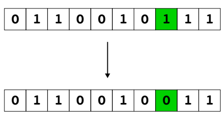
Figure 3.3

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 $x = (x_1, \dots, x_n)$ is a permutation of $(1, \dots, n)$.

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

Crossover Crossover avoiding repetitions.

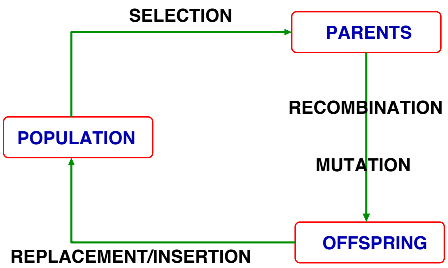


Figure 3.4: Evolutionary cycle

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 α (pheromones) and β (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 τ_{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 $\rho \in [0, 1]$:

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

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

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

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:

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

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 algorithm

```
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 the 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 algorithm

```
def pco(f, n_particles, l, u, ω, φp, φg):
    particles = [Particle()] * n_particles
    global_best = None
    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(g): g = particles.best

    while not terminationConditions():
        for particle in particles:
            rp, rg = randomUniform(0, 1), randomUniform(0, 1)
            xi, pi, vi = particle.value, particle.best, particle.vel
            g = global_best
            particle.vel = ω*vi + φp*rp*(pi-xi) + φg*rg*(g-xi)
            particle.value = particle.value + particle.vel
            if f(particle.value) < f(particle.best):
                particle.best = particle.value
                if f(particle.best) < f(g): g = 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

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. Minimax algorithm

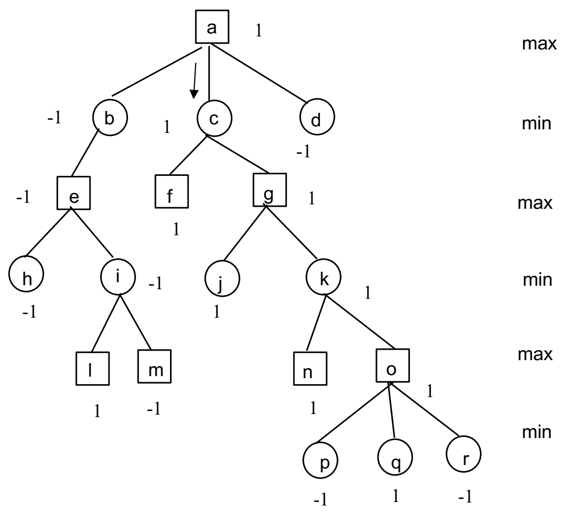


Figure 5.1: Example of game tree with propagated scores

In a game tree, each level represents the actions a that 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 algorithm

```
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). α represents the best choice found for MAX. β 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
```

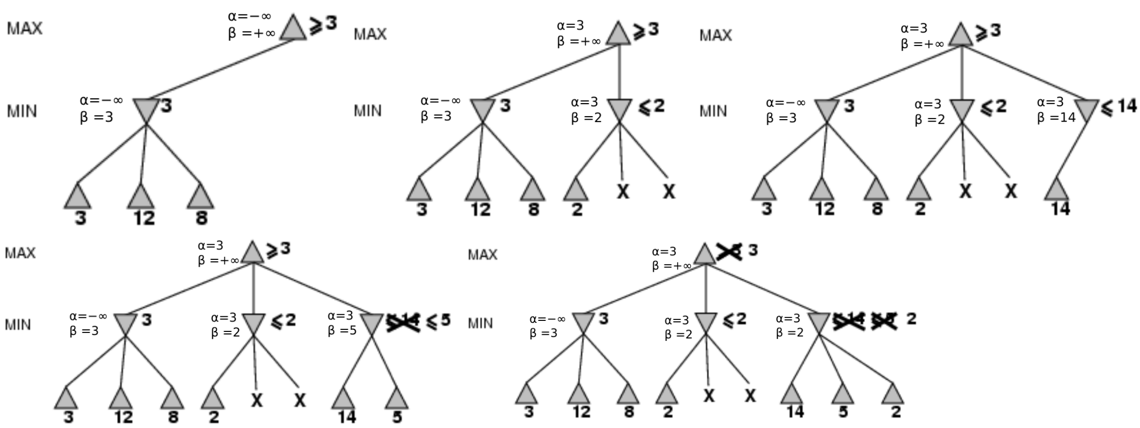


Figure 5.2: Algorithmic (left values) and intuitive (right value) application of alpha-beta

6 Automated planning 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 exits (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.

7 Generative planning

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.

7.1 Linear planning

Formulates the planning problem as a search problem where: Linear planning

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

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

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 define as:

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

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

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

We have that:

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

7.1.1 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:

`do(A, S)` = `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:

`pre-conditions` → `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:

```
holding(X, S) ∧ clear(Y, S) →
                          on(X, Y, do(stack(X, Y), s))∧
                          clear(X, do(stack(X, Y), s))∧
                          handfree(do(stack(X, Y), s))
```

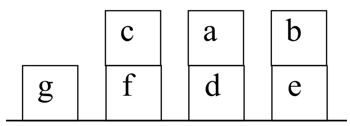
Frame axioms Besides the effects of actions, each state also have to define for all non-changing fluents their frame axioms. If the problem is complex, the number of frame axioms becomes unreasonable.

Example (Moving blocks)**.**

`on(U, V, S),diff(U, X) → 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:

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

This action can be translated into the following effect axioms:

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

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

Given the goal `on(a, b, s1)`, we prove that ¬`on(a, b, s1)` leads to an inconsistency. We decide to make the following substitutions:

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

The premise of `move` leads to an inconsistency (when applying `move` its premise is false):

¬`clear(a, S)`	¬`clear(b, S)`	¬`on(a, Y, S)`	¬`diff(a, b)`
False with {`S/s0`}	False with {`S/s0`}	False with {`S/s0`, `Y/d`}	False

Therefore, the substitution {`s1/do(move(a, Y, b), S)`} 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:

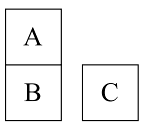
$$\texttt{poss(S)} \wedge \texttt{pact(A, S)} \rightarrow \texttt{poss(do(A, S))}$$

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)

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

7.1.2 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:

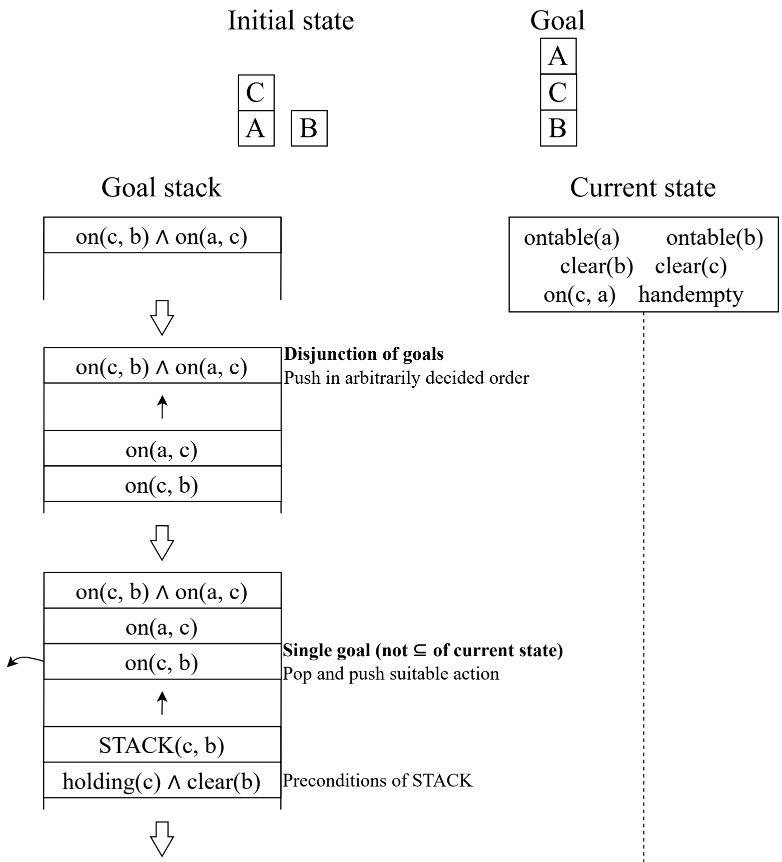
Goal stack Does a backward search to reach the initial state.

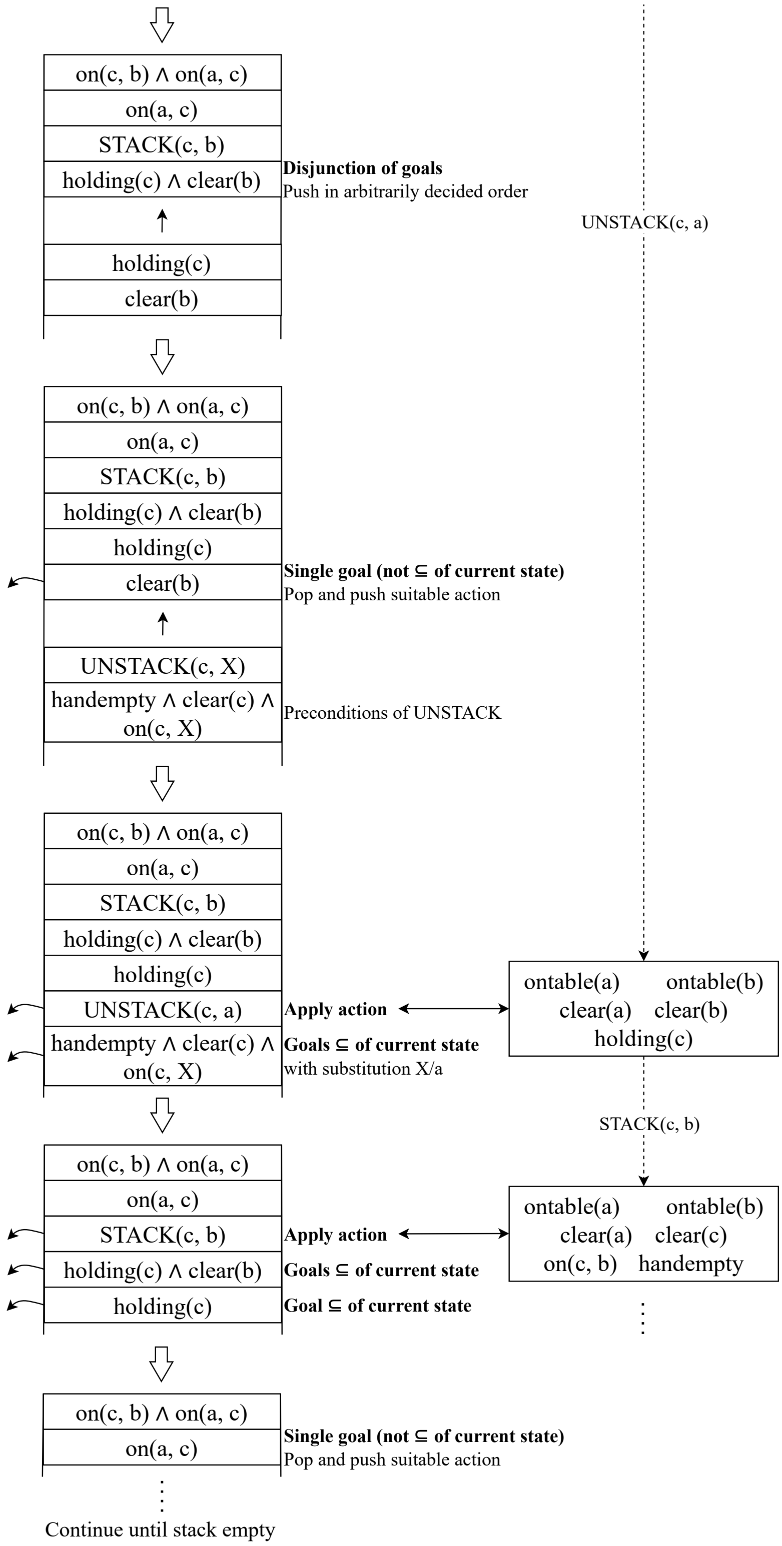
Current state Represents the forward application of the actions found using the goal stack.

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 may become very large. Heuristics may be used to avoid this.
Conjunction of goals are solved separately, but this could 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.

Sussman anomaly

7.2 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. Non-linear planning

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 the effect of S_i and precondition 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.

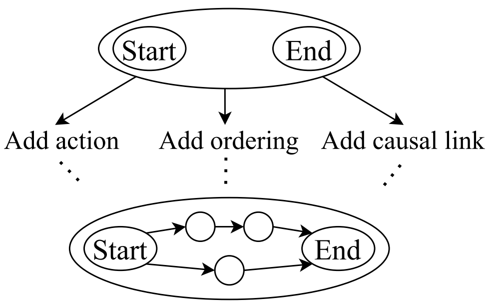


Figure 7.1: Example of search tree in non-linear planning

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

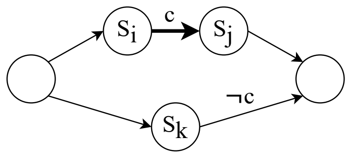


Figure 7.2: Example of threat. Causal links are represented using thick arrows.

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

Demotion

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

Promotion

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

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:

`at(home)`, `have(drill)`, `have(milk)`, `have(banana)`

The possible actions are:

`GO(X, Y)`

Preconditions `at(X)`

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

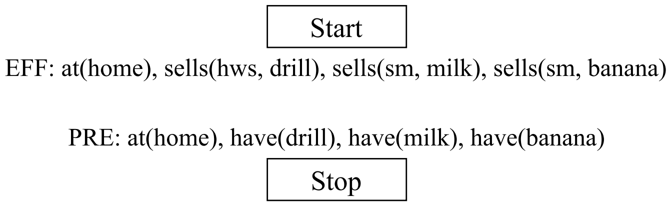
`BUY(S, Y)`

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

Effects `have(Y)`

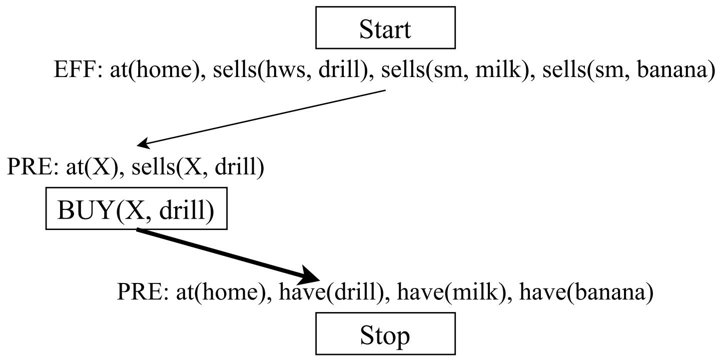
Partial order planning steps are:

1. Define the initial plan:

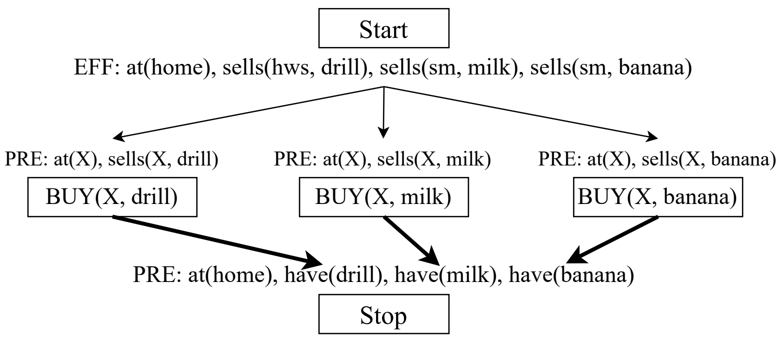


2. The loop of POP is:
   - Choose an action a_i and one of its unsolved preconditions c.
   - Select an action a_j with the precondition c in its effects.
   - Add the ordering constraint **`start`** $< a_j <$ **`stop`**.
   - Add the causal link $\langle a_j, a_i, c\rangle$ (and ordering $a_j < a_i$).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

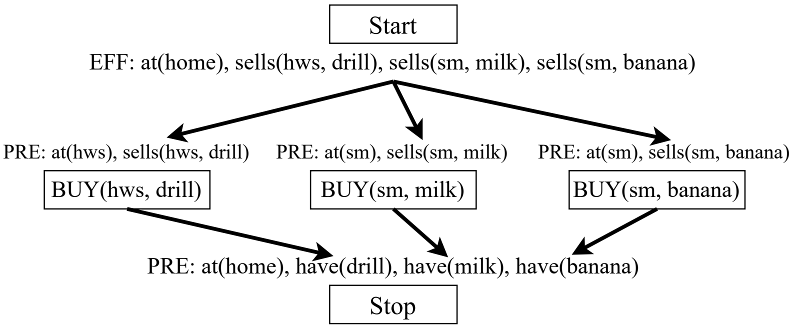
   We choose the action a_i = **`stop`** and the precondition c = **`have(drill)`**. We choose as action with c in its effects a_j = **`BUY(X, drill)`**. We therefore add to the plan the ordering **`start`** < **`BUY(X, drill)`** < **`stop`** and the causal link $\langle$**`BUY(X, drill)`**, **`stop`**,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



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



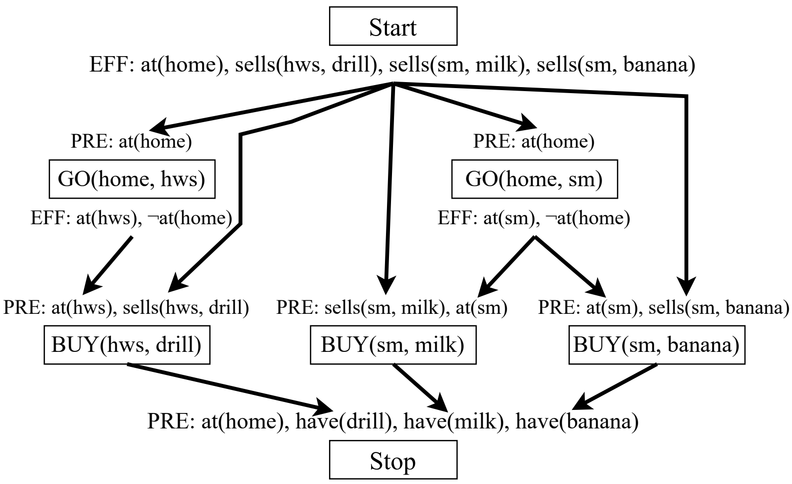
4. Now, we choose as action `BUY(X, drill)` and as unsolved precondition `sells(X, drill)`. This can be solved from the action `start` with effect `sells(hws, drill)`. We make the substitution `X/drill` and add ⟨`start`, `BUY(hws, drill)`, `sells(hws, drill)`⟩ to the causal links. The same process can 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 ⟨`GO(X, hws)`, `BUY(hws, drill)`, `at(hws)`⟩ 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 ⟨`start`, `GO(home, hws)`, `at(home)`⟩ to the causal links.

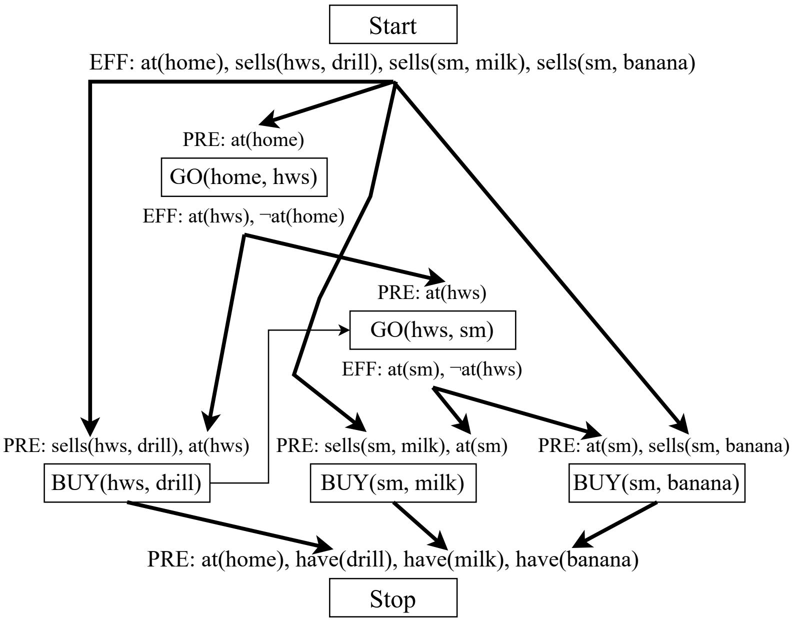
   The same process can be repeated for the `milk` and `banana` branch:



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).

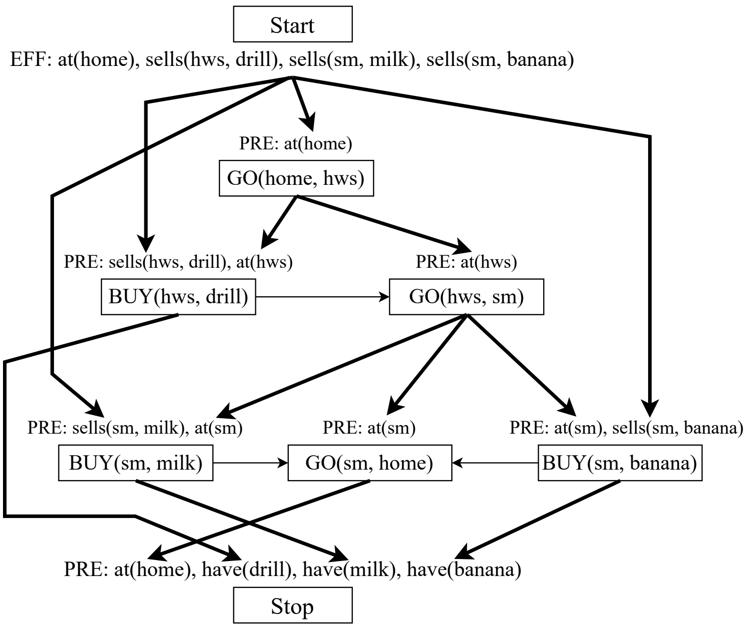
   - Instead of choosing the action `start`, we choose `GO(home, hws)` with the effect `at(hws)`. We therefore make the substitution `X/hws` and update the causal links.
   - We also resolve the threat `GO(hws, sm)` to `BUY(hws, drill)` (it removes the precondition `at(hws)`) by promoting `GO(hws, sm)`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

   `GO(home, hws)` $\rightarrow$ `BUY(hws, drill)` $\rightarrow$ `GO(hws, sm)` $\rightarrow$
   `BUY(sm, milk)` $\rightarrow$ `BUY(sm, banana)` $\rightarrow$ `GO(sm, home)`