**Module 1 FOI**

Many AI problems can be solved by exploring the so called **solution space**, It contains all possible sequences of actions that might be applied by an agent. Some of these sequences lead to a solution. The agent examines alternative sequence of actions that lead to known states and chooses, then, the best one.

The process of trying this sequence is called **SEARCH**. A search algorithm takes as input a problem and returns a solution in the form of a sequence of actions that lead from the initial state to the goal.

The choice on how to expand a search tree is called strategy. There are 2 main strategies:

* Non informed strategies: Do not use any domain knowledge: apply rules arbitarily.
* Informed strategies: Use domain knowledge: apply rules following a heuristics (with a perfect heuristics you do not need search)

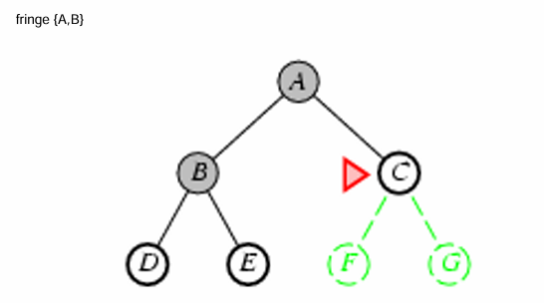
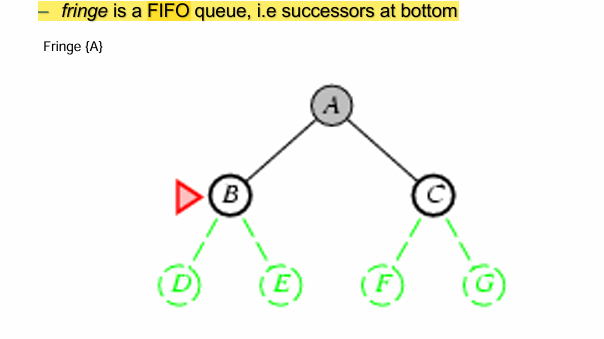
**NON-INFORMED SEARCH STRATEGIES:**

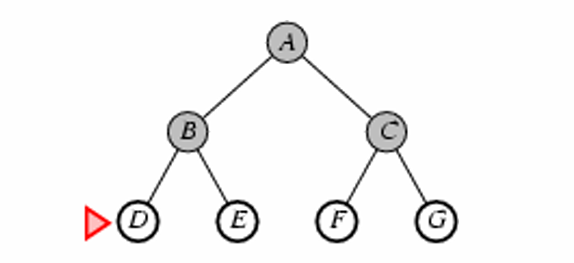
* Breadth-first
* Depth-first
* Depth-first limited depth
* Iterative deepening

Breadth-First Search

The depth of the root node is equal to 0, the depth of any other node is the depth of its parent plus 1.

Breadth-first search always EXPANDS LESS DEEP tree node, it basically means that it procede level by level.

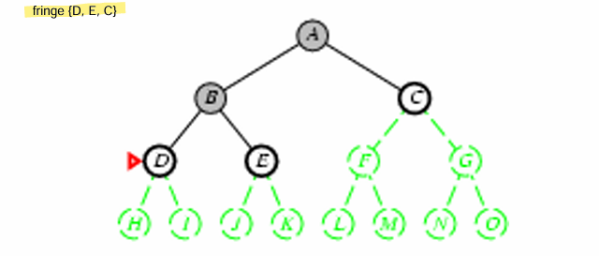
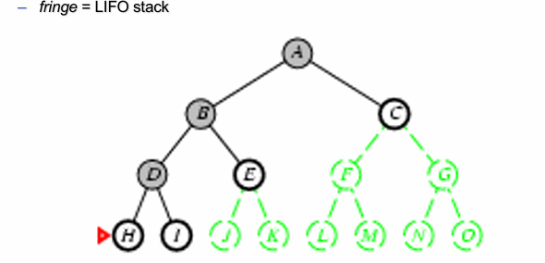
4

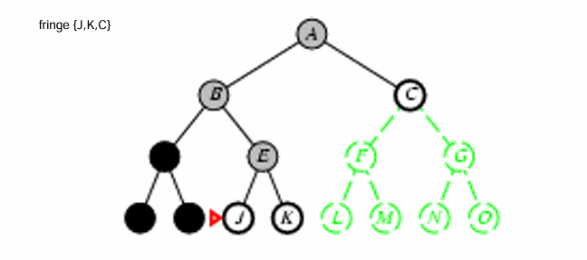


It is complete, so it guarantees to find a solution. The problem is the space that is how much memory is needed to carry out the search.

DEPTH FIRST SEARCH

It does the opposite of the breadth, this search algortihm exapands deepest nodes first, nodes at equal depth are arbitrarily selected (leftmost). DFS requires a modes memory occupation but it can be non-complete.



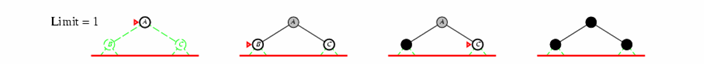


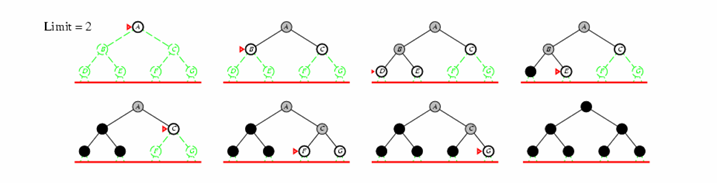
LIMITED DEPTH SEARCH

It is a depth-first variant, it includes a maximum depth parameter, when you reach the maximum depth or a failure, it explores alternative paths. The establish of a maximum limit of depth it does not necessarily solve the problem of completeness, it avoids only infinite branches.

ITERATIVE DEEPENING

Iterative deepening search avoids the problem of choosing the maximum depth limit by trying all possible depth limits. It combines the advantages of both depth and breadth first strategies. It is complete and exploes a single branch at a time.





TWO REASONING MODES

* Forward or Data-Driven: The working memory in its initial configuration contains the initial knowledge about the problem. The process ends with success when the goal to prove is in the working memory.
* Backward or Goal-Driven: The initial working memory contains the goal of the problem. Each time a rule is selected and executed, new subgoals to prove are insereted into the working memory.

**INFORMED STRATEGIES**

Evalution functions give a computational estimate of the effort to reach the final state.

BEST FIRST SEARCH

Best first search uses evaluation functions that compute a number that represents the desirability of the node expansion. Best-first means that the chosen node is the ne considered as the most desirable.

Special cases:

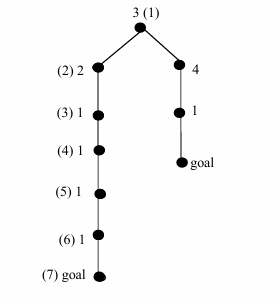
* Greedy search or hill climbing
* A\* search

The best-first try to move to the maximum of a function that “estimates” the desirability to reach the goal. It is not optimal in sense that it is not guaranteed to find the best path toward a solution.

Evaluation Function f(n) = h(n) (heuristic)

H(n) = Estimate of the cost from n to the goal.

The greedy best-first search exapands the node that seems closer to the goal.

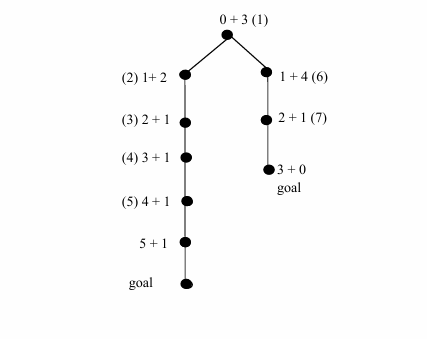


A\* ALGORITHM

Instead of only considering the distance to the goal, also consider the “cost” in reaching the node n from the root. We expand nodes for increasing values of f(n)

f(n) = g(n) + h’(n)

Where g(n) is the depth of the node and h’(n) the estimated distance from the goal. We choose the node to expand as the one for which this sum is smaller.



The algorithm does not guarantee to find the optimal path. In the example, if the node with label 5 was the goal this would have been reached before the goal on the right (optimal).

The heuristic function h’(n) is optimistic that if we always have h’(n) <= h(n) . (h(n) the true distance between the current node and the goal.

Theorem:

* If h’(n) <= h(n) for each node, then the A\* algorith, always finds the optimal path to the goal.

From Trees to Graphs

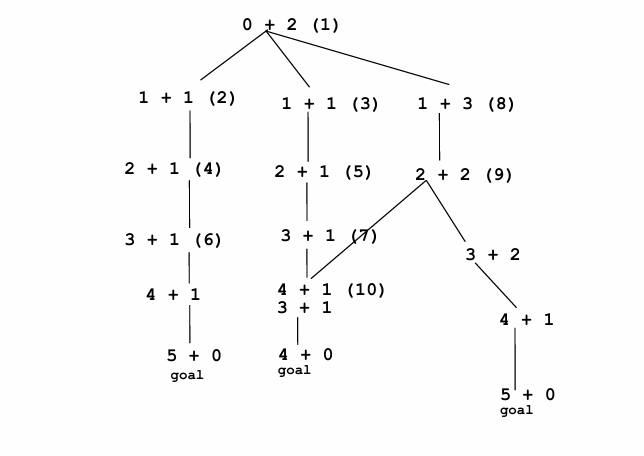
We have assumed so far that the search space is a tree and not a graph. It is therefore not possible to achieve the same node from different paths. Trees has only one arc to enter in each node, and it can have multiple children, instead if a node has 2+ arc we are using a graph.

Search in Graph with A\*

2 List:

Closed nodes: expanded nodes are removed from the list to avoid further examination.

Open nodes: Nodes still be examined



A heuristic is defined consistent if for each node n, any successor n’ of n generated by each action a

h(n) = 0 if the corresponding status is the goal

h(n) <= c (n,a,n’) + h(n’) otherwise

With consistency, we are guaranteed to find the shortest path from the root the goal and the graph-search is optimal.

Esempio pratico

Supponiamo di avere un grafo con nodi A, B e C. Il nodo obiettivo è C.

* Il costo reale da A a è c(A, a, B)=3.
* L'euristica h(B), cioè la stima del costo da B a C, è 5.
* L'euristica h(A), cioè la stima del costo da A a C, deve rispettare:

h(A)≤3+5

Se h(A)=8, l'euristica è consistente. Se h(A)=10, non è consistente e potrebbe portare a errori nella ricerca.

A heuristic is **admissible** if, for every node n, the heuristic value h(n) never overestimates the actual cost of the cheapest path to reach the goal node.

Formally:

h(n) ≤ h∗(n)

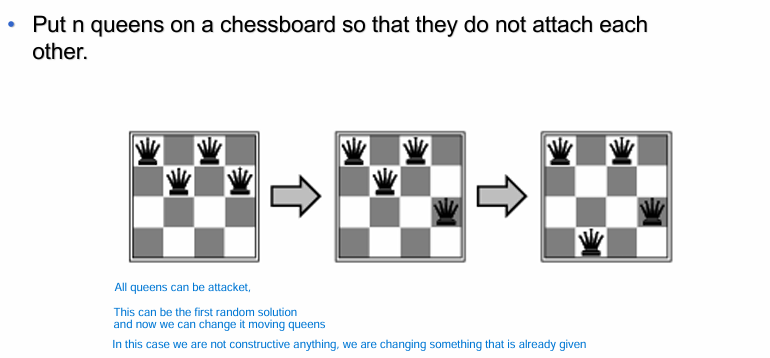
Where:

* h(n) is the heuristic value for the node n.
* h∗(n) is the actual minimum cost (the optimal cost) to go from n to the goal node.

LOCAL SEARCH

The constructive algorithms seen so far generate a solution by adding from a starting state components in a particular order.

**Local Search** is a broad term that encompasses all approaches focused on iteratively improving a solution by exploring its neighborhood.

Local Moves define a neighborhood.

Iterative Improvement

This is a specific local search strategy, characterized by: Starting from an initial solution and a move is only performed if the solution it produces is better than the current solution. It stops when a local optimum is reached.

Local Optimum

A local optimum is not a method but a characteristic of Local Search. It is the point where the algorithm stops because no better solution can be found in the neighborhood. This is can be a limitation, because the Local Search may stop without reaching the global optimum but only a local maximum.

A local maximum (local optimum) is a solution s such that for any s’ belonging to N(s) (neighborhoods), given an evaluation function f.

f(s) >= f(s’)

When we solve a maximization problem we look for a global maximum Sopt such that for any s

f(Sopt) >= f(s)

META HEURISTICS ALGORITHMS

Metaheuristics are general strategies that enhance local search methods. These strategies avoid local optima by introducing randomness and it allow “non-improving” moves to expolre new areas of the solution space

Search starts from an initial node and explores the graph moving from a node to one of its neighbors, until it reaches a termination condition.

* Neighborhood graph: to represent search space topology
* Search graph: to represent the actual search space exploration

Popular Metaheuristic Algorithms:

* Simulated Annealing: explores the search space by allowing “bad” moves, this porbability is controlled and reduced during the search, this is done to move to neighboring solution.
* Tabu Search: Explicitly exploits the search history to dynamically change the neighborhood to explore. It has a tabu list of recently visited solutions or moves and marking them as “forbidden”. This meta-heuristic algorithm solve the problem of the memory, it trys to avoids to explore again visited nodes.
* Iterated Local Search: ir perform a local search to find a local optimum, modify the current solution with a criteria or randomly and perform another local search from the perturbed solution. Repeat the process until a stoopping criterion is met.

The lesson learned is that high-level search strategies have to be applied to effectively explore the search space, like the intensification (deep search around a solution) and the diserfication (is jumping away from a solution).

* Population-Based Metaheuristics

These methods maintain a population of solutions and use collective behavior to guide the search. The basic principle of these algorithms is learning corrrelations betweem “good” solution components. One of these is the Genetic Algorithm.

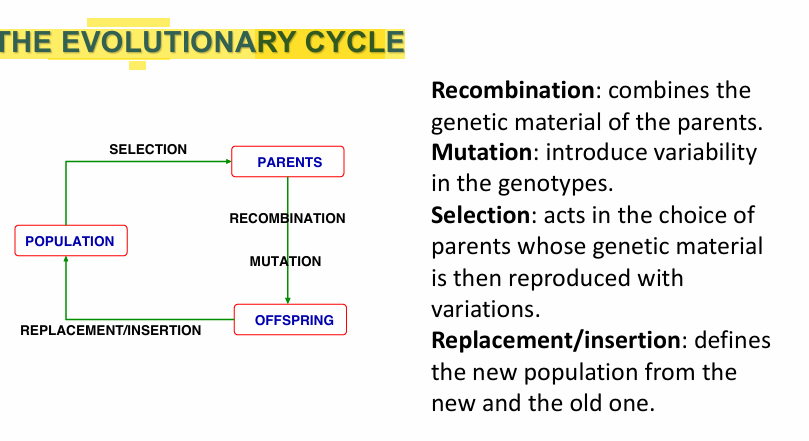
**Genetic Algorithms** (GAs) are inspired by the principles of natural selection and evolution. They simulate the process of "survival of the fittest" to evolve better solutions over generations.

KEY CONCEPTS

• The fittest individuals have a high chance of having a numerous offspring.

• The children are similar, but not equal, to the parents.

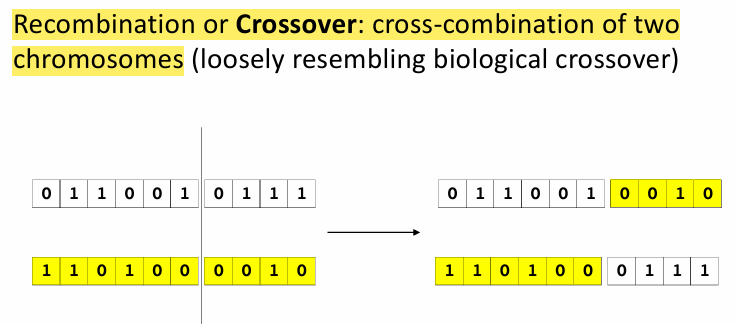
• The traits characterizing the fittest individuals spread across the population, generation by generation.

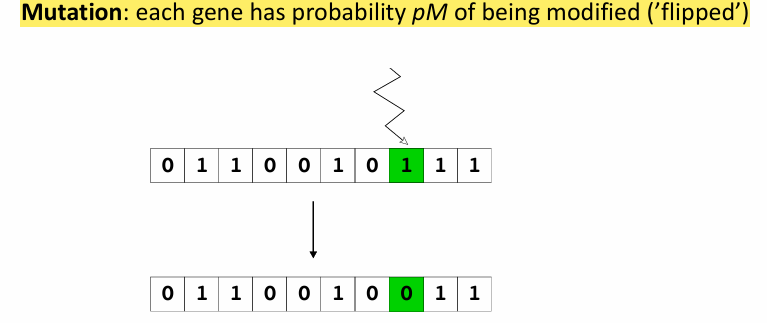


**How It Works:**

1. Generate an initial population of solutions randomly or heuristically.
2. Calculate the fitness of each individual.
3. Select parent solutions based on fitness
4. Perform recombination and mutation to produce a new generation.
5. Replace the old population with the new one.
6. Stop when a fixed number of generations is reached or no significant improvement is observed.

A population here is the set of individuals (solutions).





**SWARM INTELLIGENCE**

Swarm Intelligence (SI) is an artificial intelligence technique based around the study of collective behavior in decentralized, self-organized systems. SI systems are typically made up of a population of simple agents interacting locally with one another and with their environment.

Key Characteristics:

* No central control: Individuals (agents) act independently, based on local information.
* Simple rules: Agents follow simple rules, but their interaction produces complex and intelligent global behavior.
* Adaptability: Swarm systems can adapt to changing environments.

Swarm Intelligence is widely applied in optimization problems and machine learning. Algorithms like Ant Colony Optimization and Artificial Bee Colony Algorithm are key examples which mimic collective behavior in nature to solve complex optimization problems.

Ant Colony Optimization (ACO)

ACO is a family of metaheuristic algorithms, inspired by how ants find the shortest paths to food sources. The idea is that ants deposit a substance called pheromone as they move, other ants are attracted to areas with stronger pheromone trails. Over time, shorter paths accumulate more pheromones because they are traversed more frequently.

The Ant System is the first simplest algorithm in this family.

The problem is represented as a graph, where nodes are solution components and edges are connections with a value (pheromone) that guides the search. Artificial ants start at a random points and build solutions step by step, choosing paths based on:

* Pheromone levels: A measure of how “good” a path was in previous iterations.
* Heuristic information: A measure of how naturally promising a path is (shorter distances)

Once all ants complete their solutions, each one is evaluated. This best solution so far is update.

Pheromone Update:

* Evaporation: Old pheromones fade over time to prevent over-reliance on early solutions.
* Reinforcement: Good solutions are rewarded by increasing pheromone levels on their paths.

This process repeats for a set number of iterations or until the solution quality stops improving.

It is excellent for graph-based problems, explores multiple paths simultaneously.

Artificial Bee-Colony

The Artificial Bee Colony (ABC) algorithm is inspired by the foraging (ricerca di cibo) behavior of honeybees and is used to solve optimization problems.

Bee Roles: The algorithm divides bees into three groups:

* Employed Bees: Explore food sources (candidate solutions) and share information about their quality.
* Onlooker Bees: Choose food sources based on the information provided by employed bees (probability-based selection).
* Scout Bees: Search randomly for new food sources when old ones are abandoned (exploration).

Process:

* Initialization: Randomly generate initial solutions (food sources).
* Employed Bees Phase: Each employed bee improves its solution by exploring the neighborhood of its current food source.
* Onlooker Bees Phase: Onlooker bees select promising food sources and further refine them.
* Scout Bees Phase: If a food source is no longer improving, scouts replace it with a new random solution.

The process repeats until a stopping criterion is met (e.g., a maximum number of iterations or a desired solution quality).

It is used to train neural network

Particle Swarm Optimization

This algorithm is inspired from the analysis of interaction mechanisms between individuals that compose the swarm. With the objective to find food a single individual that finds a food source has two alternatives:

* Move away from the group to reach the food
* Stay in the group

If more than one individual entity move toward the food other flock members do the same. Gradually the whole group changes direction toward promosing areas.

In the algorithm the movement is guided by the best position found so far in search space (from individual to population) and it is updated when better solutions are discovered

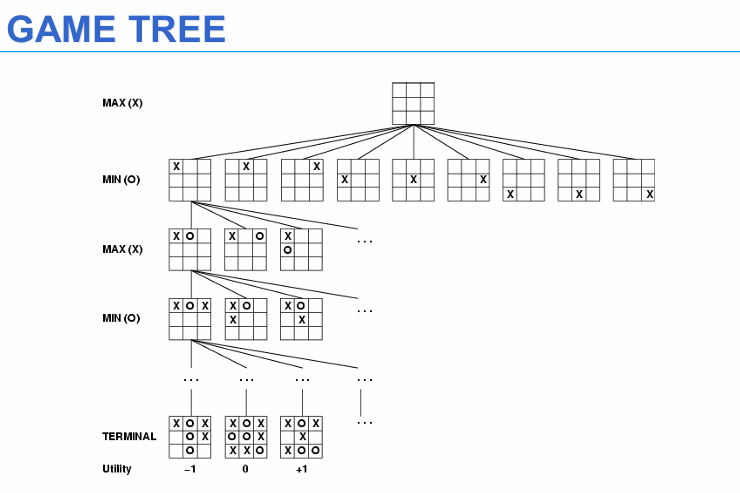
These algorithms are very simple but require an accurate parameter tuning activity

**GAMES**

Multi-agent environment that has to account for the presence of an opponent.

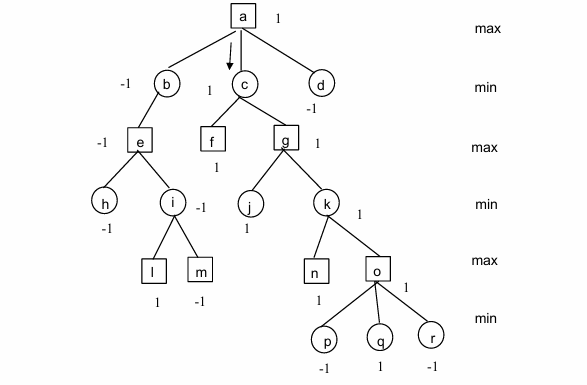
We consider games where there are 2 players (Min and Max) in which the moves are alternated.

The development of a match can be interpreted as a tree in which the root is the starting position and leaves the final position.



MIN-MAX ALGORITHM

The minmax algorith, is designed to determine the optimal strategy for MAX and to suggest, therefore, the first best move to be performed. MIN is assumed to play at his best, we are not interested in the path but only in the next move.



Levels are labeled with -1 and 1

1 is the victory of MAX, and -1 for MIN

For example: Consider the node o, Max should move, the game is over in one move if he moves to q otherwise if he moves to r or p he will lose. So o is a winning position for MAX and is labelled with +1.

If we consider the node k, whatever move MIN does, MIN loses, so the label is +1, instead if we consider the node i, Min has a winning option and the node is labeled with a –1.

In case of of parity the label is 0.

It is complete (the tree is finite) and it is optimal. But it can be too much complex for certain problem, so the solution is to apply the min max algorithm up to a certain depth.

Steps of the min-max algorithm:

* Step 1: Start from the leaves of the tree. Assign a score to each leaf based on whether it’s a win (+1 for MAX), loss (-1 for MAX), or draw (0).
* Step 2: Move up the tree:
  + At a MIN node, choose the minimum value of the child nodes (MIN tries to make it as bad as possible for MAX).
  + At a MAX node, choose the maximum value of the child nodes (MAX tries to get the best outcome).
* Step 3: Repeat until the value for the root node is determine

Temporary Values: It is possible to assign temporary values to nodes while waiting for their children to be evaluated.

Alfa-Beta Cuts

From what we have seen so far computers simply play all possible matches up to a certain depth, evaluate leaves and propagate back the evaluation. So they also consider moves and nodes that will never occur. So to try to reduce the search space one of the best-know technique is the alpha-beta cut.

For example in the previous picture when we find out that the move to c is a winning move, we do not need to expand the nodes b and d. The nodes under those will never influence the choice.

We can call ALFA the value of the best choice found on the path for MAX (the highest) and BETA the value of the best choice found on the path for MIN (the lowest).

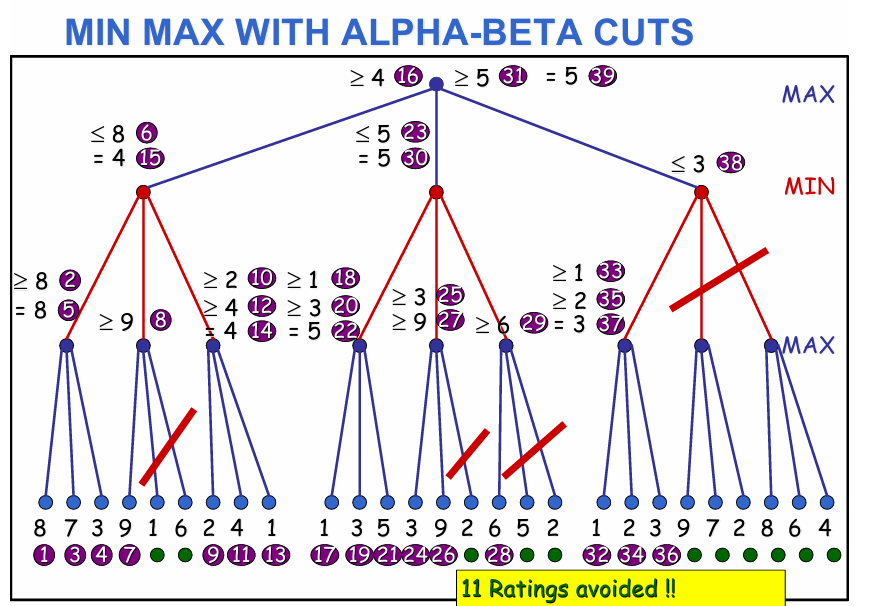
The algorithm updates ALPHA and BETA and cuts branches when their choice the wrost.

Principles:

* We generate depth-first search tree, left-to-right
* Propagate the estimated values from the leaves

If a ALPHA value is greater than or equal than a a Beta value of descending node: stop the generation of children of the descending node!

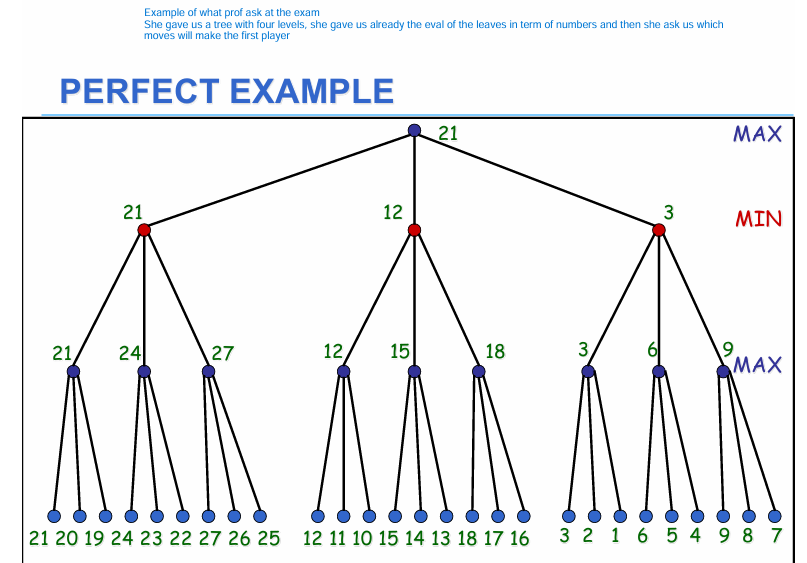
-If a Beta value is smaller than or equal than a descending node: stop the generation of Alpha children value of a of the descending node

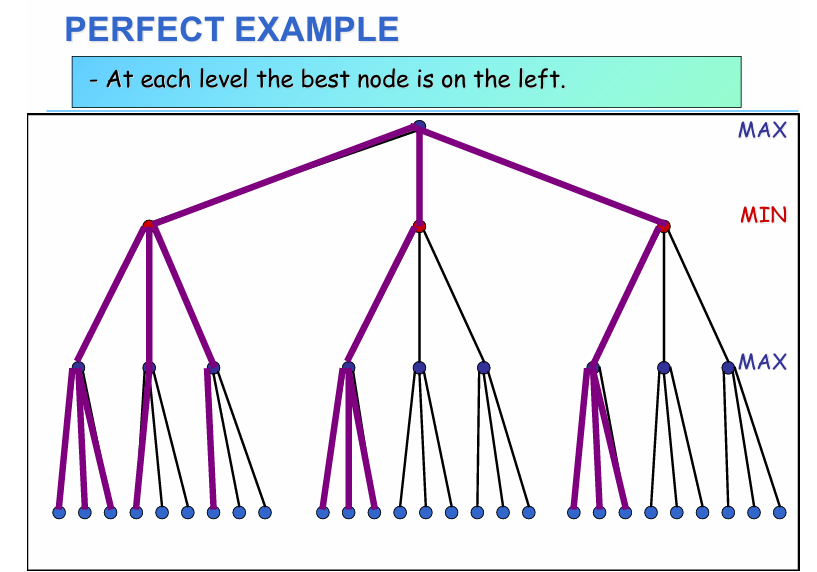


For example we can see that on the left of the picture, at a certain point we are searching a MIN <= 8, because the node max on the left is 8. When we go to the center branch, we select 9 for the Max, and we can notice that 9>8 so, Alpha (Max of the son), becomes greater than or equal to Beta we stop examining the other children because the MIN player will never allow reaching these values.

On the right we can see that we select 3 as Max values on the left branch of the MIN node. So the min node now is <= 3. But we can see that the root node is already >= 5 so, Beta (the MIN of the son) becomes less than or equal to Alpha we stop ecamining the other children because the MAX player will choose a different path.

The best case with this algorithm is when the best nodes are evaluated first. So we can have more cut.





**AUTOMATED PLANNING**

Automated Planning is an important problem solving activity which consists in synthesizing a sequence of actions performed by an agent that leads fron an initial state of the world to a given target state (goal).

The solving process for deciding the steps is called planning. There exist two main types of planning:

* Generative planning, which is a type of planning that produces the whole plan before execution. In particular, it works on a snapshot of the world in which the initial state is fully known, each action has a deterministic e ect and the plan execution is the only cause of changing in the world.
* Opposed to generative planning, there exist reactive planning, which is an on-line type of planning

Given:

* An initial state
* A set of actions you can perform
* A state to achieve (goal)

Find:

* A plan: a partially or totally ordered set of actions needed to achieve the goal from the intial state.

An automated planner is an intelligent agent that operates in a certain domani described by:

* A representation of the initial state
* A representation of a goal
* A formal description of the executable actions

It dynamically defines the plan of actions needed to reach the goal from the initial state.

**Linear Planning**

A linear planner formulates the planning problem as a search in the state space and uses classical search strategies.

The search agorithm could proceed

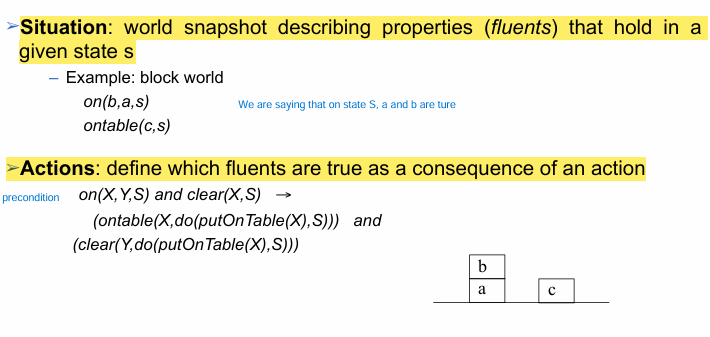
* Forward: if search starts from the initial state and proceeding when it finds a state that is a subset of the initial state that is a superset of the goal.
* Backward: if search starts from the goal and proceeds backward until it finds a state that is a subset of the initial state.

**Deductive Planning**

Deductive planning uses logics for represent how actions lead from an initial state to a goal state.

The main 2 planning are Green and Kowalsky formulations

Often this planning, like Green use as framework Situation calculus, it is a formal logical framework that provides a way to describe the evolution of the world through actions, the main idea is that the world is represented as a series of situations (states), which evolve as actions are performed.



One limitation of the situation calculus is the frame problem, it is the challenge of efficiently representing what does not change in the world when an action occurs, without explicitly listing all unchanged aspect. The frame problem makes planning computationally expensive if unchanged aspects must be explicitly accounted for in every step.

Kowalsky formulation is a logical framework that use predicates as:

* holds(rel, s/a): Describes what is true in a specific state (s) or made true by executing an action (a)
* poss(s): Indicates whether a state s is possible or reachable.
* pact(a,s): Indicates whether an action a can be executed in a state s

If a state S is possible (poss(S)) and an action A can be performed in S (pact(A,S)), then the resulting state after executing A (denoted do(A,S) is also possible (poss(do(A,S)) )

*For example*

How to represent an action called stack(X,Y) using the Kowalsky Fromulation. The action stack(X,Y) represents placing an object X on top of another object Y.

Stack (X,Y)

PreCondition: holding(X), clear(Y)

Effect: add on(X,Y), handempty, clear(X)

Delete holding(X) Clear (Y)

Kowlasky formulation

Preconditions (condition that need to be met):

pact (stack(X,Y), S) :- hold(holding(X), S), holds(clear(Y), S)

Effects/Solution (after executing the action stack(X,Y) the state changes):

holds(on(X,Y), do(stack(X,Y),Sa<))

holds(handempty, do(stack(X,Y),S))

holds(clear(X), do(stack(X,Y),S))

Frame Actions

Frame actions ensure that all other proprieties from the previous state remain unchanged, unless explicitly modified by the action). We have a frame for each action not for every proprities like before with the Green.

holds(V, do(stack(X, Y), S)) :- holds(V, S), V \= holding(X), V \= clear(Y).

This means if a property V is true in state S and is unrelated to holding(X) or clear(Y), then V remains true in the next state

**STRIPS**

It is a specific language for the actions. Easier syntax than the situation calculus. It is used for the plan construction.

Action representation:

* PRECONDITIONS: fluents that should be true for applying the move
* DELETE List: fluents that become false after the move
* ADD List: fluents that become true after the move

Frame problem solved with the Strips Assumption: everything which is not in the ADD and DELETE list is unchanged

*Example:*

pickup(X)

PRECOND: ontable(X), clear(X), handempty

DELETE: ontable(X), clear(X), handempty

ADD: holding(X)

The STRIPS Algorithm is a linear planner based on backward search the initial state is fully known, Closed World Assumption (everything that is not declared in the initial state is considered as false)

**Search in the space of Plans**

Linear planners are search algorithms that explore the state space, the plan is a linear sequence of actions to achieve the goals.

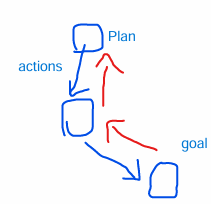
Non linear planners are search algorithms that generate a plan as a search problem in the space of plans. A non linear generative planner assumes that the initial state is fully known *Closed World Assumption*

Moreover there is the Least Commitment planning: never impose more restrictions than these that are strictly necessary.

**NON LINEAR PLANNING**

In the non linear planning, in every node i have a plan, the root node is an empty plan. So it is different from the linear planning where we have states. From the root nodes we have to start add actions in order to reach the goal (every arch between 2 nodes is an action but also we can call each one a plan refiniment operator).

We proceed backward, we start from the goal, and we try to add action backwards to reach the goal (red).



**Partial Order Planning**

It is a refinement of non-linear planning, which organizes actions in a flexible way focusing only on necessary constraints. It uses casual link and threats.

*POP Intuitive Algorithm*

* Choose an action SN that has a precondition not yet satisfied
* Select an action S (can be new) that has the required precondition C (for SN) as one of its effects
* Add the constraint that S must occur beore SN (S < SN)
* Add a casual link <S, SN, C> to record that the effect of S satisfies the precondition of SN
* Solve any threats to casual links by modifying the plan or backtracking

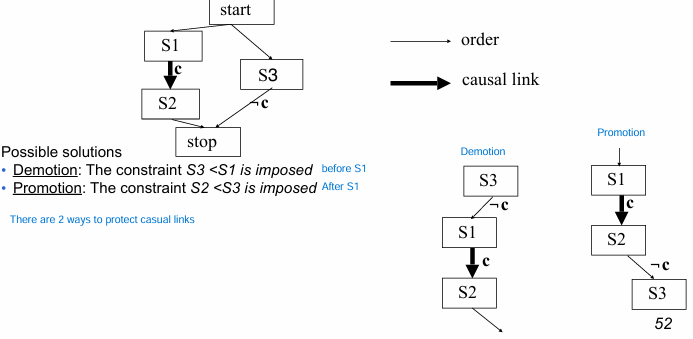
The process contintues until all preconditions for all actions are satisfied and the plan becomes complete.

A casual link is a mechanism used in POP to represent the dependecies between actions. It has three components:

* **S:** An action that produces the effect.
* **Sn:** An action that requires the effect as a precondition.
* **C:** The specific condition that links S to Sn

A threat arises when a third action S\_k has an effect that could interfere with a causal link. For example:

* Causal link: <S, Sn, C>.
* Threat: Action S\_k deletes or modifies the condition C before it is used by Sn.



Modal Truth Criterion (MTC)

Promotion and demotion alone are not enought to ensure the completeness of the planner. A planner is complete if it always finds a solution if a solution exists.

A Modal Truth Criterion is a construction process that guarntees planner completeness.

It introduces the White knight: insert a new operator or use one already in the plan between to S1 and S3 such that it restabilishes the precondition of S3 threatened by S1. In general it is always better to use Promotion and Demotion, becasue they do not add any action to the plan.

**HIERARCHICAL PLANNING**

Classical algorithms have efficiency problems in case of complex domains, there are techniques that make the algorithms more efficient.

Hierarchical planners are search algorithms that manage the creation of complex plans at different levels of abstraction, by considering the simplest details only after finding a solution for the most difficult ones. It starts to plan from the most difficult goals first and after it tries to adjust for the simpliest one.

In particular, there exist two main types of hierarchical planners:

* Planners based on values of criticality assigned to preconditions. These planners are linear planners.
* Planners based on macro operators and atomic operators. These planners are non-linear plan ners.

We can apply thic concept on top of every planner

Popular hierarchical planning algorithms:

* Strips-Like (with values of criticality)
* Partial-Order (with atomic operators (macro operator divided in smaller acitons))

ABSTRIPS

ABSTRIPS is a hierarchical planner which assigns to each precondition a specific criticality value. The algorithm proceeds at different levels of abstraction as follows:

1. A critically threshold value is fixed.
2. All the preconditions whose criticality value is less then the threshold whose value is greater or equal to the threshold value
3. STRIPS, or other planners, nds a plan that meets all the preconditions whose value is greater or equal to the threshold value.
4. The threshold value is than lowered
5. The points 2, 3, 4 are repeated until all the preconditions of the original rules have been considered.

Atomic Operators and Macro Operators

Most hierarchical planners are based on atomic and macro operators.

In particular:

Atomic operators represent elementary actions which can be directly executed by an agent.

Macro operators represents a set of elementary actions which can be decomposed into atomic actions. In this case, the planning algorithm can be either linear or non-linear. A hierarchical non-linear algorithm is similar to POP where, at each step, one can choose between reaching an open goal with an operator (including macro operators) or expanding a macro operator of the plan

**GRAPH PLAN**

Inizio modulo

Fine modulo

Graph planning is one of the most efficient type of complete and correct generative planning.

A planning graph it is a structure that for the first time inserts the notion of Time in the plan construction process, this notion is used to return the shortest possible plan. It is uses the Closed World Assumption.

Action are represented as the one in STRIPS

* Preconditions
* Add List
* Delete List

Moreover there is an action no-op that does not change the state (frame problem)

**Structure of the Planning Graph**

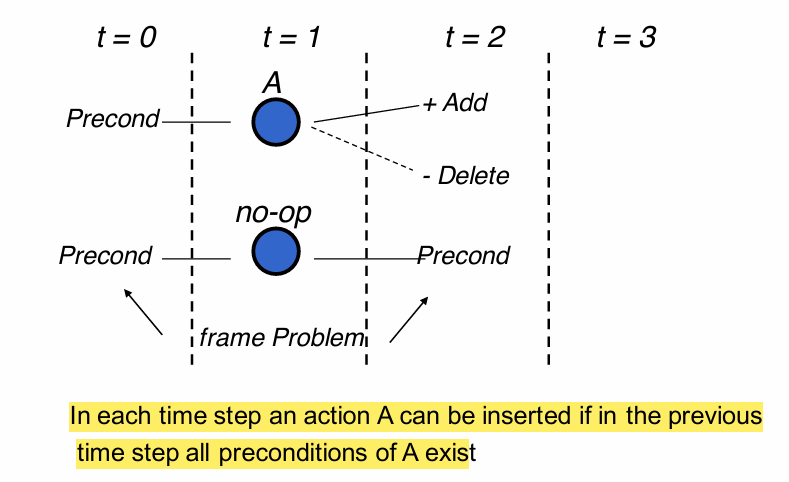
The graph is divided in different levels, nodes belong to different levels and arcs connect nodes in adjacent levels.

* Proposition level: nodes represent propositios
* Action level: nodes represent actions

Level 0 is a proposition level and it corresponds to the initial state.

We have 3 types of Arcs:

* Precondition arcs (proposition -> action)
* Add arcs (action -> proposition)
* Delete arcs (action -> proposition)



In the layer acton we put all the actions that can be executed, thanks to preconditions given at t-1, we do not care if they are incompatible.

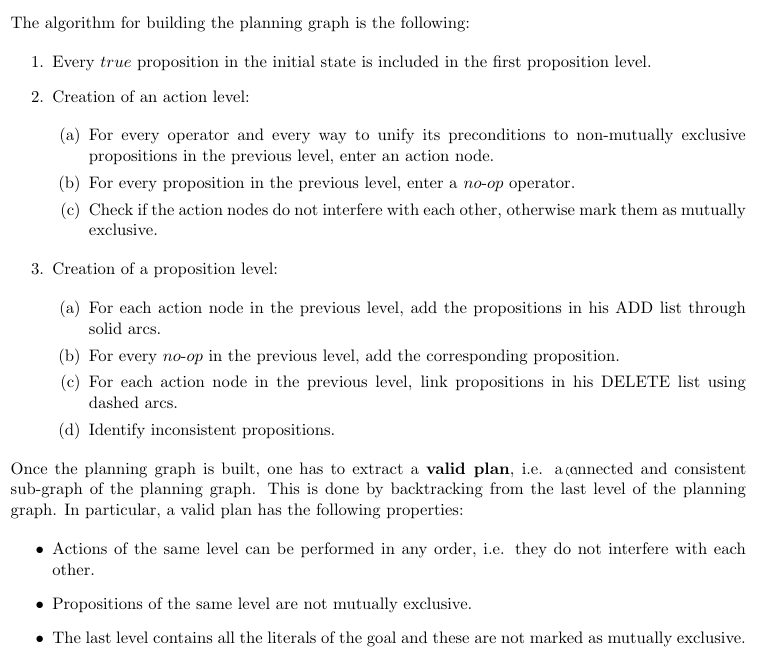
While we are building the planning graph we are not deciding which action will be executed in the solution.

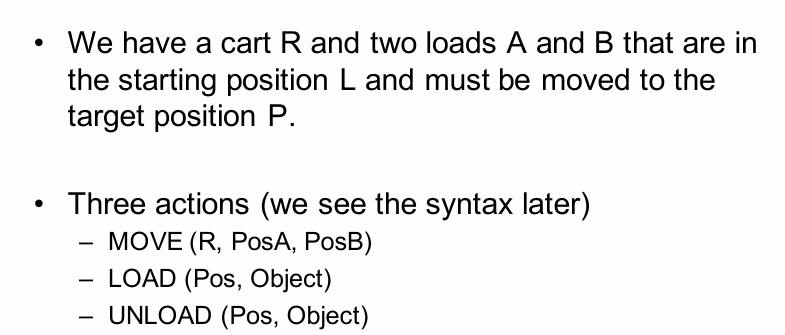
INCONSISTENCIES:

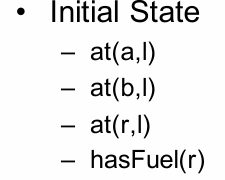
During the construction of the planning graph inconsistencies are identified, in particular

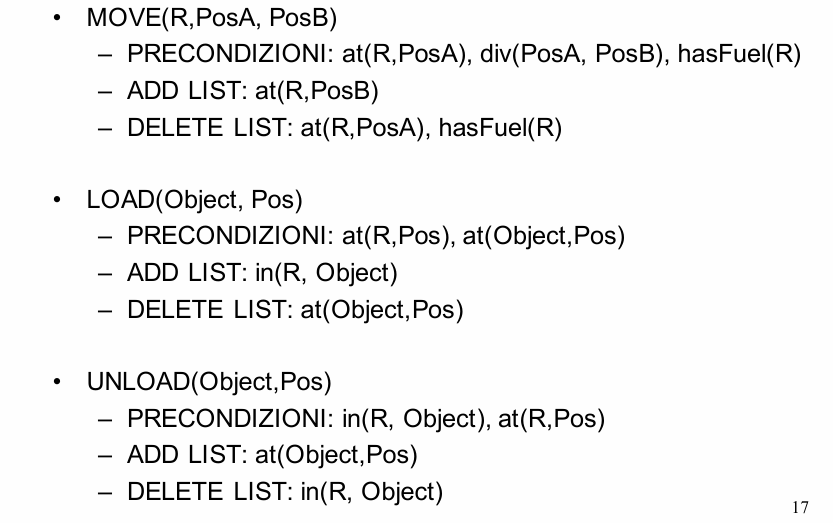
* Two actions can be inconsistent in the same time step
  + Inconsistent effect: one action negates the effect of another
  + Interfence: an action deletes a precondition of the other
  + Competing needs: 2 actions that have mutually exclusive preconditions
* Two propositions can be inconsistent in the same time step
  + One is the negation of the other
  + If all the ways to reach them are mutually exclusive

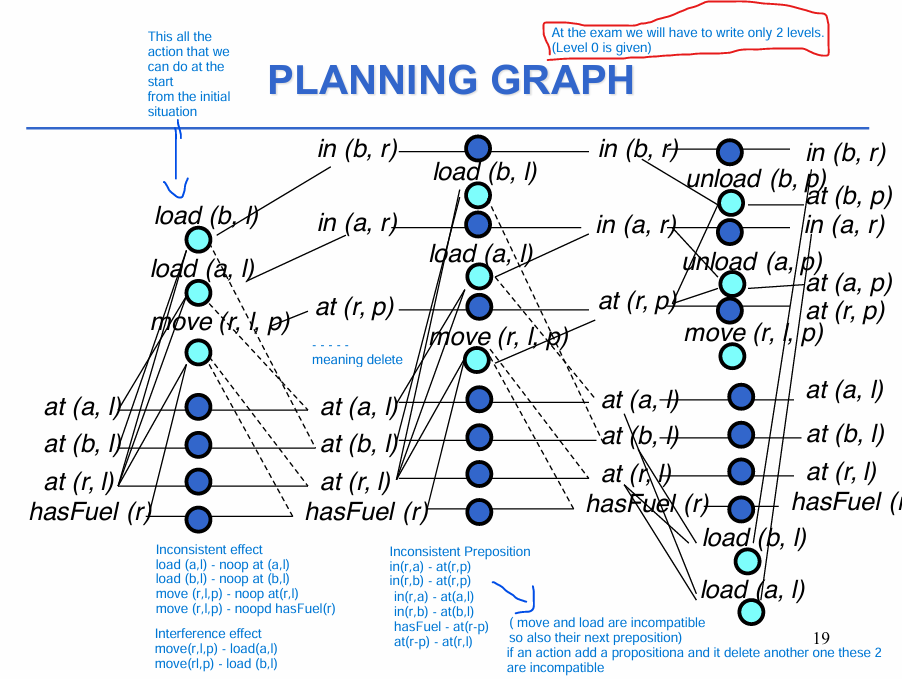
In this case the action / propositionsare mutually exclusive, they can not appear together in a plan but they may appear in the same level of the planning graph.



*Example to undersant how the algorithm works*







**CONDITIONAL PLANNING**

Conditional planning is a type of generative planning that extends its capabilities to handle uncertainty and partially known environments. It does not belong to reactive planning because it creates a structured plan in advance, even though it includes conditions to adapt to different situations.

Generative planners could have some problems during execution. For example, an action could be executed while its preconditions are not satisfied or its effects could be not the one expected. In order to avoid this problem, some planners run under the open world assumption instead of the closed world assumption. According to this hypothesis, all the unknown information is neither true nor false.

In particular:

* Unknown information can be retrieved via sensing actions, which are actions with multiple outcomes that are added to the plan.
* A conditional planner is a search algorithm which generates various alternative plans for each source of uncertainty of the plan.

Conditional planning, however, implies requires a lot of memory to be implemented. Moreover, not always all the alternative contexts are known in advance.

**Reactive Planning**

Reactive planners are on-line algorithms capable of interacting with the world, in order to deal with the dynamicity and the non-determinism of the environment. In particular:

* They observe the world during the planning stage.
* They acquire unknown information.
* They monitor the implementation of actions and check the e ects.
* They interleave planning and execution.

Moreover, pure reactive systems do not plan, they only react as triggers to world variations. Modern planners are hybrid and integrate generative planning and reactive planning in order to exploit the advantages of both approaches.

**CONSTRAINT SATISFACTION**

Many artifcial intelligence problems can be seen as constraint satisfaction problems. The objective of these problems is to find a state in which a set of constraints are met.

A **constraint** is a rule or condition that restricts the values that can be assigned to variables in a problem, ensuring that the final solution satisfies certain requirements.

**Two Main Types of Constraints**

1. **Unary Constraints**  
   These apply to a single variable. For example:

1≤ Xi ≤8

This means each variable Xi​ must take a value between 1 and 8.

1. **Binary Constraints**  
   These define relationships between pairs of variables. For example:
   * Xi ≠ Xj: Two variables cannot have the same value

In general, constraints define what is allowed and what is not in a problem, helping to reduce the number of possible solutions that need to be checked.

**CSP**

A CSP (Constraints Satisfaction Problem) is defined on a finite set of variables:

* (X1, X2, …, Xn) decision that we have to take
* (D1, D2, …, Dn) domains of possible values
* A set of contstraints

A constraint c(Xi1, Xi2 … Xik) between k variables is a subset of the cartesian product Di1 x Di2 x … x Dik that specifies which values of the variables are compatible with each other.

A solution to a CSP provides an assignment of all the variables that satisfies all the constraints.

CSPs can be solved through search, a possible search tree for a CSP is obtained after estabilishing an order for variables: each level of the tree corresponds to a variable and each node corresponds to a possible value assignment.

Each leaf of the tree would then represent an assignment of values to all the variables. If this assignment satisfies all constraints, then the corresponding leaf represents a solution to the problem, otherwise it is a failure.

Consider a depth-first search. It assigns a variable at a time. At each step we either:

• Find a solution • Discover a failure • Assign another variable

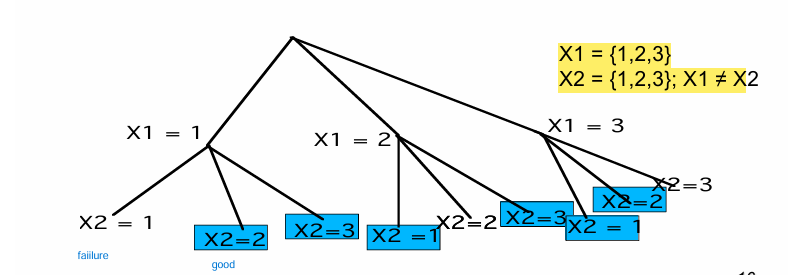
The algorithm has three degrees of freedom:

– the choice of the variable ordering;

– the choice of the ordering of values to be assigned to the current variable;

– the propagation carried out in each node.

The first 2 relate to search heuristics, the third one is what differentiates the different strategies.

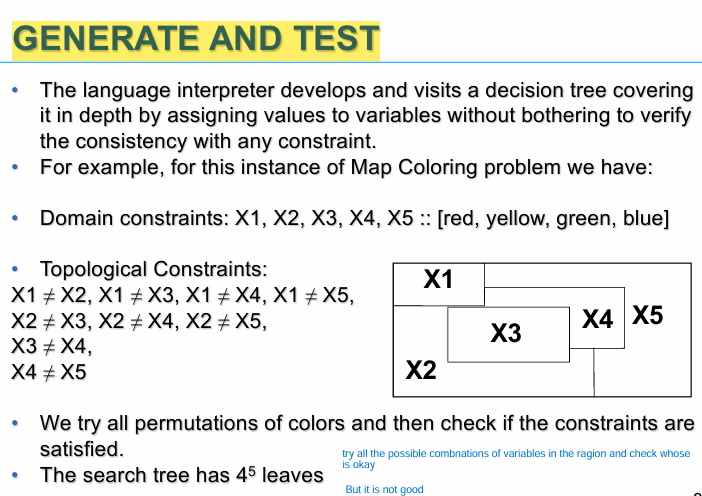
Example

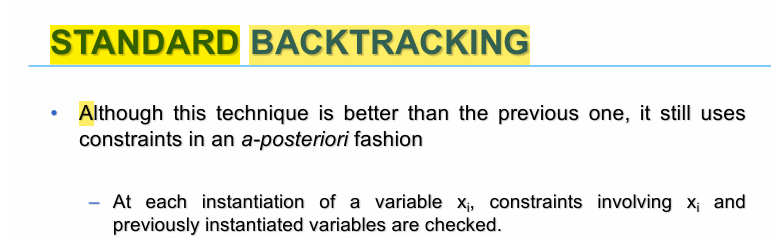
Given a CSP there are 2 possible approaches to its solution:

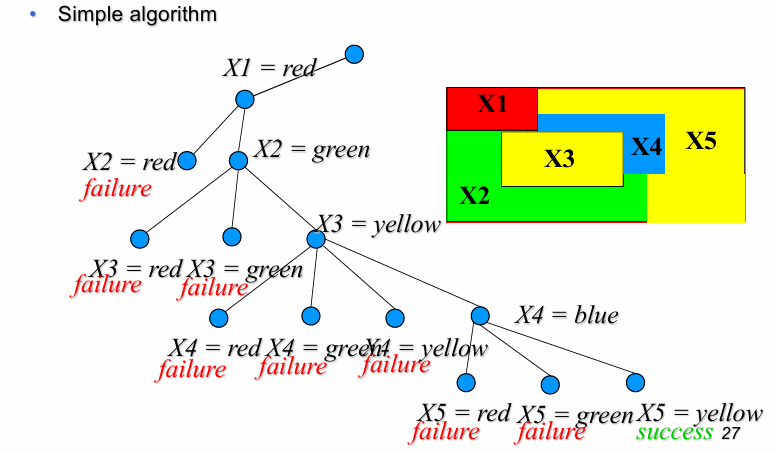
* Propagation algorithms, which are based on the propagation of constraints to eliminate a priori, while searching, portions of the search tree which would lead to a failure. It works during searching.
* Consistency techniques, which are based on the propagation of constraints in order to derive a simpler problem than the original. It works on the original problem.

**PROPAGATION ALGORITHMS**

They use the inverse concept of A Posteriori Algorithms like Generate and Test and the Standard Backtracking. Propagation algorithms work by reducing the domain of variables **before making assignments**, ensuring that only feasible values are considered during the search. Instead **Generate and Test** and **Standard Backtracking** are **not propagation algorithms** because they rely on **a posteriori validation**, meaning constraints are only checked after an assignment has been made.





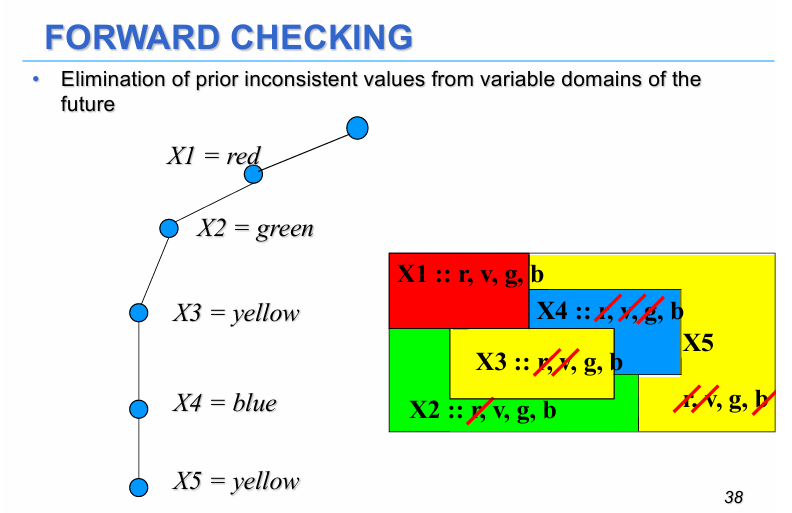


Propagation algorithms are designed to detect inconsistencies as early as possible, reducing unnecessary work. We have 3 major types of Propagation Algorithms:

***1) Forward Checking***

How Forward Checking Works

1. Variable Assignment:
   * At each step, the algorithm assigns a value to the current variable.
2. Constraint Propagation:
   * After assigning a value to a variable, Forward Checking looks at all constraints involving this variable and any unassigned variables.
   * It removes values from the domains of these unassigned variables if they are inconsistent with the current assignment.
3. Failure Detection:
   * If the domain of any unassigned variable becomes empty as a result of this propagation, the algorithm recognizes a failure and backtracks immediately.
   * This avoids wasting time exploring an invalid branch of the search tree.
4. Search Continuation:
   * If no domains are emptied, the algorithm proceeds to assign the next variable and repeats the proces

Example:

Variables X1, X2, X3 represents regions, Domain {red, green, yellow, blue} for all variables.

Constraints: Adjacent regions must have different colors.

**Step-by-Step Process:**

1. Assign X1=red.
2. Forward Checking reviews the constraints X1 ≠ X2, X1 ≠ X3 …
   * red is removed from the domains of X2, X3 , X4, X5 (all adijacent regions)​.
   * Domains after propagation for all variables: {green, yellow, blue}
3. Assign X2=green
4. Forward Checking checks X2≠X3 …
   * green is removed from ​X3, X4, X5 domain.

...

(For X3 we are not going to remove the yelow color to X5 because they are not adjacent)

If all variables are assigned the constraints are satisfied (like the example), instead if the domain of a free variable becomes empty we have a failure.

**Benefits of Forward Checking**

1. **Early Failure Detection:**
   * By detecting failures during propagation, the algorithm avoids exploring invalid branches, saving computational effort.
2. **Smaller Search Space:**
   * Forward Checking reduces the domains of unassigned variables, limiting the number of possible combinations the algorithm needs to explore.

***2) Look Ahead***

It goes beyond Forward Checking, it also checks the constraints **among all unassigned variables** to ensure that the current assignment does not eliminate future possibilities for a solution. It considers all interactions between unassigned variables to ensure that their domains remain compatible with each other. It is more complex than FC.

Example: In addition to checking adjacent nodes in a graph coloring problem, Look-Ahead checks whether the choices made leave valid combinations for all other unassigned nodes.

The are two approaches under the Look-Ahead technique: **Partial Look Ahead (PLA)** and **Full Look Ahead (FLA)**. The difference between these 2 techniques lies in the number of unassigned variables considered when checking value compatibility with constraints.

**PLA**

PLA focuses on the constraints between the **current unassigned variable** (Xh) and the **future unassigned variables** (Xh+1, Xh+2, ..., Xn​).  
For each possible value of Xh​, it checks if there is at least one compatible value in the domains of the future variables (Xh+1, …, Xn​).

Objective**:**  
Ensure that assigning a value to Xh​ does not leave future variables without valid values to satisfy constraints.

Example**:**  
Imagine assigning shifts to employees. PLA ensures that assigning a shift to the current employee leaves valid shifts for the subsequent ones.

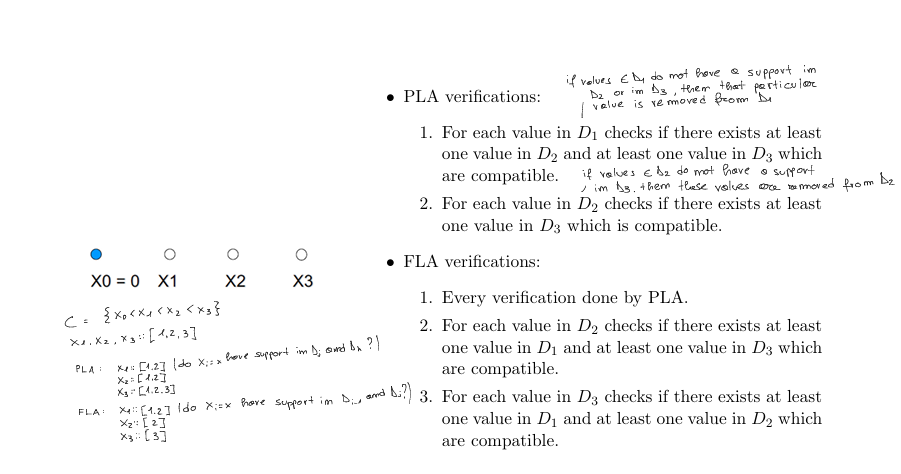
**FLA**

FLA extends PLA. Instead of only verifying the constraints between Xh and future variables (Xh+1, ..., Xn​), it checks **all other unassigned variables**.  
This means that for each possible value of Xh​, FLA verifies compatibility with:

* Future variables (Xh+1, ..., Xn​).
* Any **other unassigned variables**, including those that come **before** Xh​.

Objective**:**  
FLA ensures that assigning a value to Xh leaves valid options for **all other unassigned variables**, not just the future ones.

Example**:**  
Using the same employee shift assignment example, FLA checks not only the compatibility with future shifts but also whether the overall scheduling (including earlier shifts) remains feasible.



Search Heuristics

The heuristics act on these 2 degrees of freedom to try to ensure the achievement of a good solution in a reasonable time.

The heuristicscan be classified into:

• VARIABLE SELECTION HEURISTICS

Determine what should be the next variable to instantiate. The two most commonly used heuristics are the first-fail that chooses the variable with the smallest domain, and most-constrained principle: who chooses the variable appearing in the largest number of constraints. The most difficult variables are assigned first.

• VALUE SELECTION HEURISTICS

Determine what value to assign to the selected variable. There are no general rules here, but try first those values that are most likely to succeed

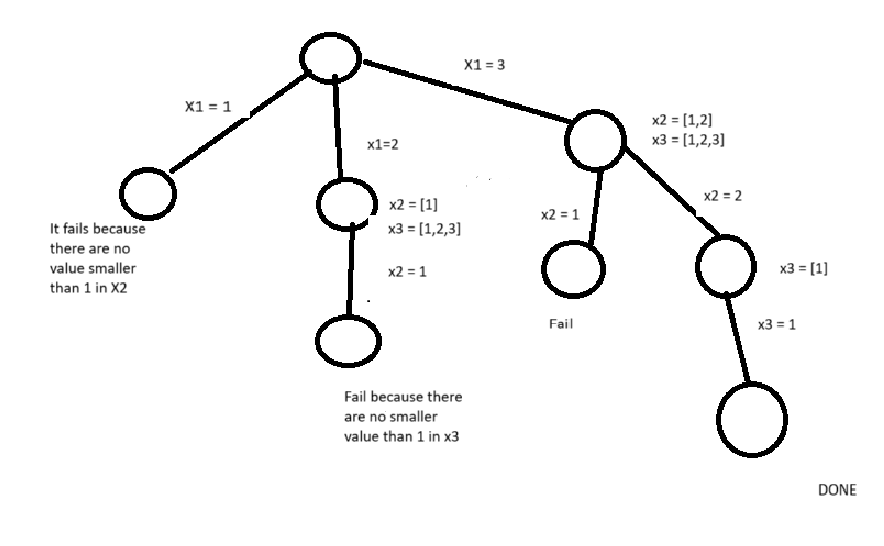
Example in classe

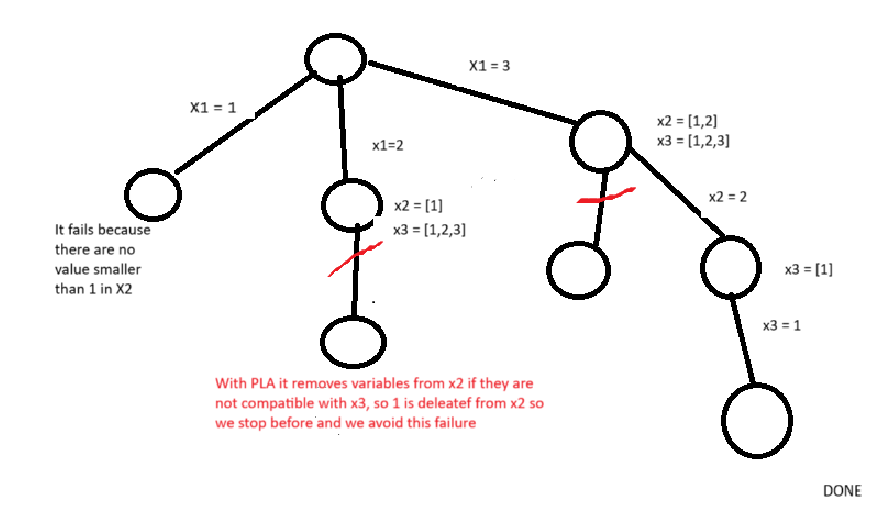
X1: [1,2,3] Constraint: X1 > X2 X2 > X3

X2: [1,2,3]

X3: [1,2,3]

FORWARD CHECKING



Partial Look Ahead

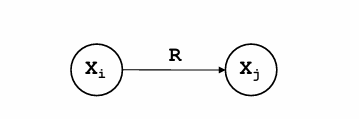
With the full look-ahead it is preatty the same in this example we delete 2,3 from x3 on the top right

**CONSISTENCY TECHNIQUES**

Consistancy techniques reduce the original problem by eliminating domain values that cannot appear in a final solution. This can be appliead at the start, or at every step of assignment as powerful propagation techniques for the not yet instantiated variables.

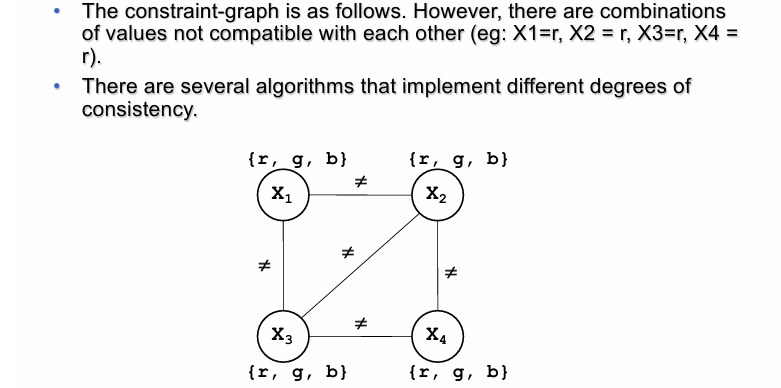
All consistency techniques are based on a representation of the problem as a network (graph) of constraints. The arcs can be oriented or non oriented, the constraint > is represented by a directed arc, while the constraints != ftom a simple arc (undirected or doubly oriented).

**Constraint Graph:** For each CSP exists a graph in which the nodes represent variables and the arcs are the constraints.



The binary constraints (R) connecting 2 nodes Xi and Xj

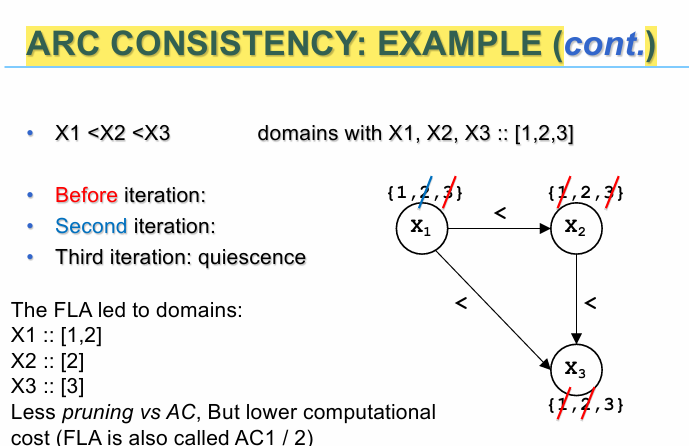
Example : Map coloring problem (2 contiguous regions are colored by different colors)



ARC CONSISTENCY: An arc a(i,j) is consistent if for each value x in Di. It exists at least one value y in Dj such that the constraint between i and j is satisfied.

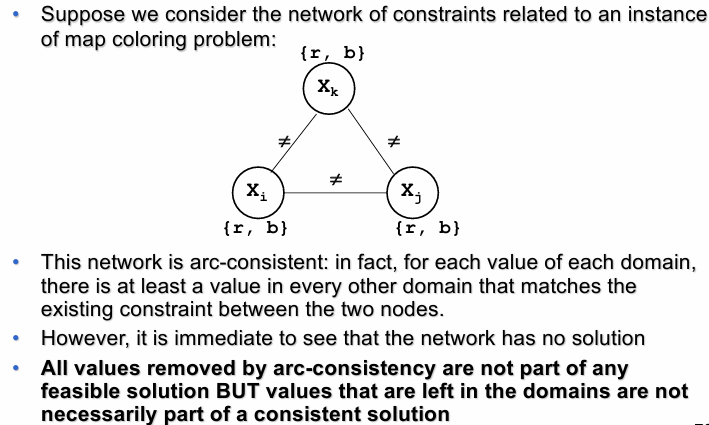
A network is arc-consistent if every arc is arc consistent.

The removal of a value from the domain of a variable makes further checks needed resulting in an iterative process.



The arc consistency can be applied:

* Before the search, as preprocessing to produce a simplified problem with the same solutions of the original one
* As a propagation step (as done for FLA) after each variable assignment it is often called Maintaining Arc Consistency



K-consistency

In principle, for a CSP of k variables, if we want to apply consistency techniques and obtain in variable domains only those values that are part of a feasible solution, we need to apply k-consistency.

For each k-1 tuple of values consistent with the constraints imposed bby the problem, there is a value for each k-th variable that satisfies the constraints among all k variables.

If a CSP containing n variables is n-consistent, then you can find a solution without search