SEARCH STRATEGIES

* A **search algorithm** takes a problem and returns a solution in the form of a **sequence of actions**. This is done by exploring the so–called **solution space**, which contains all the possible sequences of actions, and once the solution is found it will be **executed**.
* The **generation** of the sequence of actions is done following these steps:

1. **Expansion**, from a certain state, new states are generated by applying operators;
2. **Search Strategy**, at each step a new state to be expanded is chose;
3. **Search Tree**, the expansion of all states is represented as a tree.

* The strategies are **evaluated** according to four criteria:
* **Completeness**, i.e. the guarantee that a solution is found if at least one exists;
* **Time Complexity**;
* **Space Complexity**;
* **Optimality**, i.e. the guarantee that the *best solution* is found.

1. NON INFORMED STRATEGIES

* A **Non–Informed Strategy** is not based upon any domain knowledge, hence it can be used equivalently for each problem without any modification.
* We call ***b*** the *branching factor* of the tree, that is to say the average number of children of a node, ***d*** the *depth of the solution node* and ***m*** the *maximal depth* of the tree.
* The **Breadth–First Search** always expand **lesser deep nodes** (*FIFO strategy*), so that each level of the tree has been completely expanded before reaching the next one. It **ensures completeness** (actually, just if *b* is finite) but not optimality unless each operator has unitary cost, and both its **time** and **space complexity** is **O(bd)**.
* The **Uniform–Cost Search** is similar to the *Breadth–First* but it orders the node to be expanded considering their **cost** instead of their depth. By doing so, the **optimality** of the algorithm is guaranteed, though we still have the previous complexity problems.
* The **Depth–First Search** always expand **deepest nodes** (*LIFO strategy*). Though it cannot always ensure completeness due to the possible presence of loops that make ***m*** to be infinite, depth–first is widely used because it as a **linear space complexity** – **O(bm)**. Time complexity is still exponential as in breadth–first:
* the **Limited–Depth Search** is a depth–first variant in which a maximal depth ***l*** is fixed and each node having depth equal to *l* will not be expanded. The pros are that infinite branches are avoided, while the cons are that if a ***d > l*** no solution will be found.
* The **Iterative Deepening Search** combines the qualities of both *breadth–first* and *depth–first*. It basically **reiterates a limited–depth search** increasing the maximal depth *l* for each iteration until a solution is found. It is **complete**, **optimal** and has **linear space complexity** – **O(bd)** – and **exponential time complexity** – **O(bd)**.
* The **Bidirectional Search** is a different kind of search that merges the ***Data–Driven*** (*Forward Search*)approach with the ***Goal–Driven*** (*Backward Search*) approach. It ensures both **completeness** and **optimality** and has both **space and time complexity** **O(bd/2)**. However, it cannot be applicable sometimes.

1. INFORMED STRATEGIES

* An **Informed Strategy** is based upon a certain domain knowledge, generally provided in the form of a **Heuristic Function**, which is a mathematical function ***h***:*Ｓ →* ℝthat maps a state *s∈Ｓ* to a real number:
* the heuristic function provides an estimation of how good or bad is the given state, hence it turns to be useful when the algorithm chooses the next node to be expanded.
* The **Best First Search** uses the heuristic evaluation to order the non–expanded nodes so that the most desirable nodes are expanded first:
* this is a kind of *greedy approach*, as the best node is always chosen, without even consider the path done to reach it;
* even though, in the worst case, *best first search* has both **space** and **time complexity** **O(bd)** and is **non–optimal** and even **incomplete** in some cases (it has the same problems of *depth–first*), a good heuristic function generally improves substantially the performances with respect to non–informed search strategies.
* The **A\* Algorithm** is a sort of extension of the *Best First Search* that considers not only the heuristic evaluation of a node but also the cost of the path to reach it. The function evaluation h(n), where *n* stands for the node, is therefore replaced with a new function: , where g(n) stands for the cost to reach the node *n* from the *root node* and h(n) is the previous heuristic function, which should measure how distant the node **is from the goal:**
* the not yet expanded nodes are now ordered increasingly according to f(n) and the head of the list is chosen and expanded;
* in order to be **optimal**, a **feasible** (*optimistic*) **heuristic** must be chosen for A\*, that is to say a heuristic , where c(n) is the actual cost to reach the solution from the node *n*.
* when A\* is applied on a **graph** rather than on a tree, we need to keep track of both the **closed nodes** (i.e. the expanded ones) and the **open nodes** (i.e. the nodes that have been visited but not expanded yet). Moreover, the heuristic should not be only feasible but also **consistent** (*monotone*), which means that the *triangular inequality* is guaranteed the whole graph, and so we have: , where *n’* is any of the nodes in the graph and *c(n, n’)* is the minimum cost to reach *n’* from *n*.

1. LOCAL SEARCH

* While the previous search strategies behaved as *constructive algorithms* as, starting from an empty solution, they were asked to compute the *path* to reach the solution, **Local Search Algorithms** try to find a solution by **starting from an initial guess** and **iteratively improving it**:
* differently from the *constructive algorithms*, whose purpose is to explicitly *return the path* to get the solution, with **local search algorithms** we are interested in the **final state only** as we completely ignore the path done to reach it;
* in **local search algorithms**, the notion of **completeness** and **optimality** coincides, as we know various solutions (indeed we provide one of them as the initial guess), but we want to find one of those that has certain properties;
* in *constructive algorithms* we needed to list all the possible *operator* that could bring from a partial state to the next ones, in **local search algorithms** we need to define a **neighborhood function**  which maps each state *s∈Ｓ* to a set of states {s1, …,sk}, each one belonging to *Ｓ*.
* The **Iterative Improvement** (*Hill Climbing*) technique is the most basic local search algorithm which greedily search among the states present in the neighborhood of the given state and goes on choosing the best one until no other improvement can be done. Anyway, the main problem of *Hill Climbing Algorithms* is that they can get stuck in non–optimal areas or in areas where the improvement is so slight that the computational time grows too much:
* given a state s, it is said to be a **local maximum** iff , while it is called a **global maximum** iff . Hence, when stopping the *hill climbing* algorithm, we cannot be sure that the solution found is the optimal one, because it could be just the best one into its neighborhood. To reduce this possibility, we can enlarge the neighborhood of each state, bust this has the side effect to increase both the space and time complexity of the algorithm itself;
* over than local maxima, *hill climbing* can also get stuck in **plateaus**, which are those planar regions where each neighbor has the same value for the fitness function that, eventually, can lead to infinite loops over the same states;
* finally, another problem is that of **ridges**, or rather, those peaks that cannot be reached directly, and so there is the need to find other ways to reach them, something that eventually makes the computation very slow and time consuming.
* As *Local Search* has many pitfalls, a more effective and general strategy is required: here come the **Metaheuristics**. In fact, we can see a *local search* like a searching process over a graph having the initial guess as root and where the children of each node are its neighbors. Hence, *metaheuristics* are techniques that help searching in that graph in an efficient and effective way:
* the difference between a **heuristic** and a **metaheuristic** is that, while the former is *problem–dependent*, which means that for different problems there is the needs to specify different heuristic functions, the latter is ***problem–independent*** as the heuristic provided is used at a higher level to navigate the neighborhood graph, which will always have the same characteristics independently from the domain of the problem;
* generally, **metaheuristics** relies on a **non–deterministic approach** that, in most of the cases, allows to reach a pretty optimal solution in a shorter amount of time;
* *metaheuristics* are divided into **single–solution metaheuristics**, where the initial guess is of one state only, and **population–based metaheuristics**, where the initial guess is a population of different states).
* The **Simulated Annealing** is a *single–solution metaheuristic* technique that tries to avoid getting stuck on local optima by **allowing moves towards worse neighbors** **with a** **probability p** which decreases as time passes (the metaphor is that of cooling the temperature, which in physics reduces the energy and, therefore, the likely of moving to a worst state):
* in general, a new state s’ is picked from the neighbors *N*(s) of the current state s then, if f(s’) happens to be better than f(s), s’ is taken as the new state, otherwise it is taken with a certain probability p(T, s’, s) which is computed upon the temperature T and both the values of the fitness function computed for s’ and s.
* The **Tabu Search** is a *single–solution metaheuristic* that **keeps track of the k last visited states and avoids choosing them again**. By doing so, *Tabu Search* prevents from getting stuck into infinite loops, especially over plateaus.
* The **Iterated Local Search** is a *single–solution metaheuristic* that, after performing a classic local search over the initial guess s­0 that leads to a solution s0\*, it randomly perturbs this solution to get a new starting guess s1 and perform a classical local search over s1 and takes it if its fitness function is better than that one of s0\*. This process is iterated many times until one or more stopping criteria is satisfied:
* it is usual to combine those single–solution metaheuristics all together so that, starting from a guess s0, a *Simulated Annealing* is performed over this guess avoiding loops thanks to the *Tabu Search* technique, then the solution is perturbed repeatedly as in the *Iterated Local Search*.
* A **Genetic Algorithm** is a *population–bases metaheuristic* that works like a metaphor of **organisms’ evolution**. It takes into account three main factors – ***adaptation***, ***inheritance*** and ***natural selection*** – and probabilistically searches over a generation to recover the fittest individuals and use them as a starting point for the next generation:
* at first, an initial population is provided by the user;
* from this zero–generation, some individuals are chosen to be the parents of the next generation – the choices are *taken probabilistically* *but* *using as weights the values of the fitness function* of each individual so that the best individuals have more probability to be chosen;
* the offspring is obtained by applying *recombination* (crossover) among the parents and *mutations* over themselves so that those individuals are different from the previous one;
* at the end, the offspring can either entirely replace the old generation to form the new one (*replacement*), or the fittest individuals of both the old generation and the offspring can be taken to form the new one (*insertion*);
* the process is iterated until some stopping condition is verified, and eventually the best treats of each generation will be spread until the end, hence the final generation will contain the fittest individuals, from which we take the best one as solution.

1. SWARM INTELLIGENCE

* The **Swarm Intelligence** is a technique that exploits social and collective behaviors to develop a well–organized system from a multitude of single individuals:
* each individual, in fact, has no global view, but just performs simple tasks following basic rules that are consequences both of individualistic and social behaviors such as **local communication patterns** with its neighbors (e.g. the so–called **stigmergy**, which happens when an individual modifies the environment and its neighbors react to this change, like the ants do by leaving pheromone on the ground);
* on a global scale, the whole is greater than the sum of its parts, as each single individual will follow its counterparts that are behaving well thanks to a **positive feedback** that will lead to convergence to optima, while bad behaviors of single individuals will not be followed by any other individual, killing those behaviors thanks to a **negative feedback**;
* in general, *swarm intelligence* techniques rely on **multiple interaction among agents**, be them *rule based* (simple agents) or *multi–agent systems*, which also allows to perform a **distributed computation** due to de decentralized nature of the technique itself;
* the cons of those techniques, however, is that **they must be tuned** as they are generally based upon a series of hyperparameter which can significantly change how fast the algorithm converges to a solution and how good the solution is.
* **Ant Colony Optimization** (*Dorigo, 1992*) is a swarm intelligence technique that mimics the behavior of an ant colony trying to find food sources. Indeed, it has been discovered that each **ant deposit pheromone trails** when walking from the nest to the food and vice versa, and also they tend to **choose path marked with higher pheromone**. By doing so, the cooperative interaction reinforces the shortest paths that brings to food sources:
* *ACO* has a **probabilistic parametrized model** used to model the pheromone trails and can also use a **heuristic value** that represents prior information about the problem (this is useful especially during the initial phase of the algorithm, when the first paths must be marked);
* to each arc (i, j) is associated a pheromone value τij and, after each step, a certain quantity Δτij of pheromone is left on the arc for each of the *k* ants that passed through it. The pheromone value is then updated by adding each Δτij to the value (1 – ρ) ⋅ τij, where ρ is a hyperparameter called **evaporation coefficient** which models the evaporation of the pheromone during time;
* the update of pheromone values is always performed on-line, but it can be done either **step–by–step**, hence the pheromone is deposited at the end of each arc proportionally to how much the path is short, or in a **delayed** way, hence the pheromone is deposited at the end of the entire path proportionally to how much the solution was good.
* **Artificial Bee Colony** (*Karaboga, 2005*) is a swarm intelligence technique based on three different kind of individuals, namely **employed bees**, that are associated with a specific nectar source, **onlookers**, which follow one of the employed bees to the same nectar source, and **scouts**, that discover new food sources:
* when a *scout bee* found a nectar source, it becomes an *employed bee* and starts retrieving food until it finishes, then it becomes a *scout bee* again until it finds another nectar source;
* after an **initialization phase,** in which it randomly creates an initial population of variables (*the food source positions found by the scout bees*) between a lower and an upper bound, the *ABC* algorithm cycles through *three main subsequent phases*:

1. the **employed bee phase**, where each employed bee goes to the respective food source and starts looking in the neighborhood to improve the solution locally – if there no better can be found, the *employed bee* becomes a *scout bee*, causing a **negative feedback**;
2. the **onlookers’ phase**, where each onlooker bee probabilistically chooses to follow one of the *employed bees* and starts looking around its neighborhood to improve the solution – the choice is weighted basing on the quality of the respective nectar source, causing a **positive feedback**;
3. the **scout phase**, where each scout bee randomly chooses a new nectar source.

* **Particle Swarm Optimization** (*Kennedy-Eberhart, 1995*) is a swarm intelligence technique that mimics the behavior of a bird flock moving towards a common objective (e.g. reaching a food source). Each single individual, then, can either decide to **move away from the group** (*individualistic choice*) in order to do some exploration, or to **remain in the group** (*social choice*) in order to follow its counterparts:
* when an individual finds a food source, the information will be spread and other members of the flock will follow it hence, gradually the *PSO* algorithm will **converge to promising areas without being stuck into local optima** as part of the flock will still explore the space;
* moreover, the **concept of proximity** must be modeled to avoid a too slow or too fast spread of information that will lead to ignoring optima or getting suck in local ones respectively, hence generally individuals area subdivided into groups that can talk together, and various individuals can belong to different groups so that each information can be globally spread sooner or later;
* the algorithm stores the best solution found so far by the whole flock (**g**) and, for each particle *i*, its position in the N-dimensional space (**xi**), its speed (**vi**) and the best solution found so far by itself (**pi**), then

1. GAMES

* An **intelligent player** for a specific game can be created using **search strategies** over the **game tree**, which is the tree having the current state of the game as root node and each possible state that can be reached from a certain state as the children of the state:
* here we consider just ***two-player zero-sum games*** (the player will be called *MIN* and *MAX*, as their objectives will be complementary) in which each player has a ***perfect knowledge of the state*** (i.e. there is no hidden information like in card games).
* The **Minimax** (or *Minmax*, or *MIN-MAX*) algorithm is designed to **determine the optimal strategy for MAX** and, therefore, to suggest which one of the possible moves to do next, as we are actually **interested just in the next move**, not in the whole path that will lead to a win or a loss:
* first of all, the entire tree is created until each leaf node, representing a terminal state, is labeled with a *+1* in case of MAX’s win, a -1 in case of MIN’s win or a 0 in case of a draw, then each level, starting from *MAX* at level 0 (the root node), is alternatively labelled with *MIN* or with *MAX*;
* the information obtained in the leaves is now propagated back until the root node so that each internal node will assume the maximum or minimum value of its children if it is in a level labelled as *MAX* or *MIN* respectively;
* the root node, MAX, finally chooses the state that gives a higher value and makes its move according to that.

The expansion of the entire tree, however, has to be done with a search strategy and, therefore, can have a space complexity **O(bm)** or **O(bm)** depending on the strategy used, but always have a time complexity **O(bm)**:

* for most of the games, which have great branching factors, this approach is totally inefficient, hence it is compulsory to provide a **heuristic function** that returns a real value belonging to the closed interval [-1, 1] in order to label nodes up to a certain level, even though they do not represent a terminal state.
* The **Alpha-Beta Pruning** technique allows to **reduce the search space** by pruning some branches of the game tree without proceeding with the exploration:
* to each node *n* of the game tree are associated two values, namely **α** and **β**, which contain the **maximum value found out by MAX so far** and the **minimum value found out by MIN so far** respectively;
* whenever a **max node** has a value or a **min node** has a value (in general, if both **α** and **β** are constantly updated, those two conditions can be joined into the condition ) there is no need to proceed with the expansion as the actual value of this node will never be taken into consideration by the upper levels;
* in the worst case, the time complexity will be **O(bm)** as no branch will be pruned, while in the best case, when the best nodes are always evaluated first, the complexity will reduce to **O(bm/2)**, hence it is important to choose a good order of the children of a node to improve performances - in general, considering a random distribution, the complexity will be **O(b3m/4)**.

AUTOMATED PLANNING

* **Automated Planning** is a problem-solving technique which consists in synthesizing a **sequence of actions** (the ***plan***) performed by an agent to **reach a certain goal** starting from a given **initial state**:
* the *Automated Planner* is the agent itself that operates in a certain **domain** described by the ***initial state***, the ***goal*** and a ***formal description of the executable actions*** (also called *operators*).
* Each **action** (or *operator*) is **identified by a name** and described by two lists, one containing the **preconditions** of the action, which are those conditions that must hold in order to make the action executable, and the other one containing the **postconditions** of the action, which are those condition that become true or false after the execution (i.e. the effects of the action on the state):
* in many cases, the ***postconditions*** are split into an **add list**, a list of predicates that become true after execution, and a **delete list**, a list of predicates that become false after the execution.
* A planner is said to be **complete** when it always finds a plan when at least one exists, while is said to be **correct** if the returned plan brings from the initial state to a goal state.
* Two different phases can be identified when trying to solve this kind of problem:
* the actual **planning phase**, which can be done either off-line or on-line and is both *non decomposable* (or rather, there can be interaction with goals)and *reversible* (or rather, the choices made during this phase are backtrackable);
* the **execution phase**, which consists in executing the plan and so it must be done on-line, that is generally *irreversible* (or rather, non backtrackable) and *non-deterministic* (or rather, there can be some cases in which the assumptions made off-line by the planner are not valid anymore, hence some undesired effects can appear when executing the plan).
* There exist several planning techniques that use different algorithms to solve the problem:
* some of them, as **Generative Planning**, works off-line and relies on some assumptions which are often unrealistic, such as the ***atomicity*** of actions, the fact that the ***initial state is a priori fully known*** and the idea that ***each action has a deterministic effect*** because the ***plan execution is the only cause of changing in the world***, while others as **Reactive Planning** works on-line;
* many techniques like **Planning as Search** consider planning as a *search activity*either into the **state space** or into the **plan space**, or even consider states as sets of prepositions and operators as deductive rules (**Deductive Planning**).

1. LINEAR AND DEDUCTIVE PLANNERS

* A **Linear Planner** formulates the planning problem as a **search in the state space** and uses classical search strategies to solve it:
* it can proceed either **forward** (the search starts from the initial state and goes on trying to find *supersets of the goal*) or **backward** (the search starts from the goal state and proceeds backward until it finds a *subset of the initial state*);
* in the second case, generally the mechanism of **goal regression** is used, so that each *goal is divided into different* *subgoals* that are satisfied progressively by different actions, whose *preconditions become new subgoals*.
* A **Deductive Planner**uses **logics for representing states**, **goals** and **actions**, then, once the domain has been formalized, the **plan is generated as a theorem proof**:
* **Situation Calculus** is a method that uses *First Order Logic* to describe states and clauses to describe actions. A **situation** is a world snapshot that describes the properties (*fluents*) that hold in a given state s – e.g.

*p(x1, …, xn, s)*

means that the predicate *p(x1, …, xn)* is true in the state *s* – while an **action** is a relation between pre and postconditions – e.g.

*p1(x) and p2(y)* *⇾ p3(z, do(a(x), s) and ¬p1(x, do(a(x), s)*

means that the action *action(x)* has *p1(x) and p2(y)* as its preconditions and *p3(z) and ¬p1(x)* as its postconditions, indeed those are true in the next state *s’ = do(a(x), s)* obtained by performing the action *a(x)* over the state *s*.

* **Green** uses situation calculus to **build a plan on logic resolution** but, though having a *high expressivity* and being useful to *describe complex problems*, situation calculus encounters a **limitation** known as **frame problem**, which is a known *knowledge representation problem* that makes the number of frame axioms grow enormously because there is the need to **explicitly list all fluents that either change or do not change after the state transition**:
* as the solution returned by Green’s planner is an instantiation of the variable s so that the predicate goal(x1, …, xn, s) is true, then we have

s = do(actionn(params, do(actionn-1(…, …(params, do(action1(params, s0))))))

hence it will be necessary, after each action, to *use a frame axiom for each condition in the new state*, be it changed or not, which become hard to do for big problems.

* Thanks to **Kowalsky formulation** the frame problem is solved as we rely on the predicates **holds(predicate, state)**, used to describe *which predicate is true in a given state*, **poss(state)**, which indicates that a *state is possible* (or rather, reachable), and **pact(action, state)**, which indicates that *a given action can be performed in the given state* (or rather, the preconditions of the action are verified in the state):
* each new possible state can be found using the formula

poss(s) and pact(a, s) ⇾ poss(do(a, s))

* The **Stanford Research Institute Problem Solver** (*STRIPS*) is a **specific language for the actions** that has an easier but more effective syntax that situation calculus:
* each state is represented by the list of fluents that are true in that state (there is no need to explicit the state *s* itself as one state at time is considered) and the goal is represented as a state that can use variables (non-ground terms) in its fluents – the initial state obviously cannot do that;
* *STRIPS* is based on the **closed-world assumption**, which is the assumption where **everything not explicitly stated is considered false** (and not unknown), hence there will be **no frame problem** as we assume that everything not state remain unchanged;
* *STRIPS* relies on a **current state description** and on a **goal stack** where subgoals are piled so that, whenever a predicate is removed from the stack a new action (and its preconditions) are put in the stack if it is not currently satisfied in the state, while whenever an action is removed from the stack it is performed and changes the current state – this goes on until the goal stack is not empty and the plan is returned as the list of actions performed;
* whenever a series of actions bring to a failure, *STRIPS* backtracks (as long as there are other possibilities) and tries to find another path to reach the goal, still this can **lead to** **bad plans** (definitely non-optimal ones) and **enlarge the search space**, hence it could be useful to use some **heuristic strategy** to avoid the non-deterministic choice when ordering possible subsequent states.
* The **Sussman Anomaly** is frequently encountered when planning with a linear or deductive planner and it consists in the problem that shows up when a goal is made up of two (or more) subgoals in which one (or more) of them should not be done at first because its effects will destroy the necessary preconditions of the other subgoals:
* in fact, planners like *STRIPS* have no clues on which subgoal satisfy at first, hence when a goal *g* is so that *g = sg1* ∧ *sg2* it can decide to plan either for *sg1* and then for *sg2* or for *sg2* and then for *sg1* – at the end, however, the complete goal *g* will be always verified;
* the previous decision can bring to different solutions having different optimality, indeed if at the end of the planning the goal *g* is not verified because the actions done to satisfy *sg1* or *sg2* caused the removal of the other subgoal from the state, that subgoal must be reintroduced again and obtained with a new series of action, something that will eventually lead to a longer and less efficient plan.

1. NON-LINEAR PLANNERS

* A **Non-Linear Plan** is represented as a **set of actions**, a (non-exhaustive) **set of orderings** between those actions and a **set of causal links**, namely a relation between two actions *s*1 and *s*2 and a subgoal (*predicate*) *c* and written as **<*s*1, *s*2, *c*>** which means that the predicate *c*, precondition of *s*2, is obtained from the effects of *s*1:
* the **initial plan** of a non-linear planner is an **empty plan containing** **two fake actions**, namely *start* (which has no precondition and whose effects match the initial state) and *stop* (which has no effect and whose preconditions match the goal state), and an **order link between them** (*start < stop*);
* the **full plan** is then reached propagating back the actions from the goal to the initial state and adding orderings and causal links where needed;
* the **full plan** contains a set of partially specified and partially ordered operators, therefore these partial ordering should be made linear thanks to a **linearization operation** which will eventually give the actual plan.
* We say that an action *a* is a **threat** for a certain causal link *<s1, s2, c>* if the postconditions of *a* include the removal of *c* and there is **no order link** preventing *a* to be executed between *s1* and *s2*:
* whenever a new action is inserted in the plan, it is compulsory to **check if that action threatens any causal link** and, if so, to **remove that threat**.
* The **Partial Order Planning** (**POP**) Algorithm is a non-linear planner that uses the **Modal Truth Criterion** (**MTC**), namely a construction process that guarantees planner completeness, to solve threats and find a correct plan. *MCT* provides **five plan refinement methods**, one for the open goal achievement and four for threat protection:
* **Establishment** is the first method, used for *open goal achievement*, which consists either in **inserting a new action in the plan**, **adding an ordering constraint** or **performing a variable assignment**;
* **Promotion** is one of the *threat protection method* that **adds an order link that imposes the causal link to be executed before the threatening action**, i.e. *s2* < *a*;
* **Demotion** is the opposite of promotion, hence it **adds an order link that imposes the causal link to be executed after the threatening action**, *a* < *s1;*
* **White Knight** is another *threat* *protection method* to be used neither promotion nor demotion are effective because their preconditions and postconditions are exclusive. As the action *a* must be compulsory placed between *s1* and s*2*, the *white knight* **inserts a new action** (*or uses one already in the plan*) after *a* so that the preconditions of *s2*, negated by *a*, are restored;
* **Separation** is the last *threat protection method* that inserts a constraint for certain non-instantiated variables to avoid their unification to certain values.
* The **Sussman Anomaly** can be **avoided using non-linear planners**, still the generated plans can be very inefficient, even if correct:
* one way to reduce the length of the plans is trying to apply *promotion* and *demotion* before *white knight*, as the latter inserts a new action and, therefore, elongates the plan;
* also, planning is **semi-decidable**, which means that if there is at least a correct plan, the planner always returns a solution, otherwise it can go on working infinitely without stopping;
* to improve performances in planning, it would be better to use **ad-hoc methods**, not general planners like *STRIPS* or *POP*.

1. HIERARCICAL PLANNERS

* A **Hierarchical Planner** is a search algorithm that manage the creation of **complex plans** at **different levels of abstractions**, by considering the simplest details after having found a solution for the most difficult ones:
* *hierarchical planner* can work by **assigning different** **values of criticality** (proportional to the complexity of its achievement) to the preconditions or by **dividing the** **operators** (*actions*) are into **atomic operators** and **macro operators**, the former consisting in simple actions while the latter consisting in complex actions generally made up of various simple ones – still, each operator is identified by a name and its lists of preconditions and effects;
* in both cases, criticality values and atomic/macro operators must be **provided by the user**, hence those could be seen as a sort of *heuristic values* that, if are not well tuned, can worsen the effectiveness and the efficiency of the planner;
* given a goal, the *hierarchical planner* performs a **meta-level search** that generates a **meta-level plan** which uses macro actions to go from a state which is near the initial one to a state which is near the goal;
* in general, *hierarchical planners* can use either ***STRIPS-Like*** or ***POP-Like*** algorithms to create the meta-level plan.
* **ABSTRIPS** (**Abstract STRIPS**) is a *hierarchical planner* which **uses criticality values** to explore the problem at **different levels of abstraction**:
* at each level *k*, a complete plan is generated **considering each precondition having criticality *c* < *k* to be true**, hence that plan will probably be **incorrect** as the smallest details get ignored;
* the process is **iterated from the highest level** (*n)* **to the lowest** (1), and at each level k **the plan generated at level *k+1* is used as a guide to order the possible actions** – the plan generated before in fact behaves as a *heuristic* for the current planning;
* at the end, a **complete and correct plan** will be generated at level 1.
* On the other hand, **hierarchical planner using atomic and macro operators** firstly create a plan using just the macro ones, then creates a full plan by **decomposing the macro operators** and adding constraints and atomic ones where needed:
* generally, a *macro operator* is composed by a set of ordered or partially ordered *atomic operators* (a sort of subplan specific for the macro action itself) hence, after the macro plan is computed, those macro operators must be **decomposed into the atomic ones**;
* when using *POP-Like* algorithms, the planner must also check if there is any threat after each decomposition to ensure that the plan is still valid;

1. OTHER PLANNERS

* **Conditional Planners** are search algorithms that **does not rely on the closed world assumption** and, therefore, **generates various alternative plans for each source of uncertainty:**
* a *conditional plan* is therefore constituted **causal actions**, **sensing** (*common sense*)**actions** for retrieving unknown information and **several alternative partial plan** which will be **chosen (and executed) on-line** depending on the results of the agent’s observations of the world;
* *conditional planner* can generate plans even when non-conditional ones will not because a certain action essential to reach the goal is not explicitly stated;
* however, those kinds of planners require **a lot of memory** to save each possible partial plan and also **a lot of time** to compute those partial plans due to the *explosion of the search tree*.
* **Probabilistic** and **Contingency Planners** are variations of *Conditional Planners* that do not compute and store each possible partial plan but just those who are most likely to be useful during the execution phase:
* the most famous planners of this kind are *Buridan* (probabilistic planner) and *Cassandra* (deterministic contingency planner).
* **Reactive Planners** (*Brooks, 1986*) are planners that just **work on-line** in order to **acquire unknown information from the world** and plan their responses as consequence:
* **pure reactive planners** **do not actually plan** as there is no constructed plan but just a series of local decisions (they are a sort of *greedy algorithms* that use heuristic evaluations to choose the next move by **reacting as triggers to the world’s variations**);
* **modern responsive planners** are **hybrid systems** which uses both a *generative* and a *reactive approach* by **interleaving the two phases** to create an off-line plan and use it as long as no unexpected event occurs, then they backtrack and create a new plan;

1. PLANNING BASED ON GRAPHS

* **Graphplan** is a new concept of planner based on graphs that **inserts the time dimension** during the plan construction process:
* *graphplan* is a **correct** and **complete** planner and it is very efficient -still, is uses the **closed-world assumption**, hence it falls into the category of **off-line planners**.
* In *graphplan*, **actions** are represented as in *STRIPS* and there is always a **no-op** action which can be applied to a given predicate and has this predicate as both its precondition and its postcondition:
* moreover, **objects have a type**.
* While planning, *graphplan* creates a graph called **planning graph** which consists on **many parallel plans stored in the same data structure** as it contains, at each level *k*, **all the possible states that can be reached after performing *k* actions**;
* the *planning graph* is a **direct leveled graph**, that is to say **nodes can belong to different levels** and **arcs connect only nodes in adjacent levels**;
* level 0 represents the initial state, then **action levels** and **proposition levels** are interleaved;
* an **action *a*** can be **inserted in a certain level *n*** if and only if **its preconditions are present in the proposition level *n-1***, then **its effects will be propagated into the next proposition level *n+1***;
* At each level, **interfering actions** and **predicates** can appear in action and proposition level respectively, hence each level can be not consistent at all:
* whenever this happens, some **constraints** must be placed inside the level to connect those pairs of actions that cannot be performed simultaneously or those pairs of predicates that cannot be true simultaneously (**mutual exclusion**);
* in particular, a constraint between two actions is put whenever those actions have either **inconsistent effects** (one of them negates the effects of the other one), **interference** (one of them deletes the preconditions of the other one) or **competing needs** (they have mutually exclusive preconditions);
* on the other hand, a constraint between two predicates is put whenever those predicates are **one the negation of the other** or if **all the ways to reach them are mutually exclusive** – there also can be **domain dependent inconsistencies**, e.g. an object cannot be in two places at the same time;
* After the *planning graph* has been built, a **valid plan** must be extracted:
* as *graphplan* returns a partially ordered plan, that plan is considered valid if there are **no interference among actions on the same time step**, **no mutually exclusive predicate on the same time step** and **the last time step contains every literal from the goal** (and those literals are not mutually exclusive, otherwise there is no valid plan);
* the valid plan is therefore a **subgraph of the planning graph** in which **inconsistencies are removed**;
* basically, the inconsistencies found by the algorithm **prune paths in the search tree**.
* *Graphplan* also uses **memoization** to avoid the recomputation of the same state if it has proven that this state will bring to a failure:
* *memoization* consists in saving the hash value of a state in a hash table and check if a certain state has been visited before by checking if its hash value is already present into the table.
* **Fast Forward** is an **extremely efficient** **heuristic planner** which uses a combination of *hill climbing* and *A\** (the so-called **enforced hill climbing**)to reach a solution:
* from the initial state *s* the ***hill climbing* algorithm is performed until no optima *s\** is found**, then a **complete *A\* search* is run from *s\** as initial state** using the same heuristic function;
* the **heuristic** used by *FF* measures the distance to the goal when considering a relaxed version *P+* of the problem P in which **delete effects of the actions are not considered**.

CONSTRAINTS

* A **Constraint Satisfaction Problem** (**CSP**) is defined over a **finite set of variables** {X1, …, Xn}, a **related set of** **domains** {D1, …, Dn} and a **series of constraints** *ci(Xi1, …, Xik)* between *k* variable which is a subset of the cartesian product *Di1 x … x Dik* and specifies which values of the variables are compatible with each other:
* a **solution to a CSP** is an **assignment of the variable** which **satisfies all the constraints**;
* a *CSP* can be solved using a **search strategy** starting from the empty assignment *{}* as the initial state and assigning at each level one variable until the goal state is found – the max depth will be *n*, the number of variables, hence the problem will have *dn* leaves said *d* the cardinality of the domains;
* when solving a *CSP* with a *search strategy*, we are **not interested in the path** but **just on the final state**.
* A *CSP* can be solved using a **search strategy** starting from the empty assignment *{}* as the initial state and assigning at each level one variable until the goal state is found:
* we are **not interested in the path** but **just on the final state**;
* the max depth will be *n*, the number of variables, hence the problem will have *dn* leaves said *d* the cardinality of the domains;
* The tree search of a *CSP* has **three degrees of freedom**:
* the **choice of the variable ordering** and the **choice of the ordering of values to be assigned to the current variables**, which are actually related to the **heuristics** used to solve *CSPs*;
* the **propagation technique** carried out in each node, which is related to the algorithm used;
* Given a *CSP*, indeed, there are **different approaches** to solve it:
* **A Posteriori** (*No Propagation*) **Algorithms** simply search over the state and then check if the solution is admissible;
* **Propagation Algorithms** try to eliminate some values *a priori* (still *during the search*, or rather after each assignment) to avoid generating inadmissible states;
* **Consistency Techniques** eliminate each inadmissible value *before the search*

1. A POSTERIORI ALGORITHMS

* *A Posteriori Algorithms* work as standard search strategies, hence they **firstly generate each possible state from a given one** and then **test those state** until they find an admissible solution (or a failure, if there is no solution);
* The **Generate and Test** method **creates the tree until its final level**, then it tests it and, if the state found happens to be not admissible, it comes back and tries another one:
* it basically **tries each permutation of the values** (*n!*) until it finds one that satisfies all the constraints, hence it is **terribly inefficient**.
* The **Standard Backtracking** method is slightly better that the previous one as, when assigning values to the variables **immediately stops when at least one constraint is not satisfied** without proceeding until the final level of the tree:
* differently from *generate and test*, *standard backtracking* checks the consistency of the solution **at each instantiation** and not just at the end;
* it is still an *a posteriori technique* and, therefore, very inefficient.

1. PROPAGATION ALGORITHMS

* While in *a posteriori algorithms* the *constraints are used backwards* to check the consistency of the solution, with **propagation algorithms** they are **used forward to avoid inadmissible solutions before generating them**.
* The **Forward Checking** (**FC**) method **propagates each constraint involving Xi** **and** **those variables not yet instantiated** after the assignment of a variable Xi:
* *forward checking* propagates **only** the constraints involving Xi;
* the space of the solutions is reduced, hence there will be few states to be explored which will, eventually, bring to a failure state;
* whenever **a domain becomes empty**, *forward checking* **fails, backtracks and tries another path**.
* The **Partial Lookahead** (**PLA**) method, once propagated each constraint involving the assigned variable Xi as in *forward check*, **for each not yet assigned variable Xh also** **propagates each constraint involving Xh and other not yet assigned variables in {Xh+1, …, Xn}**:
* *PLA* basically propagates constraints from a given variable Xh **future variables only**, namely those variables Xk where k > h, hence **constraints are propagated on one direction only**;
* The **Full Lookahead** (**FLA**) method, once propagated each constraint involving the assigned variable Xi as in *forward check*, **for each not yet assigned variable Xh also** **propagates each constraint involving Xh and other not yet assigned variables both in {Xh+1, …, Xn} and in {Xi+1, …, Xh-1}**:
* *FLA* propagates constraints **both on** **future variables and on previous, yet not instantiated, variables**, hence **constraints are propagated on both directions**;
* still, the order in which the constraints are propagated can reduce domains in a better or worse way.

1. CONSISTENCY TECHNIQUES

* While *propagation algorithms* start to work *after the first assignment*, **consistency** **techniques** are applied **before the first assignment:**
* once an assignment is done, then, it is possible to **apply the consistency technique again** on the new state and proceed the search as usual, or to use either *propagation* or *a posteriori algorithms*;
* *consistency techniques* are base on a formal representation of the problem as a **graph of constraints** where each variable is a node and each constraint is an arc.
* **Node Consistency** (*consistency of level 1*) only cares about **unary constraints** on each variable Xi:
* in order to **make a node *i* consistent**, each value v ∈ Di that violates the unary constraint on Xi must be removed from Di.
* **Arc Consistency** (*consistency of level 2*) is obtained when each node and each arc of the graph is consistent:
* in order to **make an *arc(I, j)* consistent**, each value v ∈ Di for which it does not exist at least one value w ∈ Dj such that the constraint between *i* and *j* is satisfied must be removed (and vice versa);
* a value w ∈ Dj such that the constraint between *i* and *j* is satisfied for a value v ∈ Di is called **support of v**;
* basically, arc consistency can be obtained iterating the *full lookahead* strategy until no other improvement can be done – hence, here the ordering between constraints has no effect on the solution;
* the algorithms that use *arc consistency* as a **propagation step** are called **Maintaining Arc Consistency** (**MAC**) as it performs *arc consistency* after each assignment, the most famous one is called **AC-3**.
* It is important to notice that neither *node consistency* nor *arc consistency* can guarantee that there is an actual solution:
* consider the case in which there are three nodes and the variables has three equal domains having two values only, but the constraints say that those variables must assume different values, then the graph is arc-consistent but no solution is admissible;
* **path consistency** (*consistency of level 3*) is obtained starting from an arc-consistent graph and imposing consistency between each path *(i, j, k*), in this case the previous example would not be path-consistent, hence the failure will be caught before starting the search;
* in general, **a CSP has a solution iff it is proved to be k-consistent**, however, computing k-consistency has an exponential complexity and, in many cases, is even less efficient than trying to find a solution with search strategies, therefore it is common not to use a consistency technique greater than *arc consistency*.

1. HEURISTICS FOR CSPs

* In *CSPs*, heuristics can act on those **two degrees of freedom** to try to ensure the achievement of a good solution in a reasonable time**;**
* Heuristics can be classified into:
* **Variable Selection Heuristics**, which are used to determine **what should be the next variable to be instantiated**, such as the **first-fail** (that chooses the variable having the smallest domain) or the **most-constrained principle** (that chooses the variable appearing in the largest number of constraints);
* **Value Selection Heuristics**, which are used to determine **what should be the value assigned to a certain variable**, such as the **least-constraining principle** (that chooses the value that are most likely to succeed).
* A further classification can be of **static heuristics**, which determines the order **before starting the search** and leave it unchanged, or **dynamic heuristics**, which determines the order **after each assignment**.

1. FINAL REMARKS ON CSPs

* A **constraint solver** is a tool used to solve CSPs**,** it encapsulates various *filtering algorithms* and typically use *arc consistency* at the beginning as well as during the propagation. Moreover, those *filtering algorithms* are not always general purpose but they can be domain dependent to exploit the semantics of the constraint for efficiency reasons, for example:
* having the constraint **X ≥ Y**, it is not necessary to find a support for each value in the domains DX and DY but just cut from Dx each value ***x < min(Y)*** and from Dy each value ***y > max(X)***;
* on the other hand, the constraint **X ≠ Y** will always be arc consistent if both Dx and Dy have more than two values, while the constraint **X = Y** will bring to Dx = Dy = Dx ∩ Dy without any further computation.
* Finally, algorithms used to solve *CSPs* can be used to solve ***COPs***(**Constraint Optimization Problem**), namely those problems in which it is not enough to get a solution but **the optimal one is needed**:
* this can be achieved by adding to the *CSP* a **global constraint** (*a constraint involving each variable*) that is updated every time a feasible solution is found, then the algorithm is reiterated but will avoid solutions that does not satisfy this constraint as, even if they will bring to a solution, that will for sure be non-optimal;
* in any case, *CSPs* are **NP-hard problems**, hence no algorithm that deterministically solves them in polynomial time has been found yet – and, actually, it probably will never be.