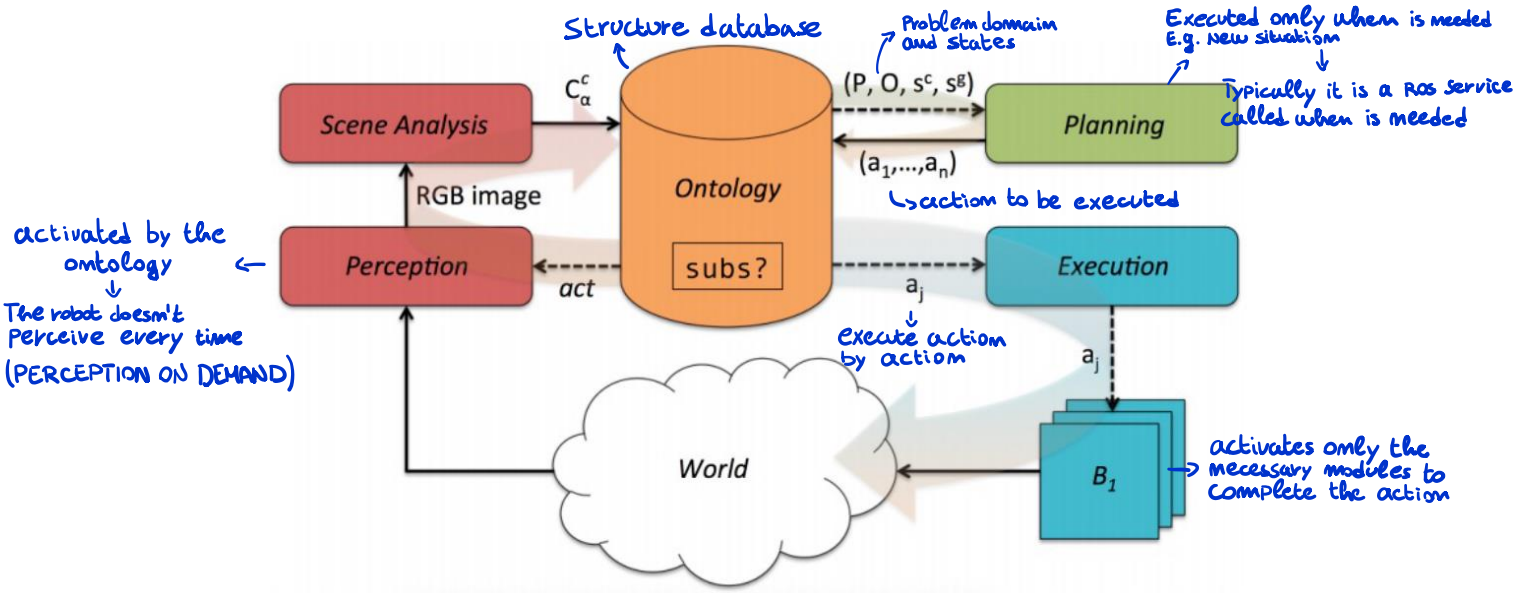
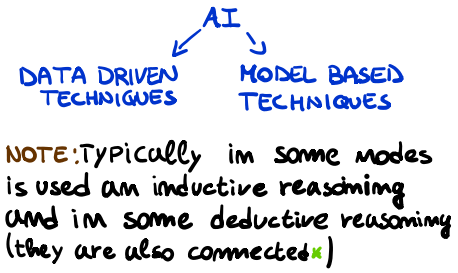
Artificial intelligence 2

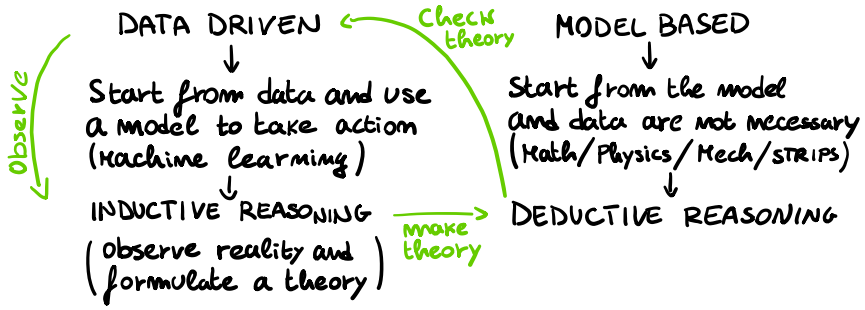
Deployment of AI models in software architectures

An AI model can be expressed with the following scheme:

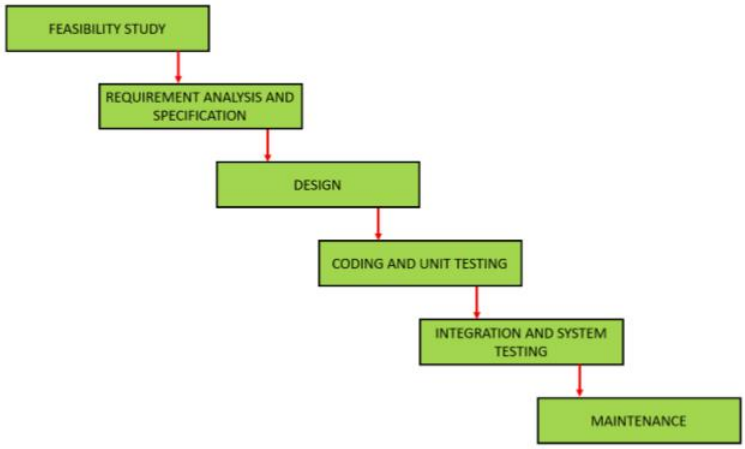


It is possible to distinguish two techniques used by AI:





Waterfall Model



- Basic approach for the development of AI models.
- The whole life cycle is divided into a set of phases; one phase can be started only after completing the previous one.
- The output of one phase is the input of the one that follows.
- Phases don't overlap with each other

OSS: since there isn't a feedback loop, the system is simple and fast to implement

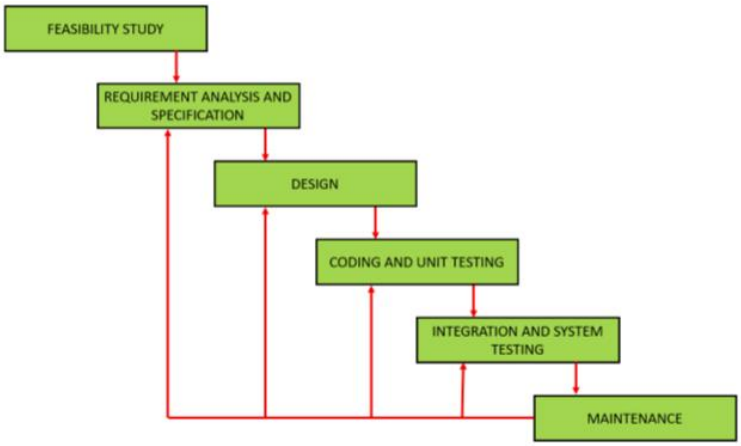
Advantages

Very simple and easy to understand, phases are carried out one at a time, each stage and milestone is clearly defined. Processes and results are well documented. Enforces good habits: DEFINE before DESIGN before CODE. Works for small projects, typically, where AI models are **deductive** (in that case the model must be true, then it's not necessary to go back).

Drawbacks

No feedback path and in-the-loop mechanism for model correction. Difficult to consider change requests.

Iterative Waterfall Model



Provides feedback paths from the last phase to its preceding phases. These enable the correction of misbehaviours by reworking only the necessary phase and spreading the corrections to the later phases.

Advantages

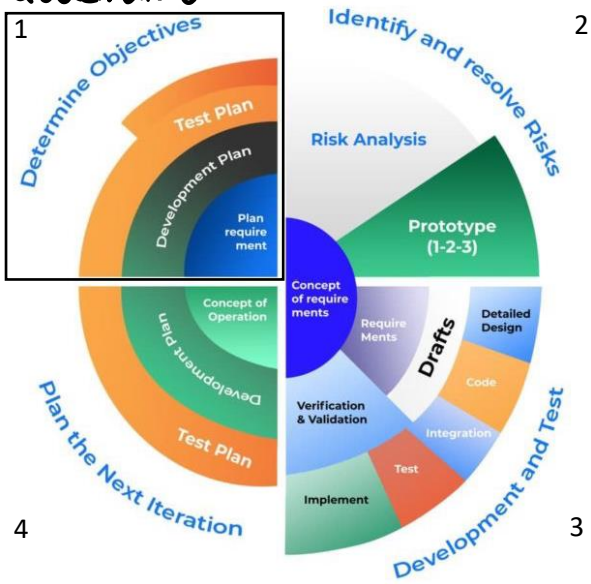
Introduction of a feedback path. Simple to understand and use.

Drawbacks

Difficult to consider change requests; incremental delivery is not barely supported; end-users cannot access intermediate delivery -> very limited interactions to collect user feedbacks.

Spiral model

Quadrant

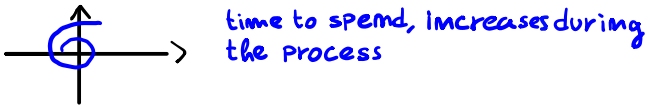


QUADRANT #1/ OBJECTIVES DETERMINATION AND IDENTIFICATION OF ALTERNATIVE SOLUTIONS:
1/ REQUIREMENTS ARE GATHERED FROM END-USERS.
2/ OBJECTIVES ARE IDENTIFIED, ELABORATED AND ANALYSED.
3/ ALTERNATIVE SOLUTIONS ARE PROPOSED UPON ENTERING THIS QUADRANT.

QUADRANT #2/ IDENTIFICATION AND RESOLUTION OF RISKS:
1/ ALL POSSIBLE SOLUTIONS ARE EVALUATED, AND THE BEST IS SELECTED.
2/ RISKS ASSOCIATED WITH THE BEST SOLUTION ARE RESOLVED.
3/ A MINIMUM VIABLE PROTOTYPE IS BUILT.

QUADRANT #3/ DEVELOPMENT OF THE NEXT PROTOTYPE ITERATION:
1/ PRIORITY FEATURES IN AI MODELS ARE DEVELOPED.
2/ APPROPRIATE TESTING OF AI MODELS IS CARRIED OUT.

QUADRANT #4/ REVIEW AND PLAN FOR THE NEXT PHASE:
1/ END-USERS EVALUATE THE SO-FAR DEVELOPED VERSION OF THE ARCHITECTURE.
2/ PLANNING FOR THE NEXT PHASE IS CARRIED OUT.



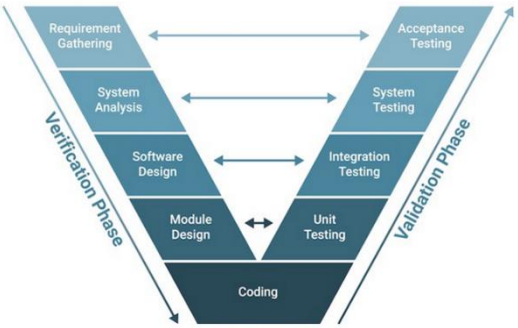
Advantages

Risk handling is managed in the loop. This model is good for projects with poor structure (don't know much about the problem), where machine learning is needed. Allows for flexibility in requirements and a prototype for the end-user is always available.

Drawbacks

Difficult to manage in practice. Estimation of time to completion and quality of results is very difficult.

V model



This model is also known as verification and validation approach. To each development stage there is associated the corresponding testing phase.

Advantages

Phases are completed one at a time. Project requirements are forced to be clear, simple, easy to understand and use. Focuses from the beginning on verification and validation; progress can be tracked easily.

Drawbacks

Can lead to high level of risks, not suitable for projects whose requirements are not clear and stable. The model doesn't support iteration of phases.

Ai architectural designed

It is the process of defining a collection of hardware, (AI-based) software components and their interfaces to establish the framework for the development of an AI-based system. Each architectural design consists of:

- A set of AI-based **components** (e.g.: planner, model, algorithm) performing the required functions
- A set of **connectors** helping in coordination, communication and cooperation among modules
- **Conditions** that the module must adhere to be integrated in the architecture
- **Semantic models** to help designers understand the overall system's properties

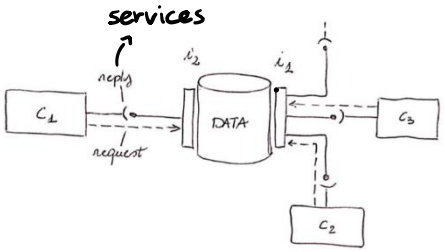
There are some different architectures:

DATA-CENTRED ARCHITECTURES/—>TYPICALLY FOR MODEL BASED

1/ A DATA STORE IS THE CORE OF THE ARCHITECTURE, IT IS ACCESSED FREQUENTLY BY OTHER COMPONENTS WHICH UPDATE, ADD, DELETE, MODIFY THE DATA PRESENT IN THE STORE.

2/ PROMOTES INTEGRABILITY. EXISTING COMPONENTS CAN BE CHANGED, AND NEW COMPONENTS CAN BE ADDED WITHOUT OTHER AI MODELS BEING NOTIFIED ABOUT IT.

3/ DATA CAN BE SHARED AMONG AI MODELS.



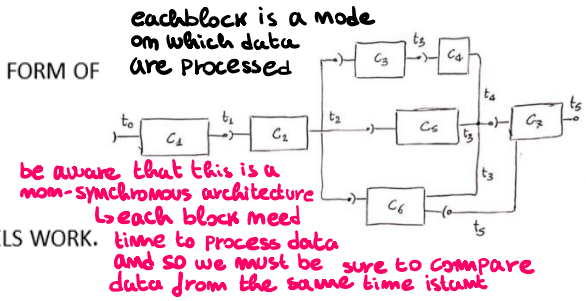
DATA-FLOW ARCHITECTURES/-> TYPICALLY FOR ML

1/ USED WHEN INPUT DATA ARE TO BE TRANSFORMED IN OUTPUT DATA USING SOME FORM OF COMPUTATION, FOR EXAMPLE A DEEP LEARNING MODEL.

2/ A SORT OF "PIPE-AND-FILTER" ARCHITECTURE.

3/ "PIPES" ARE USED TO TRANSMIT DATA FROM ONE AI MODEL TO THE NEXT.

4/ EACH AI MODEL WORKS INDEPENDENTLY, WITHOUT KNOWING HOW OTHER MODELS WORK.



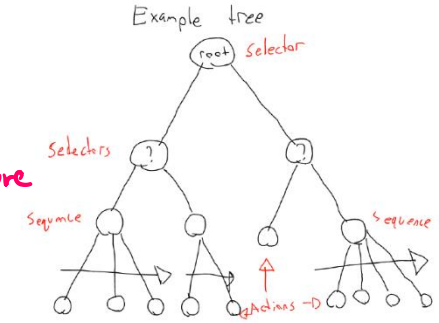
CALL-AND-RETURN ARCHITECTURES/—>TYPICALLY FOR LLM

1/ USED TO DESIGN AND BUILD AN AI-BASED ARCHITECTURE THAT MUST BE EASY TO MODIFY.

2/ TYPICALLY THEY CONSIST OF A MAIN PROGRAM AND ONE OR MORE SUB-MODELS, POSSIBLY DISTRIBUTED ACROSS MULTIPLE COMPUTERS.

3/ SUB-MODELS CAN RECURSIVELY INVOKE OTHER SUB-MODELS, FOR EXAMPLE AS IN RETRIEVAL AUGMENTED GENERATION (RAG) MODELS.

This is a symchronous architecture



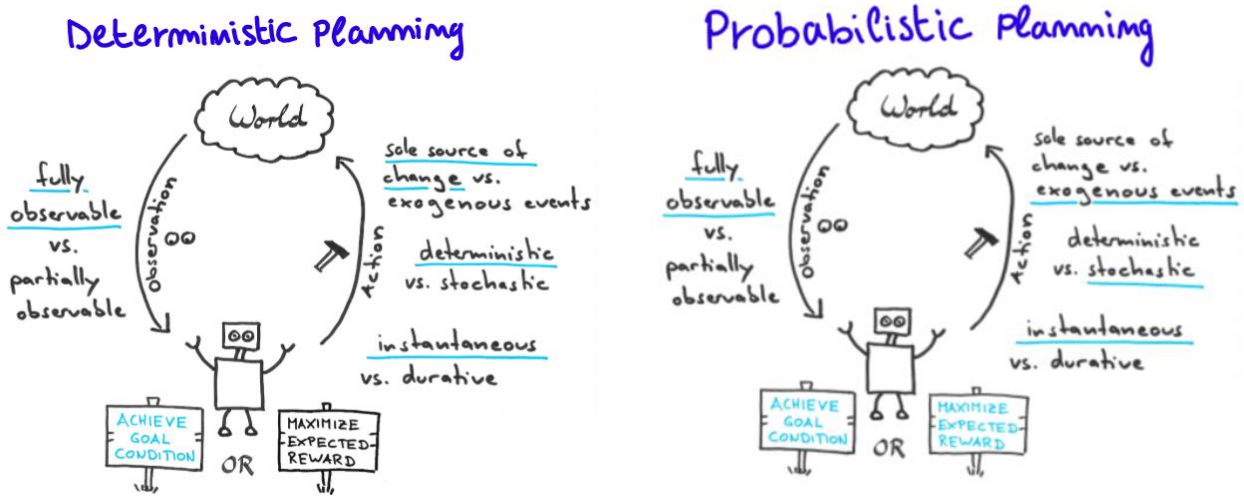
Automated Planning

There are two approaches for planning:

1. **Domain-specific**, use of specialised techniques, approaches or models that are adapted for the application domain (e.g. planning a manipulator)
2. **Domain independent**, quite generic representation and techniques that can be used in virtually any domain

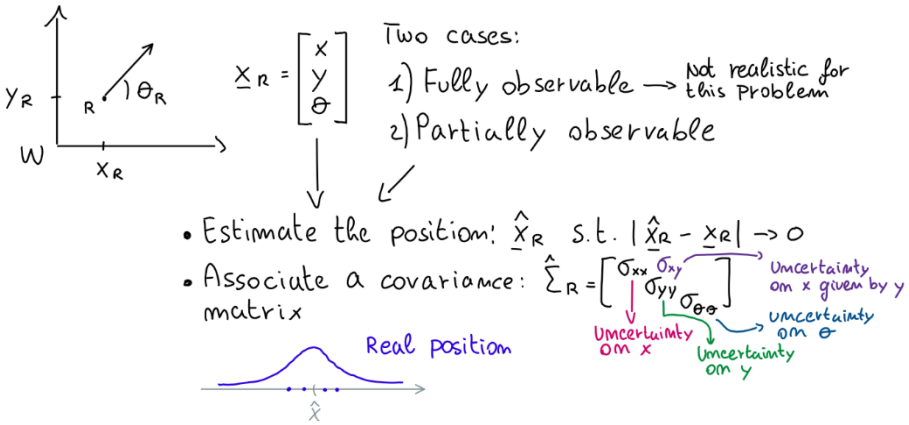
We will focus on domain-independent planning (this means that we are not constrained by the configuration)

Planning tasks

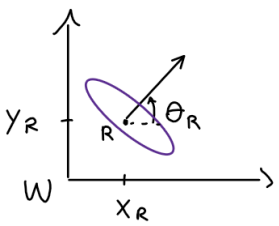


The planning task is the input information for a planning engine. It consists of: **initial state** of the world, **action** that can change the state and the **goal** to achieve. The output depends on whether we consider a classical of probabilistic planner; in the first case we can find a **solution plan**, that is a sequence of actions that allow to achieve the goal from the initial state. For the second, we can find a **policy,** that is a function that returns for each state the action to take.

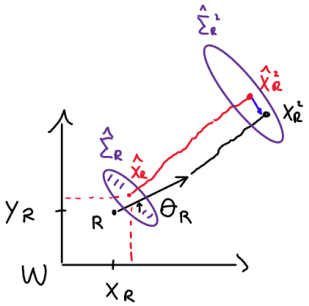
Detour: localisation using Kalman filter



The trace gives a measure of uncertainty



For the central limit, the sum of gaussians is gaussian.
A gaussian is represented as an ellipse in 2D



Supposing to remove sensors the error will increase

Activating the sensors, I can update the estimate moving to the real position then I can shrink the covariance* → 1) $\hat{x}_R \to x_R$
2) $\hat{\Sigma}_R$ more certain
↳ trace($\hat{\Sigma}_R$) smaller

In conclusion we have:
predicted motion and observation
Control action ($\underline{u}$)
↓
Predict
↓ $\hat{x}_R$, $\hat{\Sigma}_R$
Observe (update)
↓ $\hat{x}_R$, $\hat{\Sigma}_R$

oss: we are not using the real $\underline{x}_R$

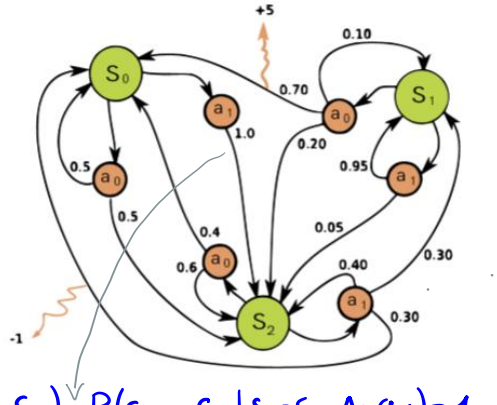
Markov decision process (MDP)

It is a discrete-time stochastic control process. It models decision making when outcomes are in part random and in part under control. The behaviour is:

1. At each time instant, the process is in some state **S**
2. A decision maker can choose an action **A** available in **S**
3. The process moves "randomly" (according to some probabilistic method) to a new state **S'** and provides a reward
4. The probability of moving from **S** to **S'** because of **A** is given by a **State Transition Function**

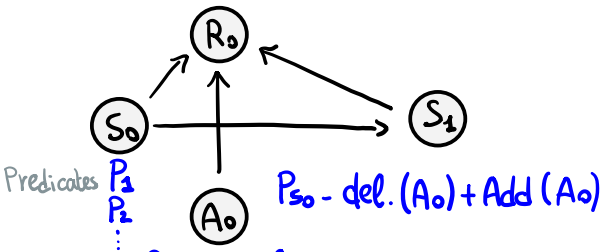
Formally, the Markov decision process is a 4-tuple (**S, A, P_A, R_A**) where:

- **S** is a set of states called **state space** (configuration space) can have discrete or continous states
- **A** is a set of actions called **action space**
- $\boldsymbol{P_A(S,S') = P(S_{t+1} = S' | S_t = S, A_t = A)}$, probability of moving to **S'** from state **S** according to action **A**
- $\boldsymbol{R_A(S,S')}$ is a reward received after moving from **S** to **S'** because of **A**



$P_{a_1}(S_0, S_2) = P(S_{t+1} = S_2 | S_t = S_0, A = a_1) = 1$

In planning (STRIPS):



The goal is to find the best **policy** $\boldsymbol{\Pi(S)}$ to guide action selection. $\boldsymbol{\Pi(S)}$, must maximise the cumulative function of random rewards.

This function is given by: $E[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})]$. Where gamma is a discounted factor, that decrease over time.

Planning Domain Definition Language (PDDL)

PDDL is a human-readable format for problems in automated planning that gives a description of the possible states of the world, a description of the set of possible actions, a specific initial state of the world, and a specific set of desired goals. The syntax is strongly influenced by LISP (an old programming language). Every formula is in prenex form and an example is: (AND (OR (ON P1 P2)(ON P2 P3))(OR (ON P2 P1)(ON P2 P3))).
The key ideas are that PDDL is focused on the notion of **predicates**, which are properties of the world that we do care about for the sake of the problem; these predicates act as "state variables". The truth value of predicates can by changed by applying actions.

EXAMPLE:

$$S_I = \begin{bmatrix} P_1 = T \\ P_2 = F \\ \vdots \\ P_n = T \end{bmatrix} \xrightarrow{A_0} S_1 = \begin{bmatrix} P_1 = T \\ P_2 = T \\ \vdots \\ P_n = T \end{bmatrix}$$

Precond
effects (P_2)

Predicates depend on the state we are in. Typically, in PDDL the rewards are given by the number of predicates that change, compatible with the goal state.

Oss: actually, the assumption of PDDL is that the states are defined only by the true predicates; so, only them will be present in the state. -> **Closed world assumption**

The inputs to the planner are:

- **A domain file**, including information about the domain and the application; encodes all of that using types, predicates and operators
- **A problem file**, which specifies the actual problem to solve within the domain and encodes that by describing the initial state, the goal state and the available objects

Domain file

It consists of the definition of the domain name, the specification of the requirements (E.g. ":ADL", ":TYPING"). The definition of types (similar to C classes), the definition of predicates and the definition of operators.

Requirements:

- **":adl",** is a super requirement which adds many other requirements such as: ":strips", ":typing", ":disjunctive-preconditions", ":equality", ":quantified-preconditions",":conditional-effects"
- **":disjunctive-preconditions",** allows the usage of OR in goals and preconditions
- **":equality",** allows the usage of = to compare objects. e.g. (= ?o1 ?o2)
- **":quantified-preconditions",** allows to express existential and universal preconditions
- **":conditional-effects",** allows to express conditional action effects, such that when something is true, then an additional effects holds. e.g. (when (and (has-chocolate ?p ?c)) (and (person-is-happy ?p))

Types:

Typing allows us to create basic and subtypes to which we can apply predicates. We use types to restrict what objects can form the parameters of an action. Types and subtypes allow us to declare both general and specific actions and predicates. E.g.

```
(:types
    site material - object
    bricks cables windows - material
)
```

We have a type *site* representing a building site and a type *material* which represents any generic building material. We then declare sub types *bricks, cables and windows* which are all types of building materials.

Predicates:

Predicates apply to a specific type of object, or to all objects. Predicates are either true or false at any point in a plan and when not declared are assumed to be false. E.g.

```
(:predicates
    (walls-built ?s - site)
    (on-site ?m - material ?s - site)
)
```

In this case, when the predicate *(walls-built ?s)* is true for a given site, then we can assume that said site has had walls built on it. When it is false, we would assume that the site does not have walls on it.

Remark: Formulas definition (recursive definition)

IN PRECONDITION:

./ <PREDICATE> IS A PRECONDITION;
./ (NOT <PREDICATE>) IS A PRECONDITION;
./ (AND <PRECONDITION> ... <PRECONDITION>) IS A PRECONDITION;
./ (OR <PRECONDITION> ... <PRECONDITION>) IS A PRECONDITION;
./ (FORALL (?X1 – TYPE1 ... ?XN - TYPEN) <PRECONDITION>) IS A PRECONDITION;
./ (EXISTS (?X1 – TYPE1 ... ?XN - TYPEN) <PRECONDITION>) IS A PRECONDITION.

IN EFFECTS:

./ <PREDICATE> IS AN EFFECT;
./ (NOT <PREDICATE>) IS AN EFFECT;
./ (AND <EFFECT> ... <EFFECT>) IS AN EFFECT;
./ (WHEN <FORMULA> ... <EFFECT>) IS AN EFFECT;
./ (FORALL (?X1 – TYPE1 ... ?XN - TYPEN) <EFFECT>) IS AN EFFECT;

Actions:

They consist of: the action name, the list of action parameters in the form (?x -type), the precondition list expressed as a formula, the effects list expressed as a formula. An action defines a transformation of the state of the world. This transformation is typically an action which could be performed in the execution of the plan. E.g.

```
(:action <action_name>
   :parameters (<argument_1> ... <argument_n>)
   :precondition (<logical_expression>)
   :effect (<logical_expression>)
)
```

```
(:action build-wall
   :parameters (?s - site ?b - bricks)
   :precondition (and (on-site ?b ?s) (not (walls-built ?s))
      (not (material-used ?b))
   )
   :effect (and (walls-built ?s) (material-used ?b)
   )
)
```

Example: domain definition

```
(DEFINE (DOMAIN GRIPPER)
(:REQUIREMENTS :ADL :TYPING)
(:TYPES ROOM BALL GRIPPER)
(:PREDICATES  (AT-ROBBY ?R1 – ROOM)
              (AT ?B – BALL ?R – ROOM)
              (FREE ?G – GRIPPER)
              (CARRY ?O – BALL ?G – GRIPPER))
```

Problem file

It consists of the definition of the problem name, the specification of the domain in which to solve the problem, the definition of the objects, the definition of the initial state (list of ground predicates initially true), the definition of the goal state (a formula).

Objects:

The objects block allows us to declare a set of objects which exist within our problem. each object name must be unique, and should be typed.

```
(:objects
   object_name_1 ... object_name_n - some_object_type
   ...
   object_name_2 ... object_name_m – another_object_type
)
```

```
(:objects
   s1 s2 s3 - site
   ba bb bc - bricks
   windowAlpha windowBeta windowCharlie - windows
)
```

Init:

The initial state (init) defines specifically what predicates are true at the start of the problem. This is not a logical expression because it is simply a list of predicates which are true. Unless the planner or domain specify otherwise all problems have the “closed world” assumption applied meaning anything not specified as true is considered false. Therefore, we only need to list things which are true.

```
(:init
   <predicate>
   ...
   <predicate>
)
```

```
(:init
   (on-site b s1)
   (on-site c s1)
   (on-site w s1)
)
```

Goal:

The goal is a logical expression of predicates which must be satisfied in order for a plan to be considered a solution. In essence it is what we want the world to look like at the end. Note that as a logical expression, if this expression excludes some predicate, then the value of that predicate is not considered important. This means that a goal should not only consist of the predicates that should be true, but also the predicate which should be false. Note that all standard logical operators such as *or* and *forall* are available as part of the goal, which means we can express multiple different goal states all of which are acceptable.

```
(:goal logical_expression)
```

Note: for the possible operators check the documentation

Example: problem definition

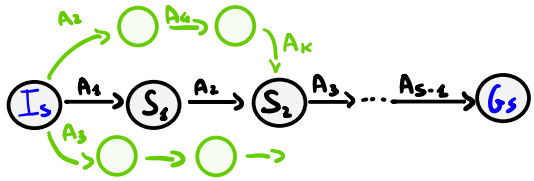
```
(DEFINE (PROBLEM GRIPPER-X-1)
(:DOMAIN GRIPPER)
(:OBJECTS      ROOMA ROOMB – ROOM
               BALL1 BALL2 – BALL
               LEFT – GRIPPER)
(:INIT  (AT-ROBBY ROOMA)
        (FREE LEFT)
        (AT BALL2 ROOMA)
        (AT BALL1 ROOMA))
(:GOAL (AND (AT BALL2 ROOMB) (AT BALL1 ROOMB)))
)
```

Planning algorithm implementation approaches

There are many approaches to implement a planning algorithm. These are: **Explicit search, SAT procedures, Hierarchical task networks.** The approaches can be of two types:

- **Satisficing:** any available plan is acceptable, and the cheaper ones are preferred
- **Optimal:** plan must be characterised by having the minimum possible cost

Explicit search



The objective is to find the best actions to take in order to reach the goal. (in green there are other solutions)

Oss: the planner will start from the goal state and then it will go back

The idea of these algorithms is to use a search space that is a subset of the state space, this induces a graph-like data structure. Nodes correspond to possible states, arcs correspond to transitions between states and paths correspond to plans.

Oss: problem (/search) solving is similar to planning, since both construct plans to achieve goals. The differences are internally and depends on: how they represent goal, states and actions, how planning component use an explicit, logical representation and how they search for planning.

Search-based problem solvers

For these, states are based on complete descriptions (each predicate is defined with its truth value) and data structures hold permutations of all the possible states. Actions are used to generate successor state descriptions, goals are state descriptions tested using heuristic functions. Finally, a plan is an unbroken actions sequence.
The key criteria for defining an algorithm encompass several crucial factors:

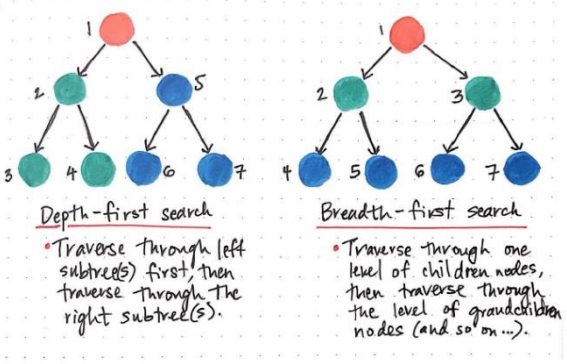
- **Completeness:** is concerned whether the strategy is guaranteed to find a solution when one exists
- **Time complexity:** defines how long does it take to find a solution (if it exists)
- **Space complexity:** is related to how much memory is needed to find a solution
- **Optimality**: is related to determining whether the strategy can find the highest quality solution

These criteria are linked to the specific problem at hand and the resources available for implementing the solution. Finally, is possible to defines:

- **Uninformed search ("Blind"):** the algorithm has no information about the current state, but it can determine if it is a goal or not, and therefore it must explore all search space. Examples: breadth-first, uniform cost search, depth-first, depth-limited search, iterative-deepening, bidirectional search, …

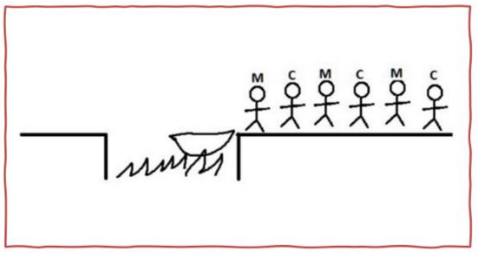
- **Informed search:** has some way for telling the "quality" of the current state, and for informing the selection of the next state to visit. Examples: greedy best first, A* (and variants, e.g., wA*, iAD*), hill climbing, gradient descent, simulated annealing, …

Depth-First search: particularly suited for scenarios where the solution lies deep within the search space and there's a need to swiftly uncover any viable solution, albeit not necessarily the most optimal one. So, the planning can be **non-optimal.**



Breadth-First search: it prioritizes the generation of levels in a systematic manner. This comprehensive approach ensures complete exploration of potential solutions, albeit at the expense of potentially elongating the search process. It finds an **optimal planning.**

Example: missionaries and cannibals

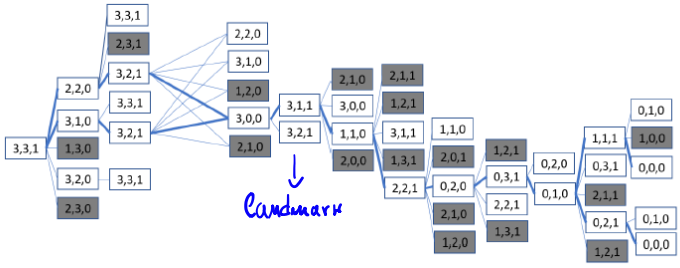


Three missionaries and three cannibals are on one side of a river, and there's a boat which can hold one or two people.

The goal is to define a plan to get everyone to the other riverside, without ever leaving a group of missionaries in one place outnumbered by cannibals in that place.

The solver is formalized in this way:

- States are modelled as ordered sequences of three numbers, e.g., (#m, #c, #b), where m is the number of missionaries, c the number of cannibals, and b indicates the presence of the boat on the initial riverbank. A start state may be S_0 = (3, 3, 1)
- Operators may be: "take 1 missionary", "take 1 cannibal", "take 2 missionaries", "take 2 cannibals", "take 1 missionary and 1 cannibal" across the river
- Goal test may imply that the goal state is S_g = (0, 0, 0)
- Cost guiding the solution may be the number of crossings (to be minimised)



Using Breadth-First search it's possible to create the graph on the left.

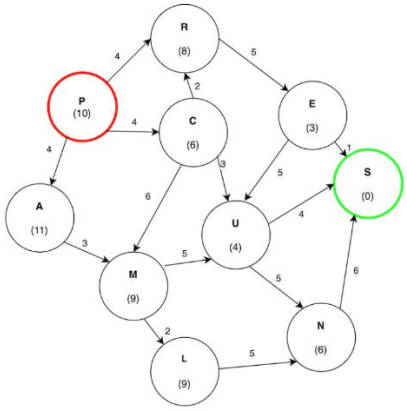
Remark: Heuristic function, it is numerical a tool that provides an approximation of the cost associated with reaching a desired goal or solution from a given state. In particular, the heuristic value is an estimation of the remaining cost for reaching the goal node "x" obtained via an "evaluation function" f(x). It is used for deciding which, among several alternative courses of action, may be the most effective to reach the goal state.

An important note is that if Markov assumption holds true, the heuristic function depends solely on the previous state making the history insignificant. This underlines the importance of **landmarks**, in fact supposing to drop the assumption, the history becomes irrelevant. However, upon reaching the landmark, the problem can be segmented, allowing us to disregard the past.

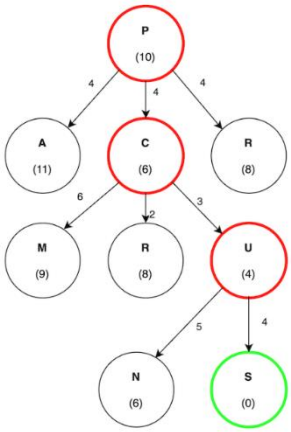
Greedy Best First search

It is an **informed search algorithm** that uses the evaluation of function F(x) has a heuristic guiding its exploration of the search space. Then follows this: always selects the node *x+1* that is **closest** to the goal node as determined by the heuristic function. In this way, it aims to consume the largest possible portion of the remaining distance towards the goal at each step, hence the name greedy.

Example: Consider finding the path from **P** to **S**



Using the Greedy Best First search the path found by the algorithm is the red one. However, there is a better path represented by **P -> R -> E -> S** which offers a lower cost. This highlights the fact that the optimal solution is not always guaranteed.



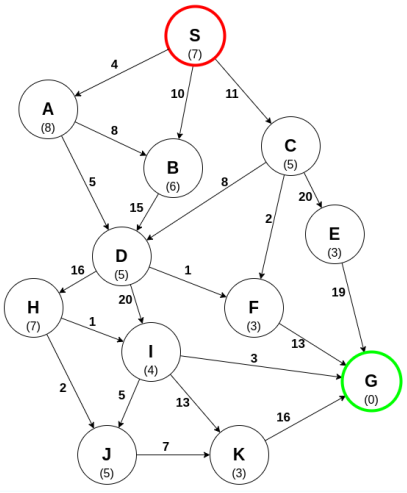
A* search

It is a variant of the Best First search where the evaluation function $F(x)$ is given by the sum of two contributions:

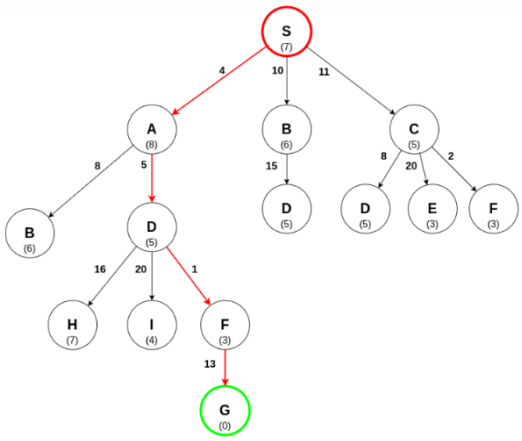
- The **heuristic function** $H(x)$, which estimates the cost of reaching the goal node from node *x.*
- The **cost function** $G(x)$, which estimates the cost of reaching node x, from the starting node.

So, $F(x)$ represents the estimated total cost of the cheapest path from the start node to the goal node passing through node *x*.

Example: Consider finding the path from **S** to **G**



Using the A* search the path found by the algorithm is the red one.



More on heuristics

We have a class of problems P, whose perfect solution cost is H*P and we want to estimate it. Sometimes can happen that the problem is too difficult to find the correct solution. In that case, is possible to use a class P' of simpler problems, whose solution cost is cheaper to compute and can be used to estimate H*p. This transformation is also called "relaxation mapping" R and allows to map instances of P into instance of P'. There are three different classes of heurstics:

- **Delete relaxation:** typically used for satisficing planning, the idea is to ignore the negative effects of planning actions and solve the corresponding relaced problem
- **Abstractions:** drop some distinctions between states, preserving the transition behaviour (reduce #states)
- **Landmarks:** an operator that must be applied or a ground formula to be true (trajectory constraint)

PDDL+

Basic planning is used only for discrete problems, with the following advanced planning concepts is possible to reason about physical quantities, using time-durative processes and modelling exogenous processes. To the classical PDDL, this version add:

- **Processes**: directly correspond to a durative action and last for as long as their pre-condition is met. A process is something like gravity's effect on a ball, increases the velocity of the ball until it either reaches terminal velocity or indeed, it hits the ground.
  Note: more than one process can be active at any time instant
- **Events**: directly correspond to instantaneous actions, and happen the instant their preconditions are met, usually with the effect of transforming their state such that their precondition is no longer met. Events are uncontrollable, and in our ball example, we might consider an event to be the ball hitting the ground. In that instance, the velocity of the ball is negated, and multiplied by some bounce coefficient.
  Note: effects must be discrete, active as soon as preconditions are met. Effects must delete their own preconditions, to avoid multiple executions of the event.

So, this type of planning is hybrid, meaning that there are actions and events that cause discrete changes to the world state and there are processes that instead produce continuous update in the world state, every time instant. Connected to this, is important to highlight that there are predicates, variables that can be only true or false and fluents, variables that change their value numerically.

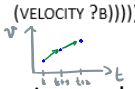
Example: Processes and events

```
(:ACTION RELEASE
:PARAMETERS (?B – BALL)
:PRECONDITION (AND (HOLDING ?B) (= (VELOCITY ?B) 0))
:EFFECT (AND (NOT (HOLDING ?B))))

(:PROCESS FALL
:PARAMETERS (?B – BALL)
:PRECONDITION (AND (NOT (HOLDING ?B)) (>= (HEIGHT ?B) 0))
:EFFECT (AND (INCREASE (VELOCITY ?B) (* #T (GRAVITY))) (DECREASE (HEIGHT ?B) (* #T
(VELOCITY ?B)))))
```

increase the variable every time step

integration time multiplied for the gravity



```
(:EVENT BOUNCE
:PARAMETERS (?B – BALL)
:PRECONDITION (AND (>= (VELOCITY ?B) 0) (<= (HEIGHT ?B) 0))
:EFFECT (AND (ASSIGN (HEIGHT ?B) (* -1 (HEIGHT ?B))) (ASSIGN (VELOCITY ?B) (* -1 (VELOCITY
?B)))))

(:ACTION CATCH
:PARAMETERS (?B – BALL)
:PRECONDITION (AND (>= (HEIGHT ?B) 4.9) (<= (HEIGHT ?B) 5.1))
:EFFECT (AND (HOLDING ?B) (ASSIGN (VELOCITY ?B) 0)))
```

In conclusion, the main modelling focus is usually on processes; they drive changes in world states. Actions are under the control of the agent, which can use them to operate on processes. Events are used to define "limits" and automatically start or stop processes.

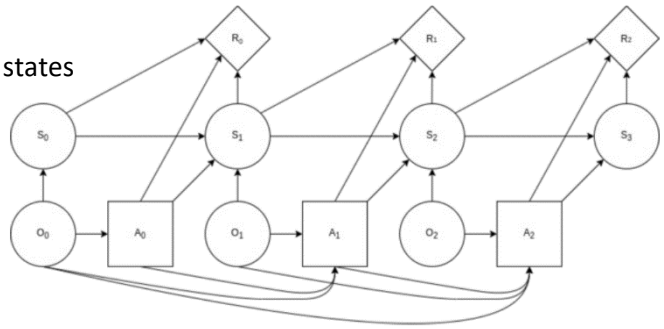
Recall: MDP is a discrete-time stochastic control process. It models decision making when outcomes are in part random and in part under control. The assumption is that state **S** is known when the decision on **A** is take. If it is not true we have to use Partially Observable MDP (POMDP).

Partially Observable Markov Decision Process

The underlying process "dynamics" is like in MDP but, in this case, the agent cannot observe directly the underlying, real state **S**. The agent must maintain a probability distribution over the set of possible states using observations, their probabilities, and the MDP.

The process is a 7-tuple (**S**, **A**, **T**, **R**, **Ω** , **O**, γ) such that:

- **S** is a set of states called the state space
- **A** is a set of actions called the action space
- **T** is a set of conditional transition probabilities between states
- **R**: **S** x **A** → **R** is the reward function
- **Ω** is a set of possible observations
- **O** is a set of conditional observation probabilities
- γ between 0 and 1 is the "weighting" factor



The behaviour is simple: at each time instant, the process is in some, not directly observable, state **S**. An agent can choose an action **A** available in **S**, whereas the choice is make based on an estimate of **S**. The process moves randomly to a new, not directly observable, state **S'**, with a probability **T**(**S'** | **S**, **A**). The agent receives an observation **O** of **S'** depending on both **A** and **S',** modelled as a probability **O**(**o** | **S'**, **A**), and uses it to build an estimate of **S'**. The agent receives a reward **R** such that **R**(**S**, **A**).

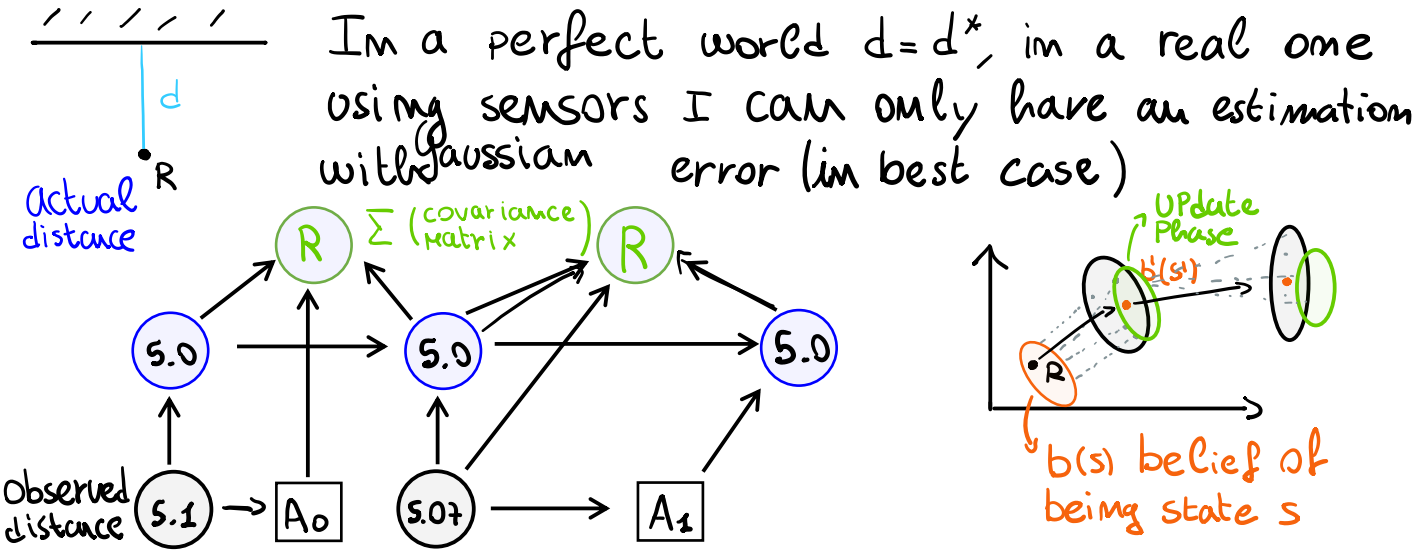
The goal is to find the best policy $\Pi(\boldsymbol{S})$ to guide action selection, that is, which action **A** should the agent choose in state **S** on the basis of its estimates. Moreover, $\Pi(\boldsymbol{S})$ must maximise the cumulative function of rewards, that is, the expected weighted sum over a potentially infinite horizon.

The important remarks are that the agent must make decisions under uncertainty about the true state. By interacting with the environment and collecting observations, the agent can update the probability distribution of the current state. So, the optimal behaviour may include information gathering actions.

The probabilistic equation associated to this process is:

$$b'(s') = \eta O(o\,|s', a) \sum_{s \in S} T(s'|s, a) b(s)$$

Example: POMPD for Kalman filter



Im a perfect world $d = d^*$, im a real one using sensors I can only have an estimation with gaussian error (in best case)

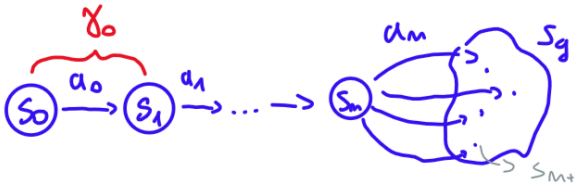
Im this case the belief in the state is given by the sum of the possible trajectory: $\sum_{s \in S} T(s'|s, a)$ where a is the control action and s' all the possible states. The observation is given by the localization using beacons

TASK-MOTION PLANNING

Def: Task domain can be represented via state transition system and is a tuple $\Sigma = (S,\ A, \gamma,\ s_0, S_g)$ where S is a finite set of states, A is a finite set of actions, $\gamma: S \times A \rightarrow S$ is the state transition function such that $s' = \gamma(s, a)$. s_0 is the start state and S_g is the set of goal state.

Oss: the state transition function describes all the possible transitions between states using actions

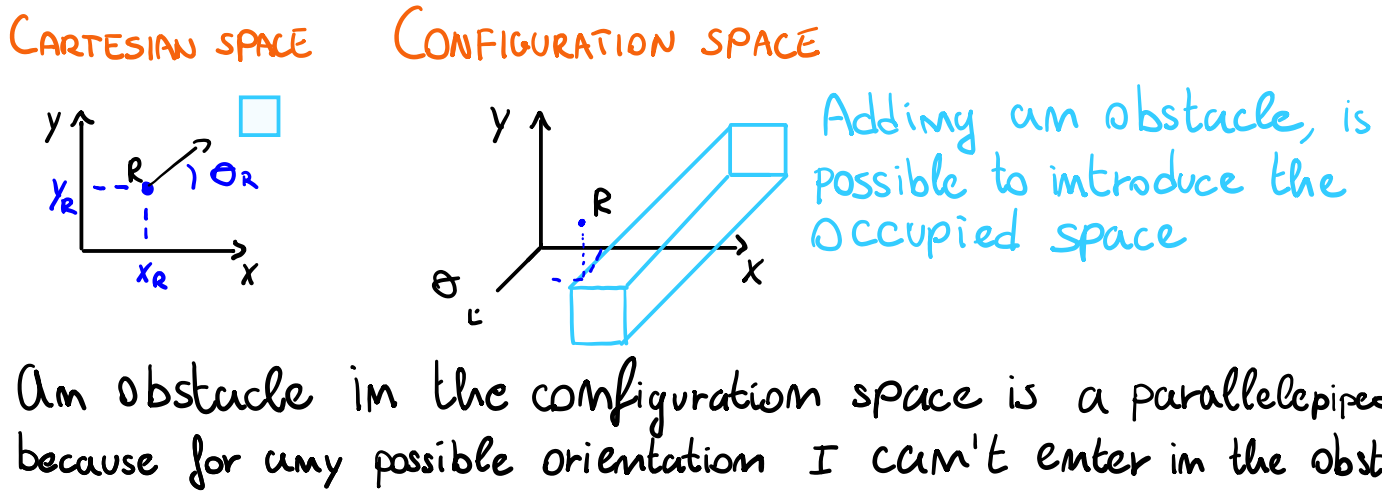
Def: Task plan for a task domain Σ is the sequence of actions $a_0, \ldots, a_n$ such that $s_{i+1} = \gamma(s_i, a_i)$, for i = 0, …, n and s_{n+1} satisfies S_g



Def: Motion planning problem is a tuple $M = (C, f, q_0, G)$ where: C is the configuration space, $f = \{1, 0\}$ determines if a configuration space is collision free (C_{free} with $f = 1$) or not; q_0 is the initial configuration and G is the set of goal configurations

Configuration space

Suppose to have a robot and the state is described by its posture, $S=(x_R, y_R, \theta_R)$. The configuration space is the space where a configuration is described as a point



An obstacle in the configuration space is a parallelepiped because for any possible orientation I can't enter in the obstacle

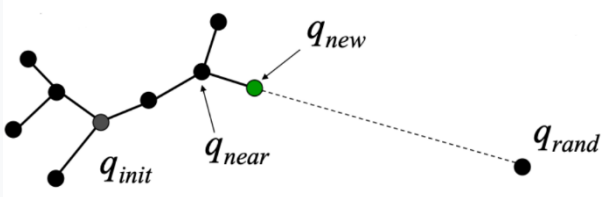
A **motion plan** essentially involves finding a valid trajectory in C from q_0 to $q_n \in G$ such that f evaluates to true for $q_0, \dots, q_n$. A motion plan can also be defines as $\tau = [0,1] \rightarrow C_{free}$ such that $\tau(0) = q_0$ and $\tau(1) \in G$. We will use a combination of the two to define the TMP problem and use a Rapidly-exploring Random Trees (RRT) based sampling strategy to generate collision free configurations

Rapidly-exploring Random Trees

It is an algorithm used in non-convex, possibly high-dimensional, search spaces (like a robot configuration space). It grows a tree from the start configuration using random samples from the search space. When a sample is drawn, a link is attempted between it and the nearest state already expanded by the tree; if the link is feasible, then the state is added to the tree. The growth process can be biased by injecting proper information.

```
Algorithm BuildRRT
    Input: Initial configuration q_init, number of vertices in RRT K, incremental distance Δq)
    Output: RRT graph G

    G.init(q_init)
    for k = 1 to K do
        q_rand ← RAND_CONF()
        q_near ← NEAREST_VERTEX(q_rand, G)
        q_new ← NEW_CONF(q_near, q_rand, Δq)
        G.add_vertex(q_new)
        G.add_edge(q_near, q_new)
    return G
```



Note: the algorithm finds a solution as soon as it finds it, then it refines

Def: Task-Motion Planning Problem is a tuple (C, M, Φ, ξ) where:

- $\Phi: S \rightarrow 2^C$, mapping states to the configuration space
- $\xi: A \rightarrow 2^C$, mapping actions to motion plans

The TMP problem is to find a sequence of actions $a_0, \dots, a_n$ such that $s_{i+1} = \gamma(s_i, a_i)$ and $s_{n+1} \in S_g$ and to find a sequence of motion plans $\tau_0, \dots, \tau_n$ such that for $i = 0, \dots, n$ it holds that:

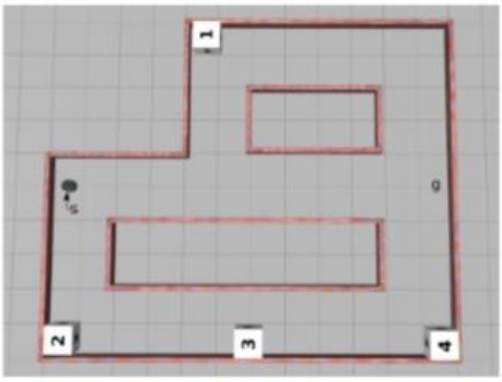
$$\tau_i(0) \in \phi(s_i) \text{ and } \tau_i(1) \in \phi(s_{i+1}) \quad (1)$$

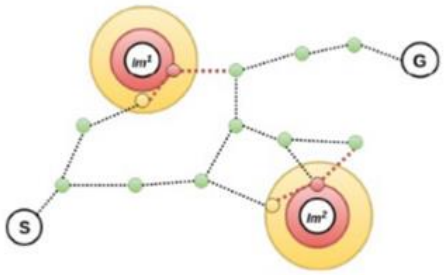
$$\tau_{i+1}(0) = \tau_i(1) \quad (2)$$

$$\tau_i \in \xi(a_i) \quad (3)$$

Example:

The goal is to move from S to G as much precisely as possible. To do so the cumulative estimate of covariance (i.e., the trace of the covariance matrix) must be as low as possible, this can be done by observing as much landmarks as possible, for as much time as possible.





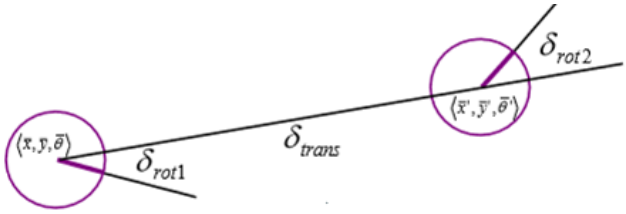
$$c_\Sigma = trace(\Sigma)$$

The odometry model is given by:

In this paper, we consider the TMP for navigation of a mobile robot operating in a pre-mapped environment. At any time k, we denote the robot pose (or configuration q_k) by $x_k \doteq (x, y, \theta)$, the measurement acquired is denoted by z_k and the control action applied is denoted as u_k. The robot kinematics is modeled using the standard odometry based motion model

$$\begin{aligned} x' &= x + \delta_{trans} \cdot \cos(\theta + \delta_{rot1}) \\ y' &= y + \delta_{trans} \cdot \sin(\theta + \delta_{rot1}) \\ \theta' &= \theta + \delta_{rot1} + \delta_{rot2} \end{aligned} \tag{4}$$

where $x_{k+1} \doteq (x', y', \theta')$ and $u_k \doteq (\delta_{rot1}, \delta_{trans}, \delta_{rot2})$ is the control applied (motion plan τ_k). For brevity we write (4) as $x_{k+1} = f(x_k, u_k) + w_k \sim \mathcal{N}(0, R_k)$, where w_k is the zero-mean Gaussian noise.



While, the measurement equation:

To process the landmarks in the environment we measure the range and the bearing of each landmark relative to the robot's local coordinate frame, which can be specifically written as

$$z_k = \begin{bmatrix} r \\ \phi \end{bmatrix} + v_k \ , \ v_k \sim \mathcal{N}(0, Q_k) \tag{5}$$

where r, ϕ are the range and bearing respectively and v_k the zero-mean Gaussian noise. For brevity, (5) will be written as $z_k = h(x_k, lm_i) + v_k \sim \mathcal{N}(0, Q_k)$. It is to be noted that we assume data association as solved and hence given a measurement we know the corresponding landmark that generated it. It is possible to relax this assumption to incorporate reasoning regarding data association within the belief space, as shown recently in [20].

Given the current belief $b[X_k]$, the control u_k, the propagated belief parameters can be computed using the standard Extended Kalman Filter (EKF) prediction as

Pose estimate ← $\begin{aligned} \bar{\mu}_{k+1} &= f(\mu_k, u_k) \\ \bar{\Sigma}_{k+1} &= F_k \Sigma_k F_k^T + R_k \end{aligned}$ (8)

where F_k is the Jacobian of $f(\cdot)$ with respect to x_k. Upon receiving a measurement z_k, the posterior belief $b[X_{k+1}]$ is computed using the EKF update equations

Kalman gain ← $K_k = \bar{\Sigma}_{k+1} H_k^T (H_k \bar{\Sigma}_{k+1} H_k^T + Q_k)^{-1}$ (measurement)

$$\begin{aligned} \mu_{k+1} &= \bar{\mu}_{k+1} + K_k(z_{k+1} - h(\bar{\mu}_{k+1}, l_i)) \\ \Sigma_{k+1} &= (I - K_k H_k)\bar{\Sigma}_{k+1} \end{aligned} \tag{9}$$

↳ Observation depends on measurements and landmarks position

where H_k is the Jacobian of $h(\cdot)$ with respect to x, K_k is the Kalman gain and $\mathbb{I} \in \mathbb{R}^{3\times3}$.

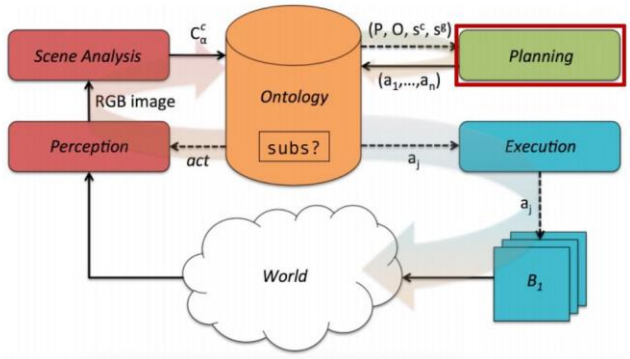
If the probabilities are gaussian $\Sigma_{k+1} = \underbrace{F_k \Sigma_k F_k^T}_{>0} + \underbrace{R_k}_{>0}$ ← noise

When the robot moves the covariance matrice increases

To design and implement all of this, is necessary to add numerical computations to PDDL. Is possible to connect a **semantic attachment** to boolean fluents. These external algorithms, process data and give as output fluents-related data.

Knowledge representation

The rationale is that in the Sense - Plan – Act architecture, there is the necessity to describe states, actions and goals. Moreover, the robot should be able to generate the predicates using perception and generate a correct plan.



CURRENT STATE GOAL STATE

A plan, as formally described in (1), is an ordered sequence of N actions whose execution changes the current state from s^c to s^g through a set of intermediate states. In a plan, each action may correspond to one or more scripted robot behaviours. For example, rotating a link l_{j+1} may require the robot to (i) keep the upstream link l_j steady with its left gripper, and (ii) rotate l_{j+1} of a certain amount with the right gripper. Such sequence shall not be encoded in the planning process, thereby reducing planning cost, but demanded to an action execution module.

If a plan is found, each action is encoded in the *Ontology*, along with all the *expected* intermediate states $s^c = s_1^e, s_2^e, \ldots, s^g = s_{N+1}^e$, which result from actions. The *Execution* module executes action by action activating the proper modules in the architecture, e.g., such *behaviours* as motion planning, motion execution, obstacle avoidance or grasping.

Each action a_j in a plan is assumed to transform a state s_j^e into a state s_{j+1}^e, such that:

$$s_{j+1}^e = \left(s_j^e \setminus eff^-(a_j)\right) \cup eff^+(a_j). \quad (6)$$

$$\mathcal{A}_j = \left(pre(\mathcal{A}_j), eff^-(\mathcal{A}_j), eff^+(\mathcal{A}_j)\right), \quad (4)$$

$$\mathcal{A}_j = \left(pre(\mathcal{A}_j), eff^-(\mathcal{A}_j), eff^+(\mathcal{A}_j), pre_a(\mathcal{A}_j), eff_a^-(\mathcal{A}_j), eff_a^+(\mathcal{A}_j)\right), \quad (5)$$

CONDITIONAL PRECONDITIONS AND EFFECTS

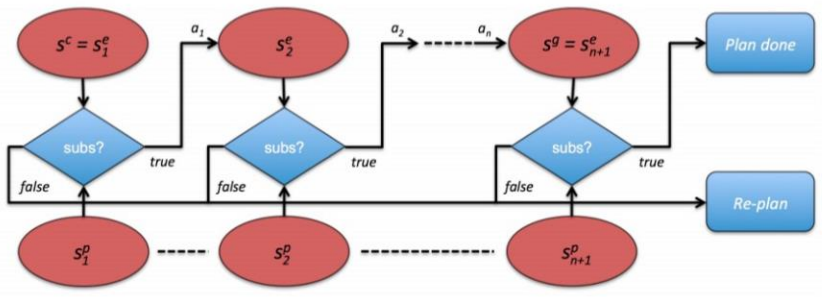
If a_j has additional conditions, then (6) is modified as:

$$s_{j+1}^e = \left(s_j^e \setminus \left(eff^-(a_j) \cup C^-(pre_a(a_j))\right)\right) \cup \left(eff^+(a_j) \cup C^+(pre_a(a_j))\right), \quad (7)$$

where conditions C^- and C^+ return the sets $eff_a^-(a_j)$ and $eff_a^+(a_j)$, respectively, if the conditions in $pre_a(a_j)$ hold, and $\emptyset$ otherwise.

Before an action is executed, the *ontology* module activates *perception* to acquire a new image. This allows to retrieve the current configuration. Every time this happens there are two possible situations:

- If the current configuration is subsumed by the description of the current state, then the execution continues with the planned action until the goal state is reached
- Otherwise, a new planning process occurs, considering the current state as new initial state and keeping the previous goal state

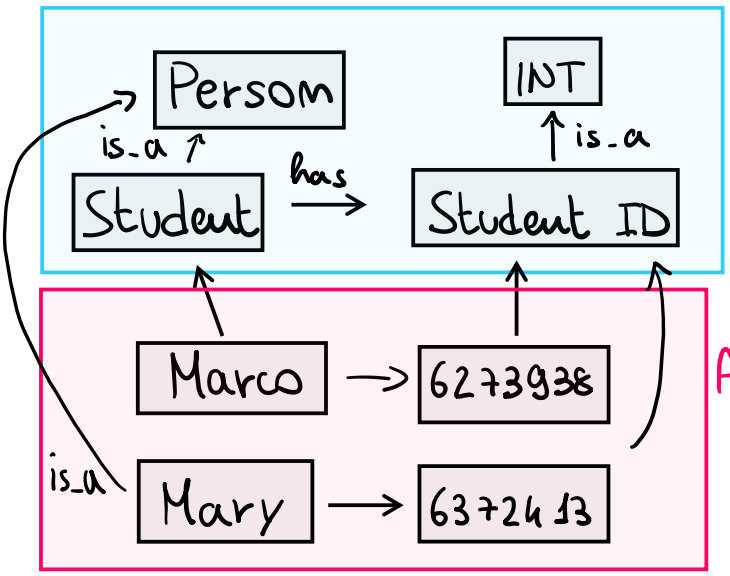


The ontology can be considered as a database, which contains the representation of the world. In it are encoded the objects and the relations between them. There are two boxes: the t**erminology box**, which contains a set of logical

concepts and **relationships**, and the **assertional box**, which refers to real objects. These allow to create a **Knowledge graph.**

Example:

Student is a person and has a student ID.
In this case, student and person are concepts,
Student ID is a INT and has is a relationship



Terminology box

To populate the knowledge domain I can say Marco is a student and has 6273938

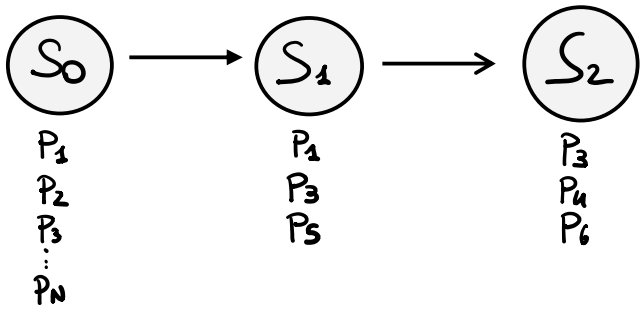
Assertional box

After having create the knowledge domain is possible to ask queries, for example:
Q1: Is Marco a person? In this case the answer is yes if there exists a path from Marco to person (Bottom-up)

Now suppose to add: Mary is a person and has 6372413. Then, the query is:
Q2: Is Mary a student? In this case the reasoning is done from up to bottom, because there is not an explicit relation between Mary and the concept of student.

Finally, the last important part regards the concept of **subsumption**. Given two formulas:
Formula 1: Fulvio is a student and has_id 2907584
Formula 2: has_id 2907584
I can ask if Subsume (F2, F1)? *Meaning that F2 is more general than F1*
The answer is yes, because I can write $F_1: A \wedge B$ and $F_2: B$

In general, I can use the subsumption to perceive the current state and decide which actions to perform.



The current state is Sc (P_1, P_2). So, I can write Sc is_a state $\wedge P_1 \wedge P_2$. Having already generated the knowledge graph I can say:

=SUBS (Sc, So) → YES, Exec (A_{0-1})
↳ NO, replan