se_14_ai_notebook

May 12, 2023

1 SE 14 Artificial Intelligence Basics

1.1 Introduction

1.1.1 What is an artificial Intelligence?

Artificial Intelligence (AI) is a complex and abstract concept that is often challenging to define. The issue lies in the inherent difficulty of defining "intelligence," which is not a direct measure but an abstract concept often linked to specific types of behavior such as problem-solving, applying knowledge creatively, and acquiring new knowledge. Intelligence tests, like IQ tests, are approximations of this concept, scoring an individual based on their problem-solving abilities.

Artificial Intelligence, as suggested by the term itself, aims to create an artificial entity that behaves in a manner we would expect from an intelligent being. There are several definitions of AI in academic literature, each focusing on different aspects. One common way to define AI, as categorized in the [RusselNorvig] reference, is with a parameterized statement: "An AI is an artificial system that [thinks / acts] [rationally / like a human]."

This statement has two dimensions. The first dimension, "thinks vs. acts," focuses on whether the AI system's actions or thought processes are the primary concern. The second dimension, "rationally vs. like a human," determines whether the AI's goal is to behave rationally or to mimic human behavior, including our less rational aspects influenced by traits like personality, emotions, or impulsiveness.

For instance, the Turing Test, a widely recognized benchmark for AI, examines whether an AI system can behave indistinguishably from a human during a text-based interaction.

For the purpose of your studies, and following the [RusselNorvig] reference, the adopted definition will be: "An AI is an artificial system that acts rationally." This definition is chosen for its didactic usefulness, as it allows the focus to be placed on actions (which are easier to define in a computer system) and rationality (which can be measured by comparing actions to goals, simpler than comparing to a real human). However, it's important to note that this definition doesn't claim to be the absolute or perfect one, and other definitions may be more suitable depending on the context.

1.1.2 What is an Agent

In the field of Artificial Intelligence, an agent is a software entity that operates autonomously within a certain environment. The agent interacts with its environment via sensors and actuators. Sensors enable the agent to perceive information about its environment. For instance, a self-driving car might use RADAR, LIDAR, or camera-based sensors to understand its surroundings. Actuators,

on the other hand, allow the agent to interact with its environment. For the self-driving car, these could include systems that control acceleration, steering, or turn signals.

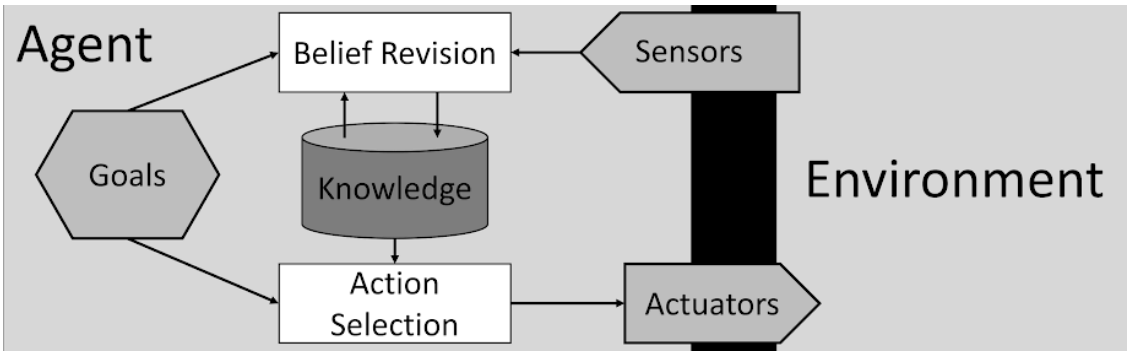
To understand AI as a rational agent, which is the perspective adopted from [RusselNorvig] for this learning resource, it's necessary to define what "rationality" means in this context. Rationality requires a clear goal representation, which might also be called a performance metric or measure. A goal could be a specific condition the agent aims to fulfill (e.g., reaching a particular destination), a situation the agent wants to avoid (e.g., avoiding collisions), or a function the agent aims to optimize (e.g., minimizing fuel consumption or travel time). An agent's behavior, which includes actions executed by its actuators, is deemed rational if it aligns with achieving these goals.

There are many different methodologies for implementing rational agents, and the study of autonomous systems and agent-oriented design are research fields in their own right. However, for the purpose of this resource, the concept of an agent is used primarily as a structuring mechanism, placing the fields commonly associated with AI into the context of the agent.

One key assumption is that the agent possesses an internal knowledge base containing information about its environment and itself. Two significant aspects of an agent's implementation are closely tied to this knowledge base:

Belief Revision: This is the process of incorporating information derived from sensors into the agent's knowledge base. Given that sensor data can be partial or inaccurate, this process might also involve correcting existing knowledge or resolving conflicts between the agent's current knowledge and new perceptions.

Action Selection: This involves deciding which actuator to activate and when. This decision-making process typically relies on the agent's current knowledge and its goals. The activation of an actuator is referred to as an "action."



1.1.3 Which topcis are part of AI?

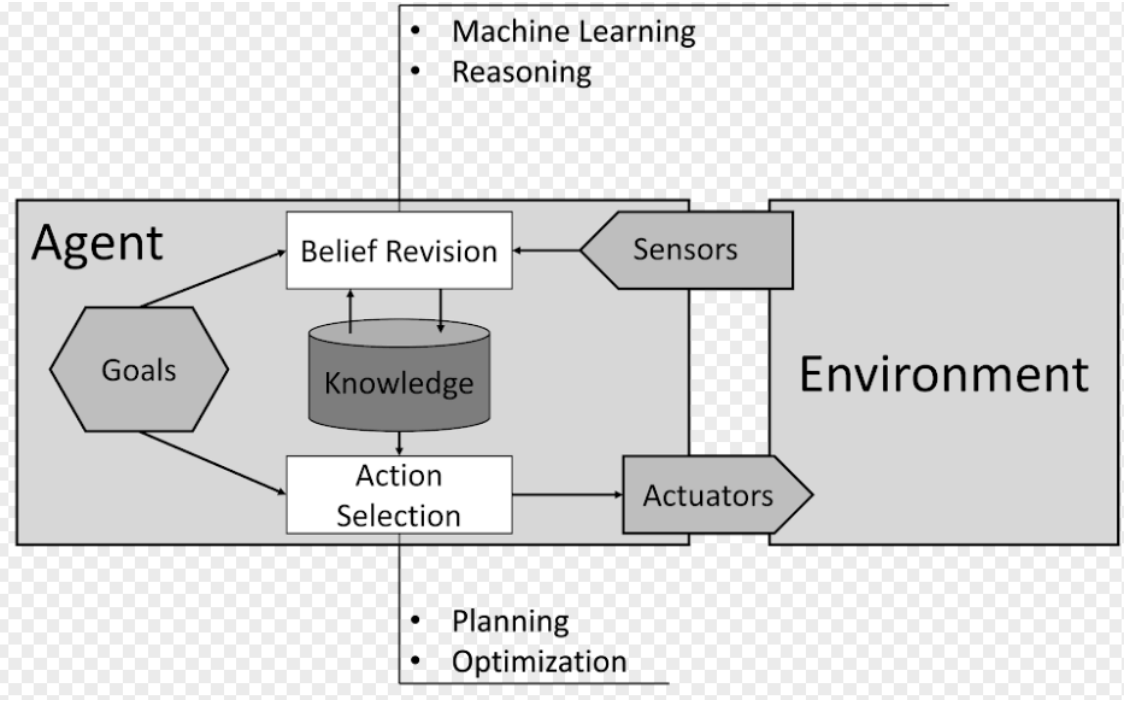
Today, a huge part of the field of AI is not directly concerned with the implementation of a rational agent. The field now also consists of several sub-fields, like machine learning or planning, that represent technologies that can be used to implement an AI but would not constitute an actual AI, even if implemented perfectly. The module AI Basics focuses on four of these fields: Optimization, Machine Learning, Planning and Reasoning.

These can be placed into the definition of a rational agent as depicted in Figure 2. Machine Learning and Reasoning can both be used for belief revision. Both can be used to derive new knowledge. Reasoning uses logical deduction to do this. E.g., by knowing that Waldi is a dog, we can derive

that Waldi can bark. Machine learning derives new knowledge from statistical correlations. E.g., after repeatedly observing that pressing a button coincides with a light bulb switching on, it may be derived that the button switches on the light.

Planning and Optimization are both mechanisms that can be used to select actions. While Planning tries to derive a sequence of actions that will transition the agent to a goal state, optimization aims to find a configuration of variable assignments that optimizes a function (setting the variables to the assigned values is the action in this context).

These four techniques form the main content of the module Artificial Intelligence Basics and will be handled in more detail in later sections of this classroom.



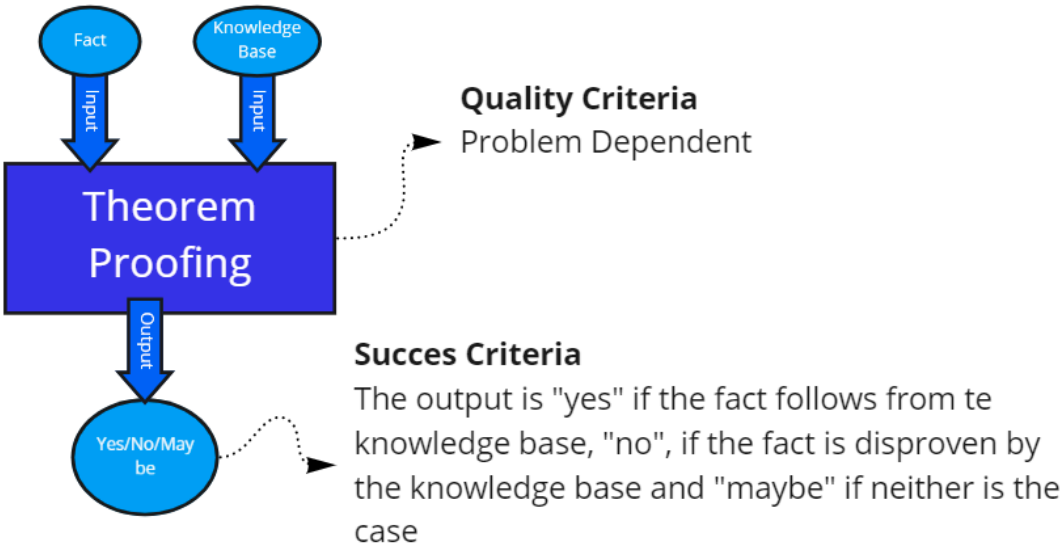
1.2 What is an AI Problem?

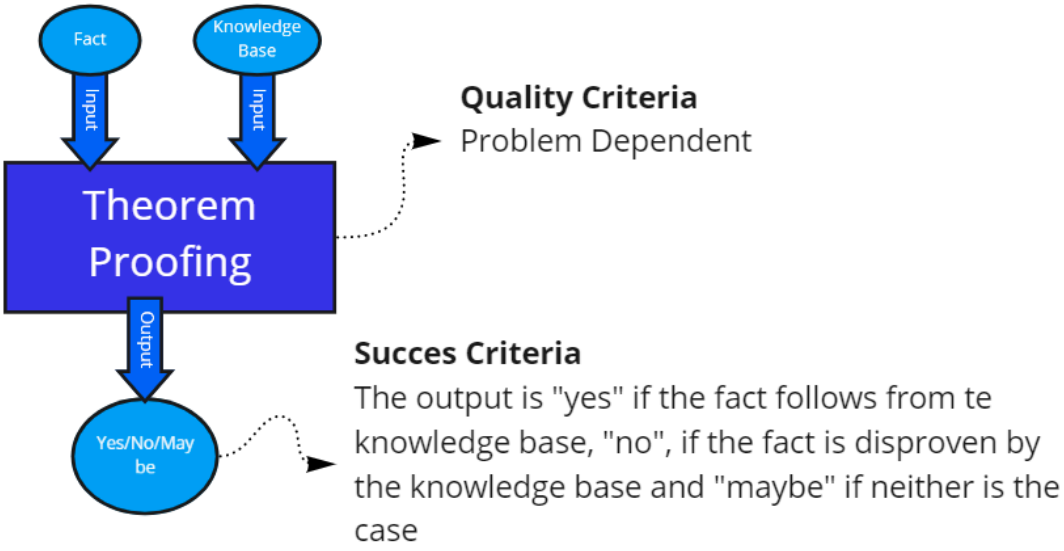
The purpose of a CODE problem that applies AI technology will often not be to implement an actual AI. They are often used to solve an AI problem, a task that requires the application of a technology associated with AI. This is another mechnism we will use to tie together the fields of AI in this classroom. We will use an abstract problem definition for AI problems for this purpose. A graphical representation of this AI problem can be seen in the attached image.

This problem definition defines the search for an algorithm that fulfills a set of requirements: Input: The algorithm needs to deal with certain input types. Output: The algorithm needs to produce specific output types. Success Criteria: The output needs to fulfill certain criteria to be considered to be the correct output. Quality Criteria: The algorithm needs to fulfill certain criteria (e.g., maximum run time, memory constraints) to be considered usable for your project. An example AI problem for calculating a route is shown in the second attached figure. The required algorithm should calcuate a route from a start to an end point through a street network and result in a sequence of navigation instructions for this route. The output is considered correct if the navigation instructions indeed get you from start to end point and it has the quality criteria of using less than

1 GB RAM, which is presumably a restriction of the system the algorithm is supposed to run on.

The parameters of the AI problem change depending on which are of AI the problem belongs to. This also implies that formulating such a problem for a task in your project should indicate to you, which area of AI is useful for solving this problem. The specific problem formulations for each area of AI will also be discussed in the later sections of this classroom.





1.3 Optimization

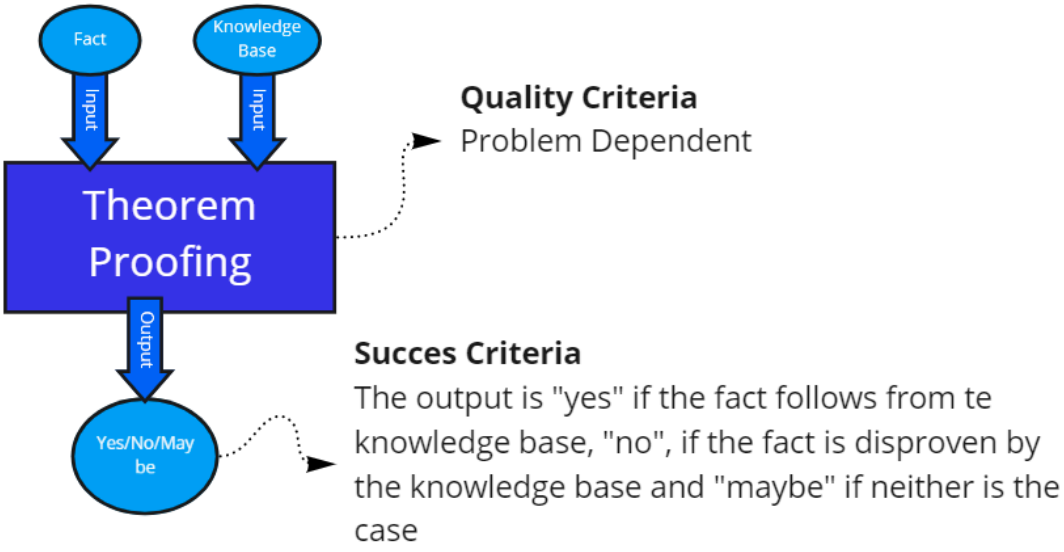
1.3.1 What is Optimization

Optimization aims to optimize a function. Specifically, it aims to find those input parameters, that optimize the output value of a function.

Let's define a simplified running example based on cynical capitalism to illustrate this. Our function is the output of a production facility (e.g., measured in profit from selling the produced items, minus production costs). This output depends on the management of that facility. Let's assume that there are two parameters that can be changed by management: the number of workers employed at the facility (#workers) and the number of hours each worker works per day (#hours). The goal of optimization is then to find values for #workers and #hours that maximize the profit. Written down as the signature of a function, this situation looks like this: revenue: Integer x Integer -> Double. This means, the function takes two integers as input (#workers, #hours) and produces a double (the revenue).

If we were to define this as an AI problem we would take this revenue function as input and would want to find an algorithm that outputs that pair of #workers and #hours that will lead to the highest revenue.

In general, an AI problem associated with optimization will look like shown in the attached image. The input is given by a function that should be optimized and the output is an assignment of the input variables of this function. The sucess criteria are defined by the requirement of the found variable assignment to maximize or minimize the function. The quality criteria are left blank here as they are highly problem dependent. They will influence which optimization algorithms can be used but will not influence whether something is regarded as optimization or not.



1.3.2 How to Solve an Optimization Problem?

There are different ways to solve optimization problems. If the function is known in mathematical notation and is differentiable, it is possible to use mathematical approaches to find its maximal

or minimal values. This can be done by calculating the first derivation and finding those points where this derivation is zero (because this signifies a maximum, minimum or saddle point of the original function). Making use of this, it is often possible to directly calculate the solution to an optimization problem, especially for polynomials.

However, in some cases the mathematical representation of the function is not known, not differentiable or contains an infinite amount of extrema. This is the case in our revenue example, where we do not have a mathematical way to represent productivity of an individual based on working hours and thus could not write down a formula that calculates the revenue. In these scenarios, the usual strategy is to try out combinations of parameters, searching for one with a good result.

In theory, once you try out all possible combinations of parameters, this also leads to the absolute maximum / minimum. However, in praxis, this is often not possible due to time or resource constraints, or because there simply are infinitely many combinations. In our revenue example, let's say trying out a combination of number of employees and working hours requires one month, to get a statistically significant revenue value. This means, even if we have only ten different numbers of employees and five different values for working hours, it'll take 50 months (i.e., more than 4 years) to find the optimum. This is often called the state space explosion: the number of states that needs to be searched to know for certain that you have found the optimum quickly exceeds the number of states that can be searched with the available resources.

For this reason, most non-mathematical optimization algorithms aim at finding ways to determine which states should be visited in which order to maximize the likelihood of finding an optimal solution (or a solution that is close to being optimal). They make use of different properties of the search space. For example, it may make sense to start at one point and always go to the neighbor with the highest value, thus climbing higher and higher until you find a point where all neighbors are lower (this is called hill-climbing).

One thing that should be stressed is that there are different types of optimization problems. They differ in the types of parameters and properties of the function. In general, we can distinguish four types of parameters : Categorical: These are parameters that can have a discrete amount of unordered values. An example are Color names (e.g., pink, red, blue, green, cyan). There is a finite number of named colors, but they have no inherent order. Discrete Numerical: These are parameters that have numerically ordered values of discrete nature. Examples are natural or whole numbers. Continuous Numerical: These are parameters that have numerically ordered values of continuous nature. An example are real numbers. Ordinal: These are categorical variables that nevertheless have a numerical order. An example are star ratings (one star, two stars, three stars, …) that represent a discrete set of categories, but has a numerical relation (two stars > one star). Typically, Optimization is concerned with numerical values. However, in some cases the other values also can appear (beyond being border cases for optimization, they will also be relevant for machine learning, which is why they are listed here).

The applicability of optimization methods often depends on the type of input and output values of the optimized function. For example, hill climbing, as described above, is only possible if a notion of neighboring state exists. In our example of a revenue function, that has two discrete input values #workers and #hours, it would be applicable. But in other functions with continuous input parameters it could not be applied directly (although a generalization of hill climbing to continuous spaces exists and is called gradient ascent/gradient descent).

Another thing that influences the choice of algorithm is the form of the function. Is it guaranteed to have exactly one optimum, or are there multiples? Is it steadily growing towards the optimum

or randomly jumping around in state space? As one can imagine, depending on these properties different approaches to finding the optimal value make sense.

So far, we've avoided talking about specific algorithms as much as possible. That's what the learning resources below are for, as depending on your problem you'll have to look into different algorithms. Here are a few general ones that you should know about, though: Hill Climbing, Gradient Descent, Simulated Annealing, Evolutionary Algorithms.

1.3.3 Simulated Annealing

Overview Simulated Annealing (SA) is a probabilistic optimization technique inspired by the annealing process in metallurgy, where it's used to alter the physical properties of a material. This method is effective in finding the global optimum of a problem, especially in large and complex search spaces where deterministic algorithms might get stuck in local optima.

Here's how Simulated Annealing works in the context of optimization:

Initialization: We start with a random solution to the problem and set an initial high "temperature", a parameter that controls the randomness of the search.

Iteration: At each step of the algorithm, we propose a new solution, typically a small random alteration of the current one.

Transition: The decision to move to this new solution depends on the change in the objective function (i.e., how much better or worse the new solution is) and the current temperature. If the new solution is better, it is always accepted. If it is worse, it might be accepted with a probability that depends on the difference in solution quality and the temperature.

Cooling: After a fixed number of iterations, the temperature is reduced according to a cooling schedule (hence the term "annealing"). This reduction gradually lessens the probability of accepting worse solutions, making the search more focused as it progresses.

Termination: The algorithm continues iterating and cooling until it reaches a stopping condition, typically a sufficiently low temperature or a maximum number of iterations. The best solution found during the process is returned as the output.

The advantage of Simulated Annealing is its ability to escape local optima by allowing worse solutions at higher temperatures. As the temperature decreases, the algorithm gradually refines the search to converge towards a global optimum.

However, its performance highly depends on the choice of parameters, such as the initial temperature, the cooling schedule, and the method for generating new solutions. These need to be carefully chosen and might require several trials to find an appropriate combination.

Detail

Simulated Annealing for beginners Finding an optimal solution for certain optimisation problems can be an incredibly difficult task, often practically impossible. This is because when a problem gets sufficiently large we need to search through an enormous number of possible solutions to find the optimal one. Even with modern computing power there are still often too many possible solutions to consider. In this case because we can't realistically expect to find the optimal one within a sensible length of time, we have to settle for something that's close enough.

An example optimisation problem which usually has a large number of possible solutions would be the traveling salesman problem. In order to find a solution to a problem such as the traveling salesman problem we need to use an algorithm that's able to find a good enough solution in a reasonable amount of time. In a previous tutorial we looked at how we could do this with genetic algorithms, and although genetic algorithms are one way we can find a 'good-enough' solution to the traveling salesman problem, there are other simpler algorithms we can implement that will also find us a close to optimal solution. In this tutorial the algorithm we will be using is, 'simulated annealing'.

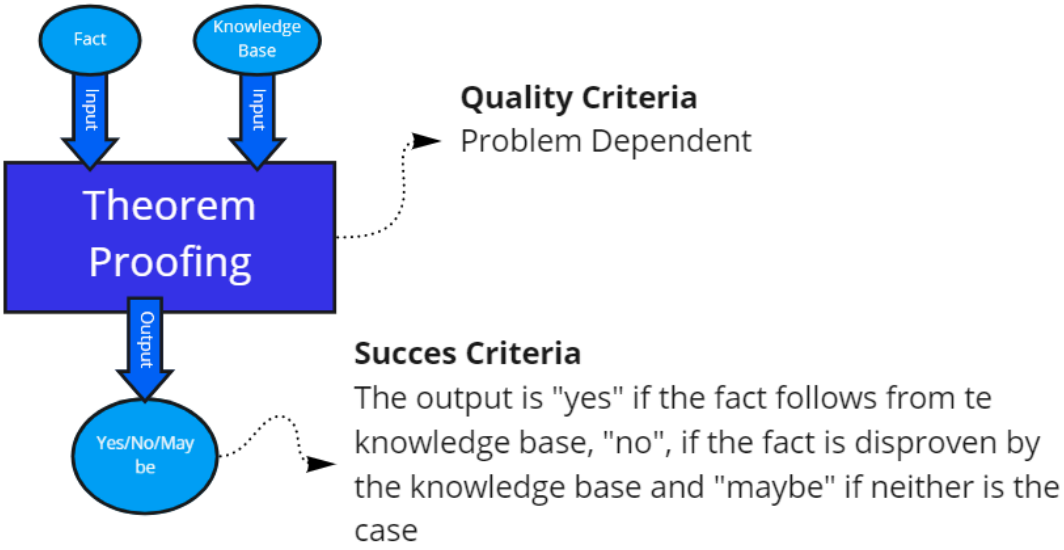
If you're not familiar with the traveling salesman problem it might be worth taking a look at my previous tutorial before continuing.

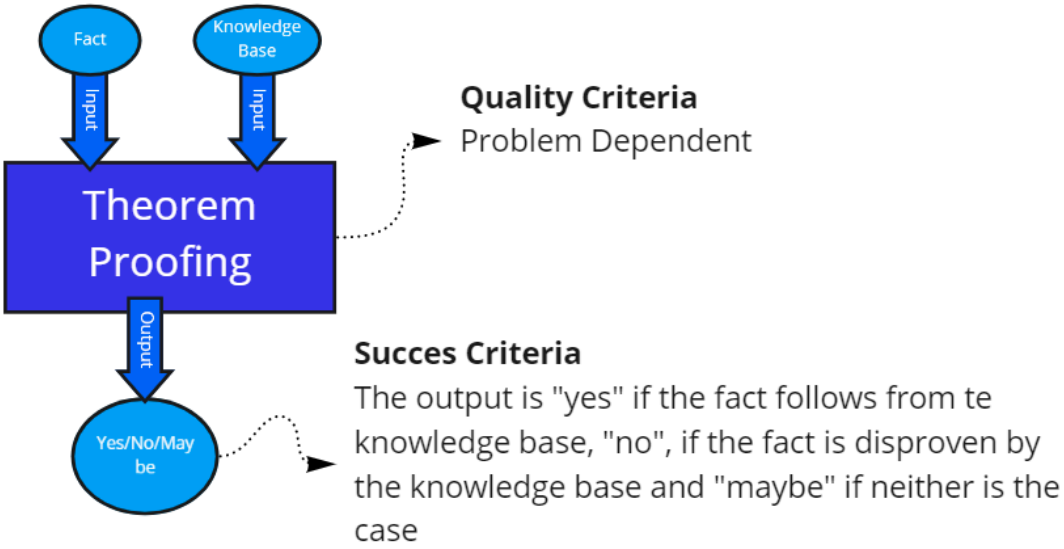
What is Simulated Annealing? First, let's look at how simulated annealing works, and why it's good at finding solutions to the traveling salesman problem in particular. The simulated annealing algorithm was originally inspired from the process of annealing in metal work. Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties. In simulated annealing we keep a temperature variable to simulate this heating process. We initially set it high and then allow it to slowly 'cool' as the algorithm runs. While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optimums it finds itself in early on in execution. As the temperature is reduced so is the chance of accepting worse solutions, therefore allowing the algorithm to gradually focus in on a area of the search space in which hopefully, a close to optimum solution can be found. This gradual 'cooling' process is what makes the simulated annealing algorithm remarkably effective at finding a close to optimum solution when dealing with large problems which contain numerous local optimums. The nature of the traveling salesman problem makes it a perfect example.

Advantages of Simulated Annealing You may be wondering if there is any real advantage to implementing simulated annealing over something like a simple hill climber. Although hill climbers can be surprisingly effective at finding a good solution, they also have a tendency to get stuck in local optimums. As we previously determined, the simulated annealing algorithm is excellent at avoiding this problem and is much better on average at finding an approximate global optimum.

To help better understand let's quickly take a look at why a basic hill climbing algorithm is so prone to getting caught in local optimums.

A hill climber algorithm will simply accept neighbour solutions that are better than the current solution. When the hill climber can't find any better neighbours, it stops.





In the example above we start our hill climber off at the red arrow and it works its way up the hill until it reaches a point where it can't climb any higher without first descending. In this example we can clearly see that it's stuck in a local optimum. If this were a real world problem we wouldn't know how the search space looks so unfortunately we wouldn't be able to tell whether this solution is anywhere close to a global optimum.

Simulated annealing works slightly differently than this and will occasionally accept worse solutions. This characteristic of simulated annealing helps it to jump out of any local optimums it might have otherwise got stuck in.

Acceptance Function Let's take a look at how the algorithm decides which solutions to accept so we can better understand how its able to avoid these local optimums.

First we check if the neighbour solution is better than our current solution. If it is, we accept it unconditionally. If however, the neighbour solution isn't better we need to consider a couple of factors. Firstly, how much worse the neighbour solution is; and secondly, how high the current 'temperature' of our system is. At high temperatures the system is more likely accept solutions that are worse.

The math for this is pretty simple:

[ ]:
```
exp( (solutionEnergy - neighbourEnergy) / temperature )
```

Basically, the smaller the change in energy (the quality of the solution), and the higher the temperature, the more likely it is for the algorithm to accept the solution.

Algorithm Overview So how does the algorithm look? Well, in its most basic implementation it's pretty simple. First we need set the initial temperature and create a random initial solution. Then we begin looping until our stop condition is met. Usually either the system has sufficiently cooled, or a good-enough solution has been found. From here we select a neighbour by making a small change to our current solution. We then decide whether to move to that neighbour solution. Finally, we decrease the temperature and continue looping

Temperature Initialisation For better optimisation, when initialising the temperature variable we should select a temperature that will initially allow for practically any move against the current solution. This gives the algorithm the ability to better explore the entire search space before cooling and settling in a more focused region.

1.3.4 Multi-Objective Optimization

Multi-objective optimization, as the name suggests, is an area of optimization that involves optimizing multiple conflicting objectives simultaneously. It's a complex process as improving one objective may lead to the degradation of another, making it difficult to find a single solution that optimally satisfies all objectives.

In most real-world problems, multiple criteria are needed for decision-making. For example, designing a car involves considering factors like cost, fuel efficiency, safety, comfort, and so forth. These criteria often conflict with each other; a more comfortable car might be less fuel-efficient or more expensive. In such cases, multi-objective optimization techniques come into play.

Key concepts in multi-objective optimization include:

- Pareto Optimality: A solution is said to be Pareto optimal if there's no other solution that improves one objective without worsening at least one other objective. The set of all Pareto optimal solutions is called the Pareto front or Pareto set.
- Trade-off: Since improving one objective might lead to the degradation of another, there's often a trade-off to be made. The decision-maker needs to make choices based on their preferences and the relative importance of the objectives.
- Dominance: One solution dominates another if it's at least as good in all objectives and strictly better in at least one objective.

There are several approaches to solve multi-objective optimization problems:

- Weighted Sum Method: This involves converting the multiple objectives into a single objective by assigning weights to each objective according to their importance. However, this approach can't capture Pareto solutions that are non-convex.
- Evolutionary Algorithms: Techniques such as the Non-dominated Sorting Genetic Algorithm (NSGA-II) and the Multi-Objective Genetic Algorithm (MOGA) use evolutionary principles to explore the search space and converge towards the Pareto front.
- Decision-making methods: These include techniques like TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution) and ELECTRE (ELimination and Choice Expressing REality), which rank solutions based on various criteria.

The goal of multi-objective optimization is to find the Pareto front and provide the decision-maker with a set of optimal trade-off solutions, from which they can choose based on their preferences.

1.3.5 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are optimization methods inspired by the process of natural evolution. These algorithms are typically used in optimization problems where the search space is large, complex, or poorly understood. The core idea behind EAs is to generate a population of potential solutions and then iteratively improve them through processes analogous to genetic inheritance and natural selection.

Here's a detailed explanation of how Evolutionary Algorithms work:

1. Initialization: The algorithm begins by creating an initial population of candidate solutions. These solutions are typically represented as strings (like chromosomes in DNA) and can be randomly generated.
2. Evaluation: Each member of the population is evaluated using a fitness function, which quantifies the quality of the solution. The fitness function is problem-specific and provides a measure of how well a solution meets the desired objectives.
3. Selection: Based on fitness, some members of the population are selected to pass on their genes to the next generation. Solutions with higher fitness have a higher chance of being selected. This process mimics the survival of the fittest principle in natural selection.
4. Crossover (or Recombination): Selected solutions are paired up and exchanged parts of their structure to produce offspring. This process is akin to genetic recombination in biology. The point(s) at which the solution string is broken and recombined is randomly chosen.
5. Mutation: To maintain genetic diversity in the population and prevent premature convergence to suboptimal solutions, mutation is applied. This involves making small, random changes to the offspring solutions. Mutation helps in exploring new points in the search space.
6. Termination: Steps 2 to 5 are repeated for the new generation, and this process continues until a termination condition is met. Termination could be after a fixed number of generations, a satisfactory fitness level is reached, or if there is no significant improvement in the population.

There are several types of Evolutionary Algorithms, including Genetic Algorithms (GAs), Evolutionary Strategies (ES), Genetic Programming (GP), and Evolutionary Programming (EP). While

the basic principles are similar, they differ in aspects like representation of solutions, variation operators used (crossover and mutation), and selection strategy.

It's important to note that while EAs can often find good solutions, they do not guarantee finding the optimal solution, especially for complex problems. Also, EAs require careful tuning of parameters, such as population size, mutation rate, and crossover rate, to work effectively.

1.3.6 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based optimization algorithm inspired by the social behavior of bird flocking or fish schooling. Developed by Kennedy and Eberhart in 1995, PSO is used in problems where the optimal solution can be represented as a point or surface in an n-dimensional space.

Here's how the Particle Swarm Optimization algorithm works in detail:

1. Initialization: The algorithm begins by creating a swarm of particles, where each particle represents a potential solution in the problem space. Each particle is initialized with a random position and a random velocity.
2. Evaluation: Each particle's fitness is evaluated based on an objective function specific to the problem being solved.
3. Update Velocity and Position: Each particle's velocity and position are updated based on its own best known position (pbest) and the best known position in the swarm (gbest). The updated velocity considers the particle's previous velocity, the distance from the particle to pbest, and the distance from the particle to gbest. A key aspect of this step is the balance between exploration (searching new areas) and exploitation (refining current promising areas), which is controlled by cognitive and social scaling parameters.
4. Update Personal and Global Bests: If the new position of a particle has a better fitness than its pbest, then its pbest is updated. Similarly, if a particle has a better fitness than the current gbest, then gbest is updated.
5. Termination: Steps 2-4 are repeated for a set number of iterations or until a stopping criterion is met. The stopping criterion could be reaching a desired fitness score, or no significant improvement in gbest over a certain number of iterations.

The result of the PSO algorithm is the gbest - the best solution found over all particles and iterations.

PSO is a simple yet powerful optimization algorithm. It has a few key parameters that need to be set, such as the number of particles in the swarm, the cognitive and social scaling parameters, and the maximum velocity a particle can achieve. These parameters can significantly influence the performance of the algorithm.

PSO has been successfully applied to a variety of optimization problems, including function optimization, neural network training, fuzzy system control, and other areas where local search strategies tend to fall into local optima.

1.4 Machine Learning

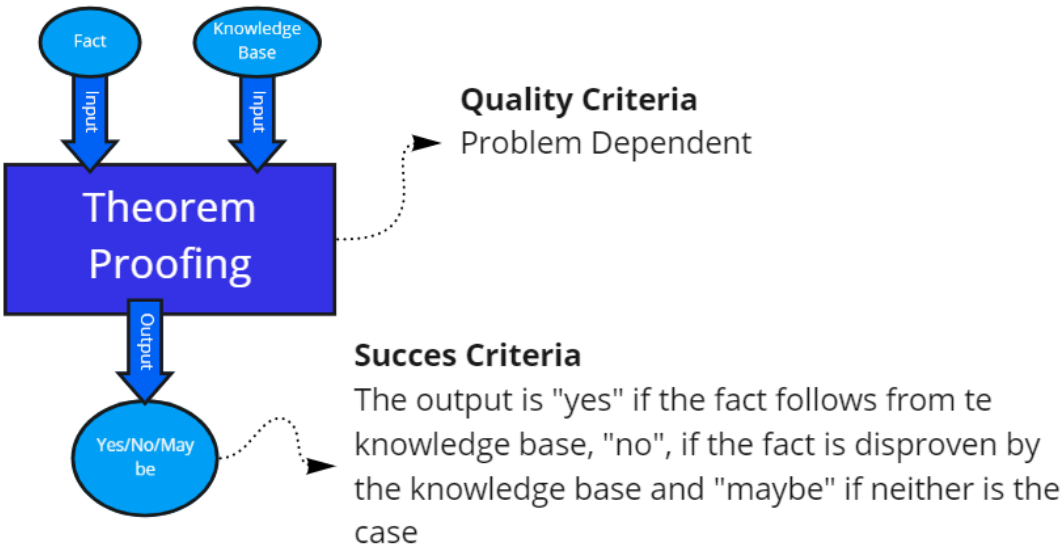
1.4.1 What is Machine Learning

Machine Learning as a field is concerned with learning correlations in structured data. It applies statistical tools, to identify these correlations. The basic idea: machine learning finds patterns in data. There are a few different types of machine learning. However, before we briefly describe these, let's start with defining the AI problem for one of them: supervised learning.

Supervised Learning aims to learn a function from examples of its input and output values. For an example, we can use the revenue function we already used for optimization: revenue: Integer x Integer -> Double that takes two integer values - a number of employees and the number of work hours per day - and produces a double value that represents the monthly revenue. While optimization is concerned with finding the combination of input parameters that maximizes this function, machine learning is concerned with finding a mathematical representation of this function. Or at least a mathematical representation of a function very close to it.

To finds this function, supervised learning needs examples of input and output of the function. If our example production facility has already run for a few years, it may have experimented with different numbers of employees and working hours and may have recorded the revenues. These are examples that can be used for machine learning. They could look like this: (5 Employees, 10 hours, 1.500 €), (7 Employees, 5 hours, 2.100 €) (8 Employees, 5 hours, 2.400 €) (10 Employees, 5 hours, 2.000 €), (9 Employees, 5 hours, 2.200 €), (11 Employees, 5 hours,1.800 €) (10 Employees, 8 hours, 800 €),… Supervised Learning tries to find a function that would produce these data points, if applied to the input. For that it needs to make certain assumptions about the function (e.g., is it linear, is it a polynom, etc.). These assumptions are usually called a machine learning model. An example is linear regression, which assumes the function is linear.

Packing this into an AI Problem, a supervised machine learning problem usually looks like the attached image. The input is a dataset with structured data samples of input and output values of a function and the output is a function that tries to reproduce the output from the input. It is considered successful if it rexplains the data samples well (we usually can't get 100% correct predictions). The quality criteria for machine learning depend on the use case.



1.4.2 How to select a Machine Learning Model?

Machine Learning requires the selection of a suitable model. The model determines which types of functions can be learned. For example, a linear regression model assumes the function is linear, i.e., of the form f(x) = a*x + b. An example of a linear regression model can be seen in the attached image. The dots represent the data points of a two-dimensional function and the line is a linear regression model.

While the model defines the shape of the function, the actual learning process is responsible for finding the function of that shape tat best explains the data points. In our linear regression model, the parameters a and b determine the shape of the linear function. A machine learning process tries to find values for these parameters such that the function is as close as possible to the data points. In the example linear regression in the attached figure it is not possible to have zero distance to all data points, because the data points are not on a line.

This also illustrates that the choice of machine learning model matters a lot. In general, the more complex the model, the better it can cover data points. However, often, covering the data points with zero error is not desirable. The reason is that data points can be inaccurate. E.g., in our example, the revenue is not solely determined by the number of employees and working hours. It may also be determined by external factors such as the season (e.g., Christmas sales). We say the data contains noise: disturbances that are not caused by the function we are trying to learn and that must be factored out. Finding a model with zero error for all data points usually means we have learned the disturbances alongside the function. We say that the learning process has “overfit” to the data.

Linear regression is just one of many machine learning models. As with optimization, we’ll not go into details of these models, but rather discuss factors that influence, which model you want to choose. And as with optimization, the type of your variables is a major factor. Depending on whether you are dealing with categorical, numerical or ordinal variables, different machine learning algorithms are used. In fact, two of the main areas of machine learning are defined as supervised learning with a different function type: Classification: is a supervised learning problem

with a function of categorical value. Regression: is a supervised learning problem with a function of (continuous) numeric value. Functions with discrete numeric or ordinal value can usually be tackled with classification or regression approaches, or a combination of them.

The opposite of supervised learning is unsupervised learning. The difference is that the data does not contain an explicit output value for the function. In terms of a learned function we could say that we have no knowledge of the output value. Unsupervised learning approaches are learning the output value along with the function. This gives rise to a different area of machine learning: Clustering: is an unsupervised learning problem that has categorical output. Clustering is useful when you want to categorize data points based on their spatial relation (e.g., learn a notion of "that blob of data points there that seems to be very similar"). While clustering will not tell you what these similarities mean, it is good at uncovering those similarities in the first place.

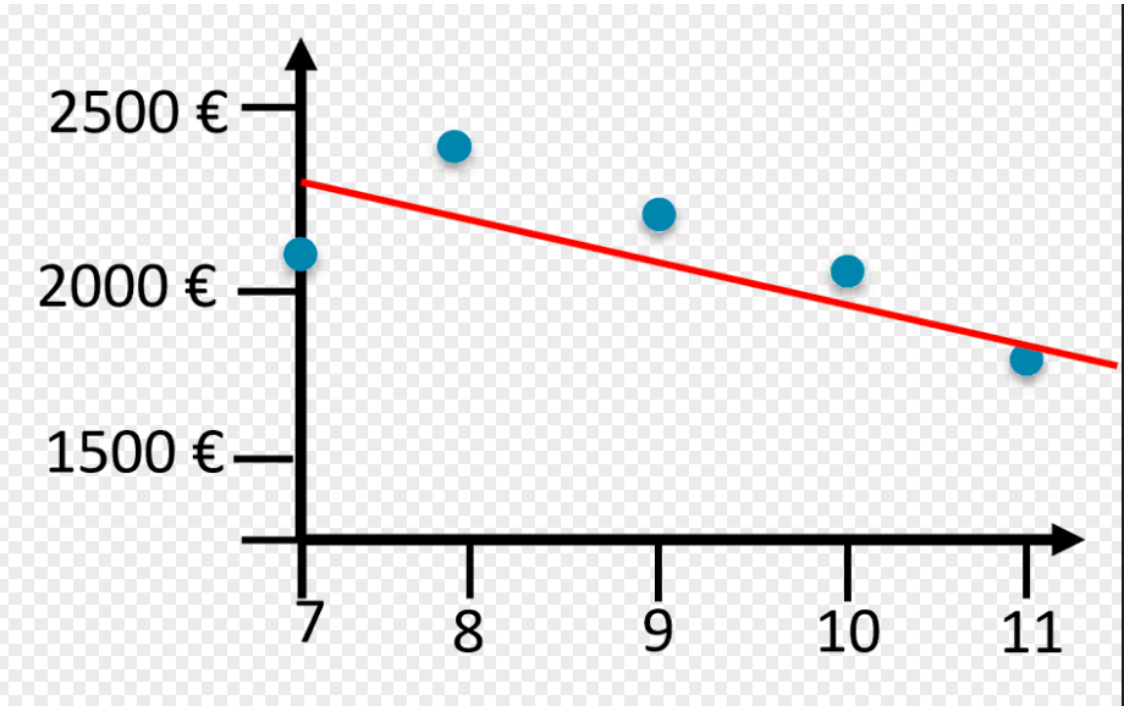
In between supervised and unsupervised learning approaches lies semi-supervised learning. Here, only some of the samples have output values and some don't.

Another big area of machine learning is called reinforcement learning. This is very closely related to the concept of agents and actions. The function that is learned is responsible for which action an agent selects in a certain situation, either directly or indirectly (e.g., by assigning probabilities or utilities to actions). If such a function was learned in a supervised manner, it would need to be trained with the right actions to take in each situation. However, in reinforcement learning environments (e.g., a game of chess) you only know how good an action was some time after you applied it (i.e., whether you won or lost the game in the end) and don't have the ability to test alternative actions (i.e., you can't easily ask your oponent to take your moves back and try a differen one). This means, instead of having absolute data about the quality of each action, reinforcement learning approaches have to make due with limited knowledge about the rewards achieved by choosing certain actions in certain situations.

These four areas (supervised, unsupervised, semi-supervised and reinforcement learning) are the main areas of machine learning and depending on your AI problem you are usually working in one of them.

There are some other differences in algorithms. For example, how much data they can be trained on at a time, how much time is required for training, how many samples they need to provide reasonable results, etc. You can find some of these when researching algorithms to use.

One thing that should be pointed out explicitly is the relation between machine learning and optimization. (Supervised) machine learning aims to explain all samples by learning the right parameters of the respective machine learning model. This is expressed by defining an error function (e.g., the mean distance between the learned function and data samples). The learning process then is the task of finding those parameters for the machine learning model that minimize the error function. E.g., for a linear model that is described as the function f(x) = a*x + b, the machine learning process finds those values of a and b that minimize the error. If this sounds familiar, it is because this is an optimization problem. And indeed, optimization is what is often behind the learning process of machine learning. For example, backpropagation, the learning algorithm for neural networks, uses gradient descent, one of the optimization algorithms we mentioned above.



1.4.3 Intro to Machine Learning

** General machine learning / data science layer, as well as the mathematics behind some individual algorithms.**

The data science pipeline often consists of several steps or layers, each with its own purpose. Here are the general layers in a typical machine learning or data science pipeline:

1. Data Collection: This is the initial phase where data is collected from various sources, which could be databases, data files, APIs, web scraping, IoT devices, etc.
2. Data Preprocessing: Data in the real world is messy. This step involves cleaning the data (handling missing values, outliers, etc.), transforming the data (normalization, scaling, encoding categorical variables, etc.), and feature engineering to make the data suitable for machine learning models.
3. Exploratory Data Analysis (EDA): EDA involves understanding the data through statistical summaries and visualizations to identify patterns, relationships, or anomalies that might influence the way we model and interpret the data.
4. Model Training: This involves selecting an appropriate machine learning algorithm and using it to learn from the data. This often involves dividing the data into a training set and a validation set.
5. Model Evaluation: The trained model is evaluated using appropriate metrics (accuracy, precision, recall, F1-score, ROC-AUC, RMSE, etc., depending on the task) to measure its performance. The model might be fine-tuned or hyperparameters might be optimized based on these evaluations.
6. Model Deployment: Once a satisfactory model is developed, it's deployed into a production environment where it can be used to make predictions on new, unseen data.

7. Monitoring and Updating: Post-deployment, the model performance is continuously monitored. If the model's performance declines, or as new data becomes available, the model might need to be retrained or updated.

As for the mathematics behind some individual algorithms, let's look at a couple of examples:

1. Linear Regression: Linear regression aims to model the relationship between two variables by fitting a linear equation to the observed data. The steps to using the method are to: (1) determine the form of the function, (2) estimate the parameters of the function. The standard method of fitting is the method of least squares, which minimizes the sum of the squared residuals, which are the distances of the observed responses to the fitted function.

2. Logistic Regression: Unlike linear regression, which outputs continuous values, logistic regression transforms its output using the logistic sigmoid function to return a probability value, which can be mapped to two or more discrete classes. It uses the concept of odds ratios to calculate the probability.

3. Decision Trees: Decision trees split the data into subsets based on feature values. These splits are made by maximizing a metric known as information gain, which is based on entropy and is used to measure the impurity of the input set. In the case of regression trees, the splits are made based on minimizing a cost function, usually the mean squared error (MSE).

4. Neural Networks: Neural networks are inspired by biological neurons and consist of interconnected layers of nodes or "neurons". Each connection between neurons can transmit a signal from one to another. The receiving neuron processes the signal and signals downstream neurons connected to it. Neurons are organized in layers, where signals travel from the input layer to the output layer, possibly through one or more hidden layers. The learning process involves adjusting the weights of the connections to minimize a cost function, typically using a method called backpropagation.

5. Support Vector Machines (SVMs): SVMs aim to find the hyperplane that maximally separates data points of different classes. The mathematics behind SVMs involves quadratic programming and Lagrange multipliers to optimize the decision boundary.

Each of these algorithms utilizes different areas of mathematics, including calculus (for optimization), linear algebra (for data and model representation),

1.4.4 Model Building and Validation

Model building and validation is a crucial part of the machine learning pipeline. It involves identifying the right model that best fits the data and validates its performance. Here's a detailed process on how to go about it:

1. Understanding the Problem and Data: The first step is to understand the problem you're trying to solve. Is it a regression problem, a classification problem, a clustering problem, or something else? Understanding the problem will guide you in the type of model to choose. It is also necessary to understand the data - its features, distribution, and relationship between variables, etc.

2. Preprocessing: Based on the understanding of the data, preprocess it to make it suitable for the model. This could include encoding categorical variables, normalizing numerical variables, handling missing values, or engineering new features.

3. Model Selection: Once the data is ready, choose an appropriate model. This would largely depend on the type of problem and the data. For example, for a binary classification problem, you could choose from logistic regression, decision trees, SVM, random forests, etc. For a regression problem, you could choose from linear regression, decision trees, random forests, etc. If the data has a lot of features and complex relationships, you might choose a more complex model like a neural network.

4. Training the Model: Train the chosen model on the training dataset. If the model has hyperparameters, you might need to tune them to get the best performance. This can be done using methods like grid search or random search.

5. Model Validation: After training the model, you need to validate its performance. This is typically done using a validation set that was not used during training. Common performance measures include accuracy, precision, recall, F1-score for classification problems and mean squared error, mean absolute error, R-squared for regression problems. If the model's performance is not satisfactory, you might need to go back to the model selection or training step.

6. Cross-Validation: To get a more robust estimate of the model's performance, you can use cross-validation. In k-fold cross-validation, the data is split into k subsets, and the model is trained and tested k times, each time training on k-1 subsets and testing on the remaining one. The average performance over the k iterations is used as the overall performance measure.

7. Testing: Once you're satisfied with the model's performance on the validation set, you can test it on a separate test set. This should give you an unbiased estimate of the model's performance on new, unseen data.

8. Model Interpretation: Finally, it's important to interpret the model. This could involve understanding the feature importance, partial dependence plots, etc. For complex models like neural networks, this could involve using techniques like SHAP or LIME.

Remember that model building and validation is an iterative process. You might not get the best model in the first attempt, but by going through these steps and refining your approach, you can eventually build a model that performs well.

1.4.5 Unsupervised Learning

Unsupervised learning is a type of machine learning where the model learns from a dataset without any explicit supervision. That is, the data is not labeled and the model needs to find patterns and relationships in the data on its own. This makes unsupervised learning more challenging and less straightforward than supervised learning, but it can also be more powerful in the sense that it can discover hidden patterns that might not be noticeable to a human.

Two main types of unsupervised learning are:

1. Clustering: This involves grouping similar instances together. Each group, or cluster, consists of instances that are more similar to each other than to instances in other clusters. Examples of clustering algorithms include K-means, hierarchical clustering, and DBSCAN.

2. Dimensionality Reduction: This involves simplifying the data without losing too much information. One way to do this is to merge several correlated features into one. For example,

Principal Component Analysis (PCA) combines features in a dataset into a new set of features, called components, which are ordered by the amount of variance they can explain.

1.4.6 Reinforcement Learning

Reinforcement learning, on the other hand, is a type of machine learning where an agent learns to make decisions by performing actions in an environment to maximize some notion of cumulative reward. The agent isn't told which actions to take, but instead must discover which actions yield the most reward by trying them out.

Key concepts in reinforcement learning include:

1. Agent: The entity that is learning and making decisions.
2. Environment: The world, or model, through which the agent moves.
3. Action (A): A choice made by the agent.
4. State (S): The situation the agent finds itself in.
5. Reward (R): Feedback from the environment. The agent's goal is to maximize the total reward.

Examples of reinforcement learning algorithms include Q-learning, Deep Q Network (DQN), Policy Gradients, and Proximal Policy Optimization (PPO).

1.4.7 Difference between Unsupervised Learning and Reinforcement Learning

The main difference between unsupervised learning and reinforcement learning lies in the learning process and the type of problem they are trying to solve:

1. Learning Process: In unsupervised learning, the algorithm learns patterns from input data without any explicit target outputs or feedback. In contrast, in reinforcement learning, an agent learns from the feedback (rewards) it receives as it interacts with the environment.
2. Problem Type: Unsupervised learning is usually used for clustering and dimensionality reduction problems where the goal is to find hidden patterns or simplify the data. Reinforcement learning, on the other hand, is used for sequential decision-making problems where the goal is to learn a series of actions that maximizes the total reward.
3. Feedback: There's no concept of reward or penalty in unsupervised learning. However, in reinforcement learning, the agent learns by receiving rewards or penalties.

In a nutshell, unsupervised learning is about finding structure in data, while reinforcement learning is about learning how to behave based on rewards and punishments.

1.4.8 Machine Learning Specialization

Regression with Multiple Input Variables Regression with multiple input variables, also known as multiple regression, is a type of regression analysis that deals with predicting a continuous outcome variable based on more than one predictor variable. The basic form of a multiple regression model is:

[ ]:
```
Y = b0 + b1*X1 + b2*X2 + ... + bn*Xn + e
```

where:

- Y is the outcome variable.
- X1, X2, …, Xn are the predictor variables.
- b0 is the y-intercept.
- b1, b2, …, bn are the coefficients of the predictor variables, indicating the change in the outcome variable for each unit change in the predictor variables.
- e is the error term.

Each predictor variable has its own coefficient, which gives us the expected change in the outcome variable for each unit change in that predictor, holding all other predictors constant.

Improving Your Model's Training and Performance

1. Vectorization: This is a method that involves performing operations on whole arrays rather than individual elements. Vectorization makes use of hardware optimization to achieve faster computation time. It is especially beneficial in high-level programming languages like Python and R, where loops can be slow.

2. Feature Scaling: This involves standardizing/normalizing the range of independent variables or features of data. Techniques such as Min-Max normalization or standardization (Z-score normalization) help to get features into a similar scale and thus prevent certain features from dominating others, which can improve the performance of some models.

3. Feature Engineering: This is the process of creating new features or modifying existing features to improve model performance. This can include creating interaction terms, creating polynomial features, binning variables, and more. Good feature engineering can often make the difference between a mediocre model and a great one.

4. Polynomial Regression: This is a form of regression analysis in which the relationship between the independent variable and the dependent variable is modeled as an nth degree polynomial. Polynomial regression can model relationships between variables that aren't strictly linear.

5. Regularization: Regularization techniques like Lasso, Ridge, and Elastic Net can be used to prevent overfitting and improve the generalization ability of the model.

6. Hyperparameter Tuning: The performance of many machine learning algorithms can be significantly influenced by their hyperparameters. Techniques such as grid search or random search can be used to find the best combination of hyperparameters.

Remember that not every technique will work for every problem, and often the best approach is to try out several and see which works best for your specific problem.

1.4.9 Classification

Classification is a type of supervised learning where the goal is to predict the categorical class labels of new instances, based on past observations. These class labels are discrete, unordered values that can be understood as the group memberships of the instances. Common examples of classification tasks are:

- Medical imaging: Identifying whether a given patient scan contains signs of a disease.
- Email filtering: Identifying whether an email is spam or not.
- Sentiment Analysis: Determining whether a given text is positive or negative.

1.4.10 Logistic Regression

Logistic Regression is a common method used for binary classification problems. It's a statistical model that uses a logistic function to model a binary dependent variable.

The logistic regression model predicts the probability that a given input point belongs to a certain class. If the estimated probability is greater than 0.5, the model predicts that the instance belongs to that class (called the positive class, labeled "1"), and otherwise it predicts that it does not (i.e., it belongs to the negative class, labeled "0").

Mathematically, the logistic regression model uses a logistic function to squeeze the output of a linear equation between 0 and 1. The logistic function (also called the sigmoid function) is an S-shaped curve defined as:

[ ]:
```
f(x) = 1 / (1 + e^-x)
```

1.4.11 Overfitting and Regularization

Overfitting is a common problem in machine learning, where a model performs well on the training data but does not generalize well to unseen data (test data). If a model is overfitting, it is likely capturing noise in the training data.

Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function. The penalty term discourages the learning algorithm from assigning too much importance to any individual feature, and thus reduces overfitting.

Two common types of regularization are L1 and L2 regularization:

- L1 Regularization (Lasso regression): Adds a penalty equivalent to the absolute value of the magnitude of coefficients. This can result in certain coefficients being shrunk to zero, effectively choosing a simpler model that does not include those coefficients.
- L2 Regularization (Ridge regression): Adds a penalty equivalent to the square of the magnitude of coefficients. This tends to distribute the coefficient values more equally.

Both types of regularization reduce the complexity of the final model, and therefore help to prevent overfitting. The regularization term is controlled by a hyperparameter, often denoted by lambda ( ). The higher the lambda, the greater the regularization, and the simpler the resulting model (i.e., with smaller coefficient values).

1.4.12 Neural networks

Neural networks are a class of machine learning models that are inspired by the structure of the human brain. They're designed to recognize patterns and make predictions by simulating the way biological neurons and synapses work. Here's an in-depth look at the concept:

1. Structure of a Neural Network: A neural network consists of layers of interconnected nodes, also known as artificial neurons or simply neurons. Each layer receives input from previous layers (or the data input in the case of the first layer) and sends output to subsequent layers. The layers in a neural network are categorized into three types:

   ```
   - Input Layer: This is the first layer in the network. Each neuron in this layer represer
   ```

- Hidden Layer(s): These are layers between the input and output layers. A neural network

- Output Layer: This is the final layer. The neurons in this layer represent the output v

2. Neuron: Each neuron takes a set of inputs, multiplies each input by a parameter (called a weight), and then sums them up. The sum is then passed through a nonlinear function, called the activation function, to produce the output of the neuron.
3. Activation Function: The activation function introduces non-linearity into the network, allowing it to learn complex patterns. Common choices for activation functions include the sigmoid function, the hyperbolic tangent function, the rectified linear unit (ReLU), and the softmax function.
4. Learning: The goal of training a neural network is to adjust the weights of the neurons in such a way that the network can accurately map input data to the correct output. This is done using a process called backpropagation and an optimization technique, such as stochastic gradient descent (SGD).

   - Backpropagation: In this process, the network makes a prediction (forward pass), and th

   - Loss Function: The loss function quantifies the error made by the network. The goal is

   - Gradient Descent: This is an optimization algorithm that's used to minimize the loss fu

5. Deep Neural Networks: When a neural network contains a large number of hidden layers, it's known as a deep neural network, and the process of training these networks is known as deep learning. These networks are capable of learning very complex patterns and are the driving force behind many state-of-the-art machine learning models.

To summarize, a neural network is a powerful computational model that's capable of learning patterns from data. It's used in a wide variety of applications, including image recognition, speech recognition, natural language processing, and more.

Gradient Descent Gradient descent is an optimization algorithm that's used to minimize a function iteratively. In the context of neural networks, this function is usually a loss function that measures the discrepancy between the network's predictions and the actual data. The goal of training a neural network is to adjust the weights and biases in such a way that this loss is minimized.

The term "gradient" in gradient descent refers to the derivative of the function at a certain point. In multiple dimensions, the gradient is a vector that points in the direction of the steepest increase of the function. The "descent" in gradient descent refers to the fact that we want to go in the opposite direction of the gradient in order to decrease the function, not increase it.

Here's how the gradient descent algorithm works in a nutshell:

1. Initialize the weights and biases with random values.
2. Feed the training data into the network and compute the output for each training example.
3. Compute the loss, which is a measure of the difference between the network's predictions and the actual labels.
4. Compute the gradient of the loss function with respect to each weight and bias in the network. This gradient tells us how much the loss would change if we increased or decreased that weight

or bias by a small amount.
5. Update the weights and biases by taking a step in the direction of the negative gradient. This step size is determined by a parameter called the learning rate.
6. Repeat steps 2-5 until the loss stops decreasing, or after a fixed number of iterations.

Learning in Neural Networks The learning process in neural networks revolves around adjusting the weights and biases to minimize the loss function, as explained above. This process is also known as training the network.

The magic of how neural networks learn lies in a method called backpropagation, which is used to compute the gradient of the loss function with respect to the weights and biases. Backpropagation is essentially the application of the chain rule from calculus to compute these gradients in a systematic way.

Once these gradients are computed, they are fed into the gradient descent (or some other optimization algorithm) to adjust the weights and biases in the network, which in turn changes the network's predictions. This process is repeated over many iterations, and over time the network's predictions get closer and closer to the actual labels. This is how a neural network learns from the data.

Note that there are several variations of the basic gradient descent algorithm that are often used in practice, such as stochastic gradient descent (SGD), mini-batch gradient descent, and variants with momentum like Adam. These methods often converge faster and are more stable than the basic gradient descent algorithm.

What is backpropagation really doing? Backpropagation is an algorithm used for training neural networks, and it is essentially a method for calculating the gradient of the loss function with respect to each weight and bias in the network. To understand what backpropagation is really doing, we first need to understand a little bit about how a neural network makes predictions.

A neural network makes predictions by passing data through layers of nodes (also known as neurons). Each node in a layer takes in data, multiplies it by a weight, adds a bias, and then applies an activation function. The outputs of one layer become the inputs to the next layer. The final layer's outputs are the network's predictions.

The goal of training a neural network is to adjust the weights and biases so that the network's predictions get as close as possible to the actual labels. To do this, we define a loss function that measures the difference between the network's predictions and the actual labels, and we aim to minimize this loss.

Here's where backpropagation comes in. Backpropagation is a method for calculating the gradient of the loss function with respect to each weight and bias in the network. The gradient tells us how much the loss would change if we increased or decreased a weight or bias by a small amount.

Once we have the gradients, we can use them to update the weights and biases. We subtract a small fraction of the gradient from each weight and bias. This process is repeated many times, and over time, the weights and biases are adjusted in such a way that the loss decreases and the network's predictions get closer to the actual labels.

To calculate these gradients, backpropagation uses the chain rule from calculus. The chain rule allows us to compute the derivative of the loss function with respect to any weight or bias in the

network by multiplying together a series of derivatives. Backpropagation starts from the output layer and works backwards through the network, hence the name "backpropagation".

In summary, backpropagation is the method we use to figure out how to change the weights and biases in a neural network to minimize the loss and improve the network's predictions. It's a crucial part of how neural networks learn from data.

Backpropagation calculus The backpropagation algorithm leverages calculus, specifically the chain rule, to compute the gradient of the loss function with respect to each weight and bias in the network. Let's break down the calculus involved.

Consider a neural network with layers, each layer having its own weights and biases. The output of each layer is a function of the output of the previous layer, and the final output of the network (its prediction) is a function of all the weights and biases in the network.

Now, let's say we have a loss function, which measures the discrepancy between the network's predictions and the actual labels. This loss function is a function of the final output of the network, and through that, it's also a function of all the weights and biases in the network.

The goal of backpropagation is to compute the gradient of this loss function with respect to each weight and bias. In other words, we want to know how much the loss would change if we tweaked each weight or bias by a small amount. This information is crucial for training the network because it tells us how to adjust the weights and biases to minimize the loss.

The chain rule of calculus comes into play because we have a function of a function situation. The loss is a function of the network's output, which is a function of the weights and biases. The chain rule allows us to compute the derivative of the loss with respect to a weight or bias by multiplying together a series of derivatives.

In the context of backpropagation, the chain rule is applied repeatedly to propagate error gradients backwards through the network, layer by layer. That's why it's called "backpropagation".

Here's a simple example to illustrate this. Let's say we have a loss function L, which is a function of the network's output y: L = L(y). And y is a function of the weights w: y = y(w). The chain rule tells us that the derivative of L with respect to w is:

[ ]:
```
dL/dw = (dL/dy) * (dy/dw)
```

This formula tells us how much the loss would change if we tweaked the weight w, and it's computed by multiplying two terms: how much the loss would change if we tweaked y, and how much y would change if we tweaked w.

In practice, the situation is more complex because we have many layers and many weights and biases, but the basic idea is the same. We use the chain rule to compute the gradient of the loss with respect to each weight and bias, and we use these gradients to adjust the weights and biases to minimize the loss.

1.4.13 Python and Machine Learning

Python is a popular language for machine learning due to its simplicity and the vast range of libraries and frameworks it offers for both machine learning and data analysis. Here are some key points to understand when using Python for machine learning:

1. Libraries: Python has a wide range of libraries that simplify the machine learning process. Some of these include:
    - NumPy: Provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these elements.
    - Pandas: Offers data structures and operations for manipulating numerical tables and time series.
    - Matplotlib: Useful for creating static, animated, and interactive visualizations in Python.
    - Scikit-learn: Provides simple and efficient tools for predictive data analysis. It is built on NumPy, SciPy, and matplotlib.
    - TensorFlow and PyTorch: Two popular libraries for creating deep learning models.
    - Keras: Provides a higher-level API for building and training deep learning models and is built on top of TensorFlow.
2. Data Preprocessing: The quality and quantity of the data that you use to train your model can greatly affect your model's ability to learn; therefore, it's crucial to preprocess your data before feeding it into a model. This might involve cleaning the data (handling missing data, removing duplicates, etc.), encoding categorical variables, normalizing numerical variables, and splitting the data into training and testing sets.
3. Model Selection: Python provides a variety of machine learning algorithms that you can choose from, ranging from linear regression to complex neural networks. It's important to understand the basics of different types of algorithms, their assumptions, advantages, and disadvantages to select the appropriate model for your specific problem.
4. Training and Evaluation: Once you've selected a model, you'll use your training data to train it. After training, it's important to evaluate your model's performance using appropriate metrics (accuracy, precision, recall, F1 score, ROC AUC, etc.). Python provides functionalities for cross-validation, grid search, and other techniques to optimize hyperparameters.
5. Deployment: After training and validating your machine learning model, the next step is to deploy the model so that it can serve predictions. This might involve saving the trained model (using libraries like pickle or joblib), and then loading it in a different environment to make predictions on new data.
6. Understanding Errors and Improving the Model: Often, the first model you train will not be the best possible model. Understanding where your model is making errors and using this to improve your model is a key part of the machine learning process. This might involve gathering more data, changing your preprocessing steps, or using a different model.

Remember, machine learning is a lot more than just knowing the Python syntax. Good machine learning practitioners understand the theory behind different models and techniques, and they know how and when to use different tools and approaches. They also understand that building a model is an iterative process, and they're always looking for ways to improve their results.

1.5 Planning

1.5.1 What is Planning?

Planning is closely related to the action selection of an agent. It answers the question "which sequence of actions will lead to the goal state". Examples of planning can be found throughout our daily life. For example, making a cup of tea requires us to boil water, then pour it into a can, add

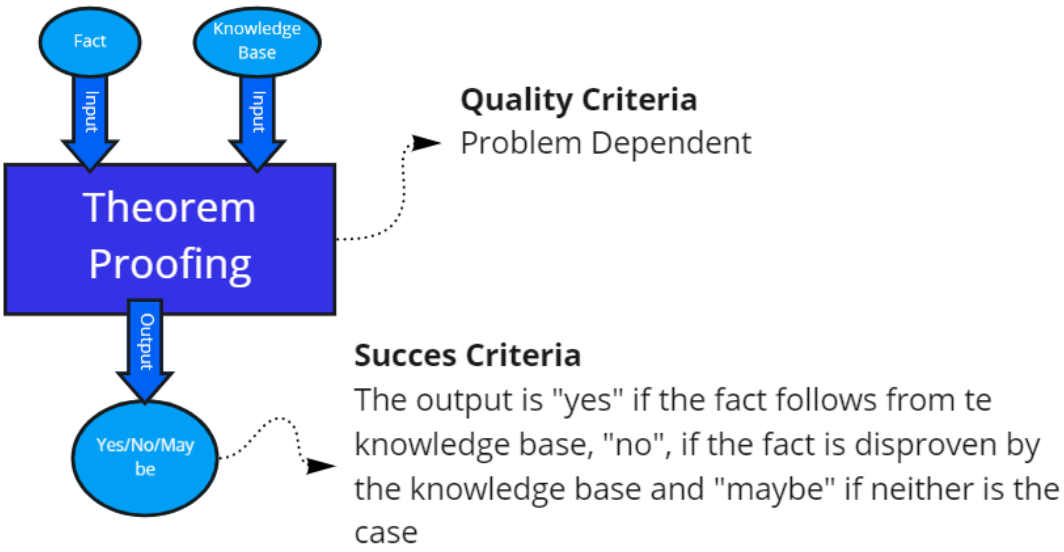
tea bags, wait a few minutes and then pour the tea into a cup. Each of these steps is an action (and we are the agent). Planning tells the agent in which order to apply which action in order to arrive at a goal state.

In general, planning requires the following parts: - A notion of state. In our running example, the state is our knowledge about the world (is there water in the can? Is there water in the cup? Is the water hot? Are teabags in the can? etc.). In the context of an agent, the state is usually reflected in the knowledge of the agent.

- A set of actions that can be performed to change the state. In our running example, we have actions like "heat water","pour water into can" etc. In the context of an agent, the actions are given by the actions of the agent.
- A state-transition function that describes the effect of actions. This function lets us predict what the follow up state is, if we perform an action in a state. In the context of making tea, this function is implicit in our experience on the effect of actions (e.g., our knowledge about the water heater tells us that water will get hot if we switch it on, our knowledge about physics will let us predict what happens if we pour water). In the context of an agent, this is usually contained in the knowledge of the agent.
- A goal function that describes which states are desirable. This function lets us judge whether a state we have found is a valid result of the planning process. While making tea, our goal is to have a cup of tea in front of us. In context of an agent, the goal is encoded explicitly as the agent's goal function.

Given these components, a planning process is a function that produces a sequence of actions, such that these actions, when applied to the current state in the given order, lead to a state that fulfils the goal function.

A general planning AI Problem can be seen in the attached image. The input of the problem is the planning problem with the parts indicated above. The output is a sequence of actions that is considered sucessful if it transitions from the start to the goal state. Quality criteria are problem dependent and not part of the general setup.



1.5.2 How to select a Planning Algorithm?

A planning algorithm finds a way from the start state to a goal state by applying actions. This is illustrated in the attached Figure. The start state is the one on top of the image. From this state, we can apply three actions to arrive in the three mid-level states. From each of these states we can apply the three actions again, to arrive at other states (and so on). By applying Action 2, then Action 1, we can arrive at a goal state (marked in green). Planning is concerned with finding this sequence. Since the search space is a tree (or a graph, if the same state can be reached via different sequences of actions), planning is also sometimes called tree-search or graph search.

As with optimization, we have a problem that is easy to solve, provided all states can be visited in a reasonable amount of time. However, this is often not the case. As can be seen in our toy example already, the number of states in the lower layers grows exponentially with the depth of the search tree. And in realistically sized planning problems, it can quickly grow into a very large number. This represents another case of the state space explosion.

To deal with this, planning algorithms aim to visit states in an order that has a high likelihood to find a goal state early. The most basic algorithms – depth-first and breadth-first search, prioritize going into the depth of one branch of the search tree (depth-first) or searching states in order of their distance to the start state (breadth first). These algorithms are often not enough to achieve good results as they don't take knowledge about the problem into account while searching for a goal state. More intelligent approaches utilize domain knowledge, to estimate which states are more likely to lead to a goal state. While planning for making tea, it may make sense to focus on those actions that happen in the kitchen and, e.g., prioritize actions like "turn on the television" way lower than "put teabag into cup". We call this type of planning a heuristic. It tells the planning algorithm where in the search tree it is most likely to find a solution and thus directs the search.

It should be noted that the notion of planning problem we described here is the most basic notion. There are a lot of extensions of the planning problem. Examples are:

- Side-conditions like finding the shortest path to a goal instead of any path

- Actions with properties. For example, an action can be associated with monetary or time costs, could be applicable only in certain states (e.g., pouring out from a tea can, is only possible if the tea can is not empty).
- Uncertainty. There may be uncertainty in the knowledge that makes up the state or in the effect of an action (i.e., the state transition function).

1.5.3 IntroAI Planning

Graph and Tree Search Problems In the context of AI and planning, problems are often represented as graphs or trees. Each node represents a state, and each edge represents a possible action or transition between states. The goal is to find a path from the initial state to a goal state.

Breadth-First Search (BFS) Breadth-First Search is a simple strategy where the root node is expanded first, then all the successors of the root node are expanded next, then their successors and so on. In other words, BFS expands nodes in a breadthward motion and uses a queue to keep track of the node to be visited and to remember the order. BFS is complete, meaning it will find a solution if one exists, but it's not always optimal.

Depth-First Search (DFS) Depth-First Search is another strategy that uses a stack instead of a queue. The root node is expanded first, but the next node to be expanded is the deepest unexpanded node. DFS goes deep into the tree with one path until it can't go deeper, at which point it retreats and tries a different path. Like BFS, DFS is complete, but it's not always optimal.

A Search* A* is a search algorithm that blends BFS's and DFS's advantages by avoiding expanding paths that are already expensive, but expands most promising paths first. It uses a heuristic function to estimate the cost to reach the goal from a particular node. The A* algorithm uses both the actual cost from the start to the current node and the estimated cost from the current node to the goal to choose the next node.

The A* algorithm is complete and it's also optimal, given certain conditions on the heuristic function. Specifically, the heuristic function must never overestimate the actual cost to reach the goal. This property is known as admissibility. For A* to guarantee finding the optimal solution, the heuristic must also be consistent (or monotonic).

Heuristics Heuristics are used to guide the search process towards the goal. They provide an estimate of the cost from a given node to the goal, helping the algorithm to select the most

promising path. In the context of A* search, good heuristics are problem-specific and should be both admissible and consistent.

A common example of a heuristic in a pathfinding problem like navigating a grid is the Manhattan distance, which calculates the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring any obstacles.

These are the basic concepts of BFS, DFS, A*, and heuristics. Depending on the specific problem and its constraints, one may be more suitable than the others.

1.5.4 Interleaving Planning and Execution

In many real-world scenarios, it is not feasible or efficient to come up with a complete plan before starting to execute actions, especially when dealing with dynamic environments that change over time or when complete information about the environment is not available. In these cases, an agent may need to alternate between planning and execution—a strategy known as interleaving planning and execution.

Interleaving planning and execution can provide several benefits. It allows an agent to start acting even when it has only a partial plan, which can be especially useful in time-critical situations. It also allows an agent to adapt its plans to new information it gains during execution.

In practice, an agent following this strategy would start by creating a partial plan, then execute the first few actions in this plan. After some execution, it would stop, observe the new state of the world, and adjust its plan based on these new observations. This cycle of planning, execution, observation, and replanning would continue until the agent achieves its goal.

1.5.5 Formulating a Planning Problem Based on the Knowledge Representation of an Agent

A planning problem can be formulated based on the knowledge representation of an agent as follows:

1. Initial State: This is the state that the agent starts in. The initial state is usually given as a set of facts that are true about the world.
2. Actions or Operators: These are the things the agent can do. Each action can change the state of the world in some way. Actions are usually defined with preconditions (things that must be true in order for the action to be executed) and effects (things that become true as a result of executing the action).
3. Goal Test: This is a way to determine whether a given state is a goal state. The goal test can be a single explicit state, or more commonly, it can be a set of states that satisfy some condition.
4. Path Cost Function: This is a way to measure the cost of a path. The cost can be the number of actions, the amount of time they take, the amount of resources they use, or any other quantifiable attribute. The objective is usually to minimize the path cost.

The knowledge representation of an agent is crucial in formulating a planning problem. It influences the definition of the initial state, the possible actions, the goal test, and the path cost function. For example, if an agent represents its knowledge using a propositional logic, the initial state and goal can be represented as a set of propositions, and actions can be represented as rules that transform these propositions.

1.5.6 Planning Under Uncertainty

In real-world environments, an agent often faces uncertainty. Uncertainty can arise due to noisy sensors, unreliable actuators, and a dynamic, unpredictable environment. When planning under uncertainty, an agent cannot predict with certainty the outcome of its actions, and it has to make decisions that maximize its expected performance, given the uncertainty it faces.

1.5.7 Markov Decision Process (MDP)

A Markov Decision Process (MDP) provides a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are widely used in optimization problems solved via dynamic programming and reinforcement learning.

An MDP is defined by four elements:

1. A set of states (S)
2. A set of actions (A)
3. A state transition function (T)
4. A reward function (R)

The state transition function T(s, a, s') defines the probability of transitioning to state s' when action a is taken in state s. The reward function R(s, a, s') gives the expected immediate reward received after transitioning from state s to state s', due to action a.

The objective in an MDP is to find an optimal policy, which is a function (s) that specifies the action that the agent will choose when in state s. The optimal policy is the one that maximizes the expected sum of rewards over all possible state sequences.

1.5.8 Partially Observable Markov Decision Processes (POMDPs)

In many practical situations, the agent cannot fully observe the environment. For example, a robot might have a limited field of view, or an online recommendation system might not know the true preferences of a user. In these cases, the agent faces uncertainty not only about the outcome of its actions, but also about the state of the world.

This kind of problem can be modeled as a Partially Observable Markov Decision Process (POMDP), an extension of MDP. A POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state.

In addition to the four elements of an MDP, a POMDP includes:

A set of observations (O) An observation function (O) The observation function O(s', a, o) gives the probability of making observation o after action a leads to state s'. Instead of knowing the exact state, the agent maintains a belief state, which is a probability distribution over all possible states, given the history of actions and observations. The agent's goal is to find an optimal policy that maximizes the expected reward over belief states.

Both MDPs and POMDPs provide a robust framework for planning under uncertainty, allowing agents to make rational decisions even when the environment is stochastic and possibly incompletely observable.

1.5.9 Advanced Planning

Advanced planning techniques go beyond the basic planning methods to handle more complex and realistic scenarios. They can deal with constraints such as time, resources, and abstraction levels. Two of these techniques are scheduling and hierarchical/refinement planning.

1.5.10 Scheduling (Planning in a Time-Based Domain)

Scheduling is a type of planning that takes time into account. It involves deciding when to execute certain actions within given time constraints. For example, consider a manufacturing process where different tasks require different amounts of time and some tasks cannot start until others have finished. The goal is to find a schedule of tasks that respects these temporal dependencies and possibly minimizes the overall completion time or maximizes the throughput.

One common method for solving scheduling problems is to represent them as constraint satisfaction problems (CSPs). Each task is a variable, each possible start time for a task is a domain value, and the temporal dependencies between tasks are constraints. Solving the CSP yields a schedule that satisfies all constraints.

Scheduling problems can also be represented as MDPs or POMDPs if there is uncertainty in task durations or outcomes. In this case, the goal is to find a policy that maximizes the expected reward or minimizes the expected cost.

1.5.11 Hierarchical/Refinement Planning

Hierarchical or refinement planning is a technique that deals with the complexity of planning problems by breaking them down into smaller, more manageable subproblems. The idea is to start with a high-level, abstract plan and gradually refine it into a detailed, executable plan.

In hierarchical planning, actions can be abstract or concrete. Abstract actions represent high-level tasks that themselves need to be planned, while concrete actions are directly executable. The planning process starts with a plan that consists only of abstract actions, and these actions are incrementally replaced with more concrete actions or sequences of actions until a fully detailed plan is obtained.

A common method for hierarchical planning is Hierarchical Task Network (HTN) planning. In HTN planning, each abstract action is associated with a set of methods that describe how to accomplish the action. Each method consists of a set of subtasks that need to be performed and possibly some constraints on how these subtasks can be ordered or interleaved. The HTN planning process involves choosing appropriate methods for the abstract tasks and ordering the subtasks to create a valid and executable plan.

These advanced planning techniques enable AI systems to handle larger and more complex problems than can be addressed with basic planning methods. They are widely used in areas such as manufacturing, logistics, project management, and autonomous robotics.

1.6 Reasoning

1.6.1 What is Reasoning?

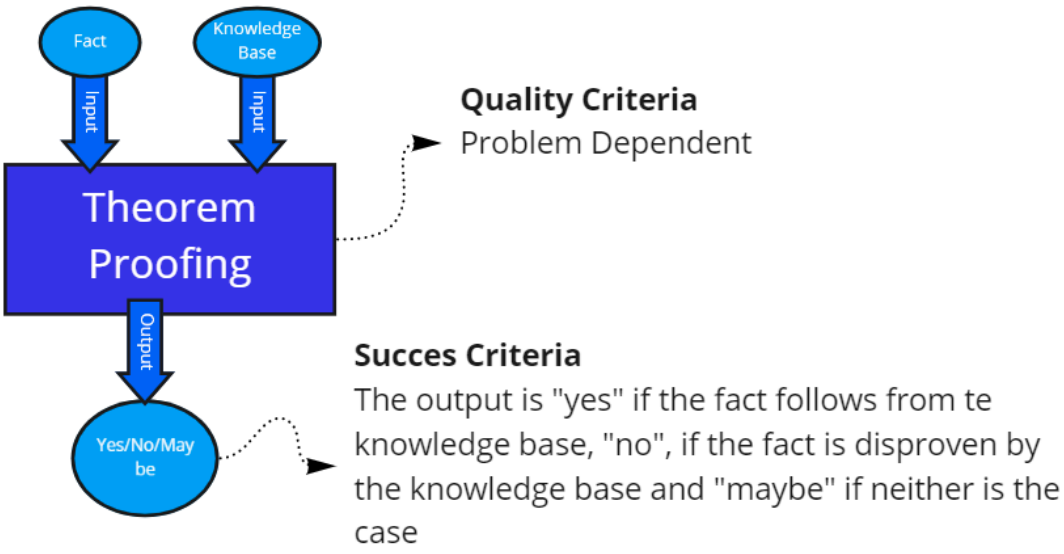
Reasoning is a technique that is closely related to the knowledge of an agent. It is concerned with the relation between individual pieces of knowledge. We call each piece of knowledge, a fact.

Examples of relations between facts are: - Mutual exclusion of negation: If I know a fact, the opposite cannot also be true. E.g., If my age is > 18 years, it cannot also be <= 18 years. - Mutually exclusive facts: Only one of a group of facts can be true at a time. For example, my favourite colour is pink, so it cannot be blue, green, red, etc. - Implication: If I know a fact, I can derive other facts from it. E.g., if I know that Waldi is a dog, it follows that Waldi can bark.

This means, individual facts can enable me to infer other facts. For example, if I know that you are a nineteen year old person that has a dog Waldi and the favourite colour black, then I also know that you are not underage, that pink is not your favourite colour and that your dog may bark.

In general, reasoning is the process of checking statements based on logical inference. Intuitively, this is a question answering of the type "is fact f true?". For example, I could ask "is your favourite colour red?" The answer to this question could be "yes", in case this can be inferred, "no" in case I can exclude red, because there is another favourite colour, or "maybe" in case neither one or the other applies.

We can call this the theorem proofing problem. The abstract theorem proofing problem is shown in the attached Figure. The input is the current knowledge, contained in a knowledge base (e.g., a collection of facts known to be true) and the fact we want to proof (also called a theorem). The output is "yes", "no", and "maye" and it is considered correct if the meaning described in the above paragraph applies. As usual, the quality criteria are highly application dependent and not prescribed here.



1.6.2 How to solve a reasoning problem?

A reasoning algorithm can be applied for different purposes. It can be used to find out, whether a statement is known to be true (searching for answer "yes"). It can also be used to find out whether a fact conflicts with what I already know (searching for answer "no"). Another way to use it is to identify gaps in my knowledge (searching for answer "don't know") in order to fill them.

Reasoning problems can be solved in multiple ways. One of the most naive is to check whether the fact is true/false in all situations that are consistent with the knowledge base. This approach is

called model checking. It hits it's limits when the amount of situations becomes big or ininite.

Another type of reasoning algorithm is closely related to planning. These algorithms apply a kind of state space search where the state space is a tree starting with the initial knowledge base as root and containing all extensions of the knowledge base that can be generated by inferring new facts and adding them to the knowledge. This is illustrated in the attached Figure. The initial state contains three facts: color(a) = pink (denoting that the favourite colour of a is pink), of_age(a)=true (denoting that a is of legal age) and dog(b)=true (denoting that b is a dog). By applying inference rules, it is possible to derive different facts in different inference steps. Adding these to the knowledge base leads to the three next states. And from these states we can go on inferring more knowledge.

The goal of the reasoning problem is to find a state that either contains the searched fact f or it's negation f. If this sounds familiar to you, then you've just discovered the close relation between reasoning and planning. Since reasoning spans a search tree, we can use planning algorithms to traverse this tree and find the goal state. It is possible to apply algorithms like depth first search or breadth first search to find a sequence of inferences to proof /disproof a statement. However, if we regard reasoning as one type of a planning problem, then there are also specific planning algorithms that can make better use of the properties of a reasoning problem.

Indeed, as with planning, machine learning and optimization, there are different variations of a reasoning problem as well. The two main differences are the knowledge representation and the inference system.

The knowledge representation is the language used to represent facts. There are several different formal logics that can be used as basis of reasoning. They differ in which kind of facts they can express, and which inferences they can support. The simplest and most widely used systems are propositional logics (enabling you to use negation, and, or, implication and equivalency) and first order predicate logics (additionally enabling the use of predicates and the use of existence and universal quantifiers). A lot of different extensions and variations of these logics exist for different purposes and domains. Examples are: - Type Theory: Enables reasoning about types and their properties. - Temporal Logic: Enables reasoning about the temporal order of events. - Agent-based logics: Enables reasoning about agents, their beliefs, intentions and actions. - Probabilistic logics: Enables the incorporation of uncertainty into facts.

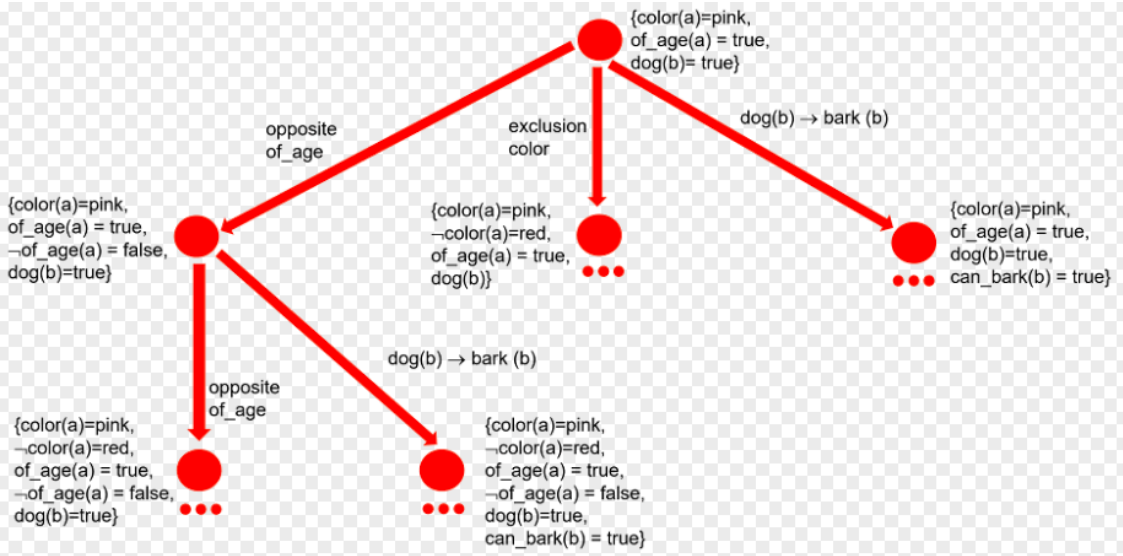
As a rule of thumb, you should always try to use a logic that is as simple as possible, but complex enough to express what you need it to express. The reason is that any additional complexity in the logic system may either take away things you can reason about (because an underlying assumption has been relaxed) or provide additional complexity to the reasoning process.

The inference system is your source of inference. It usually consists of rules of the form F1 -> F2, meaning that if fact F1 is known to be true then F2 is also true. This means, if F1 is in your knowledge base, then F2 can be inferred. Inference rules can originate from different sources. Some of them originate from the underlying logic system you use. A simple example is F1 and F2 -> F1, which means, if we know that F1 and F2 are both true then we also know that F1 is true. You can usually find this type of inference rules for a given logical system by searching for "inference rules", "implications" or "provable identities". These inference rules represent the formal workings of the used logical system.

These inbuilt logical inferences only go so far, because they can only represent the workings of the underlying formalism and are agnostic when it comes to the meaning of statements. For example,

our example inference rules for barking dogs needs knowledge that there is a thing called dog and that it likes to bark. Any formalism that does not have explicit concepts for dogs and barking cannot express this rule as an inbuilt inference rule. This often makes it necessary to express custom inference rules that are specific to the domain or application currently used. For example, dog(a) -> bark(a) is a statement in predicate logic that expresses our inference rule. These additional rules can be formulated once and then used like normal inference rules.

One thing that should be mentioned here is the relation to mathematical proofs. Often, what a mathematical proof does is showing that based on some assumptions, a certain statement holds. One way to do this is to use inference rules to show that you can arrive at the respective statement when you start out with the assumptions. Reasoning does something very similar, which is why the reasoning task described above is also sometimes called proofing.



1.6.3 Theory of Computation

The theory of computation is a branch of computer science that deals with how efficiently problems can be solved on a model of computation, using an algorithm. It is divided into three main sections: automata theory, computability theory, and complexity theory.

Automata theory involves the study of abstract machines (automata) and the computational problems that can be solved using these machines.

Computability theory focuses on whether a problem can be solved at all, regardless of resources, using algorithms. It explores concepts like Turing machines and the Church-Turing thesis.

Complexity theory, on the other hand, is not concerned with whether a problem can be solved but rather how efficiently it can be solved. It categorizes problems into complexity classes, such as P (problems that can be solved in polynomial time) and NP (problems for which a solution can be checked in polynomial time).

First-Order Predicate Logic First-order predicate logic (also known as first-order logic or FOL) is a system of formal logic used in mathematics, philosophy, linguistics, and computer science. Unlike propositional logic, which deals with simple true/false propositions, first-order logic allows

for the use of quantifiers (like "for all" and "there exists") and predicates (functions that return true or false).

In first-order logic, you can make statements like "For all x, if x is a bird, then x has feathers." This is more expressive than propositional logic, which might just have a proposition like "Polly is a bird" or "Polly has feathers."

Truth, Meaning and Proof In logic, truth, meaning, and proof are interconnected concepts. The truth of a statement refers to its correspondence to reality – a statement is true if what it asserts is the case, and false otherwise.

The meaning of a statement is related to the concepts and relationships it involves. In formal logic, the meaning is often determined by the interpretation of the symbols used in the statement.

A proof is a logical argument demonstrating that a specific statement is true in all cases covered by the assumed conditions. In other words, if you start with certain true statements (axioms) and apply rules of inference correctly, the statement you end up with will also be true.

True Statements and Provable Statements In logic, particularly in the context of formal systems, there is an important distinction between true statements and provable statements. A statement is true if it corresponds to reality. A statement is provable if it can be derived from the axioms of the system using the rules of inference.

Not all true statements are provable within a given system, a fact formalized by Gödel's incompleteness theorems. These theorems state, roughly, that for any consistent formal system that is strong enough to include basic arithmetic, there are true statements about the natural numbers that cannot be proved within the system. This highlights the limits of formal systems in capturing all mathematical truths.

1.6.4 Goal Trees and Problem-Solving

Goal Trees, also known as AND-OR trees, are used in Artificial Intelligence for problem-solving and decision-making purposes. These trees are a way to represent a problem space and the actions that can be taken to navigate that space. They are called AND-OR trees because they combine AND nodes, where all children must be achieved to consider the parent node achieved, and OR nodes, where achieving any child node is sufficient for achieving the parent node.

Let's break it down further:

1. Problem Space: This is the environment in which the problem exists, including all possible states that can be reached from the initial state by taking certain actions.
2. Goal State: This is the state that we want to reach. The problem is solved when this state is achieved.
3. Actions: These are the possible steps that can be taken from a given state. Each action transforms the current state into a new state.

In an AND-OR tree:

1. AND Nodes: These represent tasks that have multiple sub-tasks, all of which need to be achieved for the main task to be considered achieved. For instance, if the task is to "prepare

breakfast", it might be broken down into sub-tasks like "make coffee", "toast bread", and "scramble eggs". All of these need to be achieved to say that breakfast is prepared.

2. OR Nodes: These represent tasks that have multiple potential solutions, and any one solution is sufficient. For example, if the task is to "go to the airport", the solutions might be "take a taxi", "take a bus", or "ride a bike". Achieving any one of these is sufficient to achieve the parent task.

Problem-solving using Goal Trees involves the following steps:

1. Formulate the problem: Define the problem space, the initial state, the goal state, and the possible actions.
2. Construct the Goal Tree: Start with the goal state as the root of the tree. Break down the goal into sub-goals (using AND nodes) and alternative solutions (using OR nodes). Continue this process recursively until you reach actions that correspond to single steps in the problem space.
3. Search the Goal Tree: Use a search strategy to navigate the tree from the root to the leaf nodes. The search can be depth-first (exploring a path as far as possible before backtracking), breadth-first (exploring all paths one level at a time), or a more sophisticated strategy that takes into account the cost of actions or the estimated distance to the goal.
4. Extract a Solution: Once the search reaches a leaf node that achieves the goal state, the path from the root to this node constitutes a solution to the problem. This solution is a sequence of actions that leads from the initial state to the goal state.

Goal Trees are a powerful tool for structuring complex problems and finding solutions. They are widely used in AI, particularly in the areas of planning and decision-making.

1.6.5 Goal Trees and Rule-Based Expert Systems

Goal Trees (or AND-OR trees) and Rule-Based Expert Systems are two powerful concepts used in the field of Artificial Intelligence (AI) for reasoning and problem-solving. While they are distinct in their design and application, they can often work together to provide robust solutions for complex problems.

Goal Trees As discussed in the previous response, Goal Trees are a way to represent a problem space and the actions that can be taken to navigate that space. In a Goal Tree, each node represents a goal or sub-goal, and the edges represent the actions or decisions that lead to these goals. The nodes are connected in such a way that the achievement of the root goal depends on the achievement of one or more sub-goals, which can further depend on their own sub-goals, and so on.

Goal Trees can effectively structure complex problems, breaking them down into smaller, manageable parts, and they can guide the search for a solution through the problem space.

Rule-Based Expert Systems Rule-Based Expert Systems are a type of AI program designed to solve complex problems by applying a set of pre-defined rules. These rules are often based on the knowledge and experience of human experts, hence the name "Expert Systems".

A rule in such a system typically has an IF-THEN structure. The IF part represents a condition to be checked, and the THEN part represents an action to be taken if the condition is met. For

instance, a rule could be "IF the temperature is below 0 degrees, THEN start the heating system".

The system works by continuously checking the conditions of its rules and applying the actions of the rules whose conditions are met. This process is often guided by an inference engine, which controls the flow of execution and applies logical reasoning to make decisions.

Integration of Goal Trees and Rule-Based Expert Systems Goal Trees and Rule-Based Expert Systems can be combined to create powerful AI solutions. For instance, a Goal Tree can be used to structure the problem space and identify the goals to be achieved, and a Rule-Based Expert System can be used to make decisions about which actions to take to achieve these goals.

The rules in the Expert System can be defined in such a way that they guide the search through the Goal Tree, by taking actions that lead towards the achievement of the goals. The Expert System can also handle dynamic situations by updating the Goal Tree in response to changes in the environment, and it can incorporate uncertainty by assigning probabilities to the rules and making decisions based on these probabilities.

In conclusion, Goal Trees and Rule-Based Expert Systems are powerful tools for AI and reasoning, and they can often be combined to provide robust and flexible solutions for complex problems.

1.6.6 Depth-First Search (DFS):

DFS is a graph traversal algorithm that explores as far as possible along a branch before backtracking. It uses a data structure called a stack to remember the nodes to visit next. DFS can be implemented using recursion or an explicit stack.

Here's how DFS works: - Start at the initial node (root) of the graph. - Mark the current node as visited and push it onto the stack. - Explore the next unvisited adjacent node of the current node. - If there are no unvisited adjacent nodes, pop the top node off the stack and backtrack. - Repeat steps 2-4 until the stack is empty or the goal node is found.

DFS is particularly useful in scenarios where the solution is deep in the search tree and the total number of nodes is large, as it doesn't require visiting all nodes.

1.6.7 Hill Climbing:

Hill Climbing is a local search algorithm and a variant of the generate-and-test algorithm. It is an optimization technique used for solving problems where the goal is to maximize or minimize a given objective function.

Here's how Hill Climbing works: - Start at an initial state (a random or pre-defined starting point). - Evaluate the objective function for the current state. - Generate neighboring states by applying possible moves or transformations. - Evaluate the objective function for each neighboring state. - If a neighboring state provides a better solution (higher or lower, depending on the problem), move to that state and repeat steps 2-4. - If no better neighboring state is found, the algorithm terminates, and the current state is considered the optimal solution.

Hill Climbing is a greedy algorithm, meaning it always moves towards the direction of steepest ascent (or descent). This can lead to getting stuck in local optima, where the algorithm finds a suboptimal solution and cannot escape.

1.6.8 Beam Search:

Beam Search is a heuristic search algorithm that is an optimization of the Best-First Search algorithm. It is particularly useful when the search space is vast, and memory is limited. Beam Search maintains a fixed-size set of the most promising nodes, called the "beam," instead of storing all nodes in memory.

Here's how Beam Search works: - Start at the initial node (root) of the graph. - Generate the successors (children) of the current node. - Evaluate the heuristic function for each successor. - Select the top 'k' successors (where 'k' is the beam width) with the best heuristic values. - Expand the selected successors and repeat steps 2-4. - Terminate when the goal node is found or when no more successors can be generated.

Beam Search can be seen as a trade-off between exploration (searching the entire space) and exploitation (focusing on the most promising paths). By limiting the search to only a subset of nodes, Beam Search can save memory and time at the cost of potentially missing the optimal solution.

In summary, Depth-First Search, Hill Climbing, and Beam Search are search techniques used in AI for problem-solving and reasoning. DFS is a graph traversal algorithm that explores deep paths, Hill Climbing is a local search algorithm for optimization problems, and Beam Search is a heuristic search algorithm that balances exploration and exploitation while saving memory.