A.I Study Notes

Eduart Uzeir

1 Introduction

In this chapter we are going to see the concept of **Intelligence** that characterizes creatures like *Homo Sapiens* and the rising field of **Artificial Intelligence**.

1.1 What is A.I?

The definition of **Intelligence** involves other concepts like *thought process*, *reasoning*, *behaviour* and *rationality*.
We can look A.I in four ways:

- **Acting Humanly** - The *Turing Test* approach.
  In order to pass the Turing Test a computer needs to possess the following 6 capabilities:

  1. *Natural Language Processing*: The computer has to be able to communicate in a human language, e.g English.
  2. *Knowledge Representation*: The capacity to store what it knows and what it hears.
  3. *Automated Reasoning*: The ability to draw conclusions from the stored and the new information it receive.
  4. *Machine Learning*: To adapt to new circumstances and to explore patterns.
  5. *Computer Vision*: To perceive objects.
  6. *Robotics*: To manipulate objects and to move.

Eduart Uzeir
Universita' di Bologna, e-mail: eduart.uzeir@studio.unibo.it

- **Thinking Humanly** - The *Cognitive Modelling* approach.
  In this approach the *Cognitive Science* is used. If we are going to say that a given program *thinks* like a human, we must have some way to determine, how the humans think. We have to get inside the actual working of the human mind.
  There are three ways to do this:

  1. *Introspection*: Trying to catch our own thoughts as they go by.
  2. *Psychological Experiments*: Observing a person in action.
  3. *Brain Imaging*: Observing the Brain in action (EEG).

  When we have enough information about how the *mind* works we can express it into a computer program.
- **Thinking Rationally** - The *"Laws of Thought"* or *Logicist* pproach.
  Use of *Logic* to create a program to solve any problem.
  We say that a system is *Rational* if it does the **"Right Thing"**, (the ideal choice) given what it knows.
  There are two problems in this approach:

  1. Is not easy to take informal (language), knowledge and to convert it to formal knowledge used by the Logic.
  2. There is a big problem (difference) between solving a problem in principle and solving it in practice.

- **Acting Rationally** - The *Rational Agent* approach.
  Agent: Something that **act's**.
  A **Rational Agent** is one that acts so as to *achieve the best outcome*, or when there is uncertainty the *best expected outcome*.
  The **Rational Agent** model is the most suitable one for our purpose.

1.2 Why A.I is a new branch and not a Math branch?

There are three reasons why A.I is a new branch of science.

1. From the start A.I embraced the idea of duplicating human faculties such as: *creativity*, *self-improvement* and *language* use. No other field were addressing this issues.
2. The methodology used is related with Computer Science and Computer Simulation.
3. A.I is the only field that wants to build a machine that will function autonomously in complex and changing environments.

The year of birth of A.I is 1956 in Dartmouth. From then on it has pass different stages of evolution and now days is one of the most important fields in the science and technology.

1.3 Further Study

This section refers to study material obtained from other sources and/or provided by the professor.

1.3.1 Some definitions of A.I

Officially A.I is defined: *"The ability of a computer to perform tasks commonly associated with intelligent beings."*
Peter Norvig defines is as: *Knowing what to do when you don't know what to do.*
The difference between A.I and Computer Science is that, in the case of A.I we may not be able to observe everything in our environment or we are not sure about the result of our actions. On the other hand we use Computer Science to solve problems that we know how to solve.
Keep in mind that the definition of Natural Intelligence is not an easy task because there are many types of intelligence and many areas of science that study it from different perspectives. Examples such as: Is more intelligent a human or a pigeon? In a first approximation is easy to ask but there are some task that pigeons do better than humans.
The learning ability is one of the most important faculties of an intelligent organism. We can find in Nature many examples of organisms that are better than humans to learn things such as a simple motor response.
Does the Brain dimension characterize more intelligent creatures from less intelligent ones? In order to ask this question we have to consider the size of brain and the number of neurons for each intelligent being. We can argue that the size of the Brain, the number of neurons and synapsis is a good indication of intelligence. The human Brain weights around 1.4 Kg and has 85 Billion neurons.
Here is another definition of Intelligence: *"Intelligence is the ability to learn from experience, to apply knowledge to solve problems, and to adapt and survive in different environments (social and geographical).".*
In this definition, **Intelligence = Survival**.
Spearman (1923) used the term *G-factor, General Intelligence factor* for the global cognitive abilities of a person such as mathematical and linguistic abilities.
Guilford (1967) claim that there is no general intelligence.
Thurstone (1930) defines seven factors that describe Intelligence:

1. Verbal Fluidity (speak fluently a human language)
2. Numerical ability (being able to make simple numerical calculations)
3. Inference (the ability to infer conclusions using a possessed knowledge)
4. Spacial ability (the ability to move and orient in the space)
5. Velocity in perception
6. Memory (the ability to memorize information)

1.3.2 Principal Technologies and Applications of A.I

Hard Computing - computation based on symbol manipulation via structured rules. This kind of computation has three characteristics: is *consciousness, symbolic, high-level*.

- Rule-Based Systems (Optimization and Search)
- Logic and Constraint Programming
- Probabilistic Computation

Soft Computing - computation seen at mental process based on simple units that connects together creating networks. This kind of computation is *unconsciousness, sub-symbolic, connectionism (Hebb)*.

- Artificial Neural Networks (ANN)
- Swarm Intelligence

Here are some topics and techniques in A.I:

- **Knowledge Representation** - is the field of A.I that is dedicated to representing the information in a way that a computer system can use in order to solve problems and execute tasks. Three concepts closely related with Knowledge Representations are *Logic, Ontholoy, Sub-symbolic*.
- **Problem Solving** - generic ad hoc techniques used to find solution to problems. *Search, Optimization, Adversarial search in games, Constraint Satisfaction Problems*.
- **Reasoning** - is a faculty of intelligent agents that consists on making sens of things, using logic to derive conclusion based on initial knowledge. *Inference, Prolog and CLP, planning*.
- **Uncertain Reasoning** - reasoning in the conditions of uncertainty. *Probabilistic Reasoning, Markov Models, Bayesian Models*.
- **Neural Networks** - Connectionistic technologies based on simple units able to create a processing systems when connected together. *Feed-forward Networks, Recurrent Neural Networks, Deep Learning*.
- **Machine Learning and Data Analytics** - Technologies based on neural networks that learn patterns using vast amounts of data. *Supervised, Unsupervised, Renforcement Learning*.
- **Communication** - *Natural Language Processing, Human-machine communication, Automatic translation*.
- **Perception** - *Acoustic and Visual perception*.
- **Robotics** - *Sensors, Actuators, Automatic decision making*.
- **Expert System** - computer system able to emulate the human capacity of decision making. *Medicine, Finance*.
- **Natural Programming** - programming paradigms based on natural and evolutionary phenomena. *Genetic algorithms, Ant computing*.
- **Cognitive modelling** - *Cognitive states, Emotions*.

1.3.3 A.I and Logic

Logic has played a very important role in the evolution of A.I.
There are many different types of Logic, *(Propositional, First Order, Modal...)* and many techniques to represent knowledge and reasoning *(Deduction, Induction, Abduction...)*. Also we have at our disposition many tools and languages like *Prolog, Lisp, Constraint Programming etc*.
Deduction: is a reasoning process that allows to derive consequences of the assumed. B can be derived by A if B is a logical consequence of A ($A \models B$). Given the truth of the assumption A follows the truth of the conclusion B.
Introduction: inductive reasoning allows to infer B from A where B does not follow necessarily from A. The premises are viewed as providing strong evidences for the truth of the conclusion.
Abduction: is a form of logical inference that starts with an observation or a set of observations and seeks to find the most simplest and most likely explanation for the observations. If we have a theory T and a set of observations O, abduction is the process of deriving a set of explanations of O according to T and picking one of those explanations E. Formally:

- O follows from E and T :: $T \cup E \models O$
- E is consistent with T :: $T \cup E$ **is consistent**

Expert System: are Knowledge Based System that are used to solve problems in a specific and limited domain. Have very good performance, similar to a human expert on the domain. Are able to explore large databases of information and infer conclusions. The solutions are build dynamically. The search and solution building process is based on Rules.

1.3.4 A.I today and other related aspects (ethics)

Now day A.I is used in many important aspects of the life. In industry, media, medicine, finance etc. This fact has rise many question about ethics in A.I. Knowing that A.I can outperform people in many areas given enough training data and time, do we have to limit this progress in order to keep it under control. If you want to learn more abut this new perspectives of A.I, please read the book, *"Life 3.0 - Being in Human in the Age of A.I"* by Max Tegmark.
The trend of investing resources in A.I by companies is increasing rapidly. The big high-tech giants like Google, Amazon, Microsoft and others have acquired many A.I specialized companies.

1.3.5 A brief overview of A.I

A.I as we know it now started at a conference in 1956 in Dartmouth. A group of leading researchers like John McCarthy, Marvin Minsky, Claude Shannon and other

started the new field. The first period was characterized by great expectations but later on many problems appeared. The evolution of A.I passed through different stages of evolution until the 80s and 90s when it gained a lot of interest due to the large progress in computational performance and big amount of data to work with. New technologies emerged such as Machine Learning, Deep Learning ... and the rest is history as they say!

- **Rene Descartes (1596-1650)**: is one of the first philosophers that studied the connection between the body (material) and mind (non material). Is often viewed as the forerunner of the modern cognitive science and A.I.
- **Leibniz (1646-1716)**: created a machine able make numerical calculations.
- **Alan Turing**: Universal Turing Machine, the first general purpose universal computer able to compute any type of function. The book *"Intelligent Machinery"*, 1948 is considered a real A.I manifesto. The book *"Computing Machinery and Intelligence"* published 1950 includes the *"Imitation Game"* also called the *Turing Test*.
  Turing Test: If it is impossible to distinguish the behaviour of a machine from that of a human, then the machine is intelligent.
- **McCulloch and Pitts-1943**: creation of the first computational model of neuron.
- **Hebb-1949**: formulate a theory of how the neurons adapt in the learning process.
- **Rosenblatt-1958**: proposed the *"Perceptron"* model. A pattern recognition algorithm based in machine learning.
- **Dartmouth-1956**: Official start of A.I

1.3.6 State of the Art

Robotic Vehicles, driverless robotic cars able to move in the streets using sensors, actuators and machine learning techniques.
Speech Recognition, are system that can recognize, understand and synthesise human voice. This system are vastly used today in many contexts. Honorable mentions are Siri, Alexa ect.
Autonomous Planning and Scheduling
Game Playing in A.I outperform expert humans in many intellectually demanding games such as Chess and Go.
Spam Detection have achieved very good performance using machine learning and deep learning techniques.
Logistic Planning with the use of special A.I techniques like Constraint Programming or Heuristic Search has achieve very good performance on optimization problems of large size.
Robotics is growing rapidly.
Machine Translation is now able to perform real time translation in almost all the human languages.

1.4 Concepts to Remember

⇒ **Intelligence**
⇒ **Artificial Intelligence**
⇒ **Rationalism**
⇒ **Agent**
⇒ **Rational Agent**
⇒ **Turing Test**
⇒ **Logic Programming**
⇒ **Inductive Reasoning**
⇒ **Deductive Reasoning**
⇒ **Abductive Reasoning**
⇒ **Expert Systems**

2 Intelligent Agents

In this chapter we are going to define some of the most important concepts in A.I and their characteristics.

2.1 Agents and Environments

An **Agent** is anything that can be viewed as perceiving its **Environment** through **Sensors** and acting upon that **Environment** through **Actuators**. A Robot has cameras and other instruments like sensors and maybe automatic arms for actuators.
Percept is called the Agent's perceptual input at any given instant.
Percept Sequence is the complete history of everything the Agent has ever perceived.
An Agent's choice of action at any given instance can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived.

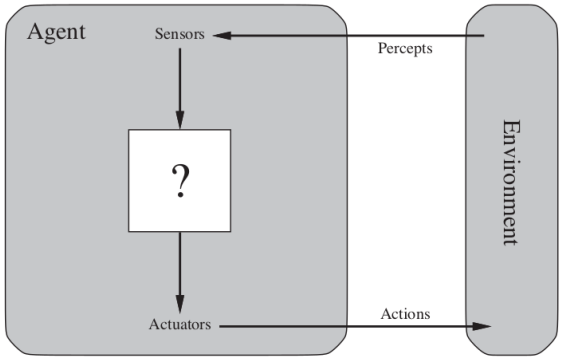

Fig. 1 Agents interact with environment through sensors and actuators.

The Agent's behaviour is described by the **Agent Function** that maps any given **percept sequence** to an action. We can characterize the the Agent Function in two ways:

- Externally, in a tabulate form, something that in reallity is not possible because the tables will be very large.
- Internally, as an **Agent Program** that is the concrete implementation of the Agent Function.

2.2 Rational Agent

Rational Agent is the one that does the right thing, in other words the table of the **Agent Function** is filled correctly. Now, what do we mean with the sentence: "The Agent does the *RIGHT* thing!". We have to consider the *Consequences* of the Agent's behaviour. When an Agent is placed inside an Environment it generate a sequence of actions (sequence of states in the Environment) that are based on the sequence of percept it receive. Now if the resulting actions are *desirable* we say that the Agent performs well.
The notion of **Desirability** is captured by the **Performance Measure**. We evaluate the Environment states and NOT the Agent state.
The **Performance Measure** evaluates any given sequence of Environment states.
General Rule : *Is better to design Performance Measures according to what one actually wants in the Environment, rather than according to how one thinks the Agent should behave.*
Rationality: What is Rational in a given time depends on:

1. The Performance Measure that define the criterion of success.
2. The Agent's prior knowledge of the Environment
3. The actions that the Agent can perform
4. The Agent's percept sequence to date

Definition 1. (Rational Agent): *For each possible percept sequence a Rational Agent should select an action that is expected to* ***maximize*** *its performance measures, given the evidence provided by the percept sequence and whatever build-in knowledge the Agent has.*

Omniscience: Knowing the actual outcome of an action and act accordingly.
Rationality vs. Perfection: Rational Agents maximize the expected performance while a Perfect Agents maximizes the actual performance.
Information Gathering: doing actions in order to modify future percepts.
Exploration: is a kind of Information Gathering to be performed by an Agent that does not know initially the Environment.
The Agent has not only to be able to gather information but also to **Learn** as much as possible from the percept.

2.3 The Nature of the Environment

We can group all the information we need in order to specify an Rational Agent under the concept of Task Environment. **Task Environment** are the problems to which the Rational Agents are the solution.

Definition 2. *Performance, Environment, Actuators, Sensors (P.E.A.S) = Task Environment*

The first thing to do when designing an Agent is to define as fully as possible P.E.A.S. For example in an autonomous taxi driverless vehicle the P.E.A.S should be:

- Agent Type: Taxi Driver
- Performance Measure: Safe, Fast, Legal, Confortable ...
- Environment: Roads, Pedestrians, Traffic...
- Actuators: Steering, Accelerator, Break...
- Sensors: Cameras, Sonars, GPS...

Here are listed some of the properties of the Task Environment.

- How much observable is the Environment (Fully, Partially, Not Observable).
- Single or Multi-Agent Environment: In the case of multi-Agent Environments we have to deal with concepts like *Competition, Cooperation, Communication, Randomized Behaviour*.
- Deterministic vs. Stochastic: We say that an Environment is Deterministic when the state of the Environment is completely determined by the current state and the actions executed by the Agent, otherwise is Stochastic.
- Uncertain: is the Environments is not fully observable or not deterministic.
- Nondeterministic Environments are those Environments where actions are characterized by their possible outcomes but no probabilities are attached to them.
- Episodic Environments: Agent's experience is divided into atomic episodes. Next episodes does not depend on the actions taken in previous episodes.
- Sequential Environments: here current decisions can influence future decisions (Chess playing).
- Other distinctions are: Static / Dynamic, Discrete / Continuous etc.

In conclusion, the hardest case is: *partially observable, multi-Agent, stochastic, sequential, dynamic, continuous and unknown Environments.*. Taxi driving is an example.

2.4 The Structure of the Agents

The job of A.I is to design an Agent Program that implements the Agent Function - the mapping from percept to the actions. This program will run on a physical part (sensors, actuators etc) that we call **Architecture**.

Agent = Architecture + Program

There are 4 basic kinds of Agent Programs:

1. Simple Reflex Agents: Agent select actions on the basis of current percept, ignoring the rest of the percept history. Usually this type of Agents are implemented using a set of *condition-action rules*, that practically are viewed as a set of *if-then* conditions. Simple Reflex Agents are very simple and of limited intelligence. This kind of Agents require that the Environment is fully observable.

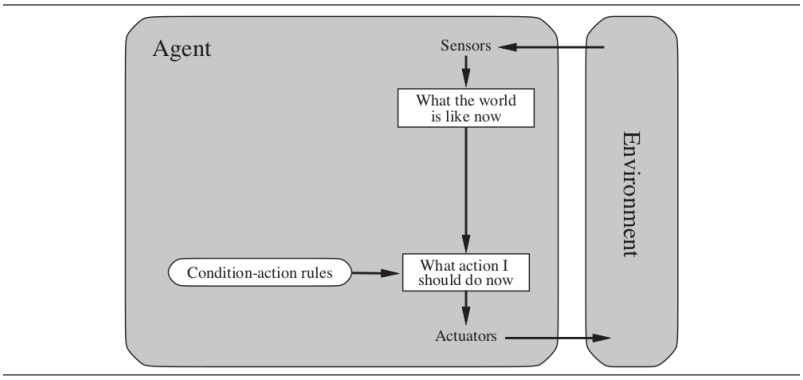

Fig. 2 Simple Reflex Agent.

2. Model-based Reflex Agent: Agent has to keep track of the part of the World it can't see now (It has to maintain an internal state that depends on the percept history in order to reflect some of the unobservable aspects of the current state). The Agent has to have a model of the World. This model contain two types of knowledge:
   - How the World evolves independently from the Agent.
   - How the Agent actions affect the World.

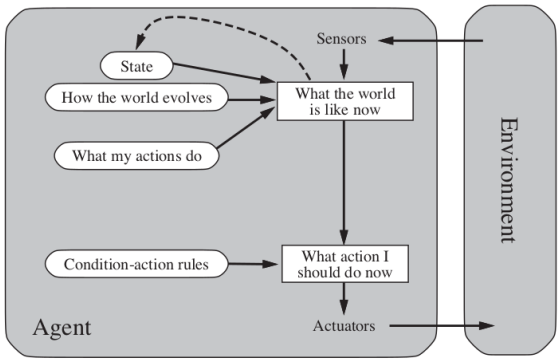

Fig. 3 Model-Based Reflex Agent.

3. Goal-based Agents: This type of Agents have a goal to achieve and can change their rules based on that goal.
4. Utility-based Agents: use an *"utility function"* as a performance measure. If the Agent has many goals it has to maximize the utility and choose the best of them.

Any **Learning Agent** has to have this four conceptual components.

1. Learning Element that is responsible to make improvements.

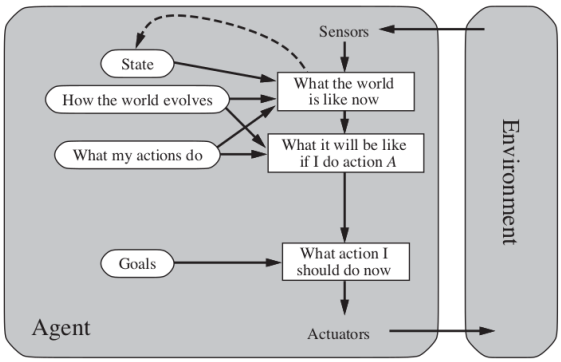

Fig. 4 Goal-Based Agent.

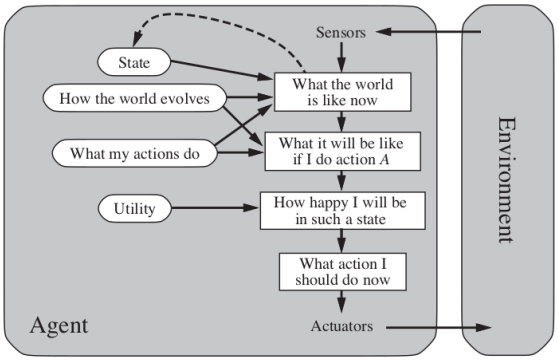

Fig. 5 Utility-Based Agent.

2. Performance Element that is responsible for selection external actions
3. Critic that evaluates the action's performance
4. Problem Generator that suggest actions that can lead to new experiences

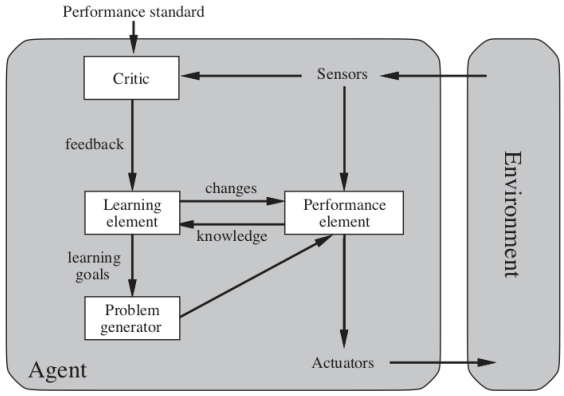

Fig. 6 Utility-Based Agent.

2.5 Concepts to Remember

⇒ **Definition of Rational Agent**
⇒ **Agent Function**
⇒ **Performance Measure**
⇒ **Task Environment**
⇒ **Agent Program**
⇒ **Simple Reflex Agent**
⇒ **Model-based Reflex Agent**
⇒ **Goal-based Agent**
⇒ **Utility-based Agent**
⇒ **Learning**
⇒ **Percept and Percept Sequence**
⇒ **Utility Function**
⇒ **Goal**

3 Solving Problems by Searching

In this chapter we are going to see the notion of **Problem-solving Agent** that is a simple Goal-based Agent.

3.1 Problem-solving Agents

As mentioned previously Intelligent Agents are supposed to maximize their performance measure, and this is easier to achieve if the Agent adopt a **goal** and try to satisfy it.
The first thing that the Agent should do is to define the goal, this process is called **goal formulation**.
Goal formulation, based on the current Agent's situation and the performance measure is the first step in the problem solving. The goal is a set of Word states that satisfies the goal.
The second thing to do given a goal is the **problem formulation**. Problem formulation is the process of deciding what actions and states to consider, given a goal.
Third, assuming that the Environment is deterministic the solution to any problem is a fixed sequence of actions.
The process of looking for a sequence of actions that reaches the goal is called **Search**.
A **Search Algorithm** takes a problem as input and returns a **solution** in the form of action sequence.

1. formulate a **goal**
2. formulate a **problem** to solve
3. solve the problem using **search**
4. **execute** the solution
5. formulate another goal
6. go back to step 1

A **problem** can be defined formally by five components:

1. **Initial state** *(In(s))*: where the Agent starts.
2. **Actions**: a description of all possible actions available to the Agent. Given a state *s*, *ACTIONS(s)* return the set of actions that can be executed.
3. **Transition Model**: is a description of what each action does, *RESULT(s,a)*. This three components (Initial state, Actions and Transition model) defines the **state space** of the problem and has the form of a **Graph**. A **Path** is the graph is a sequence of states connected by sequence of actions.
4. **Goal test**: determines whether a given state is a goal state or not.
5. **Path cost**: is a function that assigns a numeric cost to each path. *C(s, a, s')* is called a **step cost** and is the cost of the action *a* to take the Agent from the state *s* to *s'*.

A **Solution** is an *action sequence* that leads form the *initial state* to a *goal state*. An **Optimal Solution** is the solution with the lowest *path cost* among all the possible solutions. **Abstraction** is the process of removing details from a representation.

3.2 Searching for Solutions

Having formulated a problem we have now to solve it, and the solution is an *action sequence*. This sequence starts at the initial state and form a **search-tree**.
The search starts from the **root** state and at each state called **node** decide which action to execute. This is called **expansion**. The way an algorithm choose which node to expand is based on **search strategies**. Is called **frontier** the set of all nodes available for expansion at any moment.
There are four elements used to evaluate the performance of an algorithm:

1. Completeness: is the algorithm guaranteed to find a solution when there is one?
2. Optimality: Does the strategy find the optimal solution?
3. Time complexity: How long does it take to find a solution?
4. Space complexity: How much memory is needed to perform the search?

To assess the effectiveness of a search algorithm usually is considered only the **search cost** that typically depends on time complexity. We can group the search strategies in **Informed** and **Uninformed**.

3.3 Uninformed Search Strategies

Uninformed search strategies have no additional information about the states beyond that provided in the problem definition. The only thing that this strategies can do is to generate states (expand nodes) and to distinguish a goal state from a non-goal state.

3.3.1 Breadth-first search

Breadth-first search (BFS) expands the all the nodes at a given level before expanding any node of the next level. A **FIFO queue** is used to implement the algorithm.

- Completeness: yes, find always the goal node given a finite depth (d) and a finite branching factor (b).
- Optimality: not always, depend on the path costs, if the path cost is the equal then is optimal.

- Time complexity: $b+b^2+b^3+...+b^d = O(b^d)$, where b is the branching factor and d depth of the tree. As we see the time complexity is exponential and this is bad.
- Space complexity: is exponential too, $O(b^d)$. This is terrible, consider *d=16* and *b=10* then the space needed would be 10^{16} = 10 exabyte and 365 years.

First, the Memory requirements are a bigger problem for breadth-first search than is the execution time.
Exponential complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

Here is an example that shows how BFS works.

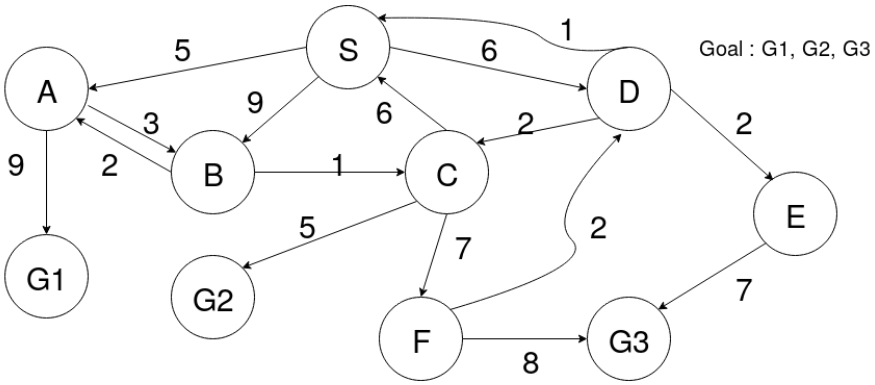

Fig. 7 Search Space of the example.

The purpose of BFS is to find the shortest path that is not necessarily the least costly one. I examine the start state **S** if it is the goal state i stop and return it otherwise i expand **S** and consider the new states from left to right. In this case the new states are **A, B** and **D**. I start from the leftmost one **A** and check if it is the goal. The answer is **No**, so i expand it. From the expansion of **A** i reached to the new states **B** and **G1**. Even though **G1** is a goal state i have to ignore it for the moment because i have to check the state **B** in the previous level. May be the case that **B** in the second level is a goal state and knowing that i am not interested about the cost here but only the shortest path i have to consider all the elements on the level *n* before going down to the level *n+1*. When i finish to examine all the nodes of level *n* i go to the next level and consider the first node on the left of the new level, expanding it and so on. In this case after expanding **B** on the third level, *(d=2), starting from d=0* we go to the node **G1** and see that we find state **G1** that is a goal state and is the one to return. The shortest path in this example is: $S-->A-->G1$.

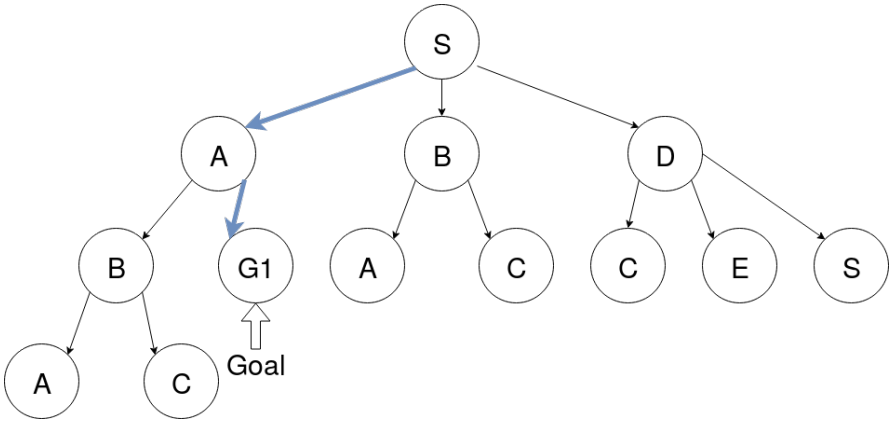

Fig. 8 Breadth-first search.

3.3.2 Uniform-cost search

Instead of expanding the shallowest node like BFS, expands the node *n* with the lowest path cost *g(n)*. This is done by sorting the frontier within a **priority queue** ordered by *g*. Here *g* is the lowest path function.
The Uniform-cost search expands nodes in order of their optimal path cost.
Uniform-cost search is characterized by the path cost rather than the depth and branches. It does not care about the number of steps in the path but only about the total cost. The example shows how the algorithm behave in the search space below. The goal is to reach one of the goal states, G1, G2 or G3 with the minimum cost. Every edge has a cost.

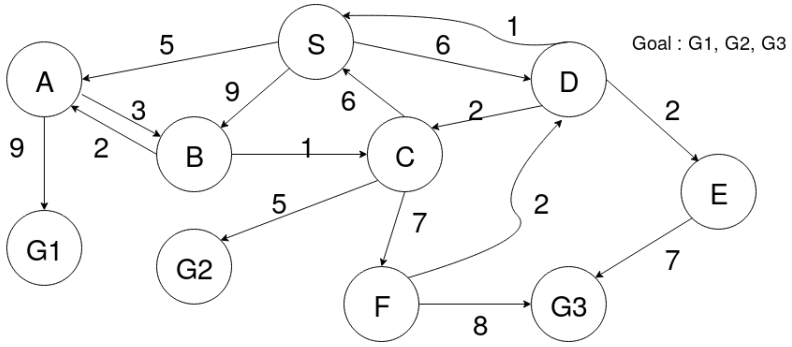

Fig. 9 Search Space of the example.

The algorithm starts at the starting state **S** and control if this state is a goal state. If **YES** then it stops and return the state otherwise expand the state **S** with all the states that we can get to from it. As we can see in the search space we can get to

the states **A**, **B** and **D** from the state **S**. Each time that we expand a node, we say that it is *visited* and we insert it in the *visited* list. At this point we calculate the total cost of each of the three new paths and pick the one with the lowest cost to expand next. Here state **A** has the lowest cost (5) so we expand it and insert **A** in the *visited* list. Now we calculate the total cost of each new path. At this point even though we reached **G1** we do not except it as final state because we have to see if there is other path leading to goal states that have lowest cost. At this point we have four active paths **B** with cost of 8, **G1** with cost 14, **B** with cost 9 and **D** with cost 6. As you can see the cheapest one is **D** and that is the one to be expanded. If a state has been visited we do not considered it anymore neither put it twice in the visited list. In the case there are more than one paths with same cost we pick one randomly or with using some policy, alphabetic order for example.
We proceed in this fashion until find the goal state with the lowest path cost. The example below show the search.

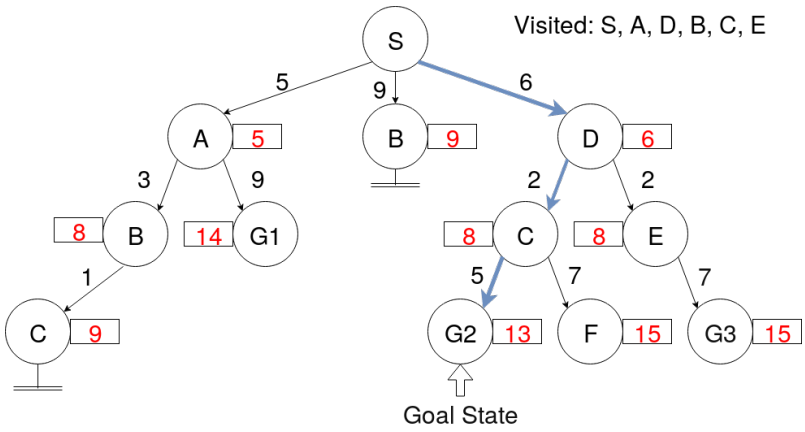

Fig. 10 Uniform Cost Search.

The algorithms stops returning the goal state with the optimal cost that in our case is **G2**.

- Completeness: possible if the steps cost is always positive, there are no loops.
- Optimality: is optimal because it expands the nodes based on their optimal path cost.
- Time complexity: $O(b^{1+\frac{c}{\varepsilon}})$, where c is the cost of the optimal solution and ε a small positive constant.
- Space complexity: the same as time complexity.

3.3.3 Depth-first search (DFS)

DFS always expands the deepest node in the current frontier of the search tree. DFS is implemented using a **LIFO queue**.

- Completeness: no in case of infinite state space.
- Time complexity: $O(b^m)$ where m is the maximum depth of any node.
- Space complexity: $O(b*m)$ where b is the branching coefficient and m the maximum depth of any node.

There is a variant of DFS called **Backtracking search** that has a space complexity equal to $O(m)$.

3.3.4 Depth-limited search

The problem of DFS with the infinite state space can be solve by putting a limit l on the depth of the tree. In this case nodes at level l are considered to have no children. This approach is called **Depth-limited search**.

- Completeness: if $l > d$ yes otherwise if $l < d$ no.
- Optimality: Depend on the relation between l and d.
- Time complexity: $O(b^l)$
- Space complexity: $O(b*l)$

3.3.5 Iterative deepening depth-first search

This algorithm increase the depth limit until it finds the goal state.

- Time complexity: $O(b^d)$.
- Space complexity: $O(b*d)$.

Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is unknown.
There is a variation of this algorithm that instead of increasing the depth increase the path cost limit. This algorithm is called **Iterative Lengthning search**.

3.3.6 Bidirectional search

Is a search algorithm that consists of two parallel searches one that starts form the initial node and the second starts at the goal node. The algorithm stops when the two searches meet.

3.4 Informed (Heuristic) Search Strategies

Informed search strategies use problem-specific knowledge beyond the definition of the problem in order to find a solution in an efficient way. The general approach is called **Best-first search**. Here a node is choose for expansion based on an **evaluation function**, *f(n)*. Now, the choice of f determine the search strategy. Most of best-first algorithms include as a component of f a **heuristic function**, *h(n)*.
h(n) = estimate the cost of the cheapest path from node *n* to the goal state.
The performance of heuristic search algorithms depends on the quality of heuristic function. One can sometimes construct good heuristics by relaxing the problem definition (applying a process called relaxation), by storing pre-computed solutions costs for sub-problems in a pattern database, or by learning from experience.

3.4.1 Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. It evaluates nodes by using just the heuristic function:

***f(n) = h(n)**, where h(n) is an approximation from state n to the goal state.*

For example in the case we want to go from a city to another passing from different other cities we may choose as a heuristic function *g(n)* the straight-line distance between the cities. Here is a numerical example.

Here the heuristic function *h(n) = "straight-line distance to the goal"*.
The path toward the goal is the sequence of nodes: 1, 2, 6, 9 that is the optimal solution to this problem. It is called **Greedy** because at each step it get as close to the goal as possible.

- Optimality: the search cost is minimal but not optimal.
- Completeness: the algorithm is incomplete even in finite state space because there is the possibility that it gets stuck into a dead end.
- Time complexity: $O(b^m)$ where m is the maximum depth and b is the branching factor.
- Space complexity: $O(b^m)$.

In the Greedy Best-first search the heuristic function is of extreme importance.

3.4.2 A* search

Is the most known Best-first search. It evaluates nodes by combining *g(n) = cost to get to node n* and *h(n) = cost to get form node n to the goal node*.

***f(n) = g(n) + h(n)**, "estimates the cost of the cheapest solution through n".*

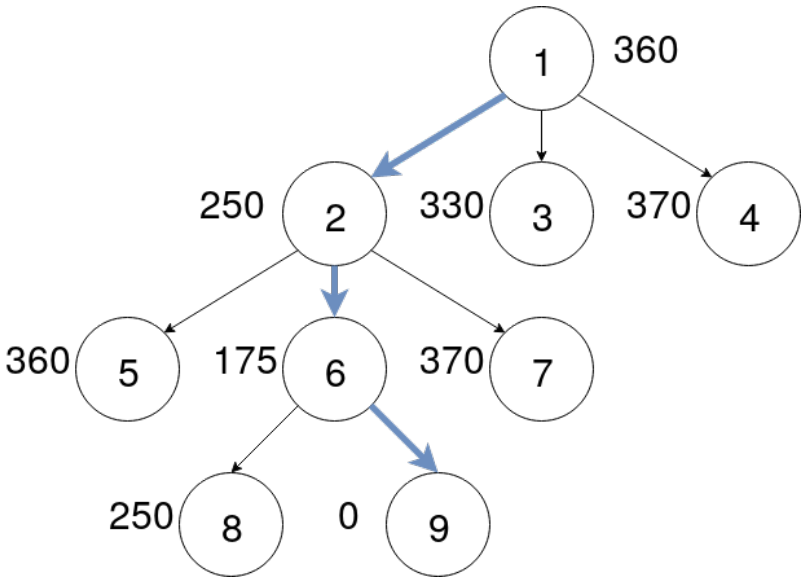

Fig. 11 Greedy Best-first Search.

In order to have optimal solution we need to use heuristic that never overestimate the cost, so either underestimate it or use the exact value.

Given the search space shown below:

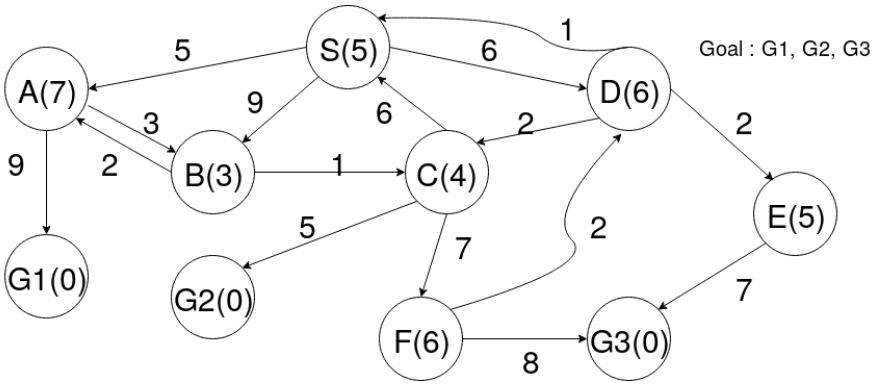

Fig. 12 Search Space of the example.

Here is an example of how **A*** works:

The algorithm A* is a modified (an informed) version of Uniform-cost search. At each state is associated a number that represent and estimation that we have about the cost from that state to one of the goal states. If in the search we find again a visited state with the lowest A* value we explore it otherwise if the A* value of the

new state is bigger that the one we have visited we just ignore it and prune that part of the search. At the end of the execution the algorithm return G2 as the goal state with the lowest cost. As we see the algorithm made a lot of work to find this solution because the heuristics were not accurate. If you look the estimation (heuristic) of starting state **S** was only 5 and we learned that the best cost to reach a goal state is 13. We conclude that even though the heuristic we used were **admissible** their accuracy was very low, underestimating the actual value.

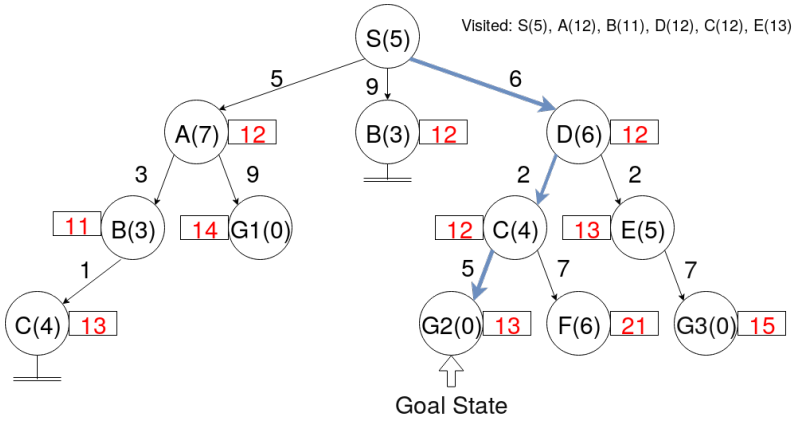

Fig. 13 A* search.

Conditions for Optimality

- Admissible Heuristic: An Heuristic is called admissible if it **never overestimates** the cost to reach the goal. This heuristics are called *optimistic* because they think that the cost for solving a problem is always less than it actually is. For example a straight-line distance between 2 points is an admissible heuristic because actually the straight-line is the smallest distance between 2 points.
- Consistency (Monotonicity): Is a strong condition required only for A* graph search. A heuristic *h(n)* is consistent if for every node *n* and every successor *n'* of n generated by any action *a*, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to *n'* plus the estimated cost of reaching the goal from *n'*. This condition is also called **triangle inequality** and states that each side of the triangle cannot be longer than the sum of the other two sides.

$$h(n) \leq c(n,a,n') + h(n'), \text{ where c is the cost going from n to n'.}$$

The tree search version of A is optimal if h(n) is admissible, while in graph-search version is optimal if h(n) is consistent.*

- Completeness: A* is complete.
- Optimality: A* is optimal and optimally efficient

- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$

3.4.3 Memory-bounded heuristic search

Some of the algorithms of this group include **Iterative-deepening A* (IDA)**, **Recursive best-first search (RBFS)** and **Memory-bounded A* (MA*)**. The common characteristic of this algorithms is that they use very little memory (linear in the depth of the deepest optimal solution) and given enough time they can solve problems that algorithms like A* cannot solve because they run out of memory. Here we are going to see an example of Iterative-deepening.

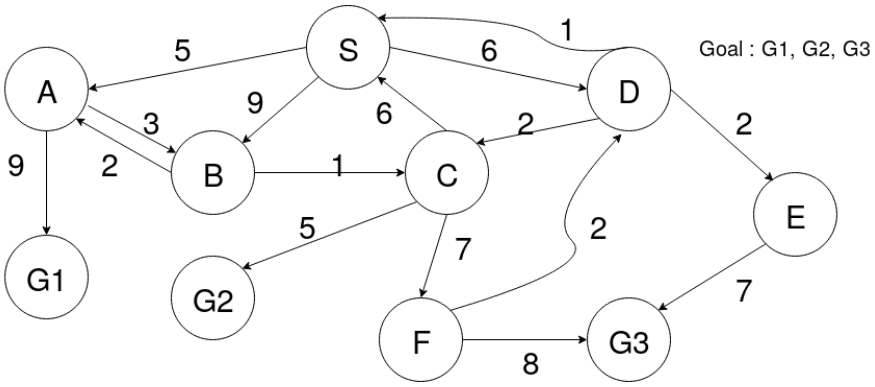

Fig. 14 Search Space of the example.

We start the algorithm at depth *d=0* that correspond to the start state **S**, and ask if **S** is a goal state. If it is, we stop the search and return **S** otherwise expand **S** increasing the depth level *d* by one and so one until we find a goal state. In our example the goal state is G1 and is found in *d=2*.

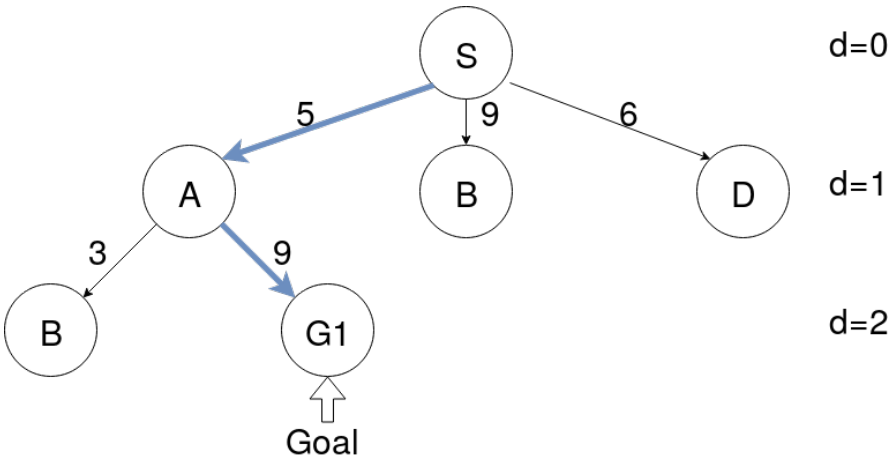

Fig. 15 Iterative Deepening.

3.5 Concepts to Remember

⇒ **Problem Solving Agent**
⇒ **Search**
⇒ **Search Tree**
⇒ **Completeness and Optimality**
⇒ **Time and Space complexity**
⇒ **Uninformed Search**
⇒ **Breadth-First Search**
⇒ **Uniform-Cost Search**
⇒ **Depth-First Search**
⇒ **Depth Limited Search**
⇒ **Iterative Deepening Search**
⇒ **Bidirectional Search**
⇒ **Informed (Heuristic) Search**
⇒ **Best-First Search**
⇒ **Greedy Best-First Search**
⇒ **A* Search**
⇒ **Admissible Heuristic**
⇒ **Memory-Bounded Heuristic Search**

4 Beyond Classical Search

In this chapter we are going to see algorithms that perform **Local Search** and **Optimization** in the state space.

4.1 Local Search Algorithms and Optimization Problems

.

The search algorithms seen in the previous section are called *systematic search algorithms* because they progress by exploring all the paths until the goal is found and by keeping them in the memory. When the search succeed to find a goal then *the whole path is the solution*.
In many problems we do not need the path to the solution but *only the solution*. In this cases **Local Search Algorithms** are the best choice because they consider only the *current node* and it's *neighbourhood* at a time. This algorithms have three main advantages:

1. Use very little memory - usually a constant amount.
2. Can find good solutions i very large, even infinite domains.
3. Are very suitable for solving **Optimization** problems where the goal is not only to find a solution but to find the best solution based on an **Objective Function**.

Examples of Objective Functions may be: Minimize the cost of an operation for example, in this case we want to find the **Global Minima** in the landscape. In the case we want to maximize a function for example the profit we are eager to find the **Global Maximum**. The fig. 16 shows an example of landscape. A **complete** local search algorithm always finds a goal if there exists one, on the other hand an **optimal** algorithm always find the global minima/maxima.

4.1.1 Hill-climbing Search

Hill-climbing Search is one of the most known local search algorithms. It is simply a loop that at each step picks the node with the higher value until the "peak" is found. This process is called **Hill-climbing**. At each step the algorithm record the current node and the objective function, this mean that the memory consumption is low.

function HILL-CLIMBING(problem) **returns** a state that is a local maximum
current ← MAKE-NODE(problem.INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of *current*
if neighbor.VALUE **then return** *current*.STATE
current ← *neighbor*

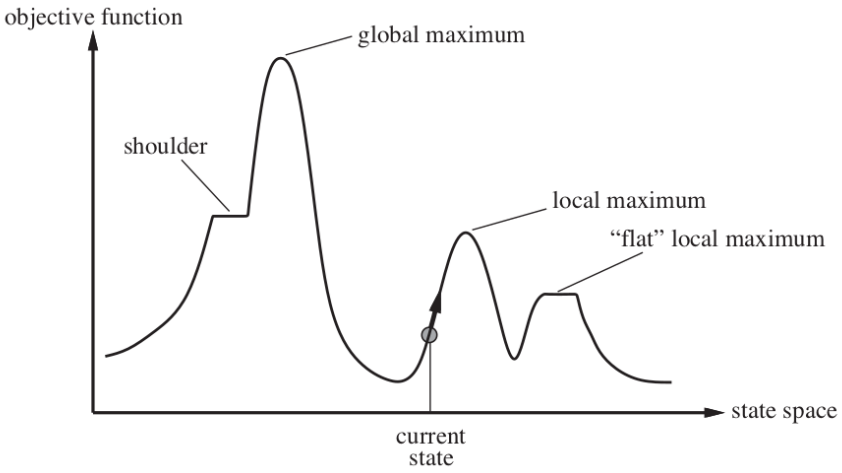

Fig. 16 Local Search.

The algorithm that we have shown is sometimes called a **greedy local search** because at each step it takes the best neighbor but without knowing what to do next. This fact is the reason of the main problems the algorithm has:

- Local maxima: the algorithm get stuck in a local maxima that may not be a global maxima.
- Plateaux: is the case when the neighbourhood is flat and there are not better solution for a while.

There are implemented many versions of hill-climbing, here we present three of them:

- **Stochastic hill climbing**: choose at random from all possible uphills.
- **First-choice hill climbing**: iterate the process of randomly selecting a neighbor for a candidate solution and only accept it if it results in an improvement.
- **Random restart hill climbing**: if a local optimum is reached start again from a random position x. Find the local optimum with regard to x, if it is better than the previous one refresh the *current* variable and start again the process. Usually the iteration when some stop criteria are reached. For example some stop criteria may be: - time criteria, after 30' of execution or - after 100 iterations or - no imporvements on the solution after some iterations etc.

4.1.2 Simulated Annealing

Is a well known local search algorithm that in order to find the global maximum allow "bad" moves toward some not good solutions (downhill) with the hope to avoid local maximum and find better solutions that the previous ones. The algorithm

proceed as follows:
For a finite number of iterations do:

- sample a new point x_t in the neighbourhood of x, $N(x)$.
- jump into the new point with a probability given by the a chosen probability function $P(x, x_t, T)$ where T is positive.
- decrease T

If $T \to 0$ then Simulated Annealing looks like Hill-climbing and when $T \to \infty$ the look like random walk.

4.1.3 Local Beam Search

In contrast with Hill-climbing that keep only the current node and the objective function in memory, **Local Beam Search** keep k possible nodes, where k is a randomly generated number. At each step all the successors of all k are generated and if one of them is the goal state the computation halt otherwise the computation proceed in parallel with all the threads exchanging useful information about the be best solution until that point. Sub-optimal solutions are removed and the most promising solutions are followed and further expanded.
Here is an example of Beam Search.

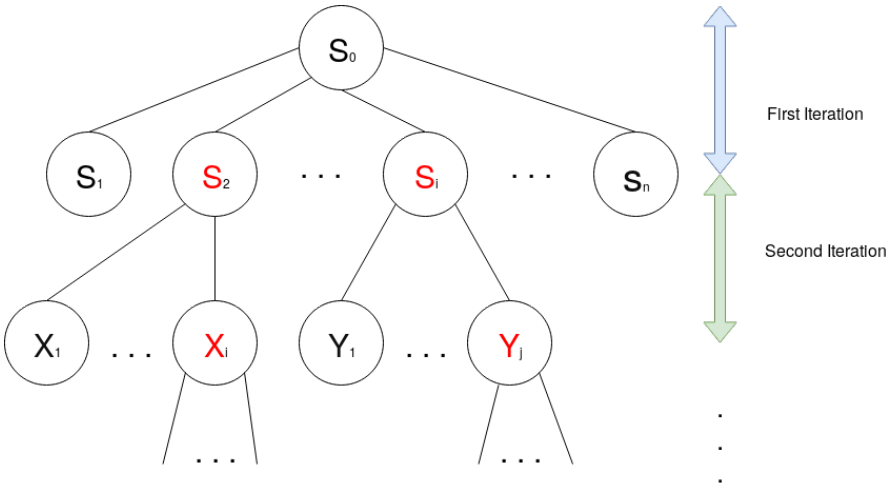

Fig. 17 Local Beam Search.

As you can see in each level the best nodes (the most promising) are expanded for further processing. This way of proceeding restricts enormously the search space and the algorithm converges quickly.

4.1.4 Genetic Algorithms (GA)

Proposed by John Holland in 1970 and try to imitate the biological idea of natural evolution. This are the steps that the algorithm follows:

- Begin with k randomly chosen states (nodes) called **population**.
- Each state, also called individual is represented by a finite string over an alphabet (chromosomes).
- An objective function called *fitness function* is define to represent the most suitable individual. Usually the highest number is the better.
- Crossover is the process of reproduction where the most fit individual are replicated.
- Mutation is the process of evolution. We permit that some strings change in order to evolve.

Now let us consider the problem of the 8-queens. This is a very famous problem and consists of placing 8 queen in an 8x8 board in such a way that they do not attack each other. In this example we are going to follow the steps of the algorithm.
First, we generate k random states, in our case we are going to consider two states (two configurations of the board). This will be our initial population.

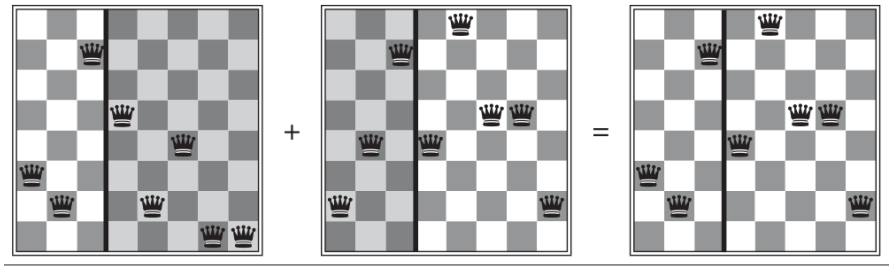

Fig. 18 8-queens example.

Second, we have to represent the individuals as a string of number in an alphabet. In our case the best solution is the enumerate the rows and the columns from 1 to 8 starting from lowest left angle of the board [1,2,3...8]. In our case the first individual (first board) will be represented by the string $s_1 = 3,2,7,5,2,4,1,1$ and the second individual (second board) is represented by the string $s_2 = 2,4,7,4,8,5,5,2$.
Third, we have to define the *fitness function* for this problem. As we mention previously a good fitness function is one with the highest value. A good fitness function here is *number of pairs that do not attack each other*. In the first board this number is 23, in the second is 24.
At this point let's extend this example with two other states. We suppose that we have two more board configuration with fitness 20 and 11.
Another intermediate step to do is to compute the probabilities that this individuals

are chosen for the next steps. One way to do this is to sum all the fitness scores and then divide each of them by the sum. In other words $S = 24 + 23 + 20 + 11 = 77$ so the probability of the first individual will be $24/77 = 31\%$, the second $23/77 = 29\%$ and so on...

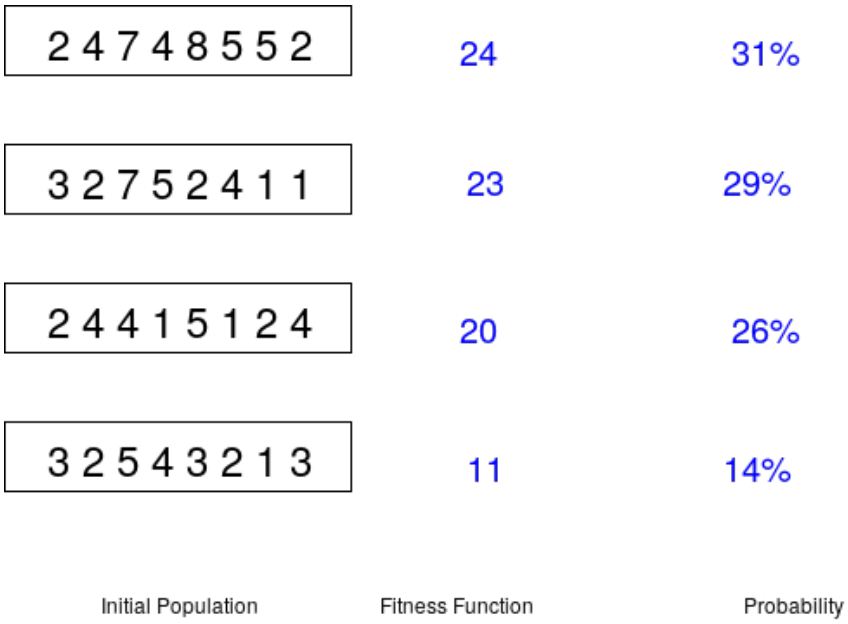

Fig. 19 Initial Population - GA.

Fourth, at this point we pick the individuals with the highest probability to reproduce. And for each pair of individual i decide randomly a *crossover point*.

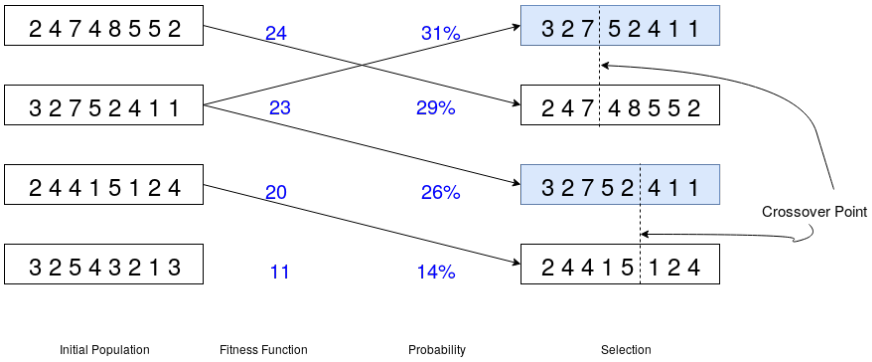

Fig. 20 Selection - GA.

Fifth, the crossover phase. In this phase we combine together the pairs of chosen individuals, more precisely we combine the *left size of the crossover point* of the first individual with the *right size of the crossover point* of the second individual.

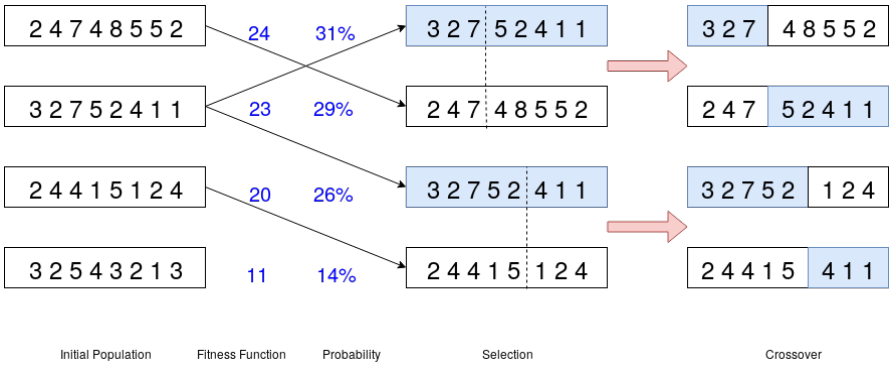

Fig. 21 Crossover - GA.

Sixth, the last phase is to randomly select a mutation for the final individuals. Fig. 22 shows the final state of the algorithm, notice that the mutations are colored in red.

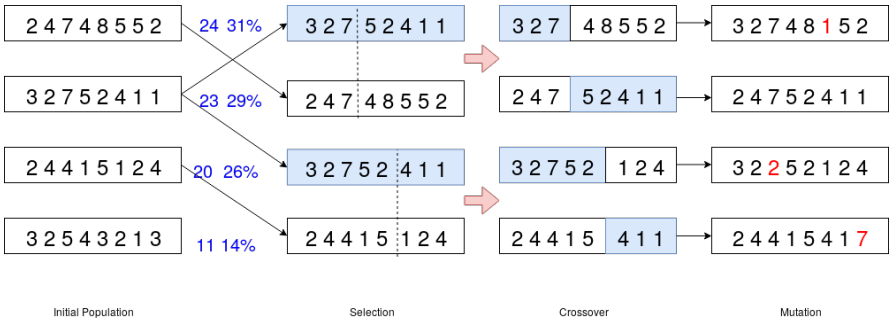

Fig. 22 Mutation - GA.

At this point the algorithm start over again form phase 1, compute the fitness function, the probabilities and so on until some termination condition is met. In the queens problem a termination condition may be the fitness score of 28 that mean that we have reached the goal state, the global minimum or we iterate so much times and no further improvement have been seen. The characteristics of Genetic Algorithms are:

- They work well with discrete and continuous problems.
- The probability to get stack to local minima is very low.

- They are expensive in terms of computation but this can be solve by parallelism.
- Finding a good *fitness function* is very important and the process my be expensive.

4.2 Local Search in Continuous Spaces

Until this point we have described discrete environments but as we know most of the problems in the real World are continuous. The algorithms we have seen cannot handle this problems because they do not have *infinite branching*, in other words the search trees have a finite number of branches. In this section we are going to see local search methods for finding the optimal solution in continuous environments. At this point we are going to talk about the **gradient**.
The **gradient** is a multi-variable generalization of the concept of **derivative** and is a vector valued function, meaning that it has both direction and magnitude. The gradient point to the direction of the greatest increase of the function and the magnitude is the slope of the graph in that direction.
Let's consider this example:
We want to build three new airports in the country such that the sum of the squared distances of each city to the airport to be minimized. Each airport A_i has its coordinates i.e $A_1 = (x_1, x_2)$, $A_2 = (x_1, x_2)$ and $A_3 = (x_1, x_2)$. This is a six-dimension space. We can compute the objective function $f = (x_1, y_1, x_2, y_2, x_3, y_3)$ easily for each state but this mean that we can get only the result of a *local minimum*. In order to find the global minimum or maximum we have to consider the **gradient**. The gradient of the objective function is the vector ∇f and gives the magnitude and the direction of the **steepest slope**.

Definitions:
Given a function $f(x,y,z)$ in three variables the gradients is given by the formula:
$\nabla f = \frac{\partial f}{\partial x} i + \frac{\partial f}{\partial y} j + \frac{\partial f}{\partial z} k$
i, j, k are the standard unit vectors in the three dimensions.

In our case the gradient is: $\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3})$.
To find the local maximum is easy, just solve the equation $\nabla f = 0$.
In order to find the global maximum we have to do a steepest-ascent hill-climbing and we ca do it by iterating the following formula:

Definitions: Finding the global maximum
$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$, α is the **step size**

The constant α is very important because it measure how big or small are our steps towards the global maximum and if we decide to proceed by big steps we are risking to overpass our goal (global maximum). Now there are many ways how to come closer to the goal at each iteration. One of the most well known methods is

the **Newton-Raphson** method. To find the roots of a function we have to solve the equation $g(x) = 0$. We proceed by computing a new estimate of the function at each iteration using the formula:

$x \leftarrow x - \frac{g(x)}{g'(x)}$, where g'(x) is the first derivative of g(x)

In our case we are dealing with multi-variable calculus so instead of the derivative we have to use the gradient. Here is the formula to solve:

Definitions: Gradient descent
$\mathbf{x} \leftarrow \mathbf{x} - H_f^{-1}(\mathbf{x})\nabla f(\mathbf{x})$, where $H_f^{-1}(\mathbf{x})$ is the Hessian matrix of second derivatives whose elements H_{ij} are given by $\frac{\partial^2 f}{\partial x_i \partial x_j}$.

Solving the previous problem in each iteration we get closer to the global minimum. However in continuous space with many variables the solution get computationally hard because of the big size of the **Hessian matrix**.
Constraint Optimization and **Linear Programming** are some of the most used techniques in the Combinatorial Optimization. Here the solutions have be in a certain form that is given by the constraints they have to satisfy.

Let see another example of how to compute gradients. Given the function $g(x,y,z) = \sqrt{x^2+y^2+z^2}$ compute $\nabla g(x,y,z)$. $\nabla g(x,y,z) = < \frac{\partial g(x,y,z)}{\partial x}, \frac{\partial g(x,y,z)}{\partial y}, \frac{\partial g(x,y,z)}{\partial z} > = \frac{x}{\sqrt{x^2+y^2+z^2}}, \frac{y}{\sqrt{x^2+y^2+z^2}}, \frac{z}{\sqrt{x^2+y^2+z^2}}$
Now let's consider this gradient in a particular point P=(2, 6, -3). $\nabla g(2,6,-3) = ?$
First i compute the denominator that is common for all the points: $\sqrt{2^2+6^2+(-3)^2} = \sqrt{49} = 7$, so the points will be:
$< \frac{2}{7}, \frac{6}{7}, \frac{-3}{7} >$

4.3 Numeric Optimization

Optimization Techniques allow us to find the extremes (minimum and maximum) of an *objective function* $f : \Re^n \to \Re$.
We know that $max(f(x)) = -min(-f(x))$ so we need only to find one of the extremes of the function, say $min(f(x))$ and easily we can compute $max(f(x))$.
Golden Section Search is a simple algorithm that has the goal to find the minimum in a function with values defined inside a segment [a,b]. Here is how the algorithm works:

1. we start considering the interval $[a_0, b_0] := [a, b]$ and we fix the *tolerance* value ε.
2. we progressively restrict the interval $[a_{k+1}, b_{k+a}] \subset [a_k, b_k]$ in order that the minimum value x* continue to be inside the new interval.
3. stop the iteration wen $|b_k - a_k| < \varepsilon$ and accept the value $\frac{b_k - a_k}{2}$ as the a good approximation of the minimum x*.

This is a simple algorithm to find the minimum and does not require the use of derivatives, the only problem is that may take some time to converge.

4.4 Searching with Nondeterministic Actions

In the case when the environment is *partially observable* or *nondeterministic* or both we have to devise a **contingecy plan** also called **strategy**. The most used method in this case is the **And-Or Search Trees**. And-Or trees are defined as a graphical representation of the reduction of problem into *conjuctions* or *disjunctions* of sub-problems. Fig. 23 shows an example of AO tree.

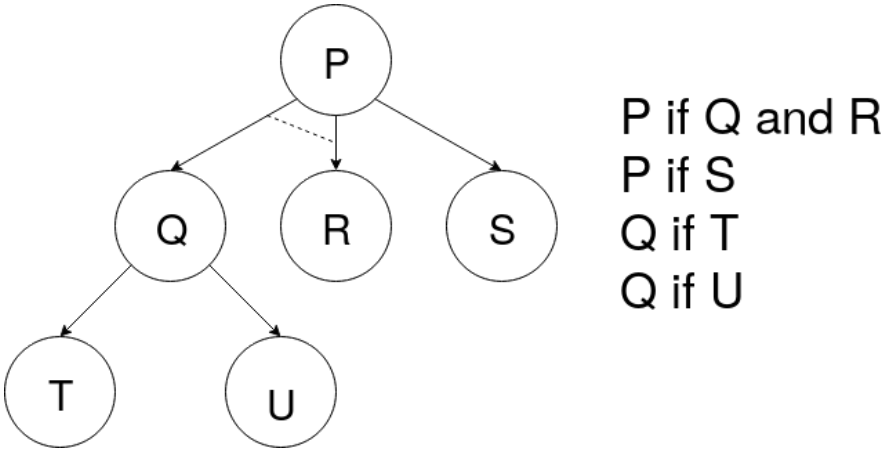

Fig. 23 And-Or tree.

4.5 Concepts to Remember

⇒ **Local Search**
⇒ **Objective Function**
⇒ **Global and Local Minimum/Maximum**
⇒ **Hill Climbing**
⇒ **Greedy Local Search**
⇒ **Simulated Annealing**
⇒ **Gradient Descent**
⇒ **Genetic Algorithms**
⇒ **And-Or Trees**
⇒ **Golden Section Search**

5 Adversarial Search

5.1 Games

Games are the typical example of multi-agent environments where each agent has to consider the action of the other agents in the environment and how this actions influence its welfare. In this situation **competition** is the situation where the **goals** of the agents are in conflict. We call this phenomena **Adversarial Search Problems** or **Games**. Here are some concepts to know about games:

- Competitive or Zero-Sum games (One player win the other lose)
- Perfect Information: players know to result of all previous moves
- Imperfect Information: players do not know the previous moves, maybe the case that they play simultaneously.
- Simple states: easy to describe the state of the game in each moment (no complicated games like soccer or basketball).

Before analyse the games we have define some terminology.

- S_0: initial state of the game
- $PLAYER(s)$: who is the player in the state s
- $ACTIONS(s)$: return the set of legal moves from the state s
- $RESULT(s,a)$: the state after the action a is take on state s
- $TERMINAL-TEST(s)$: returns "TRUE" if s is a terminal state
- $UTILITY(s,p)$: the objective function in state s for the player p

Now imagine we have two player **MIN**, **MAX** and two operations, $ACTIONS(s)$, $RESULT(s,a)$. With this data we can create a **Game Tree**. Here is an example of how a game tree look like.

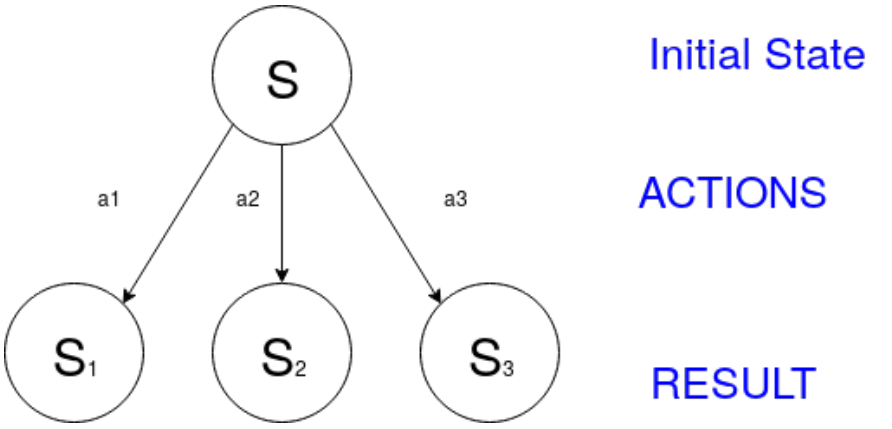

Fig. 24 Game Tree.

In the case of the *Chess* the number of nodes are 10^40 and as you can imagine cannot be explore or memorize all due to he extremely high computational complexity. In tic-tac-toe this number is lower 9! but still very high for such a simple game. Fig. 25 shows a partial tree for the game. **MAX** is the player to start first the game.

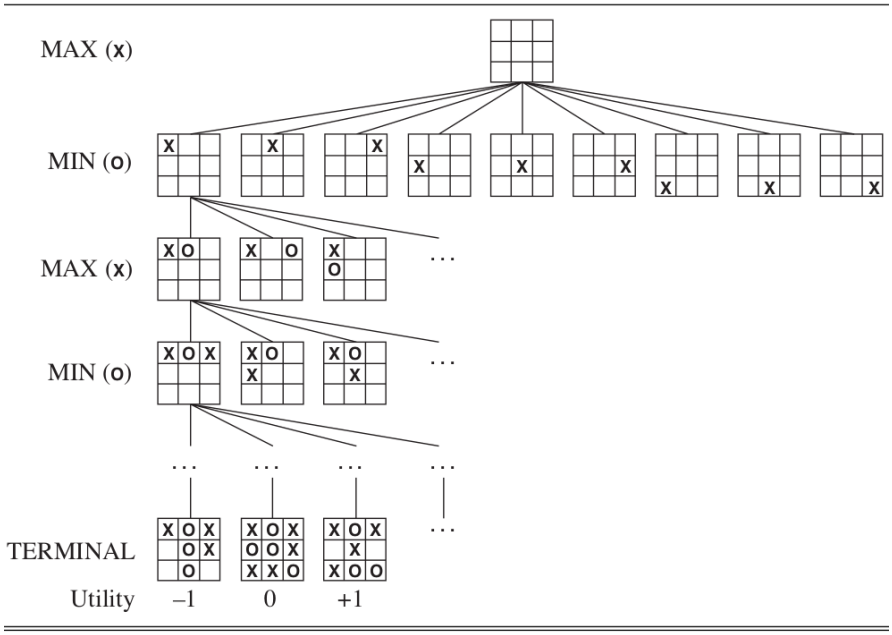

Fig. 25 Tic-Tac-Toe partial game tree.

Utility shows the outcomes of the game in the terminal state.

5.2 MINIMAX Algorithm

The optimal solution in a game is a sequence of actions that leads to a goal state, in other words that leads to a win state. In order to win the game a player, say MAX has to device a **strategy**. Given a game tree the optimal strategy can be determined by the **minimax value** of each node. Fig. 26 show formally the algorithm.

MINIMAX algorithm perform a complete **Depth-First Search** of the game tree. If the maximum depth of the tree is m and there are b legal moves (branching factor) at each point then we have a *time complexity* $O(b^m)$ and a *space complexity* $O(bm)$. This very high complexity makes MINIMAX an impracticable algorithm even for

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Fig. 26 Minimax Algorithm

simple games. We can consider MINIMAX for more than 2 players. In this case we would have a vector of possible values in the UTILITY function and the algorithm may change slightly. This new version is called $\alpha\beta - pruning$.
Fig. 27 shows an example of Minimax.

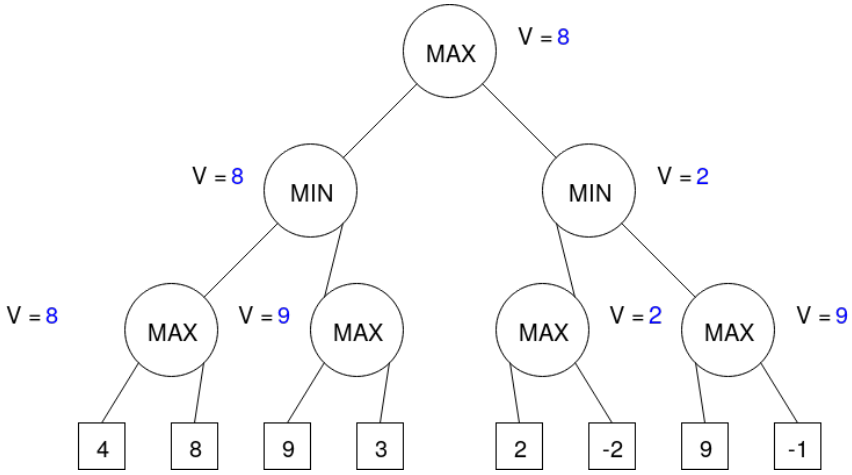

Fig. 27 Minimax Algorithm - Example

5.3 ALPHA-BETA Pruning

The problem with **MINIMAX Search** is that the number of game states that it has to examine is exponential in the depth of the tree. There ate techniques that can alleviate this problem by cutting half the number of states that we have to visti. This techniques are base on the concept of **Pruning**. The algorithm that we are going to study is called $\alpha\beta - pruning$. Here the idea is than we want to prune some nodes for efficiency reasons. The algorithm rely into 2 values α and β.

- α: is the best choice of MAX so far, (highest value)

- β: is the best choice so far for MIN, (lowest value)
- $[\alpha, \beta]$: each node keeps track of its α and β value

MIN always update β and MAX always update α. We start the algorithm assigning $\alpha = -\infty$ and $\beta = +\infty$

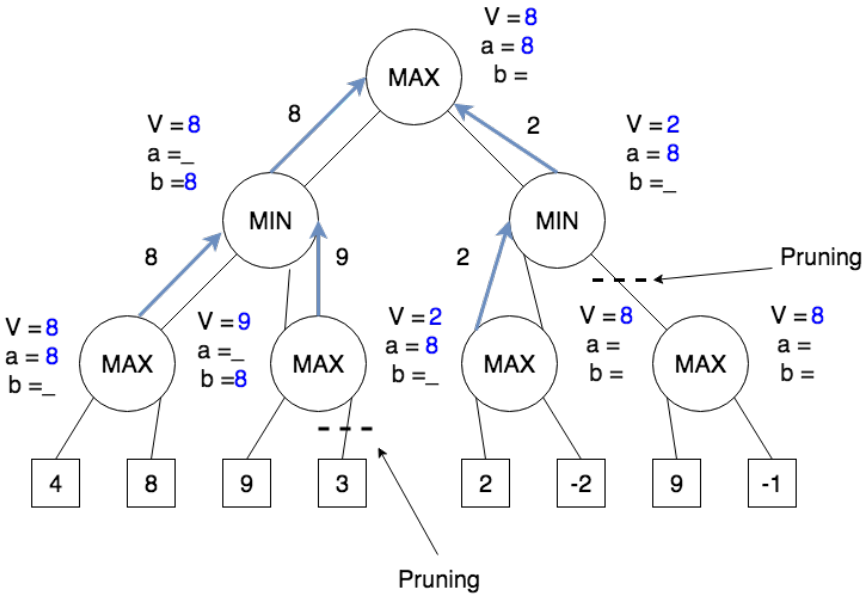

Fig. 28 Alpha-Beta Pruning - Example

The complexity of $\alpha\beta - pruning$ can be reduced into $O(b^{\frac{m}{2}})$.
There are also Non-deterministic games, where we have to include a level of **chance** nodes between MAX and MIN nodes. In this class are included games like backgammon or card playing.

5.4 Concepts to Remember

⇒ **Games**
⇒ **Perfect Information Games**
⇒ **Game Tree**
⇒ **MiniMax Algorithm**
⇒ **Alpha-Beta Pruning**

6 Constraint Satisfaction Problems

In this section we are going to study **Constraint Satisfaction Problems, (CSP)** that are a very important class of Combinatorial Optimization Problems. This problems are very hard to solve because they have exponential complexity. To solve this kind of problems we use different techniques, one of which is **Constraint Programming, (CP)**. But what is CP?
CP is a declarative programming paradigm that allow as to model and solve Combinatorial Optimization Problems (COP) by providing high level construct like global constraints, logical operators etc. The technique that CP use to solve COPs is systematic backtracking tree search.

6.1 Defining CSP

A **Constraint Satisfaction Problem** is modeled as a triple of sets: (X, D, C) where:

- X: is a set of variables called decision variables, $X = X_1, X_2, ...X_n$
- D: is a set of domains, one for each variable X, $D = D_1, D_2, ...D_n$
- C: is a set of constraints that specify the possible values that the variables can take. A constraint is a relation between variables $C_i(X_j...X_k)$.

A Solution for a CSP is an assignment of value to the variables which satisfies all the constraints simultaneously. An example of CSP is the **Map-coloring**: Given a map of states, color the states with different colors in such manner that no adjacent states have the same color. Let's consider the this particular problem with the Australian Map. This problem can be expressed with the following CSP:

- $P = (X, D, C)$
- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D = \{red, green, blue\}$
- $C = \{SA \neq WA, SA \neq NT, SA \neq Q,$
  $SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$

The **Variables** can have discrete, finite domains or infinite domains. Also the values in the **Domain** can be of any type (integers, reals, booleans etc). We many types of **Constraints**, unary, binary, global etc. One of the most important global constraints is $Alldifferent(X_1, X_2, ...X_n)$ that forces all the variables to be different from each other.

Fig. 29 show the problem and fig. 30 the solution.

Fig. 29 Alpha-Beta Pruning - Example

6.2 Local Consistency and Constraint Propagation

As we said previously CP uses systematic search to *extend a partial* assignment of values to variables into a complete and consistent one. Of course not all assignment are consistent, so we have to find a way to detect this kind of inconsistent assignments. **Local Consistency** is a form of **inference** that allow us to detect inconsistent partial assignments. In this way we can do less job in search. Is called Local consistency because we examine the constraints one at a time because the global consistency is NP-hard. To recap *Local Consistency is a property that we want to guarantee and we do that using Propagation*.
Propagation are all the techniques that are used to enforce Local Consistency. The way that this techniques and methods are implemented are let free so anybody can implement them, important is the goal, enforce the Local Consistency. There are many kinds of Consistencies but the most used are **Domain Based Consistencies**, like **Arc Consistency**, **Generalized Arc Consistency** and **Bounds Consistency**. This kind of consistencies are able to detect inconsistencies of the type: $X_i = j$ and if this assignment is inconsistent they eliminate the value j from the domain of variable X_i via **propagation**.

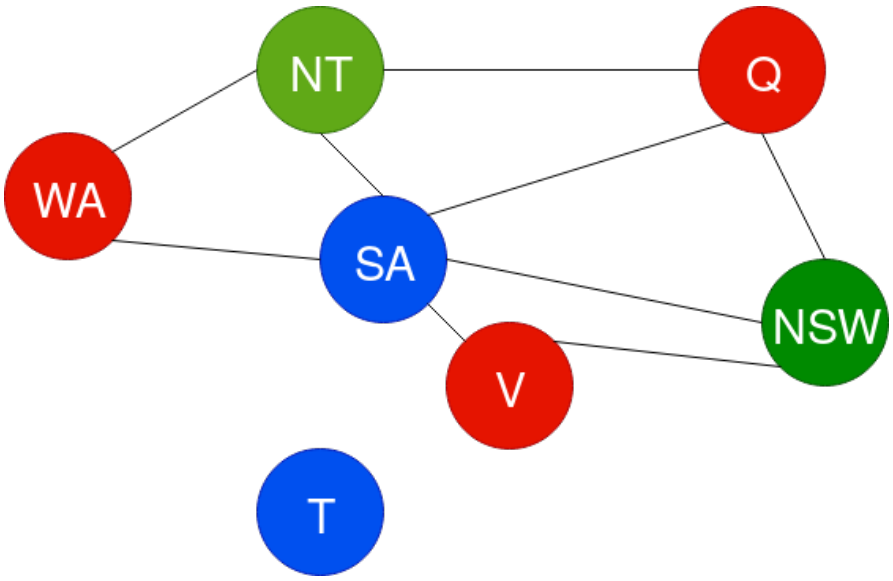

Fig. 30 Alpha-Beta Pruning - Example

6.3 MiniZinc

MiniZinc is a high level modelling language that is used to state Constraint Satisfaction Problems. We use MiniZinc to write the high level code that models a CSP and later MiniZinc use one of 20 solvers to solve the CSP. The compilation process pass through a low level representation called FlatZinc and then to the solver.

MiniZinc $\Rightarrow$ **FlatZinc** $\Rightarrow$ **Solver X**

Here is our first example of MiniZinc. Suppose we have this problem:

maximize 25B + 30T
such that $(1/200)B + (1/40)T \leq 40$
$0 \leq B \leq 60$
$0 \leq T \leq 40$

We can model in this way the previous Problem using MiniZinc:

```
var 0..60: B;
var 0..40: T;
constraint (1/200)*B+(1/140)*T ¡= 40;
solve maximize 25*B + 30*T;
output [”B = ”, show(B), ” T = ”, show(T)];
```

Some MiniZinc Syntax:

- Parameters: are variables like in all programming languages, we assign them a value: **int: i = 3;**
- Decision Variables: are the variables used in the model: **var 0..4: i;** or **var 1,2,3,5: j;**
- Many types: Integers, Booleans, Arrays, Strings etc.
- Strings; used only for output: **show(v);**
- Arithmetic Expressions: all the operators
- Data file: each model can take as input an instance of data called a data file (.dzn)

A MiniZinc model may have the following elements:

1. Inclusion elements: $include < filename >$;
2. Output: $output < listofstrings >$;
3. Variable declaration
4. Variable assignments
5. Constraints: $constraint < booleanexpression >$;
6. Solve element in different forms: *solve*
7. Predicates and tests
8. Annotations

Fig. 31 shows the complete code for the Knapsack problem in MiniZinc.

```
int: n;           % number of objects
int: weight_max; % maximum weight allowed (capacity of the
knapsack)
array[1..n] of int: values;
array[1..n] of int: weights;
array[1..n] of var int: take; % 1 if we take item i; 0
otherwise

var int: profit = sum(i in 1..n) (take[i] * values[i]);
solve maximize profit;

constraint        % all elements in take must be >= 0
forall(i in 1..n) ( take[i] >= 0 )    /\
sum(i in index_set(weights))( weights[i] * take[i] ) <=
weight_max;

output [show(take), "\n"];
```

Fig. 31 Knapsack Problem - Example

For further study please check the website of MiniZinc[1].

[1] https://www.minizinc.org/

6.4 Concepts to Remember

⇒ **CSP**
⇒ **CP**
⇒ **Local Consistency**
⇒ **Constraint Propagation**
⇒ **Global Constraints**
⇒ **MiniZinc**

7 Logical Agents

In this chapter we are going to study **Knowledge-Base Agents**, that are agents that have an internal representation of the **knowledge** and use a process called **reasoning** to access and manipulate this knowledge. In order to be able to reason we have to create a formalism to express knowledge. So here we are going to introduce formal **Logic**. For our purpose **Propositional** and **First-Order Logic** will be enough.

7.1 Knowledge-Base Agents

We call **Knowledge-Base, (KB)** a set of **sentences** or propositions expressed in a special language called **knowledge representation language**. The sentences that are not derived from other sentences are called **Axioms**. We can add sentences in a KB by a process called **TELL** and query the KB with **ASK**. We use **Inference** to derive new sentences for existing ones. Every formal language has a well defined **syntax** that is used to represent all the well-formed sentences and a **semantics** that define the meaning of the sentences. The semantics express the truth value of a sentence in a model (world).
If a sentence α is true in a model M we say that α satisfies M.
A sentence β follows from a sentence α is written in this form $\alpha \models \beta$ and represent the notion of **logical entailment**. This formulation is to say that in every model that α is true β is also true.

Definition:
$\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$

An **inference algorithm** is an algorithm that can derive conclusions based on some premises. We want that this algorithms to be both **sound** and also **complete**. **Soundness** mean that every formula that can be proved in a system is logically valid with respect to the semantics of the system. **Completeness** is the property of an inference algorithm to derive any entailed sentence. This two properties are very desirable in logic.
If KB is true in the real world then any sentence α derived by a sound inference procedure is guaranteed to be true in the real world. Important is also the concept of **grounding** that asks the question. How do we know that our KB is true in the real world?!

7.2 Propositional Logic

Propositional Logic is the most simple kind of Logic we use to express knowledge and to formalize natural language. When we define a Logic we have to define first

the **syntax** then **semantics**. In the case of Propositional Logic we have the following elements.

- **Symbolic aids**: used to help us in order to express the sentences in a clear manner. This aids are "parenthesis" (,).
- **Connectives**: Permit to create complex sentences combining together atomic sentences. There are 5 of them $\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\Longleftrightarrow$
- **Propositional Constants**: Top ($\top$) and Bottom ($\bot$).
- **Propositional Variables**: An infinite number of atomic propositions $p_0, p_1 ....$

Also we have a set of rules that allow us to create complex sentences:

Rules:
$\omega ::= \top | \bot | p_i | \neg\omega | (\omega \wedge \omega) | (\omega \vee \omega) | (\omega \Longrightarrow \omega) | (\omega \Longleftrightarrow \omega)$

The previous definitions are all we need in Propositional Logic and the following figure summarize this.

$$
\begin{array}{rcl}
Sentence & \rightarrow & AtomicSentence \mid ComplexSentence \\
AtomicSentence & \rightarrow & True \mid False \mid P \mid Q \mid R \mid \ldots \\
ComplexSentence & \rightarrow & (\ Sentence\ ) \mid [\ Sentence\ ] \\
 & \mid & \neg\ Sentence \\
 & \mid & Sentence \wedge Sentence \\
 & \mid & Sentence \vee Sentence \\
 & \mid & Sentence \Rightarrow Sentence \\
 & \mid & Sentence \Leftrightarrow Sentence \\
\text{OPERATOR PRECEDENCE} & : & \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Fig. 32 Grammar of Propositional Logic in BNF

At this point after the **Syntax** we have to define the **Semantics** of the Propositional Logic. Semantics defines the rules that assign truth value to each sentence with respect to a particular model. We use the **Truth Table** to define the Semantics. See the following figure for more details.

Propositional Entailment is is co-NP complete meaning that in worse case the computational complexity of the algorithm is exponential in the input size.

Until now we have verified **entailment** by means of **model checking**, so by enumerating all the models where the entailment are true. There is another method

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Fig. 33 Truth Table - Semantics of Propositional Logic

called **Theorem Proving**, meaning proving entailment applying some well defined *rules* of inference. We are going to apply directly inference on sentences of the KB in order to arrive on a proof without checking the models.
Logical Equivalence ($\equiv$): We say that 2 sentences α, β are logically equivalent if they are true in the same set of models. $\alpha \equiv \beta$ for example $A \wedge B \equiv B \wedge A$.

Logic Equivalence:
$\alpha \equiv \beta \iff \alpha \models \beta \wedge \beta \models \alpha$

Validity: A sentence is valid if it is true in *all* possible models. Valid sentences are called **Tautologies** and they are necessary true. An example of a tautology is the following: $A \wedge \neg A$. Because $\top$ is *true* in all models every valid sentence is equivalent to $\top$.

Deduction Theorem:
For any sentence α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Longrightarrow \beta)$ is valid.

Conclusion: We can decide $\alpha \models \beta$ by checking that $\alpha \Longrightarrow \beta$ is true.

Satisfiability: A sentence is *satisfiable* is it is *true* in or satisfied by *some* models. The *satisfiability, (SAT)* problem can be checked my enumerating all models until one is found that satisfies the sentence. SAT in Propositional Logic is NP-complete. **Validity** and **Satisfiability** are closely related: α is valid iff $\neg\alpha$ is unsatisfiable and α is satisfiable iff $\neg\alpha$ is not valid.

Reductio ad Absurdum:
$\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisbiable.

Proving β from α by checking the unsatisfiability of $\alpha \wedge \neg\beta$ is also called **confutation** or **proof by contradiction**.
In general we use **Inference Rules** in order to derive/infer a **Proof**. Here we are going to see some of them:

Modus Ponens:
$$\frac{(\alpha \Longrightarrow \beta) \quad \alpha}{\beta}$$
if $\alpha \Longrightarrow \beta$ and α is given i can infer β.

AND-Elimination:
$\frac{(\alpha \wedge \beta)}{\beta}$ or $\frac{(\alpha \wedge \beta)}{\alpha}$
if a conjuction of two terms is given i can infer each of them separately.

Monotonicity: States that the set of entailed sentences can only increase if more information is added into the KB.

Another way to prove sentences is by applying the **Resolution** rule. The Resolution takes as input a clause (**disjuction of literals**) and a new literal and produce a new clause. Here is an example of how this rule works:

$$\frac{(A \vee B) \quad (\neg A \vee \neg C)}{B \vee \neg C}$$

A resolution-based theorem prover can, for any sentences α and β in the Propositional Logic decide wheather $\alpha \models \beta$.
Resolution works only for clauses - disjuction of literals ($\alpha \vee \beta$).
Every sentence in Propositional Logic is logically equivalent to a conjuction of clauses.
A sentence expressed as a conjuction of clauses is called Conjuctive Normal Form (CNF)

Here is an example of applying logical rules to transform $B \iff (A \vee C)$ into a CNF.

1. $B \iff (A \vee C)$
2. $(B \implies (A \vee C)) \wedge ((A \vee C) \implies B)$
3. $(\neg B \vee A \vee C) \wedge (\neg (A \vee C) \vee B)$
4. $(\neg B \vee A \vee C) \wedge ((\neg A \vee \neg C) \vee B)$
5. $(\neg B \vee A \vee C) \wedge (\neg A \vee B) \wedge (\neg C \vee B)$

Resolution algorithm in order to verify that $KB \models \alpha$ shows that $(KB \wedge \neg \alpha)$ is *unsatisfiable*.
How the algorithm works:

- First: we transform $(KB \wedge \neg \alpha)$ in a CNF and then we use it as an input to the algorithm.
- Second: each pair that contain complementary literals is resolved to produce a new clause which is added in the set if it is not already present.
- Third: this process continues until one of this two things happens:
  1. there are no new clauses that can be added meaning that KB does not entail α.
  2. the last two clauses resolve into an *empty clause* , that means that KB entails α.
- Four: End

Definite Clause are a disjuction of literals where *exactly one is positive*. $\neg A \vee B \vee \neg C$ **Horn Clause** is used for a simplified form of resolution. Horn clauses are a disjuction of literals where *at most one is positive*. **Goal Clause** is called a clause with no positive literals.

We want to work with KB that have only *definite clauses* for 3 reasons:

- Every definite clause can be written as an implication whose *premise* is a conjuction of positive literals and whose *conclusion* is a single positive literal. $(\neg A \vee \neg B \vee C) :: (A \vee B) \Longrightarrow C$. In Horn form the *premise* is called *body* and the *conclusion* is called *head*. A *single positive literal* is called *fact*.
- Forward/Backward chaining algorithms are used to do *inference* of Horn clauses. This algorithms are the base of *Logic Programming*.
- The complexity of proving *entailment* in Horn is *linear* on the size of KB.

Forward-Backward Chaining algorithm takes as input a KB and a proposition we call it *query* and defines if the query is entailed by the KB. This algorithm is **sound** and **complete** and at the end the algorithm reach the so called **fixed point** where no more computation ca be done.

⇒ **Knowledge Base**
⇒ **Syntax and Semantics of Propositional Logic**
⇒ **Truth Tables**
⇒ **Entailment**
⇒ **Validity**
⇒ **Satisfiability**
⇒ **Soundness**
⇒ **Completeness**
⇒ **Model Checking**
⇒ **Inference Rules**
⇒ **Resolution**
⇒ **Horn Clauses**

8 First-Order Logic

In this chapter we are going to see *First-Order Logic*, a more expressive and powerful language than *Propositional Logic*.

8.1 FOL

We can use the best elements of PL such as the fact that is *declarative*, *compositional*, *context-independent* and *unambigous* and some aspects of the Natural Languages. We can associate nouns to **objects**, verbs to **relations** and **functions**. A **function** is a relation from one input to one output.

$$
\begin{array}{rcl}
Sentence & \rightarrow & AtomicSentence \mid ComplexSentence \\
AtomicSentence & \rightarrow & Predicate \mid Predicate(Term, \ldots) \mid Term = Term \\
ComplexSentence & \rightarrow & (\ Sentence\ ) \mid [\ Sentence\ ] \\
& \mid & \neg\ Sentence \\
& \mid & Sentence \wedge Sentence \\
& \mid & Sentence \vee Sentence \\
& \mid & Sentence \Rightarrow Sentence \\
& \mid & Sentence \Leftrightarrow Sentence \\
& \mid & Quantifier\ Variable, \ldots\ Sentence \\
\\
Term & \rightarrow & Function(Term, \ldots) \\
& \mid & Constant \\
& \mid & Variable \\
\\
Quantifier & \rightarrow & \forall \mid \exists \\
Constant & \rightarrow & A \mid X_1 \mid John \mid \cdots \\
Variable & \rightarrow & a \mid x \mid s \mid \cdots \\
Predicate & \rightarrow & True \mid False \mid After \mid Loves \mid Raining \mid \cdots \\
Function & \rightarrow & Mother \mid LeftLeg \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Fig. 34 Syntax of FOL

The models in FOL are interesting because they have objects in them. We call **domain** of a model the set of objects it contain. The basic syntactic elements of FOL are the symbols used to represent objects (**Constant symbols**), relations (**Predicate symbols**) and functions (**Function symbols**). Each predicate or function has a defined number of parameters we call (**arity**).
In addition to *objects, predicates, functions* each model include also an **interpretation**.
A **Term** is a lexical expression that refers to an object. We can combine together *terms* that refers to objects and *predicates* that refer to relations to form **Atomic Sentences** also called **atoms** that are use to state *facts*. An Atom is formed by a predicate followed by a list of terms e.g $Brother(John, Jim)$.. We can create complex atomic sentences e.g $Married(Father(John), Mather(John))$.
Complex Sentences are sentences created combining together *Atomic Sentences* using *connectives*.
$\neg Brother(LeftLeg(Richard), John)$. or $Brother(Richard, John) \wedge Brother(John, Richard)$.
Quantifiers allow us to express properties of entire collections of objects. FOL has 2 quantifiers:

- Universal Quantifier ($\forall$): allow us to make statements about all the elements of a domain. For example: $\forall x King(x) \implies Person(x)$. this sentence mean: for all *variable* x, if x is a King then x is a Person also.
- Existential Quantifier ($\exists$): allow us to make statements about *some* elements of the domain. For example $\exists x Person(x) \wedge King(x)$. We can read this as : "There exists an x such that x is a person and a king". $\exists x P$ means that P is *true* if there exists at least one oject x.

The natural connective for $\forall$ is $\implies$ and for $\exists$ is $\wedge$. We can write this $(\forall, \implies)$ and $(\exists, \wedge)$.

We can have more than one quantifiers together in a sentence, this are called **Nested Quantifiers**.

- Same type of quantifiers: "Brothers are Siblings", $\forall x \forall y Brothers(x, y) \implies Siblings(x, y)$.
- Different type quantifiers: "Everybody loves somebody":: $\forall x \exists y Loves(x, y)$.
  "There is someone that is loved by everyone" :: $\exists y \forall x Loved(x, y)$.
  As you can see the order of the quantifiers is very important.

$\forall x(Crown(x) \vee (\exists x Brother(Richard, x)))$. A variable belongs to the innermost quantifier that mention it.

There is another manner to make *atomic sentences* in FOL, and this is **Equality Symbol (=)**. Equality symbol is used to state that two terms refer to the same object, e.g: $Father(John) = Henry$.
If we want to state that "Richard has at least two brothers" we write:
$\exists x, y Brother(x, Richard) \wedge Brother(y, Richard) \wedge \neg(x = y)$.
the previous definition is different from the following:
$\exists x, y Brother(x, Richard) \wedge Brother(y, Richard)$.
The second sentence do not specify that x and y are different (do not refer to the

same object) and this is the reason why the two sentences are different from each other. Here are the De Morgan rules for FOL.

$$\begin{array}{lll} \forall x \ \neg P \equiv \neg\exists x \ P & & \neg(P \vee Q) \equiv \neg P \wedge \neg Q \\ \neg\forall x \ P \equiv \exists x \ \neg P & & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\ \forall x \ P \equiv \neg\exists x \ \neg P & & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\ \exists x \ P \equiv \neg\forall x \ \neg P & & P \vee Q \equiv \neg(\neg P \wedge \neg Q) \ . \end{array}$$

Fig. 35 De Morgan rules for FOL

Once defined the FOL we have to use it for our purposes. Imagine we have a KB, we use TELL to add sentences in the KB and ASK to retrieve information from the KB.
Assertion: are called the sentences added in the KB with the use of **TELL**. For example:
TELL(KB, King(John)).
TELL(KB, Person(Richard)).
TELL(KB, $\forall x King(x) \Longrightarrow Person(x)$).
Queries: are used to ask question to the KB.
ASK(KB, King(John)). :: "true"
Here **ASK** are called *queries* or *goals*. In the case we ask questions like: *"Which values of x make the question true"* then we have to define another type of queries called **ASKVARS** that returns a set of values.
ASKVARS(KB, Person(x)).
This query will return all the values in KB that satisfies the predicate, Person(x). This is all the set of values $x/John$ and $x/Richard$ in our KB. This kind of answer is called a **substitution** or **binding list**.
Every KB can be viewed as a set of **axioms** that from which we can entail **theorems**.

8.2 Numbers, Sets and Lists

Numbers Theory can be defined starting from a very small set of axioms. In fact we need a **predicate** we call *NatNum*, one **constant symbol** we call *0* and one **function symbol** we call *successor, S*. We use the **Peano axioms** to define the Natural numbers in a recursive manner:
$NatNum(0)$.
$\forall n NatNum(n) \Longrightarrow NatNum(S(n))$.

The previous assertions meaning is: 1) "0 is a Natural Number", 2). for every object n, if n is a Natural number, the successor of n (S(n)) is also a Natural number.

So we can view the Natural numbers as: ”0, S(0), S(S(0))...). We can add more axioms in order to be able to implement operations such as (+).
Sets are a very important mathematical object. We can express them using the constructs of FOL.

- Constant: empty set .
- Predicates: One unary predicate *Set*, two binary predicates: $x \in s$ (x is included in the set s) and $s1 \subseteq s2$ (s1 is a subset of s2).
- Functions: Three binary functions: $s1 \cap s2$ (intersection), $s1 \cup s2$ (union) and $x|s$ (a new set that results form adjoining x to the set s).

There are 8 axioms used to implement sets for example one of them is:
$\forall s Set(s) \iff (s = \{\}) \vee (\exists x, s2 Set(s2) \vee s = \{x|s2\})$
For the other axioms refer to the book.
Lists have some other characteristics that has to be take in consideration, for example list are *ordered* and also the same element may appear *multiple* times in a list. To define the lists we need:

- Constant: *Nil* is a constant list without no elements.
- Functions: *Cons, Append, First, Rest*
- Predicate: *Find*

Using the previous literals we can define the lists,
$[A,B,C] :: Cons(A,Cons(B,Cons(C,Nil)))$.

8.3 Prolog

Prolog is a *logical* and a *declarative* Programming Language. In a declarative language the programmer specifies a **goal** and leave to the system to work out the way of how to achieve it.
Prolog programs specifies *relationships* between objects and properties of the objects. For example we can say: *John has a bike* - here we are declaring the ownership relationship between 2 objects.
Prolog program contain 3 elements: **Facts, Rules, Queries**. **Facts** describe an *explicit* relationship between objects and properties objects might have (”John has a bike”, ”Hair is black”, ”Apple is a company” etc).

- names of properties/relationships begin with **lower** case letters.
- objects appear as comma-separated arguments withing parentheses (a,b,c).
- relationship name appear as the first term.
- a period (.) must be at the end of the facts.
- facts are also called *predicate* or *clause*.
- example: **teaches(fabio, uux).** or **teaches(andrea, ml).**
- all the *facts* together form Prolog’s database called *Knowledge Base*.

Rules define implicit relationships between objects (e.g brother relationship) and object properties (A is child of B if B is parent of A). Rules are **non-unit** clauses, facts are **unit** clauses.

- variable names starts with a capital letter.
- example: **guide(Teacher,Student):-**
  teacher(Teacher,CourseId), studies(Student,CourseId).

Queries allow as to ask questions about object and object properties. Queries are beased on *facts* and *rules*. Suppose if we want to know if andrea teaches course uux, we can write:
?-teaches(andrea,uxx).

Syntax of Clauses:

- **:-** means *if* or *is implied by*. Left side of :- is called the **Head** and the right side is called **Body**.
- **,** means *and, logic conjuction*.
- **;** means *or, logic disjuction*.
- **Program = Facts + Rules + Queries**.
- **Program = Set of clauses**.
- **Three type of clauses: facts, rules, queries**.
- **Procedure** is a set of clauses about the same relation.

Family Relationship in Prolog: Prolog is a programming language for *symbolic, non-numeric computation*. Let us define the KB for the family relationship. We define relationships by adding **tuples** in the KB. Once added we can query the KB about relationships. Each clause in the KB has to terminated by a **full-stop, (.)**.
In examples such **parent(bob,tom).**, parent, bob, tom are called **atoms** while in **parent(bob,X).** X is a **variable**.

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```

At this point we can define other relationships such as: mother, female, sister, grandparent etc.

```
mother(X,Y):-parent(X,Y), female(X).
sister(X,Y):-parent(Z,X), parent(Z,Y), female(X), X\== Y.
haschild(X,_).
grandparent(X,Z):-parent(X,Y), parent(Y,Z).
```

Let's define the predecessor relationship that uses **recursion**.

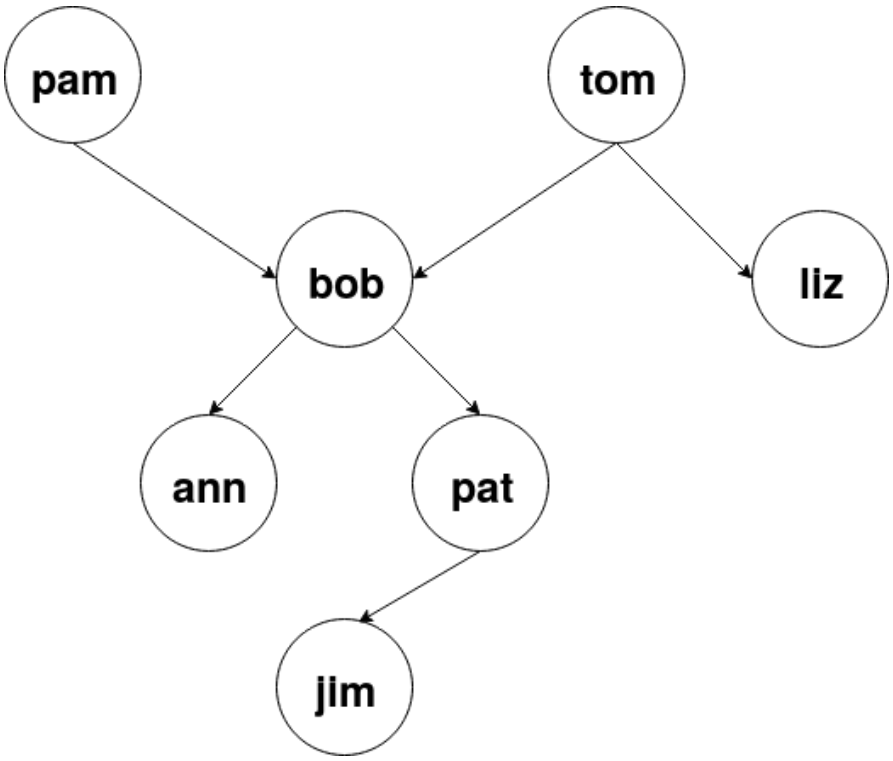

Fig. 36 Family Relationship - Example

```
predecessor(X,Y):- parent(X,Y). %base case
predecessor(X,Z):- parent(X,Y), predecessor(Y,Z). %recurssive case
```

Data Objects in Prolog

- **Atoms**: (tom, pat, x100, *x_*34...) can be constructed in 3 ways:
  1. string of low case letters, digits or _ starting by a letter.
  2. string of special characters $<----->$ or $====>$ etc
  3. string of characters enclosed in single quotes even in capital letters ('Eduart').
- **Numbers**: we have *Integers* and *Reals*.
- **Variables**: are string of letters, digits or _ that starts with an upper-case letters. (X, Member_name, _abc...).
- **Structures**: are data objects that have multiple components. The components can also be structures.

Lists are a simple data structure that allow us to represent *collections* of items, e.g L=[blue, red, green, black]. A list can be **empty, []** or **non-empty**. In the second case the first item of the list is called *Head* and the remaining part of the list is called *Tail* and is itself a list. Lists are a very important construct in Prolog.

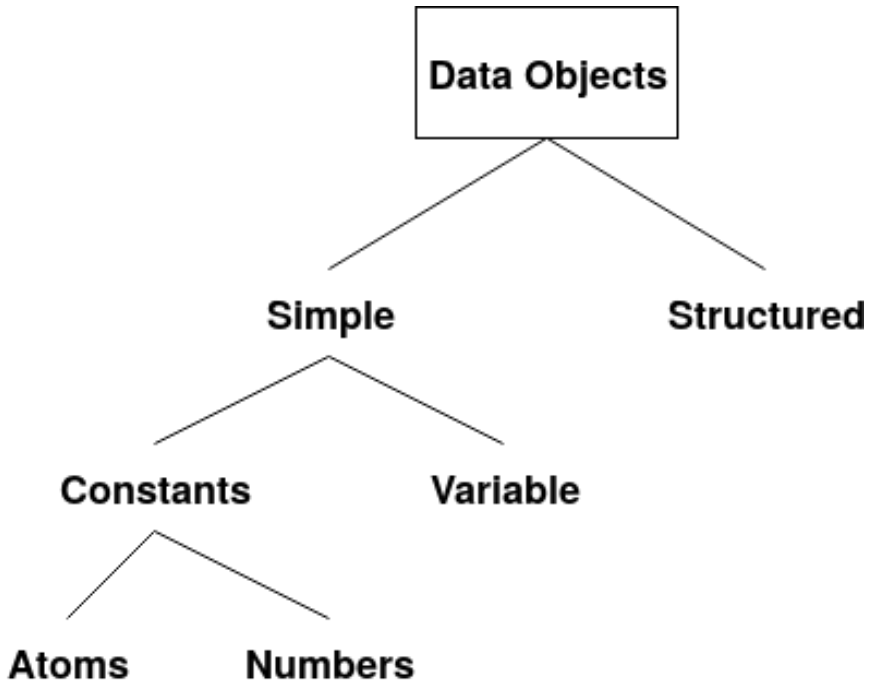

Fig. 37 Data Objects in Prolog

Let's see now some examples of Prolog:

```
Max and Min of 2 given numbers - max(X,Y,Max)
max(X,Y,X):-X>=Y.
max(X,Y,Y):-X<Y.
```

Membership, check if an item is presented in the list. We can use also the build-in function *member*.

```
Membership - lmember(X,L).
lmember(X,[X|_]).
lmember(X,[_|TAIL]):-lmember(X,TAIL).
```

Length of a list

```
Length of a list - length(L,N).
length([],0).
length([_|TAIL], N):-length(TAIL, N1), N is N1+1.
```

Concatenation of a list

```
Concatenation of 2 lists - conc(L1,L2,L3).
conc([],L,L).
conc([X1|L1],L2,[X1|L3]):-conc(L1,L2,L3).
```

Deletion of an element X from the list

```
Deletion - del(X,L).
del(Y,[Y],[]).
del(X,[X,|L1],L1).
del(X,[Y|L],[Y|L1]):-del(X,L,L1).
```

Appending an element to a list

```
Appending an element to a list - app(X,L1,L) (we use member())
member(X,[X| _]).
member(X,[_ |TAIL]):-member(X,TAIL).)
app(A,A,T):-member(A,T),!. %we use cut (!)
app(A,TAIL,[A|TAIL]).
```

Here the (!) symbol is used to abandon the search in the case when the previous predicate is *true*.

Inserting an element in a list, using the previous definition of del().

```
Insert - ins(X, L, R) - using del(X,L))
ins(X,L,R):-del(X,R,L).
```

N-th element of a list

```
Find N-th element of a list
nlist(1,[X—L],X).
nlist(N,[Y—L],X):-N1 is N - 1, nlist(N1,L,X).
```

At this point let us discus **Matching** or **Unification** in Prolog. Matching means that given two terms to check if they are *identical* or the variables in both terms can have the same object after being instantiated. For example:
date(D, M, 2006) = date(D1, feb, Y1). means: **D=D1, M=feb, Y1=2006**.

Before giving the general rules for the *matching/unification* we have to recall that there are 3 types of terms in Prolog:

1. **Constants** that may be **atoms** or **numbers**.
2. **Variables**.
3. **Complex terms** (**Structures**): functor(term1, term2,....).

The general rule is *Two terms match, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal.* Now we are ready to give the 4 rules that are used to decide matching:

Matching/Unification Rules
1. If *term1* and *term2* are constants, then *term1* and *term2* match if and only if they are the same atom, or the same number.
2. If *term1* is a variable and *term2* is any type of term, then *term1* and *term2* match, and *term1* is instantiated to *term2*. Similarly, if *term2* is a variable and *term1* is any type of term, then *term1* and *term2* match, and *term2* is instantiated to *term1*. (So if they are both variables, they're both instantiated to each other, and we say that they share values.).
3. If *term1* and *term2* are complex terms, then they match if and only if:

1. They have the same functor and arity.
2. All their corresponding arguments match
3. and the variable instantiations are compatible. (I.e. it is not possible to instantiate variable X to mia, when matching one pair of arguments, and to then instantiate X to vincent, when matching another pair of arguments.)

4. Two terms match if and only if it follows from the previous three clauses that they match.

Another important topic in Prolog is **Backtracking**. Sometimes we want to prevent backtracking for *efficiency* reasons.
Preventing Backtracking

- Prolog will automatically backtrack if this is necessary for satisfying a goal.
- Uncontrolled backtracking may cause inefficiency in a program.
- "Cut", (!) can be used to prevent backtracking.

Now let us consider an example: Suppose that we have the following 3 rules:

- R1: if $X < 3$ then Y=0. :: **f(X,0):-X<3.**
- R2: if $3 =< X$ and $X < 6$ then Y=2. :: **f(X,2):-3=<X, X< 6.**
- R3: if $6 <= X$ then Y=4. :: **f(X,4):-6 =<X.**

At this point let us ask:
?- f(1,Y), 2 < Y.
The first goal (f(1,Y)) uses R1 and instantiate Y=0, while the second goal fails because (2<0), the Prolog *backtracks* to use the other rules in order to solve the goals. Now we can see that all this predicates are mutually exclusive, when one is true the others are false, so backtracking is only a **time-consuming** process that we want to prevent. We can prevent this by using Cut (!). So we modify our code like this:

- **f(X,0):-X<3, !.** %Rule 1
- **f(X,2):-3=<X, X< 6, !.** %Rule 2
- **f(X,4):-6 =<X.** %Rule 3

Let's consider another example when we use **cut (!)**
max(X,Y,Max) = where (max = X if X >= Y) and (max = Y if X < Y)
This code is translated in Prolog:

```
max(X,Y,X):-X>=Y.
max(X,Y,Y):-X<Y.
```

Now we can change the algorithm:
max(X,Y,Max) = where (if X >= Y) then (Max = X) otherwise (Max = Y)

```
max(X,Y,X):-X>=Y,!.
max(X,Y,Y).
Another way to write this algorithm using cut is:
max1(X,Y,Max):- X>=Y, Max=X, !; Max=Y.
```

Negation as failure: we want to represent in Prolog the sentence "Mary like all animals but snakes", how to do it?

```
like(mary, X):- snake(X), !, fail.
like(mary, X):- animal(X).
```

Not-Relation: not(Goal) is True if the Goal is False.

```
not(P):- P, !, fail; true.
```

⇒ **FOL**
⇒ **Relations/Predicates**
⇒ **Objects/Constants**
⇒ **Functions**
⇒ **Terms**
⇒ **Atomic Sentences**
⇒ **Complex Sentences**
⇒ **Quantifiers**
⇒ **Nested Quantifiers**
⇒ **Equality Symbol**
⇒ **Assertions**
⇒ **Queries**
⇒ **Numbers**
⇒ **Sets**
⇒ **Lists**

9 Economic, Philosophic and Ethic aspects of AI

Artificial Intelligence is a very broadly studied topic in different aspects. In this chapter we are going to see some of this aspects.

9.1 Why Now?

The last years we have seen an explosion of new technologies and especially those involving Artificial Intelligence, and the obvious question is *Why Now*?

- **Explosion of Computational Power**: Today computers are extremely powerful and cheap.
- **Parallel and Distributed Computation**
- **Big Data**: There are many interconnected devices that produce large amount of data. This all kind of data can be used to by machines to learn.
- **Better Algorithms and Models**: New technologies like Deep Learning etc.
- **Good reusable Open Source Software**
- **More and more people and device connected every day in Internet**: People and IoT interconnected every day

The rise of AI can have many implications in the job market. Many jobs are in risk and peoples can be substituted by machines and robots. An Oxford research in 2013 found that 47% of jobs in United States will be replaced by automation in 20 years. A more recent publication (2017) estimate that 77% of jobs in China and 69% in India are in risk to be replaced by AI. This lead to *greater inequalities* and as the history has shown the people/countries with a fast adoption of the technology have increase the income by 82%. On the other hand *Innovation* is important for the growth and AI is important for innovation. We don't want to stop the innovation but accelerate it in a wise manner. In the case of management, AI can be very helpful for the managers.

9.2 Philosophical Aspects of AI

There are many definitions about AI:

- **Weak AI**: Can we build machines that can act as they were intelligent?
- **Strong AI**: Can we build machines that are actually intelligent?

To ask this question we have to define first what is *intelligence*. In some degree we can claim that *weak AI* is possible and existing in our days.
Consciousness Objection to AI: Can machines *think*? Machines that pass the Turing test can be seen as machines that think but this is only a simulation of thinking

(**Chines Room Experiment**).
Other Objections to AI

1. **Theological Objection**: Only creatures of God can think
2. **Mathematical Objection**: Based on the Incompleteness Theorem of Godel that any strong theory contain theorems that cannot be proven in the formal theory itself.
3. **Various Disabilities**: Machines cannot have properties of people (Kindness, Happy, Empathy etc)
4. **Lady Lovelace's Objection**: Any machine do not have any pretension to do something, no predefined goals
5. **Informal Behaviours**: We do not have rules to describe informal behaviours of humans, how to teach them to machines?

   There are two opposite views:

1. **Biological Naturalism**: mental states are high-level emergent features that result from low-level physical processes in the neurons and is the unspecified properties of neurons that matters.
2. **Functionalism**: a mental state is any intermediate casual condition between input and output. Any two systems with isomorphic casual processes would have the same mental states. Therefore, a computer program could have the same mental states as a person. The assumption is that there is some level of abstraction below which the specific implementation does not matter. This fact is shown in the **Brain Replacement Experiment**.

9.3 Ethical Aspects of AI

Machine Ethics: Computational and Philosophical assumptions for machines that can take *autonomous* moral decisions. Questions like:
1- Autonomous Vehicles: which should be killed in case of an accident?
2- Medical Robots: should they said the truth to patients?
How to guarantee that machines do not take immoral decisions?!

- Simple Rules (Asimov Rules of Robotics) are not enough given contexts
- Simulation of moral decisions and actions of humans are not enough because humans often make bad and immoral decisions in this sens *machines need to be 'Saints'*.

  We would need to solve this 3 huge problems:

- A normative ethics which solve all existing moral dilemmas and which is accepted by most humans.
- A translation of such ethics in computational terms.
- The ability to incorporate commonsense reasoning in machines.

Another problem is that the human behaviour and machine behaviour are ruled by different laws: An example is the **The Trolley Problem**.
Trolley Problem is a version of ethical dilemma that urge to make a choice, *You are going sacrifice one to save five or ....* This dilemma is related with *Utilitarianism* that states that the morally correct decision is the one that maximises the well-being for the greatest number of people.