

ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SECOND CYCLE DEGREE/TWO YEAR MASTER IN
ARTIFICIAL INTELLIGENCE

Course

**FUNDAMENTALS OF ARTIFICIAL
INTELLIGENCE AND KNOWLEDGE
REPRESENTATION**

Held by

Prof. Michela Milano
Prof. Federico Chesani
Prof. Paolo Torroni

Written by

Giorgio De Simone

Academic Year 2024-2025

© 2025 Giorgio De Simone

Contents

Preface	1
I Module 1	3
1 Introduction to Artificial Intelligence	4
1.1 Introduction to AI techniques	4
1.2 AI applications	38
1.3 Trustworthy AI	55
2 Search strategies	65
2.1 Non-informed Search strategies	65
2.2 Informed Search strategies	81
3 Local search algorithms	103
3.1 Local search	103
4 Swarm Intelligence	124
4.1 Characteristics and applications	124
5 Games	147
5.1 Games with opponents as search	147
6 Planning	176
6.1 Introduction to Planning	176
6.2 Practising with Deductive Planning	197
6.3 Continuing on Planning	203
6.4 Planning based on Graph	233
6.5 Hierarchical Planning	249
6.6 Conditional Planning	261
7 Practical implementations of Planning	267
7.1 Linear Planning: STRIPS	267
7.2 Non linear Planning	272
8 Constraints	285
8.1 Constraints satisfaction	285
8.2 Exercises on Constraints	326

Appendices	333
A Search strategy	333
B Agent-based simulation	354
C Graph-based Planning	378
C.1 Graphplan	378
C.2 Planning Domain Definition Language	388
C.3 SATPLAN & Blackbox	390
C.4 Fast Forward	392
C.5 Exercises	394
Bibliography	405

Preface

The idea of writing these notes born with the intention of collecting the entire material of the *Second Cycle Degree/Two Year Master in Artificial Intelligence* offered by *University of Bologna*, dividing it in different volumes depending on the different courses held with the purpose of creating a true series of books.

Specifically here I wrote the notes related to the *Fundamentals of Artificial Intelligence and Knowledge Representation* course – held by Professor Michela Milano, Professor Mauro Gaspari, Professor Paolo Torroni and Professor Federico Chesani – which introduces the fundamental principles and methods used in Artificial Intelligence to solve problems, with a special focus on the search in the state space, planning, knowledge representation and reasoning, and on the methods for dealing with uncertain knowledge.

Prerequisites: user-level knowledge of a high-level programming language, in order to successfully understand case studies and applications presented during the lessons.

The book itself is divided in three modules.

- Module 1.
 - Introduction to Artificial Intelligence: historical perspective, main application fields, introduction to knowledge-based systems and architectural organization.
 - Problem-solving in AI: representation through the notion of state, forward e backward reasoning, solving as a search and search strategies. Games. Constraint satisfaction problems.
 - Local Search methods, meta heuristics, solving through decomposition, constraint relaxation, branch-and-bound techniques.
 - Introduction to Planning, Linear planning, partial order planning, graph-based methods (GraphPlan), Scheduling.
 - Below it is shown a schedule of the Module content, schedule that is suggested to follow (but that is not mandatory for the purpose of understanding).
 1. Chapter 1.
 2. Chapter 2.
 3. Appendix A.
 4. Chapter 3.
 5. Chapter 4.

- 6. Appendix B.
 - 7. Chapter 5.
 - 8. Chapter 6.
 - 9. Appendix C.
 - 10. Chapter 7.
 - 11. Chapter 8.
- Module 2.
 - Introduction to knowledge representation and reasoning.
 - Representing Terminological Knowledge: semantic networks, description logics, foundation of ontologies.
 - Representing actions, situations, and events.
 - Rule-based systems: Prolog and extensions, meta-interpreters, DCG, planning in prolog, Prolog for temporal reasoning with the Event Calculus, LPAD.
 - Forward chaining and RETE, Drools.
 - Module 3.
 - Uncertainty.
 - Probabilistic Reasoning.

At the end of the book, the reader knows the main knowledge representation techniques and reasoning methods that underlie artificial intelligence problem solving. The reader is able to develop simple solvers for artificial intelligence systems.

Part I

Module 1

Chapter 1

Introduction to Artificial Intelligence

1.1 Introduction to AI techniques

What is ‘Intelligence’?

‘Intelligence’ has been defined in many ways, including: the capacity for logic, understanding, self-awareness, learning, emotional knowledge, reasoning, planning, creativity, critical thinking and problem solving. More generally, it can be described as the ability to perceive or infer information, and to retain it as knowledge to be applied towards adaptive behaviors within an environment or context.

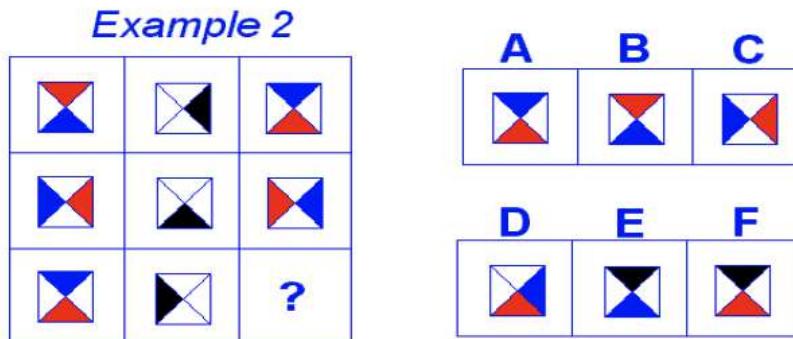


Figure 1.1

Looking at the image above is important to say that for humans is very intuitive to solve problems like the one presented while for a *machine* is extremely complex.

Turing: Can machine think?

In his article *Computing Machinery and Intelligence* (1950), Alan Turing presented is famous *Test*.

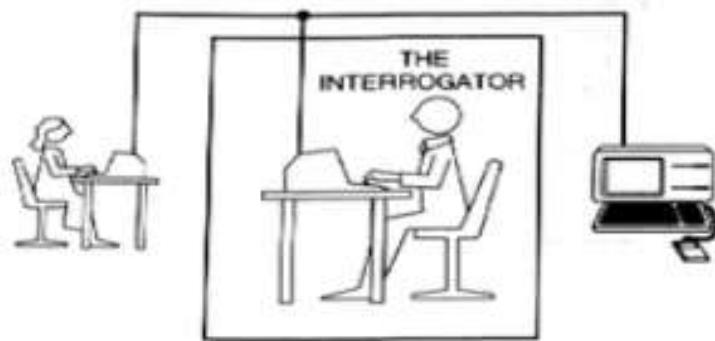


Figure 1.2

- Problem of defining a universal machine and defining thinking (ambiguous).
- Imitation game.
- Three actors: a human, a computer, a interrogator in a different room. The interrogator should classify the human and the machine.
- Is it possible that a computer can mislead the interrogator and be classified as a human?
- Various objections to this test.

Turning test and AI

- Computers should have the following abilities:
 - natural language processing;
 - knowledge representation;
 - automatic reasoning;
 - machine learning.
- Beyond Turning test¹ – Physical interactions: Robotics, Artificial Vision, Speech, Movement, etc.
- In 2014, a chatbot (Eugene Goostman), mimiking the answer of a 13 years old boy, has succeeded the test.
- Cleverbot (Machine Intelligence Prize 2010), SIRI (Apple), Cortana (Microsoft), Alexa (Amazon) etc.
- Often built indexing previous conversations. What is missing is coherence, state of dialogue.
- Methodological perspective: engineering approach fostering emulation more than achievement of real intelligence in limited domains.

¹AI goes beyond the Turing test, there are other aspects to consider.

Beyond Turing test

Mathematical puzzles: require deep language understanding, common-sense, reasoning capabilities, multimodal analysis.

Geometrical problem solving – Allen insitute.

Example: *Using integer numbers from 2 to 7, fill in the free tiles such that the sum of the three numbers around the black circle is 11.²*

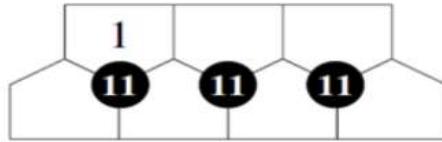


Figure 1.3

Turing forecast: true or false?

'I believe that in about fifty years time it will be possible to programme computers, with a storage capacity of about 10^9 , to make them play the imitation game so well that an average interrogator will not have more than 70% chance of making the right identification after five minutes of questioning... I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machine thinking³ without expecting to be contradicted.'⁴

AI System Classification

Weak AI vs Strong AI:

- *Weak AI:*
 - research question: is it possible to build systems that act *as if* they were intelligent?
- *Strong AI:*
 - research question: is it possible to build systems that *are* intelligent?
That *have* conscious minds, wills and sentiments?

General AI vs Narrow AI:

²It is a very easy problem for humans (but it requires visual skills, reasoning abilities *etc.*). Machines are still very far from resolving this kind of problem; they are able to understand the text and the request but still not able to reach the solution. Between the comprehension of this problem and the solution there are humans; this is the gap between us and machines. An huge gap!

³The first half of Turing sentence is already realized while the second is still very far; the concept of 'Machine Thinking' is no yet realized.

⁴From: A. Turing, 'A Computing machinery and intelligence', Mind, vol. 59, n. 236, pp. 433-460, 1950.

- *General AI* refers to systems which are able to cope with any generalized tasks which is asked of it, much like a human;
- *Narrow AI* refers to AI which is able to handle just one particular task. AI systems display a certain degree of intelligence in a particular field but they remain computer systems that perform highly specialized tasks for humans, within that narrow field.

Artificial Intelligence

- Born in 1956 (Minsky, McCarthy, Shannon, Newell, Simon).
- The study is to proceed on the basis of the conjecture that *every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.* [McCarthy 1955]
- Which definition of *Intelligence*? Which definition of *Artificial Intelligence*? Some definitions:
 - study of how to make computers do things that humans do better;⁵
 - study of how to build computers that pass the Turing test (reasoning, natural language understanding, learning). If situated in an environment also perception, vision, movement, robotics;
 - other definitions of AI do not link directly to human intelligence but rather on the ability of interacting with and adapting to an environment.

Short history

- 1943-1956; the birth of AI:
 - neural networks; chess programs; theorem provers.
- 1952-1969; initial enthusiasm and big expectations:
 - general problem solver; LISP language; neural networks.
- 1966-1974; down to earth:
 - some programs were not adequate (ELIZA, syntactic translations, neural networks), other were too computationally expensive (combinatorial explosion).
- 1969-1979; knowledge based systems:
 - deep knowledge in a limited domain; expert systems.
- 1980-1988; AI enters in industry:
 - commercial successful expert systems; fifth generation project in Japan; companies working in AI, research funds.
- 1988-2010; web and internet era:

⁵Transitory definition (see chess).

- decision support systems, robotic agents, natural language.
- 2010-today; machine learning and deep learning:
 - big data and massive computing power have enabled deep networks to work properly; augmented intelligence and perception; industrial interest, industry 4.0.

Two main AI approaches

- *Top-down or symbolic AI:*
 - symbolic representation of knowledge;⁶
 - logics, ontologies, rule based systems, declarative architecture;⁷
 - human understandable models.
- *Bottom up, or connectionist approach:*
 - neural networks; knowledge is not symbolic and it is ‘*encoded*’ into connections between neurons;⁸
 - concepts are learned by examples;
 - non understandable by humans.

Reasoning and Logic

- *Deductive reasoning* (Aristotelian syllogism - deductive logic):
 - from: ‘*All men are mortal and Socrates is a man*’ then: ‘*Socrates is mortal*’;
 - okay, but it does not allow us to ‘*learn*’ new knowledge.
- *Inductive reasoning* (Learning, reverse deduction):
 - from the observation that several birds flying then *all* the birds fly (and penguins?);
 - production of new knowledge at the expense of correctness.
- *Hypothetical or abductive reasoning* (dual of deductive):
 - from the observation of death of Socrates and knowing that *all* men are mortal hypothesizes that Socrates is a man. (and if he were a cat?);
 - it dates back to the causes through observation of the effects at the expense of correctness.
- *Reasoning by analogy* (metaphorical, case-based):

⁶We are talking about symbols that have a semantic form.

⁷For example the kind of reasoning like *if () ... then ()* it's proper of this field.

⁸Neural network for example use complex mathematical models. There is even the *Universal Approximation Theorem* which substantially says that whatever function of a certain sector can be approximated and that the approximation grade depends on the number of neurons involved.

- it does not require a model or a lot of data, but it uses the principle of similarity; Socrates and John ‘resemble’: if Socrates loves philosophy then John loves philosophy;
 - K-Nearest-Neighbor and Support Vector Machine (SVM).
- *Constraint reasoning and optimization*: using constraints, probability, statistics (Bayes theorem).

Logic, automatic proof and Prolog

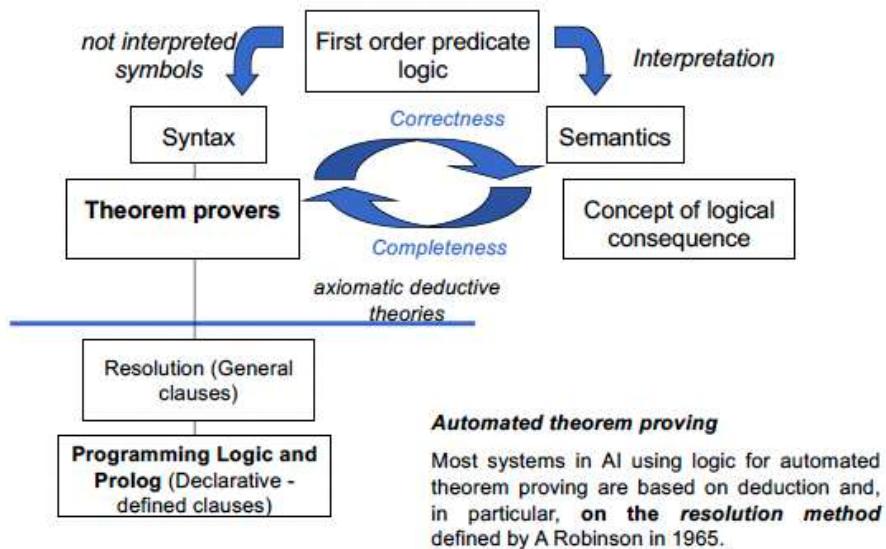


Figure 1.4

Prolog is a programming language associated to AI and to axiomatic-deductive computational linguistics.

There are other languages that concerns also the other kind of reasonings introduced previously.

Different algorithm interpretations

Level of generality and ‘intelligence’ increases:

- algorithm = data structures and instructions;
- algorithm = logic (knowledge) + control (engine inference);
- algorithm = examples (experience) + machine learning.

Declarative languages and Prolog

- Algorithm = logic + control.

- Problem knowledge is independent from its use:
 - * it expresses WHAT and not HOW;
 - * high modularity and flexibility;
 - * design scheme at the base of most KNOWLEDGE BASED SYSTEMS (Expert Systems).
 - LOGIC: knowledge about the problem determines correctness and efficiency.
 - CONTROL: solution strategy determines the efficiency.
- PROLOG: PROgramming in LOGic:
 - it is the best known Logic Programming language. It is built on the ideas by R. Kowalski and the first realization by A. Colmarcheur (1973).

Prolog Program

A PROLOG PROGRAM is a set of *horn* clauses representing:⁹

- FACTS about the object under examination and the relationships;
- RULES on objects and relationships (*if ... then*);
- GOALS (clauses with no head), on the basis of definite knowledge.

Example: two individuals are colleagues if they work for the same company.

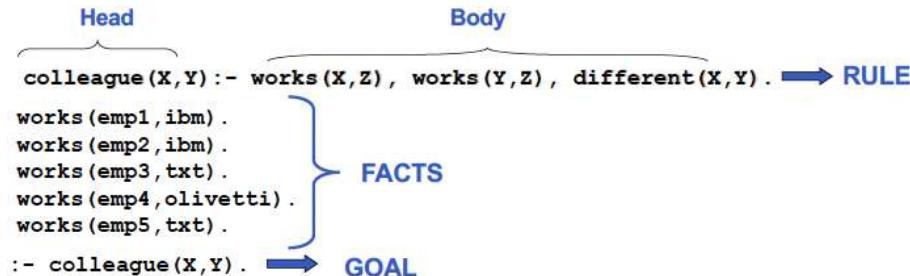


Figure 1.5

⁹A *horn* clause is a specific type of logical clause used in formal logic, particularly in the context of logic programming languages like Prolog. Horn clauses have a specific structure and are commonly used to express rules and relationships in a logical form. A horn clause is typically expressed in one of the following forms: *positive horn clause* ↔ if A is true, then B is true (in this form, A is the antecedent (or body) of the clause, and B is the consequent (or head) of the clause. It expresses a rule that if A is satisfied, then B must also be satisfied); *negative horn clause* ↔ if A is false, then B is true (similar to the positive horn clause, this form states that if A is not satisfied (i.e. it is false), then B must be satisfied). Horn clauses are particularly useful in logic programming because they lend themselves well to efficient inference and reasoning. The most common use of horn clauses is in Prolog programming, where they are used to define rules and relationships that can be queried to derive logical conclusions based on the specified facts and rules.

Let's explain in detail:¹⁰ the Prolog code provided represents a simple rule-based system to determine whether two individuals are colleagues based on the information about where they work. Let's break down the code and the goal.

1. Rule definition:

```
colleague(X, Y) :- works(X, Z), works(Y, Z), different(X, Y).
```

This rule states that two individuals, X and Y, are considered colleagues if the following conditions are met:

- `works(X, Z)`: individual X works for a company Z;
- `works(Y, Z)`: individual Y also works for the same company Z;
- `different(X, Y)`: X and Y are different individuals (not the same person).

2. Facts:

the provided facts specify where several individuals work:

```
works(emp1, ibm).
works(emp2, ibm).
works(emp3, txt).
works(emp4, olivetti).
works(emp5, txt).
```

These facts establish the employment relationships between employees (*e.g.* emp1 works at IBM, emp3 works at TXT).

3. Goal:

```
:- colleague(X, Y).
```

This goal is asking Prolog to find pairs of individuals (X and Y) who are colleagues based on the `colleague/2` rule defined earlier.

Now, let's analyze the goal and see what Prolog can deduce from the provided facts and rule:

- Prolog will attempt to find pairs of X and Y such that they meet the conditions specified in the `colleague/2` rule;
- it will look for pairs of individuals who work for the same company (Z) and are different individuals (X and Y are not the same person).

Given the provided facts, here are the possible solutions to the goal:

- X = emp1, Y = emp2: both emp1 and emp2 work at IBM and are different individuals;
- X = emp2, Y = emp1: this is the same as the previous solution but with the order of X and Y reversed;

¹⁰Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is intended primarily as a declarative programming language: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query (goal) over these relations. Prolog has declarative programming techniques *i.e.* do not establish *how* to solve a problem, but '*only*' the problem itself.

- $X = \text{emp3}, Y = \text{emp5}$: both emp3 and emp5 work at TXT and are different individuals.

These are the pairs of individuals who meet the criteria (and that Prolog will find) for being colleagues based on the provided facts and the `colleague/2` rule.

Example: sum of two integer numbers.

- Declarative definition.¹¹

```
sum(0, X, X). % -> FACT
sum(s(X), Y, s(Z)) :- sum(X, Y, Z). % -> RULE
```

- `sum` is a non interpreted symbol.
- Integer numbers defined with the successor structure `s(X)`.
- Recursive definition.
- Many possible queries (algorithms):

```
:- sum(s(0), s(s(0)), Y). % -> QUERY
:- sum(s(0), Y, s(s(s(0)))).
:- sum(X, Y, s(s(s(0)))).
:- sum(X, Y, Z).
:- sum(X, Y, s(s(s(0)))), sum(X, s(0), Y).
```

Let's explain in detail: the Prolog code you provided defines a predicate `sum/3` that represents the addition of two natural numbers using Peano arithmetic, which is a way to represent natural numbers in terms of a successor function (`s`) and zero (0). This code defines addition in a recursive manner.

1. Base Case:

```
sum(0, X, X). % -> 0 + x = x
```

This rule states that the sum of zero and any number X is X. This is the base case of the addition operation.

2. Recursive Case:

```
sum(s(X), Y, s(Z)) :- sum(X, Y, Z). % s(x) = x + 1
% if x + y = z then s(x) + y = s(z)
```

This rule defines the addition of two numbers using the successor function (`s`). It states that the sum of `s(X)` (the successor of `X`) and `Y` is `s(Z)` if the sum of `X` and `Y` is `Z`. This is the recursive case, where addition is defined in terms of smaller additions until the base case is reached.

Here how the addition predicate works:

- `sum(0, X, X)` serves as the base case, where adding zero to any number `X` results in `X`;

¹¹We want to define the sum without using any mathematical formula.

- for the recursive case, `sum(s(X), Y, s(Z))` means that if you want to add the successor of X to Y , you first add X to Y , which results in Z , and then take the successor (s) of Z . This is equivalent to regular addition in natural numbers, where $s(X)$ represents $X+1$.

Here are some examples to illustrate how this addition predicate works:

- `sum(0, 3, 3)` represents adding 0 to 3, which is 3;
- `sum(s(s(0)), s(s(s(0))), s(s(s(s(s(0))))))` represents adding 2 to 3, which is 5;
- `sum(s(s(s(0))), 0, s(s(s(0))))` represents adding 3 to 0, which is 3.

Let's explain better: the Prolog code provided defines a predicate called `sum/3` that calculates the sum of two numbers represented using Peano arithmetic. Peano arithmetic is a way of representing natural numbers using only two operations: zero (0) and the successor function ($s/1$), where $s(N)$ represents the successor of the number N . Let's break down the code.

```
% FACT: Base case
sum(0, X, X).
```

This is a Prolog fact. It defines the base case for the `sum/3` predicate. It says that the sum of 0 and any number X is X . In other words, `sum(0, X, X)` states that when you add 0 to any number X , the result is X . This serves as the base case for the recursive definition of addition.

```
% RULE: Recursive case
sum(s(X), Y, s(Z)) :- 
    sum(X, Y, Z).
```

This is a Prolog rule that defines the recursive case for the `sum/3` predicate. It states that the sum of $s(X)$ and Y is $s(Z)$ if the sum of X and Y is Z . In other words, it's saying that to calculate the sum of $s(X)$ and Y , you first calculate the sum of X and Y , and then you increment the result by one (*i.e.* $s(Z)$).

Here's how the code works with an example.

1. `sum(0, X, X)`. is the base case. If you try to calculate the sum of 0 and any number X , the result is X .
2. The recursive case `sum(s(X), Y, s(Z)) :- sum(X, Y, Z)` handles the addition of non-zero numbers. It says that to calculate the sum of $s(X)$ and Y , you first calculate the sum of X and Y , and then you increment the result by one ($s(Z)$).

For example, using this code:

- `sum(s(s(0)), s(s(0)), Result)` will compute $2 + 2$ and bind `Result` to $s(s(s(s(0))))$, which represents 4 in Peano arithmetic;
- `sum(s(s(s(0))), s(0), Result)` will compute $3 + 1$ and bind `Result` to $s(s(s(s(0))))$, which is also 4.

This code defines addition in a way that aligns with the principles of Peano arithmetic and uses recursion to perform the addition operation.

Finally implementing a complete code like:

```
% FACT: Base case
sum(0, X, X).

% RULE: Recursive case
sum(s(X), Y, s(Z)) :- sum(X, Y, Z).

% GOAL/QUERY
:- sum(s(s(s(0))), s(0), Result)
```

Let's go through the recursive steps to calculate `sum(s(s(s(0))), s(0), Result)` step by step until we reach the base case. We'll start with the original query and break it down.¹²

```
sum(s(s(s(0))), s(0), Result).
```

1. This query calls the `sum/3` predicate with the first argument as `s(s(s(0)))`, the second argument as `s(0)`, and an unbound variable `Result` to store the result.
2. According to the rule `sum(s(X), Y, s(Z)) :- sum(X, Y, Z)`, we move to the next step and query `sum(s(s(0)), s(0), IntermediateResult)`, where `Result = s(IntermediateResult)`.
3. Apply RULE: `sum(s(0), s(0), SubIntermediateResult)`
where `IntermediateResult = s(SubIntermediateResult)`.
4. Apply RULE: `sum(0, s(0), SubSubIntermediateResult)`
where `SubIntermediateResult = s(SubSubIntermediateResult)`. At this point, we've reached the base case according to the fact `sum(0, X, X)`, the sum of 0 and any number `X` is `X`. Therefore, `SubSubIntermediateResult` is bound to `s(0)`.
5. Now, we start unwinding the recursive calls. Using the intermediate results, we can calculate the sum for each level:

```
Result = s(IntermediateResult)
= s(s(SubIntermediateResult))
= s(s(s(SubSubIntermediateResult)))
= s(s(s(s(0))))
```

6. So, the result of `sum(s(s(s(0))), s(0), Result)` is `s(s(s(0)))`, which represents $3 + 1 = 4$ in Peano arithmetic.

The code:

```
% FACT: Base case
sum(0, X, X).

% RULE: Recursive case
```

¹²The rule `sum(s(X), Y, s(Z)) :- sum(X, Y, Z)`. is mathematically equivalent to $\text{sum}(X, Y, Z) \implies \text{sum}(s(X), Y, s(Z))$ so given `sum(s(X), Y, s(Z))` we can reconstruct the chain referring to `sum(X, Y, Z)` which implies `sum(s(X), Y, s(Z))`.

```
sum(s(X), Y, s(Z)) :- sum(X, Y, Z).
```

%GOAL/QUERY
`: - sum(X, Y, Z)`

When you issue the query `: - sum(X, Y, Z)` you are essentially asking the Prolog interpreter to find all possible solutions for X, Y, and Z that satisfy the `sum/3` predicate. Prolog will search for solutions based on the facts and rules defined in the program.

Here are a few examples of solutions you might get:

1. $X = 0, Y = s(s(0)), Z = s(s(0))$ (indicating $0 + 2 = 2$ in Peano arithmetic);
2. $X = s(s(s(0))), Y = 0, Z = s(s(s(0)))$ (indicating $3 + 0 = 3$ in Peano arithmetic);
3. $X = s(0), Y = s(0), Z = s(s(0))$ (indicating $1 + 1 = 2$ in Peano arithmetic).

The specific solutions will depend on the relationships defined in your knowledge base and the particular values of X, Y, and Z that satisfy those relationships.

These are just a few examples of the solutions that satisfy the query `: - sum(X, Y, Z)`. Prolog will generate all possible combinations of values for X, Y, and Z that satisfy the `sum/3` predicate based on the rules and facts defined in the code.

Running the following code in a Prolog interpreter:

```
% FACT: Base case
sum(0, X, X).

% RULE: Recursive case
sum(s(X), Y, s(Z)) :- sum(X, Y, Z).

%GOAL/QUERY
: - sum(X, Y, s(s(s(0)))).
```

It will find and display the solutions for the goal/query. In this case, it's calculating the sum of X and Y such that the result is `s(s(s(0)))`, which represents the number 3 in Peano arithmetic.

Here are the possible solutions for the query `: - sum(X, Y, s(s(s(0))))`:

- $X = 0, Y = s(s(s(0)))$ (indicating $0 + 3 = 3$ in Peano arithmetic);
- $X = s(0), Y = s(s(0))$ (indicating $1 + 2 = 3$ in Peano arithmetic);
- $X = s(s(0)), Y = s(0)$ (indicating $2 + 1 = 3$ in Peano arithmetic);
- $X = s(s(s(0))), Y = 0$ (indicating $3 + 0 = 3$ in Peano arithmetic).

These are the possible solutions that satisfy the query `: - sum(X, Y, s(s(s(0))))`. Prolog will generate all valid combinations of values for X and Y that result in `s(s(s(0)))` when summed using the `sum/3` predicate based on the rules and facts defined in the code.

Deduction and induction in logic programming

Note: do not try to understand exactly what is shown below; we want just to show not in detail that with logic programming (we will see in details ahead) is possible to induce the machine to make *deduction*, *induction* (and even *abduction*).

Deduction:¹³

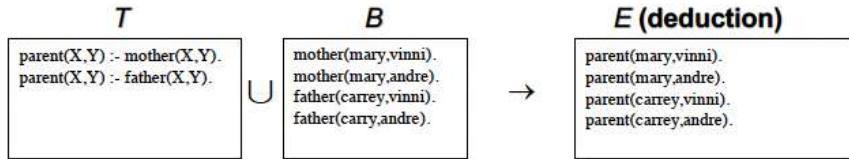


Figure 1.6

Induction:¹⁴

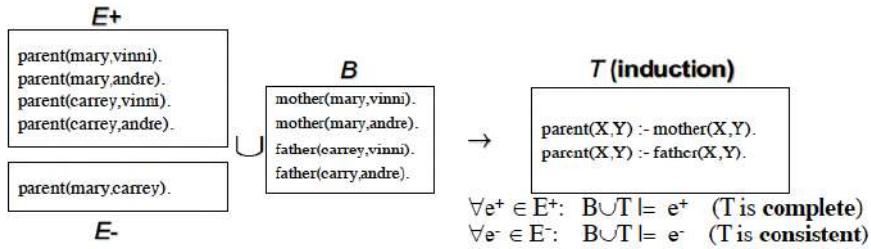


Figure 1.7

Referring to the image above we want just clarify some aspects (just for general knowledge).¹⁵

- The symbol \models in logic represents the concept of ‘entailment’. In logical notation, it is used to indicate that a particular statement or set of statements logically implies or entails another statement. In other words, if you have a set of premises or axioms and you can derive a conclusion from them using the rules of logic, you can say that the conclusion is entailed by the premises, denoted as:

- Premises \models Conclusion

This symbol is commonly used in formal logic, mathematical logic, and philosophy to express the idea of logical implication or deduction. It signifies that if the premises are true, then the conclusion must also be true based on the logical relationships between them.

¹³We extract already existent informations.

¹⁴We produce new informations. The machine produces new informations; this does not mean that the information is correct, but just that is produced.

¹⁵They will be seen in detail ahead or in other courses.

- In logic, the term ‘*consistent*’ refers to a property of a set of statements or propositions. A set of statements is said to be consistent if it is possible for all the statements in that set to be true simultaneously without leading to a logical contradiction or inconsistency. In other words, a set of statements is consistent if there is no way to derive a statement and its negation (opposite) from the set.
- In logic, the term ‘*complete*’ can have different meanings depending on the context. Two common uses of completeness in logic are related to formal systems and theories (let just mention the first one):
 - completeness of a Formal System (Logical Completeness): in the context of formal logic, a formal system is said to be complete if, within that system, it is possible to prove (or disprove) every statement or proposition that is true (or false) in the domain of that system. In other words, a formal system is complete if it can fully capture and address all valid inferences within its scope. Gödel’s completeness theorem and the completeness of certain proof systems are examples of this concept.

- With the given facts and rules in Prolog (case E^- of the image above):

```

mother(mary, vinni).
mother(mary, andre).
father(carrey, vinni).
father(carry, andre).
parent(mary, carrey).

parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

```

You can have the following relationships.

1. parent(mary, vinni) (Mary is the mother of Vinni).
2. parent(mary, andre) (Mary is the mother of Andre).
3. parent(carrey, vinni) (Carrey is the father of Vinni).
4. parent(carry, andre) (Carrey is the father of Andre).
5. parent(mary, carrey) (Mary is the parent of Carrey). This is true because is given from the beginning as a fact, the formula $\forall e^- \in E^- : B \cup T | = e^-$ is respected.¹⁶

Abduction in logic programming

NOTE: as said before do not try to understand and interpret exactly what is shown below; just try to learn the general aspects (*i.e.* that with the following syntax we can make the machine make *abduction*).

Abduction:¹⁷

¹⁶Note that also if this information goes versus the common sense (if `mary` and `carry` are the parents how is it possible that `mary` is parent of `carrey`? Remember that in a previous note we said that induction produces new information, but not necessarily this information is *correct* [correct in comparison to the common sense; but what is the common sense for a machine?]).

¹⁷Make hypothesis.

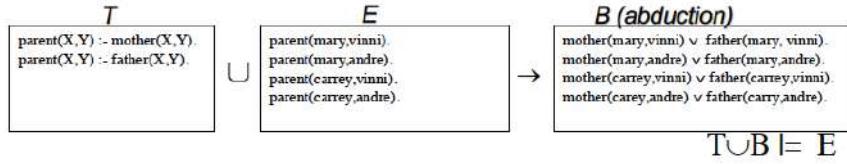


Figure 1.8

Often we use integrity constraints to control hypothesis.

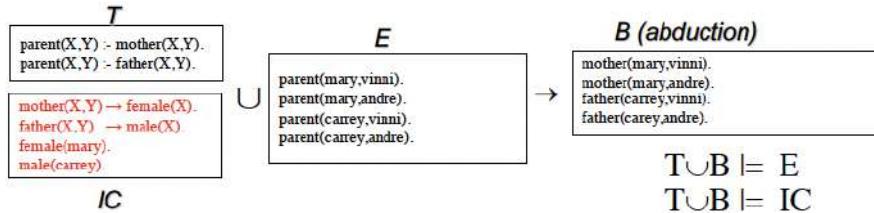


Figure 1.9

Prolog

- Great revolutionary idea.
- The language is not linked to Von Neumann architectures.
- European versus American.
- Japanese Fifth generation project in the 80 (failed).
- Technology has made gigantic steps in the meantime...
- Why Prolog has not spread as believed in the 80s?
- Many applications, ideas, research, extensions Prolog in AI, but not all.

Many Prolog implementations.

- *SWI Prolog*: widely used and well integrated with Semantic Web:
<http://www.swi-prolog.org>
- *tuProlog*: Java based Prolog used for internet applications developed by Unibo.
<http://apice.unibo.it/xwiki/bin/view/Tuprolog/>

Machine learning

Learning from experience, learning from one's mistakes, learning from teachers, learning by imitation, is a unique feature of intelligence.
Empiricist approach (which opposes to the rationalist approach).

Definition 1.1.1 (Learning).

Learning is constructing or modifying representations of what is being experienced [Michalski 1986].

Definition 1.1.2 (Learning).

Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time [Simon 1984].

Machine learning is one of the hot topics of Artificial Intelligence.

All the most important ICT companies are investing money on ML and hire people with a background in ML: Google, Facebook, IBM, Baidu, Disney.... WHY?... They have many data... And want to extract value from data.

The era of big data...

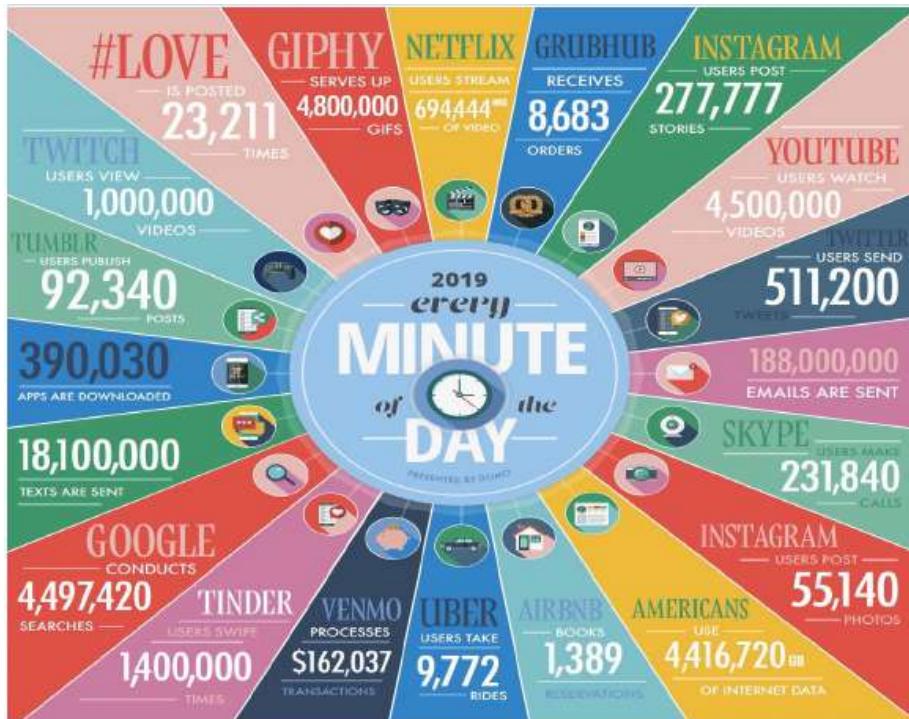


Figure 1.10

Definition 1.1.3 (Supervised Learning).

Supervised Learning:

- it starts from a set of examples given by a teacher ('training set');

- it solves problems of classification/regression (e.g. patterns recognition).

Definition 1.1.4 (Unsupervised Learning).

Unsupervised Learning:

- through observations and discovery;
- from the outside it does not get any help, but in the system that is in charge of analysing the informations available, to classify and structure them and to discover patterns;
- clustering/data mining.¹⁸

Definition 1.1.5 (Reinforcement Learning).

Reinforcement Learning:

- learning optimal behaviour from past experiences;
- observation of good and bad choices made (through rewards and punishments), and modify the behaviour accordingly;
- mainly learns from mistakes;
- finds many applications in robotics.

Machine learning applications

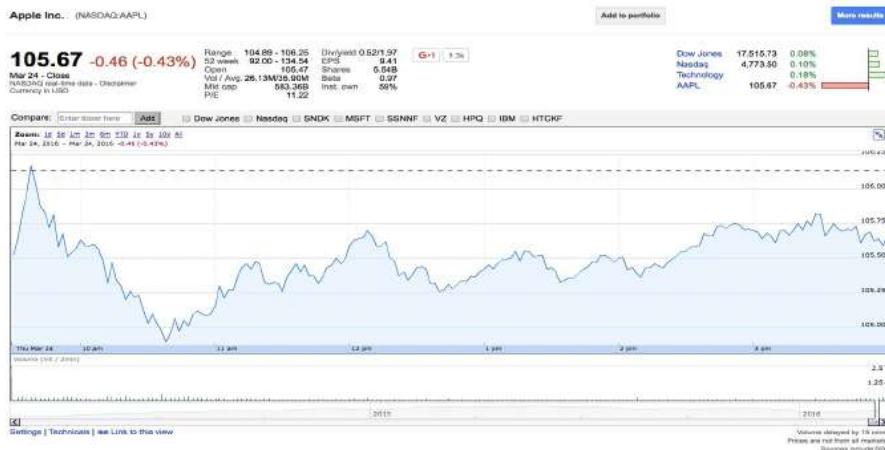
What can we do with machine learning?

- Classify incoming e-mails as spam or not.

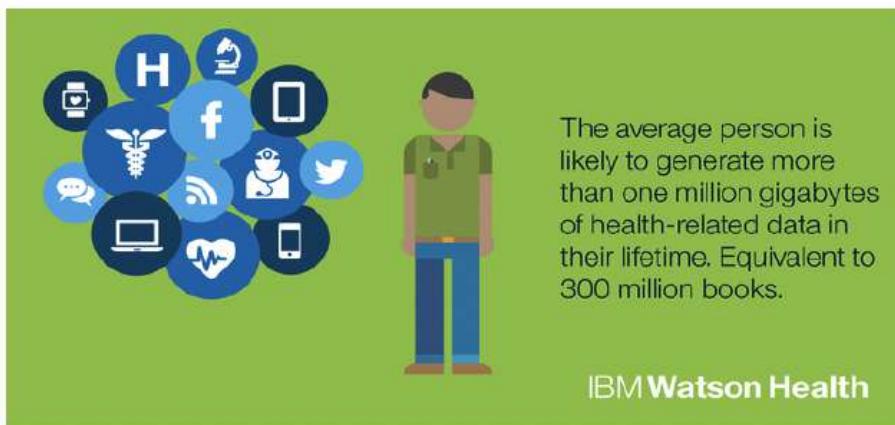


¹⁸Clustering and data mining are two related concepts in the field of data analysis and artificial intelligence. Clustering (Clustering analysis): clustering is a data analysis technique that aims to divide a set of data into groups or clusters, so that objects within the same cluster are similar to each other, while objects in different clusters are dissimilar. The goal is to find hidden structures or patterns in the data without the need to label the data in advance. It is often used to explore and understand the internal structure of data, as well as to organize large amounts of information in a more manageable way. Data mining: data mining is a process that involves the discovery of patterns, relationships, or meaningful information within large datasets. This process uses a variety of statistical, mathematical, and algorithmic techniques to uncover hidden knowledge or insights in the data. Data mining can encompass various activities, including clustering, classification, regression, association, and more. It is widely used in fields such as marketing, scientific research, cybersecurity, and many others to make data-driven, informed decisions. In summary, clustering is a specific technique within data mining that focuses on dividing data into homogeneous groups, while data mining is the broader process of extracting useful information from data, which can involve a range of different techniques, including clustering.

- Apple shares price prediction.



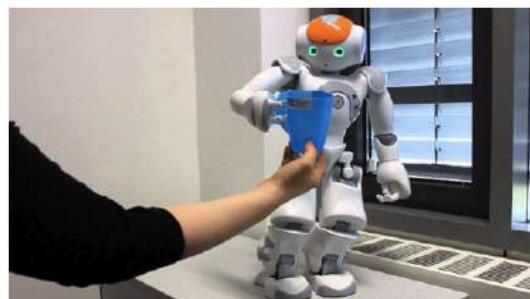
- Diagnosis based on exams of a patient.



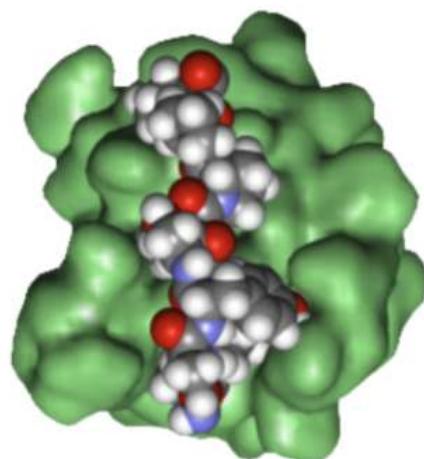
- Understand VAT numbers.

80322-4129 80206
 40004 14310
 37872 05753
 5502 75216
 35460 44209

- Teach a robot to grab a cup.



- Design a molecule with specific properties.



- Text translation.



- Convert a voice in text.



- Automatically write the caption to a figure.



- Beat the world champion in a game.



Machine learning task: Classification

Definition 1.1.6 (Classification).

Classification is the task of approximating a mapping function f from input variables X to discrete/categorical output variables y .

The output variables are often called labels or categories or classes.

The mapping function predicts the class or category for a given observation.
Supervised learning task.

- a classification problem requires that examples be classified into one of two or more classes;
- a classification can have real-valued or discrete input variables;
- a problem with two classes is often called a two-class or *binary classification problem* while a problem with more than two classes is often called a *multi-class classification* problem;

- a problem where an example is assigned multiple classes is called a multi-label classification problem;
- the *classification accuracy* is the percentage of correctly classified examples out of all predictions made.

Machine learning task: Regression

Definition 1.1.7 (Regression).

Regression is the task of approximating a mapping function f from input variables X to a continuous output variable y .

A continuous output variable is a real-value, such as an integer or floating point value. These are often quantities, such as amounts and sizes.

Supervised learning task:

- a regression problem requires the prediction of a quantity;
- a regression can have real valued or discrete input variables;
- a problem with multiple input variables is often called a *multivariate regression* problem;
- a regression problem where input variables are ordered by time is called a *time series forecasting* problem;
- because a regression predictive model predicts a quantity, the skill of the model must be reported as an error in those predictions, like the RMSE.¹⁹

Machine learning task: Clustering

Definition 1.1.8 (Clustering).

Clustering is the problem of organizing objects into groups whose members are similar in some way.

Clustering determines the intrinsic grouping in a set of *unlabeled data*.

Clustering is the main *unsupervised learning task*.

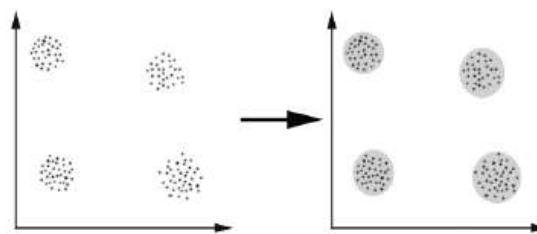


Figure 1.11

¹⁹RMSE stands for ‘Root Mean Square Error’. It is a common metric used in statistics and machine learning to measure the average magnitude of errors or the differences between predicted and observed (actual) values in a dataset. RMSE is often used to assess the accuracy of a predictive model, such as a regression model. The formula for calculating RMSE is as follows: $RMSE = \sqrt{\frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n}}$.

Clustering types:

- *hard clustering*: each element can be assigned to one and only one group;
- *soft or fuzzy clustering*: each element can belong to more than one group.

Most widely used algorithm for hard clustering k-means MacQueen 1967.

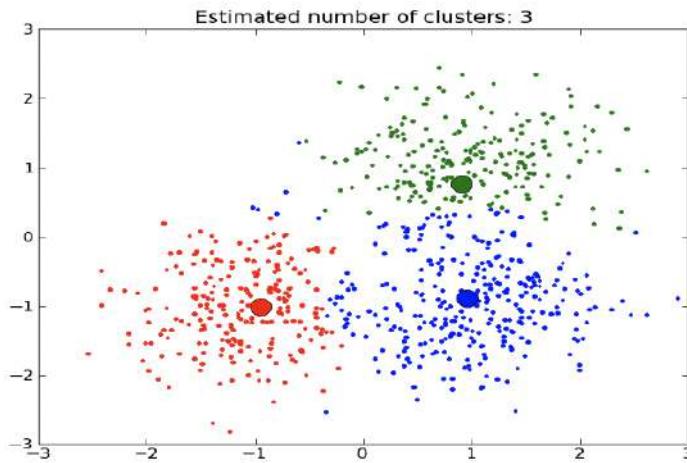


Figure 1.12

Symbolic and sub-symbolic learning

Regarding the *sub-symbolic learning* we can say that it is often more powerfull than *symbolic* but at the end of the working-day we are not able to obtain the laws that are learnt from data.

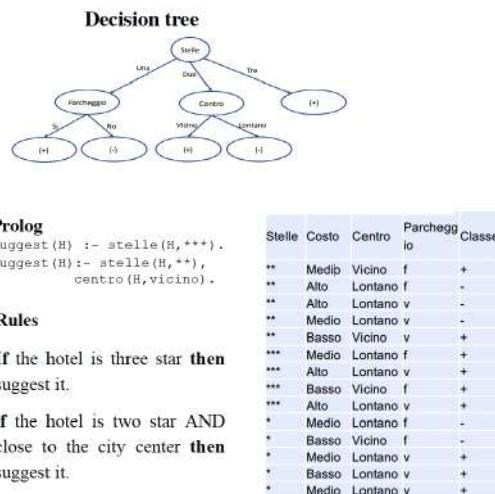


Figure 1.13

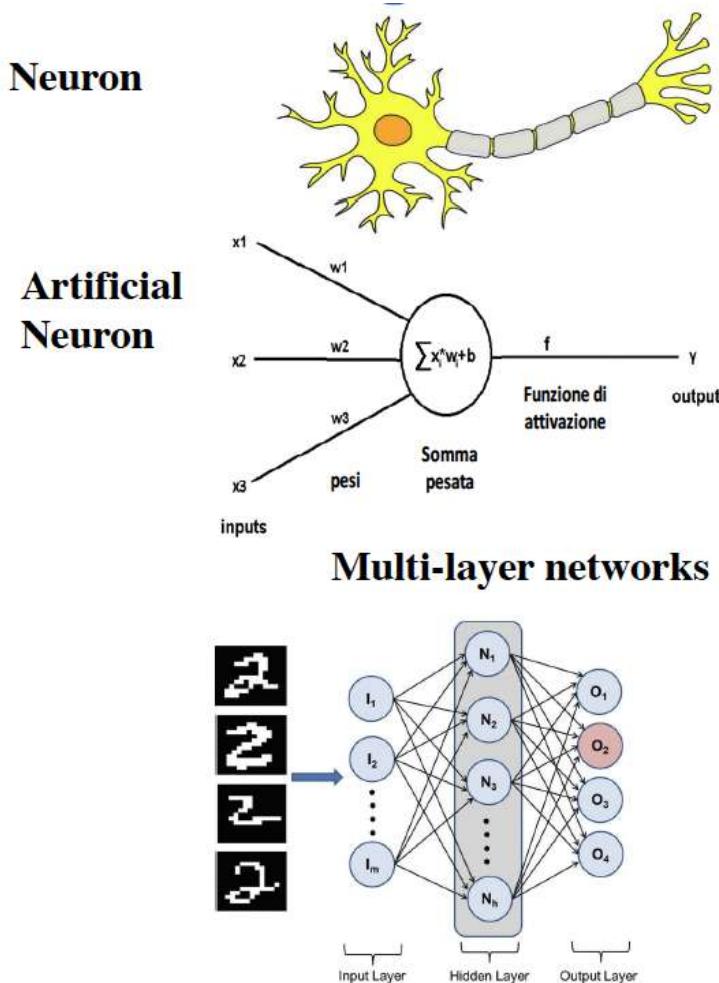


Figure 1.14

Regarding *artificial neurons* we can say that the only things that change in the illustrated models are the *weights* and thanks to the *Universal Approximation Theorem*²⁰ we can approximate any function depending on the numbers of neurons that we have in our network.

²⁰The Universal Approximation Theorem is a fundamental result in the field of machine learning and artificial neural networks. It states that a feedforward neural network with a single hidden layer containing a finite number of neurons (or units) can approximate any continuous function to a desired level of accuracy, provided that the network has a sufficient number of neurons in the hidden layer. In simpler terms, the theorem asserts that a neural network with just one hidden layer can, in principle, learn and approximate any function if it has enough neurons in that hidden layer. This is a powerful and important result because it suggests that neural networks are highly expressive and can capture complex and non-linear relationships in data.

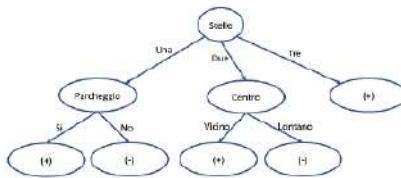
Symbolic learning: decision trees

Possible training set of hotels:

Stelle	Costo	Centro	Parcheggio	Classe
**	Medio	Vicino	f	+
**	Alto	Lontano	f	-
**	Alto	Lontano	v	-
**	Medio	Lontano	v	-
**	Basso	Vicino	v	+
***	Medio	Lontano	f	+
***	Alto	Lontano	v	+
***	Basso	Vicino	f	+
***	Alto	Lontano	v	+
*	Medio	Lontano	f	-
*	Basso	Vicino	f	-
*	Medio	Lontano	v	+
*	Basso	Lontano	v	+
*	Medio	Lontano	v	+

Figure 1.15

Corresponding decision tree:



Rules

If the hotel is three star **then** suggest it.

If the hotel is two star AND close to the city center **then** suggest it.

Figure 1.16

Artificial Neuron

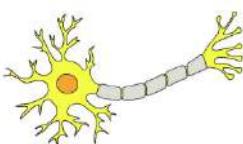


Figure 1.17

Idea: directly simulate brain functioning in a computer, and build an intelligent machine from artificial neurons.

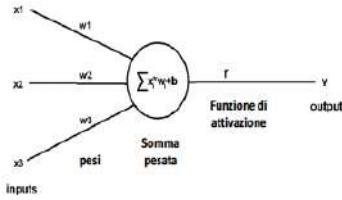


Figure 1.18

First mathematical model of an artificial neuron has been inspired by the biological neurons proposed in 1943 by McCulloch and Pitts.

A neuron receives a set of inputs, makes a weighted sum and then applies an activation function to calculate the output.

The output is controlled by an activation function: each neuron is active only in case its input exceeds a certain threshold.

Two input perceptron

The *perceptron model* is one of the simplest and earliest forms of artificial neural networks, which serves as the building block for more complex neural network architectures. It was developed by Frank Rosenblatt in the late 1950s. The perceptron is primarily used for binary classification tasks, where it decides whether an input belongs to one class or another.

Here's a basic explanation of the perceptron model. Components of a perceptron.

1. *Input Features*: the perceptron takes a set of input features, typically represented as a vector. These features can be numerical values, binary values, or any other data that can be quantified.
2. *Weights*: each input feature is associated with a weight, which represents the importance or contribution of that feature to the perceptron's decision. The weights are parameters that the perceptron learns during training.
3. *Summation Function*: the perceptron calculates the weighted sum of its input features, which is often represented as the dot product of the input features and their corresponding weights.
4. *Bias*: a bias term is added to the weighted sum. The bias allows the perceptron to account for a certain level of error or offset. Like the weights, the bias is also a parameter learned during training.
5. *Activation Function*: the perceptron applies an activation function to the weighted sum plus the bias. The most common activation function used in the classic perceptron model is a step function, often a simple binary step (0 or 1). If the result of the activation function is above a certain threshold, the perceptron outputs one class (*e.g.* '1'); otherwise, it outputs the other class (*e.g.* '0').

Training a perceptron: the primary goal in training a perceptron is to find the appropriate values for the weights and bias that allow it to make accurate

classifications. This is typically done through a supervised learning process where labeled data are used to update the parameters.

The training algorithm for a perceptron is known as the ‘*perceptron learning rule*’. It adjusts the weights and bias based on the errors made during classification. The algorithm works by making incremental updates to the weights and bias until the perceptron can correctly classify all training examples.

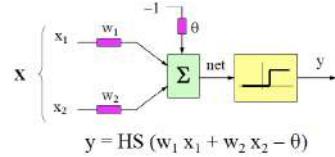


Figure 1.11

The patterns belonging to the class will be those such that:

$$w_1 x_1 + w_2 x_2 - \theta > 0$$

Linear separation of the input space

$$x_2 > -\left(\frac{w_1}{w_2}\right)x_1 + \frac{\theta}{w_2}$$

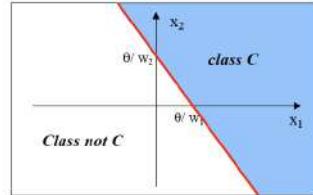


Figure 1.20

Learning an AND gate

$$y = 1 \text{ (output)} \text{ if } x_2 > -\left(\frac{w_1}{w_2}\right)x_1 + \frac{\theta}{w_2}$$

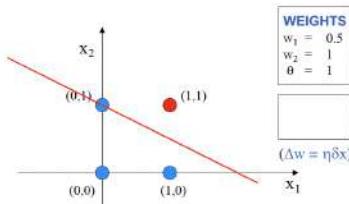


Figure 1.21

Neural Networks

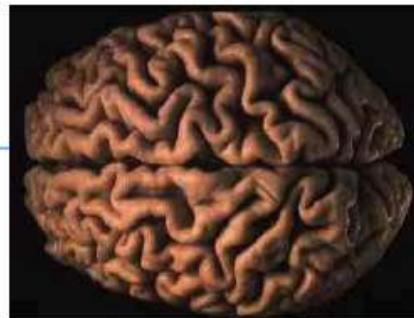


Figure 1.22

To represent more complex concepts networks of neurons should be built: multilayer architectures (forward or feedforward networks) and new learning algorithms.

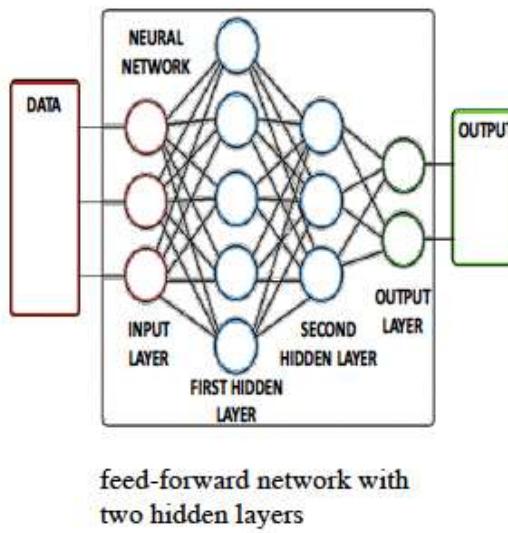


Figure 1.23

Significantly different approach from the symbolic one. The knowledge is not explicit but inherent in the network structure and connection weights.

Three-layer networks

They are able to separate convex regions.²¹

²¹Convex regions, in geometry and mathematics, refer to areas or sets of points that satisfy the properties of convexity. A region is considered convex if, for any two points inside the region, the line segment connecting those points also lies entirely within the region. In other

Number of edges \leq Number hidden neurons.²²

NOTE: we have the input layer, an hidden layer and the output layer (refer to the previous image).

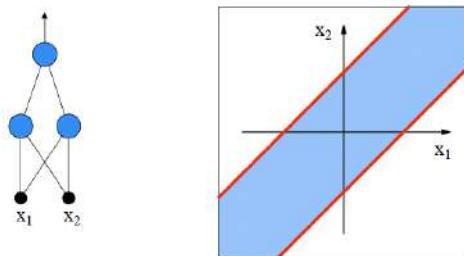


Figure 1.24

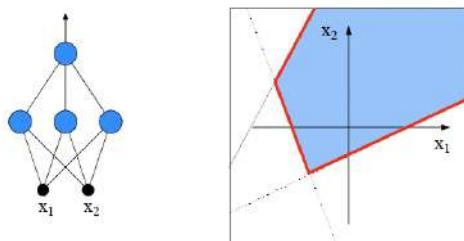


Figure 1.25

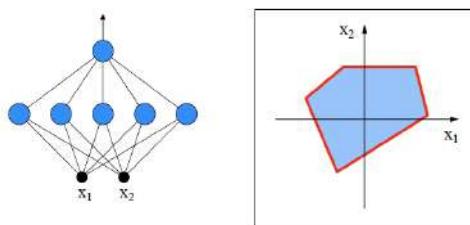


Figure 1.26

Four-layer networks

They are able to separate regions of every shape.

words, a region is convex if it doesn't contain any 'dents' or 'cavities' and has a smooth, bulging shape.

²²In a 2D plane, the edges of a convex region (convex polygon) are line segments that define the boundary of the region. These edges connect the vertices of the polygon. Here are some key characteristics of the edges of a convex region.

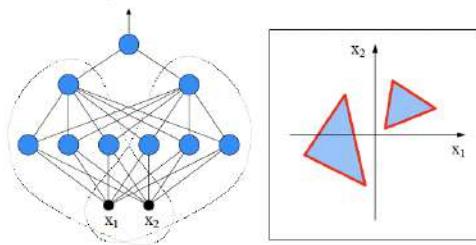


Figure 1.27

Neural networks: power and limitations

Power and limitations.

- The perceptron can represent only linear functions. If, however, we use multi-layer networks, the expressive power increases substantially.
- Universal Approximation Theorem: *A feed-forward network²³ with one hidden layer and a finite number of neurons can approximate any continuous function with desired accuracy.*
- Interesting theory, but it says nothing about how to configure the neural network and how to apply the learning algorithms to get a good approximation. What output functions? How many neurons? How many layers?
- From the 80s more complex and detailed models have been designed for artificial neurons, with different non-linear functions (for example the *sigmoid*).²⁴
- Also various architectures have been identified, and more sophisticated learning algorithms have been developed.

Deep Neural Networks and Deep Learning



Figure 1.28

²³A feed-forward neural network (FFNN), also known as a feed-forward artificial neural network or a multilayer perceptron (MLP), is one of the fundamental architectures in artificial neural networks. It is called ‘feed-forward’ because the data flows through the network in one direction, from the input layer to the output layer, without any feedback loops or cycles.

²⁴The sigmoid function, often referred to as the logistic function, is a mathematical function that maps any input to an output between 0 and 1. It has an S-shaped curve and is defined by the formula: $\sigma(x) = \frac{1}{1+e^{-x}}$

Definition 1.1.9 (Deep Learning).

Deep Learning: models and algorithms that use neural networks with many neurons and many layers. They can learn complex functions, trying to identify most relevant data (rough).

The most commonly used DNNs consist of a very large number of levels (even some hundreds).

Computational expensive training process.

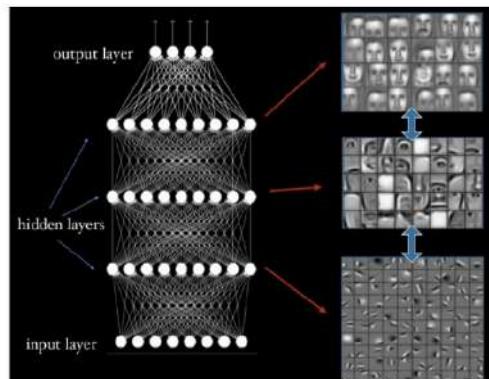
The supervised algorithms currently get good performance with around 5,000 examples for each category and surpass humans with 10 million examples.

It is not a simulation of the brain that has more neurons and a much more complex structure. With a further development of computing power, we will have a comparable number of neurons with human in 2050.



Figure 1.29

From: I. Goodfellow, Y. Bengio, A. Courville: “Deep Learning”, MIT Press, <http://www.deeplearningbook.org>, 2016



https://leonardoaraujosantos.gitbooks.io/artificial-inteligen.../content/deep_learning.html

Figure 1.30

In deep learning concepts are learnt through a hierarchy of features, from the simplest to more abstract and complex. Representation is implicitly distributed in different layers.

Automated Planning

Automated Planning is an important problem solving activity which consists in synthesizing a *sequence of actions* performed by an agent that leads from an

initial state of the world to a given target state (*goal*).

Given:

- an initial state;
- a set of actions you can perform;
- a state to achieve: *goal*.

Find:

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

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

- a representation of the initial state;
- a representation of a goal;
- a formal description of the executable actions (also called operators) in terms of precondition and effects.

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

Example: Block World

Actions:

- STACK (X, Y):
 - IF: holding (X) and clear (Y);
 - THEN: handempty and clear (X) and on (X, Y).²⁵

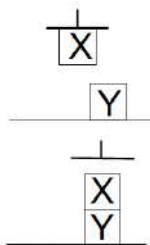


Figure 1.31

- UNSTACK (X, Y):
 - IF: handempty and clear (X) and on (X, Y);
 - THEN: holding (X) and clear (Y).

²⁵Symbolic: IF in the hand there is (X) and on (Y) there is nothing implies THEN the hand is empty and on (X) – that is on (Y), i.e. on (X,Y) – there is nothing.

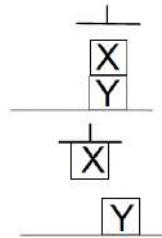


Figure 1.32

- PICKUP (X)
 - IF: ontable (X) and clear (X) and handempty;
 - THEN: holding (X).

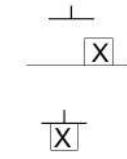


Figure 1.33

- PUTDOWN (X):
 - IF: holding (X);
 - THEN: ontable (X) and clear (X) and handempty.

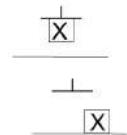


Figure 1.34

Planning Techniques

- Deductive planning / planning as search/ linear planning in *Foundations of AI*.
- Nonlinear planning:
 - Partial Order Planning (POP).
- Hierarchical planning.
- Conditional planning.

- Graph-based planning.
- Planning for robotics paths.
- Planning as emergent behavior: swarm intelligence.
- Reinforcement learning (to learn a policy).

Swarm Intelligence



Figure 1.35

Nature has developed various forms of distributed intelligence:

- our organism, species selections for adapting to environmental changes (*Genetic algos*);
- coordination among animals (ants building huge nests or moving heavy objects without any central coordination) (*Swarm Intelligence*).



Figure 1.36

Who is governing all this? Who is giving instructions, predict future dynamics, produce plans and maintain equilibrium?

- These smart behaviours emerge autonomously with no central coordination nor supervision.
- Development of intelligent systems based on natural metaphores, that are robust and adaptive.
- Swarm intelligence and swarm robotics.

Ants and food search

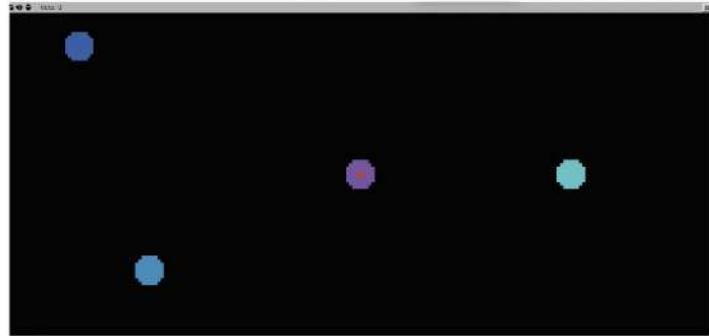


Figure 1.37

Description of how ants search food (refer to the figure above).

1. Violet circle corresponds to the ant nest; light blue circle and light green circle correspond to the food which is equidistant from the nest; blue circle represent the food which is more distant from the nest.
2. Ants begin to occupy space outside the nest in a casual manner.
3. Communicating among them through pheromones, ants go first to light blue and green circles that are equidistant and then go back with the food to the nest.
4. Ants begin to occupy space outside the nest in a casual manner.
5. Communicating among them through pheromones, ants go to blue circle and then go back with the food to the nest.

Genetic and evolutionary algorithms



Figure 1.38

Genetic algorithms (and evolutionary algorithms in general) are inspired by the natural evolution and have been developed in the 70s by John Holland.

- We start from an initial random population of individuals, that creates new individuals through specific laws (*crossover - mutation*).

- *Fitness*: ensures that only the best individuals are kept.
- *Mutation*: new elements introduced randomly.
- *Cross-over*: combines good parent solutions.
- Fitness is not always easy to define and can be asked to the user in some cases.
- These algorithms are useful when a model of the problem is hard to be defined.

Reinforcement learning

Learning policies: how to act given a system state.

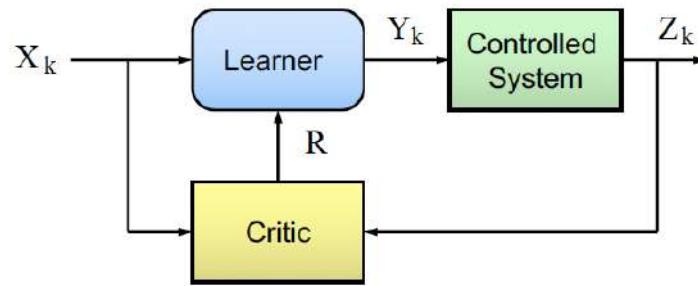


Figure 1.39

Reinforcement: punishment or reward.

1.2 AI applications

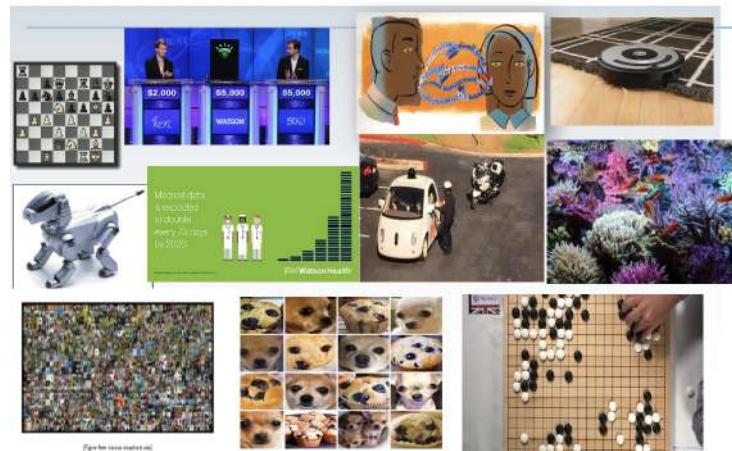


Figure 1.40

Many applications.

- Decision support systems:
 - very specialized on a given domain;
 - knowledge based and control.



Figure 1.41

- Formal systems and games (Chess/GO):
 - limited number of moves and space;
 - based on explicit, non-ambiguous rules.



Figure 1.42

- Natural language – question answering (Watson):
 - ambiguous, implicit, context dependent;
 - based on cognitive states.



Figure 1.43

- Vision (ImageNet):

- object recognition/classification;
- uncertainty and noise (symbolic/sub-symbolic).



Figure 1.44

- Robotics and Autonomous systems (Robot):
 - agents working in an environment;
 - dynamic and real-time decisions.



Figure 1.45

Knowledge based systems

‘The power of an intelligent system primarily derives by the quantity and quality of the knowledge it has on the problem.’ (Feigenbaum)

Various applications: planning, prediction, diagnosis.

Knowledge acquisition: bottleneck of KBS.

Problems:

- the human expert cannot be substituted, but supported in long and repetitive tasks;
- different knowledge sources (often in contrast and incomplete);
- knowledge changes in time;
- knowledge is not always explicit (discovery).

Learning and data-mining techniques.

Decision support systems

Systems that support and do not replace human experts in their decision making process through:

- scenario generation, evaluation, visualizations, predictions.

Various degrees of decision support:

- data analysis for description;
- diagnosis;
- prediction;
- optimization.

Decision automation is one step forward as it excludes the human intervention.

Human intervention in decisions

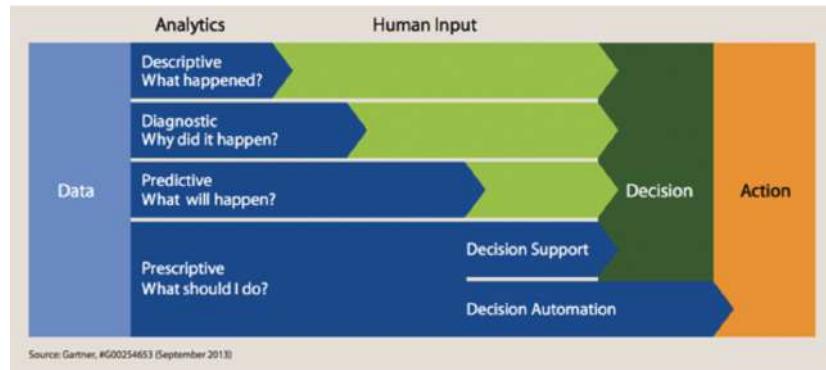


Figure 1.46

Descriptive analytics

Data are used to describe the system:

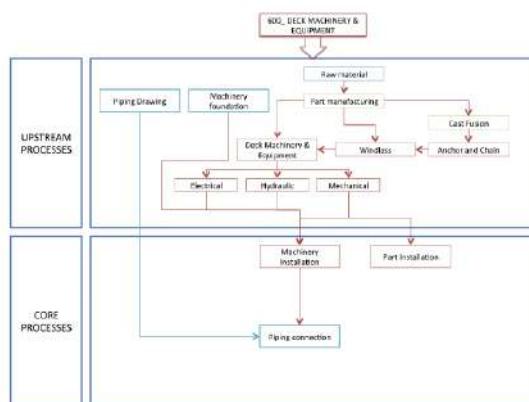
- visualization dashboards;
- business intelligence;
- statistical reports;
- geo-referenced data.

Human intervention is largely needed.

Example of descriptive analytics

Configuration of a productive system/process with environmental constraints.
For each process component we have to characterize data on:

- energy consumption;
- water consumption;
- emissions in the air;
- emissions in the water;
- time of work;
- cost.



Static data: system characterization

Figure 1.47

Energy baseline

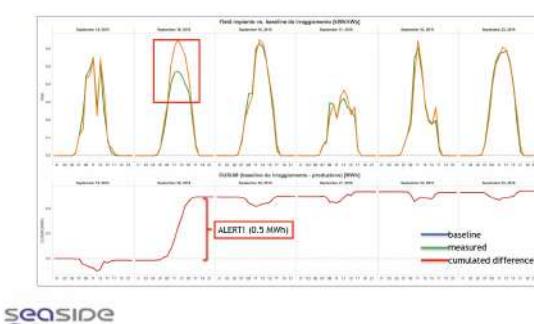


Figure 1.48

Diagnostic analytics

Diagnostic analytics:

- uses data to understand causes;
 - fault diagnosis;
 - root cause analysis;
 - in order to reach decisions human intervention is still largely needed.

Example of diagnostic analytics

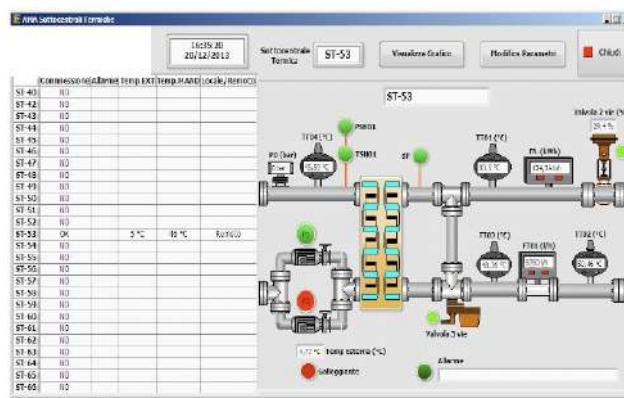


Figure 1.49

System for analyse data from a plant and to realize a diagnostic system. From plant data coming from sensors plus fault data is possible:

- identify temporal series that precede a fault;
 - classify the temporal series before the fault to identify the type of fault;
 - identify the most informative sensors;
 - create models to identify causes.

Predictive analytics

Uses data to predict future system evolutions:

- simulation systems;
 - temporal series prediction;
 - machine learning classification and regression.

To reach decisions we need *what if* analysis.

Example of predictive analytics

System to predict the consumption of a photovoltaic system.

- Input:
 - historical temporal series of past production;
 - meteo forecast;
 - day/month/year.
- Output:
 - future production.

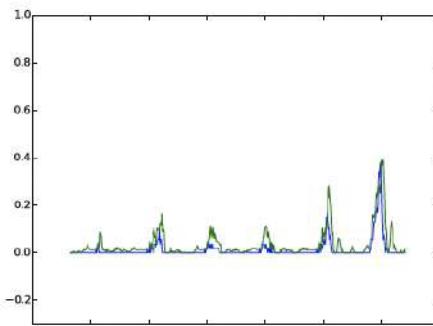


Figure 1.50

System to predict the photovoltaic adoption depending on diverse incentive schema.

- Model:
 - agents divided by type;
 - agents characterised by social and economic aspects;
 - past data (incentives/adoption).
- Output:
 - simulative model.

Decision support – prescriptive analytics

Prescriptive analytics:

- to reach decisions we need to select the preferred scenario:
 - optimization systems (mono and multi objective);
 - combinatorial problem solvers;
 - logic based solvers.

Example of prescriptive analytics

Decision support systems for proposing incentives to foster a given photovoltaic adoption.

- Optimization system that takes as input:
 - simulator model;
 - economic/financial/territorial;
 - objective functions.
- Output:
 - alternative scenarios;
 - impacts.

Human intervention in decisions

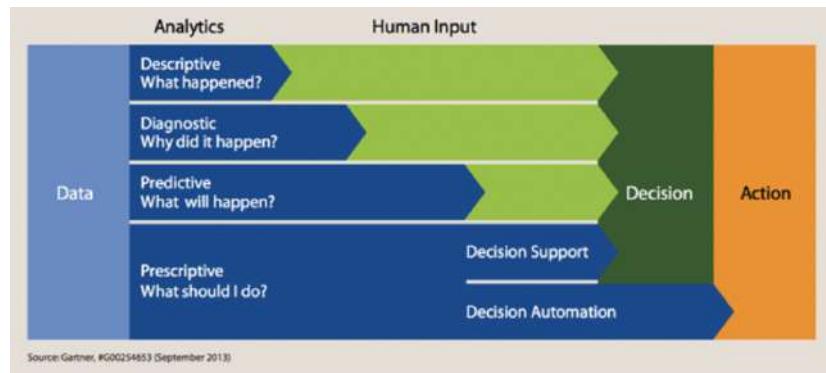


Figure 1.51

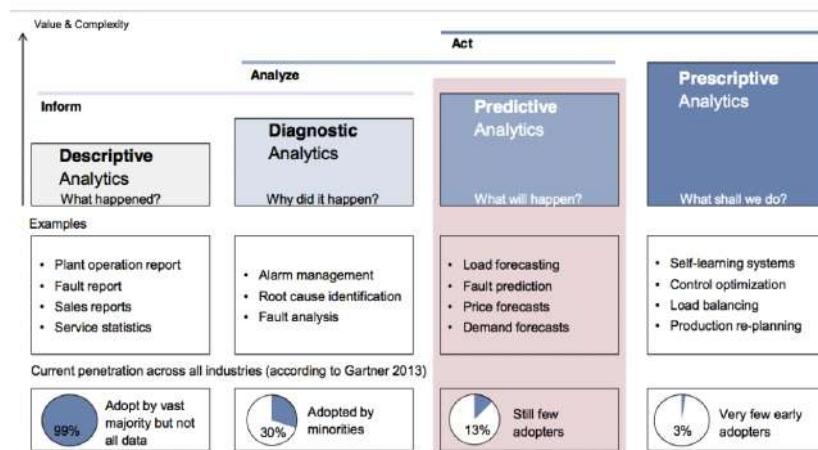


Figure 1.52

Application domains of DSS

Decision support systems apply to a large variety of domains:

- industry (manufacturing, automotive, avionics, food, energy *etc.*);
- public sector (health, mobility, public administration);
- global challenges (climate change, poverty, conflict reduction, environmental health, natural disaster mitigation).

Games: Chess challenge



Figure 1.53

Deep Blue, IBM Risk 2000...

- evaluates 200 millions of moves per second;
- knows 600.000 chess opening.

In 1997 Deep Blue beats Kasparov: is this intelligence?
Chomski about Deep Blue:

“a computer program’s beating a grandmaster at chess is about as interesting as a bulldozer’s ‘winning’ an Olympic weight-lifting competition.”

Algorithm minmax: the brute force

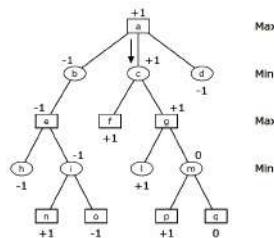


Figure 1.54

We will see in detail this algorithm; in general:

- the *minmax algorithm* is designed to define the optimal strategy for one player (*max*) and for suggesting the best move: for doing that the player has to make the hypothesis that *min* plays at his best;
- huge problem space dimension *e.g.* first move: 400 alternatives, second move: 144.000... combinatorial explosion;
- we need an evaluation function that evaluates the position of each piece and the overall position on the chessboard.

Games AlphaGO

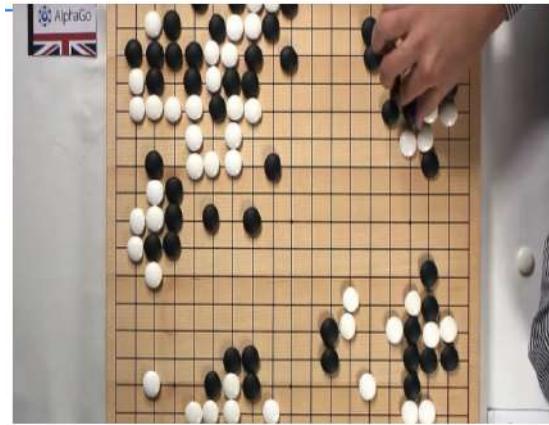


Figure 1.55

AlphaGo has been developed from DeepMind (acquired by Google in 2014). It learned to play videogames (49 games for Atari 2600) better than humans:

- deep learning and symbolic techniques;
- March 2016: AlphaGo beat Lee Sedol, world champion of Go;
- way more complex than chess: 10^{170} possible positions (10^{50} for chess);
- IJCAI Marvin Minsky Medal for Outstanding Achievements in AI in 2017. Michael Wooldridge, chair of the IJCAI Award:

“What particularly impressed IJCAI was that AlphaGo achieves what it does through a brilliant combination of classic AI techniques as well as the state-of-the-art machine learning techniques that DeepMind is so closely associated with. It’s a breathtaking demonstration of contemporary AI...”

DNN Learning but not Deep Reasoning!

After 240 minutes of training, [the system] realizes that digging a tunnel through the wall was the most effective technique.

But the system does not know neither what is a tunnel nor what is a wall. It has learnt contingencies in specific scenarios.

Question answering and NLP Watson (IBM)



Figure 1.56

Jeopardy is one of the most popular quiz in US since 1964.

You have possible answers (clues) and participants have to decide which is the most appropriate question for that answer. *E.g.* clue: The President of the United States in the sixties; proper question: Who is Kennedy?

Watson, the supercomputer developed by IBM, won against Ken Jennings, famous for the record of 74 consecutive victories and Brad Rutter, in Feb 2011: 2 victories and one tie for Watson.

The knowledge in Watson has been built from texts, encyclopedia and the web. Watson is large like 10 fridges, it performs 80 trillion operators per second and reads 200 million pages in 3 seconds.

John Searle:

“Watson doesn’t know it won on ‘Jeopardy’! IBM invented an ingenious program not a computer that can think.”

Noam Chomsky:

“Watson understands nothing. It’s a bigger steamroller. Actually, I work in AI, and a lot of what is done impresses me, but not these devices to sell computers.”

Watson tells us that intelligence is a mixture between algorithms, knowledge and the way this is stored, used, organized:

- almost 1.000.000 million lines of code, 5 years development (20 men);
- memory: 20 TB, 200 million pages (almost 1.000.000 books).

Due tasks: text generation (humans better than computer) and answer generation (computer better than humans).

Element	Number of cores	Time to answer one Jeopardy! question
Single core	1	2 hours
Single IBM Power 750 server	32	<4 min
Single rack (10 servers)	320	<30 seconds
IBM Watson (90 servers)	2 880	<3 seconds

Figure 1.57

Internet and the Semantic Web

“The Web contains everything an intelligent agent should know. Search Engines always allow to retrieve the required information. The Web is a ‘distributed’, ‘emergent’, ‘autonomous’ and ‘complete’ repository of human knowledge.”

Internet is full of information and knowledge and it can be used by humans and computers. It should be structured and made easier to retrieve.

- Semantic web... ‘uses’ and ‘reasons on’ Web data.
- WWW development, memory cost reduction and the presence of unstructured big data, the augmented computing power, has changed the nature of AI applications.
- Storing all human knowledge and storing common sense in a usable way is possible. Can it be achieved now?
- Knowledge should be organized and structured: ontologies.

Simple example

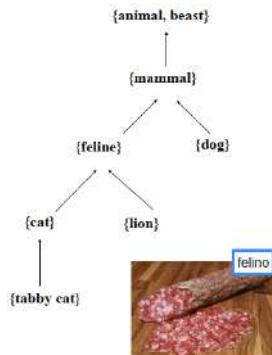


Figure 1.58

Through ontologies, we can add semantics. The cat and lion are both feline so are subclasses of this concept.

Each time I search for «feline» both cats and lions images will be found even if none of them has been tagged as feline.

Ontologies are also used to solve natural language ambiguity. In Italy for example *feline* is a salami type.

WordNet: <https://wordnet.princeton.edu/>

BabelNet <https://babelnet.org/>

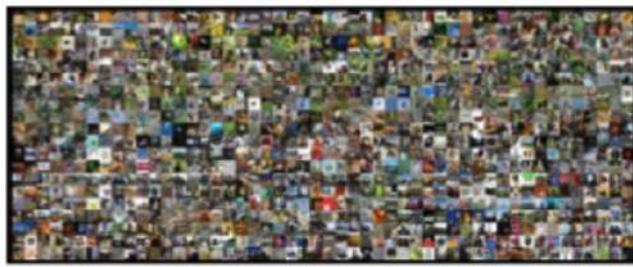
Text-to-speech

Google: *Tacotron 2*, is a text-to-speech system based on neural networks.
<https://google.github.io/tacotron/publications/tacotron2/index.html>

“George Washington was the first President of the United States.”

One of the audio is human while the other is synthetic voice.
They are indistinguishable.

Computer Vision



[Figure from vision.stanford.edu]
Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... & Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. arXiv preprint arXiv:1409.0575.

Figure 1.59

ImageNet Challenge universal image classifier:

- 14 M images;
- 20 k categories;
- tagged via crowdsourcing;
- labelled with the WordNet hierarchy.

In 2011 classification error 26%, today 3% (humans are less efficient: error 5%).



[Figure from Krizhevsky et al., 2012]

Figure 1.60

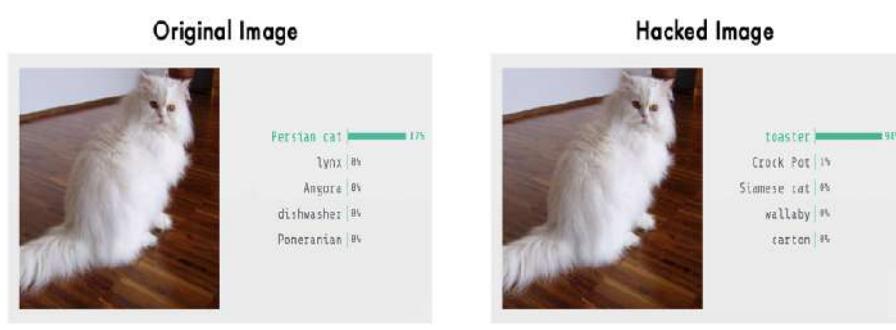
Captioning

Describe the content of an image (from Google).



Figure 1.61

DNN: sometimes they make big mistakes!



"Only modification of some pixels not evident to the human eye".
<https://medium.com/@ageitgey/machine-learning-is-fun-part-8-how-to-intentionally-trick-neural-networks-b55da32b7196>

Figure 1.62

DNN²⁶ can be unstable: by applying some perturbations to inputs we can arbitrarily change the output.

Only modification of some pixels – which were not present during the training – not evident to the human eye can lead to big mistakes.

If we look at the right part of the figure above we can see very big mistakes; imagine what could happen if this kind of errors would happen on a board computer of a car with autonomous drive.

²⁶Deep Neural Network.

Can we trust DNN

DNN are extremely effective in perception tasks but...

“Are data and computational power eager and work in a known and stable world.”

We could have problems if we work on new data that are far from the training data (overfitting).

Data might be biased (more on this later).

“Hardly explainable.”

Millions of parameters that are hard to understand not only by users but also by developers. Not easy to explain the reason of a decision.

“Do not distinguish between causality and correlation.”

DNN are statistical techniques that can discover complex and non linear correlations between input and output but these are not necessarily cause-effect relations. Just curve-fitting.

“Not trustworthy.”

They can make big mistakes that are ‘different’ from mistakes made by humans.
Source: ‘The book of why: The New Science of Cause and Effect’; Judea Pearl, Dana Mackenzie, 15 May 2018.

Source: Gary Marcus, ‘Deep Learning: A Critical Appraisal’. CoRR abs/1801.00631 (2018).

Source: ‘Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models’, W. Samek, T. Wiegand, K. Muller (August 2017).

Far from causality

Funny example: <http://tylervigen.com/spurious-correlations>

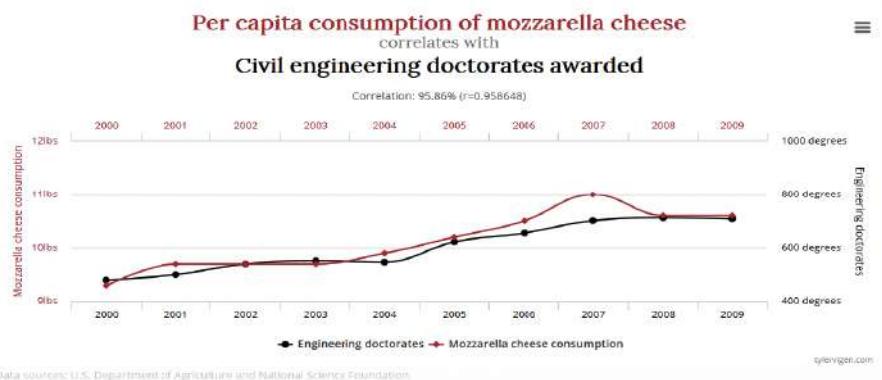


Figure 1.63

Mind needs a body

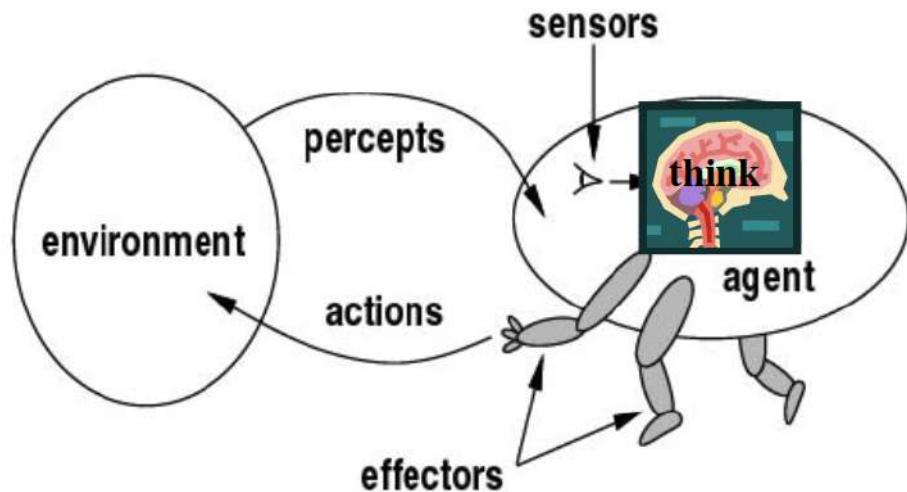


Figure 1.64

Atlas doing Parkour (Boston Dynamics)

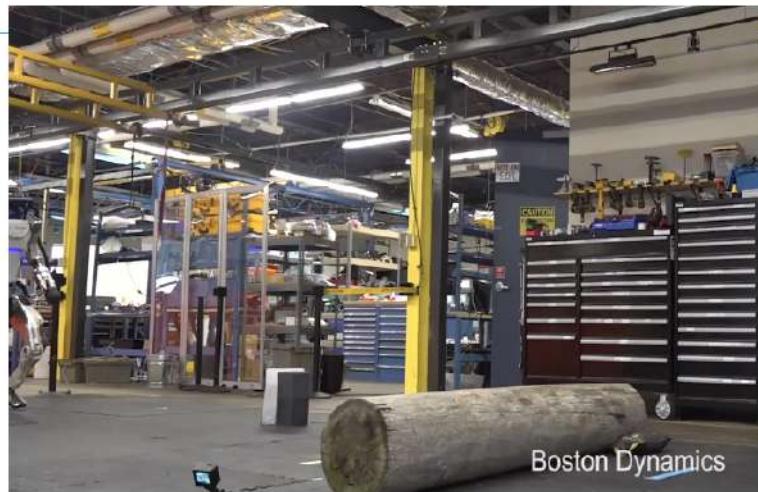


Figure 1.65

Robocup

Contest started in Japan in 1997 with the goal of creating a soccer team within 2050 able to play against (and possibly beat) the world champion team.

Too ambitious? Deep Blue has been created 50 years after the first computer was invented, Moon landing (1969) 50 years after the first airplane.

New challenge very different from games: immersed in an environment.

Autonomous Robots, with real-time sensing, reaction, communication, vision, perception, movement, coordination, planning, learning capabilities.

Full Turing Test.

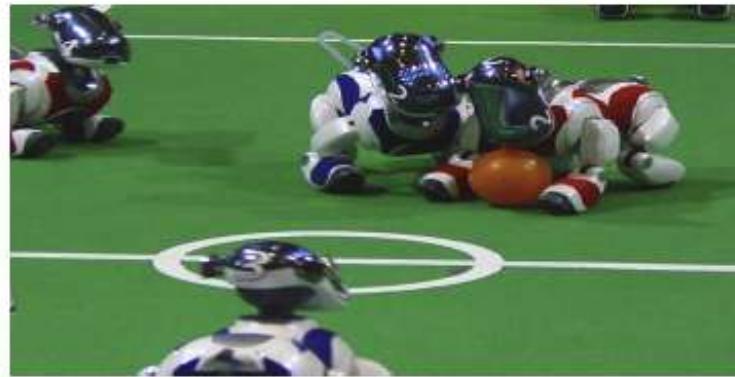


Figure 1.66

Nao: structure

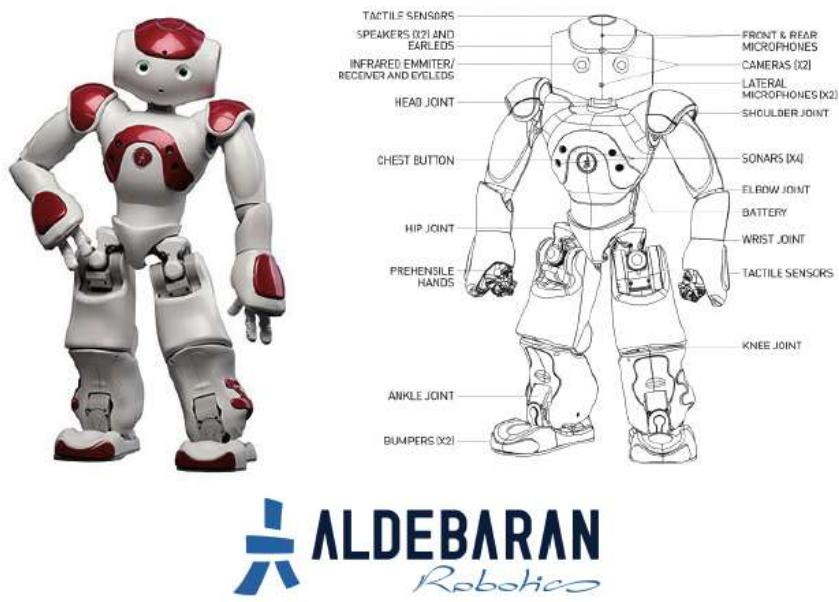


Figure 1.67

Nao and Robocup

Figure 1.68

By Nao-Team HTWK, University of Lipsia.

Nao

Nao robot model, if properly trained, is able also to:

- reads a text (*e.g.* in italian);
- plays with natural language;
- solves math expressions.

1.3 Trustworthy AI

Should Artificial Intelligence be regulated?

“... In recent decades, however, a consensus has emerged around the idea of a rational agent that perceives and acts in order to maximally achieves its objectives... Up to now, AI has focused on systems that are better at making decisions; but this is not the same as making better decisions... well aligned with human values.”

AI should be beneficial with applications related to health, climate change, energy, smart cities, food, equity, inclusion and sustainability at large, but it can also be applied to dangerous applications like the ones on autonomous weapons. Also, AI will bring substantial societal impacts: job losses, fake news generation, election control through social influence, personal data privacy. EU has delivered guidelines for trustworthy AI.

Source: Stuart Russell, “Provably Beneficial Artificial Intelligence”, 2017

<https://people.eecs.berkeley.edu/russell/papers/russell-bbvabook17-pbai.pdf>

Source: Amitai Etzioni, and Oren Etzioni. ‘Should Artificial Intelligence Be Regulated?’ *Issues in Science and Technology* 33, no. 4 (Summer 2017).

Ethics guidelines for trustworthy AI

Delivered by the European Commission’s High-Level Expert Group on Artificial Intelligence (AI HLEG).

- Trustworthy AI has two components:
 1. it should respect fundamental rights, applicable regulation and core principles and values, ensuring an ‘ethical purpose’;
 2. it should be technically robust and reliable since, even with good intentions, a lack of technological mastery can cause unintentional harm.
- Incorporate the requirements for Trustworthy AI from the earliest design phase: Accountability, Data Governance, Design for all, Governance of AI Autonomy (Human oversight), Non-Discrimination, Respect for Human Autonomy, Respect for Privacy, Robustness, Safety, Transparency.
- Foresee training and education, and ensure that managers, developers, users and employers are aware of and are trained in Trustworthy AI.

Source: <https://ec.europa.eu/digital-single-market/en/news/draft-ethics-guidelines-trustworthy-ai>

Which properties of AI?

Some properties.

- *Fairness*: decisions should not be discriminatory. We should be sure for instance that race or gender are not influencing decisions. But data are biased. Amazon automatic curricula selector was giving preferences to male candidates as the data set was biased (experiment closed in 2017). Microsoft chatbot Tay learning from Twitts to behave as a nazist.
- *Transparency*: a system behaviour should be understandable under every circumstances and based on a comprehensible model.
- *Verifiability*: formally prove that the system is correct with respect to some property.
- *Explainability*: being able to explain the decision taken and the factors that have determined it.
- *Accountability*: responsibility for the decision taken.
- *Accuracy*.
- *Privacy*.

These features are not always all needed. For instance I would like to understand why I am not eligible for a loan, maybe I am not interested in understanding why the hoover made a given path.

Source: When Computers Decide: European Recommendations on Machine-Learned Automated Decision Making ACM, 'Statement on Algorithmic Transparency and Accountability,' 12 January 2017.

Private traits are predictable from digital records of human behavior

Easily accessible digital records of behavior, Facebook Likes, can be used to automatically and accurately predict a range of highly sensitive personal attributes: sexual orientation, ethnicity, religious and political views, personality traits, intelligence, happiness, use of addictive substances, parental separation, age, and gender.

- A dataset of 58,000 volunteers have made available their Facebook Likes and detailed personal data, profiles etc. for machine learning.
- The learnt model is enough accurate and discriminates among different categories (homosexual and heterosexuality with 88% accuracy, Democrats and Republicans with 85% accuracy).
- Implications on privacy (Cambridge Analytica).

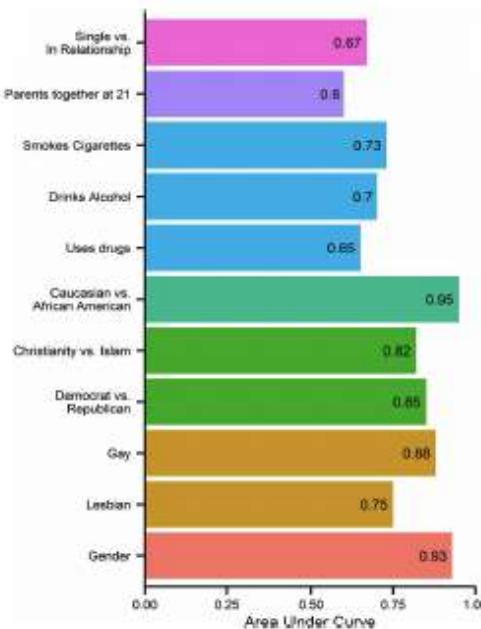


Figure 1.69

Source: When Computers Decide: Michal Kosinski, David Stillwell and Thore Graepel. 'Private traits and attributes are predictable from digital records of human behavior'. PNAS April 9, 2013. 110 (15) 5802-5805.

Explainable AI is an open challenge

It is not easy to make a sub-symbolic system explainable. The AI community is actively working on these themes.

Not yet general ideas, but good promising results on specific domains.

Fairness, Interpretability, Explainability ECAI-IJCAI Workshops 2018.

Toward an integration of the two souls of AI for combining the advantages of both approaches in hybrid architectures.

Integrate *deep learning*, which is excellent for perception and machine learning (but is a black box) with *symbolic systems* that are *transparent* and are able to perform reasoning and abstraction.

Source: ‘Learning Explanatory Rules from Noisy Data’, Richard Evans, Edward Grefenstette DeepMind, London, UK Journal of Artificial Intelligence Research (2018).

Source: ‘Neural-Symbolic Learning and Reasoning: Contributions and Challenges’ Artur d’Avila Garcez et alii., The 2015 AAAI Spring Symp., 2015.

Source: L.G. Valiant, ‘Knowledge Infusion: In Pursuit of Robustness in Artificial Intelligence’. In FSTTCS 2008.

Source: Probabilistic Inductive Logic Programming Editors: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. LNCS 4911 2008.

Human-centered Artificial Intelligence



Figure 1.70

Ethical, Legal and Social aspects



Figure 1.71

Toward a beneficial AI

“Up to now, AI has focused on systems that *are better at making decisions; but this is not the same as making better decisions.* No matter how excellently an algorithm maximizes, and no matter how accurate its model of the world, a machine’s decisions may be ineffably stupid, in the eyes of an ordinary human, *if its utility function is not well aligned with human values ... This problem requires a change in the definition of AI itself, from a field concerned with pure intelligence, independent of the objective, to a field concerned with systems that are provably beneficial for humans.*”

Source: ‘Provably Beneficial Artificial Intelligence’, Stuart Russell 2017.

Beneficial AI

“... a change in the definition of AI itself, from a field concerned with pure intelligence, independent of the objective, to a field concerned with systems that are provably beneficial for humans.”

Source: ‘Provably Beneficial Artificial Intelligence’, Stuart Russell 2017.

AI can have a beneficial impact on society, economy and environment, the three pillars of sustainable development.

Computational Sustainability (NSF expedition).

Mixture of computational techniques for dealing with big societal challenges.

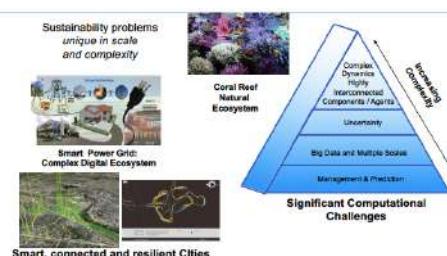


Figure 1.72

Flood management

Integration of descriptive, predictive, prescriptive models.

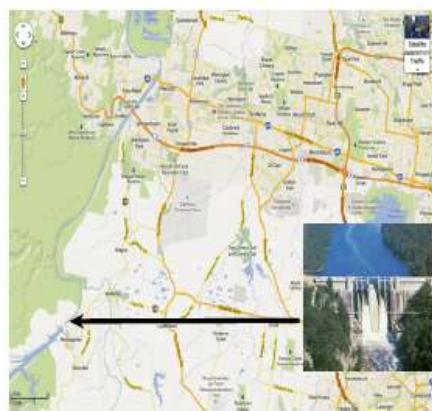


Figure 1.73

- People involved:
 - 70.000 persons.
- Evacuation profile:
 - 50 residential zones (evacuation nodes);
 - 10 evacuation centers (safe nodes);
 - 125 transit nodes (intersections);
 - 458 edges (road segments).
- Flooding scenarios:
 - past data on previous floods.

Descriptive model: road network and infrastructure



Figure 1.74

- Road network and infrastructure.
- Max flow for each road.
- Population density.
- Past data.

Flood Simulation: predictive model

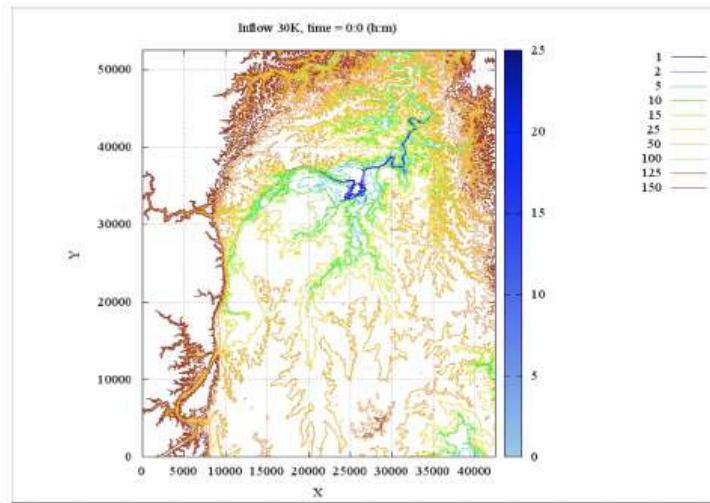


Figure 1.75

Evacuation planning: prescriptive model

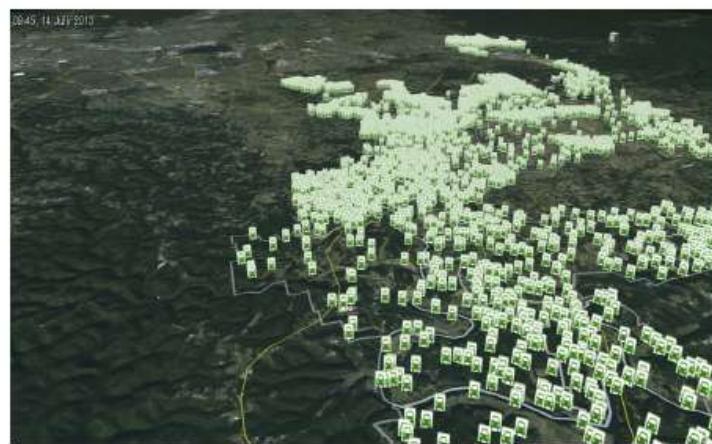


Figure 1.76

Human factor

Plans should model the human aspects.

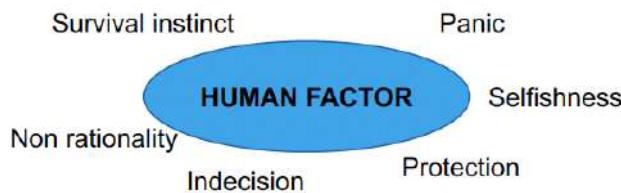


Figure 1.77

Ethical factor

Plans should model the ethical aspects.

- Are ethical values universal?
- Are ethical principles easily and clearly defined?

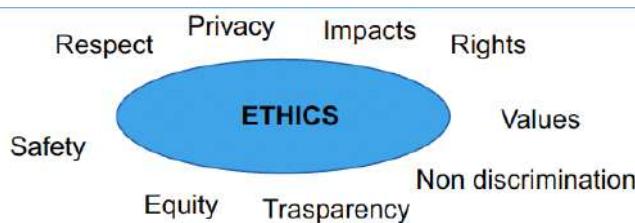


Figure 1.78

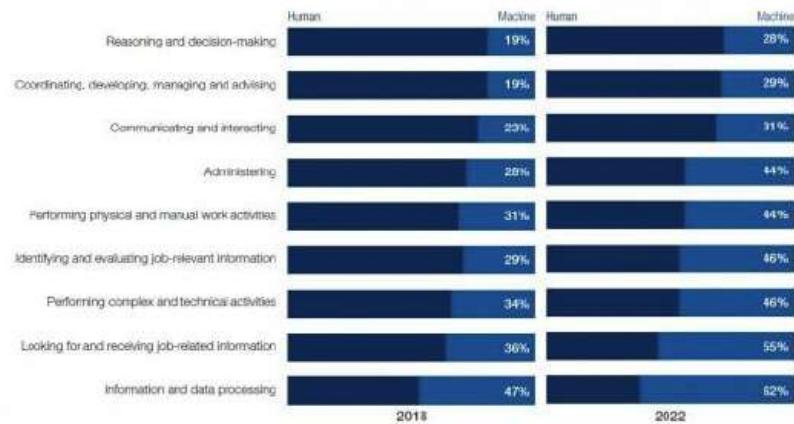
Application domains for AI

From 'Artificial intelligence and life in 2030', Stanford University – Sept 2016 – AI eight domains with high impact:

- transports (intelligent cars, self driving cars, transport planning, on-demand transport);
- domotics (companion robots);
- health (clinic support, health data analisys, health robotics, elderly care);
- education (tutoring system and on-line learning);
- inclusion of poor classes;
- safety and security;
- job market;
- entertainment (social platforms, games, arts and creativity).

Ratio of human-machine working hours 2018 vs 2022

Figure 5: Ratio of human-machine working hours, 2018 vs. 2022 (projected)



Source: Future of Jobs Survey 2018, World Economic Forum.

Figure 1.79

EU landscape

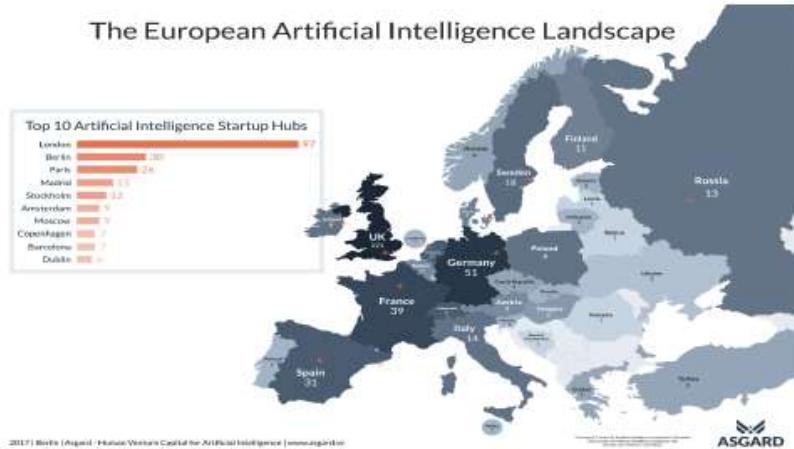


Figure 1.80

Where AI

Some references.

- European Association for Artificial Intelligence (EurAI), (previously EC-CAI). Founded in 1982, is a scientific umbrella association organizing ECAI, European Conference on Artificial Intelligence.

- *Association for the Advancement of Artificial Intelligence (AAAI)* (previously *American Association for Artificial Intelligence*). Founded in 1979, it is a scientific association organizing the *AAAI* conference on Artificial Intelligence.
- *Associazione Italiana per l'Intelligenza Artificiale (AI*IA)*. Founded in 1988, is a scientific italian association and organizing the *Conferenza Italiana di IA*.
- The largest AI conference in the world is the *International Joint Conference on Artificial Intelligence (IJCAI)*. It will be organized in Bologna in 2022 together with *ECAI*.
- *Artificial Intelligence Journal*, major scientific journal of the field.
- *The Journal of Artificial Intelligence Researchers*, other important scientific journal.

Chapter 2

Search strategies

2.1 Non-informed Search strategies

Searching for solutions

Many AI problems can be solved by *exploring the so called solution space*. It contains all possible sequences of actions that might be applied by an agent. Some of these sequences lead to a solution (*if it exists*):

- the agent examines alternative sequences of actions that lead to known states and chooses, then, the best one;
- the process of trying this sequence is called **SEARCH**;
- it is useful to think about the search process like building a *search tree* whose nodes are states and whose branches are operators/actions.

A search algorithm takes as input a problem and returns a solution in the form of a sequence of actions.

Once the solution is found, the suggested actions can be performed:

- this is called **EXECUTION**.

Generate sequences of actions.

- **Expansion:** one starts from a state, and applying the operators (or successor functions) will generate new states.
- **Search strategy:** at each step, choose which state to expand.
- **Search tree:** it represents the expansion of all states starting from the initial state (the root of the tree).
- The leaves of the tree represent either states to expand or solutions or dead-ends.

Search Trees

Basic idea: off-line, simulated exploration obtained by expanding states that have been already explored.

```

function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree

```

Figure 2.1

Example

A holiday in Romania: currently we are in Arad.

Flights landing in Bucharest tomorrow.

Goal:

- to be in Bucharest.

Problem:

- states: being in a city;
- actions: travel between two cities.

Solution:

- sequence locations: *e.g.* Arad, Sibiu, Fagaras, Bucharest.

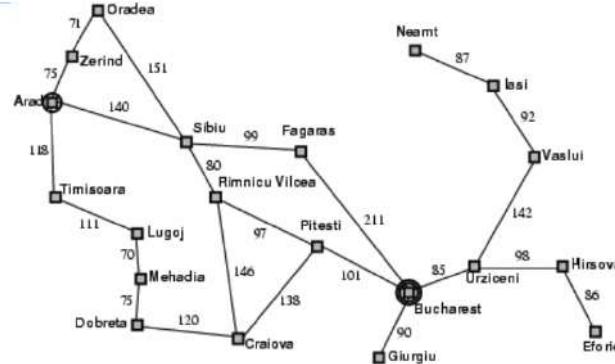


Figure 2.2

Note in the figure above (look at Arad on the left) that we have three initial operators: we can move only to three cities at the beginning.

Problem Formulation

The *problem* shown in the previous example is defined by five points.

1. Initial state: e.g. ‘at Arad’;
2. Successor functions: $S(x) = \{ \text{set of action-state pairs} \}$;
 - e.g. $S(\text{Arad}) = \{ (\text{Arad} \rightarrow \text{Zerind}, \dots) \}$.
3. Test goal, could be:
 - explicit: e.g. $x = \text{‘At Bucharest’}$;
 - implicit: e.g. $\text{checkmap}(x)$.
4. Cost of the path: e.g. distance travelled, the number of actions (hops) performed etc. $C(x, a, y) \geq 0$.
5. Solution: a *solution* is a sequence of actions that lead from the initial state to the goal.

Example

Let’s return to the concrete example. Consider the following.

Definition 2.1.1 (Fringe).

We define Fringe the set of unexpanded node in any part of the tree.

If we look at the Figure 2.2, at the initial state, the only element in the Fringe is Arad: $\text{Arad} \in \text{Fringe} = \{ \dots \}$, as shown in the figure below.

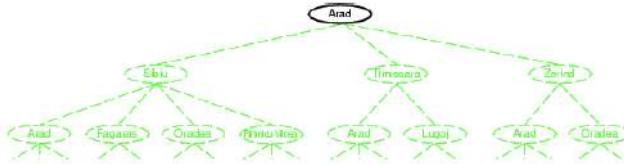


Figure 2.3

After a first expansion we have (look at the image below):

- $\text{Arad} \notin \text{Fringe}$;
- $\{ \text{Sibiu}, \text{Timisoara}, \text{Zerind} \} \subseteq \text{Fringe}$.

Why ‘Arad’ doesn’t belong to Fringe? Can we come back? It depends on how we define the operator.

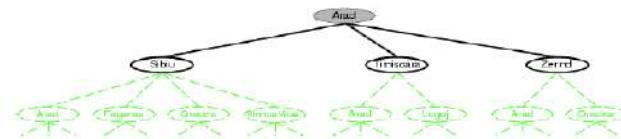


Figure 2.4

After another expansion (look at the image below):

- $\{ \text{Arad}, \text{Sibiu} \} \not\subseteq \text{Fringe};$
- $\{ \text{Arad}, \text{Fagaras}, \text{Oradea}, \text{Rimnicu Vilcea}, \text{Timisoara}, \text{Zerind} \} \subseteq \text{Fringe}.$

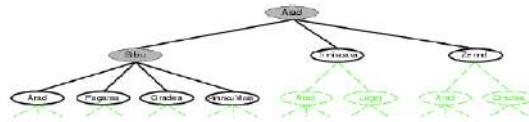


Figure 2.5

Node

Each node in the search tree corresponds to a data structure containing:

- the state;
- the parent node;
- the operator which has been applied to obtain the node;
- the depth of the node;
- the cost of the path from the initial state to the node.

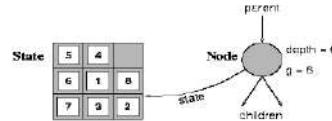


Figure 2.6

Implementation

```

function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST[problem][STATE[node]] then return SOLUTION(node)
    fringe  $\leftarrow$  INSERT-ALL(EXPAND(node, problem), fringe)
  _____
  function EXPAND(node, problem) returns a set of nodes
    successors  $\leftarrow$  the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
      s  $\leftarrow$  a new NODE
      PARENT-NODE[s]  $\leftarrow$  node; ACTION[s]  $\leftarrow$  action; STATE[s]  $\leftarrow$  result
      PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
      add s to successors
    return successors
  
```

Figure 2.7

Search Strategy: effectiveness

Some questions.

- Can we find a solution?
 - If we structure the problem bad, there is the risk to enter into a loop. Furthermore we should consider that could be different solutions and some of them are better than the others: for example some can be achieved with a cost lower than the cost of the others.
- Is it a good solution?
 - Minimum path solution: on-line cost.
- What is the cost of the search?
 - Time to find a solution: off-line cost.
- Total cost = search path cost + cost of the search.
- Choosing states and actions → Importance of abstraction.

Example: the game of 8

We have a three by three board with eight tiles and one empty space (blank). We have to find the moves that lead to a given position as goal.

- States: position of each of the tiles.
- Operators: the blank moves to the right, left, up and down.
- Test target: description of the final state.
- Walk cost: each move costs 1.

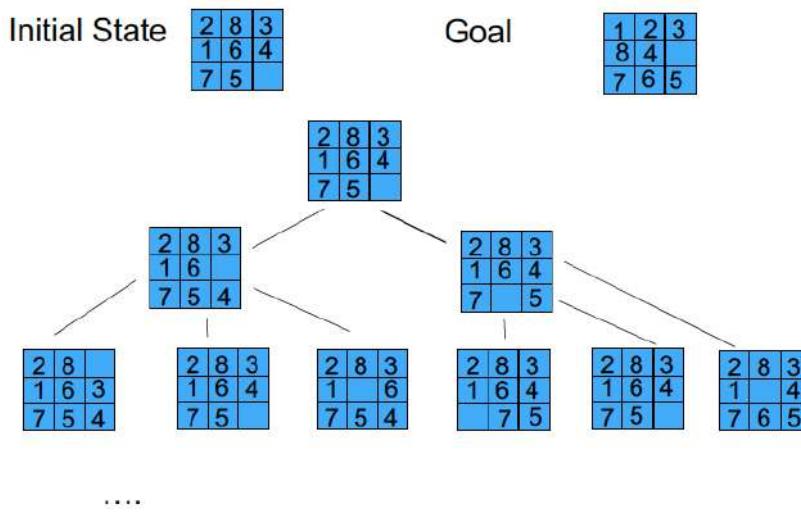


Figure 2.8

Note:

- a good way to describe a *state* is through a *matrix*;
- a good way to describe the *operator* that performs the passages is to consider only how is possible to move the white box without considering the boxes with numbers; this last monitoring would be more complicated.

Search strategies

The choice on how to expand a search tree is called **strategy**.

At each node more actions may be performed (actions: operators of applications). Strategy, two possibilities.

- **Non informed strategies:** do not use any domain knowledge; apply rules arbitrarily and do an exhaustive search strategy.
 - In practical for some complex problems.
- **Informed strategies:** use domain knowledge: apply rules following *heuristics*.¹
 - If we had a perfect heuristics you do not need search.

The strategies are evaluated according to four criteria.

- **Completeness:** does the strategy guarantees to find a solution if one exists?
- **Time complexity:** how long does it take to find a solution?
- **Space complexity:** how much memory is needed to carry out the search?
- **Optimality:** does the strategy find the best solution when there are more solutions?

Search strategies: Non informed strategies

Non informed strategies techniques:²

- breadth-first (at a uniform cost);
- depth-first;
- depth-first limited depth;
- iterative deepening.

¹For example, travelling among Romanian cities, a heuristic would be taking the decision to go either in a city or to another choosing the shortest path; this doesn't implicate automatically that we arrive to a solution or that the solution we find is the best; maybe there are other factors to consider.

²The words 'Non informed' mean that we don't use any knowledge of the problem.

The general search strategy

How can we define a general search strategy?

What logic could implement it?

There are several description we can give but the most important thing is to capture its essence.

A general search strategy could be schematize roughly with the following pieces of ideal code, just to give an overview on the logic behind.

The '*Queuing-Fn*' function mentioned below is a function passed to append the nodes obtained by the expansion.

```
function GENERAL-SEARCH(problem,strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Figure 2.9

```
function GENERAL-SEARCH(problem,QUEUING-FN) returns a solution, or failure
    nodes  $\leftarrow$  MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
    loop do
        if nodes is empty then return failure
        node  $\leftarrow$  REMOVE-FRONT(nodes)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
        nodes  $\leftarrow$  QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end
```

Figure 2.10

Breadth-first search

Definition of Depth:

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

Breadth-first search always *expands less deep* tree nodes.

In the worst case, if depth is d and the branching factor is b then the maximum number of nodes expanded in the worst case will be b^d . Time and space complexity (many paths stored in memory).

$$1 + b + b^2 + b^3 + \dots + b^{d-1} \longrightarrow b^d \quad (2.1.1)$$

This strategy ensures **Completeness**, but we don't have *efficient implementation* on single-processor systems (multiprocessor architectures).³

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	1 seconds	11 kilobytes
4	11111	11 seconds	1 megabyte
5	10^6	18 minutes	11 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11111 terabytes

Figure 2.11

The main disadvantage is the excessive memory footprint. The example assumes that the branching factor is $b = 10$. It expands 1000 nodes/second. Each node uses 100 bytes of memory.

Some considerations:

- the problem of memory seems to be the most serious;
- *breadth first search* always finds the min-cost path if the cost equals the depth (otherwise we should use another strategy that always expands the least cost node \longrightarrow *uniform cost strategy*);
- the *uniform cost strategy* is comprehensive and, unlike the *breadth first search*, ideal when operators have not uniform costs (same temporal and spatial complexity as *breadth first search*).

Breadth-first search: an implementation

Breadth-first expands the nodes at lower depths.
Implementation.

- Fringe is a FIFO queue, *i.e.* successors at bottom.

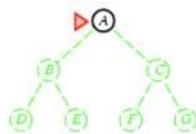


Figure 2.12

³It would be convenient to use this strategy only if our problem is not too big; we need an exponential memory.

- At equal depth we go to the left by convention.

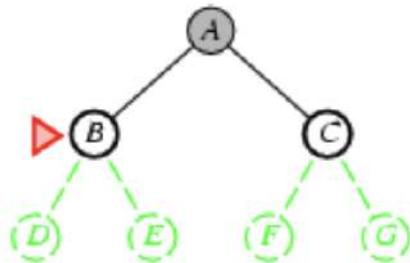


Figure 2.13

- At this point we have:
 - D, E → depth: 2;
 - C → depth: 1; it has the lower depth.

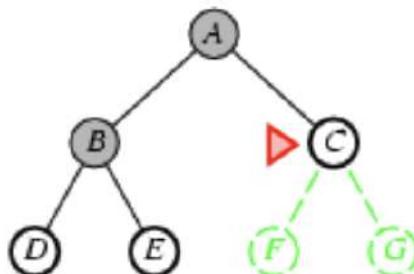


Figure 2.14

- Finally:

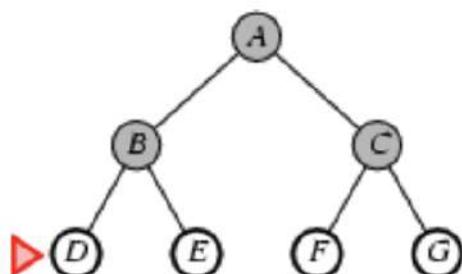


Figure 2.15

- QueueingFn = Add the successors to the end of the queue.

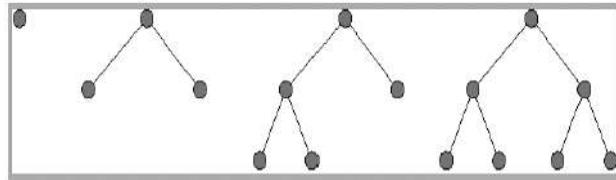


Figure 2.16

Properties of breadth-first search

Main properties.

- Complete? Yes, (if b is finite).
- Time? $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$ i.e. exponential in d .
- Space? $O(b^d)$ (keeps every node in memory).
- Optimal? Yes (if cost = 1 per step); not optimal in general.

Space is the big problem: can easily generate nodes at 1 MB/sec so 24 hrs = 86 GB.

- b - maximum of the search tree branching.
- d - depth of the least-cost solution.
- m - maximum depth of the state space (may be infinite).

Uniform-cost search

Each node is labelled with a cost $g(n)$.

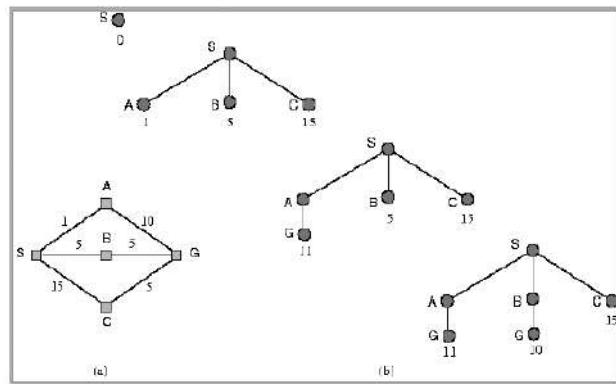


Figure 2.17

QueueingFn = Enter the successors in order of increasing path cost.

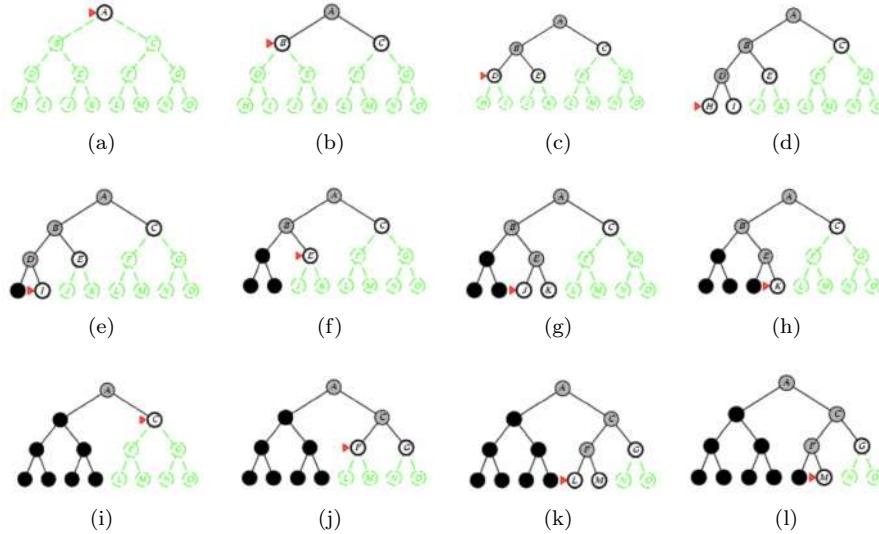


Figure 2.18

Depth first search

Characteristics:

- depth-first expands deepest nodes first;
- nodes at equal depth are *arbitrarily* selected (leftmost);
- depth-first search requires a modest memory occupation;
- for a state space with branching factor b and maximum search depth d we have to store $b \cdot d$ nodes;
- the temporal complexity is rather similar to that of breadth-first search;
- in the worst case, if the maximum depth is d and the branching factor is b the maximum number of nodes expanded is b^d (time complexity);
- depth-first search is *efficient* from an implementation point of view: one path at a time is stored (a single stack);
- depth-first search can be *non-complete* with possible loops in the presence of infinite branches.

Depth first search: an implementation

Expands the deeper nodes first.

Implementation (see Figure 2.18).

- Fringe = LIFO stack.

QueueingFn = Enter the successors to the top of the stack. It is assumed that the nodes of depth 3 does not have successors.

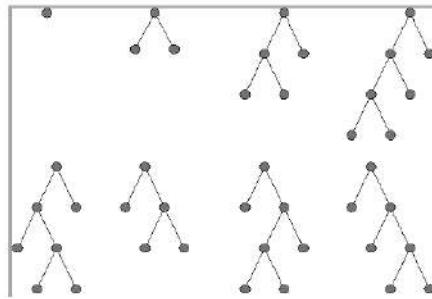


Figure 2.19

Properties of depth-first search

Main properties.

- Complete? No: fails in infinite-depth spaces, spaces with loops; modify to avoid repeated states along path \Rightarrow complete in finite spaces.
- Time? (b^m) i.e. terrible if m is much larger than d , but if solutions are dense, may be much faster than breadth-first.
- Space? $O(b \cdot m)$ i.e. linear space!
- Optimal? No.

Where:

- b - maximum of the search tree branching;
- d - depth of the least-cost solution;
- m - maximum depth of the state space (may be infinite).

Example: depth-first

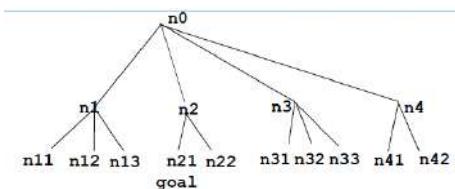


Figure 2.20

Depth-first: nodes expanded added at the head of L (list).

- $n0$
- $n1, n2, n3, n4$
- $n11, n12, n13, n2, n3, n4^4$

⁴At this point we're expanding $n11$.

- $n_{12}, n_{13}, n_2, n_3, n_4$ ⁵
- n_{13}, n_2, n_3, n_4
- n_2, n_3, n_4
- n_{21}, n_{22}, n_3, n_4
- Success (expanding n_{21})

Note that data stored are put at the end.

Example: breadth-first

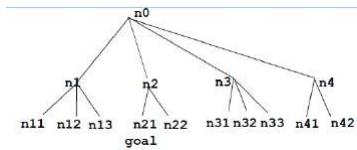


Figure 2.21

Breadth-first: nodes expanded appended to L.

- n_0
- n_1, n_2, n_3, n_4
- $n_2, n_3, n_4, n_{11}, n_{12}, n_{13}$
- $n_3, n_4, n_{11}, n_{12}, n_{13}, n_{21}, n_{22}$
- $n_4, n_{11}, n_{12}, n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}$
- $n_{11}, n_{12}, n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$
- $n_{12}, n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$
- $n_{13}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$
- $n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{41}, n_{42}$ (expanding n_{21})
- Success

Note the difference of memory with the previous.

Limited Depth search

Limited Depth search:

- it is a depth-first variant;
- it includes a *maximum depth* parameter;

⁵After expanded n_{11} we expand n_{12} .

- when you reach the maximum depth or a failure, it explores alternative paths (if they exist), then alternative paths at less than one unit of depth, and so forth (*backtracking*);
- you may establish a maximum limit of depth (it does not necessarily solve the problem of completeness);
- avoids infinite branches.

The nodes at depths L have no successors.

Recursive Implementation.

```

function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred?  $\leftarrow$  false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred?  $\leftarrow$  true
    else if result  $\neq$  failure then return result
  if cutoff-occurred? then return cutoff else return failure

```

Figure 2.22

Iterative Deepening

Iterative Deepening.

- Iterative deepening search avoids the problem of choosing the maximum depth limit by trying all possible depth limits.
 - Start with 0, then 1, then 2...
- It combines the advantages of depth and breadth-first strategies. It is complete and explores a single branch at a time.
- Many states are expanded multiple times, but this does not worsen considerably the execution time.
- In particular, the total number of expansions is:

$$(d + 1) \cdot 1 + d \cdot b + (d - 1) \cdot b^2 + \dots + 3 \cdot b^{d-2} + 2 \cdot b^{d-1} + b^d.$$
- In general it is the favourite search strategy when the search space is very large.
- It can emulate the breadth-first search through repeated applications of the depth first search with an increasing depth limit.
 1. $C = 1$.
 2. Apply depth-first with depth limit of C , if you find a solution stop.
 3. Otherwise, increase C and go to step 2.

Iterative Deepening

```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem
  for depth  $\leftarrow 0$  to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

```

Figure 2.23

Iterative Deepening search L = 0



Figure 2.24

Iterative Deepening search L = 1



Figure 2.25

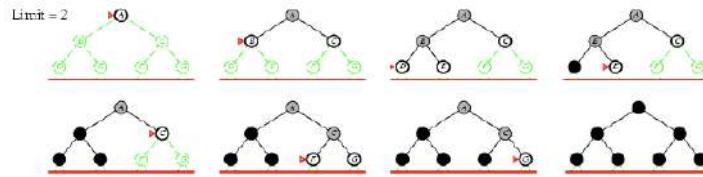
Iterative Deepening search L = 2

Figure 2.26

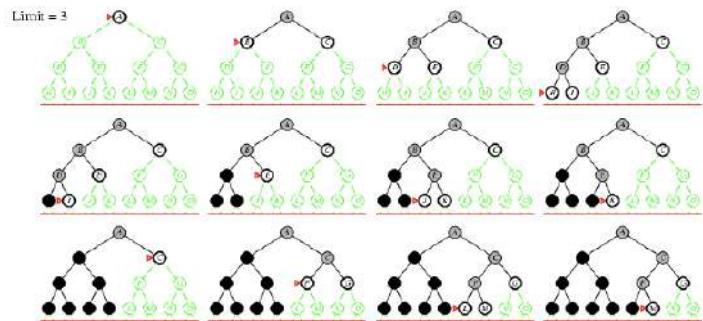
Iterative Deepening search L = 3

Figure 2.27

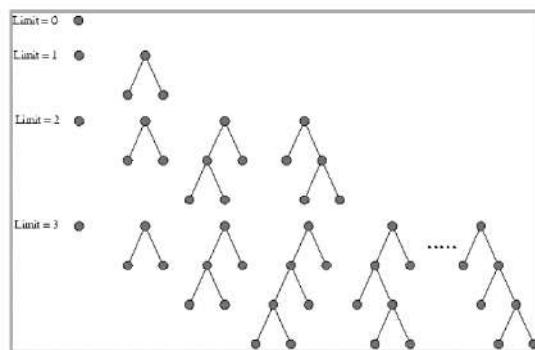
Iterative Deepening search

Figure 2.28

Properties of iterative deepening search

Main properties.

- Complete? Yes.
- Time? $(d + 1) \cdot b^0 + d \cdot b^1 + (d - 1) \cdot b^2 + \dots + b^d = O(b^d)$.
- Space? $O(b \cdot d)$.
- Optimal? Yes, if step cost = 1. Can be modified to explore uniform-cost tree.

Where:

- b - maximum of the search tree branching;
- d - depth of the least-cost solution;
- m - maximum depth of the state space (may be infinite).

Comparison of search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Figure 2.29

b = branching factor; d = depth of the solution; m = maximum depth of the search tree; l = depth limit.

2.2 Informed Search strategies

The intelligence of a system cannot be measured in terms of search capacity, but in the ability to use knowledge about the problem to reduce/mitigate the combinatorial explosion.

If the system had some control on the order in which possible solutions are generated, then it would be useful to use this order so that solutions have a high chance to appear before.

Intelligence, for a system with limited processing capacity is the wise choice of what to do next.

Informed strategies

According to Newell-Simon.

- Uninformed search methods in a search space of depth d and branching factor b have space complexity proportional b^d to find a goal in one of the leaves.
- This is unacceptable for certain complex problems. So we can resort to expanding the nodes using heuristic domain knowledge (evaluation functions) to decide which node to expand first.

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

The time spent to evaluate a node by means of a heuristic function should correspond to a reduction in the size of the space explored.

- Trade-off between the time it takes to solve the problem (baseline) and time spent in reasoning on top of it (meta-level).
- Uninformed search strategies have no meta-level activities.

Problems.

- How to find the correct evaluation functions, *i.e.* how to select the most ‘promising nodes’?
- It is difficult to numerically characterize the empirical knowledge about the problem.
- Not always the obvious choice is the best.

A first example

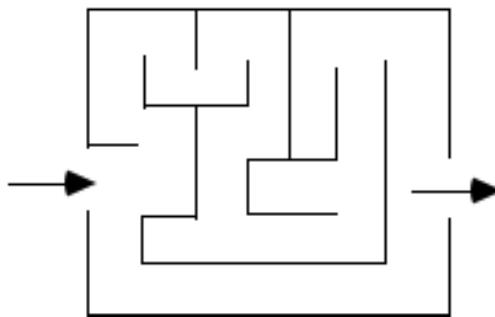


Figure 2.30

Heuristic choice: always move to reduce the distance from the exit.
In the case of this maze we would choose a longer way.

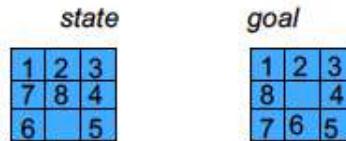
Example: game of 8

Figure 2.31

The *distance* from the goal can be estimated based on the number of boxes out of place:

- left-distance 2;
- right-distance 4;
- high-distance 3.

1	2	3
7		4
6	8	5

Figure 2.32

It might be that the heuristics does not estimate the exact distance from the solution:

- distance = 3 but 4 moves are needed to reach a solution.

Best-First search

Best-first search uses *evaluation functions* that compute a number that represents the desirability of the node expansion.

Best-first means that the chosen node is the one considered as the most desirable.

`QueueingFn` = insert successors in descending order of desirability.

Special cases.

- Greedy search or hill climbing.
- *A** search.

Best-First:

- try to move to the maximum (resp. minimum) of a function that ‘estimates’ the desirability (resp. cost) to reach the goal.

Greedy:

1. let L be a list of the initial nodes of the problem, ranked according to their distance from the goal (ascending order);

2. if L is empty fail, otherwise let n be the first node of L ;
3. if n is the goal return solution (the path to reach the goal);
4. otherwise remove n from L and add to L all the children of n , then order the entire list L based on an estimate of the relative distance from the goal and return to step 2.

function BEST-FIRST-SEARCH(*problem*, EVAL-FN) **returns** a solution sequence

inputs: *problem*, a problem
EVAL-FN, an evaluation function

Queueing-Fn \leftarrow a function that orders nodes by EVAL-FN
return GENERAL-SEARCH(*problem*, *Queueing-Fn*)

Figure 2.33

It uses the general search algorithm and *evaluation function* EVALFN.

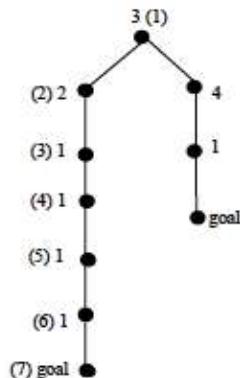


Figure 2.34

The *best-first* search is not optimal in the sense that it is not guaranteed to find the best solution, or *the best path* toward a solution (remember the maze?). This is because the *best-first* technique tries to find as soon as possible a node with 0 distance from the goal and not the node with the lowest depth.

Example

Let's reconsider the previous example which has the goal to be in Bucharest.

- Evaluation function $f(n) = h(n)$ (*heuristic*).
- $h(n)$ = estimate of the cost from n to the goal.
- E.g. $h_{SLD}(n)$ = straight-line distance between n and Bucharest.
- The *greedy best-first search* expands the node that *seems* closer to the goal.

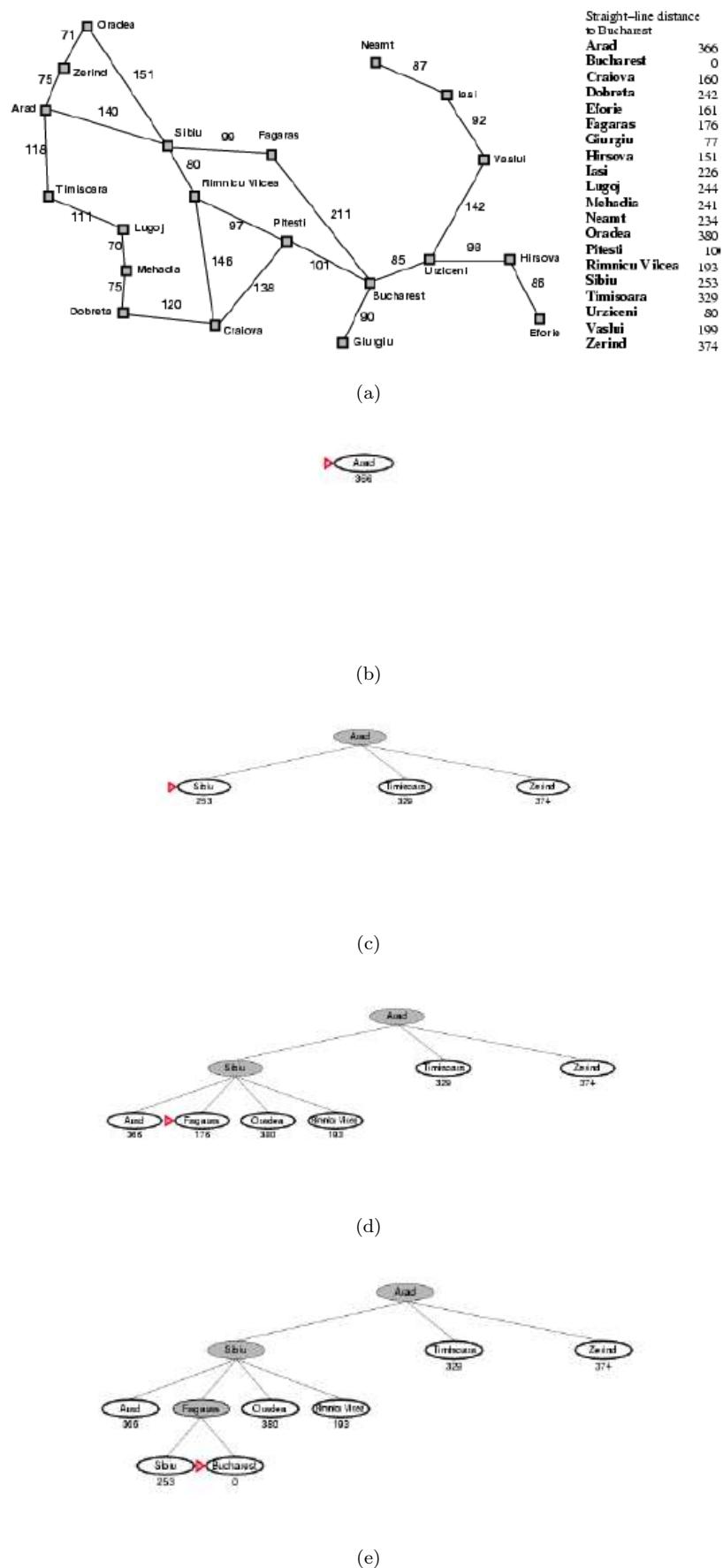


Figure 2.35

Best-first search: analisys

A brief analisys of *best-first* search:

- *best-first* search is not optimal and may be incomplete (the same pittfall as the depth-first search);
- in the worst case, if the depth is d and the branching factor b the maximum number of expanded nodes will be b^d (time complexity);
- in addition, it keeps in memory all nodes, the spatial complexity coincides with the temporal one;
- with a good heuristic function, the spatial and temporal complexity can be reduced substantially.

A* algorithm

A* algorithm:

- instead of only considering the distance to the goal, also consider the ‘cost’ in reaching the node n from the root;
- we expand nodes for increasing values of $f(n)$:

$$f(n) = g(n) + h'(n)$$

where $g(n)$ is the depth of the node, and $h'(n)$ the estimated distance from the goal;
- we choose the node to expand as the one for which this sum is smaller.

Basically we try to combine the benefits of depth-first search (efficiency) with the ones of uniform cost search (optimality and completeness).

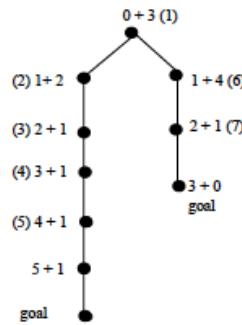


Figure 2.36

1. Let L be a list initial nodes of the problem.
2. If L is empty fail; otherwise let n be the node for which $g(n) + h'(n)$ is minimal.
3. If n is the goal return solution (the path to reach it).

4. Otherwise remove n from L and add to L all the children of n labeled with $g(n) + h'(n)$. Return to step 2.

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

Example

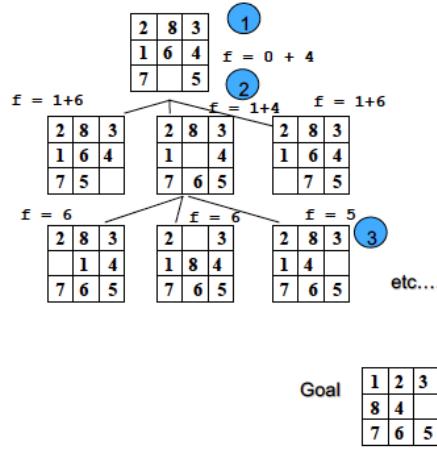


Figure 2.37

$$f(n) = g(n) + h'(n), \text{ where:}$$

- $g(n)$ = depth of the node n ;
- $h'(n)$ = number of tiles in the wrong place.

A* algorithm: analysys

A brief analysys of A* algorithm search:

- A* does not guarantee to find the optimal solution (it depends on the heuristic function);
- suppose you indicate with $h(n)$ the true distance between the current node and the goal;
- the heuristic function $h'(n)$ is *optimistic* that if we always have $h'(n) \leq h(n)$;
- this heuristic function is said to be *feasible*;
- theorem: if $h'(n) \leq h(n)$ for each node, then the A* algorithm always finds the optimal path to the goal;
- obviously the perfect heuristic $h' = h$ is always feasible;
- if $h' = 0$ we always obtain a feasible heuristic function \Rightarrow breadth-first search.

A* algorithm optimality proof

Theorem 2.2.1 (Optimality).

If $h'(n) \leq h(n)$ for each node, then the A* algorithm always finds the optimal path to the goal.

Proof: Suppose you have created a sub-optimal goal G_2 and to have it in the queue. Let n be an unexpanded node in the queue such that n is on the shortest path to the optimal goal G .

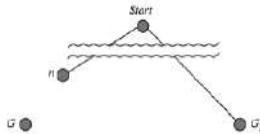


Figure 2.38

Consider:

- $f(G_2) = g(G_2)$ as $h(G_2) = 0$;
- $g(G_2) > g(G)$ as G_2 is sub-optimal;
- $f(G) = g(G)$ as $h(G) = 0$;
- $f(G_2) > f(G)$ from above;
- $h'(n) \leq h(n)$ as h' is feasible permissible;
- $g(n) + h'(n) \leq g(n) + h(n) = f(G)$ (n is on the path to G);
- $f(n) \leq f(G)$.

Then $f(G_2) > f(n)$, and A* will never select G_2 for expansion. ■

Example: search with feasible heuristic function

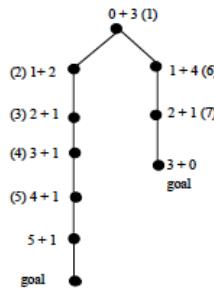


Figure 2.39

The previous choice for the game of 8 (counting the tiles in the wrong position as h') is always feasible because every move can reduce the distance to at most one unit.

At this point we discuss the following exercise not in detail, but just to understand the problem and the logic of resolution.

Exercise

Let us consider the following puzzle:

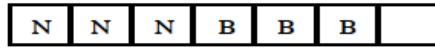


Figure 2.40

We have three black tiles, three white tiles and a blank one.⁶
Operators are the following:

- a tile can move to a blank one if this is close to it (cost = 1);
- a tile can move to a blank one jumping over at most two tiles (cost = distance covered, *i.e.* the number of jumped tiles plus 1).

The goal is to have all white tiles at the left of black tiles, no matter where the empty tile is.

Build a heuristic function $f(n) = g(n) + h'(n)$ for each node of this problem and show the search tree generated by the A* algorithm.

Heuristic functions

Number of tiles out of place: black tiles are out of place if they occupy positions 1, 2, 3 while the white tiles are out of place if they occupy positions 5, 6, 7.

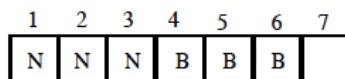


Figure 2.41

$$h'_1 = 5.$$

Is this admissible?

The distance between black tiles and positions 4,5,6.

Is this admissible? NO, in the following goal position we have:

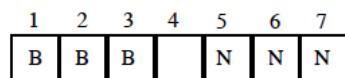


Figure 2.42

$$h = 0 \text{ while } h'_2 = 3.$$

How to correct it? We can use the minimum distance between the black tiles and any goal position. We call it h'_3 .

⁶'N' stands for 'nero' (italian word for 'black') and 'B' stands for 'bianco' (italian word for 'white').

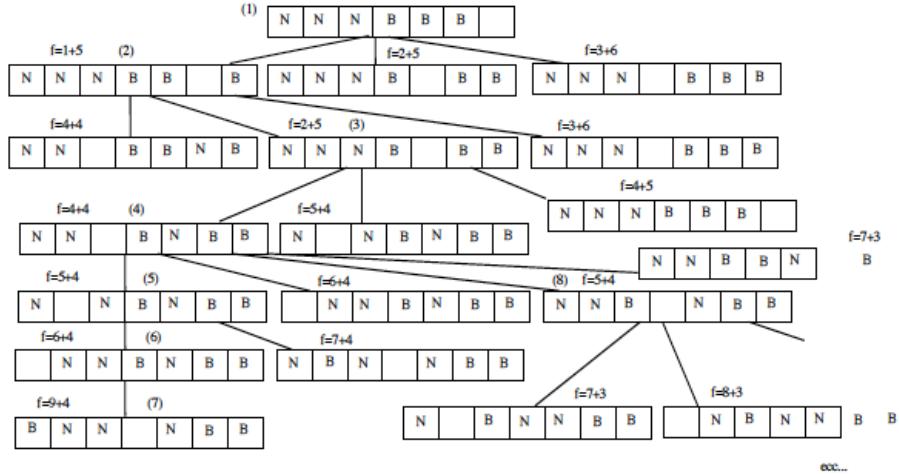
Search space with heuristic h'_1 

Figure 2.43

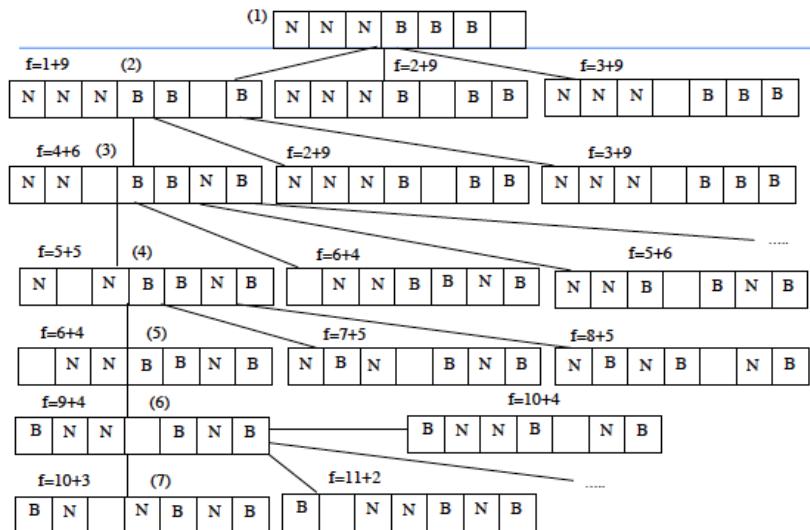
Search space with heuristic h'_3 

Figure 2.44

Let's return to discussion inherent A* algorithm.

A* example

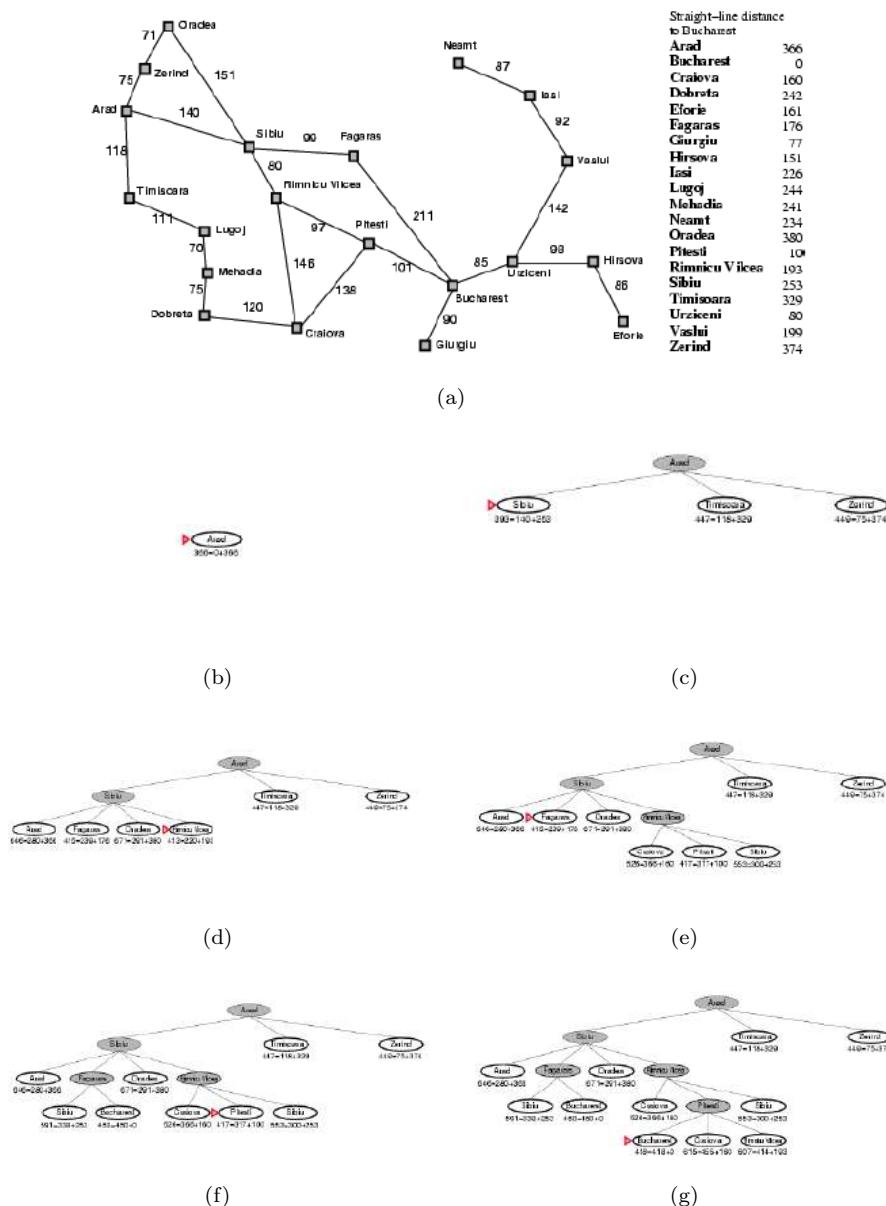


Figure 2.45

Eligible heuristic functions

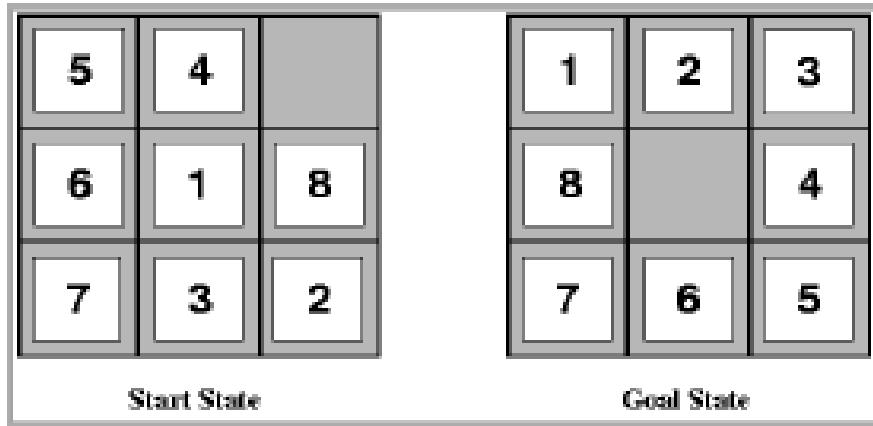


Figure 2.46

We can define different heuristics. For example.

- h_1 = number of tiles that are out of place ($h_1 = 7$).
- h_2 = the sum of the distances from the initial and final position of each tile. The distance is a sum of the horizontal and vertical distances (*Manhattan distance*). The tiles 1 to 8 in the initial state have a distance $h_2 = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$.

Both heuristics are eligible.

Let's consider the following point.

- As $h'_1 \leq h'_2$ then h'_2 is better.
- It is better to use a heuristic function with higher values, provided it is optimistic.
- How to invent heuristics?
- Often the cost of an exact solution of a relaxed problem is a good heuristic for the original problem.
- If the problem definition is described in a formal language you can build relaxed problems automatically.
- Sometimes you can use the maximum among different heuristics.
- $h'(n) = \max(h_1(n), h_2(n), \dots, h_m(n))$.

Example: game of 8

Description: a tile can move from square A to square B if A is adjacent to B and B is empty.

Relaxed problems remove some conditions:

- a tile can move from square A to square B if A is adjacent to B (Manhattan distance);
- a tile can move from square A to square B if B is empty;
- a tile can move from square A to square B in one hop (tiles out of place).

From trees to graphs

We have assumed so far that the search space is a tree and not a graph. It is therefore not possible to achieve the same node from different paths.

This assumption is of course simplistic: think of the game of 8, the missionaries and cannibals *etc.*

How to extend the previous algorithms for dealing with graphs?

Graph-search

```

function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed  $\leftarrow$  an empty set
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)

```

Figure 2.47

Search in graph with A*

Two lists: *open* and *closed nodes*.

- *Closed nodes*: expanded nodes are removed from the list to avoid further examination.
- *Open nodes*: nodes still to be examined.

A* searching in a graph instead of a tree.

Changes:

- the graph can become a tree with repeated nodes;
- add the list of closed nodes and assume that $g(n)$ evaluates the minimum distance of node n from the starting node;

A* algorithm for a graph

A* algorithm for a graph.

1. Let L_a be the open list of initial nodes of the problem.
2. Let n be the node for which $g(n) + h'(n)$ is minimal. If the list is empty, fail.
3. If n is the goal then stop and return the path to reach it.
4. Otherwise remove n from L_a , enter it in the list of closed nodes L_c and add to L_a all the children of n , labelling them with the cost from the starting node to n .
5. If a child node is already in L_a , do not add it, but update its label in case its cost (root to node) is smaller than the one it had.
6. If a child node is already in L_c , do not add it to L_a , but if its cost is better, update its cost and the one of its children and descendant.
7. Returns to step 2.

Example

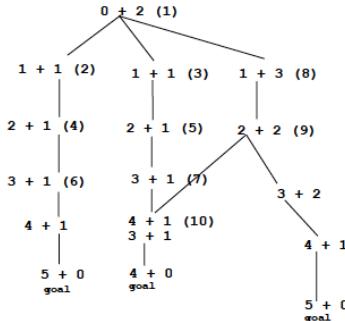


Figure 2.48

Let's propose two exercises.

Exercise 1

Consider the following game: three tiles and one blank.



Figure 2.49

The operator order is the following: blank moves right, then left, then up and then down. Show the search space generated. After that show how to explore this space with a depth first and then breadth-first.

Show how the state space changes if we do not allow any move that goes back to a previous state on the same path.

Solution:

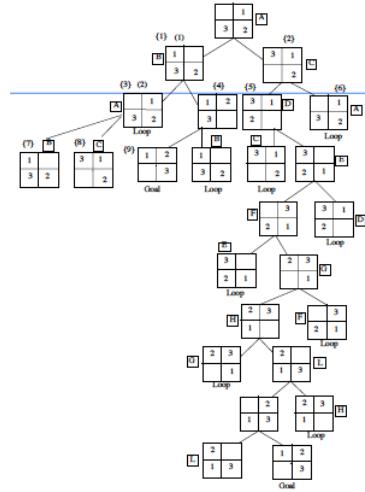


Figure 2.50

Solution:

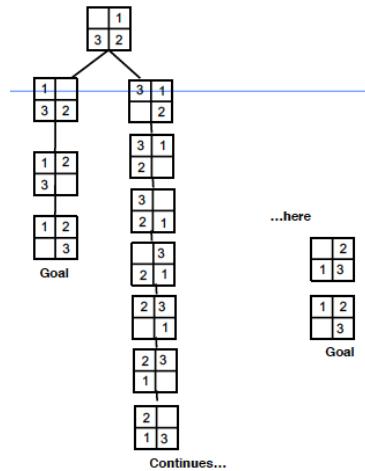


Figure 2.51

Exercise 2

We have two bins: the first has capacity of 5 liters and is full of water, the second has capacity of 2 liters and is empty.

- We want to obtain one liter of water in the second bin.

- We can transfer water from one bin to the other, we can throw water away, but we cannot obtain new water.
- Show the search space of the problem. Explain after how many steps we reach the goal using breadth-first and depth-first.

Note: here we also have *failure node* which wasn't present in the previous exercise.

Solution:

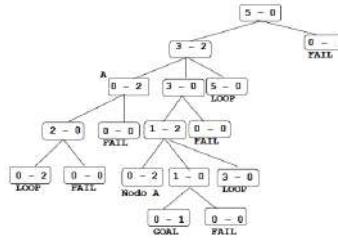


Figure 2.52

Consistent heuristic (monotone)

A further condition of h : *consistency* (or *monotonicity*)

Definition 2.2.2 (Consistent heuristic).

A heuristic is *consistent* if for each node n , any successor n' of n generated by each action a :

- $h(n) = 0$ if the corresponding status is the goal;
- $h(n) \leq c(n, a, n') + h(n')$ otherwise.

With monotonicity, we are guaranteed to find the shortest path from the root to the goal.

We report a theorem, the proof of which we can skip.

Theorem 2.2.3 (A* optimality using graph-search).

If $h(n)$ is consistent then A^* using graph-search is optimal.

Proof: if h is consistent we have that:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

$f(n)$ never decreases along a path.

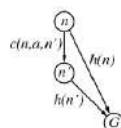


Figure 2.53

■

Before seeing the following exercise please review Exercise 1 and Exercise 2 in the previous pages.

Exercise 3

We have 12 coins in three piles. In the first pile we have 4 coins, in the second we have 7 of them, while the third we have a single coin.

We have a rule for moving coins: we can move coins from a pile A to a pile B if coins in B double.

Show the search space⁷ starting from the configuration $<4, 7, 1>$ and the goal is $<4, 4, 4>$.

Solution:

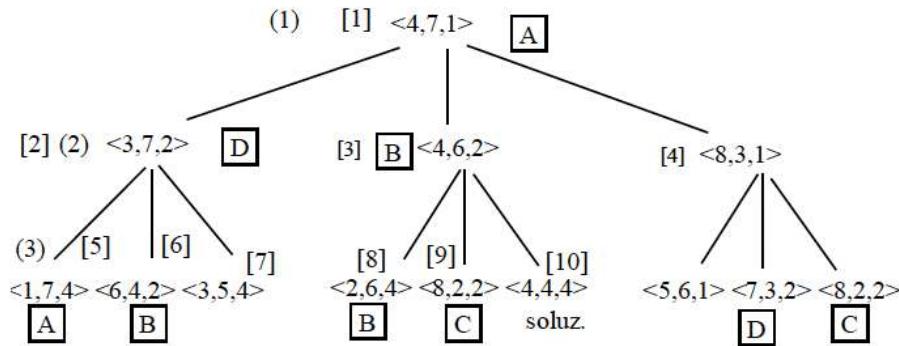


Figure 2.54

Note that the problem presents a symmetry: $<4, 7, 1>$ is equal to $<1, 7, 4>$. Looking at the figure above we highlight that to have the entire search tree we should have explored also the unexplored $<3, 5, 4>$ and $<5, 6, 1>$ in the third expansion line, but to avoid to make the tree too long these expansions are omitted.

Informed search strategies

Remember:

- a heuristic function $h(n)$ is *admissible* if it never overestimates the cost for reaching the goal;
- a heuristic function $h(n)$ is *consistent (monotone)* if, for each node n , for every successor n' of n applying an operator a , the following holds:

$$h(n) \leq c(n, a, n') + h(n')$$

⁷Note for the exercises. When it is asked to show the search space we need to write anything: all possible operators applied to all possible nodes; we should enumerate all possible steps in order to make clear in what order we explore. If we find a *loop* for instance, we write '*loop*' and through some letters as reference, we highlight the process that is involved in the loop. An example of *loop* could be to find two equal nodes in the *same branch* (or in the same logic unit for which the same process would repeat), in this case we write the same letter next to each of the two nodes.

Exercise 4

Review the following subsections from page 89 to page 90.

- Exercise.
- Heuristic functions.
- Search space with heuristic h'_1 .
- Search space with heuristic h'_3 .

Exercise 5

Let us consider the following graph. Red numbers on arcs are costs while green numbers on nodes are heuristic values. Arcs are not oriented so they can be traversed in both directions. Consider A as the initial state and G as a goal.

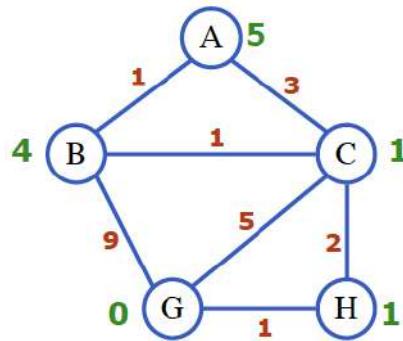


Figure 2.55

Show the A* search tree avoiding loops.

Answer to the following questions.

- Is the heuristic consistent and admissible?
- Does the algorithm find the minimum cost path to the goal? If not what should I change in the heuristic to make it consistent?

Solution (Russel-Norvig A* on graphs):

Actual path	Cost	Heuristic	List of expanded nodes
A	0	5	

A
 $\mathbf{0+5 = 5}$

Figure 2.56

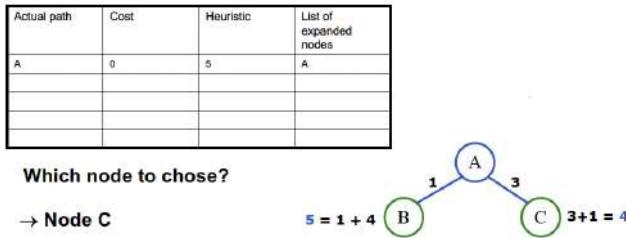


Figure 2.57

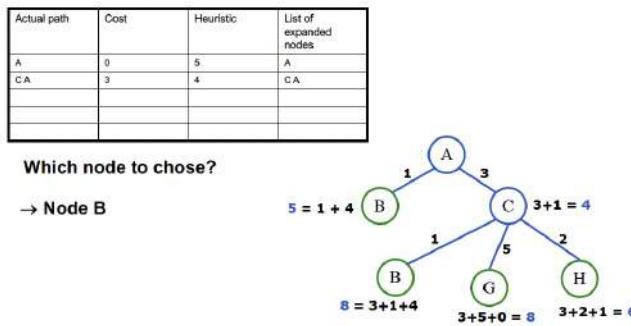


Figure 2.58

Looking at the previous image we can see that in the tree we have two *B* letters, but we are not into a loop because they are in two different branches.

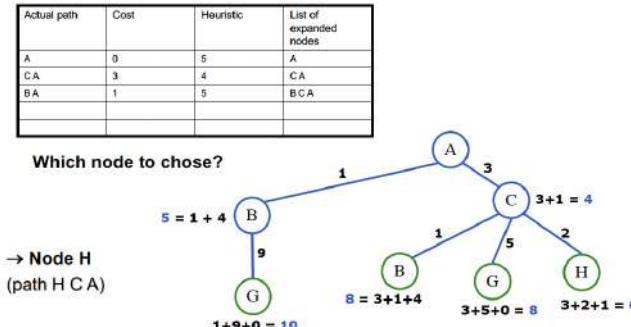


Figure 2.59

In the figure below maybe there is some error; something was not clear during the lesson.

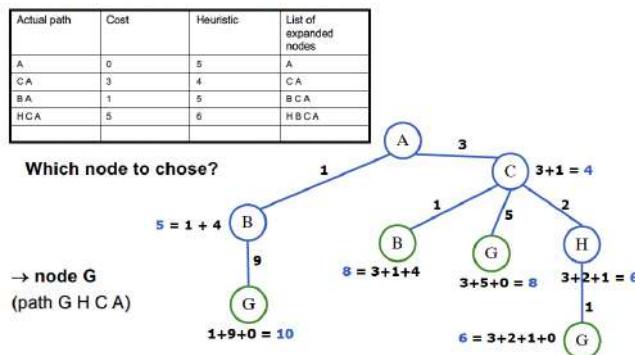


Figure 2.60

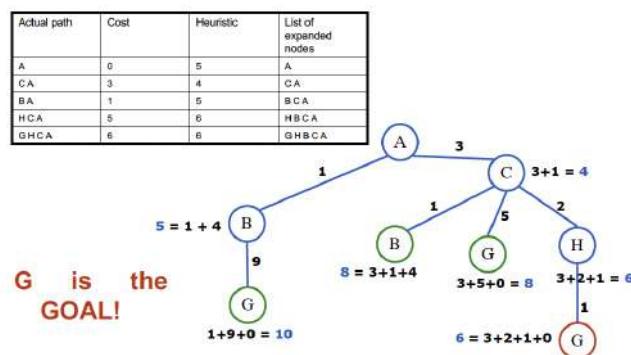


Figure 2.61

Some observations.⁸

- The solution found is: ACHG, with a cost of 6.
 - This is not the optimal solution. ABCHG has cost of 5.
 - This happens because the heuristic function is not consistent. In this case the A* algorithm for graphs does not guarantee to find the optimal solution.
 - $h(n) \leq c(n, a, n') + h(n')$.
 - Optimality in a A* algorithm on graphs is guaranteed if the heuristic is consistent.
 - For obtaining optimality, we have two ways.
 1. Chose a consistent heuristics.
 2. Modify the algorithm and updating the cost to reach a node (the g cost) each time a node is reached through a better cost path. This update should then be propagated properly.

⁸At this point you can skip these observations, they were not present during the lesson, we just report here for completeness; you can directly pass to the following Exercise.

- How to correct the heuristics?
- If we suppose that the heuristic value of C is 3, then the heuristic will be consistent and the A* algorithms for graphs will select the optimal path.

1. $L_a = [A]$, $L_c = []$
2. $L_a = [(AB, 5), (AC, 6)]$
 $L_c = [A]$
3. $L_a = [(ABC, 5), (AC, 6), (ABG, 10)]$
 $L_c = [A, B]$
4. $L_a = [(ABCH, 5), (AC, 6), (ABCG, 7), (ABG, 10)]$
 $L_c = [A, B, C]$
5. $L_a = [(ABC HG, 5), (AC, 6), (ABCG, 7), (ABG, 10)]$
 $L_c = [A, B, C, H]$
6. Select G and it is the goal.

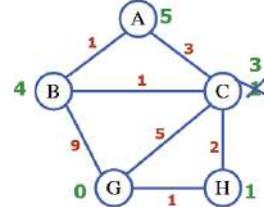
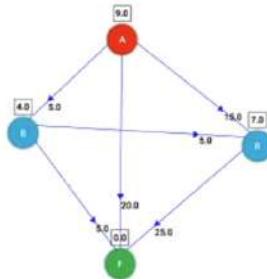


Figure 2.62

Exercise

Exercise 2 (6 points)

Consider the following graph, where A is the starting node and F the goal node. The number on each arc is the cost of the operator for the move. Close to each node there is the heuristic evaluation of the node itself, namely its estimated distance from the goal:



- Show the search tree generated first Best-First and then by the A* algorithm along with the order of expansion of nodes. In case of ties, chose the node to expand in alphabetical order. Consider as the heuristic $h(n)$ the one indicated in the square close to each node in the figure.
- Is the heuristic admissible?
- Which is the cost of the path found by Best-First and by A*?

Figure 2.63

Solution:

- Best-First (cost of the path: 20)

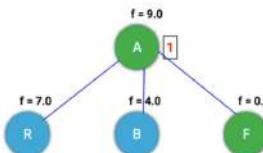


Figure 2.64

- A* (admissible heuristic; cost of the path: 10)

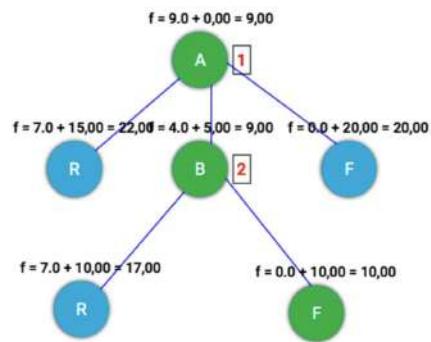


Figure 2.65

Chapter 3

Local search algorithms

3.1 Local search

In many cases treated so far we've seen that we can solve a problem through the exploration of a search space and this search space is represented by a tree search (a structure that for each node you have a parent, many children...) essentially we give an initial state, a goal to reach, and we build a path to reach it through operators. Very traditional way to solve AI problems; basically optimization problem.

The problem of solving in this way is that the space grows exponentially with the problem dimension; grows exponentially in general. So we have two ways to approach this.

- One is to explore the tree search in a smart way through heuristics that for example guide us fast to a solution.
- Or, as we will see, techniques that eliminate (with certain constraints) parts of the search space because they are for example infisible with constraints, you will not find any solution there, so you cut them. Even graph search but in general we refer to tree search.
- Or other smarter ways.

Let us give an intuition of the meaning of *local search*: when you want to build a solution that, for example, start from a city and bring us to another city, then in the search tree at each step you have to reach a city, so you're basically creating a solution that is a path and at each level of the tree you're building a little part of this path, so this is called *constructive algorithm*.

Now suppose we're in a completely different environment: we pick a random solution path, then we change it a little (*e.g.* changing a little some parameters) and consequently to this find another solution, if this new solution is better we keep it otherwise we keep the first and we continue in changing.

Local search: general overview

Some main points:

- the *constructive algorithms* (seen so far) generate a solution by adding to a starting (empty or initial) state solution components in a particular order;
- *local search algorithms start from an initial solution and iteratively try to improve it through local moves*;
- local moves define a neighborhood;¹
- applicable when the path to reach a goal/solution is not important;
- several problems (for example, the n queens problem)² have the property that the state description contains all the information necessary for understanding if one configuration is a solution or not (the path is irrelevant);
- in this case local search algorithms often provide a valid alternative;
- for example, we can start with all the queens on the board and then move them to reduce the number of attacks.

N queens problem

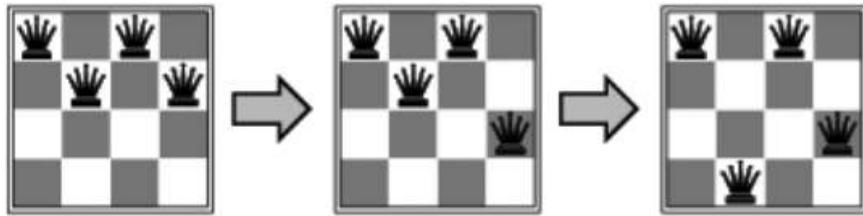


Figure 3.1

Put n queens on a chessboard so that they do not attack each other.³

Looking at the image above we have that we start from a random initial configuration; then we move down the last queen on the right (very little change, *local move*, we took just one queen and we changed it [this just to show you what we

¹If we find a better solution, we go there.

²We have a chessboard ($n \times n \longleftrightarrow \text{rows} \times \text{columns}$) and we want to place n queens (queens are chess pieces that attack in horizontal, in vertical and on the 2 diagonals; furthermore we cannot have 2 queens on the same row, column and diagonal at the same time) 1 per row and 1 per column so that they will not attack each other. If we want to use a *constructive* method it means that at each step we place just one queen: first row \rightarrow one queen, second row \rightarrow second queen... adding 1 queen per step. In *local search* instead we can put queens randomly in our chessboard and then we move them, shift them a little bit from one configuration to another and you stop when no queens attack other queens. So are 2 completely different ways of solving: *local search* \rightarrow goes from one config to another, best for optimality problem (like finding minimum cost path/shortest path...) and in general a problem that permits to start with a solution and then from it begin to optimize; *tree (graph) search* \rightarrow explore the entire search space. What is the difference? With *tree search* you can in principle find one that is optimal, you're guaranteed to find the optimal solution; with *local search* you have to forget this optimality, you have not guarantees, but in many cases if finding an optimal solution is too computational expensive with search tree, you can solve with *local search* and accept a solution that is good enough even if it is not optimal.

³This is the goal.

mean with *local move*]) landing to a second configuration; then... landing to a third configuration.

So how can I measure if, for example, the first configuration is better than the second configuration? In this case our measure of how good is a solution is the number of constraint violations, so *how many queens are under attack*:

- first configuration: all 4 queens are under attack;
- second configuration: 3 over 4 are under attack;
- till reaching the 0 conflict.

So this is one possible path using *local search* that at each step is improving the quality of a solution.

Starting from an initial configuration, obviously we have a lot of *local move* we can perform (take the first queen, second... move down of 3 cells, 2 cells and so on). In general the *set* of all possible configurations that we obtain starting from a solution applying one local move is called *neighborhood*.

Neighborhood

Definition 3.1.1 (Neighborhood).

Neighborhood structure:

$$N : S \longrightarrow 2^S \quad (3.1.1)$$

Assign to every $s \in S$ a set of neighbors $N(s) \subseteq S$. $N(s)$ is called the neighborhood of s .

So the *Neighborhood* is a function N that goes from a set S (possible states) to the *power set* 2^S (the set of all subset of S) of the possible states.

Example: TSP

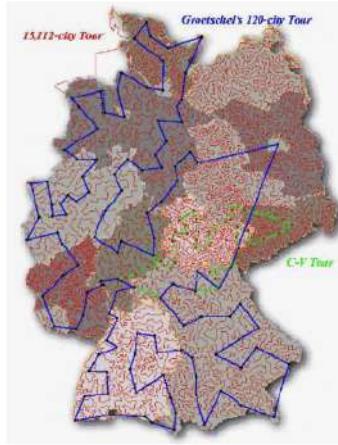


Figure 3.2

Let's try to consider an example of neighborhood.

The ‘*Travelling Salesman Problem*’ (TSP) is a well-known in the literature; there are many problems approached with this technique, logistic problems.

Given an undirected graph, with n nodes and each arc associated with a positive value, find the Hamiltonian tour with the minimum total cost.

Short detailed introduction: you have a graph (undirected), nodes basically represent cities and you want to travel covering all cities but visiting each city only once (enter the city, leaving the city and you never visit it again); so you’re basically creating a path, this path should be closed in a cycle (not just a path like in a graph), a circuit and it is *Hamiltonian* (*Hamiltonian circuit*) because each city is visited once. It’s interesting because we can give it a mathematical formalization.

Neighborhoods for TSP: k-exchange

How we try to solve a problem like TSP utilizing *tree search*? We start from a city, then at each level of our search tree we try to reach our city, and we reach the goal when you have performed n levels (where n is the number of cities we have to cover) and you end on the first one (return to the initial city).

We can also solve using *local search*. How?

We start with a cycle (just a random cycle), whatever it is, and then we can change it according to some local moves; the most traditional moves for TSP is called the *k-exchange*.

1. 2-opt⁴ performs two exchanges.

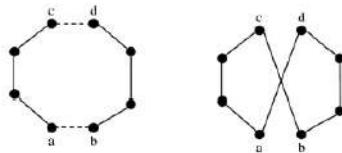


Figure 3.3

2. 3-opt⁵ performs three exchanges.

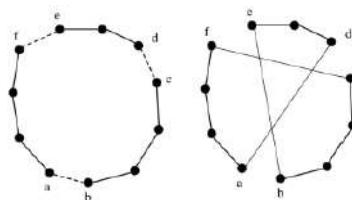


Figure 3.4

⁴2-opt is the name of the neighborhoods structures

⁵Pick 3 arcs, remove them from the solution and you create another solution by conjuncting vertices (obtaining a circle) and we need to compare the new solution with the first one. Of course in the 3-opt we would have more many neighborhoods than 2-opt because we can pick combination of 3 arguments in comparison to 2 and in general k args with $k \in \mathbb{N}$.

So, for example, considering the first case we have a circuit, we pick randomly 2 arcs (a-b and c-d), remove them from the solution, and then we add 2 other arcs (c-b and d-a), so basically remove 2 arcs and connect vertexes in a different way. How many of these neighborhoods we have? In a set of 8 arcs (we have 8 segments conjuncting 8 points) you can pick a couple of arcs (among 8) and then every time you select these two arcs you have a single neighborhood that is created by connecting c-b and d-a.

Why not c-a and b-d? Because otherwise we would not have a circuit (we would obtain 2 disconnected parts).

We could assume solution represented as a permutation of numbers $(1, 2, \dots, n)$ ⁶
Some neighborhoods:

- 2-swap: swap two numbers in the sequence;
- k-cycle: select k positions and cycle on their numbers: *e.g.* $(1, \mathbf{2}, 3, \mathbf{4}, \mathbf{5}, 6, 7, 8)$
 $\longrightarrow (1, \mathbf{5}, 3, \mathbf{2}, \mathbf{4}, 6, 7, 8)$ (first cycle meaning for example that you go from city 1 to city 2, from 2 to 3 etc.) where:
 - $(1, \mathbf{2}, 3, \mathbf{4}, \mathbf{5}, 6, 7, 8) \longleftrightarrow s \in S;$
 - \longrightarrow represents the N application *i.e.* just a neighborhood application;
 - $(1, \mathbf{5}, 3, \mathbf{2}, \mathbf{4}, 6, 7, 8) \longleftrightarrow N(s)(\subseteq S) \in 2^S.$

Iterative improvement - Hill climbing

The simplest *local search* algorithm is called *hill climbing*, the basic one (no simpler algorithm exists):

- start with an initial solution;
- apply the neighborhood;
- if I find a solution that is improving the current solution, then I pick this better solution, I move there.⁷ Basically I move from a solution to another only if the second solution is better than the first one. If in a neighborhood I don't find any improving solution the algorithm stops: look around me, there is nothing better, then I stay here.

So summarizing on iterative improvement - Hill climbing algorithm:

- very basic local search algorithm;
- a move is only performed if the solution it produces is better than the current solution (also called hill-climbing);
- the algorithm stops as soon as it finds a local minimum;
- several variants: *best*, *first*, *stochastic first*, etc.

⁶This could be a good convention: cities can be represented by numbers and any permutation of the set of numbers represent a solution, a circle, the important thing is that a permutation must represent a connected circle, not a disconnected one. So some permutations are not feasible, but we can manage this distortion by exploring the neighborhoods and discarding them according to some conditions, then you compare with the previous solution to see what is better.

⁷For this reason is called *hill climbing*.

- *first*: as soon I find one improving solution I move there \longleftrightarrow I start to create new solutions with local moves and, as soon as I find a better one I move there.
- *best*: I analyze the entire neighborhood anyway and then I pick the best one in the neighborhood \longleftrightarrow I can explore the all neighborhood, generate always the entire neighborhood, then I pick the best one in the neighborhood and I move there.

So what is the problem here? The problem is that when you are in a solution that is optimal locally this is not a guaranty that the solution is optimal globally, because we are just looking a narrow part of the tree (our neighborhood)

Iterative improvement

High-level algorithm:

```
s <- GenerateInitialSolution()
repeat
    s <- BestOf(s, N(s))
until no improvement is possible
```

Main drawbacks:

- not effective in exploring the search space (*e.g.* it stops at the first local optimum encountered);⁸
- does not remember the search states already reached.⁹

Summarizing quote:

‘Like climbing Everest in thick fog with amnesia.’

Local optima

So, what we are looking for? We are looking for *global optima*; so consider the following aspects:

- a *local maximum* is a solution s such that for any s' belonging to $N(s)$, given an evaluation function f :

$$f(s) \geq f(s')$$

- when we solve a maximization problem we look for a *global maximum* s_{opt} *i.e.* such that *for any* s :

$$f(s_{opt}) \geq f(s)$$

- analogous considerations for the *minimum*;
- the larger the neighborhood the more likely a *local maximum* is a *global maximum* (but obviously there is a problem of computational effort). So, the larger the neighborhood the higher the quality of the solution.

⁸Not global optima.

⁹Basically it means it does not remember the states visited, so it may happen to visit the same state a lot of time.

We can have also very effective *mixed search strategy*: for example, mixing *local search* with *tree search*:

- they create neighborhoods that are very very large;
- how they solve these large neighborhoods? By exploring them exhaustively through *tree search*.

Other problems



Figure 3.5

Suppose we can give a functional representation of our solution states in our search space as the graph presented in the image above; think of it as 3 mountains that should be climbed and points along the mountains are solutions.

- Local Optima:
 - states that are better than all the neighbors, but worse than other states that are not nearby.
- Plateau:
 - flat areas of the landscape in which neighboring states have the same value. In which direction to move? (Random choice).¹⁰
- Ridge:
 - it is a higher area adjacent to those where we should go, but we can not go there directly. So we need to move in another direction to get there.

Consider the ‘first mountain’ in the figure above. Suppose you start with a solution at very base level on the left. As arrows show exploring the neighborhoods we find improving solution till point ‘A’. But then, exploring this neighborhood we find that there is no improving solution (consider the directions of the arrows near ‘A’). So ‘A’ represents a *local optima* while we can see that is ‘B’ the *global optima* (max value of the function). So how we can reach ‘B’? Or by chance (starting with a solution that falls in the ‘red wall’ of the figure) or we need to approach with other strategies.

¹⁰So this can limit the efficiency of a local search algorithm.

Landscape

The landscape is a visual representation of the evaluation function associated to all possible states of a problem.

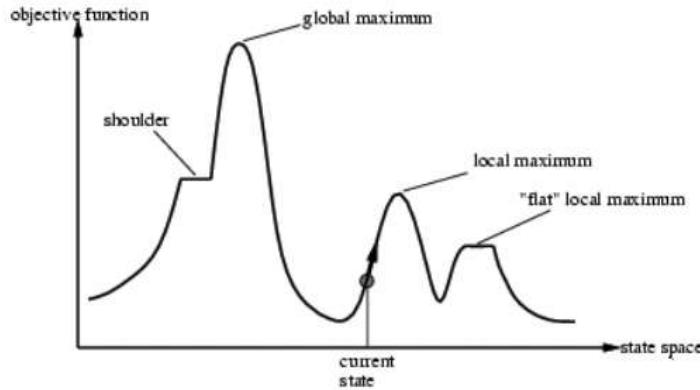


Figure 3.6

Meta heuristics

Local search has many pitfalls:

- does not remember anything;
- only *local optima*;
- really unlikely it provides a very good solution (in particular if the neighborhood is too small).

With the scope of reducing these collaterals, *local search* is never used in isolation. But *local search* is a very important component of a huge set of algorithms that are called *meta heuristics* (used a lot in industries applications).

More effective and general search strategies are required.¹¹

For example:

- accept non-improving moves;¹²
- change neighborhood or cost function during search;
- use high-level search strategies.

Local search can be seen as search processes over a graph.

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

- *Neighborhood graph* to represent search space topology.¹³
- *Search graph* to represent the actual search space exploration.

¹¹Just a very simple and good improving could be just exploring locally and restart from different solutions many times but remembering states and comparing them in the different explorations.

¹²Moves that ends in a worse solution. Not always, sometimes, for example probabilistically.

¹³Basically gives me the structure of the neighborhood of a given state.

Neighborhood graph

Neighborhood of $s :=$ set of neighbors of s with Hamming distance equal to 1.¹⁴

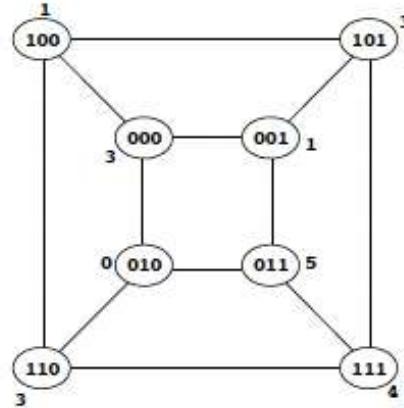


Figure 3.7

Looking at the image above, suppose I have a solution s that has 3 components, 3 bits, the possible states of the possible neighborhoods represent a neighborhood graph with each node with one possible solution ((100), (101), ...) and arcs represent the neighbors.

So, for example, if I start with the solution $s = (000)$ then $N(s) := \{ (001), (010), (100) \}$, each solution has 3 neighbors.

Search graph

The *search graph* is a graph where you move from one arc to another, for example, with a given probability.

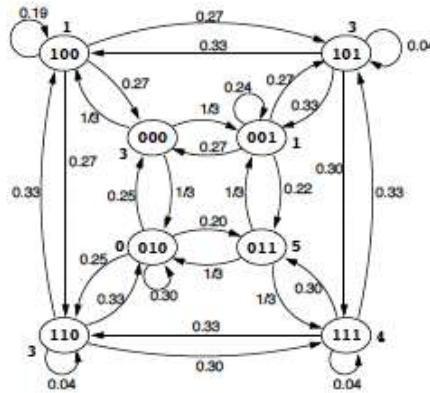


Figure 3.8

¹⁴Flipping one bit creates a solution that is of the neighborhood of S .

Simulated annealing

Let's start with one of the most famous *meta-heuristic* – called *simulated annealing* – which is a very frequently used algorithm; consider the followings about it:

- originates from statistical mechanics (Metropolis algorithm);
- it allows moves resulting in solutions of worse quality than the current solution;
- the probability of doing such a move is decreased during the search;¹⁵
- usually: $P(\text{accept down-hill move } s') := \exp\left(-\frac{f(s') - f(s)}{T}\right)$ ¹⁶

Simulated annealing: high level algorithm

The high level algorithm of *simulated annealing* could be represented like the following:

```

s <- GenerateInitialSolution()
T <- T0
while termination conditions not met do
    s' <- PickAtRandom(N(s))
    if f(s') > f(s) then
        s <- s' # s' replaces s
    else
        Accept s' with probability P(T, s', s)
    end if
    Update(T)
end while

```

Where:

- $s \leftarrow \text{GenerateInitialSolution}()$ \longleftrightarrow start with generating an initial solution;
- $T \leftarrow T_0$ \longleftrightarrow initialize T with T_0 ;
- **while** termination conditions **not** met **do** \longleftrightarrow this algorithm goes on till reaching termination condition that can be reaching the limit of a number in the iteration, reaching a particular state and so on;
- **if** $f(s') > f(s)$ **then** \longleftrightarrow if **if** $f(s') > f(s)$ always replace s with s' ;
- **Accept** s' **with probability** $P(T, s', s)$ \longleftrightarrow otherwise, if not, (s' is worse than s) then accept s' as new solution with probability $P(T, s', s) \sim \exp\left(-\frac{f(s') - f(s)}{T}\right)$
- **Update(T)** \longleftrightarrow at the end of the above cycle we update T meaning that we are reviewing the probability of selecting a worse node.

¹⁵So, in general, you explore a neighborhood and you move only if the possible new state is better than the previous one, but sometimes with a decreasing probability you move to a worse solution.

¹⁶ T usually represents *temperature*; main application is mechanics (so temperature of a metal etc.).

Simulated annealing: other considerations

The temperature T can be varied in different ways.

- Logarithmic $\longleftrightarrow T_{k+1} = \frac{T_k}{\log(k+k_0)}$
 - The algorithm is guaranteed to converge to the optimal solution with probability 1. Too slow for applications.¹⁷
- Geometric $\longleftrightarrow T_{k+1} = \alpha T_k$ where $\alpha \in (0, 1)$.¹⁸
- Non-monotonic: the temperature is decreased (intensification¹⁹ is favored), then increased again (to increase diversification²⁰).

Intensification and *Diversification* are two important term in *meta-heuristic*.

- *Intensification* is the part of the search space in which you explore the neighborhood, intensify your solution in a neighborhood of a given point, given state, you explore with local moves the same neighborhood, the neighborhood is mostly the same.
- *Diversification* is when you jump somewhere else in your space.

So, for example, the restart is a *diversification* while the neighborhood exploration is an *intensification*.

Tabu search

Another very widely used and well known meta heuristic algorithm which is a *local search* algorithm with *memory* is the *tabu search* algorithm; so we are targeting one of the main drawback of *local search* which is not remembering the states already visited (filling *amnesia*). More in detail.

- Explicitly exploits the search history to dynamically change the neighborhood to explore
 - *Tabu list*:²¹ keeps track of recent visited solutions or moves²² and forbids them \implies escape from local minima and no cycling.²³
- Many important concepts developed ‘around’ the basic TS version (*e.g.* general exploration strategies).

High level algorithm:

```
s <- GenerateInitialSolution()
TabuList <- []
```

¹⁷It could take basically infinite time.

¹⁸Here at every step you basically are reviewing temperature by calibrating the α parameter.

¹⁹So when I decrease a temperature I focus on improving moves, I focus on a given neighborhood.

²⁰When I increase temperature again I’m more favorable of accepting worsening moves and with worsening moves I jump somewhere else.

²¹A set of the states that you don’t want visit again.

²²Or even similar configurations.

²³Because if you have a local minima that goes in the *Tabu list*, you never reach it again.

```

while termination conditions not met do
    s <- BestOf(N(s) / TabuList)
    Update(TabuList)
end while

```

Where:

- **s** <- **GenerateInitialSolution()** \longleftrightarrow assign to *s* an initial state;
- **TabuList** <- [] \longleftrightarrow assign at the beginning the empty list to the **TabuList**;
- **s** <- **BestOf(N(s)/ TabuList)** \longleftrightarrow I pick the best solution from the neighborhood minus the states in the **TabuList**.
- **Update(TabuList)** \longleftrightarrow then I update the **TabuList** with new *s*. Of course the **TabuList** cannot only be increased because otherwise after a while you have a huge amount of memory occupied, so you keep states on the **TabuList** for a given amount of time and then they are removed.

Storing a list of solutions is sometimes inefficient, therefore moves²⁴ are stored instead.

But storing moves we could cut good not yet visited solutions.

We use *Aspiration Criteria* (*e.g.* accept a forbidden move toward a solution better than the current one).²⁵

Tabu search: more in detail

High level algorithm:

```

s <- GenerateInitialSolution()
InitializeTabuLists(TL1, ... ,TLr )
k <- 0
while termination conditions not met do
    AllowedSet(s,k) <- {z in N(S) | no tabu condition
                           is violated
                           or at least one
                           aspiration condition
                           is satisfied}
    s <- ChooseBestOf(AllowedSet(s,k))
    UpdateTabuListsAndAspirationConditions()
    k <- k + 1
end while

```

Where:

- **InitializeTabuLists(TL1, ... ,TLr)** \longleftrightarrow we can in more details initialize with several **TabuList**, for example TL1, ... ,TLr;
- **aspiration condition is satisfied** \longleftrightarrow *Aspiration condition*: even if is in a **TabuList** but such state/move is improving I can accept it anyway.

²⁴That you don't want to do.

²⁵We obviously can create many variants of this *Tabu Search*.

Iterated local search

The third meta heuristic that we see is the *iterated local search*.

Uses two types of SLS²⁶ steps.

- *Subsidiary local search steps* for reaching local optima as efficiently as possible (intensification).
- *Perturbation steps* for effectively escaping from local optima (diversification).

Also: acceptance criterion to control diversification vs intensification behaviour.
High level algorithm:

```

s0 <- GenerateInitialSolution()
s* <- LocalSearch(s0)

while termination conditions not met do
    s' <- Perturbation(s*, history)
    s*' <- LocalSearch(s')
    s* <- ApplyAcceptanceCriterion(s*, s*', history)
end while

```

Where:

- `s0 <- GenerateInitialSolution()` \longleftrightarrow start with an initial solution;
- `s* <- LocalSearch(s0)` \longleftrightarrow apply `LocalSearch` to `s0` finding `s*` which improves `s0` (`s*` local maximum or local minimum); it comes from a round of local search;
- `s' <- Perturbation(s*, history)` \longleftrightarrow considering also `history` we perturb `s*` going somewhere else and reaching `s'`;
- `s' <- Perturbation(s*, history)` \longleftrightarrow *diversification*;
- `s*' <- LocalSearch(s')` \longleftrightarrow *intensification*;
- `s* <- ApplyAcceptanceCriterion(s*, s*', history)` \longleftrightarrow on the basis of `history`, `s*` (first local max/min), `s*' (second local max/min)`, and following some *acceptance criteria* we chose if accept `s*` or not.

Lesson learned

So, when do we apply these algorithms? If you want explore and solve a problem using *tree search* (in particular we are referring to *optimization problem*) the first thing that we have to do is to *model the problem*. But using *metaheuristics* you don't have to model the problem, just have to define:

- what is a *state*;
- how to move from one state to another (what is a *neighborhood*);

²⁶Specific local search.

That's it. So it is not difficult to create such an algorithm but making them really work is not so easy; because there are *many parameters* (like *temperature*, α , the way I define the probability, how to structure the **TabuList**, etc.). Consider also that we have seen super simplified metaheuristic; the real ones have a very large number of parameters. So the number of parameters is a problem but also to find a proper tuning for each of them.

So, summarizing, what we have learned:

- modeling is an issue;
- high level search strategies have to be applied to effectively explore the search space (intensification/diversification);
- importance of search history;
- parameter tuning often critical.

Population based metaheuristics

Up to now we have just considered search algorithm that start from one state, evolve the state if we find improving solutions... basically we move from one state to another one (see also the graphs that we have seen).

Another huge sets of metaheuristics are instead based on *populations of states*. Instead of just considering one state that is moving to one state to another and another... I consider simultaneously many of them.

Population based metaheuristics:

- evolution of a set of points in the search space;
- some are inspired by natural processes, such as natural evolution and social insects foraging behavior;²⁷
- *basic principle*: learning correlations between ‘good’ solution components.

Genetic algorithms

Genetic algorithms are widely used in industry because they offer optimization tools based on algorithms that have their core in *Genetic algorithms* and they are very easy to implement in different problems and they do not require any modelling capabilities; as already said genetic algorithms are inspired by *evolution*. Three key components.

- *Adaptation*: organisms are suited to their habitats.
- *Inheritance*: offspring resemble their parents.
- *Natural selection*: new, adapted types of organisms emerge and those that fail to change adequately are subject to extinction.

So, how can we transfer these concepts into algorithms?

²⁷We will also see ahead the part related to *Swarm Intelligence* that really works by creating population of solutions and changing them by exploiting some natural metaphors.

Key concepts

We should consider that each individual represents a state and so has an evaluation function, so we can also have many evaluation functions, the ones that have the higher value of the evaluation function have more chances to stay alive...
Key concepts:

- the fittest individuals have a high chance of having a numerous offspring;
- the children are similar, but not equal, to the parents;
- the traits characterizing the fittest individuals spread across the population, generation by generation.

*GAs*²⁸ are not meant to simulate the biological evolutionary processes, but rather aimed at exploiting these key concepts for problem solving.

Genetic algorithm metaphor

We can use the following *metaphor*. Where:

BIOLOGICAL EVOLUTION		ARTIFICIAL SYSTEMS
Individual	↔	A possible solution
Fitness	↔	Quality
Environment	↔	Problem

Figure 3.9

- ‘A possible solution’ ↔ each individual represents a possible solution as already said;
- ‘Quality’ ↔ the evaluation function.

The evolutionary cycle

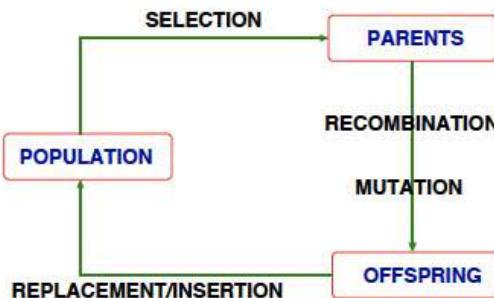


Figure 3.10

²⁸Genetic algorithms.

This is the cycle the genetic algorithm follows: we start with a *population*, we select the fittest individuals that becomes *parents*, we combine these fittest individuals through *recombination/mutation* finding *offspring* and then we have to insert it as a new *population* (we have also to consider that if we want to mantain costant the number of population we have to remove the old ones and insert the new ones).

Summarizing.

- *Recombination*: combines the genetic material of the parents.
- *Mutation*: introduce variability in the genotypes.
- *Selection*: acts in the choice of parents whose genetic material is then reproduced with variations.
- *Replacement/Insertion*: defines the new population from the new and the old one.

Terminology

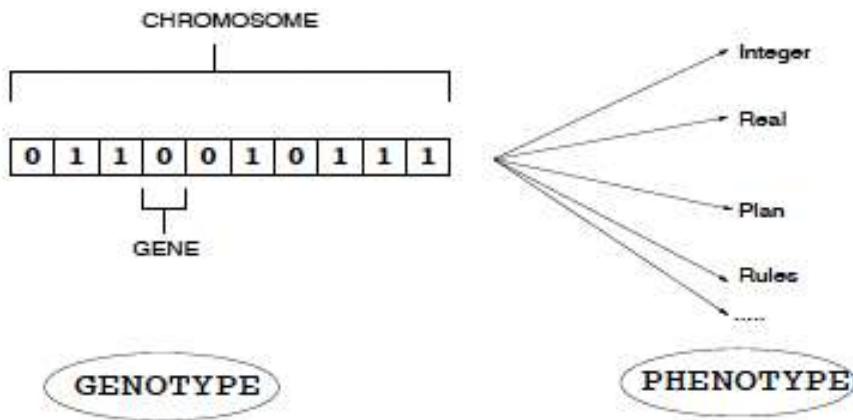


Figure 3.11

Some terminology:

- a *population* is the set of individuals (solutions);
- individuals are also called *genotypes*;
- *chromosomes* are made of units called *genes*;
- The domain of values of a gene is composed of *alleles* (e.g. binary variable \longleftrightarrow gene with two alleles).

Looking at the figure above note that, for example, the chromosomes can be either integer, real numbers or even plans or structures *etc.*

Genetic operators

Let's see the main genetic *operators*.

- *Recombination* or *Crossover*: cross-combination of two chromosomes (loosely resembling biological crossover).

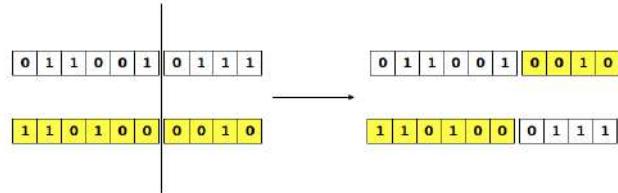


Figure 3.12

In the figure above we can see on the left column Parent A and Parent B; we select Parent A and Parent B because they are fit and we want to reproduce their component in the population. The middle arrow represents the *recombination operator*. We can act like this.

- *Mutation*: each gene has probability p_M of being modified ('flipped').

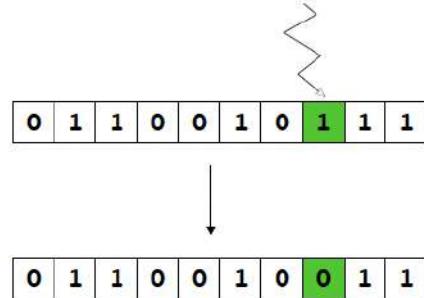


Figure 3.13

- *Proportional selection*: the probability for an individual to be chosen is proportional to its fitness. Usually represented as a roulette wheel.

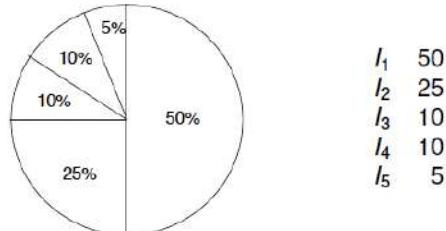


Figure 3.14

Depending on the level of their fitness, the first half of solutions with the highest evaluation function value has the 50% of being selected, then we have that the remaining 25% of solutions have the 25% of being selected... then we go with 10%, 10% and finally 5%.

- *Generational replacement*: the new generation replaces entirely the old one.
 - Advantage: very simple, computationally not expensive, easier theoretical analysis.
 - Disadvantage: it might be that good solutions are not maintained in the new population.

Alternatively, one can keep the best n individuals from the old and the new population (or choose any variant of this scheme).

Genetic operators: example

So we have seen for example *mutation* and *crossover* when we have bits and we saw that they are quite simple, let's see *mutation* and *crossover* in other cases like *real valued variables* and *permutation*.

- *Real valued variables*:
 - solution: $x \in [a, b]$ with $a, b \in \mathbb{R}$;
 - mutation: random perturbation $x \rightarrow x \pm \delta$, accepted if $x \pm \delta \in [a, b]$;
 - crossover: linear combination $z = \lambda_1 x + \lambda_2 y$, with λ_1, λ_2 such that $a \leq z \leq b$.
- *Permutation*:
 - solution: $x = (x_1, x_2, \dots, x_n)$ is a permutation of $(1, 2, \dots, n)$;
 - mutation: random exchange of two elements in the n -ple;
 - crossover: like 2-point crossover, but avoiding value repetition.

Genetic algorithm

High level algorithm:

```

Initialize Population
Evaluate Population
while Termination conditions not met do
    while New population not completed do
        Select two parents for mating
        Apply crossover
        Apply mutation to each new individual
    end while
    Population <- New population
    Evaluate Population
end while

```

Where:

- **Initialize Population** \longleftrightarrow by creating randomly (initial state is completely random) states in our space (note: this is already *diversification* because we are exploring different parts of our search space);
- **Evaluate Population** \longleftrightarrow assign the evaluation function to each of these individuals;
- **while Termination conditions not met do** \longleftrightarrow execution time limit reached or satisfactory solution(s) have been obtained or stagnation (limit case: the population converged to the same individual).

Genetic algorithm: summary

Intuition:

- crossover combines good parts from good solutions (but it might achieve the opposite effect);
- mutation introduces diversity;
- selection drives the population toward high fitness.

Pros:

- extremely simple;
- general purpose;
- tractable theoretical models.

Cons:

- coding is crucial;²⁹
- too simple genetic operators.

Example: 8 queens

Let's give this very simple example: this is the n -queens problem, so every time you put a queen in the chessboard you know that it attacks on the row and on the column and on the two diagonals; furthermore we cannot have 2 queens on the same row, column or diagonal at the same time.

Looking at the image below we can notice:

- focusing on the sequence of digits (3, 2, 4, 6, 5, 8, 7, 1) on the right we can say that the first small square is related to the first row (and the number 3 inside is indicating the third column), the second small square is related to the second row and so on;
- the entire (3, 2, 4, 6, 5, 8, 7, 1) is representing a solution meaning:
 - 3 \longrightarrow the first queen in the first row in column 3;
 - 2 \longrightarrow second queen in the second row is in column 2;

²⁹Meaning that for the parameters you're taking into account you have to decide: for example how to crossover/to split, decide the probability... are all things you have to decide.

- etc.;
- in general it represents a permutation because you need to have one queen per column, so every gene corresponds to a row; a permutation means that you cannot place 2 queens on the same column, so you are coding your constraints already (2 of them except the one in the diagonal) in your gene.
- crossover: combine two parents \longleftrightarrow how can we combine them? We want to minimize the number of queens that are *under attack*. We want to minimize the conflicts.

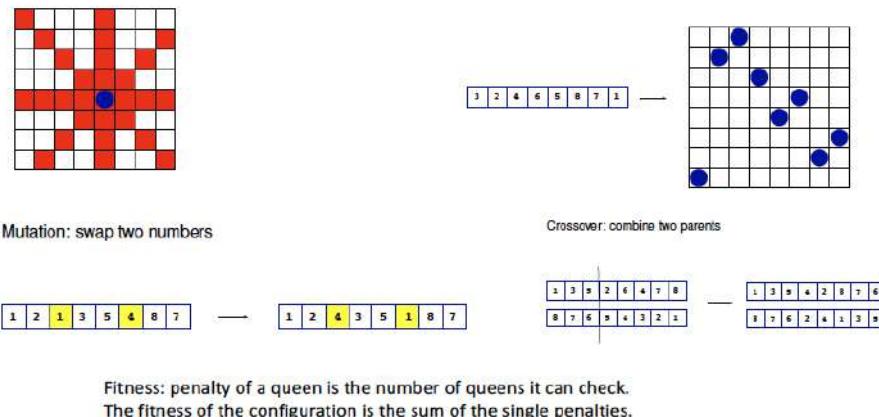


Figure 3.15

Design guidelines

Local search/metaheuristics approach preferable when:

- neighborhood structures create a correlated search graph;³⁰
- computational cost of moves is low;
- ‘inventing’ moves is easy.

Population-based approach preferable when:

- solutions can be encoded as composition of good building blocks;³¹
- computational cost of moves in local search is high;
- it is difficult to design effective neighborhood structures;³²
- coarse grained exploration is preferable (*e.g.* huge search spaces).

³⁰Meaning that the neighborhood structures induce a structure on the topology and so it is an efficient structure to explore.

³¹And these building blocks come from parents.

³²You want to avoid to create neighborhoods to explore or creating them is difficult.

Software frameworks

Comet (by *Dynadec*) www.dynadec.com

- Constraint-based metaheuristics framework.
- Proprietary programming language (free use for academics).
- Efficient data structures.

EASYLOCAL++ (by Luca Di Gaspero and Andrea Schaerf) tabu.diegm.uniud.it/EasyLocal++

- C++ stochastic local search framework (GPL license).
- User should instantiate abstract classes.
- Analysis tool also available (upon request).

Chapter 4

Swarm Intelligence

4.1 Characteristics and applications

So we can start this new topic, a topic very much aligned with the topic that we explained in the last chapter, namely '*population-based-methods*'; we have seen one example which is the one related to *genetic algorithms*.

This *population-based-method* is meant to create a set of solutions putting all together and evaluating each of them and then basically creating new solutions starting from the initial population by operators like *cross-over*, *mutations*, etc. The nice thing of this is that you don't have to model the problem but you simply have to encode in the *chromosomes*, in the different solutions, one possible assignment of your variable and this is not very difficult to do because till you have an *objective function/evaluation function* that is able to evaluate how good/bad is one solution then you've done; remain just to pull all this together in the population and repeating the process till the solution is reached.

These characteristics enables even people that are not very familiar with programming or optimization techniques that require to model the problem with constraints, variables and so on... to use *population-based-methods* that are more simpler because you have just to encode your solution in individuals of your populations and you've done. Of course even these algorithms don't work by magic, they require a lot of tuning of parameters that are many.

In this chapter we start with similar methods that go under the umbrella called *Swarm Intelligence*. Most of them are also *population-based-methods* where you do similar things seen in genetic algorithms: you have to represent a solution of your problem as a component of these swarm intelligence algorithms and then there are methods that either build constructively this solution or change this solution (like in *local search*); but in any case you have to just let the algorithms work.

Swarm Intelligence

Swarm Intelligence (SI) is an artificial intelligence technique based around the study of collective behavior in decentralized, self-organized systems.

SI systems are typically made up of a population of simple agents interacting locally with one another and with their environment. Although there is normally no centralized control structure dictating how individual agents should behave,

local interactions between such agents often lead to the emergence of global behavior. Examples of systems like these can be found in nature, including ant colonies, bird flocks, animal herds, bacteria molds and fish schools (from Wikipedia).

Mind is social

Some aspects.

- *Human Intelligence is the result of social interaction.*¹
- Evaluate, compare and imitate other individuals, learn from experience emulating the successful behavior of others, enable people to adapt to complex environments and situations through the discovery of optimal patterns, beliefs and behaviors (Kennedy & Eberhart, 2001).
- *Culture and cognition are consequences of human social choices.*
- Culture emerges when individuals become similar through a process of social learning.
- To model human intelligence there is need to model individuals in a social context.

Features of a SI system

An SI system has the following features:

- a set of simple individuals² with limited capacities;
- individuals are not aware of the system in its global view;
- local communication patterns (direct or indirect - stigmergic);³
- distributed computation: no centralized coordination of individual activities;⁴
- robustness;⁵
- adaptivity;⁶
- *natural metaphors*:⁷
 - ant colonies;

¹It is not an endogenous individual and isolated skill.

²A very simple set of moves for example.

³As we said in the previous page, intelligence is social so we can communicate. How agents can communicate? *Direct*: agent A tells something to agent B. *Indirect or Stigmergic*: for example if a professor before leaving a class write something on a blackboard and then some alumn goes to the blackboard and reads the text, this is an indirect communication; the professor changed the environment – modifying the blackboard – and from this the alumn took a message.

⁴Just any agent performing its own task locally.

⁵Being distributed makes the system robust, meaning that even if some of these agents fail, the overall swarm is able to achieve a given performance.

⁶These algorithms are able to adapt to changes in the environment.

⁷All these algorithms take inspiration from *natural metaphors*.

- bee colonies;
- fish shoals;
- bird flocks.

Self-organization

The main feature of *Swarm Intelligence* is what is called *self-organization*. Why self-organization? Because as we seen there is no centralized agent that coordinates the activity of the population of agents.

Ingredients.

- Multiple interactions among agents:
 - simple agents (*e.g.* rule based);
 - multi-agent systems.
- There are two extremely important ingredients that make the algorithms work, *feedback mechanisms*:
 - *positive feedback*:⁸
 - * reinforcement of common behaviours;
 - * amplification of random fluctuations and structure formation;
 - *negative feedback*:⁹
 - * saturation;¹⁰
 - * competition;¹¹
 - * exhaust resources.

Stigmergy



Figure 4.1

Stigmergy is a form of indirect communication.

An agent modifies the environment and others react to this change.

⁸It basically means: I have observed some ‘good’ behaviour from other individuals (high performing individuals) and I want to imitate, to boost, to keep it.

⁹Basically you want to avoid to imitate bad behaviour or failures.

¹⁰Saturated a resource for instance.

¹¹Compete for a single resource.

- Example: ants communicate through pheromone. Note in the left part of the figure above, for instance, the curved column they form together while walking, using the release of pheromone they change the environment in a way they attract ants to follow the same path.
- In the right part of the figure above we have a man over a giant nest of termites that built such structure without any central coordination.

Swarm Intelligence algorithms

Many algorithms exist based on the swarm intelligence concept; in this course we will see three of them.

- Ant Colony Optimization ACO (Dorigo, 1992).
 - Algorithm based on the behaviour of ants. Positive feedback based on pheromone trails. Positive feedback based on pheromone trails that reinforce components that contribute to the problem solution.¹²
- Artificial Bee Colony Algorithm (Karaboga, 2005).
 - Algorithm based on the behaviour of bees. Population of bees that look for nectar.¹³
- Particle Swarm Optimization PSO (Kennedy & Eberhart, 1995).
 - Algorithm based on the observation of bird flocks or fish schools. Stigmergy as communication and imitation of neighborhoods.¹⁴

Ant colony

From the observation of the ants we discover that:

- ants deposit pheromone trails while walking from the nest to the food and viceversa;
- ants tend to choose (more likely) the paths marked with higher pheromone concentrations;¹⁵
- cooperative interaction leads to the emergent behavior to find the shortest path.¹⁶

So how do we embedd these observations into the algorithm?

Ant behaviour

Some examples of ants behaviour.

¹²Here every agent is the same, has the same structure, the same type.

¹³Here, differently from the previous, we have different kind of agents that interact, different bees that have different functions and different tasks to do.

¹⁴Here we have again the same agents have the same structures and the same tasks; this is widely used in robotics.

¹⁵Note that the pheromone evaporates, so it means that if an ant has followed a path one hour ago there is no more pheromone, while if one second ago the pheromone is higher; and ants tend to choose – it is not deterministic – the path marked with higher pheromone concentration.

¹⁶Optimal solution.

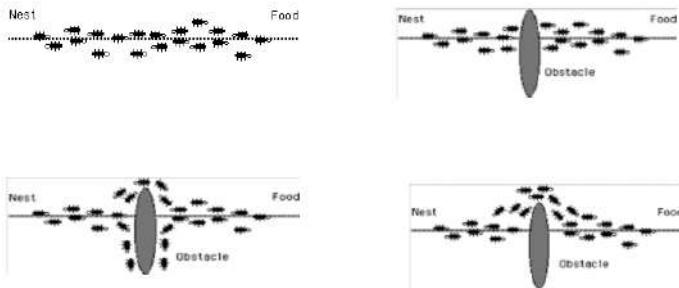


Figure 4.2

Looking at the figure above.

1. Figure on the top left: the ants start basically moving around randomly and at some point they converge to the shortest path from the nest to the food.
2. Figure on the top right: if you change something in the environment – putting for example an obstacle in the middle – we can see *adaptivity*.
3. Figure on the down left: at the beginning the ants move randomly splitting them on the left and on the right of the obstacle, note that we have the part at the top of the figure that is shortest to get the food in comparison to the down part.
4. Figure on the down right: after a while the ‘algorithm’ is running, ants realize the difference in the path length, and almost all ants will follow the shortest path.

Double bridge

Another very famous picture is the following.

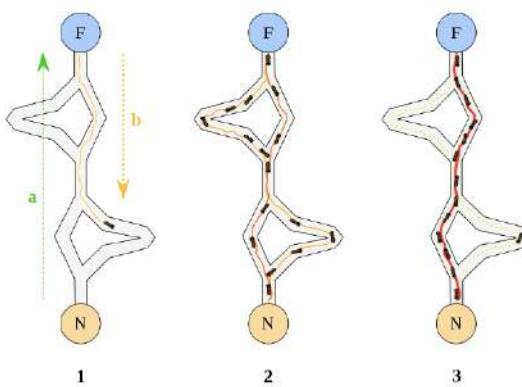


Figure 4.3

1. We have the yellow circle representing the *Nest* and the blue circle representing the *Food*; furthermore ants go back and forth (see the green and yellow arrows).
2. Initially ants split randomly in all the path, there is a sort of exploration of the environment.
3. But after a while the large majority of them will follow the shortest path.¹⁷ But observe that still there is an ant that is not following the optimal path,¹⁸ as we told before the behaviour is not deterministic and this is absolutely fundamental because you need to consider that these algorithms are probabilistic; if you create a *deterministic* version of ant colony algorithm it does not work: scientists tried to do that *i.e.* to remove the probabilistic part. Remember also that in *genetic algorithm* we had a probabilistic part: we give a probability that a given bit is flipped or a given part of chromosomes changes, we have a probability for doing that.

Ant colony optimization

Ant colony optimization has the following features.

- Probabilistic parametrized model – the pheromone model – used to model pheromone trails.¹⁹
- Ants build solution components in an incremental way.²⁰
- We create a solution through stochastic steps on a fully connected graph²¹ called construction graph: $G = (C, L)$.
 - Vertices C are solution components.²²
 - Arcs L are connections.²³
 - States are paths on G .
- Constraints can be represented to define what is a consistent solution.²⁴

¹⁷This is called *merging behaviour* and it is extraordinary because differing from some algorithms that we have seen, nobody tells the ant how to follow the shortest path, there are not imperative commands, you don't code anything. Each ant is just represented by a very basic algorithm, the behaviour emerges from the interaction of very stupid agents.

¹⁸It is necessary to continue exploring other alternative paths.

¹⁹*Probabilistic* is the key.

²⁰Each ant creates a single solution by a *constructive process*; remember when we talked about *tree search*, that was a constructive process: at each step you chose one assignment. Ants are doing the same, they just go, decide to go to a node and go there building a path.

²¹Note that could be important to apply mathematical *topology* concepts in order to optimize and discover new things.

²²So, for example, going to city A to city B is a part of the entire solution path; so cities are components of our solution. But of course we can map in a graph also other kinds of problem *e.g.* in the *8 queens problem* every step is placing one queen and there is a node in a graph because one queen can be placed either in position 1 or 2 or 3 *etc.* So if you find a mapping between your solution components and the vertices of this graph then you can solve the problem through *ant colony algorithm*.

²³And these arcs basically represent possible moves because you can go from one solution component to the other through a connection.

²⁴I can basically structure our solution components in an intrinsic or explicit way that avoids to have inconsistent solutions or I can enable the system to violate constraints but then

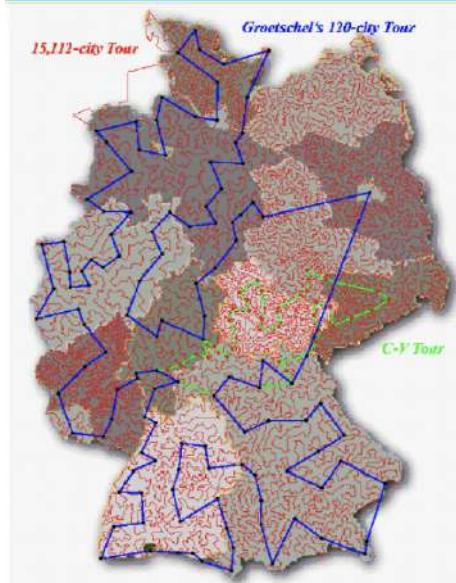
Example: TSP

Figure 4.4

We have already talked about this problem which is very important:

- you have a set of cities and you have to find a minimum cost hamiltonian tour covering all of them (once and only once);
- note that in the figure we have basically three colored graph:
 - the green one is the TSP instance covering the smaller group of cities;
 - the blue one covers 120 cities;
 - the red one 15.112 cities.

So how can we model this TSP in ACO (ants colony...)? It is very easy being that we already had the graph, the graph is our problem so there is a 1-1 mapping.
A TSP model in ACO:

- nodes of G (solution components) are cities to be visited;
- arcs are connections between cities;
- a solution is an Hamiltonian path in the graph;
- constraints are used to avoid sub-cycles: each ant can visit a city once.

you have to pay a cost for violating constraints. So, for example, remember when we talked about the *4 queens problem* metaheuristic: at the beginning we put a random assignment of queens in a way that permits 2 queens to attack each other; this is a constraint violation because our constraint was not 2 queens attack each other; every queen should be safe; this was a constraint of the problem. So if you violate a constraint you have to pay a cost. You can admit this solution in the path of finding a good solution but you have to pay a cost: then you move one obtaining a lower cost; move another one obtaining another lower cost till reaching the zero cost that means you have satisfied all the constraints finding a feasible solution.

Information sources

Let's try to go inside one ant and try to understand how this very basic simple agent behaves. Artificial ants rely on two informations while moving: *the pheromone* that is an information passed from other ants (how much pheromone I find in the environment – and this is basically related to the communication/interaction with other agents to stigmergy); the second is an *heuristic value* that starts from the following consideration: why don't we provide the ants an idea of what is the problem they are solving, not the overall goal, but still an idea of what path to choose.²⁵

Specifically.

- Connections, solution components or both have the following associated information:
 - pheromone τ ;
 - heuristic value η .
- The pheromone value abstracts natural pheromone trails and codes the long term memory²⁶ of the global search process.
- The heuristic value represents the prior background knowledge on the problem.²⁷

Ant system

How does an ant choose the next move? We said that each ant starts from a starting node (the NEST) and wants to reach the FOOD with the minimum path. So how each ant decides to go?

- Memory is used to remind past paths.
- Starting from node i , we have to probabilistically choose the next consistent node to visit.
- The probabilistic choice depends on:

²⁵For example, if I have to find the shortest path from the nest to the food and I have many alternative paths around me what can I do? Would you choose the longest arc or the shortest arc? Generally I would choose the shortest because I try to move with small steps towards the goal and this is an heuristic value of your problem. Another example: if you try to take a set of objects solving the *knapsack problem* (it is a problem where you have a given capacity to put your objects in your knapsack, and you have to minimize the cost respecting the capacity), so it means that you have to select the minimum cost set of objects filling your knapsack, so if you want a minimum cost assignment what would you choose? Which object would you choose first? The object that cost much or that cost less? If you only consider the cost of course you would choose that with the lower cost, or, better, you can order these objects with an indicator that consider cost and weight, so if I have a better relation between the cost and the weight I have a better indicator and I choose it. So these are examples of heuristic values we can give. In our specific case, the case of ants, we can give an heuristic value that gives an info on each ant on which is the best move (locally of course) related to the problem that I have to solve.

²⁶Why the long term memory? Because the pheromone stays for a while and only after some time evaporates. But the intensity of this value reminds you the quantity of ants that followed that path.

²⁷Background knowledge because it provides you interesting informations on which is the best next move.

- pheromone trail τ_{ij} ;
- heuristics $\eta_{ij} = 1/d_{ij}$.

And it is summarized in the following relation:

$$p_{ij} = \begin{cases} \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{k \text{ feasible}} \tau_k^\alpha \eta_k^\beta} & \text{if } j \text{ consistent} \\ 0 & \text{otherwise} \end{cases} \quad (4.1.1)$$

Where:

- α and β are parameters that weight the importance of the pheromone and the heuristics;
- the denominator – in the j consistent case – represents just a normalization factor to make the sum of the overall probability in the end being equal to 1;
- in the denominator – in the j consistent case – we put k as index because the k feasible indexes are a subset of all j ;
- in the other case the probability is 0 because if the path ij is not consistent/feasible then simply the probability is 0. In practical terms consistent means: suppose that there is not a direct path from city i to city $j = 2$ because there is an obstacle, then I can't go directly there, so in that case we have $p_{ij} = 0$.

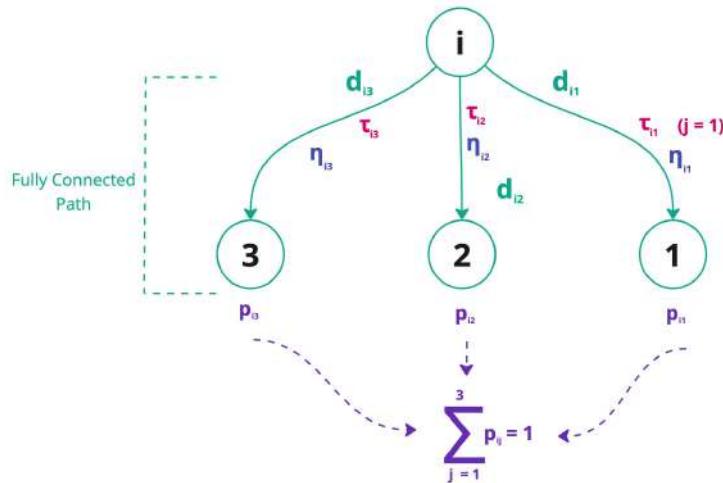


Figure 4.5

Looking at the figure above: the ant is in the node i and has to choose one from the node $j = 1$ or $j = 2$ or $j = 3$. On each of the 3 arcs starting from i I have two informations:

- the pheromone τ_{ij} provided by other ants;
- the heuristic η_{ij} which is in this case the inverse of the distance d_{ij} between i and j \longleftrightarrow the longer the distance, the smaller the probability of going there.

Each choice has a probability p_{ij} and the sum $\sum_{j=1}^n p_{ij}$ has to be 1. So how I do compute this probability? As indicated in the (4.1.1).

Every time an ant has created a path you have to update the pheromone, because we want to simulate an ant that walking releases pheromone. How do we update/create it?

The pheromone is updated with the following rule:

- $\tau_{ij} \leftarrow (1 - \varrho)\tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k$ where ϱ is the *evaporation coefficient*,²⁸
- where $\Delta^k\tau_{ij}$ is characterized as follow:

$$\Delta^k\tau_{ij} = \begin{cases} \frac{1}{L_k} & \text{if ant } k \text{ used arc } (i,j) \\ 0 & \text{otherwise} \end{cases} \quad (4.1.2)$$

- L_k ²⁹ represents lenght of the path³⁰ followed by ant k .

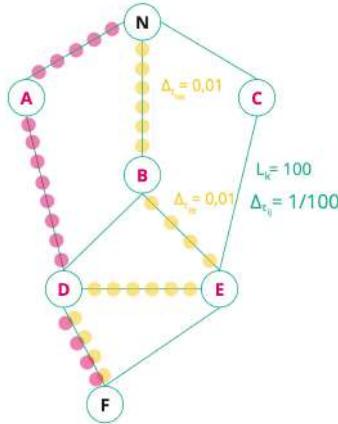


Figure 4.6

²⁸In the $\tau_{ij} \leftarrow (1 - \varrho)\tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k$ formula τ_{ij} at the left of the arrow represents the new value, m represents all the ants, k is not an exponent but represent only an index and ϱ is another parameter to tune. In general we can read it with the following lens: the pheromone on a given arc is created by taking into account the one that was already there, the fact that we have to consider that has a little evaporated so why $(1 - \varrho)\tau_{ij}$ but then you sum the new contributions of the ants that have chosen that arc in their path i.e. $\sum_{k=1}^m \Delta\tau_{ij}^k \leftrightarrow |m|$ represents the cardinality of all ants and you consider the data related to pheromone that has been released by ant k to travers path (i, j) , basically we consider also the effort – a sort of amplifier factor – of all ants that have used this path.

²⁹In this specific problem we want – as said in general – to enforce good solutions and either avoid or reinforce less bad solutions. Here we want to find the shortest path, so every time an ant has completed the path (from the NEST to the FOOD) you know the lenght of the particular path taken (for example seeing the following image the red path has lenght 70 while the yellow 100; so the red one which is shorter is better) leaving equal pheromone to each arch for all ants (that is the thing that is done in nature), you can also optimize this aspect saying: I want to put more pheromone on an arc if it is part of a shorter path, so why we define $\Delta\tau_{ij}$ as $1/L_k$. The longer the path \Rightarrow the worse the solution is \Rightarrow the less the pheromone released.

³⁰Overall path of ant k , so you update at the end: you leave all ants work, then when they find the solutions, only at that time, you stop, and then update the pheromone trail.

Consider the above graph which has a node for the nest N and one for the food F . We start sending many ants along the graph, each ant at each step build its own solution: for example ant A follows the red highlighted path while ant B the yellow. At the end of the process, when all ants have created their own path I will consider each arc separately and I reason in this way: how many ants pass in arc $N \rightarrow A$? 1;³¹ how many ants pass in arc $B \rightarrow E$? 1; how many ants pass in arc $D \rightarrow F$? 2...³²

Basic schema

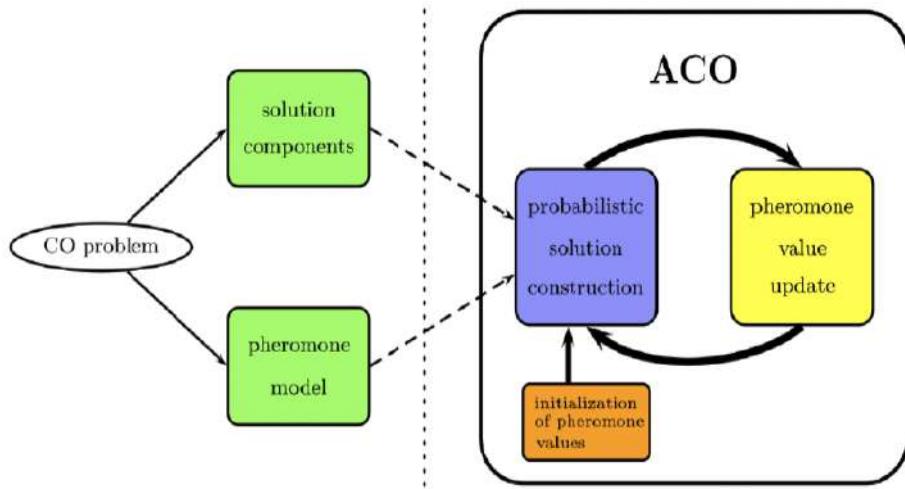


Figure 4.7

Let's see the basic schema.

- *CO problem* \longleftrightarrow let's have a combinatorial problem like the *knapsack problem*, the *8 queens problem*, the *TSP problem*... any problem where you want to create a solution, for example, with a minimum cost.
- *Solution components \wedge pheromone model* \longleftrightarrow you have to create 2 things. *One* you have to understand what are the solution components (in the case of TSP, for example, arcs are solution components from city A to city B; in the case of knapsack is to put one object into the knapsack; in the case of 8 queens problem is assembling queens in the cells; solution components depend from the problem, this is a modelling part and depends on your capabilities of modelling the problem; the modelling part is not very complicated, you need just to understand a representation encoding your solution. *Second* the pheromone model, you have to decide how evaporates the pheromone, how to release the pheromone, how every ant will choose on the basis of pheromone and/or heuristic.

³¹In this case I update the pheromone only for that ant.

³²We have 2 of them, so I take the pheromone that was already there, I apply the evaporation ρ and plus I sum 2 pheromone traces of the two ants that have chosen that path. So the pheromone here is larger in this arc than in the arch with only 1 ant.

- *ACO* \longleftrightarrow then, given those 2 parts regarding the modelling of our problem, we can start our algorithm.
- *Probabilistic solution construction* \longleftrightarrow you construct probabilistic solution, so each ant probabilistically chooses a part, and then another part and so on till the solution.
- *Pheromone value update* \longleftrightarrow then once reached the solution, the ‘*food*’, you update the pheromone value and you go back.
- *Initialization of pheromone values* \longleftrightarrow the initialization of pheromone values can be done either with 0 values of arcs simulating that each ant does not know at all where to go or with random initial values (using initially small values because if you instead would use big values you are going to conditionate your problem).

ACO system

First example of Ant Colony Optimization:

- ants build a solution following a path on the construction graph;³³
- a transition rule is followed to choose the next node to visit;
- the heuristics and the pheromone are used;
- pheromone values are updated on the basis of the *quality of the solution found* by the ants.

Ant-system algorithm

High level algorithm.

```

InitializePheromoneValues()
while termination conditions not met do
    for all ants a in A do
        sa <- ConstructSolution(tau, eta)
    end for
ApplyOnlineDelayedPheromoneUpdate()
end while

```

Where.

- **while** termination conditions **not** met **do** \longleftrightarrow remind you something? We used this kind of **while** in all meta-heuristic; you have to decide termination conditions because you are not guaranteed to find an *optimal solution*. A termination condition can be for instance a timeout, certain number of iterations and so on.
- **sa** \leftarrow **ConstructSolution**(tau, eta) \longleftrightarrow I have a constructed solution for all arcs. Note that the **tau** and **eta** are referring respectively to τ and η .

³³If you have 100 ants at the end of a run you will have 100 solutions (one per ant) and then you will restart updating the pheromone for every path considering the 100 solutions that you have.

- `ApplyOnlineDelayedPheromoneUpdate()` \longleftrightarrow then using these solutions I apply `DelayedPheromoneUpdate`. I update at the end of the solution construction. First I have to put, then I update the pheromone.³⁴

Algorithm

The following is the detailed algorithm; see the description of the functions mentioned ahead:

```
while termination conditions not met do
    ScheduleActivities
        AntBasedSolutionConstruction()
        PheromoneUpdate()
        DaemonActions() # optional
    end ScheduleActivities
end while
```

Where:

- `AntBasedSolutionConstruction()`
 - Ants move by applying a stochastic³⁵ local decision policy that uses values of pheromone and heuristic on graph components.
 - While moving, ants take track of the partial solutions (paths) that have been built.³⁶
- `PheromoneUpdate()`³⁷
 - Ants update the pheromone during the solution construction (online step-by-step pheromone update).³⁸
 - Ants can update backward the pheromone on components used on the basis of the quality of the overall solution (online delayed pheromone update).
 - Evaporation is applied all time.

Of course we are not perfectly replicating the natural behaviour of ants, we are cheating in order to make the algorithm work: of course in the delayed update we are cheating – in nature there is a real time update – in order to make the update considering a global factor (all the solutions performed by all ants).

Another way to cheat in the name of optimization is the following:

- `DaemonActions()`

³⁴Another valid alternative is that I update the pheromone while the ants are walking: each ant does a move and immediately the pheromone is updated but in this case you don't have informations on how long is the path to each of these arcs belongs.

³⁵Note that it is not a deterministic solution, but probabilistic.

³⁶They ‘remember’, they need to remember the path that have followed. Why? Because at the end of the process you have to update the pheromone.

³⁷There are two kinds of update, we can have: *step by step* vs *delayed overall update*.

³⁸For example can be done considering the lenght L_k walked so far or the lenght to reach a specific node instead of considering the lenght of the entire path; but the logic can be mantained.

- Are ‘centralised’ actions that cannot be executed by the single ants:³⁹
 - * local search procedure⁴⁰ applied to each solution built;
 - * collection of global information to decide whether to leave additional pheromone to guide search from a global perspective.

ACO system

Papers on the course web-site.

- MAX-MIN Ant System.
- Hyper-cube Framework.
- Multi-level ACO.
- Beam ACO.

EU Project: SWARM-BOT, SWARMANOID:

- <http://www.swarm-bots.org/>
- video on youtube: <http://www.youtube.com/watch?v=seGqy032pv4m/watch?v=3YDkbltzMmA>

Honey Bee colony

As already said ant colony algorithm is relying on the fact that the agents are all the same, every ant might have different parameters α and β but basically the behaviour is the same (what we have explained).

Artificial Bee Colony algorithm – which is not widely used, as we told before – instead has agents that have *different tasks*, different capabilities.

We refer to *Artificial Bee Colony algorithm* as *ABC algorithm*.

Let's see several aspects.

- Artificial bees of three types:
 - *employed bees* that are associated with a specific nectar source;⁴¹
 - *onlookers* that observing the employed bees choose a nectar source;
 - *scouts* that discover new food sources.
- Initially, food sources are discovered by scout bees. Then food is consumed and the source exhausted (by other bees). The employed bees (that have consumed the food) in that source become scout.
- Food position: solution (we have as many solutions as employed bees).
- Food quantity: fitness.

³⁹In general, because we need to maintain the philosophy proper of *swarm intelligence* these ‘centralised’ actions are not actions that ‘control’ ants or ‘impose’ specific moves to ants.

⁴⁰We have seen them; we can merge their techniques to find local optima, for example, and use the information in the update to improve the algorithm. So improves the solutions in a local optima.

⁴¹We will see ahead what we mean with ‘nectar source’.

Note: there is an huge difference between ACO and ABC. In ACO each ant builds a solution step by step; in ABC instead we don't build any solution step by step, but this is really a metaheuristic algorithm because you have already solutions – an assignment of values to variables – this is considered a *food source*, how much food there is in this source depends on the evaluation of that solution.

For example, just an outline.

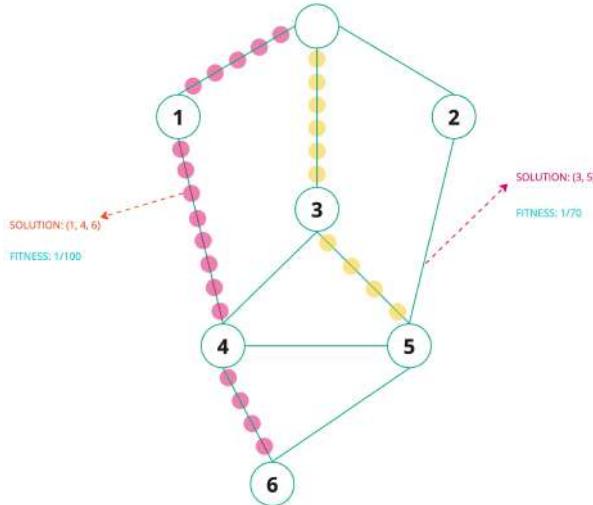


Figure 4.8

In the image above we have 2 solutions (representing 2 food sources, 2 points in my search space); one which is highlighted in red and the other in yellow.

The first one is associated with a nectar source that is $1/100$ and the other with $1/70$ (so, of course the second represent a larger source of food, so better).

So every employed bee has a solution, so there is a starting overall solution (initial set of solutions),⁴² there is no building of solutions.

ABC algorithm

Let's see this very simple ABC algorithm:

```

InitializationPhase()
repeat
    EmployedBeePhase()
    OnlookerBeePhase()
    ScoutBeePhase()
    Sol=BestSolutionSoFar
Until (Cycle=MaxCycleNum or MaxCPUtime)

```

Where.

- `EmployedBeePhase()` \longleftrightarrow bees are associated with nectar source; then they explore around in the neighborhood of the solution (*Intensification*).

⁴²Represented by a full assignment of variables: bee, position of the food, quantity of the food (fitness).

- **OnlookerBeePhase()** \longleftrightarrow onlooker bees basically decide to go on a nectar source on the basis of its fitness value; probabilistic decision. Onlookers are flying trying to decide which of already discovered solution has to be reached. The result of this is that more promising solutions, the ones that have more food because they have higher fitness, since they are promising they are explored by a larger number of bees (probabilistically) and therefore you can explore a larger neighborhood of a promising solution because you have many bees that are all there.
- **ScoutBeePhase()** \longleftrightarrow scouts discover new solutions; they are *diversifying* our process. Remember: *Intensification* and *Diversification* are two important features of metaheuristic.

More specifically:

- **InitializationPhase()**

- A set of food source positions⁴³ are randomly selected by the bees and their nectar amounts are determined.
- Each solution X_m ($m = 1, \dots, N_{pop}$) is composed of n variables X_{mi} ($i = 1, \dots, n$). Each variable is subject to a lower and upper bound and initialized to

$$X_{mi} = lb_i + rand(0, 1)(ub_i - lb_i) \quad (4.1.3)$$

For example, a random initial phase could be: 4 bees/4 solutions, 3 variables for each solution, where:

- * variables $\longleftrightarrow x_1, x_2, x_3 \in [0, 10]$ with 0 lower bound, 10 upper bound;
- * population:
 - Bee 1 $\longleftrightarrow (5, 4, 7) \longleftrightarrow (x_1, x_2, x_3)$;
 - Bee 2 $\longleftrightarrow (7, 1, 3)$;
 - Bee 3 $\longleftrightarrow (4, 5, 2)$;
 - Bee 4 $\longleftrightarrow (1, 8, 7)$.

Suppose now we have a function, the evaluation function of the solution (*obj function*). Suppose we are trying to maximize. Each bee shares the info about the nectar amount:

- population:
 - Bee 1 $\longleftrightarrow (5, 4, 7) \longleftrightarrow^{44} 100$;
 - Bee 2 $\longleftrightarrow (7, 1, 3) \longleftrightarrow 70$;
 - Bee 3 $\longleftrightarrow (4, 5, 2) \longleftrightarrow^{45} 101$;
 - Bee 4 $\longleftrightarrow (1, 8, 7) \longleftrightarrow^{46} 60$.

- **EmployedBeePhase()**

⁴³To each of the bees you provide 1 solution.

⁴⁴Nectar amount: quantity of food associated to a point in the solution landscape.

⁴⁵If we are maximizing, this is the best

⁴⁶They should be the values output of the fitness function.

- After sharing the information about the nectar amount, every employed bee goes to the food source area visited by herself at the previous cycle since that food source exists in her memory, and then chooses a new food source by means of visual information in the *neighborhood* of the present one.⁴⁷
- Fitness function of X_m : we use the objective function of the problem to compute the fitness as follows:⁴⁸

$$ftn(X_m) = \begin{cases} \frac{1}{1+obj(X_m)} & \text{if } obj(X_m) \geq 0 \\ 1 + |obj(X_m)| & \text{if } obj(X_m) < 0 \end{cases} \quad (4.1.4)$$

- **OnlookerBeePhase()**⁴⁹

- An onlooker bee chooses a food source depending on the probability value associated with that food source:⁵⁰

$$p_m = \frac{ftn(X_m)}{\sum_{i=1}^{N_{pop}} ftn(X_i)} \quad (4.1.5)$$

Positive feedback.

- **ScoutBeePhase()**

- Scouts choose nectar sources randomly.
- The employed bees that cannot improve the solution after a given number of attempts become scouts and abandon the food source.
- Negative feedback.

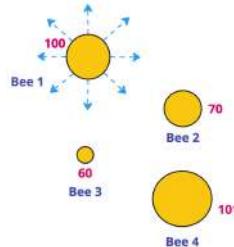


Figure 4.9

Look at the figure above.

⁴⁷The exploration of the neighborhood is done through *Local Search*.

⁴⁸Or viceversa depending if we are maximizing/minimizing.

⁴⁹Refer always to the example of the 4 bees above. So then we have other bees that observe the four solution we have already in the population and decide to help probabilistically. So assigned the 4 solutions we have a probability to be reached by another bee. The higher the solution (fitness of the solution), the better the solution, the better the probability that other onlooker bees reach it. So an onlooker bee chooses its food source – among the already explored food sources, no new ones – depending on the probability. So referring to the example the solution with nectar amount 101 has the greatest probability to be reached while that with 60 the lowest.

⁵⁰In the formula below the denominator represents a normalization factor because the sum of all probabilities should be 1.

The larger the dots the higher the food. So consider the following situation: Bee 1 is exploring around the neighborhood of the solution because is more likely to find promising solutions around other promising solutions (*Intensification*). The onlookers come following the probability to help the exploration of the neighborhood, to help intensification around promising zones.

So employed bees that cannot improve the solution become scouts in the sense that say: I'm not interested any more to the source in which I am, I want to find new ones! They generate solution randomly in the same way we have done at the beginning.

So Bee 1 as employed bee explores the neighborhood associated to 100 nectar ($\leftrightarrow (5, 4, 7)$) —> then becomes scout and generates/chooses nectar source randomly like in the `InitializationPhase()`: (8, 3, 7) with 80 nectar —> then becomes an employed bee again and start exploring the neighborhood and so on... so we have pure scout bees plus scout bees that are employed bees that cannot improve or have exhausted the solution.

ABC applications

As it can be seen the ABC applications are quite limited.

- Training neural networks: optimization function that represents the mean squared error.
 - Karaboga D., Basturk B., Ozturk C., (2007) *Artificial bee colony (ABC) optimization algorithm for training feed-forward neural network*, *Modeling Decisions for Artificial Intelligence, LNCS 4617*. 318-319.
- Numerical Optimization.
 - http://chern.ie.nthu.edu.tw/gen/4a-Karaboga-tr06_2005-original.pdf
- Combinatorial Optimization.
 - Karaboga D., Basturk B., (2007) *Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems*, *Advances in Soft Computing, LNCS 4529*. 789-798.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) has been proposed in 1995 by:

- a psycho-social scientist James Kennedy;
- an electrical engineer Russel Eberart.⁵¹

The research activity starts from the analysis of interaction mechanisms between individuals that compose the swarm.

Particularly interesting when the whole swarm has a common goal such as food search.

The observation of rules that guide the bird flock moves shows that each individual entity has three driving trends:

⁵¹They studied a lot how intelligence is relying on social behaviour.

- follow neighbours;
- stay in the flock;
- avoid collisions.

With these rules it is possible to describe and model the collective moves of a flock with *no common objective*.

PSO adds a common objective: food search.

With a common objective, a single individual that finds a food source has two alternatives:

- move away from the group to reach the food (individualistic choice);
- stay in the group (social choice).

If more than one individual entity moves toward the food other flock members do the same.

Gradually the whole group changes direction toward promising areas. The information propagates to all members.

With PSO we can solve optimization problems⁵² with the following analogy:

- individuals: tentative configurations that move and sample the solution in a N-dimensional space;⁵³
- social interaction: each individual agent takes advantage from other searches moving toward promising regions (best solution globally found).⁵⁴

Search strategy can be found as a balance between exploration and exploitation:

- exploration: individual agents that search for a solution;
- exploitation: social behaviour which is the exploitation of other individuals successful behaviour.⁵⁵

PSO neighborhood

The concept of neighborhood is slightly different from what seen in the other algorithms. In the context of this algorithm a neighborhood is the set of individuals that are affected by a single individual, so a single individual is connected to a neighborhood, and all these individuals belonging to its neighborhood are affected by the movement or action of a single individual. So something like we're creating a structure that links together these individuals by creating sub-groups and the intersection between different sub-groups is not empty, so in this way the information can propagate among the all swarm. So the *neighborhood* tells me who I am connected with: simply which are the individuals that are influenced by the action of a single individual.

More specifically.

⁵²In the same fashion of ACO and BCO.

⁵³Each individual represent an N-dimensional point in the solution space. For example in Bee Colony we start with a tentative configuration.

⁵⁴Each individual knows which is the best solution the individual has found so far and the best solution the entire swarm has found so far; this last global solution is the unique global information used by this algorithm.

⁵⁵I want to imitate a succesfull behaviour.

- A feature that appears to be important is the concept of proximity:
 - individuals are affected by the actions of other individuals that are closer to them (sub-groups);
 - individuals are part of more sub-groups, and then the spread of information is globally guaranteed.
- Sub-groups are not tied to the physical proximity of the configurations in the parameter space but are a priori defined⁵⁶ and may take into account also considerable shifts between individuals.

PSO algorithm

General characteristics.

- PSO optimizes a problem by setting a population (swarm) of candidate solutions (particles).
- PSO moves these particles in the search space through simple mathematical formulas.⁵⁷
- The movement of particles is guided by the best position found so far in search space (from individual to population)⁵⁸ and it is updated when better solutions are discovered.
- $f: \mathbb{R}^n \rightarrow \mathbb{R}$ fitness or cost function to minimize.⁵⁹ it takes a solution (vector) and produces a fitness value.⁶⁰ The gradient of f is not known.⁶¹
- Goal: find solution a such that $f(a) \leq f(b)$ for all b in the search space.⁶²

Specific characteristics.

- S number of particles in population.
- Each particle has:
 - a position $x_i \in \mathbb{R}^n$ in the search space;
 - a speed $v_i \in \mathbb{R}^n$.⁶³
- p_i is the best solution found so far by particle i .

⁵⁶So, for example, when I start the algorithm I decide that all individuals are connected, or that 1 individual is connected to other 10 individuals and so on... is a structure a priori decided.

⁵⁷Exploring around through mathematical formulas.

⁵⁸In the sense that the movement of each particle is guided by two important informations: one is the best position found so far in the search space from the individual itself; the second is the best solution found so far by the overall population. And these two values are updated everytime a solution better than the previous is discovered.

⁵⁹Of course we need to have a cost function in order to minimize or maximize depending on our intention, if we are solving an optimization or minimization problem.

⁶⁰How good or how bad is this solution.

⁶¹Meaning that this algorithm really apply not only when you don't have any model, but even when the f is not differentiable; we need just a value associated to each configuration of values.

⁶²If we are minimizing.

⁶³Direction and intensity of the movement.

- g is the best solution found so far by the entire swarm.
- For each particle $i = 1, \dots, S$ do:⁶⁴
 - initialize the particle's position with a uniformly distributed random vector: $x_i \sim U(b_{lo}, b_{up})$ where b_{lo} , b_{up} are the lower and upper boundaries of the search-space;⁶⁵
 - initialize the particle's best known position to its initial position: $p_i \leftarrow x_i$;
 - if $f(p_i) < f(g)$ update the swarm's best known position: $g \leftarrow p_i$;
 - initialize the particle's velocity: $v_i \sim U(-|b_{up} - b_{lo}|, |b_{up} - b_{lo}|)$.⁶⁶
- Until a termination criterion is met⁶⁷ (e.g. number of iterations performed, or adequate fitness reached), repeat:
 - for each particle $i = 1, \dots, S$ do:
 - * pick random numbers: $r_p, r_g \sim U(0, 1)$;
 - * update the particle's velocity:⁶⁸

$$v_i \leftarrow \omega v_i + \varphi_p r_p (p_i - x_i) + \varphi_g r_g (g - x_i);$$
 - * update the particle's position:⁶⁹ $x_i \leftarrow x_i + v_i$;
 - * if $f(x_i) < f(p_i)$ do:
 - update the particle's best known position: $p_i \leftarrow x_i$;
 - if $f(p_i) < f(g)$ update⁷⁰ the swarm's best known position: $g \leftarrow p_i$.
 - Return g : best found solution.
 - Parameters $\omega, \varphi_p, \varphi_g$ should be carefully selected as they strongly influence the effectiveness and the efficiency of the PSO method.⁷¹

Parameter tuning

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

Tedious and error prone operation. It is hard to find the optimal parameter configuration.

⁶⁴Algorithm initialization.

⁶⁵Very similar to what seen with Bee Colony.

⁶⁶These two values represent kind of a *vector*.

⁶⁷Again we just remember that all *swarm* algorithm are based on ‘termination conditions’ because we have not any guarantee that these algorithms will find the *optimal solution*. We iterate and at some point we need to stop.

⁶⁸The velocity is updated considering a factor proportional to the previous velocity ωv_i plus a contribution that depends on something proportional to the difference between the best individual solution and the current position (we are evaluating how good/bad is the current position in comparison to the best) $\varphi_p r_p (p_i - x_i)$, plus a contribution that is proportional to the difference between the best global solution and the current position $\varphi_g r_g (g - x_i)$. The values $\omega, \varphi_p, \varphi_g$ are constant.

⁶⁹We are ‘moving’.

⁷⁰It is intelligent to do this evaluation only if the first one *i.e.* $f(x_i) < f(p_i)$ passes.

⁷¹Note that each particle could have its own parameters, so there is the possibility to have to tune a large number of them.

Techniques for automatic parameter tuning:⁷²

- ParamILS⁷³ <http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/>
- Available on line.

Robot based on PSO

Article in which is introduced PSO:

- Kennedy, J.; Eberhart, R. (1995). ‘Particle Swarm Optimization’. Proceedings of the IEEE International Conference on Neural Networks. IV. pp. 1942-1948;
- <http://www.engr.iupui.edu/~shi/Coference/psopap4.html>

Movie robots:

- <http://www.youtube.com/watch?v=RLIA1EKfSys>

Source in motion:

- <http://www.youtube.com/watch?v=nul8nYIQ8ug&feature=related>

Swarm Intelligence videos

We conclude this chapter proposing some usefull videos that show some major implementation of swarm intelligence:

- <https://www.youtube.com/watch?v=seGqy032pv4> (Swarm-bots: Self-assembly and Cooperative Transport)⁷⁴
 - In this video there is a huge heavy object highlighted in red that has to be moved forward among 6 small blue highlighted robots. Perceiving the red light these robots link to the heavy object (or to other small robots that have become red) becoming red themselves and try to push all the time the object forward. But only when there are all 6 robots pushing they are able to move the heavy object. They are collaborating to achieve a given goal. The idea is that alone a single robot is not able to do anything but together they can cooperate to accomplish and get the goal, even if there is not centralized controll.⁷⁵

⁷²We have essentially seen that all *swarm algorithms* require parameters tuning: imagine you have for a given problem like 5000 particles and you want to tune every parameters, 3 parameters for each particle, which implies 15000 parameters to tune. This clearly is not an operation that can be done manually. It is done through machine learning or metaheuristic, we have different parameter tuning techniques applied to the parameter space; in this space, for example, is evaluated convergence of the parameters or other mathematical value, or a certain configuration is evaluated changing the type of space through certain transformations etc. In general we can have either parameter tuning algorithm for specific kind of solvers or general purpose algorithm like ParamILS below that being very general can be applied to any algorithm.

⁷³Iterated Local Search.

⁷⁴Between parenthesis we insert, when possible, key-words that can be useful to find the video in case the url will change; put them in the default search bar of the server indicated into the url (in this case www.youtube.com).

⁷⁵Ant colony concepts.

- <https://www.youtube.com/watch?v=M2nn1X9Xlps> (Swarmanoid, the movie)
 - In this project called *Swarmanoid* you have different kinds of robots each of them able to perform a specific task; they are not controlled by any central unit but they coordinate with each other in order to obtain a given task.⁷⁶
- <https://www.youtube.com/watch?v=RLIA1EKfSys> (Robot Swarm driven by Particle Swarm Optimization algorithm)
 - We have multiple virtual small robots, different simulated 2D scenarios, and they want to find a small blue circle which represents the food. On the right of the screen you can see red dots indicating position and white arrows indicating the velocity vector, and red circle surrounding the dots representing the neighborhood (in order to propagate informations). Initially, they explore with a random configuration.⁷⁷
- <https://www.youtube.com/watch?v=nul8nYIQ8ug> (Particle swarm optimization: moving food source)
 - The same of the previous conceptually but now the food is moving; also with the food moving the algorithm works.⁷⁸

⁷⁶Bee Colony concepts.

⁷⁷Particle Swarm concepts.

⁷⁸Particle Swarm concepts.

Chapter 5

Games

5.1 Games with opponents as search

We are going to conclude the part related to *search* with the following argument. Remember that we have basically seen:

- tree search;
- metaheuristic;
- population based methods:
 - among population we have seen *Swarm Intelligence algorithms*.

Now we are going to change a little bit the scope going in the field of *Games*.

Games with opponents as search

Now we are going to explore a little games with opponents as search; let us to consider some aspects:

- multi-agent environment that has to account for the presence of an opponent;
- game theory which is a branch of the economy;
- currently computers have outperformed humans in many games such as Othello, Checkers, Chess, Backgammon and GO;¹
- the game of Go is extremely complex and the world champion has been recently beaten (March 2016).

¹We are considering games in which both players can see the entire situation of the chess-board/board. We need this. It is not like in card games in which you have to guess in some way. We don't have to guess, for example, about the cards of the opponent. They need to be games that have complete knowledge.

Games

We consider games with the following properties:

- two-player games (we call them MIN and MAX) in which the moves are alternated and players have complementary objective functions (win and lose);
- games with perfect knowledge in which players have the same information (not true in card games such as poker, bridge, *etc.*).²

The development of a match can be interpreted as a tree in which the root is the starting position and leaves the final positions.³

Games in AI



Figure 5.1

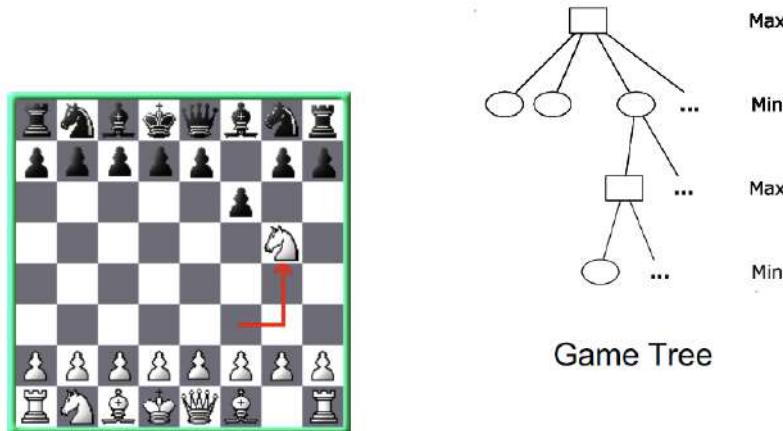


Figure 5.2

In the figure above note that MAX and MIN are alternating.

²Not hidden parts of the game. We don't have to rely on probability.

³This aspect is very important because we can have a description of all overall possible trees. We can have all possible matches that you can do starting from an empty chessboard. At each level (MIN/MAX alternating) I consider all possible moves and counter moves of a player, so I can have a tree with all possible matches. It is easy to imagine that in a game the tree is huge. So exploring it very exhaustively could be very expensive in many terms and generally we cannot do it unless we have an incredible calculus power.

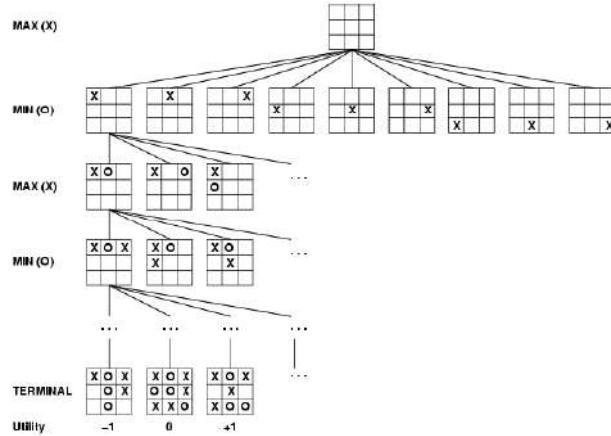
Game Tree

Figure 5.3

The game shown in the figure above is a very simple game in which we have 2 players: one pose a circle (O) and the other an 'X'; the goal is to put 3 circles or 3 'X' in a row/column/diagonal.

In the 'first move' (second row of the figure above) you have 9 positions to put an 'X'.

For each 'X' put in the previous step you have 8 counter moves of putting the circle (third row of the figure above) and so on... till saturate the square.

Note that the tree related to this game is relatively small.

Note that we have three tags indicating the final result:

- $-1 \longleftrightarrow \text{MIN wins};$
- $0 \longleftrightarrow \text{tie};$
- $+1 \longleftrightarrow \text{MAX wins}.$

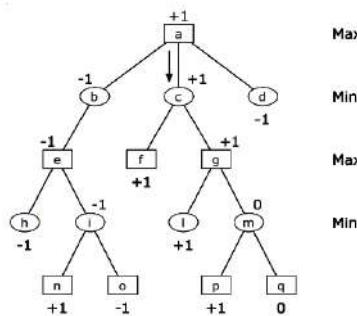
MIN-MAX algorithm

Figure 5.4

On the basis of the previous considerations we can create the MIN-MAX algorithm; see ahead to understand how the algorithm works and how the following figure is constructed.

- The minmax algorithm is designed to determine the optimal strategy for MAX and to suggest, therefore, the first best move to be performed.⁴
- MIN is assumed to play at his best.⁵
- We are not interested in the path but only in the next move.

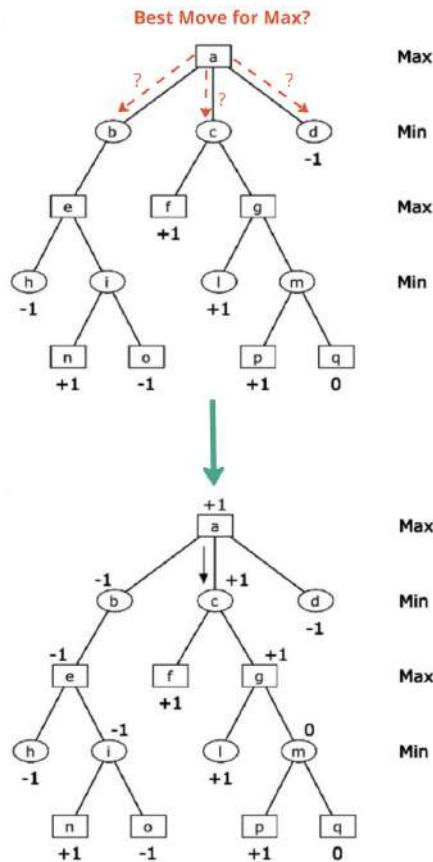


Figure 5.5

⁴So it is designed to determine the optimal strategy for the first player (only one player). If MAX is the first one the value +1 will be associated to MAX and basically at the end of the MIN/MAX algorithm you will suggest MAX to do the *first move*; you are suggesting only the first move, you are not suggesting a path but just the first move which is based on the development of the overall tree. So the algorithm works in this way: you explore the all game tree that is: all possible moves for MAX, all possible moves for MIN. You obtain leaf nodes and these leaf nodes are terminal positions where the match ended (-1 victory of MIN, +1 victory of MAX, 0 a tie). If you can open and explore the all search tree that is easy.

⁵We assume that also the opponent is playing at his best.

More specifically consider the image above; so we need to create an algorithm that starts from the first tree and select the *best first move* for MAX. Let's complete the first tree applying the algorithm in order to obtain the second tree which shows that the best move is going to c .

- Focusing on node i we are playing as MIN (circle). MIN here has to select between $+1$ (leaf n) and -1 (leaf o); of course MIN will select -1 so MIN has a winning option,⁶ therefore we put -1 on leaf i .
- Focusing on node e we are playing as MAX (square). Because of the previous point we have -1 on leaf h and -1 on leaf i , so MAX in e has a loosing position and we will put -1 on leaf e .
- On node b MIN has a winning position so we put -1 .
- On node m MIN has to chose between $+1$ and 0 so MIN will choose 0 and we put 0 on leaf m .
- On node g MAX has to chose between $+1$ and 0 so we put $+1$.
- On node c MIN has to choose between $+1$ and -1 , so in any case it has a loosing position and we put $+1$ on it.

Therefore the *best first move* for MAX is going to leaf c .

Note: what do you think could be the main problem for a general tree of games? Even for games that are not so difficult the search space will be huge. So at some point we need to stop the exploration and to evaluate how good or how bad is a partial node, where there is no a victory, is something in between, so you need to evaluate how good or how bad is a given configuration, and you should do that with heuristics. In the following page we will see the description of the MIN/MAX algorithm in case we can explore the overall search tree but of course this is not always the case or better it is a very rare case because in general you have to stop at some point, after 2 or 3 moves you have to stop. For example in the case of chess we have, only at the beginning of the game, 400 possible moves, they become more than 144000 for the second move and so on... till 35^{100} nodes. This implies a huge amount of time even for the fastest computer. So it is recommandend to use – and we have to – some evaluation function/heuristic function that observes a given position/configuration of the chessboard and evaluates it assigning a value that tell us how good/how bad is that configuration.

MIN-MAX

Leaves are labeled with 1 and -1 .

1 is the victory of MAX and -1 is the victory on MIN.

MIN tries to get to -1 (minimizer), MAX to $+1$ (maximizer).

Note that to understand the $+1, -1, 0$ configuration of the tree, start reading from the end and go back.

MAX moves for square nodes while MIN for circle nodes.

Consider the following image and the following logic. Let us explain how the algorithm works.

⁶Remember: MIN is assumed to play at his best.

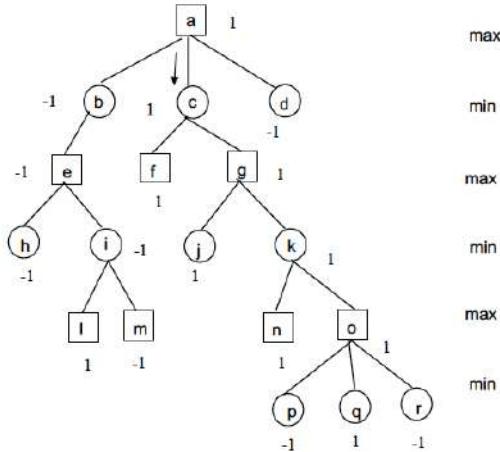


Figure 5.6

Consider the node **o**.

- MAX should move. The game is over in one move. MAX can move to **r** or **p** and lose, or he can move to **q** and win.
- Therefore **o** is a winning position for MAX and is labeled with a (+1).

Consider the node **k**.

- Whatever move MIN does, MIN loses. So the label is (+1).

Consider the node **i**.

- MIN has a winning option and the node is labeled with a (-1).

Then:

- a node where MAX has to move has a label equal to the maximum of the children labels. The opposite happens for MIN.

MIN-MAX algorithm

Let's see the main characteristics of the algorithm.

- To evaluate a node **n**:
 1. expand the entire tree under **n**;⁷
 2. evaluate the leaves as winners for MAX or MIN;⁸
 3. select **n'** as an unlabeled node whose children are labeled. If it does not exist then return the value assigned to **n**;
 4. if **n'** is a node in which MIN has to move, then label it with the minimum value of the children label; if **n'** is a node in which MAX has to move, then label it with the maximum value of the children label; return to 3.

⁷In some case this represents a huge expansion! So we need to know how to cut the tree in an optimal way.

⁸In some games and in some cases it could be a value for a tie.

- In case of parity the label is 0.
- Indeed we can assign temporary values to nodes and update them when children have a value.⁹
- Complexity in time and space = b^d .¹⁰

MIN-MAX algorithm in detail

Let's see in details.¹¹

- To evaluate a node n in a game tree.
 1. Put n in L , a list of open nodes (not yet expanded nodes).
 2. Let x be the first node in L . If $x = n$ and there is a value assigned to it, then return this value.
 3. Otherwise, if x has an assigned value V_x , p is the father of x and V_p a provisional value assigned to it:
 - (a) if p is a MIN node, $V_p = \min(V_p, V_x)$;
 - (b) otherwise $V_p = \max(V_p, V_x)$;
 - (c) remove x from L and return to step 2.
 4. If x is not assigned any value and is a leaf node, assign it either a 1, or a -1, or a 0. Put x in L because we need to update its ancestors and return to step 2.¹²
 5. If x is not assigned any value, and is not a terminal node, assign $V_x = -\inf$ if x is a MAX and $V_x = +\inf$ if it is a MIN. Add the children of x to L and return to step 2.¹³
- Complexity in space: $b \cdot d$.

MIN-MAX properties

Main properties.

- *Complete?* Yes (if the tree is finite).¹⁴
- *Optimal?* Yes (against an opponent who plays at his/her best).
- *Temporal complexity?* $O(b^m)$.
- *Space complexity?* $O(bm)$ (depth-first).¹⁵
- For chess, $b \sim 35$, $m \sim 100$ for ‘reasonable’ games \implies this solution is out of reach!

We need to prune the tree.

⁹Like $+\infty$ or very big number for the MIN; Like $-\infty$ or very low number for the MAX.

¹⁰Where d refers to depth and b to the branching factor. You can imagine that if b and d are very big this approach is not valid.

¹¹In any case, see examples ahead to understand better.

¹²Depending on whether it is a solution for MAX, MIN or a tie.

¹³This is the version in which we pre-assign values $\pm\infty$ to every node as soon as we extract it from the list.

¹⁴Otherwise we could have infinite loops.

¹⁵Depth-first is not complete (referring to the first point), but MIN-MAX is complete because the tree is finite, there are no infinite loops; so also depth-first in absence of infinite loops is complete.

MIN-MAX algorithm revised

Some considerations.

- If we have to develop the whole tree, the procedure is very inefficient (exponential).¹⁶
- If b is the branching factor and d is the depth, then the number of nodes becomes b^d .
- Solution (Shannon, 1949): look forward few levels and assess the configuration of a non-terminal node. In practice we apply the MIN-MAX algorithm up to a certain depth.
- Use an evaluation function for estimating the quality of a certain node:¹⁷
 - $e(n) = -1$ if MIN wins;
 - $e(n) = +1$ if MAX wins;
 - $e(n) = 0$ if they have the same probability.

We can use intermediate values for $e(n)$.¹⁸

Example

In chess, we add the value of all pieces of each player and normalize the result to have a value between -1 and $+1$.

For example a weighted sum of values (linear):

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s) \quad (5.1.1)$$

Where:

- $w_1 = 9$;
- $f_1(s) = (\text{number of white queens}) - (\text{number of black queens})$, etc. It could be refined taking into account the relative positions: is the king protected? Do pawns protect any piece? etc.

Trade-off between search and evaluation function.¹⁹

In any case we need to select an evaluation function, $e(n)$.²⁰

¹⁶So at a certain point we need to cut, to stop.

¹⁷Basically, starting from a node, if I go down expanding a few levels I cannot really say if this is a winning position for MAX or for MIN because the entire match is not concluded, I stop well before the end of the game. So we are in an intermediate configuration: how can we decide if we are in a solution that is good for MAX or good for MIN? We have to use *evaluation functions* (heuristic functions), essentially as human players do (they stop, analyze the history and consequence of a move).

¹⁸Other than -1 , $+1$, 0 . We are in probabilistic cases.

¹⁹Trade-off between the time you take for computing a heuristic (because if it is a very complicated heuristic could happen that you take too much time to compute it and therefore this effort would not pay) in relation to the search you are avoiding by stopping at that level.

²⁰This function can be defined either by an expert of the game or learnt by using past games, history of the game.

MIN-MAX algorithm part 2

Now, if we have an evaluation, then of course we can change a little bit the algorithm. Look at the point 4 below.

1. Put n in L , a list of open nodes (not yet expanded nodes).
2. Let x be the first node in L . If $x = n$ and there is a value assigned to it, then return this value.
3. Otherwise, if x has an assigned value V_x , p is the father of x and V_p a provisional value assigned to it:
 - if p is a MIN node, $V_p = \min(V_p, V_x)$;
 - otherwise $V_p = \max(V_p, V_x)$;
 - remove x from L and return to step 2.
4. If x is not assigned any value and is a leaf node or you *decide not to expand the tree further, assign it the value using the evaluation function $e(x)$.*²¹ Put x in L because you will have to update the ancestors and return to step 2.
5. If x is not assigned any value, and is not a terminal node, assign $V_x = -\infty$ if x is a MAX and $V_x = +\infty$ if it is a MIN. Add the children of x to L and return to step 2.

MIN-MAX pseudo-code

```

function MINIMAX-DECISION(state) returns an action
  v  $\leftarrow$  MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow -\infty$ 
  for a, s in SUCCESSORS(state) do
    v  $\leftarrow \max(v, \text{MIN-VALUE}(s))$ 
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow \infty$ 
  for a, s in SUCCESSORS(state) do
    v  $\leftarrow \min(v, \text{MAX-VALUE}(s))$ 
  return v

```

Figure 5.7

Let's analyze this pseudo-code.

²¹We can decide if expanding the tree or not and evaluate the node.

- **function MINIMAX-DECISION** \longleftrightarrow here we return an **action**; you take a **state**; you take the **MAX-VALUE** of a **state** and assign it to **v**; then return the **action** in the **SUCCESSORS** of the **state** with a value that is **v**. So I have chosen either the MIN or the MAX for **v** and therefore I will suggest that move.
- **function MAX-VALUE** \longleftrightarrow when we have a **TERMINAL-TEST(state)** which means either I have concluded the search toward a victory position for MIN, MAX (or a tie) or **TERMINAL-TEST** tells me I will not expand further this node (because I've reached the maximum depth, for example, or I have spent enough time in exploring that part). It can be also a **CUTOFF-TEST** with a depth. And basically when I have reached this terminal node then I will return the **UTILITY** or the evaluation of the state that provides me an heuristic evaluation of the node. Make you notice that the **v** in **v <- MAX(v, MIN-VALUE(s))**, if we are in the MAX case, it is labeled with the maximum between the previous value of **v** and the minimum value deriving from the successors of **v**. This is a recursive algorithm (because this is a tree search) and you basically explore: take the minimum value that comes from the children of **v** (why the minimum? Because if **v** is a maximum its children are minimum and so on; the opposite for the MIN case).
- **function MIN-VALUE** \longleftrightarrow the same seen for the MAX is valid with an opposite logic for the MIN.

Evaluation replaced by **TERMINAL-TEST** with: if **CUTOFF-TEST(state, depth)** then return **EVAL(state)**. It also updates depth at each recursive call.

Problems

Let's describe in general the main problems of the MIN-MAX algorithm.

- How do we decide if to expand a node or not?²²
- Note: If $e(n)$ was *perfect*²³ I would not have this problem.²⁴ I could stop after the first level and evaluate the children of the root node.
- Simple solution from the computational point of view: always expand nodes up to a certain depth **p**.
- Problems:
 - more tactically complicated moves²⁵ (with higher variance for $e(n)$) should be evaluated with higher depth up to quiescence when $e(n)$ values change more slowly.
- Horizon effect:

²²So, at some point I have not only to create an evaluation function but I also have to decide whether I go on or stop.

²³Meaning that the evaluation function is able to guide us with the 100% of accuracy.

²⁴I would take the root node, expand only the first set of children.

²⁵Are not evaluated after only one step, they could need more steps to get a meaningful value. It could happen that more going down (expanding) values of $e(n)$ start to stabilize.

- with useless moves, extend depth to values larger than p , so the basic moves are not really taken into account.
- Solution: sometimes it pays to do a secondary search, focused on the best move choice.²⁶

Alpha-Beta cuts

Now we are going to see another part that is extremely important for cutting the tree. The first big cut that we have seen is to limit the depth or to stop at some point and evaluating the nodes. Now we will provide a possible second solution for pruning nodes from this very big search tree.

- From what we have seen so far computers simply play all possible matches up to a certain depth, evaluate leaves and propagate back the evaluation.
- So they also consider moves and nodes that will never occur.²⁷
- You should try to reduce the search space.
- The best-known technique is that of the *alpha-beta cut*.

Example

Let's see how this *alpha-beta cut* works.

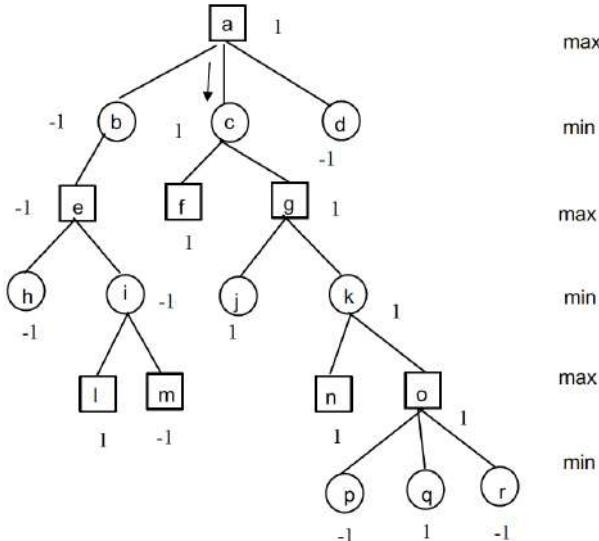


Figure 5.8

Looking at the image above notice:

²⁶I will try to understand which is the best move for a given player and then I will go deeper with these moves because we want to have a clearer idea about the promising paths.

²⁷That means that we are also evaluating some nodes that are completely useless because these nodes will be never chosen by any player ever. So we need to use these informations in order to prune and reduce the search space.

- when we find out that²⁸ the move to **c** is a winning move²⁹, we do not need to expand the nodes **b** and **d**;
- the nodes under **b** will never influence the choice.³⁰

Alpha-Beta cuts: general principles

General principles of the *alpha-beta cuts*.

- Consider a node **N** in the tree. Will the player move to that node?
- If the player had a better choice (we call it **M**) in the parent node level or at any point along the path, then **N** will never be selected. If we reach this conclusion we can eliminate **N**.
- Call *Alpha* the value of the best choice found on the path for MAX (the highest) and *Beta* the value of the best choice found on the path for MIN (the lowest).
- The algorithm updates *Alpha* and *Beta* and cuts branches when their choice is the worst.

Let's see an example.

Example

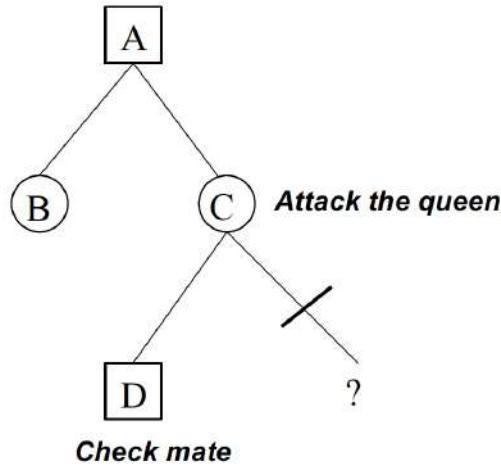


Figure 5.9

No matter which is the evaluation of the other children of C (I realize that I should never move to C).³¹

²⁸After having explored with depth-first.

²⁹So it is a 1.

³⁰The same if we would have nodes under **d**, so we can cut both **b** and **d**.

³¹Assuming the perspective of MAX (start from A as a square) from C (here we are playing as MIN) if we move to D (and playing as MIN at its best we always chose D, so MAX loses) we have a Check mate, so MAX loses; so moving to C we have a worst case.

Another example

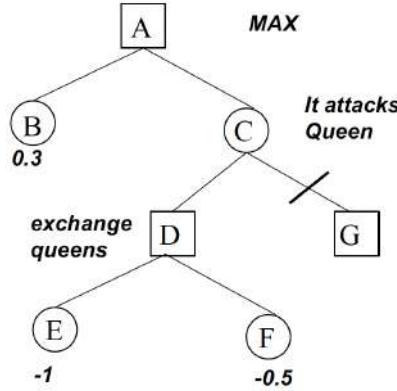


Figure 5.10

What about when we have some values?

- Max in A avoids C because B is better. At most max gets from C a -0.5 so 0.3 is better.
- The subtree in G can be cut as soon as I receive the value of D. Indeed: $C = \min(-0.5, G)$.³² $A = \max(0.3, \min(-0.5, G)) = 0.3$.
- Since A is independent of G, the tree under G can be cut.

Another example

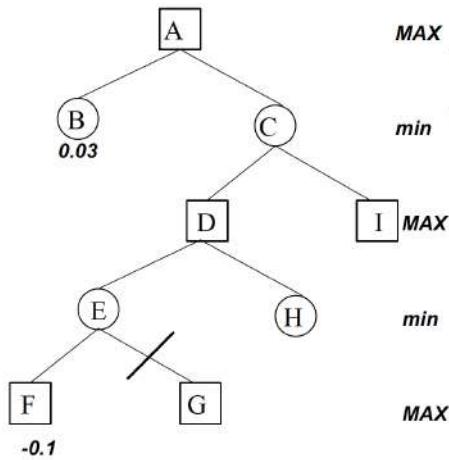


Figure 5.11

Looking at the image above.

³²Because we are playing as a MIN here.

- G is on the search path that will be developed?
- If G is on the search path, then also E is. From \min perspective E can always obtain either -0.1 or worst which is always worst than 0.03 for \max . Then G cannot be in the current search path.

MIN-MAX

A typical exercise presents a tree with labels like below and it will be asked to identify which is the move suggested to the first player; always like that (it's a thing that you should be able to do in almost 1 minute). So we:

- start from leaves already labeled;
- recognize that we have a MAX level and a MIN level;
- suggest the move.

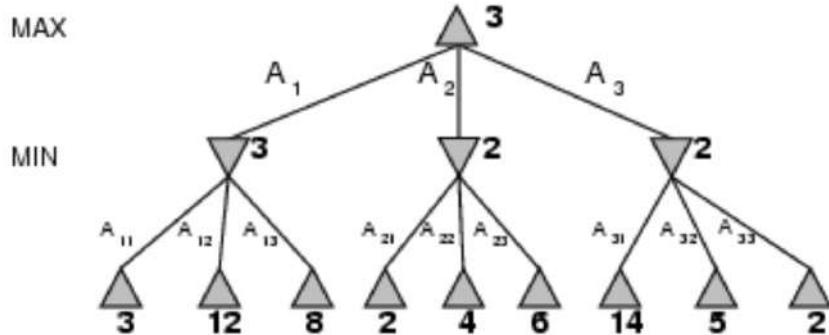


Figure 5.12

Starting from left (to right, going *depth-first*) back to the top, we have:

- among 3, 12 and 8 if we are playing as \min (see one step above) we choose 3, indeed 3 is the label of the \min player in the second level;
- among 2, 4, 6 we chose 2;
- among 14, 5, 2 we choose 2;
- then, playing as \max (see 1 step above) among 3, 2, 2 we choose 3.³³

Now we are going to see how to apply the *Alpha-Beta cut*; we will see it in a way that is slightly different from the canonical one, a more intuitive way that we are going to show (but if you want you can apply the canonical method).

Alpha-Beta cuts

We proceed always *depth-first*; in general when you search you don't have the entire search tree at the beginning but you're exploring it during search.

³³All this to recap how the MIN-MAX algorithm works.

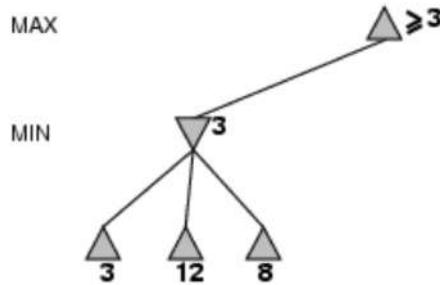


Figure 5.13

Looking at the image above:

- think that at the beginning we have no value for the MAX at the first level, after the first expansion in the *depth-first* way at the left we give it ≥ 3 ;
- we open the MIN node and expand the 3 children. In general the left most branch is always exhaustively evaluated, unless we have a victory for MIN (in this case we are expanding MIN) that would be immediately chosen; for example if we realize that the left most 3 below is a victory for MIN we should chose it avoiding to expand the others (12, 8) and I can cut them;
- anyway here we can see that as soon as you have obtained the values of the subtree at the left \implies MAX got already some information: we know that MAX will extract either 3 or something that is larger/greater because is maximizing. So I label the first MAX triangle with ≥ 3 .

Let's go on...

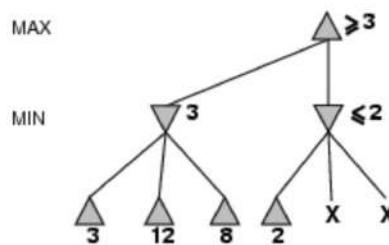


Figure 5.14

Looking at the image above:

- note that also here we give ≤ 2 after having expandend the first child;
- here as soon as we explore the second MIN at the center, then MIN choses something that is either 2 or something that is less or equal to 2; but MAX (at the beginning) has already a 3, so if in the unkown nodes (in the figure

labeled with X) we will have values that are lower than 2 we don't mind³⁴ because MAX (the final scope is that MAX wins) should never choose this node, so we can cut/stop this expansion.

Let's go on...

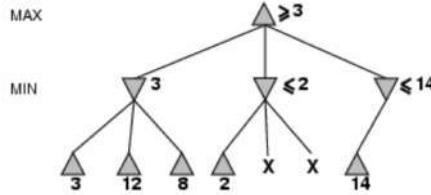


Figure 5.15

Continuing to expand we can see that MIN has as first child 14. Should we stop? MIN has something that is either 14 or something lower than 14, so I cannot stop because if I will have something between 3 and 14 (included) then MAX will chose this path here.

Let's go on...

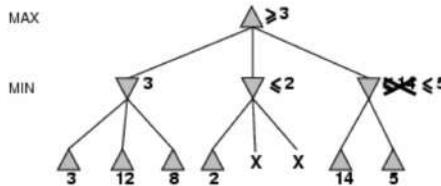


Figure 5.16

Let's go on...

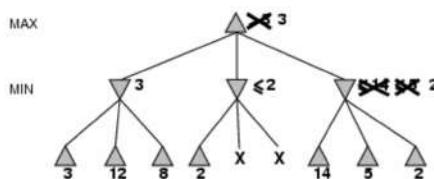


Figure 5.17

Looking at the image above:

- so in the center block we avoid the expansion of the children (the X) avoiding to do operations that are not meaningful;

³⁴Remember also that MAX and MIN play also at their best. Here, playing as MIN, we will choose 2 if in the unknown nodes we will find something greater than 2 or we will choose a value less than 2 if we find it in the unknown nodes. But assuming the perspective of the MAX we can avoid other actions because from this expansion we could obtain at most 2 which is lower than 3.

- with the last child of the MIN to the right we can definitely say that the value associated to the first MAX is 3 and that the best path is the most left.

So this represents the variant developed regarding the *Alpha-Beta cut*:

- you should label the nodes with greater/lower or equal (\geq or \leq) than a value;
- then you would ask yourself whether you have to continue to expand the nodes and you don't have to do that in case you have something better already discovered by the other player.

Principles:

- we generate depth-first search tree, left-to-right;³⁵
- propagate the (estimated) values from the leaves.³⁶

The Alpha-Beta Terminology and Principle

Let's see how the algorithm works in practise; so the algorithm that is applied by our computers. What we saw in the past pages explain an intuitive way to apply the logic, but let's see the canonic algorithm. So we have definitions for *Alpha* and *Beta* and comparing these two categories of values we decide whether or not to cut a given subtree.

Specific terminology:

- the (temporary) values in MAX-nodes are *ALPHA-values*;
- the (temporary) values in MIN-nodes are *BETA-values*.

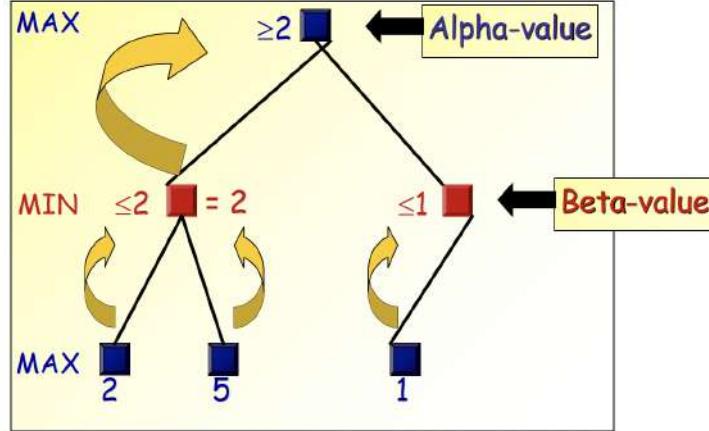


Figure 5.18

³⁵Explore the left most etc.

³⁶Propagate back, from bottom to the top.

Let's try to understand practically what are *Alpha* and *Beta*. We said that when I have a temporary value in a node, in the MAX node this value is the *Alpha value*.

Let's see what we have in the figure above:

- remember we are exploring in a *depth-first* way (left-first); so we go down on the left till expanding MIN children;
- first we expand the left child which has value 2 and so we assign ≤ 2 to MIN as temporary value;
- then we expand the other child which has value 5, so playing as a MIN we confirm 2 as value for the MIN;
- then we propagate back this information to MAX which will be assigned of the *temporary*³⁷ value ≥ 2 (*Alpha value*);
- then we explore the right branch and so on with this logic.

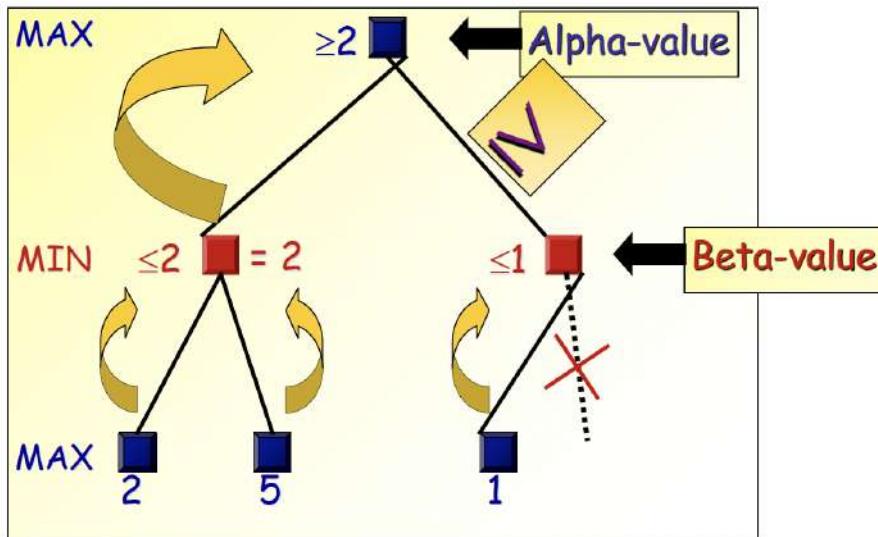


Figure 5.19

Principle.

- If a *Alpha-value* is greater than or equal than a *Beta-value* of a descending node: *stop the generation of children of the descending node!*³⁸
- If a *Beta-value* is smaller than or equal than a *Alpha-value* of a descending node: *stop the generation of children of the descending node!*

³⁷We explored only the left branch.

³⁸See the figure above.

MIN MAX with Alpha-Beta cuts

Let's see the evolution of this exercise (this is an example of the exercises that we will have at the exam).

First thing that it is asked is to compute the MIN-MAX algorithm.

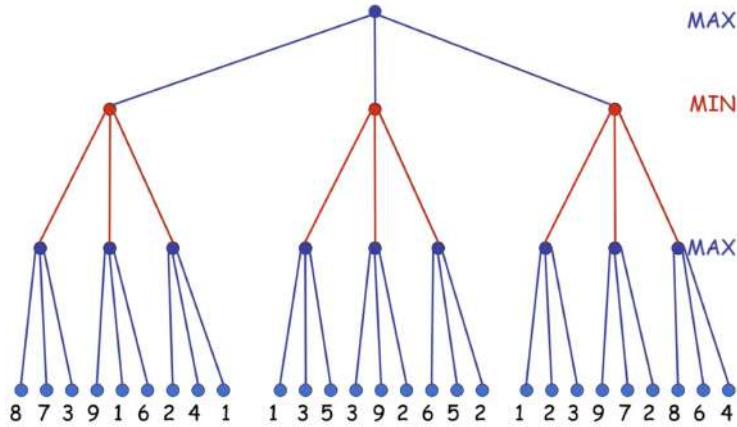


Figure 5.20

Example of real scenario: we can associate the figure above with the final configuration of the chessboard in a chess game and asking us: with this given configuration which is the best game I can play? And do that for all the possible final configuration of the chessboard in a chess game.

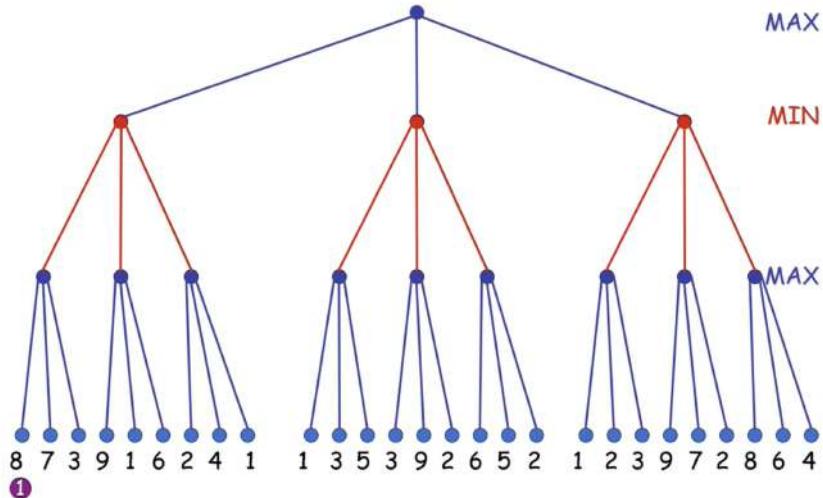


Figure 5.21

Looking at the figure above: the 1 in violet stands for first action, the 2 in violet that we will see ahead for the second action and so on.
So here the first node we explore is 8.

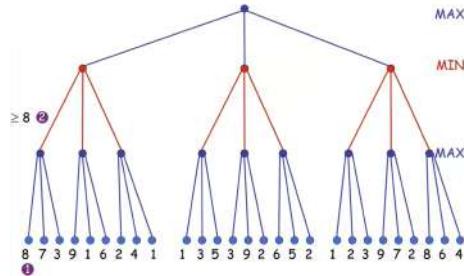


Figure 5.22

Looking at the figure above: so as soon I have the value 8, I update the *Alpha value* (we are playing as a MAX) with ≥ 8 .

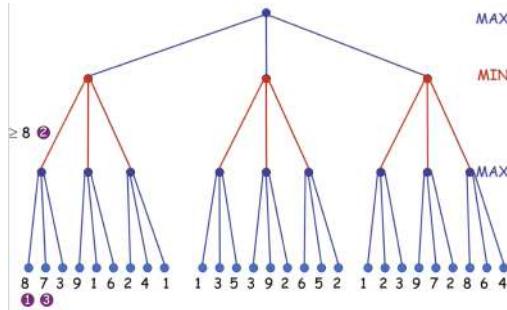


Figure 5.23

Looking at the figure above: then I explore 7; 7 is lower than 8 so I keep 8.

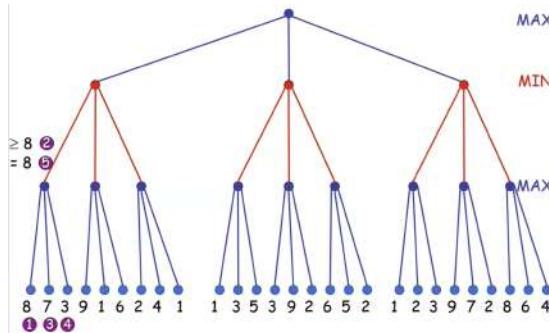


Figure 5.24

Looking at the figure above: then I explore 3; 3 is lower than 8 so I keep 8 and furthermore I confirm it because I have expandend all the children.

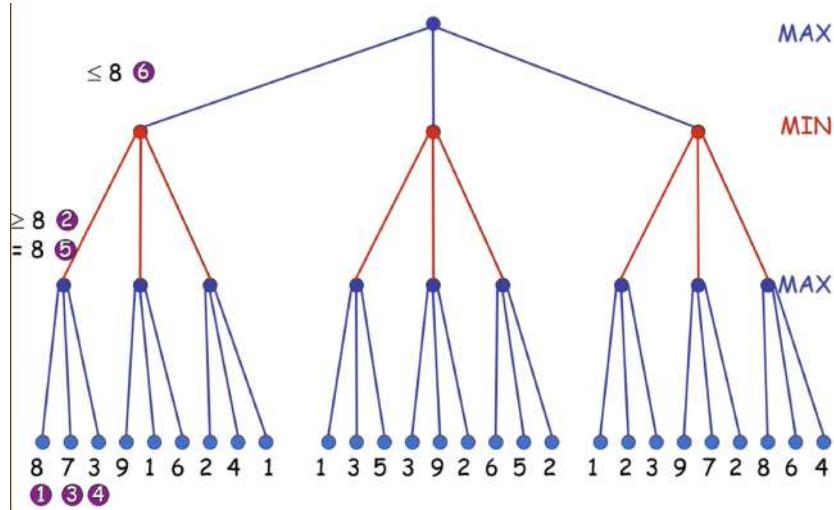


Figure 5.25

Looking at the figure above: then as soon as we have the stable value 8 for the MAX, we will update the value for the MIN with the *Beta value* ≤ 8 .

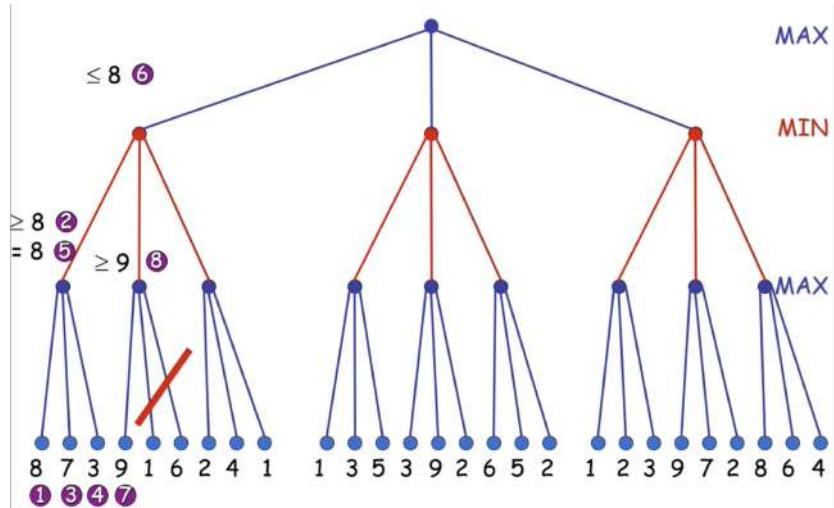


Figure 5.26

Looking at the figure above: then continuing the explanation we find that the first child of the second MAX at the second level is 9, so we update the *Alpha value* for the MAX to ≥ 9 but so we would have a *Beta value* (≤ 8) that is smaller than (or equal to) an *Alpha value* of a descending node, so we stop the generation of children of the descending node!

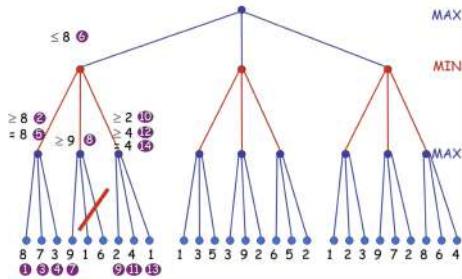


Figure 5.27

And so on...

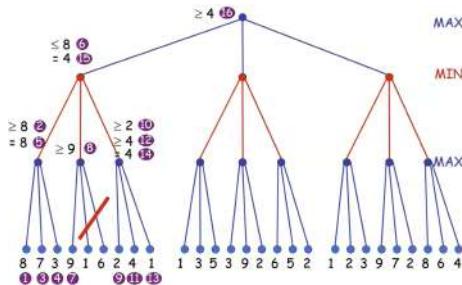


Figure 5.28

Till the entire development...

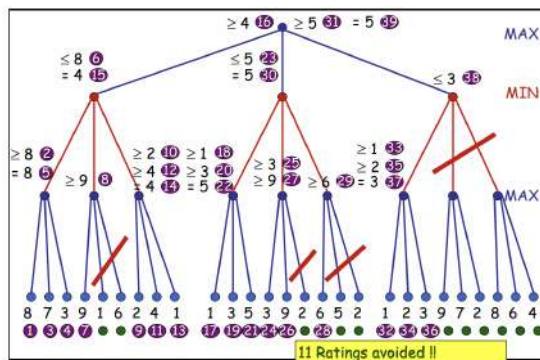


Figure 5.29

Algorithm Alfa-Beta

Let's specifically see the algorithm (the important thing is to understand the macro-logic seen before; this is a formalization).

To evaluate a node n in a game tree.

1. Put n in L , a list of open nodes (not yet expanded nodes).

2. Let x be the first node in L . If $x = n$ and there is a value assigned to it, return this value.
3. Otherwise, if x has a value assigned Vx , let p be the father of x . If x has not an assigned value, go to step 5.
 - (a) We need to determine if p and its children can be removed from the tree. If p is a MIN node, Alpha is the maximum of all current values assigned to the brothers of p and of nodes that are ancestors of p .
 - (b) If there are no such values $\text{alpha} = -\text{infty}$.³⁹
 - (c) If $Vx \leq \text{alpha}$, then remove p and all his descendants from L (dually if p is a MAX node).
4. If p cannot be eliminated, let Vp be its current value. If p is a MIN node, $Vp = \text{min}(Vp, Vx)$, otherwise $Vp = \text{max}(Vp, Vx)$. Remove x from L and return to step 2.
5. If x is not assigned to any value and is a terminal node, or we decide not to further expand the tree, give it a value using the evaluation function $e(x)$. Leave x in L because you have to update the ancestors and return to step 2.
6. If x is not assigned to any value, and is not a terminal node, assign $Vx = -\text{infty}$ if x is a MAX and $Vx = +\text{infty}$ if it is a MIN. Add the children of x to L and return to step 2.

Alfa-Beta pseudocode

Let's specifically see the pseudo-code (the important thing is to understand the macro-logic seen before; this is a formalization).

```

function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v  $\leftarrow$  MAX-VALUE(state,  $-\infty, +\infty$ )
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state,  $\alpha, \beta$ ) returns a utility value
  inputs: state, current state in game
     $\alpha$ , the value of the best alternative for MAX along the path to state
     $\beta$ , the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow -\infty$ 
  for a, s in SUCCESSORS(state) do
    v  $\leftarrow$  MAX(v, MIN-VALUE(s,  $\alpha, \beta$ ))
    if v  $\geq \beta$  then return v
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return v

```

Figure 5.30

³⁹Or equivalently a very big negative number.

```

function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: state, current state in game
     $\alpha$ , the value of the best alternative for MAX along the path to state
     $\beta$ , the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$ 
  for a, s in SUCCESSORS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta))$ 
    if  $v \leq \alpha$  then return v
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return v

```

Figure 5.31

Effectiveness of cuts

Some point to consider.

- Obviously if you always evaluate the worst nodes, then best nodes are always in the search path and there is no cut.
- The best case is when the best nodes are evaluated first.⁴⁰ The other are always cut (of course it is entirely theoretical).
- In this case we can reduce the number of nodes (which is b^d) to about $b^{d/2}$ (in practice, it is reduced by the square root branching factor, or you can look forward twice more at the same time).⁴¹
- In the average case with random distribution of the node values, the number of nodes becomes about $b^{3d/4}$.
- So it is important to choose a good order of the children of a node.
- Also note that all the results mentioned herein are for an ‘ideal’ game tree with depth and branching fixed for all the branches and nodes.
- Repeated States, the list of closed nodes (see graph search).

Perfect example

A complete *perfect* tree is the following.

Obviously is a theoretical case.

It would be very difficult to have such a case in a real scenario.

⁴⁰Searching left most, depth first. See the tree in the example before: best cases are when best nodes are at the left. Best case ever is to have victory for the first player in the first subtree.

⁴¹Evaluation done considering a *perfect example*; see ahead to have also a graphical view of what is the perfect distribution.

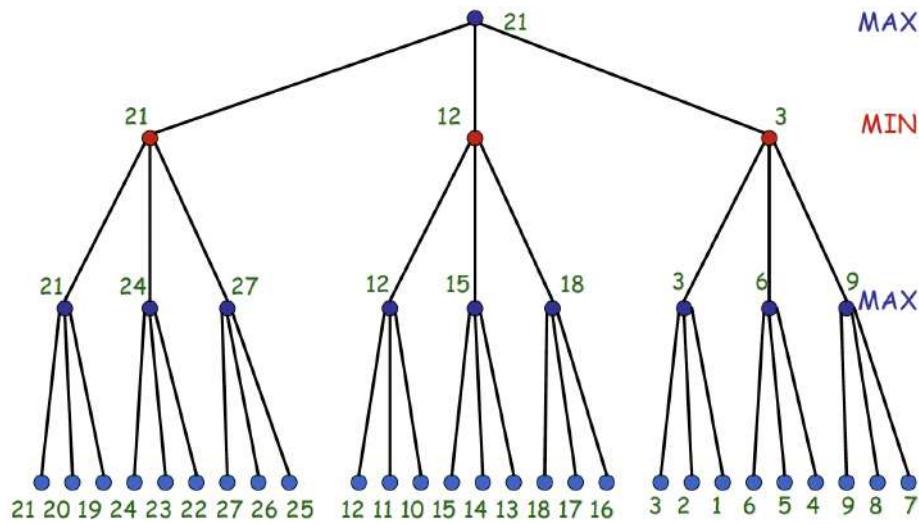


Figure 5.32

At each level the best node is on the left.

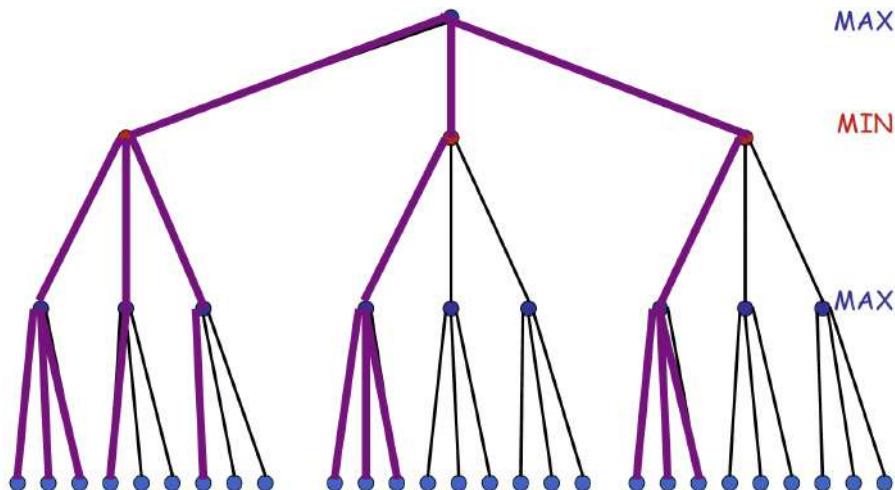


Figure 5.33

The game of chess

The problem size is enormous. Only at the beginning of the game we have 400 possible moves, they become more than 144,000 for the second move and so on. In particular, chess has a branching factor of ~ 35 and ~ 50 moves for each player. So we have 35^{100} nodes (actually lawful moves are 10^{40}).

There must be an evaluation function. In particular, it will give a weight to each piece, but this is not enough.

It must also take account of the relative position of the pieces (two towers in a column sometimes are more important than a queen) and the moment of the

game (opening, intermediate stages, end).

Example: evaluation of a knight

The moment of the game is also important. For example, knights are important in the middle of the game but they are not important in the end of the game with just a few pieces.

But even giving a weight to all these components is not enough, we also need a function that better links all these parameters.

The other choice is how much to go deep into the solution tree. The machine is expected to respond in a time comparable to that of a human player.

A computer processes around 1000 positions per second (but can even reach 2500).

Each move takes a maximum of 150 seconds. So a computer processes around 150000 possible moves that correspond to about 3-4 levels playing at beginner level.

Alpha-Beta cuts become essential

Current programs explore about 7 levels and process about 250000 configurations for each move but under certain circumstances it can explore up to 20 levels and 700000 configurations.

Almost all programs use the time that the human player uses to choose his move. It has been estimated that a human player, even if expert, actually never explores for more than 5 levels, with substantial cuts. He then uses a heuristic evaluation function.

All chess programs consult the library of openings (there are openings that have been previously fully explored and that can affect the entire game).

While a computer is very strong in the middle part of the game, the human player is smarter at the end, when exploring many moves is less important. But today's chess programs avoid this and tend to use libraries and specialized algorithms for the final.⁴²

Deep Blue

On 10/05/1997 (very old), in New York, DeepBlue beat the world champion in a match of six games (match DeepBlue - Kasparov: 2-1 and three draws).

There is a system of evaluation of the playing strength (*Elo*) capable of measuring the progress of human and artificial players.

Points are earned in official tournaments.

- Beginner player: 500 Elo points.
- Master: 2200.
- World Champion: 2800.
- Deep Blue: 3000.

⁴²Indeed here can be used also *machine learning* techniques for learning special evaluation functions.

Artificial players can be classified into two categories according to whether they use generic hardware (PC) or special hardware (the case of Deep Blue).

In particular, Deep Blue used a general-purpose parallel machine with specialized chip processors for generating moves and evaluations.⁴³

DeepMind approach

DeepMind was a small company in London co-founded, in 2011, by Demis Hassabis, child chess prodigy, video game designer and computational neuroscientist. In 2014 DeepMind was bought for hundreds of millions of dollars by Google. Algorithm based on deep learning (neural networks) and learning that teaches itself to play (even at video games), and often much better than human players.⁴⁴ The algorithm has been trained on 49 different games for the Atari 2600, all developed for generations of teenagers.

It managed to beat at the game of GO the European champion in October 2015.

Press release March 9, 2016

The Google supercomputer beats the world champion of *Go*:⁴⁵

- the first round of the historic match between artificial intelligence and man went to AlphaGo program DeepMind Technologies, which has vanquished the South Korean Lee Sedol, considered the best player in the ancient chinese board game.

Go is a board game originated in China over 2500 years ago and very popular in East Asia. Played over 40 million people worldwide, it has simple rules: players, in turn, must place the white or black stones on a table, trying to capture the opponent's stones or dominate the empty spaces to conquer the territory. But the simplicity of the rules hides a profound complexity of the game: the possible positions are higher than the number of atoms in the universe.

Go Board



Figure 5.34

⁴³This is a very rude approach if we look at the modern techniques; great power of calculus taken for the search.

⁴⁴More modern approach.

⁴⁵If chess has a huge search space, the game of *Go* is very very more complicated: the tree search is extremely larger.

Exercise

Consider the following game tree where the first player is Max. Show how the *min-max algorithm* works and show the *alpha-beta cuts*. Also, show which is the proposed move for the first player.⁴⁶

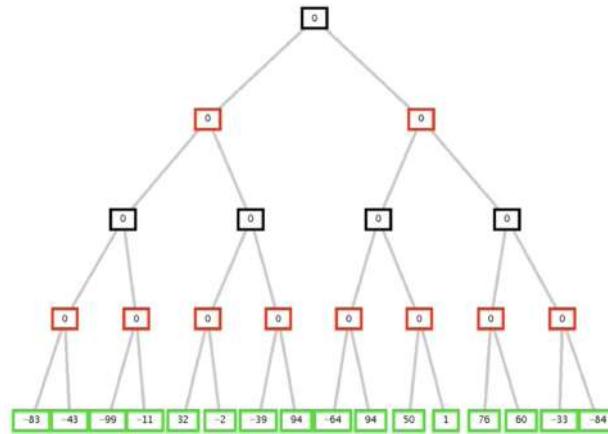


Figure 5.35

Suggestion: you can directly solve the exam here on the same tree: the important thing is that you label the nodes and you clearly explain which arcs are removed.

MIN-MAX solution:

Min-max

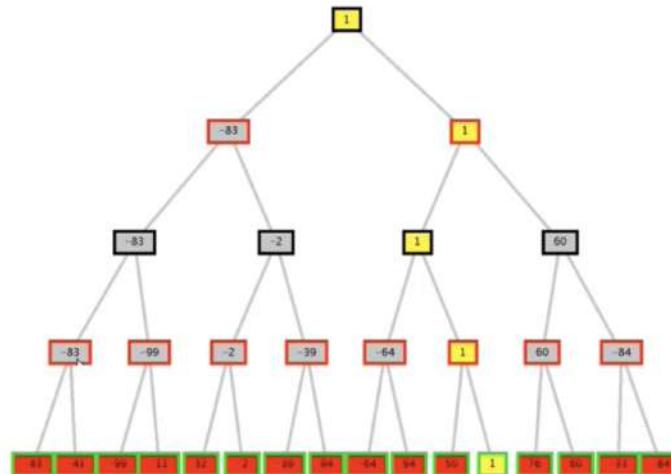


Figure 5.36

⁴⁶Just the first move; it is not requested the entire path. This is a sample of the exercises proposed in the exams.

Note that the suggested move is the move that follows the yellow path; so the one that bring the first 1 to the second 1.

Alpha-Beta solution:

Alfa-beta:

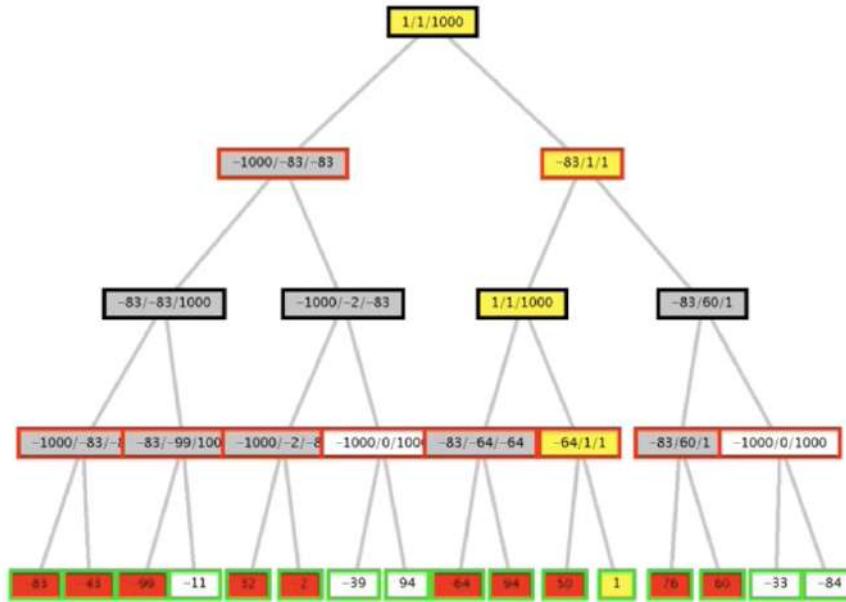


Figure 5.37

We reported in the exercise the values for *Alpha* and *Beta*.

In the figure above the white squares indicate the parts that are cut: for example on the left as soon as we find -99 we cut the branch. The same reasoning for the other two branches highlighted in white.

Chapter 6

Planning

6.1 Introduction to Planning

This topic became very crucial around 90s in US basically for helping logistics and organization of military purposes.

Planning is a vital activity of human beings. We are planning all the time and mostly in an *intuitive way*; this last aspect should highlight how complex for a machine would be to plan.

Automated Planning

Automated Planning is an important problem solving activity which consists in synthesizing a *sequence of actions* performed by an agent that leads from an initial state of the world to a given target state (*goal*).¹

Given a description:

- an initial state;
- a set of actions you can perform;²
- a state to achieve (*goal*).

Find:

- a *plan*, a partially or totally ordered set of actions needed to achieve the goal from the initial state.

Planning is:

¹Analizing this description/definition, after having covered *Search Strategies*, you can immediately say that this is a *search problem* because we have *states* (initial state of the world), you can perform alternative actions in this state and these actions make evolve the state into another state till a desired final state. So, in principle *Automated Planning* is a *Search Problem*: the action is the operator that enables to move from one state to another state. In fact, Planning is a Search Problem; we will see how to solve Planning with Search Strategies: which is the search space that is generated by a planner and it can be built in two different ways (we will see in this section) but the problem is that this Planning problem is too complex and the search space in general is too large to be exhaustively explored and therefore we end up creating specific algorithms that take into account planning in a specific way in the sense that they are still search algorithms, but they need to be manipulated in a specific way.

²Your agent can perform.

- one application per se;
- a common activity in many applications such as:
 - diagnosis, test plans and actions to repair (reconfigure) a system;
 - scheduling;³
 - robotics.⁴

In fact, Planning is often a subapplication of other larger application.

Automated Planning: basics

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

1. a *formal representation of the initial state*;
2. a *representation of a goal*;
3. a *formal (not ambiguous) description of the executable actions (also called operators)*.

These three points define the so called *Domain knowledge*.

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

Action representation

A planner relies on a formal description of the executable actions. It is called *Domain Theory*.

Each action is identified by a name and declaratively modeled through *preconditions* and *postconditions*.

- Preconditions are the conditions which must hold to ensure that the action can be executed.⁵
- Postconditions represent the effects of the action on the world.⁶

Often the Domain Theory consists of operators containing variables that define *classes of actions*. A different instantiation of the variables corresponds to a different action.

³Problem of assigning a starting time and resources to a set of activities for reaching any goal. Before the scheduling problem you have to define what activities you have to perform and this is a planning task.

⁴For making a robot acting autonomously in an environment you need to plan its activities.

⁵E.g. if you want to move an object on a table from a position to another one the preconditions could be that the object is in the initial position, that your hand is free for picking the object up; if these preconditions are true then you can apply the action.

⁶When you apply an action, something changes in the state and these changes are described in the postconditions.

The Block World example

The *Block World* is a very widely used example because it is easy to understand; but notice that if we have many many objects, many configurations and actions it would be difficult to solve.

In the *Block World* problem we have blocks with a name (better on top of them) and basically what you can do is you can *stack* one on the top of the other or *unstack*, *pickup* from the table and *putdown* on the table.

In total four actions.

With these 4 actions you can solve a *Block World* problem starting from an initial configuration/disposition and wanting to reach a desired configuration.

Actions:

STACK(X, Y)	#Action
IF: holding(X) and clear(Y) #Precondition	
THEN: handempty and clear(X) and on(X, Y) #Postcondition	

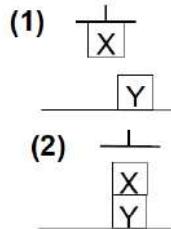


Figure 6.1

UNSTACK(X, Y)
IF: handempty and clear(X) and on(X, Y)
THEN: holding(X) and clear(Y)

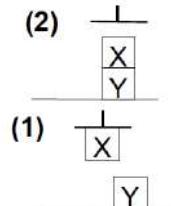


Figure 6.2

PICKUP(X)
IF: ontable(X) and clear(X) and handempty

THEN: holding(X)

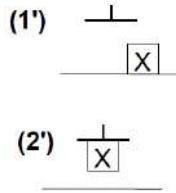


Figure 6.3

PUTDOWN(X)

IF: holding(X)

THEN: ontable(X) and clear(X) and handempty

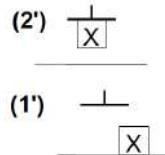


Figure 6.4

Planning

Solving process for deciding the steps (actions) that solve a planning problem should be:

- *non decomposable*:⁷ there can be interaction between subgoals;
- *reversible*:⁸ the choices made during the plan generation are backtrackable.

A planner is *complete* if it always finds a plan when it exists.⁹

⁷Basically the thing is that sometimes if you have to reach 2 different goals they may interact and this interaction could be dangerous: *e.g.* the system reaches the first goal and ask itself why the need of reaching the second one, there can be conflicts in such a way that determine the impossibility of reaching the main goal that is represented by the satisfaction of the 2 subgoals. We will see examples on that.

⁸The thing is that: we're operating offline, so we are not executing anything while planning in the sense that you take a picture of the current state, then you plan in your planning world, then – once you've created a plan – you start the execution for achieving the goal. The thing is that this description of the world is supposed to be static, nobody changes this state unless the planner which is the only actor that changes the description of the world (in this way is not properly realistic). The thing is done offline, so you before need to plan and only then you execute, so why the choices made during plans generation should be backtrackable, because you're not executing anything, you're just thinking to execute.

⁹If a plan doesn't exist, the planners can run indefinitely; and this is a property of the problem.

A planner is *correct* when the solution found leads from the initial state to the goal.¹⁰

Execution

The execution is the implementation of the plan.¹¹

- *Irreversible*: often the execution of an action is not backtrackable.¹²
- *Non deterministic*:¹³ the plan can have effects that are different from what we expect. Working in the real world pertains uncertainty. In some cases the plan may fail. Then we can in principle find a recovery plan from scratch or partially.

Planning techniques

Here a list of the major planning techniques.

- Deductive planning.¹⁴
- Planning as search.¹⁵
- Linear planning.
- Nonlinear planning - Partial Order Planning (POP).
- Hierarchical planning.
- Conditional planning.
- Graph-based planning.

Generative planning

It is an off-line planning that produces the whole plan before execution. It works on a snapshot of the current state (initial state).

It is based on some (often unrealistic) assumptions:

- *atomic time*: actions are not interruptible;
- *deterministic* effects;

¹⁰The planning problem, it is called *semi-decidable* and basically this planning problem has the property that you are sure that if a plan exists a *complete planner R* can find it but if the plan does not exist it can run indefinitely and this is a huge problem.

¹¹The execution instead happens in the ‘real’ world and starts doing things.

¹²If you put something on the table in a given position you can pickup it and then put it there again; but if you destroy the object you cannot for example reassemble the broken parts.

¹³For example a robot can move an object on a table from an *x* position to an *y* position; but it may happen that the object has a given shape and if it falls on a side instead of staying on its base this change could bring perturbations of the trajectory and this change could require an adaptation of the robot movements that need to be taken into account. In general the plan is made in an ideal world while the execution in the real world can present many variables.

¹⁴It is a plan that is based on logic, we will see some basic concepts of logic needed to understand the concepts here.

¹⁵We will see all the search strategies seen so far for solving a planning problem and we will see 2 possible configurations of the search space.

- the initial state is a priori *fully known*;
- the plan execution is the *only cause of changing in the world*.

It is opposed to reactive planning.¹⁶

Planning as Search

Let's start with this. We will see that indeed planning is a search activity; you can see planning in many different ways, but always has a search strategy behind. So, planning can be seen as a search activity.

Many different views of planning as search: states and operators change.

- *Deductive planning as theorem proving*: states are sets of propositions that are true in a given state of the world, operators are deductive rules.
- *Planning as search in the state space*: states are sets of propositions that are true in a given state of the world, operators are actions.
- *Planning as search in the plan space*: states are partial plans and operators are plan refinement/completion moves.¹⁷

Linear Planning

When we talk about linear planning we consider a linear plan as a complete and totally ordered set of actions: you order actions linearly; then we will see non-linear planning where the plan can be a partially ordered set of actions.

The search algorithm could proceed:

- *forward*:¹⁸ if search starts from the initial state and proceeding when it finds a state that is a superset of the goal;
- *backward*:¹⁹ if search starts from the goal and proceeds backward until it finds a state that is a subset of the initial state.

¹⁶Reactive planning does not rely on any description of the initial world but it simply reacts to a stimulus.

¹⁷You start with an empty plan.

¹⁸You can build a search tree in 2 ways: the most intuitive is that I start from the initial state and apply all possible operators. When I stop? When I find a state which is a goal, that contains a goal; I have a final state where I have a description of this state that has many different properties but the important thing is that among these properties that are true there is the goal that I'm looking for, so the *state* is a *superset* of the goal. Looking at the Figure below we can see that we start with a description of the initial state. In the initial state there are many operators that are applicable: how can I define if an operator is applicable? I just check that the *precondition* of this action are true in the *initial state*; if they are, I can execute the action.

¹⁹The other way. Some search/plan are too goal oriented/goal focused, so I would not really explore all possible plans that I can do starting from the initial state. Notice that I proceed backward when I'm offline (I'm searching for the plan). When the plan is found then it is applied by the executor from the *initial state* to the *goal* (so the direction is again inverted). When we stop going backward? Do you think we need all the properties of the initial state? A superset of the initial state? Need we all the description of the *initial state*? No! When we reach a subset of the initial state we can stop: because when we are in the execution phase in which we start from the *initial state* (remember what we said above) then we already have/contain a state that is going to lead us to the goal. Look at the Figure below. What do you think we can apply here as operators? Operators are actions that need to be

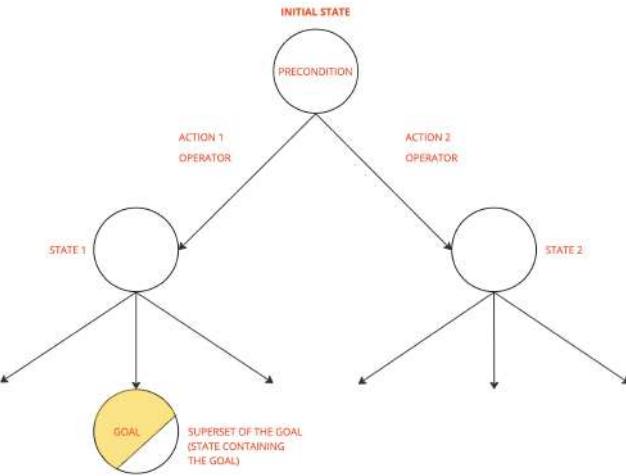


Figure 6.5: Forward Planning.

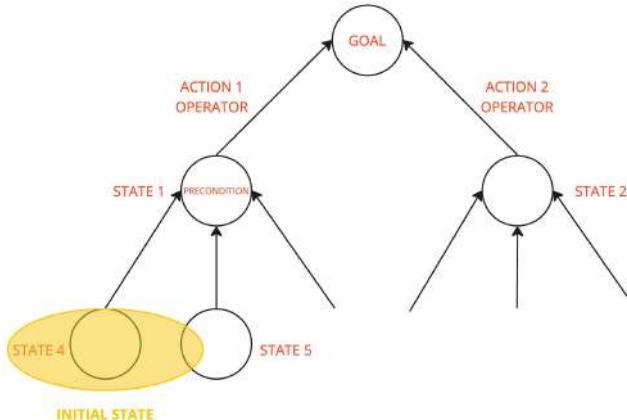


Figure 6.6: Backward Planning.

Planning as Forward Search

Refer to the image below.

Planning as Forward Search is not computationally affordable for large search spaces.

We need ways for pruning the search space or heuristics to guide the search.

inverted in the sense that *I can apply an action here if in its effect there is the goal*: among all the effects of the action I can find my goal. In a state I have/write preconditions. So in this backward procedure, *preconditions become other goals* because when you are in the execution phase and you execute Action 1 you need to have preconditions that are feasible with the main Goal.

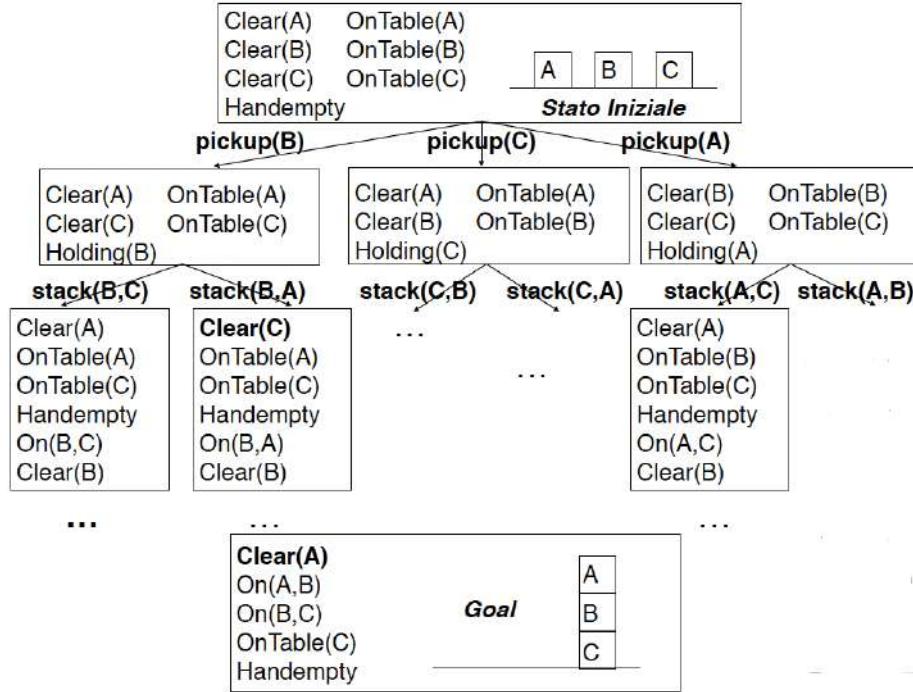


Figure 6.7

Planning as Backward Search

Backward mechanism is more goal driven; notice that we are not *executing* action in a reverse way, we just consider actions in a reverse way only in the abstract world in the generative planning, so during the planning. For doing that *i.e.* starting from the goal and applying ‘reverse actions’ to find the pattern we want, we need to apply this technique that is called *Goal Regression*.

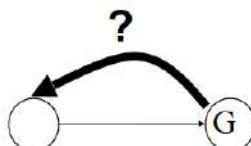


Figure 6.8

Goal Regression: it is a mechanism to reduce a goal in subgoals during search by applying rules (actions).²⁰

Given a goal G and a rule R , we have three lists:²¹

²⁰Basically you have a goal and we need to create a previous *state* starting from the *Goal* and an *Operator/Rule/Action*.

²¹As we have seen every Action/Operator has Preconditions and Effects, but the effects can be either positive or negative (*e.g.* if I put down an object on the table I have that as a positive thing means that it becomes *true* that is on the table, but becomes *false* the fact that I’m holding the object: the effect are ‘*not holding the object and the object is on the table*’).

- PRECOND: Plist;
- DELETE: Dlist;
- ADD: Alist.

Regression of G through R indicated as $\text{Regr}[G, R]$ is:²²

- $\text{Regr}[G, R] = \text{true}$ if G in Alist;
- $\text{Regr}[G, R] = \text{false}$ if G in Dlist;
- $\text{Regr}[G, R] = G$ otherwise.

Let us make an example.

Example

Suppose we have the action `unstack` that picks an object and removes it where it was on top of another object.

If we want to reach `holding(b)` in the action `unstack`, since `holding` is in the `ADD: Alist` (when I unstack an object I necessarily hold it in my hands) of course the regression of this `holding(b)` with the `unstack` is `true`.

Let's see more schematically:

- Given R1: `unstack(X, Y)`:
 - PRECOND: `handempty`, `on(X, Y)`, `clear(X)`
 - DELETE: `handempty`, `on(X, Y)`, `clear(X)`
 - ADD: `holding(X)`, `clear(Y)`
- We have:
 - $\text{Regr}[\text{holding}(b), R1] = \text{true}$ ²³
 - $\text{Regr}[\text{handempty}, R1] = \text{false}$ ²⁴
 - $\text{Regr}[\text{ontable}(c), R1] = \text{ontable}(c)$ ²⁵
 - $\text{Regr}[\text{clear}(c), R1] = (Y = c) \text{ or } \text{clear}(c)$ ²⁶
 - * if $Y = c$ then $\text{Regr}[\text{clear}(c), R1] = \text{true}$ with $R1 = \text{unstack}(X, Y = c)$.²⁷
 - * else $\text{Regr}[\text{clear}(c), R1] = \text{clear}(c)$.²⁸

Looking at the image below: if I have 3/3 preconditions and one is reached by a given action as an effect and the other 2 are untouched by the action of course these 2 should be re-brought in the *state* otherwise I'll lose them. It will be clearer with the next example!

²²This is how I can apply Goal Regression through an operator R that has `DELETE: Dlist` – which means those propositions that are made *false* by the actions (like ‘not holding the objects’) – and `ADD: Alist` – which means those propositions that are made *true* by the actions (like ‘the object is on the table’).

²³Follow the explanation above.

²⁴If I want to reach `handempty` of course it is `false` because if I perform `unstack` then `handempty` becomes `false`.

²⁵And if it is `ontable(c)` which is something that is not touched by the `unstack` then it will brought back by this goal regression. See the figure below.

²⁶It means that we should have `or Y = c` which leads to the `true` condition or `Y != c` and `X != c` which confirm `clear(c)` because if we had `Y != c` and `X = c` we would have `false` and the pattern would not be good.

²⁷Because if I unstack something `X` from `c` it becomes `true` that `c` will be clear performing

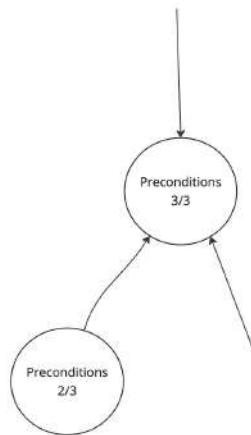


Figure 6.9

Example

Notice that in the exam are proposed exercises that require the following mechanism of resolution.

Initial State:
`clear(b), clear(c), on(c, a)`
`handempty, ontable(a), ontable(b)`

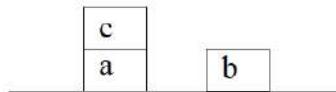


Figure 6.10

Goal: `on(a, b), on(b, c)`

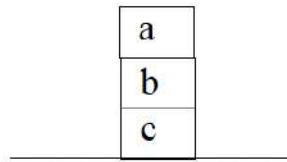


Figure 6.11

such action.

²⁸If $Y \neq c$ we can have other 2 cases: the first one would be if $X \neq c$ and $Y \neq c$ then the atom `clear(c)` it is not touched by the action `unstack(X, Y)` so we confirm `clear(c)`; the second one would be if $X = c$ and $Y \neq c$ then the atom `clear(c)` it is not reached by the action `unstack(X, Y)` so we should have `holding(c)` and `clear(c)` should belong to `Dlist` and `Regr[clear(c), R1] = false` (we should add an `elif` in the part above with this case), so we have a `false` and we need to put away the pattern.

So we have the following.

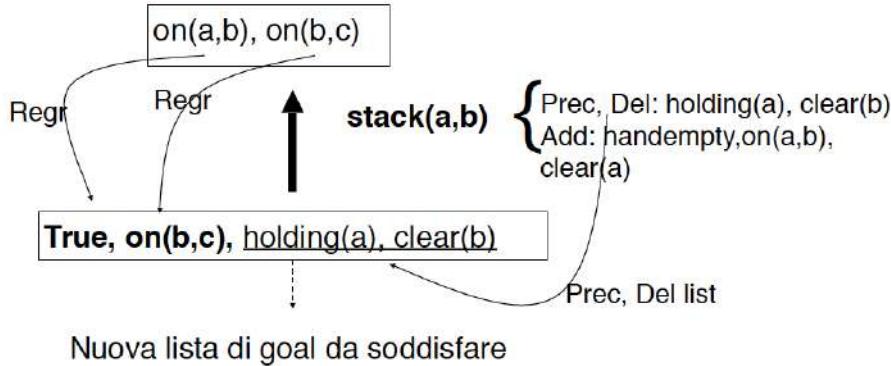


Figure 6.12

So basically I have to create the previous state starting from `stack(a, b)` (the operator) that reaches just one of the 2 components of the goal *i.e.* `on(a, b)` and `on(b, c)`; in *linear planners*²⁹ I need to define from which one I need to start first. Here in the example we start from `on(a, b)` and for reaching `on(a, b)` I have to use the action `stack(a, b)` because is clearly the only one that is able to achieve the `on(a, b)`. So how do I create this previous state? You create it through *goal regression* that means that: `on(a, b)` is away because I reached it with the action `Regr[on(a, b), stack(a, b)] = true` so it's done; then I have to put all the preconditions of the `stack(a, b)` action³⁰ (and the preconditions are `holding(a)` and `clear(b)` that correspond to the `Dlist`) and then `on(b, c)` is not touched by `stack(a, b)` action is neither in the `Alist` nor in the `Dlist`, so it is not changed by this action: so I will shift it back to the previous state;³¹ why? Because now I have to reach `on(b, c)` with the action `stack(b, c)` for example.

Looking at the figure above we have the `True` value reached through `Regr` to `on(a, b)`; `True` in the sense that we can remove it: you can always remove a `True` atom from conjunction of propositions because it does not change the total truth value of the conjunction: `on(a, b)` is done, you've achieved with the action `stack(a, b)`.

²⁹Linear planners have problems with the so called ‘interacting goals’. Consider the question: if I have to put `b` on top of `c` and then `a` on top of `b`, then it’s important that we first perform `b` on top of `c` and then `a` on top of `b`; now this is a thing that an human knows but a planner does not. First you need to do something and then something else and if you invert the order then it is a mess, because if you first put `a` on top of `b` then you have to remove `a` from `b` and put `b` on top of `c` but you think you have already achieved `a` on top of `b`, instead you didn’t. This is the main problem of linear planners: interacting goals. How planners solve this situation? They don’t solve this situation in the sense that there would be a planner that might fail. One could ask if we can give a priority, the answer is no because we’re using search algorithms and these are general purpose algorithms, remember that we are not using any domain knowledge; of course if you have a planning problem in a given domain you might want to add some domain knowledge but in this course we will see general purposes planners so they don’t have any knowledge on the domain they’re solving.

³⁰Because if we think to the execution phase we need to have all the preconditions of an action if we want to perform that action.

³¹In general I need to bring back to the previous state those atoms/propositions that are not touched by the actions. You need to reach them with other actions, they’re not yet `True`.

Backward algorithm

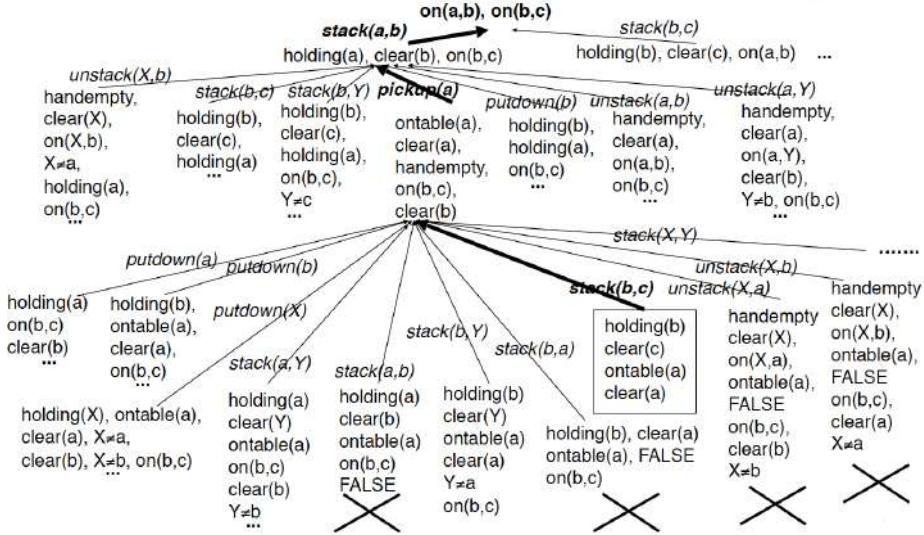


Figure 6.13

Backward algorithm

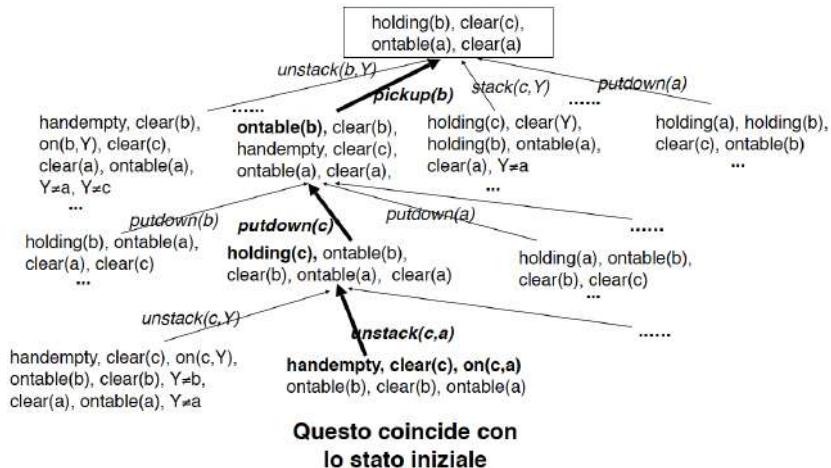


Figure 6.14

Deductive Planning

Another way of solving planning through a general purpose method – that is not tailored on planning – is to use *Logics* for representing anything.

Deductive planning uses logics for representing states, goals and actions and generates a plan as a theorem proof.³² Green³³ and Kowalsky³⁴ formulations.³⁵

Situation Calculus

This is the description of how *Green* proposed to formulate a planning problem as a logic theory.

First order logic to describe states and clauses to describe actions.

- *Situation*: world snapshot describing properties (*fluents*) that hold in a given state s .

– Example: block world.

```
on(b, a, s)
ontable(c, s)
```

Comparing to the previous planning:

- * $\text{on}(b, a, s) \leftrightarrow \text{on}(b, a)$ is **true** in the state s .
- * $\text{ontable}(c, s) \leftrightarrow \text{ontable}(s)$ is **true** in the state s .

So in comparison to the previous planning we changed the notation considering a property in a given state s .³⁶ So *fluents* with a *state*.

- *Actions*: define which *fluents* are true as a consequence of an action.

```
on(X, Y, S) and clear(X, S) ->
  (ontable(X, do(putontable(X, S))) and
   (clear(Y, do(putontable(X, S))))
```

How can we describe actions? If in a state S you have that X is on top of Y and X is clear then what? You can apply for example an action that is called *putontable* that makes 2 actions basically (*unstack*(X), *putdown*(X)) in the state S ; better explained you say: if, in state S , X is on top of Y and X is clear then you have that X will be on the table in the state derived by applying the action *putontable*(X) in S and you will also have that Y (you've removed X from Y) will be clear in the same state where you apply *do(putontable(X, S))*. You can express action in this way!

Another example:



Figure 6.15

³²Since with logics we can prove logical consequences.

³³Inefficient.

³⁴Widely used.

³⁵We will see them.

³⁶An additional parameter that describes the state in which the property is **true**. So, for example, in the initial state that we can indicate as s_0 and we have all the propositions that are **true** in the initial state. You can start from this.

Where the arrow indicates the implementation of the following:

– `do(stack(X, Y), S) := S1`

Determining a new state $S1$ obtained applying `stack(X, Y)` in the state S .

Notation:

```
holding(X, S) and clear(Y, S) ->
  (on(X, Y, S1)) and
  (clear(X, S1))
```

Deductive planning

Green uses *situation calculus* to build a planning based on logic resolution. He finds a proof of a formula containing a state variable. At the end of the proof the state variable³⁷ will be instantiated to the plan to reach the objective.

Example

These axioms describe propositions that are true in the initial state $s0$.

A.1 `on(a,d,s0)`

A.2 `on(b,e,s0)`

A.3 `on(c,f,s0)`

A.4 `clear(a,s0)`

A.5 `clear(b,s0)`

A.6 `clear(c,s0)`

A.7 `clear(g,s0)`

A.8 `diff(a,b)`³⁸

A.9 `diff(a,c)`³⁹

A.10 `diff(a,d)`...

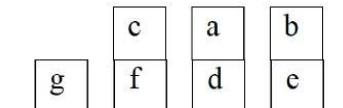


Figure 6.16

We remind that considering the formalization adopted by this formulation we have:

³⁷At the end of the proof the state variable describes the sequence of actions *e.g.* `do(stack(X, Y), S)` etc.

³⁸Mean a different from b

³⁹Mean a different from c

- capital letters (X, Y, \dots) indicate *variables*;
- lowercase letters (a, b, \dots) indicate *constants*.

Actions are clauses:⁴⁰

`move(X,Y,Z)`

```
clear(X,S) and clear(Z,S) and on(X,Y,S) and diff(X,Z) ->
  clear(Y,do(move(X,Y,Z),S)), on(X,Z,do(move(X,Y,Z),S))
```

moves a block X from Y to Z , starting from a state S and ending in a state $do(move(X,Y,Z),S)$ that translates in the following axioms (effect axioms):⁴¹

A.11 $\sim clear(X,S)$ or $\sim clear(Z,S)$ or $\sim on(X,Y,S)$ or $\sim diff(X,Z)$
or $clear(Y,do(move(X,Y,Z),S))$

A.12 $\sim clear(X,S)$ or $\sim clear(Z,S)$ or $\sim on(X,Y,S)$ or $\sim diff(X,Z)$
or $on(X,Z,do(move(X,Y,Z),S))$

So the thing that you do very intuitively – but that we will study formally in the courses that treat Logics – is the following: for proving a theorem (or goal) in first order logic I negate a theorem (or goal) and I put the negation together with our background knowledge, the knowledge that we have; if we find a contraddiction \Rightarrow this implies that the goal is true and it is proved.

Given a goal we find a proof through the resolution process.

1. Take the goal: GOAL:- `on(a,b,S1);`
2. Negate the goal: $\sim on(a,b,S1);$
3. This leads to a contraddiction (basically *proof by confutation*):

(A.12) { $X/a, Z/b, S1/do(move(a,Y,b),S)$ }

```
 $\sim clear(a,S)$  or  $\sim clear(b,S)$  or  $\sim on(a,Y,S)$  or  $\sim diff(a,b)$ 
(A.4), (A.5), (A.1), (A.8)
{S/s0}, {S/s0}, {S/s0, Y/d}, true
```

⁴⁰The action `move(X,Y,Z)` represents a new move that we've not described before. In general we can define any kind of move/action; the important thing is that your executor is able to do that, able to perform this move. Here we are just defining an action more concise since represents basically the combination of `stack/unstack` that we're able to do. So, reading also the code below, the action `move(X,Y,Z)` tells you: if in the state S we have that X is clear, Z is clear, X is on Y and $X \neq Z$ then I will obtain a new state that is the one where I start from S and I apply the action `move(X, Y, Z)` and in this new state Y is clear and X is on top of Z .

⁴¹We have not yet done *resolution* in first order logic (for example with Prolog) anyway consider the following: the action `move(X, Y, Z)` is a clause and basically we can transform it in two clauses (A.11 and A.12, note the tilde inside the code indicates negation) that are '*disjunction of positive and negative literals*'. In logic, *literals* refer to the atomic components of logical expressions; specifically: *positive literals* \rightarrow a positive literal is a variable that is directly asserted to be true, for example P is a positive literal; *negative literals* \rightarrow a negative literal is a negated variable meaning it is asserted to be false, for example $\neg Q$ (read as 'not Q ') is a negative literal. In propositional logic, literals are the simplest components that can either stand alone or be combined with other literals using logical connectives (like `and`, `or`, `not`) to form more complex logical expressions. Literals are the building blocks from which propositional formulas are constructed.

So $\neg \text{on}(a, b, S_1)$ leads to a contradiction and we have a proof for $\text{on}(a, b, S_1)$ with the substitution $S_1/\text{do}(\text{move}(a, d, b), s_0)$. This is our plan!

Now suppose we want to solve a more complex problem:⁴²

- Goal: $\text{on}(a, b, S), \text{on}(b, g, S)$;
- Solution: $S/\text{do}(\text{move}(a, d, b), \text{do}(\text{move}(b, e, g), s_0))$. Where:
 - first state: s_0
 - second state: $\text{do}(\text{move}(b, e, g), s_0)$
 - third state: $\text{do}(\text{move}(a, d, b), \text{do}(\text{move}(b, e, g), s_0))$

To solve this problem we need a complete description of the state after each move.

Situation calculus

Three points to consider.

- *Plan construction:* deduction, goal proof.
 - Example: $\text{:- ontable}(b, S)$ means: does a state S exist in which $\text{ontable}(b)$ is **true**? Yes with $S = \text{do}(\text{putontable}(b, S*))$.
- *Advantages:* high expressivity, can describe complex problems.
- *Limitations:* frame problem.⁴³

Frame problem

Let us consider again:



Figure 6.17

Where the arrow indicates the implementation of the following:

- $\text{do}(\text{stack}(X, Y), S) := S_1$

Determining a new state S_1 obtained applying $\text{stack}(X, Y)$ in the state S . Notation:

```
holding(X, S) and clear(Y, S) ->
  (on(X, Y, S1)) and
  (clear(X, S1))
```

⁴²Here we're asking: is there any state S in which we find $\text{on}(a, b)$ and $\text{on}(b, g)$? The answer of this resolution is 'yes' if S is a state where you apply to s_0 a $\text{move}(b, e, g)$ and to the resulting state a $\text{move}(a, d, b)$. So notice that this concatenation of do is in fact our plan. So just to say that we've a general method – using logic – for solving planning problems. It is efficient? No, we will see.

⁴³Let's investigate it.

If we have a state S and we create a new state like for example `do(stack(X, Y), S)` there is a little problem in Logic: the problem is, if besides `holding(X, S)` and `clear(Y, S)` in the state S we have also other propositions that are not touched by the action, we have to explicitly state everything that is not touched by the action again in the next state.

So we can resume the problem in the next three points:

- knowledge representation problem;
- we have to explicitly list all fluents that change and that DO NOT change after a state transition;
- if we have a complex domain the number of these axioms grows enormously.

Example

Let's complete the previous example.

To describe an action, beside effect axioms we have to specify all fluents that would not change by the execution of an action (frame axioms). In our example we would need:

```
on(U, V, S) and diff(U, X) -> on(U, V, do(move(X, Y, Z), S))
clear(U, S) and diff(U, Z) -> clear(U, do(move(X, Y, Z), S))
```

Regarding:

```
on(U, V, S) and diff(U, X) -> on(U, V, do(move(X, Y, Z), S))
```

Consider: if U is on V in the state S and this condition is not touched by the action `move(X, Y, Z)` in the same state, then we need to bring this untouched condition also in the next state.

We need a frame axiom for each condition that is not changed by each action; this would imply that \Rightarrow if the problem is complex, too many axioms.

Kowalsky formulation

Therefore Kowalsky proposed a formulation that is a lot more efficient still using Logics and it is called *Event Calculus*; Kowalsky is one of the two inventors of Prolog language (one of the most common language used by researchers in Logic and AI).

Let's see the master keys of this formulation:

- we use the predicate `holds(rel, S/A)`⁴⁴ to describe all the relations `rel` that are true in a given state S or made true by the execution of an action A ;
- predicate `poss(S)` that indicates if a state S is possible (namely reachable);⁴⁵
- predicate `pact(A, S)` to indicate that it is possible to execute an action A in a state S , namely the preconditions of A are true in S .⁴⁶

⁴⁴In this way we don't have so many predicates as before e.g. `holding`, `clear`, `on`, ... but only one that is `holds`.

⁴⁵The initial state is always reachable, we're already there.

⁴⁶`pact` means *preconditions-actions*.

If a state S is possible and the preconditions of an action A are satisfied, then it is possible the state produced by S after the execution of A :

```
poss(S) and pact(A,S) -> poss(do(A,S))
```

We need *one frame assertion per action* (advantage w.r.t Green).⁴⁷

In the previous example we would have:⁴⁸

```
holds(V,S) and diff(V,clear(Z)) and diff(V,on(X,Y)) ->
    holds(V,do(move(X,Y,Z),S)))
```

That states that every term different from `clear(Z)` and `on(X,Y)` (*delete list*) are true after the execution of the action `move`.⁴⁹

Let's see better in an example.

Example

When your knowledge about Prolog will be enough we suggest to do this example/exercise in Prolog because offers a very compact representation.

Given the Goal:⁵⁰

```
Goal
:- poss(S), holds(on(a,b),S), holds(on(b,g),S)
```

And the initial state:⁵¹

Initial State

```
poss(s0)
holds(on(a,d),s0)
holds(on(b,e),s0)
holds(on(c,f),s0)
holds(clear(a),s0)
holds(clear(b),s0)
holds(clear(c),s0)
holds(clear(g),s0)
```

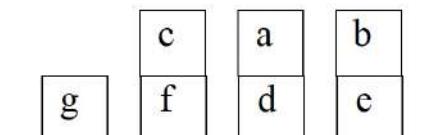


Figure 6.18

⁴⁷While in the Green formulation you have to write one frame condition per property.

⁴⁸With the Kowalsky formulation we would have this kind of logics; go ahead, it will be clearer.

⁴⁹So the previous formulation avoid an entire list.

⁵⁰The Goal is (see below): is it possible to find an S such that of course it is reachable `poss(S)`, and where `holds(on(a,b))` and `holds(on(b,g))`? Let's see if it is possible.

⁵¹I start with this initial state (see also the figure below). Note that the formulation adopted here is very different from the Green formulation/representation: `on(a, d, s0)` has 3 arguments while `holds(on(a, d), s0)` in which `on(a, d)` becomes a *term* (*i.e. term* \leftrightarrow argument of a predicate); '`on(a, d)` holds in $s0$ ' and so on... we have a description of the initial state in this way.

Use Prolog to solve it.

Let's see the action `move(X, Y, Z)` with this new formulation. Remember – with an high level description – with the `move(X, Y, Z)` we have that X is on top of Y, Z is clear and so we pickup X and put it on top of Z. That are translated in the *lower language* in the following.

- Effects of `move(X,Y,Z)`:⁵²

```
holds(clear(Y), do(move(X,Y,Z), S))
holds(on(X,Z), do(move(X,Y,Z), S))
```

- Preconditions of `move(X,Y,Z)`:⁵³

```
pact(move(X,Y,Z), S) :-  
    holds(clear(X), S), holds(clear(Z), S), holds(on(X,Y), S), X \= Z
```

- Frame conditions:⁵⁴

```
holds(V, do(move(X,Y,Z), S)) :- holds(V, S), V \= clear(Z), V \= on(X, Y)
```

- Clause for state reachability:⁵⁵

```
poss(do(A,S)) :- poss(S), pact(A,S)
```

- Goal:⁵⁶

```
Goal  
:- poss(S), holds(on(a,b), S), holds(on(b,g), S)
```

Yes per `S = do(move(a, d, b), do(move(b, e, g), s0))`

Observation for the exam

In the exam we will have a *planning exercise* that will be solved with a specific algorithm, specifically either *strips* or *pop* that we are going to see.

In the theoretical part it will be also asked: it will be taken an action and it will be asked to describe with the Kowalsky notation such action.

So let's do an example of the exercises that are likely to be proposed during the exam.

⁵²Effects of the move are facts; they are not contained in a clause. `holds(clear(Y), do(move(X,Y,Z), S))` it is read: in the state resulting from applying the `move(X,Y,Z)` in the state `S` it is true that `clear(Y)`.

⁵³This is read: in the state `S` the preconditions of the action `move(X,Y,Z)` are true if `clear(X)`, `clear(Z)`, `on(X, Y)` and `X` is different from `Z` are true.

⁵⁴But then we still need to write *Frame actions* because we are in Logic and therefore if you do not explicitly state that a given proposition is true in a state it means that is false. This is called ‘clause word assumption’: everything that you state is true and what you don't state is false (this is first order logic). We have to write one frame condition per action while in the Green formulation you have to write one frame condition per property. So as frame condition we can say that: a condition `V` holds in a state where from `S` I apply `move(X,Y,Z)` if `V` was hold before (in `S`) and it is not `clear(Z)` and `on(X, Y)`; why? Because `clear(Z)` and `on(X, Y)` are in the *delete list* of `move`, so the only thing that are deleted are `clear(Z)` and `on(X, Y)`, all the rest is kept, it is broad forward.

⁵⁵Then of course we have a clause for the reachability: if a given state `S` is reachable and the precondition of an action `A` are true in the state `S` then it is reachable also the state where I apply `A` to `S`.

⁵⁶Is it possible a state `S` where I have both `on(b, g)` and `on(a, b)`? Yes for `S` obtained applying `move(a, d, b)` to a state obtained applying `move(b, e, g)` to the initial state `s0`.

- Describe the UNSTACK(X, Y) action with the Kowalsky notation.

UNSTACK (X, Y)

IF: handempty and clear (X) and on (X, Y)

THEN: holding (X) and clear (Y);

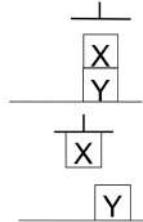


Figure 6.19

Kowalsky description?

Let's see a real example taken from an exam.

Planning exercise

In the initial state described by the following atomic formulas:

[at (me, home), in (book, shelf), at (shelf, home), at (table, home), in (coke, fridge), at (fridge, home), handempty, off (stereo), electronic_equipment (stereo)]

you want to reach the goal:

have (me, book), on (stereo), ontable (coke)

- the actions are modeled as follows:

take (Me, Item)

PRECOND: handempty, at (Me, Location), in (Item, Box), at (Box, Location)
DELETE: handempty, in (Item, Box)

ADD: have (Me, Item)

put_on_table (Me, Item)

PRECOND: have (Me, Item)
DELETE: have (Me, Item)
ADD: handempty, ontable (Item)

switch_on (ElectrEq)

PRECOND: electronic_equipment (ElectrEq), off (ElectrEq)
DELETE: off (ElectrEq)
ADD: on (ElectrEq)

Solve the problem using the POP algorithm. The causal links and the threats encountered are highlighted.

Figure 6.20

As you can see it is asked to solve this with a given algorithm (in this case POP algorithm). Now we have not yet covered this part so we are not interested in it. We are instead interested to the following exercise that is related to the one above.

Exercise 5

- 1) Model the action **take** (preconditions, effects and frame axioms), the initial state and the goal of exercise 4 using the Kowalsky formulation
- 2) Show two levels of graph plan when applied to exercise 4.
- 3) What is conditional planning and what are its main limitations?
- 4) What is Particle Swarm Optimization and which are its main features?
- 5) What is modal truth criterion and why it has been defined.

Figure 6.21

Let's see the resolution.

- Initial state is very simple, basically is *holds* (*every atoms/propositions listed above*) in the initial state that we call **s0** (note **s** is lower case).

```
holds(at(me, home), s0)
holds(in(book, shelf), s0)
...
holds(electronic_equipment(stereo), s0)
```

- Now let's consider the **take** action → the **take** action has one effect so we have only one fact to describe the effect:

```
EFFECTS: holds(have(Me, Item), do(take(Me, Item), S)) ->
          In any state S where I apply the take(Me, Item),
          the resulting state holds have(Me, Item)
          (positive effect, ADD list)
```

- Now let's consider the *preconditions* action and let's remember that:

```
pact(move(X, Y, Z), S):-  
  holds(clear(X), S), holds(clear(Z), S), holds(on(X, Y), S),  
  X \= Z
```

It is read: the preconditions action in a given state are true if in that state the needed preconditions are hold.

```
PRECONDITIONS: pact(take(Me, Item), S) :-  
  holds(handempty, S), holds(at(Me, Location), S),  
  holds(in(Item, Box), S), holds(at(Box, Location), S)
```

```
preconditions action of take(Me, Item) in a state S holds  
if S holds all the preconditions of the action  
(so refer to the PRECOND list)
```

- Now the frame axioms: only one. Why? Because it is asked to model the action **take** and for each action we have one frame axiom. The frame axiom what should look at? At the *DELETE list*: the only thing that should become FALSE in the next state after the execution of the **take** action are in the *DELETE list i.e. handempty and in(Item, Box)*. All the rest, all the remainings should be there. Remember for the **move(X, Y, Z)** we had:

Frame conditions:

```
holds(V, do(move(X, Y, Z), S)):- holds(V, S),
V \= clear(Z), V \= on(X, Y)
```

Every property will hold after the execution of the `move` if it was hold before and it is not in the *DELETE list* → this is what the *Frame Condition* is telling.⁵⁷

FRAME AXIOMS:

```
holds(V, do(take(Me, Items), S)):- holds(V, S),
V \= handempty, V \= in(Item, Box)
```

Holds any property `V` in the state where I applied `take(Me, Items)` in a state `S`, if `V` was hold before the application of the action and `V` is not in the *DELETE list*

- Now remains the goal: the goal has all the `holds` that we want to reach, but also the `poss` that indicates the reachability.

```
GOAL:- poss(S), holds(have(me, book), S),
       holds(on(stereo), S), holds(ontable(coke), S)
```

6.2 Practising with Deductive Planning

Let us get confident with some aspects of *deductive planning* practising with some exercises.

Deductive Planning

Let us consider the following initial state expressed with the Kovalsky formulation.

```
holds(on(a,d),s0)
holds(on(b,e),s0)
holds(on(c,f),s0)
holds(clear(a),s0)
holds(clear(b),s0)
holds(clear(c),s0)
holds(clear(g),s0)
```

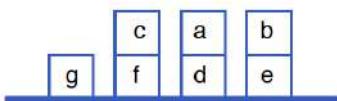


Figure 6.22

⁵⁷Note that the `\=` represents the Prolog notation for ‘not’.

Consider the action `move(X, Y, Z)`.⁵⁸



Figure 6.23

Where in the figure above on the left we have *Preconditions*:

```
on(X, Y)
clear(X)
clear(Z)
```

And on the right we have the Effects of the action:

```
clear(Y)
on(X, Z)
```

If you notice, some of these *preconditions* are deleted by the action itself (if they are touched by the action [*Delete list*]); they are no longer true.

Let's proceed.

We have facts that model the effects of the action:

```
%Effects move(X, Y, Z):
holds(clear(Y), do(move(X, Y, Z), S)).
holds(on(X, Z), do(move(X, Y, Z), S)).
```

That can be read as: starting from any state S if I apply the `move(X, Y, Z)` then I obtain `clear(Y)`; *positive effects etc.*

Then we have the *frame condition*:

```
%Frame condition for move(X, Y, Z):
holds(V, do(move(X, Y, Z), S)) :-
    holds(V, S),
    V \= clear(Z),
    V \= on(X, Y).
```

The frame condition takes into account the fact that in logic – *first order logic* – if you don't explicitly state something, it is *false*. So what is not *true* is *false* (there are other theories in logic in which what is not there is *unkown*). So the thing is that everything that is not touched by an action should be moved forward in the next state through the *preconditions*. In the Green formulation we have one frame condition per property while in the Kowalsky per action which is very better since we have more properties than actions. Basically the frame condition takes into account also the *Delete list*.

The frame condition is read: if a given condition V was `true` before in a state S then it is still `true` in a state where I apply `move(X, Y, Z)` to/in S , if V is not

⁵⁸We want to model this action.

in the *Delete list*, the only properties that I don't want to carry on in the next state are those in the *Delete list*; for all the others I will keep them.

Then we have the *preconditions*:

```
% Clause for the preconditions of move(X,Y,Z):
pact(move(X,Y,Z),S) :-
    holds(clear(X),S), holds(clear(Z),S),
    holds(on(X,Y),S), X\=Z.
```

Notice: why do I have to explicitly state that X is different from Z ? Because on *prolog* if you don't state this it may happen that if I have `clear(A)` where A is a block, A could be intended both as X and Z and there would be possible the unification of X with A and Z with A , so X with Z .

Then we have the *general clause* (remember to include always the initial state).

```
%Clause for the reachability of a state:
poss(s0).
poss(do(A,S)) :-
    poss(S),
    pact(A,S).
```

Now if you want to create a plan you have to state a *Goal*:

```
%Goal:
:- poss(S), holds(on(a,b),S), holds(on(b,g),S).
```

Where `:-` (semicolon and a minus) is the notation for a *Goal* which is basically '*a clause with an empty head*'.

It is read: does any S exists such that it is reachable of course and in S we have two conditions that hold: `on(a,b)`, `on(b,g)`.

When you solve this goal you unify S with a list of composed terms like `do(Action in a state)...` till $s0$ and proceeding *backward*.

Deductive Planning: exercise

Exercise: use the block world but change actions.

The exercise is: try to model the following actions – as did before – with the Kowalsky notation. In the following pages there are the solutions; try to do on your own.

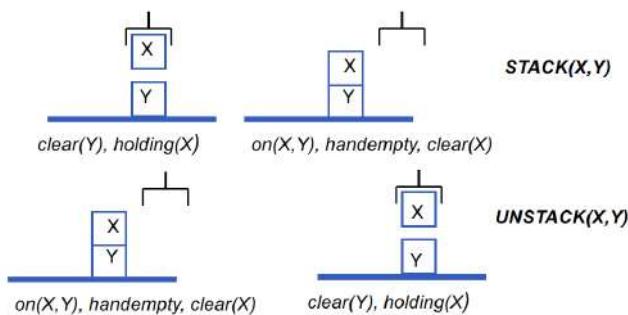


Figure 6.24

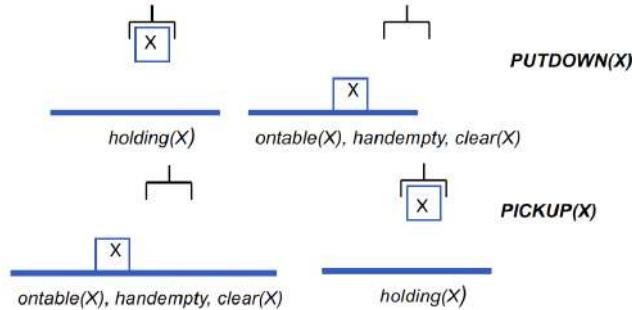


Figure 6.25

Resolution:

```
%Effects stack(X,Y):
holds(clear(X), do(stack(X,Y),S)).
holds(on(X,Y), do(stack(X,Y),S)).
holds(handempty, do(stack(X,Y),S)).

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

%Clause for the preconditions of stack(X,Y):
pact(stack(X,Y),S):-
  holds(holding(X),S), holds(clear(Y),S).

%Clause for the reachability of a state:
poss(s0).
poss(do(A,S)):- 
  poss(S),
  pact(A,S).

%Effects unstack(X,Y):
holds(holding(X), do(unstack(X,Y),S)).
holds(clear(Y), do(unstack(X,Y),S)).
holds(handempty, do(stack(X,Y),S)).

%Frame condition for unstack(X,Y):
holds(V,do(stack(X,Y),S)):- 
  holds(V,S),
  V\=clear(X),
  V\=handempty,
  V\= on(X,Y).

%Clause for the preconditions of unstack(X,Y):
pact(unstack(X,Y),S):-
```

```

    holds(clear(X),S), holds(handempty,S),
    holds(on(X,Y),S).

%Effects pickup(X):
holds(holding(X), do(pickup(X),S)).

%Frame condition for pickup(X):
holds(V,do(pickup(X),S)):- 
    holds(V,S),
    V\=clear(X),
    V\=ontable(X),
    V\= handempty.

%Clause for the preconditions of pickup(X):
pact(pickup(X),S):-
    holds(ontable(X),S), holds(clear(X),S),
    holds(handempty, S).

%Effects putdown(X):
holds(holding(X), do(putdown(X),S)).

%Frame condition for putdown(X):
holds(V,do(putdown(X),S)):- 
    holds(V,S),
    V\=holding(X).

%Clause for the preconditions of putdown(X):
pact(putdown(X),S):-
    holds(holding(X),S).

```

Deductive Planning: exercise

In general, in the exams, we have to solve a planning exercise, so one planning exercise to solve either with *strips* or *pops*, so either with *linear planning* or *not linear planning*. What is in general asked in an exam is to take an action and model it through the Kowalsky notation; sometimes also in the Green but since Kowalsky is more compact it is generally asked Kowalsky. In addition we give the *initial state* and the *goal* (which are the initial state and the goal of the entire planning exercise; see ahead). So in general is asked to model:

- initial state;
- goal;
- effects;
- preconditions;
- frame.

Model the following actions:

```
%Load of an object
load(Object ,Trolley ,Location)
PREC: at(Object ,Location) , at(Trolley ,Location)
ADD LIST: in(Object , Trolley)
DELETE LIST: at(Object ,Location)
```

Let's do a simple paraphrase.

- `load(Object ,Trolley ,Location)` \leftrightarrow you want to load an object on a trolley in a given location.
- PREC: `at(Object ,Location)`, `at(Trolley ,Location)` \leftrightarrow Preconditions are that the object should be in the location and the trolley should be in the same location too. Note that we have the same variable for `Location`, we unify.
- ADD LIST: `in(Object , Trolley)` \leftrightarrow the object is in the trolley.
- DELETE LIST: `at(Object ,Location)` \leftrightarrow the object is no more in the location.

Let's list the other actions:

```
%Transport
drive(Trolley ,Location1 ,Location2)
PREC:at(Trolley ,Location1) , connected(Location1 ,Location2)
ADD LIST: at(Trolley ,Location2)
DELETE LIST: at(Trolley ,Location1)

%Unload of an object
unload(Object , Trolley ,Location)
PREC:at(Trolley ,Location) , in(Object , Trolley)
ADD LIST: at(Object ,Location)
DELETE LIST: in(Object , Trolley)
```

With the following initial state and goal:

```
%Initial State:
in(carico1 ,carrello1) , at(carrello1 ,milano)
connected(milano ,bologna) , connected(bologna ,roma)

%Goal:
at(carico1 ,roma)
```

So:

```
%Initial State:
holds(in(carico1 ,carrello1) ,s0).
holds(at(carrello1 ,milano) ,s0).
connected(milano ,bologna).
connected(bologna ,roma).
```

Note: the `connected` property does not depend on the state.⁵⁹

⁵⁹Why? Because, in general, if you don't have any action that changes the connection between cities, that is something that cannot be taken into account as a property in a given state, it always holds independently from the state in which I am. The planner can change the `in`, `at` properties but not the `connected` one, so we can also choose to do not link with a state!

```
%Goal:  
:- holds(at(carico1,roma),S).
```

```
%Reachability  
poss(s0).  
poss(do(A,S)):-  
    poss(S),  
    pact(A,S).
```

Let's finish:

```
holds(in(Object, Trolley), do(load(Object, Trolley, Location), S)).  
pact(load(Object, Trolley, Location), S):-  
    holds(at(Object, Location), S),  
    holds(at(Trolley, Location), S).  
holds(V, do(load(Object, Trolley, Location), S)):- holds(V, S),  
    V \= at(Object, Location).  
  
holds(at(Trolley, Location), do(drive(Trolley, Location1, Location2), S)).  
pact(drive(Trolley, Location1, Location2), S):-  
    holds(at(Trolley, Location1), S),  
    connected(Location1, Location2).  
holds(V, do(drive(Trolley, Location1, Location2), S)):- holds(V, S),  
    V \= at(Trolley, Location1).  
  
holds(at(Object, Location), do(unload(Object, Trolley, Location), S)).  
pact(unload(Object, Trolley, Location), S):-  
    holds(at(Trolley, Location), S), holds(in(Object, Trolley), S).  
holds(V, do(unload(Object, Trolley, Location), S)):- holds(V, S),  
    V \= in(Object, Trolley).
```

Let's now continue with the theory.

6.3 Continuing on Planning

From now on we will see methods that are specifically for *Planning*. Of course they are methods that have the description of a state with *Logic*, with predicates; of course they will explore a search space; but they have specific data structures that are meant to simplify the process of creating a plan.

STRIPS

STRIPS – Stanford Research Institute Problem Solver.

- Specific language for the actions. Easier syntax than the situation calculus (less expressive more efficient).
- Ad hoc algorithm for the plan construction.

We are seeing linear planning like *STRIPS* because they are the basics; *STRIPS* for example are not longer used so much for doing planning in fact, but ingredients from linear planners are used to develop the new planners used nowadays.

STRIPS: language for states

State representation:

- fluents that are true in a given state.⁶⁰
- ```
on(b, a), clear(b), clear(c), ontable(c).
```

Goal representation:

- fluents that are true in the goal state;<sup>61</sup>
  - we can have variables.<sup>62</sup>
- ```
on(X, a)
```

STRIPS: language for actions

Action representation (3 lists).

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

```
Example Move(X, Y, Z)
```

```
Preconditions: on(X,Y), clear(X), clear(Z)
Delete List: clear(Z), on(X,Y)
Add list: clear(Y), on(X,Z)
```

Sometimes *ADD* and *DELETE list* are glued in an *EFFECT list* with positive and negative axioms:⁶³

```
Example Move(X, Y, Z)
```

```
Preconditions: on(X,Y), clear(X), clear(Z)
Effect List: -clear(Z), -on(X,Y), clear(Y), on(X,Z)
```

Frame problem solved with the *Strips Assumption*:⁶⁴

- everything which is not in the ADD and DELETE list is unchanged.

Let's see on real cases actions:

⁶⁰The same already seen; note below that the state is not explicitly written, there is no need for explicitly write the state.

⁶¹The same already seen; I will explain the goal as the fluents that are true in a given state.

⁶²on(X, a) below means I want to reach a state in which something – represented by the variable X is on a; so I need to have something on a but I don't mind.

⁶³Note that we use below the symbol '-' meaning \neg .

⁶⁴If I have a property which is not touched by any action it goes on in the next states.

```

pickup(X)
PRECOND: ontable(X), clear(X), handempty
DELETE: ontable(X), clear(X), handempty
ADD: holding(X)

putdown(X)
PRECOND: holding(X)
DELETE: holding(X)
ADD: ontable(X), clear(X), handempty

stack(X,Y)
PRECOND: holding(X), clear(Y)
DELETE: holding(X), clear(Y)
ADD: handempty, on(X,Y), clear(X)

unstack(X,Y)
PRECOND: handempty, on(X,Y), clear(X)
DELETE: handempty, on(X,Y), clear(X)
ADD: holding(X), clear(Y)

```

STRIPS: algorithm

Let's describe the STRIPS algorithm.

- Linear planner⁶⁵ based on backward search.⁶⁶
- Initial state fully known (*Closed World Assumption*).⁶⁷
- Two data structures:
 - goal stack;⁶⁸
 - description of the current state.⁶⁹
- Algorithm:⁷⁰

```

Initialise the stack with the goals to reach
while stack not empty do
  if top(stack) = a and aT in S
    (note that a can be an and of goals or a single one)
    then pop(a) and execute substitution T on the stack

```

⁶⁵Meaning it provides a fully ordered list of actions.

⁶⁶Because the goal stack proceed *backward* (but remember that the action execution proceeds *forward*).

⁶⁷*Closed World Assumption*: everything that is stated is true while what is not stated is false (principle of Logic); it means I know everything in my world; I have a complete knowledge; there is nothing unknown.

⁶⁸Goal stack \equiv the goal we want to reach. The *stack* data structure has the property '*last in/first out*' which is the life policy of extracting things from the stack in the sense that the last element added to the stack is the first one to be removed. See next section for more details.

⁶⁹Initially the current state \equiv initial state. So in the description of the current state we need to include the initial state s_0 which contains all the properties that are true in the initial state.

⁷⁰Just try to understand what follows intuitively.

```

else if top(stack) = a
    then Select rule R with a in Addlist(R),
        pop(a), push(R), push(Precond(R));
else if top(stack) = a1 and a2 and ... and an
    then push(a1), ..., push(an)
else if top(stack) = R
    then pop(R) and apply R on S

```

Let's analize some terms of the algorithm.

- Initialise the stack with the goals to reach \leftrightarrow the goal stack basically enables us to proceed backward (from the goal to the initial state), because we are starting from the goal and we want to find an *Action* whose effects match the goal, so as we did for *Backward Search*: want to reach the goal `on(X,Y)` then, the action `stack(X,Y)` reaches exactly that goal, so I can use `stack()`, but for using `stack` I have to upload on the goal stack what? The preconditions of the action `stack(X,Y)` and so on... Every time I extract from the stack a goal then I have to look for an action that reaches that goal. If I had more than one goal, so a conjunction of goals, I will decide an order and start with the first.
- while `stack not empty` do \leftrightarrow while the stack is not empty, if the top of the stack is a property the unifies through a given substitution with the current state then remove the property from the stack and execute the substitution on the stack (meaning that if `a` contains variables then you can unify these variables with constants and then we apply the substitution to the all stack). This happens if the goals/properties that are on the top of the stack are already true in the current state. Otherwise if they are not true in the current state, then we have to select an action/rule `R` that has the property `a` in its `Addlist` in the sense that the action needs to reach `a`, then I remove (pop) from the stack `a`, add the action `R`, add the preconditions of `R`. If the top of the stack is a conjunction then I insert/push the properties in the conjunction in any order. Finally if the top of the stack is an action, then I pop the action and I execute the action in the current state (and this is the only *forward* step).
- if `top(stack) = a and aT in S` \leftrightarrow when I extract a goal I can either find an action that reaches the goal or this goal is already true in the current state, if so (last condition) I can remove it. So the cases are:
 - I execute a goal and it's not true in the current state, so I need an action;
 - I extract a goal that is already true in the current state. I've done, I remove this goal;
 - I extract a conjunction of goals, then I will find an order and I start with the first.

I can extract an action from the goal stack (why an action? Because I said that if I extract a goal I need to execute an action and then the preconditions of the action are inserted in the goal stack because they again need to be reached). When I extract the action and that's important,

this is the only time for which I proceed *forward*, so I go in the current state and I execute the action; of course we will show an example and all will be clearer. But the main idea is that we have these two data structures: one is proceeding *backward* while the state *forward*. After seeing the next concrete example it would be clearer to understand the algorithm.

- then `push(a1), ..., push(an)` \leftrightarrow note that the order in which subgoals are inserted in stack is a non deterministic choice point. The end of goals should be verified afterward – interacting goals.

Data structure: stack

A stack is a fundamental data structure in computer science that operates on a Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. Below are presented the key characteristics and operations associated with a stack.

- Key Characteristics.
 1. *LIFO Order*: the most recently added element is the one that is removed first.
 2. *Sequential Access*: elements are accessed in a specific sequence, starting from the top.
 3. *Dynamic Size*: stacks can grow and shrink dynamically as elements are added or removed.
- Basic Operations.
 1. *Push*: adds an element to the top of the stack.
 2. *Pop*: removes the element from the top of the stack.
 3. *Peek (or Top)*: retrieves, but does not remove, the element at the top of the stack.
 4. *IsEmpty*: checks if the stack is empty.
 5. *Size*: returns the number of elements in the stack.
- Implementation example in Python (simple implementation of a stack using a list in Python).

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        if not self.is_empty():
```

```

        return self.items[-1]
    return None

def is_empty(self):
    return len(self.items) == 0

def size(self):
    return len(self.items)

# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack.peek())  # Output: 3
print(stack.pop())  # Output: 3
print(stack.pop())  # Output: 2
print(stack.is_empty())  # Output: False
print(stack.pop())  # Output: 1
print(stack.is_empty())  # Output: True

```

- Applications of Stacks.
 1. *Function Call Management*: stacks are used to manage function calls and recursion in programming languages.
 2. *Expression Evaluation*: stacks are used to evaluate arithmetic expressions, particularly in postfix notation (also known as Reverse Polish Notation).
 3. *Syntax Parsing*: compilers use stacks to parse syntax and check for balanced parentheses and other matching delimiters.
 4. *Backtracking*: algorithms that involve backtracking, such as certain puzzles and pathfinding algorithms, use stacks to remember previous states.
 5. *Undo Mechanisms*: applications like text editors use stacks to implement undo functionality.

Stacks are simple yet powerful data structures that are essential for many algorithms and systems in computer science.

STRIPS: algorithm – some notes

Let's make some important considerations:

- the problem is divided into subgoals which might interact;⁷¹
- many possible goal orderings;
- at each step we select one subgoal from the goal stack;

⁷¹And therefore we need to keep the \wedge , the conjunction on the stack, since there are many possible goal orderings and we chose non-deterministically one of this ordering.

- when we have a set of actions that reach a goal, we execute them on the state that proceeds forward;
- the process goes on until the stack is empty;
- when at the top of the stack we find an **and of goals**, we need to check that this is still satisfied in the current state before removing it; if it is not, we have to reinsert the **and** and change order.

Example

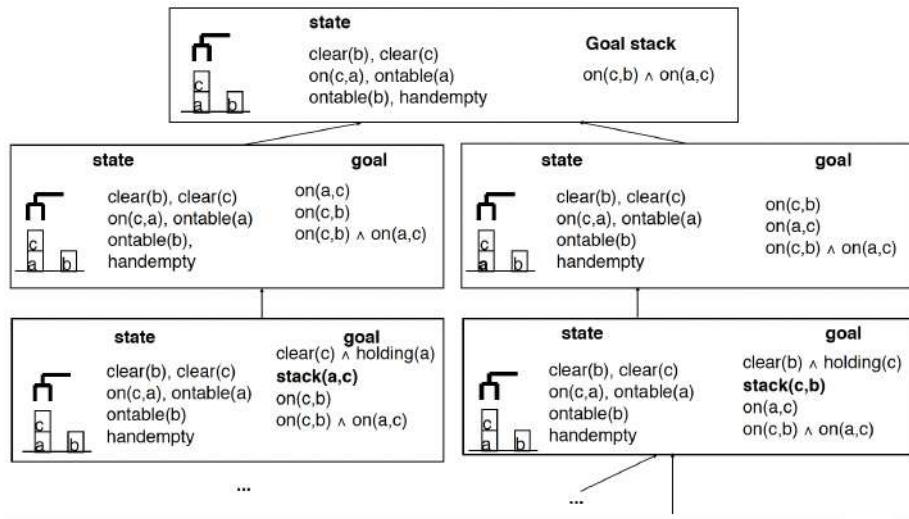


Figure 6.26

Let's consider an example.

- In the image above we start on the top with the *initial state*; in the *goal stack* $on(c,b) \wedge on(a,c)$ we need to choose an order since we have a conjunction: we choose – for example proceeding to the left in the figure – first to reach $on(a,c)$ and then $on(c,b)$.
- Proceeding on the left (second level), when I extract $on(a,c)$ I ask myself what should I do for reaching $on(a,c)$? First of all I need to check if $on(a,c)$ is in the state; in our case no! So, then I have to find an action that reaches that specific goal.
- The action is **stack(a,c)** – see the third level going down to the left – (we will write actions in bold, all the non-bold things are *preconditions* or *goals*). When I load on the stack the *action*, I have to load on the stack also its *preconditions* because I want to be able to execute these actions.
- Symmetrically on the right, let's go on with this other part. This is a linear planner meaning that it will find a single chain/path of actions that are linearly totally ordered; so sometimes the order with which I execute the action is crucial because it could happen that with one order I will not find

anything while in the other order I will find the perfect plan. So goals are in general interacting; it may happen they're interacting and then for the order could be important; in our case we have a non-deterministic choice point, in one part I use one order and in the other the opposite order. Note that the *state* has not changed. Every time I have a conjunction of goals I should choose an order (see on the third level to the bottom on the right $\text{clear}(b) \wedge \text{holding}(c)$).

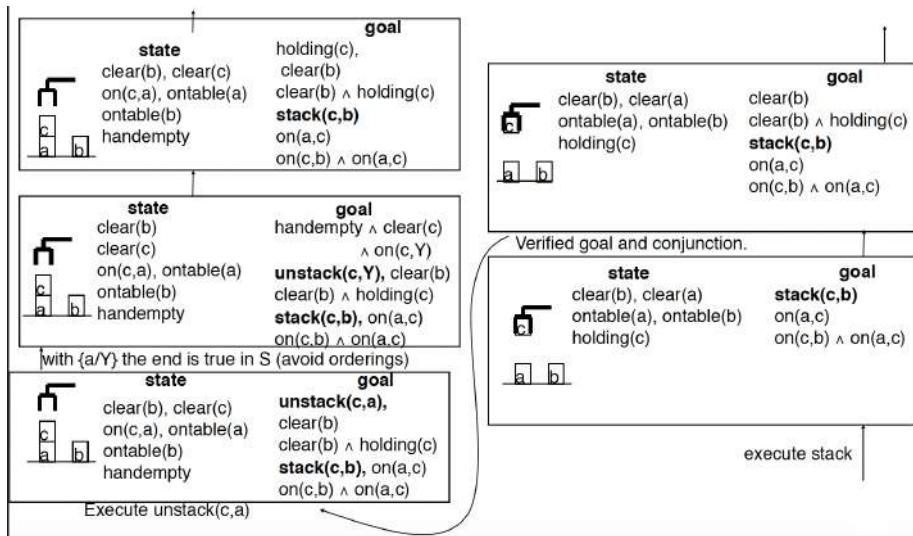


Figure 6.27

Let's continue.

- Continuing on the top left of the figure above, here I chose **holding(c)** first, **clear(b)** then; but I could do the opposite.
- Continuing on the left second level: to reach **holding(c)** I have to use the action **unstack(c,Y)**; why we have this Y? Because we don't know what is below of c. Of course we add also the preconditions of the action.
- Between the second and the third level on the left we have a sentence meaning that the preconditions are true with Y = a in the state S so I can remove them and putting Y = a inside the action.
- Continuing on the left third level: the only time in which a state changes is when you have an *action* coming out from the *goal stack* meaning that all its preconditions are true in the current state and now I can execute the action forward. Note here is the first time I can execute an action in the *forward* direction, note when I execute the action the state changes in relation with the *Add/Delete list* of the action.
- Continuing on the right: **clear(b)**, **clear(b) \wedge holding(c)** are true in the current state, I remove them because I can execute the action. Note: why we leave always the conjunction (**clear(b) \wedge holding(c)**)? Because

it may happen that for reaching the second condition we will destroy the first; but we need both simultaneously (*interacting goals*).

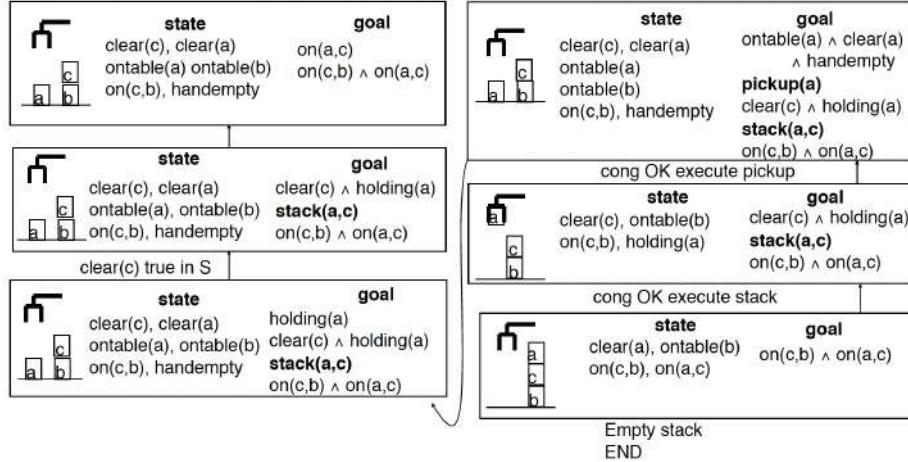


Figure 6.28

Let's continue.

- Continuing on the top left of the figure above: having executed the action **stack(c,b)** the state has changed. Now we have to reach **on(a,c)** (note that **on(c,b)** is already true).
- Between the second and the third level on the left there is **clear(c) true in S**; since this is true, it suggests an order.

The solution is:

1. **unstack(c,a);**
2. **stack(c,b);**
3. **pickup(a);**
4. **stack(a,c).**

Now let's go back to the algorithm.

STRIPS: pittfalls

There are some pittfalls to consider.

1. *Very large search space.*

- In the example we have seen a single path but there are many alternatives:
 - non deterministic choice in the ordering;
 - more actions applicable to reduce a goal.⁷²

⁷²E.g. **holding(X)** can be reached by both **unstack(X,Y)** and **pickup(X)**, I have choiche points that make branches.

- *Solution:* heuristic strategies.
 - To select the goal.
 - To select the action.
 - * *Means-Ends analysis:*⁷³
 - find the most significant difference between the state and the goal;
 - reduce that difference before.

2. *Interacting goals.*⁷⁴

- Interacting goals G1, G2:
 - plan actions for reaching G2;
 - then to solve G1 we destroy what we have done for G2;
 - at the end of the planning G2 is not true anymore.
- Complete solution:
 - try all possible orderings of goals and subgoals.⁷⁵
- Practical solution (STRIPS):⁷⁶
 - solve them independently;
 - verify afterward;
 - if the conjunction is not true, change ordering.

Sussman anomaly

Sussman anomaly represents the most largely used example for showing *interacting goals* and their effects.

Initial state:

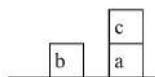


Figure 6.29

```
clear(b),
clear(c),
on(c,a),
ontable(a),
ontable(b),
handempty
```

⁷³But generally I can use any heuristic strategy that enables to reduce the search space.

⁷⁴This is the most important pitfall of STRIPS; really difficult to be managed: why? Because I plan the actions for reaching one goal (of them), and then, when I solve the other goals I could destroy what I've done for reaching the first. And at the end of this process I think I've reached 2 goals but in fact I've reached just one.

⁷⁵Number of orderings that is equal to the permutation of these goals so it goes on with the factorial. This raise a number of paths which has a factorial cardinality in comparison to the number of goals that you have to reach $n! = n \cdot n - 1 \cdot \dots \cdot 1$ which grows very very fast.

⁷⁶In general the practical solution that STRIPS proposes is the following: I solve them independently, at the end I leave on the stack the 'and' (\wedge) conjunctions there, and then at the end I check weather the conjunction is satisfied; if it is not I will re-upload the goals on the stack using a different order.

Goal: `on(a,b)`, `on(b,c)`

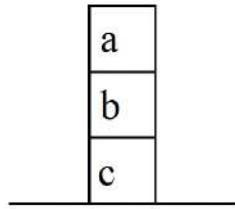


Figure 6.30

Two possible initial stacks:

```
% (1)
on(a,b)
on(b,c)
on(a,b) and on(b,c)
```

```
% (2)
on(b,c)
on(a,b)
on(a,b) and on(b,c)
```

We chose (1).

We apply the STRIPS algorithm for the first goal and obtain:

1. `unstack(c,a);`
2. `putdown(c);`
3. `pickup(a);`
4. `stack(a,b).`

Current state:

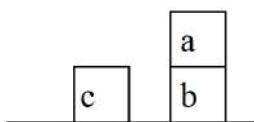


Figure 6.31

Now we plan for the second goal `on(b,c)`.

5. `unstack(a,b);`
6. `putdown(a);`
7. `pickup(b);`
8. `stack(b,c).`

Current state:

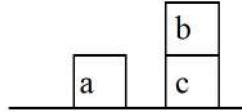


Figure 6.32

The conjunction $\text{on}(a,b) \wedge \text{on}(b,c)$ is not valid.
So we reinsert $\text{on}(a,b)$ in the goal stack and obtain:⁷⁷

9. `pickup(a);`
10. `stack(a,b).`

We have obtained what we need, *but not that efficiently*.⁷⁸

Search in the space of Plans

We change approach in the sense that we consider *non-linear* planners. Basically non-linear planners differ from *linear* planners for the fact that they do not have to provide totally ordered list of actions.

- Linear planners are search algorithms that explore the state space: the plan is a linear sequence of actions to achieve the goals.
- Non-linear planners are search algorithms that generate a plan as a search problem in the space of plans: in the search tree each node is a partial plan and operators are plan refinement operations.⁷⁹
- A non-linear generative planner assumes that the initial state is fully known *Closed World Assumption*:⁸⁰ everything that is not explicitly stated in the initial state is considered as false.
- *Least Commitment planning*: never impose more restrictions than those that are strictly necessary. Avoid making decisions when they are not required. This avoids many backtracking.⁸¹

⁷⁷We reinsert the sub-goal that is not true in the current state.

⁷⁸Not optimal, we destroy a plan that we already applied. All linear planners have the problem of *interacting goals*. In fact we solve the problem of interacting goals by changing approach completely; see ahead.

⁷⁹Up to now we've considered planners as searching in the state space where in each node we've the representation of the current state of the world ($\text{on}(a,b)$, $\text{on}(b,c)$...). When we look at *non-linear* planners we start with an *empty plan* (initial node) and the goal of course is a complete plan (that of course is correct for the scope that I want to reach and it is safe; we will see what *safe* means). At each step in this search space what should I do? I will have an operation of *plan refinement*; if I have an empty plan the most intuitive refinement is to add an *action* to the plan; there are other plan refinement operations because I have basically to consider the interacting goals: just intuitively, I will proceed to reach each goal independently with its own chain that is not ordered with the others. As soon as I find that there is an interaction of an action in one plan with another plan, then I will impose, for example, an ordering constraint (these are plan refinement operations) or other things; we will see.

⁸⁰All STRIPS assume this assumption.

⁸¹Why should I decide an order – for example – if it is not really needed that I order the

Non-linear planning

A non-linear plan is represented as:

- a set of *actions* (instances of operators);
- a (not exhaustive) set of *orderings*⁸² between actions;
- a set of ‘*causal links*’ (described later).

Initial plan: it is an empty plan with two fake actions.⁸³

- *start*: No preconditions; its effects match the initial state.
- *stop*: No effects; its pre-conditions match the goal.
- Ordering: start < stop.⁸⁴

At each step either the set of operators or the set of orderings or the set of causal links is increased until all goals are met.

- Non required orderings are not posted.

A solution is a set of partially specified and partially ordered operators.

To obtain a real plan the partial order should be linearized in one of the possible total orders (operation of linearization).⁸⁵

Example: plan to wear shoes

Let’s make a very intuitive example: put socks and wear shoes. It is clear that left and right goals don’t interact. I have two actions: `WearShoe(Foot)` and `PutSocks(Foot)`.

Goal:

- `shoeOn (dX), shoeOn (sX)`.

Actions:

- `WearShoe(Foot)`.
 - PRECOND: `socksOn (Foot)`.⁸⁶
 - EFFECT: `shoeOn (Foot)`.
- `PutSocks(Foot)`.

actions? It pays off because if you take the wrong decision then you have to backtrack and lose time and computational capacity in undoing something that was not added. *E.g.* if an action A provides *preconditions* of the action B, then of course they need to be ordered: A first, B then (because A provides something for B); they have causal relation. But if they are not related why should I order them? Any order could be fine.

⁸²Not totally ordered, but partially ordered.

⁸³With the fake actions *start* and *stop* you’re bringing to the plan the information about where/when to start and where/when to end.

⁸⁴Of course start should come before stop.

⁸⁵All these points will be clearer after an example.

⁸⁶In the initial plan which is empty (remember the *Closed World Assumption*) it is not true, so I cannot apply this action.

- PRECOND: $\neg \text{socksOn}(\text{Foot})$.⁸⁷
- EFFECT: $\text{socksOn}(\text{Foot})$.

Initial plan:⁸⁸

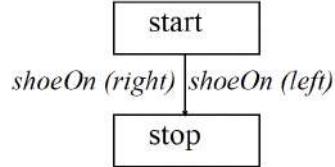


Figure 6.33

Final partial plans:⁸⁹

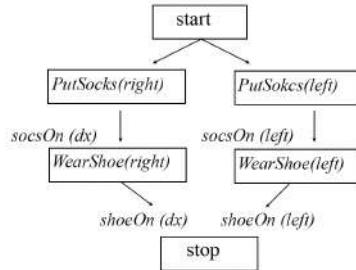


Figure 6.34

Partial Order Planning: intuitive algorithm

Let's introduce here a very important concept: '*causal link*' (before going into the algorithm); '*causal link*' is a structure that tells you why you've inserted an action in the plan; it is a triple (3 informations) (S_i, S_j, c) where c represents the condition for which this causal link holds while S_i, S_j are actions. The causal link tells you: we've inserted S_i in the plan because S_j requires c for its execution; obviously it introduces also ordering constraints. *Causal links* need to be protected: it means that if somewhere in your plan there is an action that negates c meaning that contains c in its delete list, so cancels c , such action cannot be executed between S_i and S_j , so it should go either before S_i or after S_j for example.

Let's see the algorithm.

⁸⁷Negation of something that is false is true, so I can apply this action in the initial plan.

⁸⁸See below: preconditions of `stop` match the goal.

⁸⁹See below; I want to have `shoeOn` on the right and `shoeOn` on the left. So for the left and the right we have `PutSocks` and `WearShoe`: these are 2 plans that are independent, so why should I order them? The only order that is required is that you have to wear socks before putting up shoes. So I start with the initial plan. At each step I add one action, refining the plan reaching the goal. But an *execution of real plan needs linearization*: what does it mean? We need an operative order (sequence of actions) among the possible total orders to really execute. *E.g.* before all the actions for left, then all the actions for right; or before `PutSocks` left-right, then `WearShoe` left-right.

- While (plan not complete) do:⁹⁰
 1. select an action SN that has a precondition c not satisfied;
 2. select an action S (new or already in the plan) that has c among its effects;
 3. add the order constraint $S < SN$;
 4. if S is a new action⁹¹ add the constraint $Start < S < Stop$;
 5. add the causal link (S, SN, c) ;⁹²
 6. solve any threat on causal links.
- End.

Considerations:

- in case of failure if choice points exist, the algorithm backtracks and it explores alternatives;
- a causal link is a triple that consists of two operators S_i, S_j and a subgoal c ; c should be precondition of S_j and effect of S_i ;
- a causal link stores the causal relations between actions: it traces why a given operator has been introduced in the plan;
- causal links help tackling the problem of interacting goals.

$$S_i \xrightarrow{c} S_j$$

Figure 6.35

Causal links and threats

Let's go here in a deeper detail, remember that the concept of '*causal link*' is the crucial concept of *non-linear planners*.

An action $S3$ is a threat for a causal link $(S1, S2, c)$ if it has an effect that negates c and no ordering constraint exists that prevents $S3$ to be performed between $S1$ and $S2$.

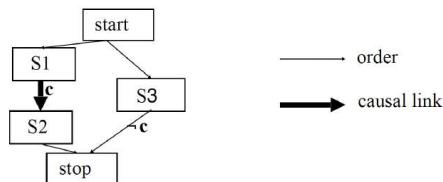


Figure 6.36

⁹⁰When we've achieved all goals – so no more open goals – and there are no threats in my plan, that's a complete plan.

⁹¹If it is already in the plan we've already the constraint.

⁹² S has been inserted in the plan because SN requires c and S has c among its effects.

Looking at the figure above note that this is another chain not ordered with the other; what we said before is that if we have no order between $(S1, S2)$ and $S3$ every *linearization* is fine so also $(S1, S3, S2)$ for example, but obviously this is not okay because $S3$ would delete c which is the precondition of $S2$. So I need to avoid $S3$ going between $S1$ and $S2$.

Possible solutions.⁹³

- *Demotion*: the constraint $S3 < S1$ is imposed;
- *Promotion*: the constraint $S2 < S3$ is imposed.⁹⁴

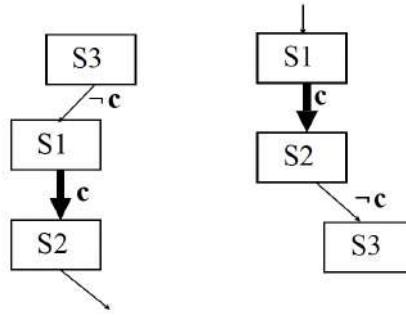


Figure 6.37

Example: purchasing schedule

Let's make an example: we want to make a banana milkshake at home.

- Initial state:⁹⁵

```
at(home), sells(HWS, drill), sells(sm, milk), sells(sm, banana)
```

- Goal:

```
at(home), have(drill), have(milk), have(banana)
```

- Actions:⁹⁶

```
Go(X, Y):
```

```
PRECOND: at(X)
```

```
EFFECT: at(Y), not at(X)
```

```
buy(S, Y):
```

```
PRECOND: at(S), sells(S, Y)
```

```
EFFECT: have(Y)
```

⁹³They are not the only way to solve a threat. We will see others but for now consider them.

⁹⁴With these kind of operation we intend as '*to make causal links safe*'. So just intuitively you have to understand that a *non linear planner* is a partially ordered plan where I've reached the all possible goals and all causal links are safe from threats.

⁹⁵Below as method arguments we've places selling things; in particular sm stands for *supermarket* and HWS stands for *hardware shop*.

⁹⁶Below we can read the variable S like *shop that sells Y*.

- Initial plan (known null):⁹⁷



Figure 6.38

Let's start with the procedure.

- First step:

- select a precondition (goal) to be fulfilled: `have(drill)`,⁹⁸
- select an action that has `have(drill)` as an effect: `buy(X, Y)`,⁹⁹
- plan refinement:
 - * link variable Y with the term `drill`;¹⁰⁰
 - * impose ordering constraints `Start < buy < Stop`;
 - * insert the causal link `<buy(X, drill), stop, have(drill)>`.¹⁰¹

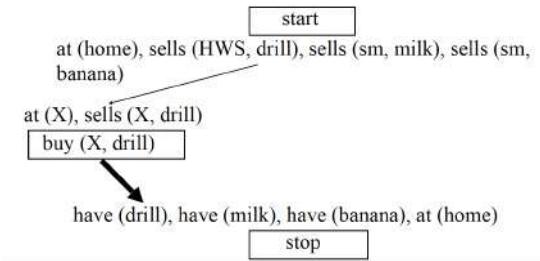


Figure 6.39

- Same procedure for:

- `have(milk);`
- `have(banana).`

⁹⁷Remember that `start` and `stop` are two fake actions; `start` has as its effect the initial state while `stop` has as precondition the goal.

⁹⁸We select one of the four goals we want to achieve.

⁹⁹Is the only action that has `have(drill)` as an effect.

¹⁰⁰We leave X as variable, we don't know yet where to buy `drill`.

¹⁰¹I've inserted the `buy` because the `stop` action requires `have(drill)` as precondition. Below notice that we still need to solve the open goals left (*i.e.* `have(milk)`, `have(banana)`, `at(home)`), plus the precondition of the selected action (*i.e.* `at(X), sells(X, drill)`).

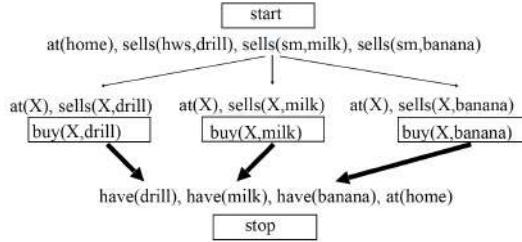


Figure 6.40

- Select `sells(X, drill)`, true in the initial state imposing $X = \text{hws}$. The same happens for `sells(X, milk)` and `sells(X, banana)` with $X = \text{sm}$. Add causal links.¹⁰²

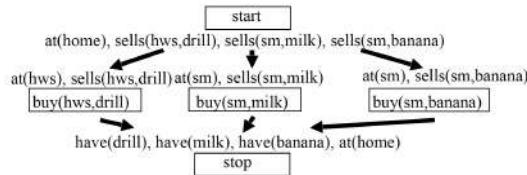


Figure 6.41

- Select:

- `at(hws)` precondition of `buy(hws, drill)`;¹⁰³
- Add `go(X, hws)` in the plan along with orderings and causal links.¹⁰⁴

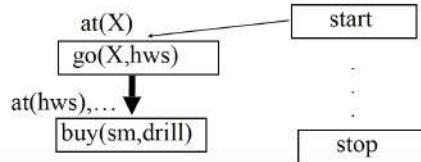


Figure 6.42

¹⁰²We have to satisfy the preconditions of the `buy`: remember that the preconditions of the `buy` have variables inside (when I want to buy the drill I don't know exactly where it is); how can I know where it is? When I select the `sells(X, drill)` then what can I use? Is there an action already in the plan (or a new one) that satisfies `sells(X, drill)`? Is there an action that has already this as an effect? Yes, the `start` action. At this point and only at this point, we unify X with hws and this X is contained in the `buy` and also in the `at`, is the same X (clearly for the other twos, being 2 other instantiations of an action the X is different). So we create 3 causal links between the `start`, the 3 `buy` (one for `sells(hws, drill)`, the second for `sells(sm, milk)` and the third one for `sells(sm, banana)`).

¹⁰³Now we want to satisfy the precondition `at`: `have(drill), have(milk), have(banana)` have been reached (not yet `at(home)`) plus `sells(hws, drill), sells(sm, milk), sells(sm, banana)`. We have this `at` which is now open.

¹⁰⁴I have only one action that has `at` as its effect i.e. `go`. Again here try to consider the variables: I want to be at the `hws`, `hws` is the destination of the `go` action. But I still don't know where I start from. So, since in the initial state I've `at(home)` I can use it.

- Select $\text{at}(X)$ as precondition of $\text{go}(X, \text{hws})$, true in start with $X = \text{home}$ and protect the causal link.

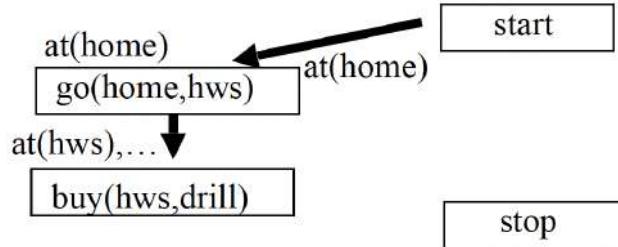


Figure 6.43

- Same procedure for $\text{at}(\text{sm})$.¹⁰⁵

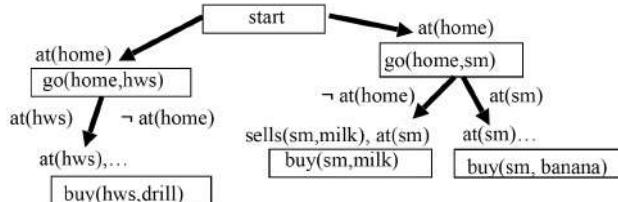


Figure 6.44

- Solve the conflict between actions $\text{go}(\text{home}, \text{hws})$ and $\text{go}(\text{home}, \text{sm})$:

- if the agent performs $\text{go}(\text{home}, \text{hws})$ it cannot be $\text{at}(\text{home})$ to perform $\text{go}(\text{home}, \text{sm})$ and viceversa;
- *imposing ordering constraints does not work*;¹⁰⁶
- backtrack on the solution step of $\text{at}(X)$ (precondition of $\text{go}(X, \text{sm})$);
- use¹⁰⁷ $\text{go}(\text{home}, \text{hws})$ instead of start with $X = \text{home}$ to satify $\text{at}(X)$ with $X = \text{hws}$;

¹⁰⁵Look at the figure below in the middle. We have here a syntactic problem: we have that $\text{go}(\text{home}, \text{hws})$ negates $\text{at}(\text{home})$; the other causal link $\langle \text{start}, \text{go}(\text{home}, \text{sm}), \text{at}(\text{home}) \rangle$ (where we can see start as S, $\text{go}(\text{home}, \text{sm})$ as SN and $\text{at}(\text{home})$ as C) has $\text{at}(\text{home})$ which is negated on the other side and the 2 actions $\text{go}(\text{home}, \text{sm})$, $\text{go}(\text{home}, \text{hws})$ are not ordered but also if ordered we would have no benefit; so one action would threat the other causal link and viceversa: as the algorithm suggests we need to *backtrack* in order to solve the conflicts. Remember that these planners do not have any idea of what these actions are, what is their meaning; they use *syntactic* rules and through syntactic rules you've achieved something that is *semantically* consistent. This is crucial for AI. This is true for all the *symbolic* part of AI.

¹⁰⁶Also because if we decide to perform *demotion* on $\text{go}(\text{home}, \dots)$ then go would be before start and nothing goes before start or if we decide to perform *promotion* we should put one go after the other go that has as precondition $\text{at}(\text{home})$ which is negated by the previous go .

¹⁰⁷There is another action able to provide at somewhere ($\text{at}(X)$) and it is go .

- so we have $\text{buy}(\text{hws}, \text{drill}) < \text{go}(\text{hws}, \text{sm})$.¹⁰⁸ This way $\text{at}(\text{hws})$ is protected by the causal link between $\text{go}(\text{home}, \text{hws})$ and $\text{buy}(\text{hws}, \text{drill})$ (*promotion*).

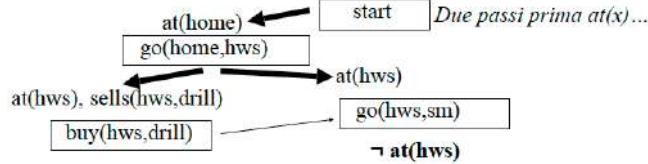


Figure 6.45

- Last step:

- solve $\text{at}(\text{home})$ of stop : the only way is to put the action $\text{go}(\text{home})$ before stop .



Figure 6.46

- Complete and safe plan:¹⁰⁹

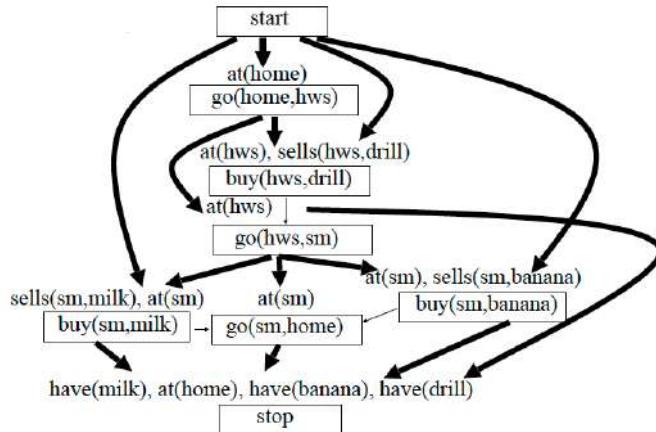


Figure 6.47

¹⁰⁸Because this $\text{go}(\text{hws}, \text{sm})$ has **not** $\text{at}(\text{hws})$ and this negation threatens the causal link between $\text{go}(\text{home}, \text{hws})$ and $\text{buy}(\text{hws}, \text{drill})$ because I have to be at the hardware shop for buying the drill, but with *promotion* I can solve the problem.

¹⁰⁹So as you can see in the figure below we've only 3 ordering constraints (the normal arrows): the first ordering constraint is because there is the *promotion* $\text{buy}(\text{hws}, \text{drill}) < \text{go}(\text{hws}, \text{sm})$ discussed before and then because we need to be at the sm to buy milk and banana. The rest are causal links (bold arrows); then any linearization is a plan. If we focus on the level where we have basically normal arrows the level means: in any order buy milk and banana.

- A final plan is obtained by ordering all actions:

```

1. go(home, hws);
2. buy(hws, drill);
3. go(hws, sm);
4. buy(sm, milk);
5. buy(sm, banana);
6. go(sm, home).

```

Partial Order Planning Algorithm (POP)

Now we go with more detail into seeing the algorithm:

```

function POP (initialGoal Operators) returns plan
plan := INITIAL_PLAN(start, stop, initialGoal)
loop
  if SOLUTION(plan) then return plan;
  SN, C := SELECT_SUBGOAL(plan);
  CHOOSE_OPERATOR(plan, operators, SN, C);
  RESOLVE_THREATS(plan)
end

function SELECT_SUBGOAL(plan)
  select SN from STEPS(plan) with unsolved precondition C;
  return SN, C

procedure CHOOSE_OPERATOR(plan, ops, SN, C)
  pick an S with effect C from ops or from STEPS(plan);
  if S does not exist then fail;
  add the causal link <S, SN, C>
  add the ordering constraint S < SN
  if S is a new action added to the plan
  then add S to STEPS(plan)
  add the constraint Start < S < Stop

procedure SOLVE_THREAT(plan)
  for each action S that threatens a causal link between Si and Sj,
  choose either
    demotion: add the constraint S < Si
    promotion: add the constraint Sj < S
    if NOT_CONSISTENT(plan) then fail

```

Let's analyze from top to bottom the algorithm.

- Regarding the first block of code.

- function POP (initialGoal Operators) returns plan
 - * We have a function POP (Partial Order Planner) that takes the InitialGoal and Operators and will return the plan.
- if SOLUTION(plan) then return plan

- * If `plan` is a `SOLUTION` then I return the `plan`; a `SOLUTION` means what? When do I have to stop? When the `plan` has reached all the goals without threats? So this `SOLUTION(plan)` checks if we are in that condition.
- `SN, C := SELECT_SUBGOAL(plan);`
- * Otherwise then I will select the `SUBGOAL C` from the `plan`, coming from the action `SN`.
- `CHOOSE_OPERATOR(plan, operators, SN, C);`
- * Then we can chose an `OPERATOR` for reaching `C` of `SN`. The `CHOOSE_OPERATOR` of course needs the `plan` that has been built up to now, the operators that are new possible actions not yet inserted into the `plan` and of course the action `SN` and the condition `C`.
- Regarding the second block of code.
 - `select SN from STEPS(plan) with unsolved precondition C;`
 - * What is `STEPS(plan)`? Are those actions that are already been inserted into the `plan` (initially we've just the `Start` and the `Stop`) with an unsolved precondition `C`.
- Regarding the third block of code.
 - `procedure CHOOSE_OPERATOR(plan, ops, SN, C)`
 - `pick an S with effect C from ops or from STEPS(plan);`
 - * The procedure `CHOOSE_OPERATOR` is very important because basically for choosing an operator you need to pick up `S` that has `C` as an effect either from `ops` (meaning *new actions*) or from `STEPS(plan)` (meaning actions that are already there into the `plan`).
 - `add the ordering constraint S < SN`
 - * Every time we've a causal link we've also an ordering constraint (not viceversa) because `C` is a precondition of `SN`, so `S < SN`.
- Regarding the fourth block of code:
 - `demotion: add the constraint S < Si`
 - `promotion: add the constraint Sj < S`
 - * *demotion* and *promotion* are not the only way to solve threats; let's see ahead.

Modal Truth Criterion (MTC)

Promotion and *demotion* alone are not enough to ensure the completeness of the planner. A planner is complete if it always finds a solution if a solution exists. *The Modal Truth Criterion* is a construction process that guarantees planner completeness.

A *Partial Order Planning algorithm* interleaves goal achievement steps with threat protection steps.

The *MTC* provides five plan refinement methods (one for the open goal achievement and 4 for threat protection) that ensure the completeness of the planner.

PARTIAL PLANS (We're exploring a Search Space that is the Space of Partial Plans)

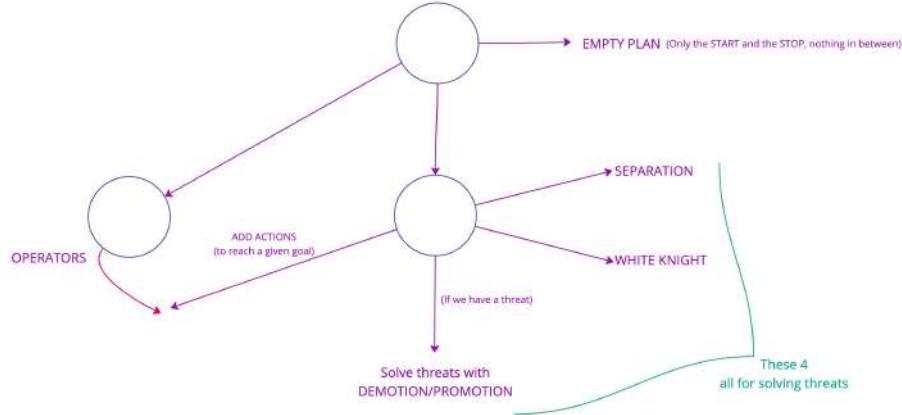


Figure 6.48

Let's enumerate the methods.

1. *Establishment*: open goal achievement by means of: (1) a new action to be inserted in the plan, (2) an ordering constraint with an action already in the plan or simply (3) of a variable assignment.
2. *Promotion*: ordering constraint that imposes the threatening action before the first of the causal link.
3. *Demotion*: ordering constraint that imposes the threatening action after the second of the causal link.
4. *White knight*: insert a new operator or use one already in the plan between S3 and S2 such that it establishes the precondition of S2 threatened by S3.

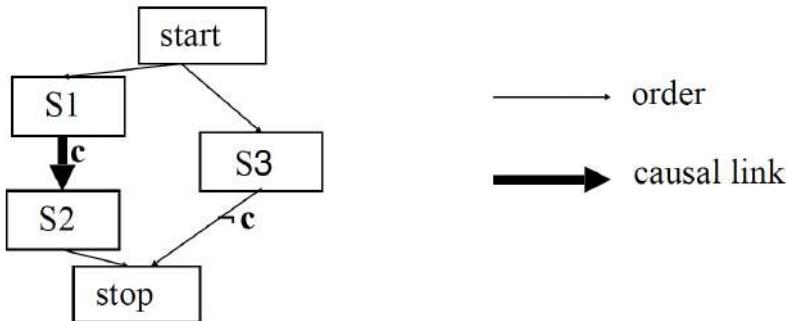


Figure 6.49

Suppose I don't find a plan using *demotion/promotion*; this means that S3 cannot go before S1 and cannot go after S2; so S3 goes in between S1 and S2. What happens now is that S1 is imposing c, S3 is destroying c so I need to add a new action that establish c again and this action is called a *White knight*.

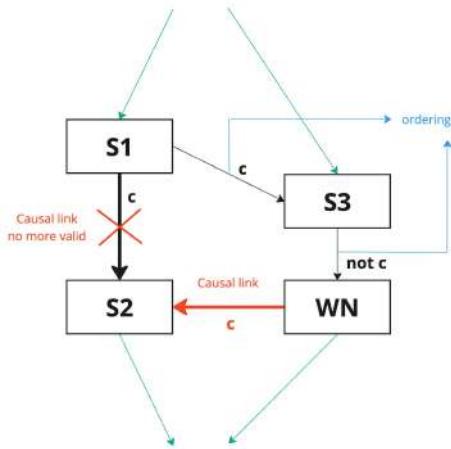


Figure 6.50

Of course this is not efficient, the plan generated by POP are not optimal, they can be very long and inefficient but they are *complete*.

5. *Separation*: insert non codesignation constraints between the variables of the negative effect and the threatened precondition so to avoid unification. This is useful when variables have not yet been instantiated. *E.g.* given the causal link

$$\text{pickups}(X) \xrightarrow{\text{holding}(X)} \text{stack}(X, b)$$

Figure 6.51

any threat imposed by `stack(Y, c)` can be solved by imposing `X != Y`.

Separation is very rarely used. Let's be a little bit more explicit: let's consider the BLOCK WORLD.

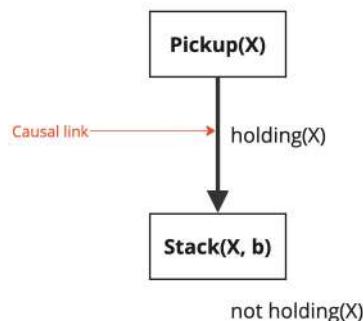


Figure 6.52

Now suppose that in another part of the plan we have another link in the same situation.

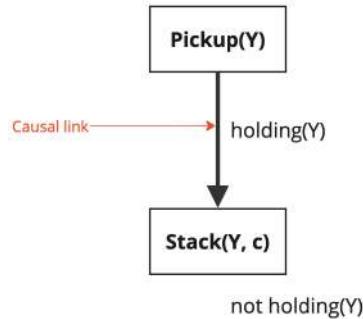


Figure 6.53

The 2 situations are not ordered (we've only causal links), so in principle the $\text{Stack}(Y, c)$ would threat the first causal link because if X and Y would be the same variable we would have a threat; and viceversa with $\text{Stack}(X, c)$.

Generally they are used in this order by the algorithm. Of course there are cases in which we are not able to solve threats with these methods. In these cases we can *backtrack*!

Example: Sussmann Anomaly

Initial state ([IS]):

```
%[IS]
clear(b), clear(c), on(c, a), ontable(a),
ontable(b), handempty.
```

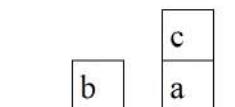


Figure 6.54

Goal ([G]):

```
%[G]
on (a,b), on (b, c).
```

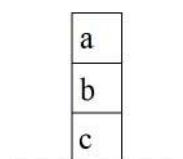


Figure 6.55

Actions:

```
%actions
pickup(X)
PRECOND: ontable(X), clear(X), handempty
POSTCOND: holding(X) not ontable(X), not clear(X),
           not handempty

putdown(X)
PRECOND: holding(X)
POSTCOND: ontable(X), clear(X), handempty

stack(X, Y)
PRECOND: holding(X), clear(Y)
POSTCOND: not holding(X), not clear(Y), handempty,
           on(X, Y), clear(X)

unstack(X, Y)
PRECOND: handempty, on(X, Y), clear(X)
POSTCOND: holding(X), clear(Y), not handempty,
           not on(X, Y), not clear(X)
```

Initial plan:¹¹⁰

```
Orderings: [start < stop]
List of Causal links: [].
Goal agenda: [on(a, b), on(b, c)]
```



Figure 6.56

Establishment: chose two actions (without orders) that meet the goals:

```
stack(a, b)
PRECOND: holding(a), clear(b)
POSTCOND: handempty, on(a, b), not clear(b), not holding(a)

stack(b, c)
PRECOND: holding(b), clear(c)
POSTCOND: handempty, on(b, c), not clear(c), not holding(b)

Orderings:
[start < stop, start < stack(a, b), start < stack(b, c),
 stack(a, b) < stop, stack(b, c) < stop].

List of Causal links:
[<stack(a, b), stop, on(a, b)>, <stack(b, c), stop, on(b, c)>].
```

¹¹⁰Below Goal agenda refers to the set of goals that are still open.

Goal agenda:

[holding(a), clear(b), holding(b), clear(c)]

The actions are not ordered. At the moment we only know that both of them will occur.

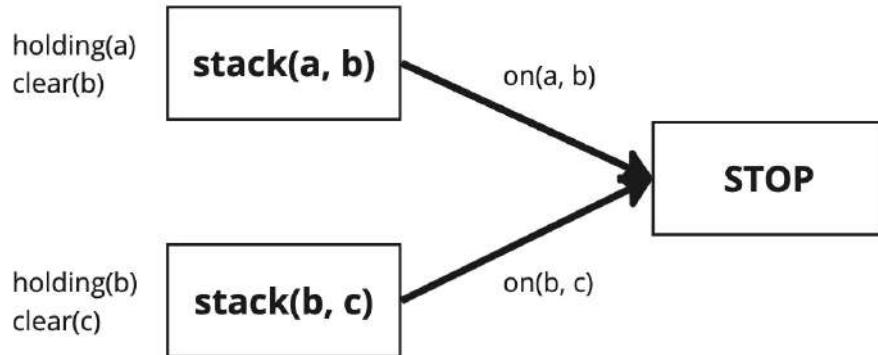


Figure 6.57

Some of the preconditions on the agenda (`clear(b)` and `clear(c)`) are already met in the initial state: just add the causal link.

Orderings:

[see above]

List of Causal links:

[..., <start, stack(a, b), clear(b)>, <start, stack(b, c), clear(c)>].

Goal agenda:

[holding(a), holding(b)]

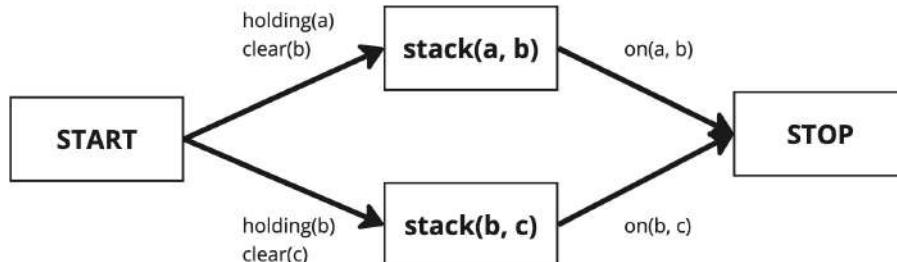


Figure 6.58

Now we proceed with the establishment of `holding(a)`, `holding(b)`.

`pickup(a)`

PRECOND: `ontable(a)`, `clear(a)`, `handempty`

```

POSTCOND: not ontable(a), not clear(a), holding(a),
not handempty.

pickup(b)
PRECOND: ontable(b), clear(b), handempty
POSTCOND: not ontable(b), not clear(b), holding(b),
not handempty.

Orderings:
[..., pickup(a) < stack(a, b), pickup(b) < stack(b, c)]

Causal links:
[..., <pickup(a), stack(a, b), holding(a)>, <pickup(b),
stack(b, c), holding(b)>]

Goal Agenda:
[ontable(a), clear(a), ontable(b), clear(b), handempty]

Partial current plan:

```

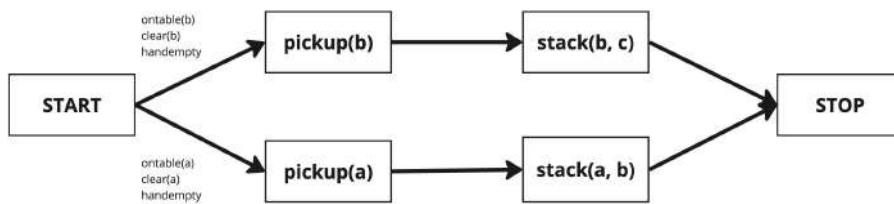


Figure 6.59

`ontable(b), ontable(a), clear(b)` are met in the initial state.

Orderings:
[see above]

Causal links:
[..., <start, pickup(a), ontable(a)>, <start, pickup(b),
ontable(b)>, <start, pickup(b), clear(b)>]

Goal Agenda:
[handempty, clear(a)]

`stack(a, b)` threatens `<start, pickup(b), clear(b)>` as `not clear(b)` is one of its effects and nothing prevents it to precede `pickup(b)` and invalidate its precondition `clear(b)` \implies impose (*promotion*) `pickup(b) < stack(a, b)`. `handempty` is true in the initial state.

The new causal link `<start, pickup(b), handempty>` is threatened by `pickup(a)` \implies impose (*promotion*) `pickup(b) < pickup(a)`.

The precondition `handempty` of `pickup(a)` cannot be satisfied from the `start` because `pickup(b)` preceding `pickup(a)` would negate it¹¹¹ \implies we apply the

¹¹¹Remember that it's quite always the case that if you have 2 threats that are 'crossed' –

white knight using the plan action `stack(b, c)`¹¹² that reinforces `handempty`:
`pickup(b) < stack(b, c) < pickup(a)`.¹¹³

Orderings:

```
[..., pickup(b) < stack(a, b), pickup(b) < pickup(a),
  stack(b, c) < pickup(a)]
```

Causal links:

```
[..., <start, pickup(b), handempty>,
  <stack(b, c), pickup(a), handempty>]
```

Goal Agenda:

```
[clear(a)]
```

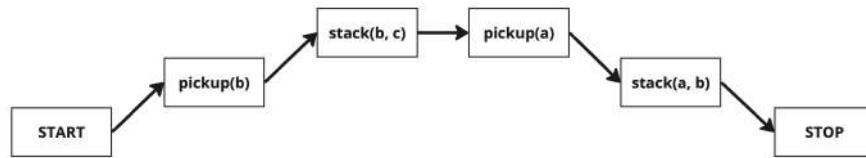


Figure 6.60

To satisfy `clear(a)` we can add the action:¹¹⁴

```
unstack(X, a)
PRECOND: handempty, on(X, a), clear(X)
POSTCOND: handempty not, clear(a), holding(X) not on(X, a)
```

Current partial plan:

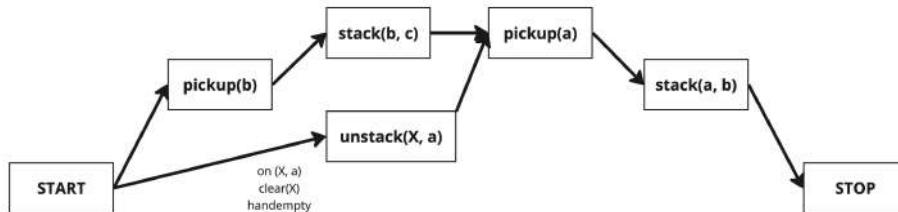


Figure 6.61

as in the case of `pickup(b)` and `pickup(a)` – you need to use a *white knight*. What does it mean *crossed*? `pickup(a)` has **not** `handempty` in its effects and threatens the causal link between `start` and `pickup(b)` and viceversa for `pickup(b)`. So with ‘*crossed*’ we mean 2 actions that are threatening each other causal links; 90% of the cases they are solved with *white knight*. Also if in our example we are working with `start` action and we cannot put nothing before `start`, what we’ve stated is valid in general.

¹¹²An action already in the plan; we use it because makes true `handempty`.

¹¹³Remember that at this level in which I apply the *white knight* I’ve removed the causal link between the `start` and the `pickup(a)` because of `handempty` and I’ve added the new causal link between the `stack(b, c)` and `pickup(a)` because of `handempty`. See the figure below.

¹¹⁴`clear(a)` was not true in the initial state and therefore I’ve to remove whatever is on top of `a` from `a` with the `unstack`. How can we choose algorithmically an action to reach `clear(a)`? For example one reason is that such action needs to have `clear(a)` among its effect.

In the lists below consider that `pickup(a)` has `clear(a)` as its precondition.

Orderings:

```
[..., start < unstack(X, a), unstack(X, a) < stop,
unstack(X, a) < pickup(a)]
```

Causal links:

```
[..., <unstack(X, a), pickup(a), clear(a)>
```

Goal Agenda:

```
[on(X, a), clear(X), handempty]
```

The three preconditions are met on the agenda in the initial state by imposing $X = c$ in the move `unstack(X, a)`.

`unstack(c, a)` is a threat to the causal link `<start, pickup(b), handempty>`¹¹⁵
 \implies add a new move *white knight*.¹¹⁶

```
putdown(c)
PRECOND: holding(c)
POSTCOND: ontable(c), clear(c), handempty, not holding(c)
```

Ordering as well:

```
unstack(c, a) < putdown(c) < pickup(b)
```

We then used two types of *white knights*, one using an action which is already in the plan, and one introducing a new action.

Now all the preconditions are met so we can linearize the plan (add the ordering constraints) and possibly instantiating not yet instantiated variables, to get a concrete plan:

1. `unstack(c, a);`
2. `putdown(c);`
3. `pickup(b);`
4. `stack(b, c);`
5. `pickup(a);`
6. `stack(a, b).`

Closing remarks

Closing remarks about *non linear planners*.

- It is always preferable to apply *promotion* and *demotion* before *white knight* (in particular when the action inserted is a new action not belonging to the plan).¹¹⁷

¹¹⁵Since `unstack(c, a)` has `not handempty` among its effects.

¹¹⁶We need to re-establish the `handempty`.

¹¹⁷Because of course inserting a *white knight* makes the planner longer. Remember that every time you use a *white knight* you need to cancel a causal link and insert a new one!

- Unfortunately non-linear planners can generate very inefficient plans, even if correct.
- Planning is semi-decidable: if there is a plan that solves a problem the planner finds it,¹¹⁸ but if there is not, the planner can work indefinitely.
- In domains of increasing complexity it is hard to use a correct and complete planner, which is efficient and domain independent. Often we use ad-hoc methods and heuristics.

Planning in practice

Many applications in complex domains.

- Planners for the Internet (search for information).
- Softbots:
<https://www.washington.edu/research/pathbreakers/1991a.html>
- Management of space missions:
<http://ti.arc.nasa.gov/tech/asr/planning-and-scheduling/>
- Robotics:
<http://www.robocup.org/>
- Graphplan developed by Carnegie Mellon University:
<http://www.cs.cmu.edu/~avrim/graphplan.html>
- Industrial production plans.
- Logistics.

Classical algorithms have efficiency problems in case of complex domains.
 There are techniques that make the algorithms more efficient \Rightarrow *Hierarchical planning*.

6.4 Planning based on Graph

We are going to see one of the most efficient planner that has ever built: *Graph-plan*.

Planning based on Graph

In 1995 a new concept of planner based on graph has been proposed by Blum and Furst, *CMU – GRAPHPLAN*.¹¹⁹

While planning it creates a graph called *Planning Graph*: at each step of the search this data structure is extended.¹²⁰

¹¹⁸Completeness.

¹¹⁹Even if it is quite old, the concepts behind this are used nowadays in the most efficient planners.

¹²⁰The data structure contains in principle all possible plans that one can build starting from the initial state adopting a *compact* representation.

Graphplan inserts the *TIME* dimension in the plan construction process.¹²¹ It is a correct and complete planner among the most efficient that have been built.

Graphplan: features

We're still in the world of generative planners so they take a snapshot of the initial state, they plan offline and when they've done, the plan can be executed. So are still valid all the hypotheses that we've done, so basically the planner is the only agent that change the world, *Closed World Assumption*, meaning that everything that it is stated in the initial state is true, what it is not stated is supposed to be false which means that the planner has *complete knowledge* on the world that is describing.

Let's list the main features.

- Graphplan uses the *Closed World Assumption* falling into the category of off-line planners.
- Graphplan returns either the shortest possible plan or returns an inconsistency.¹²²
- Graphplan has another very nice feature: it puts together two ways of performing search, *backward search* and *forward search*: *forward* is from the initial state and proceed forward while the *backward* starts from the goal and proceed backward.
- Graphplan inherits from linear planners the *early commitment* feature: *e.g.* the action A is executed at time step 2.¹²³

¹²¹Time is not really related to the duration of activity like in *scheduling* – in scheduling for example you want to schedule activities that have a given duration in time and that you want to place on the timeline – but is a kind of *TIME-STEP* which means basically the following: if you put 2 actions at the same *TIME-STEP* it means that either you can execute them in parallel or can be executed in any order, so they are not ordered; this is the concept of *TIME* which is not related to *duration*; *TIME* is a kind of *clock* that you've in your system and basically tells you if 2 actions are ordered meaning that one is executed at *TIME-STEP 1*, and the other at *TIME-STEP 2*, then of course they have an ordering constraint – one should come before the other – but if 2 of them at the end of the process stay in the same *TIME-STEP*, they are not ordered and therefore they can be executed in any order (*linearization*) or in parallel (for example if you have 2 agents).

¹²²This is a concept not seen so far: Graphplan is *optimal* in the sense that either you have a failure (and at some point you stop and you say I don't find any plan) otherwise the shortest possible plan is produced (a feature not present in STRIPS, POPS *e.g.* the possibility of using a *White Knight* excludes the possibility of having the shortest; if you basically choose the *Modal Truth Criterion* meaning that you use 4 ways for solving threats and 1 way for achieving goals then you're *complete* but this does not mean that you're *optimal*, indeed in general partial order planners produce quite long plans as in the case of the purchasing example that we've seen [go home, back to the hardware shop, go to the supermarket, ...]).

¹²³Apparently this *early commitment* seems to be not a good feature in comparison to *least commitment* so why use it here? Remember that *least commitment* in *partial order planning* asserts: why should I take a decision that is not needed? Because if take a wrong decision I need to backtrack. But here we use *early commitment* because in principle we put together all possible plans, so among all possible plans of course you have both good/bad decisions but you have all there in the planning graph, and therefore the *early commitment* means that if you have all the possible paths from the initial state to the goal compressed in this planning graph then of course you have to perform *early commitment* because you've all possible choices.

- Graphplan inherits from non-linear partial order planners the ability to create partially ordered sets of actions.¹²⁴
- It generates parallel plans.

Graphplan

Actions are represented as the ones in STRIPS:

- PRECONDITIONS;
- ADD LIST;
- DELETE LIST.

Objects have a type.¹²⁵

There is an action `no-op` that does not change the state (*frame problem*).¹²⁶
States are represented as sets of predicates that are true in a given state.¹²⁷

Planning graph

The planning graph is a *directed leveled graph*:¹²⁸

- nodes belong to different levels;
- arcs connect nodes¹²⁹ in adjacent levels.¹³⁰

Level 0 corresponds to the initial state.¹³¹

In the planning graph proposition levels¹³² and action levels are interleaved and correspond to increasing time steps.¹³³

In the planning graph interfering actions and propositions in a time step t can appear.

In a *planning-graph* there are different levels.

- Proposition level: nodes represent propositions.¹³⁴

¹²⁴Meaning that can be actions that can be executed in any order.

¹²⁵If you have an action `move X from A to B` and now suppose that in your semantic world you have different objects like cars, trees, buildings... so the action makes sense if X is a car because for example trees and buildings cannot be moved; from that the need to introduce the *type*. So the actions would be `move X[car] from A to B`.

¹²⁶This `no-op` action is very important because can solve the *frame problem*; what is not changed by an action should be brought forward to the next state with no changes. So this `no-op` operation has generally the task of moving forward those things that are not changed.

¹²⁷Traditional description (think to the initial state).

¹²⁸*Directed* in the sense that arcs are arrows with a direction and a verse. *Leveled* in the sense that nodes stay in layers like in bipartite graph: a bipartite graph is a graph whose vertices (nodes) can be divided in two disjoint sets (layers) such that every edge (arc) connects a vertex in one set to a vertex in the other set; two sets: the vertices of the graph are split into two distinct groups; edge connections: edges only exist between vertices in different sets, never within the same set.

¹²⁹No arcs between nodes of the same level, no arcs that connect the first level to the fourth one: only first to second... adjacent.

¹³⁰These levels are the *TIME-STEPS*.

¹³¹You start by putting in the level 0 all those that are true in the initial state.

¹³²Level in which you can find the description of a state.

¹³³PROPOSITION/ACTION/PROPOSITION/ACTION...

¹³⁴Properties that are true in a given state.

- Action level: nodes represent actions.

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

Arcs are divided into:

- Precondition arcs (proposition \rightarrow action);
- Add arcs (action \rightarrow proposition);
- Delete arcs (action \rightarrow proposition).

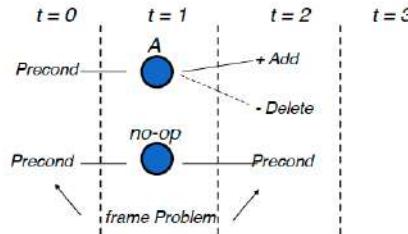


Figure 6.62

Looking at the image above let's do some considerations.

- At the $t = 0$ I have everything which is true in the initial state; these fluents represent preconditions of the actions that I put in the next stage.
- At the $t = 1$ we have the **no-op** operation: as already said this **no-op** takes any not changed fluent of the previous level and bring it forward to the next level.
- In each time step an action **A** can be inserted if in the previous time step all preconditions of **A** exist.¹³⁵
- At the $t = 2$ again we have a proposition layer in which you have everything that is added from the action linked with continuous arcs, while we have a dotted arcs that are connected with those fluents that are deleted by the action.
- Remember always that all these are syntactic rules, not semantic.
- There are special actions representing the ‘doing nothing’ activity. These actions are called **no-op** or **frame actions**.

Each action level contains:

- all actions that are applicable at that time step;
- constraints connecting pairs of actions that cannot be performed simultaneously.¹³⁶

¹³⁵This is related to the so called *frame problem* and for which the **no-op** operations are fundamental.

¹³⁶E.g. at the same level I can have both **move K from Bologna to Milano** and **move K from Bologna to Torino** but in some syntactic way (not semantic); these two actions are mutually exclusive, you either take one or the other. So in the same level of a planning graph you might have inconsistencies, so the need of constraints.

Each proposition level contains all literal that might result from any choice of actions in the previous time step including no-op.¹³⁷

Note: the construction process of the planning graph does not imply *any choice* on the selection of the action that will be inserted in the plan.¹³⁸

Inconsistencies

Now, let's try to understand how can I have a syntactic way for understanding if 2 actions are mutually exclusive or not and if 2 propositions are mutually exclusive or not. It's very important that the way we use should be syntactic because for example the planner doesn't know anything about what '*move*' means other than letters or symbols. No meaning at all while of course we have, being humans.

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

- two *actions* can be inconsistent in the same time step;
- two *propositions* can be inconsistent in the same time step.

In this case the actions/propositions are *mutually exclusive*:

- they can not appear together in a plan;¹³⁹
- *but they may appear* in the same level of the planning graph.¹⁴⁰

Inconsistent actions

Now let's see these syntactic ways to understand these inconsistencies. Remember that the fact that 2 actions are on the same level means that they can be executed in any order; from that various inconsistencies emerge.

- *Inconsistent effects*: one action negates the effects of another.
 - The move action (part, dest) has the effect not at (part) while the no-op action on at (part) has this effect.
- *Interference*: an action deletes a precondition of the other.
 - The move action (part, dest) has the effect not at (part) while the no-op action on at (part) has this as a precondition.¹⁴¹
- *Competing needs*: two actions that have mutually exclusive preconditions.
 - The load share (load means) has as a precondition not in (load means) while the action unload (load means) has as a precondition for (load means).

¹³⁷What it means is that in the action level I have `move K from Bologna to Milano` and `move K from Bologna to Torino`, in the next step you will have 2 propositions: one that `K is in Milano`, one that `K is in Torino`; but also you will have the constraints that they are mutually exclusive since also the previous related actions were mutually exclusive.

¹³⁸We are not doing any choice, we are just squeezing in this planning graph all possible plans that I can build.

¹³⁹The final plan that I extract from the planning graph.

¹⁴⁰It's fine to have inconsistencies in the planning graph, but not in the final plan.

¹⁴¹If we find for example an interference we need to link the interfering actions because if in the final plan we choose one we cannot choose also the other.

Inconsistent propositions

Two propositions are inconsistent if:

- one is the negation of the other,¹⁴²
- if all the ways to reach them are mutually exclusive.¹⁴³

Furthermore, there might be domain dependent inconsistencies:¹⁴⁴

- *E.g.* an object cannot be in two places at the same time in the same time step.

Example

We have a cart R and two loads A and B that are in the starting position L and must be moved to the target position P.

Three actions (we see the syntax later):

- MOVE(R, PosA, PosB);¹⁴⁵
- LOAD(Pos, Object);¹⁴⁶
- UNLOAD(Pos, Object).

Let's see *preconditions*, *add list* and *delete list* for the actions.

- MOVE(R, PosA, PosB):
 - PRECONDITIONS: at(R,PosA), div(PosA, PosB), hasFuel(R);¹⁴⁷
 - ADD LIST: at(R,PosB);
 - DELETE LIST: at(R,PosA), hasFuel(R).
- LOAD(Pos, Object):
 - PRECONDITIONS: at(R,Pos), at(Object,Pos);
 - ADD LIST: in(R, Object);
 - DELETE LIST: at(Object,Pos).
- UNLOAD(Pos, Object):
 - PRECONDITIONS: in(R, Object), at(R,Pos);
 - ADD LIST: at(Object,Pos);
 - DELETE LIST: in(R, Object).

¹⁴²In Bologna, not in Bologna.

¹⁴³The consequence of moving from Bologna to Milano is being at Milano. The consequence of moving from Bologna to Firenze is being at Firenze. So at Milano, at Firenze are mutually exclusive because the ways to reach them are mutually exclusive. So the mutual exclusion is also a consequence of starting from mutually exclusive actions.

¹⁴⁴Considering also these inconsistencies can speed up the process but we do not use them for our scopes. We only use syntactic ways of understanding these inconsistencies.

¹⁴⁵R represent a cart.

¹⁴⁶Pos represent a location.

¹⁴⁷Here div stands for *diverse*.

Objects:

- Cart r;
- Objects a, b;
- Locations l, p.¹⁴⁸

Initial State:

- at(a,l);
- at(b,l);
- at(r,l);
- hasFuel(r).

Goal:

- at(a,p);
- at(b,p).

Planning Graph example

After having described the problem above let's start with the operative part. This is something quite often asked in the exam as theoretical question: write 2 levels¹⁴⁹ of graph plan using a description given of a certain planning problem. So starting from the initial state, we create an *action level* and a *proposition level*.

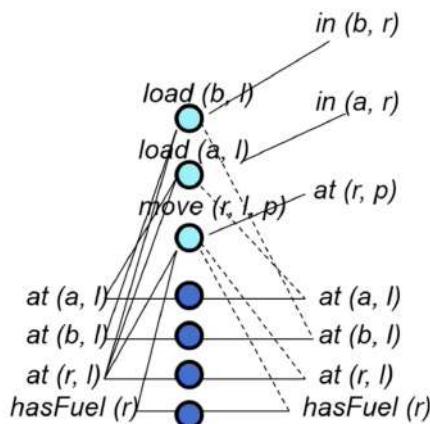


Figure 6.63

In the figure above we can distinguish *three levels* (individuated by the three columns from the left to the right).

¹⁴⁸Here **Cart**, **Objects**, **Locations** are *types*.

¹⁴⁹So 2 levels starting from the initial state (excluded); it's not 2 levels meaning: Actions-Propositions (first), Actions-Propositions (second).

- Initial State (proposition level).
- First level (action state).
- Second level (proposition level).

Now let's focus on what happens between these levels/states starting from the Initial State to First level and so on.

- For creating the action level: ask yourself, starting from the problem available actions listed before, which actions are applicable in TIME STEP 1, meaning which actions have their preconditions that are true in the initial state. We are going forward; but note that we're not branching like in tree search (I can go either here or there...) but we're keeping everything there. Note that the blue circles represent the `no-op` actions. We have to use these `no-op` one per proposition; so I will bring forward all the 4 initial propositions. Then its obvious that the actions at the first level can be incompatible, but before understanding which are incompatible let's write also the effect list (Add list and Delete list of the actions, note the continuous and dotted arrows). The `no-op` actions simply bring forward everything. Now we can think about inconsistencies.

Inconsistencies between actions.

- For interference:
 - `load(a,1)` and `move(r,1,p)`;
 - `load(b,1)` and `move(r,1,p)`;
 - `load(a1)` with `no-op` on `at(a,1)`;
 - `load(b,1)` with `no-op` on `at(b,1)`;
 - `move(l,r,p)` with `no-op` on `at(r,1)`;
 - `move(l,r,p)` with `no-op` on `hasFuel(r)`.
- Inconsistent effects.
 - It's easy we need to find those conditions that have a continuous arc and a dotted arc. One action delete `at(r,1)` another add `at(r,1)`; the first thing we can say is that the `move(r,1,p)` is incompatible with the `no-op at(r,1)` and so on... This is the easiest, just controlling continuous and dotted arcs.
 - Then we pass to *interference*; note that you can have actions that are emerging both from *inconsistent effect* and *interference*. Again we notice that we are extracting all these informations directly from the syntax, no semantics at all! Finally we should investigate also *competing needs* but let's go with *inconsistencies* about propositions.

Inconsistencies between propositions:¹⁵⁰

¹⁵⁰Why two propositions are inconsistent? Well, if one is `p` and the other is `not p` of course they're inconsistent (one is the negation of the other) but also two propositions are inconsistent if any path to reach the first is inconsistent with any path to reach the second (if all the ways to reach them are mutually exclusive).

- $\text{in}(a, r)$ and $\text{at}(r, p)$;
- $\text{in}(b, r)$ and $\text{at}(r, p)$,¹⁵¹
- $\text{in}(a, r)$ and $\text{at}(a, l)$;
- $\text{in}(b, r)$ and $\text{at}(b, l)$;
- $\text{at}(r, p)$ and $\text{at}(r, l)$;
- $\text{at}(r, p)$ and $\text{hasFuel}(r)$.

This is what asked during the exam: building two levels and showing inconsistencies between actions and propositions.

Clearly *Graphplan* goes on building *action levels* and *proposition levels*... when do we stop? When you find a *level that contains the GOAL* (all the propositions that are required in the *GOAL*; in our example $\text{at}(a, p)$, $\text{at}(b, p)$); of course a thing to consider is that these 2 should not be incompatible of course!

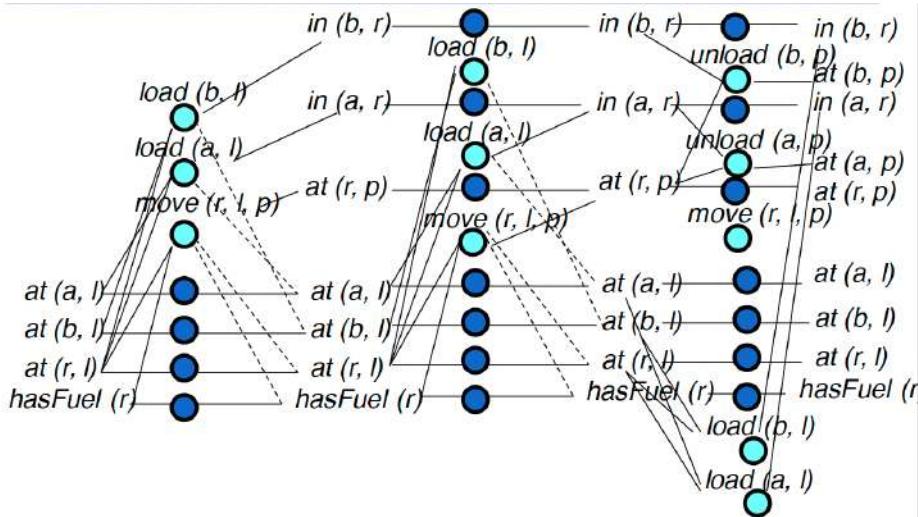


Figure 6.64

So as soon as you find a plan in which your goal is true you stop. So now we have an expanded data structure up to a level that contains all possible plan squeezed. Now we ask ourselves: let's try proceeding *backward* to extract a *plan* which is called a *valid plan* from this structure. But are we guaranteed that we find it? No, and in case we don't find it, then we expand additional levels asking at each expansion: now it is possible to find a valid plan? So if there is no plan this could run indefinitely but this is not a problem of the algorithm, is a feature of the problem itself. So if there is no plan you can run indefinitely, but if a plan exists not only you find this plan but it is also the shortest possible one: because you start trying to find it as soon as you have a level in which the *GOAL* is true and doing so step by step. As soon you find the *GOAL* you have a chance to find a valid plan.

¹⁵¹Are inconsistent because they come from the $\text{load}(b, l)$ and the $\text{move}(r, l, p)$ that are inconsistent at the previous level. We are propagating forward the inconsistencies.

Planning Graph: algorithm

The planning graph is built as follows.

- All true propositions in the initial state are inserted in the first proposition level.
- Creation of *action level*:
 - for every operator and every way to unify its preconditions to propositions in the previous proposition level, enter an action node IF two propositions are not labeled as *mutually exclusive*;¹⁵²
 - in addition, for every proposition in the previous proposition level, add a **no-op** operator;
 - check if the action nodes do not interfere each other otherwise mark them as mutually exclusive.¹⁵³
- Creation of *proposition level*:
 - for each action node in the previous level, add propositions in its *add list* through solid arcs and add dotted arcs connected to the propositions in the *delete list*;
 - do the same process for the **no-op** operators;
 - mark as mutually exclusive two propositions such that all the ways to achieve the first are incompatible with all the ways to reach the second.

Algorithm for the construction of the planning graph.

1. *Initialization*: all true propositions in the initial state are included in the first proposition level.
2. *Creating an action level*:
 - (a) \forall operator/action \forall every way to unify its preconditions to *not mutually exclusive* propositions in the previous level, enter an action node;
 - (b) for every proposition in the previous proposition level, enter a **no-op** operator;
 - (c) identify the mutual exclusive relationship between the newly constructed operators.
3. *Creating a proposition level*:
 - (a) for each action node in the previous action level, add the propositions in his *add list* through solid arcs;
 - (b) for every no-op in the previous level, add the corresponding proposition;
 - (c) for each action node in the previous action level, link propositions in his *delete list* by dashed arcs;
 - (d) identify incompatible propositions.¹⁵⁴

¹⁵²Of course if you have for example **at Milano**, **at Bologna** they are mutually exclusive.

¹⁵³So basically we check for incompatibilities.

¹⁵⁴Propagating forward the incompatibilities found in the previous levels.

Extraction of a valid plan

Once the planning graph is built, we have to extract a *valid-plan*, i.e., a connected and consistent subgraph of the planning graph.

Features of a valid plan:

- actions in the same time step can be performed in any order (do not interfere);
- propositions at the same time step are not mutually exclusive;
- the last time step contains all the literals of the goal and these are not marked as mutually exclusive.

Theorems

The extraction of the planning graph is based on a set of theorems (proved in papers).

1. If there is a valid plan then this is a subgraph of the planning graph.¹⁵⁵
2. In a planning graph two actions are mutually exclusive in a time step if a valid plan that contains both *does not exist*.¹⁵⁶
3. In a planning graph two propositions are mutually exclusive in a time step if they are inconsistent, i.e. one of them *denies* the occurrence of the other.

Important consequence: the inconsistencies found by the algorithm prune paths in the search tree.

Valid plan

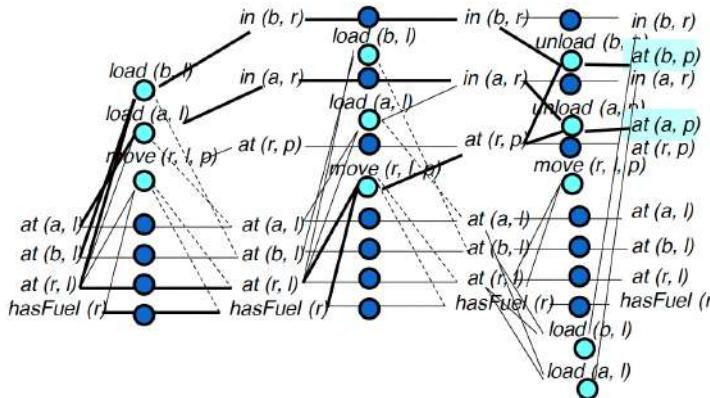


Figure 6.65

¹⁵⁵This is a very strong result because tells that you are *complete*.

¹⁵⁶This is basically linking what we've identified as '*mutual exclusion*' in the planning graph.

Basically the inconsistencies found by the algorithm prove paths in the search tree, when you're looking for a valid plan backward you have to use the inconsistencies because if you're proceeding backward and you want to select two propositions in a level but they are mutually exclusive you don't continue of course.

Let's see how to extract a valid plan using the previous example (see the figure above).

In the last level we've found `at(b,p)`, `at(a,p)` which are the propositions characterizing the *goal*.

We may try to proceed backward and find a selection (subgraph) of the planning graph that contains a valid plan.

The two `unload` are the actions that achieve `at(b,p)`, `at(a,p)`, then the first `unload` (first starting from the top) requires `in(b,r)`, `at(r,p)` while the second requires `in(a,r)`, `at(r,p)`. Then to reach `at(r,p)` we require `move(r,l,p)` and `in(b,r)` requires `load(b,l)`, and `in(a,r)` requires `load(a,l)`.

Note that `load(b,1)` and `load(a,1)` for example can be executed in any order, even in parallel if we have multiple agents. And so on, follow the bold lines backward in the figure.

Algorithm

```
function GRAPHPLAN(problem):
    graph = GRAFO_INIZIALE (problem)
    targets = GOAL (problem)
    do loop:
        if objectives not mutex last step:
            Sol = ESTRAI_ SOLUTION (graph, objectives)
            if Sol ≠ fail: return Sol
            else if LEVEL_OFF (graph): return fail
            = ESPANDI_GRAFO graph (graph, problem)
```

Figure 6.66

Let's analyze.

- The first node contains the initial planning graph. This contains only one time step (proposition level) with true propositions in the initial state. The initial graph is extracted from `GRAFO_INIZIALE (problem)`.
- The goal to reach is extracted from the function `GOAL (problem)`.
- If goals are not mutually exclusive in the last level, then the planning graph *could* include a valid plan. The valid plan is extracted through *backward search* `ESTRAI_SOLUZIONE(graph, objectives)` that provides either a solution or a failure.
 - Proceed level by level to better exploit the mutual exclusion constraints.
 - Recursive method: given a set of goals at time t the algorithms looks for a set of actions at time $t - 1$ who have such goals as add effects. *The actions should not be mutually exclusive.*
 - The search is the hybrid breadth/depth first and *complete*.

- *Memoization* (not a typo).¹⁵⁷
 - If at some step of the search, a subset of goals is not satisfiable, graphplan saves this result in a hash table. Whenever the same subset of goals is selected in the future will automatically fail.

First example: Block World

Let's see with this example also the syntax used.

- Three blocks A, B, C that are of *type OBJECT*:

```
(BlockA OBJECT)
(BlockB OBJECT)
(BlockC OBJECT)
```



Figure 6.67

- Initial state:

```
(preconds
  (On-table blockA)
  (On-table blockB)
  (On blockC blockA)
  (Clear blockB)
  (Clear blockC)
  (Arm-empty))
```

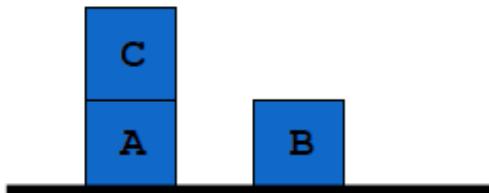


Figure 6.68

- Goal:¹⁵⁸

¹⁵⁷Sometimes we use this *memoization* that could be basically seen as a trick for making the procedure more efficient. It is a practise that is quite often used in search algorithm: so in general if you find a configuration that leads to a failure it is saved in memory and every time the configuration appears the system intercepts a failure.

¹⁵⁸Place them in *block-facts* file.

```
(effects
  (On blockA blockB)
  (On blockB blockC))
```

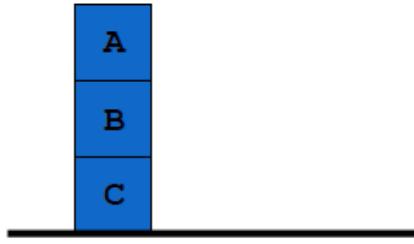


Figure 6.69

- Actions:¹⁵⁹

```
(operators
  PICK-UP
  (Params (<ob1> OBJECT))
  (preconds
    (Clear <ob1>) (on-table <ob1>) (arm-empty))
  (effects
    (Holding <ob1>)))

(operators
  PUT-DOWN
  (Params (<b> OBJECT))
  (preconds
    (Holding <b>))
  (effects
    (Clear <b>) (arm-empty) (on-table <b>)))

(operators
  STACK
  (Params (<b> OBJECT) (<underob> OBJECT))
  (Preconds (clear <underob>) (holding <b>))
  (Effects (arm-empty) (clear <b>)
    (On <b> <underob>)))

(operators
  UNSTACK
  (Params (<b> OBJECT) (<underob> OBJECT))
  (Preconds (on <b> <underob>) (clear <b>)
    (Arm-empty))
  (Effects (holding <b>) (clear <underob>)))
```

Note that the PICK-UP action has one parameter from which depend preconditions and effects.

¹⁵⁹Place them in *block-ops* file.

Given the configuration of the Initial state (see the related image) as exercise try to write the first level of the planning graph (one action level, one proposition level).

Which are the actions that are applicable on the initial state? `Unstack(c,a)` and the `Pickup(b)`, just these two. Are they compatible or incompatible? They are incompatible because one of them is deleting `Arm-empty` which is the precondition for the other and viceversa (in the sense that both actions are deleting `Arm-empty` which is a precondition for both). Then of course we also have incompatibilities with the `no-ops` since `Unstack` is deleting `on(c,a)` and it would be incompatible with the `no-op on(c,a)`, it is also deleting `Arm-empty` so it is also incompatible with the `Arm-empty`.

Fast Forward

Not only *Graphplan* has been used alone as a single algorithm but has been inserted also in other algorithms. For example is used for providing an heuristic. *Fast Forward* (FF) is a *extremely efficient* heuristic planner¹⁶⁰ introduced by Hoffmann in 2000.

- Heuristic = in each state S we compute an estimate of the distance to the goal.
- *Basic operation:* hill climbing¹⁶¹ + A^* :
 1. from a state S , examine all successors S' ;
 2. if a successor state S^* exists better than S , move on it and go back to point 1;
 3. if there is no state with a better evaluation, a complete A^* search is run, using the same heuristic.

Let's see what is the heuristic, after making a short recap regarding A^* .

A^* algorithm: short recap

Let's do a short recap of the A^* algorithm.

- Instead of only considering the distance to the goal, also consider the ‘cost’ in reaching the node n from the root.
- We expand nodes for increasing values of $f(n)$ i.e. $f(n) = g(n) + h'(n)$. Where $g(n)$ is the depth of the node, and $h'(n)$ the estimated distance from the goal.
- We choose the node to expand as the one for which this sum is smaller.

¹⁶⁰It means that we have no guarantees for *completeness* (while yes for *correctness*).

¹⁶¹Basically it is created an hill climbing which is a local search algorithm that wants to proceed towards those states surrounding that have a better evaluation function.

Heuristic function

The heuristic is based on *graphplan*. What does it do? Basically every time you want to understand which is the distance between a given state to your goal. This is a very important heuristic for planning: how far is from my goal a given state? Why should I say that a state is better than another? Essentially because it is close to the *goal node*. So the thing that you do is that you have to evaluate which is the distance and *fast forward* (FF) makes a run of *graphplan* but on a ‘*relaxed-problem*’ that does not consider the delete effect actions; so you consider actions without taking into account their delete effects; of course this *graphplan* is faster in comparison to a normal *graphplan* because not deleting anything it has no inconsistencies and therefore you just go on and you find a state where you have the *goal* and from which you compute backward finding a *valid plan* in this relaxed problem and the number of actions in the resulting plan is considered as ‘*distance*’.

- Given a problem P , a state S and a goal G , FF considers a relaxed problem $P+$ that you get from P by *neglecting delete effects* actions.

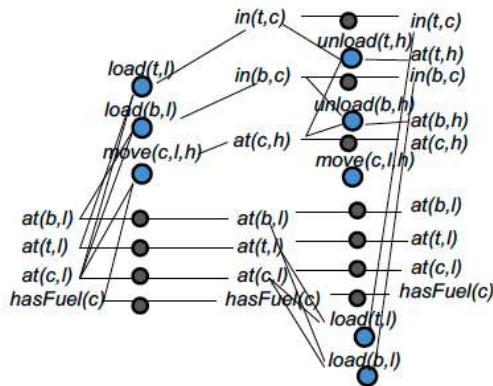


Figure 6.70

- FF solves $P+$ with *graphplan*.
- The *number of actions* in the resulting plan is used as heuristics.

FF actually uses a so-called ‘enforced hill climbing’.

```

function FF(problem): Return solution or fails
  S = Initial_state (problem)
  k = 1
  do loop:
    explore all states S 'at k steps'
    if a Better state S* is found: S = S *
    else if k can be increased: k = k + 1
    else perform complete A* search
  
```

Figure 6.71

- In practice it is a complete breadth first search.
- A solution is always found, unless the current one is not a dead end.

References

Graphplan:

- <http://www.cs.cmu.edu/~avrime/graphplan.html>
- A. M. Blum and Furst, ‘Fast Planning Through Graph Analysis’, Artificial Intelligence, 90: 281-300 (1997).

Fast Forward:

- <http://members.deri.at/~joergh/ff.html>
- J. Hoffmann, ‘FF: The Fast-Forward Planning System’, in: AI Magazine, Volume 22, Number 3, 2001, Pages 57-62

Blackbox for projects:

- <http://www.cs.rochester.edu/u/kautz/satplan/blackbox/index.html>
- SATPLAN: <http://www.cs.rochester.edu/u/kautz/satplan/index.htm>
- Henry Kautz and Bart Selman, ‘Planning as Satisfiability’, Proceedings ECAI-92.
- Henry Kautz and Bart Selman, ‘Unifying SAT-based and Graph-based Planning’, Proc. IJCAI-99, Stockholm, 1999.

6.5 Hierarchical Planning

Now we would like to conclude with other two kind of planning techniques (for which won’t be asked to solve planning exercises during the exam, but you should be able to describe them, what algorithms they have and so on...). They are *hierarchical planning* (which we are going to see in this section) and *conditional planning*.

Hierarchical Planning: description

Hierarchical planners are search algorithms that manage the creation of complex plans at *different levels of abstraction*, by considering the simplest details only after finding a solution for the most difficult ones.¹⁶²

We need a language that enables operators at different levels of abstraction.

¹⁶²As human beings, hierarchical planning is something that we do normally; suppose to plan a trip: the first thing that we have to do is to go to *New York* planning to take two flights: *from Bologna to London, from London to New York*. This can be considered our *High level plan*. Then when we have really to leave, starting preparing our luggage and everything necessary, at this point we’re creating the *details* of the big steps previously thought (what putting in our bag, how to go from home to the airport) etc. So you can see the presence of high level actions called *Macro Actions* which are translated into low level actions called *Micro Actions* which are detailed actions that can be executed by an *agent*. There is also another way to treat *hierarchical planning* that is not based on these *Macro/Micro actions*, but is based on the levels of *criticality* assigned to preconditions: this is very strange on

- Values of criticality assigned to the preconditions.
- Atomic operators *vs* Macro operators.

All operators are again defined by *preconditions* and *effects*.
Popular hierarchical planning algorithms.

- STRIPS-Like.
- Partial-Order.

Given a goal, the hierarchical planner performs a *meta-level search* to generate a *meta-level plan* which leads from a state that is very close to the initial one to a state which is very close to the goal.

The plan is then completed with a lower level search, taking account of details omitted at the previous level.¹⁶³

So a hierarchical algorithm must be able to:

- organize high (*meta-level*);
- expand abstract plans into concrete plans:¹⁶⁴
 - planning abstract parts in terms of more specific actions (planning basic level);
 - expanding already prebuilt plans.

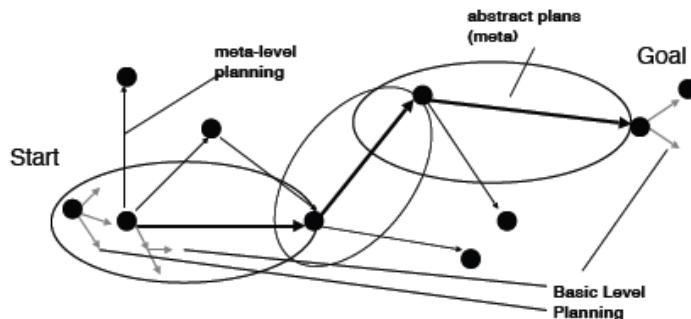


Figure 6.72

the sense that is counter intuitive. So, summarizing, we can use two ways (*Micro/Macro operators* or *criticality values*), but, in general, the thing that you do is to do an *High level plan* and then you can refine it afterwards. The *High level plan* can be built in any way we know and in every way you want, so we can use *linear planners*, *non linear planners*, *graphplan* and so on... In this volume we are going to show two examples of planning techniques used in *hierarchical planning* (but it is possible to use also other techniques): we are going to see an algorithm that is based on *STRIPS* and one that is based on *POP*, but they are working at different levels of abstraction.

¹⁶³So an higher level search creates the big blocks or, also, for example achieves the most difficult conditions to be achieved and then a lower level search refines it creating the details that are omitted at the previous level.

¹⁶⁴Concrete means that the plans are executable by the agent supposed to execute our plan. For example, *go from Bologna to London*, this pick action is not directly executable from an agent in the sense that the agent does not know the semantics, you need to give him all the operative details for executing concretely the necessary steps.

Abstrips

Ab stands for *abstract*. It consists in the less intuitive way to produce a *hierarchical planning*; less intuitive because it is based on a *critical value* associated to the preconditions of an action. The idea is the following: if we have different *goals* to be achieved, the thing that I want to do is to concentrate first on those that are most difficult to be achieved. So, if you want to define how difficult is to achieve a given condition, you can associate a *critical value* to this goal in some way. Suppose, for example, that we are in the *Block-World* (so we are putting blocks one on top of the other and so on) and we have a set of conditions `on(X,Y)`, `ontable(X)`, `clear(X)`, `handempty`, `holding(X)`. Which one do you think is the most difficult to be achieved? Intuitively `on(X,Y)` is the most difficult to achieve (consider that, for example, it is the only that takes into account two variables in comparison to the others, or also if you want to use semantics as human being this is the condition most similar to building towers), then we would have `ontable(X)`, `clear(X)`, `holding(X)` (one variable) and the simplest is `handempty` (zero variable).

So *Abstrip* is a *hierarchical planner* who enhances the Strips definition of actions with a *criticality value* (proportional to the complexity of its achievement) to each precondition.

The planning algorithm proceeds at different levels of abstraction spaces. At each level, lower level preconditions are ignored.

Abstrips fully explores the space of a certain level of abstraction before moving on to a more detailed level: *length search*.

At every level of abstraction, it generates a complete plan.

Application examples:

- construction of a building;
- organization of a trip;
- draft a top-down program.

Abstrips: methodology of solution

Let's list the steps composing the solution:

1. a threshold value is fixed;
2. all the preconditions whose criticality value is less than the threshold value are considered true; *simply a way to not consider them*;
3. STRIPS¹⁶⁵ finds a plan that meets all the preconditions whose value is greater or equal to the threshold value;
4. it then uses the full plan pattern obtained as a guide and lowers the value of the threshold;
5. it extends the plan with operators that meet the preconditions whose level of criticality is higher than or equal to the new threshold value;

¹⁶⁵At every level we can use a different planner, not necessarily STRIPS.

6. it lowers the threshold value until all the preconditions of the original rules have been considered.

It is important to give good critical values to preconditions!

Example (Algorithm Strips-Like)

Initial state:

```
clear(b), clear(c), on (c,a),
handempty, ontable(a), ontable(b)
```

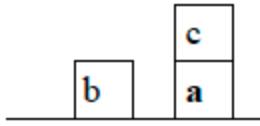


Figure 6.73

Goal: $\text{on}(c,b) \wedge \text{on}(a,c)$

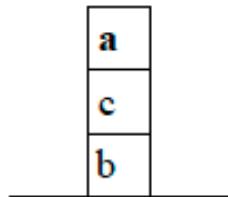


Figure 6.74

We specify a hierarchy of goals and preconditions assigning criticality values (that reflect – from top to bottom – the degree of difficulty in satisfying them):

```
on (3)
ontable, clear, holding (2)
handempty (1)
```

We use the STRIPS rules:¹⁶⁶

```
pickup (X)
PRECOND: ontable (X), clear (X), handempty
DELETE: ontable (X), clear (X), handempty
ADD: holding (X)

putdown (X)
PRECOND: holding (X)
DELETE: holding (X)
ADD: ontable (X), clear (X), handempty
```

¹⁶⁶Then, apart from taking into account abstraction levels, we proceed usually with STRIPS techniques.

```

stack (X, Y)
PRECOND: holding (X), clear (Y)
DELETE: holding (X), clear (Y)
ADD: handempty, on (X, Y), clear (X)

unstack (X, Y)
PRECOND: handempty, on (X, Y), clear (X)
DELETE: handempty, on (X, Y), clear (X)
ADD: holding (X), clear (Y)

```

First level of abstraction: we consider only the preconditions with criticality value 3 and you get the following goal stack:

```

on (c, b)
on (a, c)
on (c, b) ^ on (a, c)

```

The action **stack(c,b)** is added to the plan to meet **on(c,b)**.

Since its preconditions are all less critical than 3 they are considered to be met.¹⁶⁷

A new state is generated by simulating the execution of action **stack(c,b)**.¹⁶⁸

state description	goal stack
clear (b)	on (a, c)
clear (c)	on (c, b) ^ on (a, c)
ontable (a)	
ontable (b)	
on (c, b)	
handempty	
on (c, a)	

Figure 6.75

Note: in the description of the state only effects with criticality greater than or equal to 3 are added/deleted.¹⁶⁹ There may be inconsistencies, but this is fine!¹⁷⁰

The action **stack(a,c)** is added following the same process for **stack(c,b)**. The complete plan at the *First Level* is:

1. **stack(c,b);**
2. **stack(a,c).**

Second level of abstraction: we restart from initial state and in the goal stack we insert the initial goals, the actions computed at the *First Level* and their preconditions (along with a goal ordering).¹⁷¹

¹⁶⁷They are **holding(c)** (not true but considered true) and **clear(b)**.

¹⁶⁸We extract **on(c,b)** from the goal stack.

¹⁶⁹In this case **on(c,b)** which is in the ADD list of **stack(c,b)**.

¹⁷⁰Of course there may be! This because we've considered as true propositions that are not effectively true without considering the critical value.

¹⁷¹For example the orderings in the list below are: **holding(c)** (top) first **clear(b)** (bottom) second and **holding(a)** (top) first **clear(c)** (bottom) second.

```

holding(c)
clear(b)
holding(c) ^ clear(b)
stack(c, b)
holding(a)
clear(c)
holding(a) ^ clear(c)
stack(a, c)
on(c, b) ^ on(a, c)

```

The condition `holding(c)` is satisfiable with the action `unstack(c, X)` and the stack becomes:

```

clear(c)
on(c, X)
clear(c) ^ on(c, X)
unstack(c, X)
clear(b)
holding(c) ^ clear(b)
stack(c, b)
holding(a)
clear(c)
holding(a) ^ clear(c)
stack(a, c)
on(c, b) ^ on(a, c)

```

The preconditions of `unstack(c, X)` to be considered at the *Second Level* (`clear(c)` and `on(c, X)`) are all met in the initial state with the substitution `X/a`.

Executing the action `unstack(c, a)` on the initial state results in:

state description	goal stack
<code>clear(b)</code>	<code>clear(b)</code>
<code>clear(a)</code>	<code>holding(c) ^ clear(b)</code>
<code>ontable(a)</code>	<code>stack(c, b)</code>
<code>ontable(b)</code>	<code>holding(a)</code>
<code>handempty</code>	<code>clear(c)</code>
<code>holding(c)</code>	<code>holding(a) ^ clear(c)</code>
	<code>stack(a, c)</code>
	<code>on(c, b) ^ on(a, c)</code>

Figure 6.76

And so on until you get the full plan of the *Second Level*:

1. `unstack(c, a);`
2. `stack(c, b);`
3. `pickup(a);`
4. `stack(a, c).`

Third level of abstraction: we have the following goal stack:

```

handempty ^ clear(c) ^ on(c, a)
unstack(c, a)
holding(c) ^ clear(b)
stack(c, b)
handempty ^ clear(a) ^ ontable(a)
pickup(a)
holding(a) ^ clear(c)
stack(a, c)
on(c, b) ^ on(a, c)

```

Only the precondition `handempty` still needs to be considered.

The search for a solution to the *Third Level* in this case is simply a check: the solution at the *Second Level* is also correct for the *Third Level* and the search stops.

Homework: check it! At the *First Level* the valid plan:

1. `stack (a, c);`
2. `stack(c, b).`

Would fail causing backtracking.

Hierarchical planning: Macro/Atomic operators

This kind of hierarchical planning should be the most intuitive way of creating a hierarchical plan.

Two types of operators.

- *Atomic* operators.
 - Atomic operators represent elementary actions typically defined as STRIPS rules.
 - * Atomic operators can be directly executed by an agent.
- *Macro* operators.
 - Macro operators in turn represent a set of elementary actions:¹⁷² they are decomposable into atomic operators.
 - * Macro operators before execution should be further decomposed.
 - * Their decomposition can be precompiled or from plan.

We can have also different types of decomposition.

- *Precompiled decomposition*: the description of the macro operator also contains the *decomposition* - the sequence of basic operators to be executed at run-time. Let's see an example: suppose we have a robot for cooking; we have the traditional items of the structure plus two new things that are *decomposition* and *order*.¹⁷³

¹⁷²In a way macro operators are little plans.

¹⁷³Basically *decomposition* and *order* tell us that to achieve this macro action `Cook(X)` I have a little plan, a partial plan; since the list of basic operators can be precompiled, then you have a lot of time because when you put the action `Cook(X)` in the plan, you are putting one action instead of 4, so you save time. Furthermore note that referring to the figure below you can execute `S2` and `S3` in any order.

```

ACTION: Cook(X)
PRECOND: have(X), have(pot), have(salt) in (water, pot)
EFFECT: cooked(X)

DECOMPOSITION: S1: boil (water), S2: put (salt, water),
                S3: put (X, pot), S4: boilinWater (X)
ORDER: S1 < S2, S1 < S3, S2 < S4, S3 < S4

```

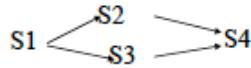


Figure 6.77

- *Planned decomposition*: the planner must perform a low-level search for synthesising the atomic action plan that implements the macro action.¹⁷⁴

The planning algorithm can be either linear or non-linear.

A hierarchical non-linear algorithm is similar to POP (partial order planner) where at each step one can choose between:

- reach an open goal with an operator (including macro operators);
- expand a macro step of the plan (the decomposition method can be precompiled or schedule).

Let's describe the *hierarchical partial order planner* function:

```

function HD_POP (initialGoal, methods, operators) returns plan
    plan := INITIAL_PLAN (start, stop, initialGoal)
loop
    if SOLUTION (plan) then return plan;
    choose between
        SN, C := SELECT_SUBGOAL (plan);
        CHOOSE_OPERATOR (plan, operators, SN, C);

        SnonPrim := SELECT_MACRO_STEP (plan)
        CHOOSE_DECOMPOSITION (SnonPrim, methods, plan)
        SOLVE_THREATS (plan)
end

```

Let's describe the logic implemented by the function above.

- As parameter of the function HD_POP we have the usuals for POP and return a plan.

¹⁷⁴Consider that these *macro actions* are composed by *decompositions* – as illustrated in the image above – that can be created by the modeller or can be also created in an automatic way; so we can either precompile in the sense that I'm a modeller and I decide that the decomposition for `Cook(X)` is `S1, S2, S3, S4` or, if we look at the figure above we can see that is a plan and we are able to create a plan automatically. So we can decide to plan automatically how to decompose a *macro action* considering that the plan is built as follow: the **precondition** of the *macro action* are our **initial state**, the **effect** are our **goal** and then we can easily form a plan inside.

- Then we start with an initial plan that, since we're considering *partial order planning* it is composed by the *false actions* `start` and `stop` where `start` has as postconditions the initial state and the `stop` has as preconditions the goal that you want to reach.
- Then, if the plan is already a solution, you return this plan.
- Otherwise you select between either a subgoal that is still open and you choose an operator for reaching it or select a macro-step and you decompose this macro-step.
- Finally solve threats.

So, basically, the difference between POP and HD_POP is only the `CHOOSE_DECOMPOSITION`, only this new point.

Example (POP-like Algorithm)

Imagine again to have a cooking robot.

Initial state:

```
have(pot), have(pan), have(oil), have(onion)
have(pasta), have(tomato), have(salt), have(water)
```

Atomic actions:¹⁷⁵

```
ACTION makePasta(X)
PRECOND: have(pasta), cooked(pasta), Sauce(X)
EFFECT: made(pasta, X)

ACTION boil(X)
PRECOND: have(X), have(pot), in(X, pot), plain(X)
EFFECT: boiled(X)

ACTION boilinWater(X)
PRECOND: have(X), have(pot), in(water, pot), boiled(water),
          in(salt, water), in(X, water)
EFFECT: cooked(X)

ACTION cookSauce(X)
PRECOND: have(X), have(pan), in(onion, pan), fried(onion),
          in(X, oil)
EFFECT: sauce(X)

ACTION fry(X)
PRECOND: have(X), have(pan), have(oil), in(oil pan), in(X, pan),
          plain(oil)
EFFECT: fried(X)

ACTION put(X, Y)
PRECOND: have(X), have(Y)
EFFECT: in(X, Y), not plain(Y)
```

¹⁷⁵Actions that can be executed.

Goal:

```
made(pasta, tomato)
```

Macro actions:

```
ACTION: Cook(X)
PRECOND: have(X), have(pot), have(salt), in(water, pot)
EFFECT: cooked(X)
DECOMPOSITION: S1: boil(water), S2: put(salt, water),
                 S3: put(X, pot), S4: boilinWater(X)
ORDER: S1 < S2, S1 < S3, S2 < S4, S3 < S4
```

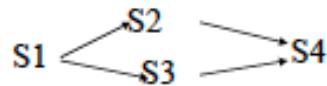


Figure 6.78

```
ACTION: MakeSauce(X)
PRECOND: have(X), have(oil), have(onion), have(pan)
EFFECT: sauce(X)
DECOM: S4: put(oil pan), S5: put(onion, pan), S6: fry(onion),
       S7: put(X, oil), S8: cookSauce(X)
ORDER: S4 < S6, S5 < S6, S6 < S7, S7 < S8
```

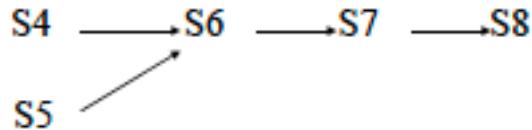


Figure 6.79

Initial empty plan:

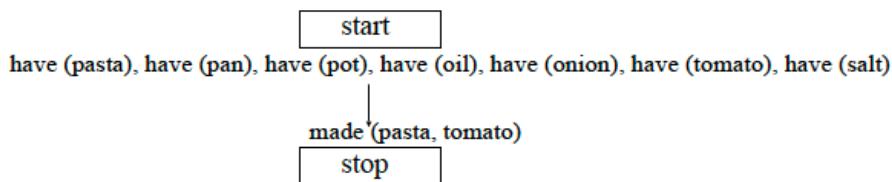


Figure 6.80

At every step you can choose between:

- reach an open goal with an operator (including macro operators);
- expand a macro step of the plan (the decomposition method can be precompiled or already planned).

Complete plan:

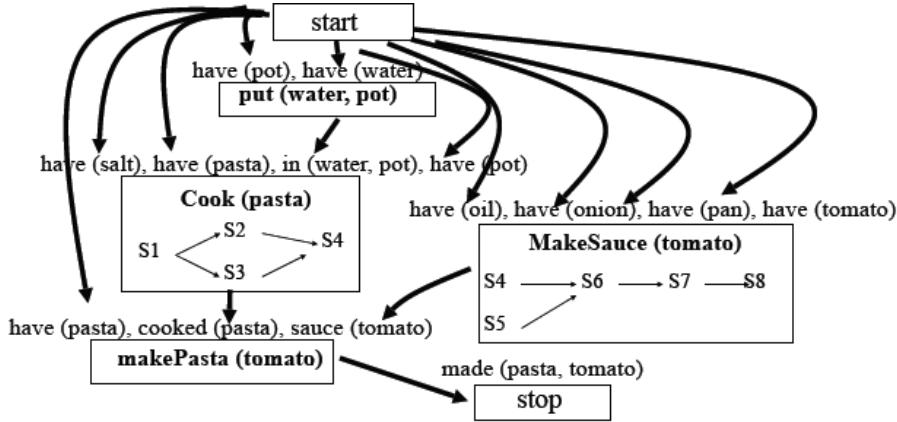


Figure 6.81

A possible final plan:¹⁷⁶

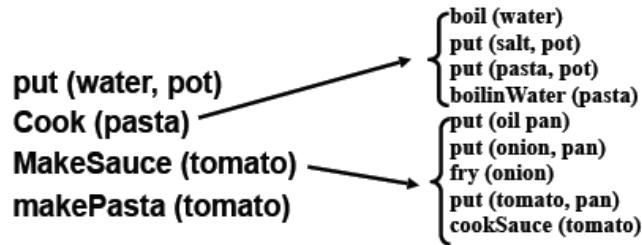


Figure 6.82

Decomposition

To ensure the decomposition is safe, some properties must be guaranteed.
Constraints on the decomposition.

- If the A macro action has the effect X and is expanded with the plan P:¹⁷⁷
 - X must be the effect of at least one of the actions in which A is decomposed and it should be protected until the end of the plan P;
 - each precondition of the actions in P must be guaranteed by the previous actions in P or it must be a precondition of A;
 - the P actions must not threaten any causal link when P is substituted for A in the plan.

¹⁷⁶Looking to the figure below the arrows are indicating that at this point we can *open the box* of macro actions; but attention when we decompose, see next section.

¹⁷⁷The set of micro actions through the macro action is expanded.

Only under these conditions you can replace the macro action A with the plan P . Replacing A with P :

- When replacing A with P , the *orderings* and *causal links* should be added.
 - Orderings:
 - * for each B such that $B < A$ then $B < \text{first}(P)$ is imposed (first action of P);¹⁷⁸
 - * for each B such that $A < B$ then $\text{last}(P) < B$ is imposed (last action of P).
 - Causal links:
 - * if $\langle S, A, C \rangle^{179}$ is a causal link in the initial plan, then it must be replaced by a set of causal links $\langle S, Si, C \rangle$ where Si are the actions of P that have C as a precondition and no other step of P before it has C as a precondition.
 - * if $\langle A, S, C \rangle$ is a causal link in the initial plan, then it must be replaced by a set of causal links $\langle Si, S, C \rangle$ where Si are the actions that have C as an effect and no other step of P after it has the effect C .

Execution

Generative planners¹⁸⁰ build plans that are then executed by an *executing agent*. Possible problems encountered during execution.

- An action should be executed and its preconditions are not satisfied:
 - incomplete or incorrect knowledge;
 - unexpected conditions;
 - the world changes independently from the planner.
- Action effects are not the one expected:
 - errors of the executing agent;
 - non deterministic/unpredictable effects.

While executing, the agent should *perceive* the changes in the world and *Act* accordingly.

Some planners run under the hypothesis of *Open World Assumption* as opposed to the Closed World Assumption: they consider that the information that is not explicitly stated in a state is not false, but *unknown*.

The unknown information can be retrieved via *sensing actions* added to the plan.

Sensing actions are modeled as causal actions.

¹⁷⁸It can be possible that P has two actions at the beginning like in the `MakeSauce(tomato)` of the previous example. In that case the ordering is imposed for both.

¹⁷⁹Causal link between S and A because of C .

¹⁸⁰All the planners seen so far are generative planners meaning that they take pictures of our world, they plan offline and then, when they have finished, they provide this plan to the executing agents which formally execute it.

The preconditions are the conditions that must be true to perform a certain observation, while postconditions are the result of the observation.¹⁸¹
 Two possible approaches.¹⁸²

- Conditional Planning.
- Integration between Planning and Execution.

6.6 Conditional Planning

Conditional Planning: description

A *conditional planner* is a search algorithm that generates various alternative plans for each source of uncertainty¹⁸³ of the plan.

A *conditional plan* it is therefore constituted by:

- causal actions;¹⁸⁴
- sensing actions for retrieving unknown information;¹⁸⁵
- several alternative partial plans of which only one will be executed depending on the results of the observations.

Example: Conditional Planning

Imagine to have a robot able to inflate a tire in case this is flat, able to implement some *causal actions* like for instance `remove(X, Y)` (if you have a tire X on a given hub Y and X is not intact you can remove the tire X) or `puton(X, Y)` (the opposite of `remove(X, Y)`) and so on, plus obviously *sensing actions* which are what characterize *conditional actions*.

Causal actions:

```
remove (X, Y)
PRECOND: on (X, Y), not intact (X)
EFFECT: not on (X, Y), off (X), clearHub (Y)

puton (X, Y)
PRECOND: off (X), clearHub (Y)
EFFECT: on (X, Y), not off (X), not clearHub (Y)

inflate (X)
PRECOND: intact (X), flat (X)
EFFECT: inflated (X), not flat (X)
```

Sensing action:

```
check (X)
PRECOND: tire (X)
EFFECT: knowsWhether (intact (X))
```

¹⁸¹As you can imagine these *sensing actions* introduce a very big complexity in terms of memory and computational costs.

¹⁸²See next section for more details.

¹⁸³So a very big number of plans, quite exponentially.

¹⁸⁴Traditional actions seen so far, actions that modify the world.

¹⁸⁵They don't modify the state.

Initial state:

```
on (tire1, hub1), flat (tire1), inflated (spare),
off (spare), tire(tire1)
```

Goal:

```
on(X, hub1), inflated (X)
```

Let's see how it works. Let consider a conditional planner based on POP (partial order planners).

A traditional planner (without sensing actions) would produce the following plan:

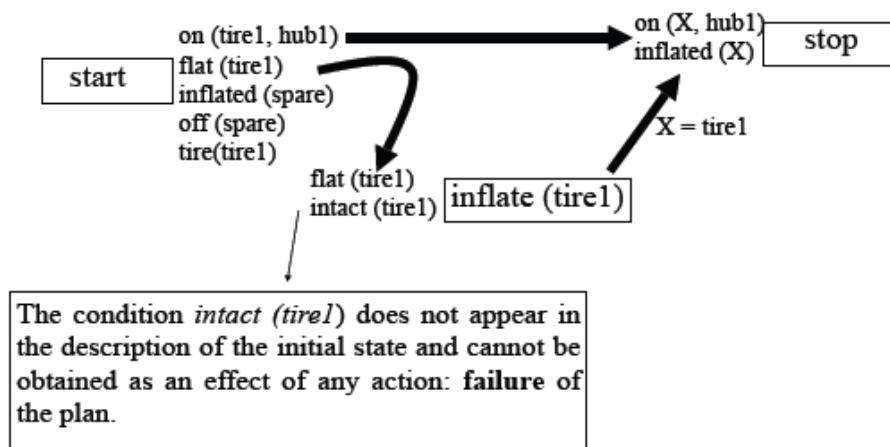


Figure 6.83

As usual we start with an empty plan. Then we consider the `stop` and the two goal conditions as its preconditions *i.e.* `on(X, hub1)`, `infalted(X)`. For the `on(X, hub1)` we have on the initial state that is already true `on(tire1, hub1)`; then regarding `infalted(X)`, since we identified $X = \text{tire1}$ for the previous condition, becomes `infalted(tire1)` which has `flat(tire1)` and `intact(tire1)` has preconditions. The condition `intact(tire1)` does not appear in the description of the initial state and cannot be obtained as an effect of any action: *failure of the plan*.

The other alternative would be to identify $X = \text{spare}$ since in the initial state we already have `infalted(spare)`. But upon backtracking the plan fails again as we get $X = \text{spare}$: remove `(tire1, hub1)`, puton `(spare, hub1)` as the precondition `not intact (tire1)` cannot be achieved in any way.

Using the sensing action one can build a *conditional plan*:

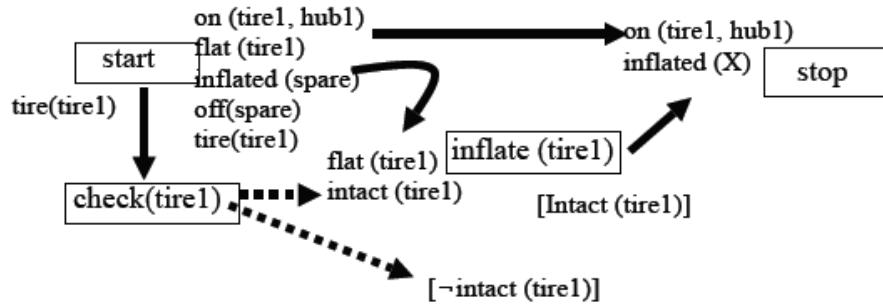


Figure 6.84

Looking at the image above and specifically to the dashed arrows we can understand the term *conditional*: *conditional* since here with the `check(tire1)` we can have two possible outcomes: `intact(tire1)` and therefore this plan is complete, or `not intact(tire1)`.

So:

- `inflate(tire1)` is the correct plan in only one executing scenario: a *context* in which `intact (tire1)` is true in the initial state;
- we should generate a copy of the goal for every other executing scenario and generate a corresponding plan for each of them;
- we have an exponential number of plans.

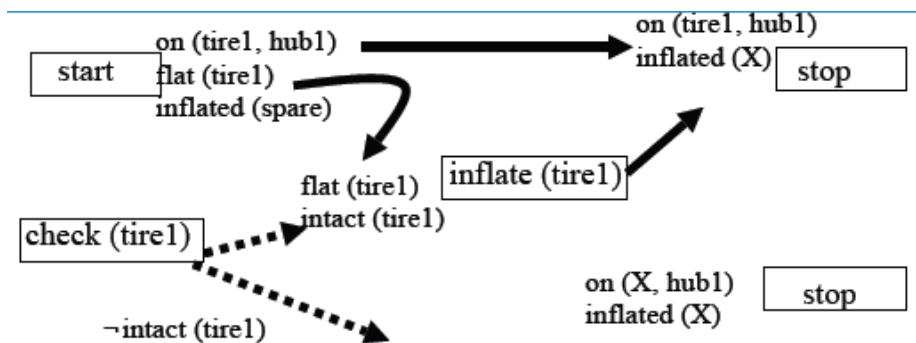


Figure 6.85

In the image above you can see that we reproduce the `stop` twice and therefore a copy of the goal.

Conditional plan:¹⁸⁶

¹⁸⁶As you can see below we have two plans (in one conditional plan): one executed if the sensitive action responds with `intact(tire1)` and the other if responds with `not intact(tire1)`.

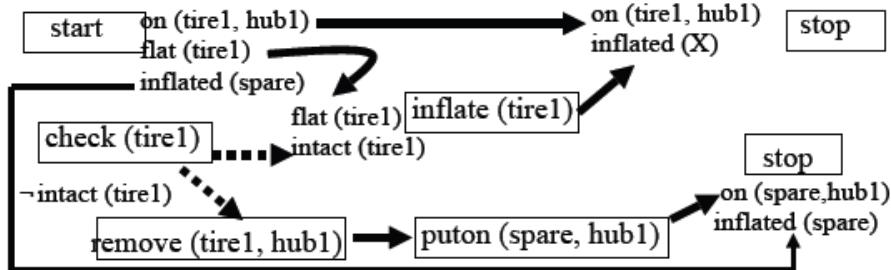


Figure 6.86

Conditional Planning: limitations

Let's list the most important limitations by which conditional planning can be affected:

- combinatorial *explosion of the search tree* with high number of alternative contexts;¹⁸⁷
- a comprehensive plan which takes account every possible contingency might require a lot of *memory*;¹⁸⁸
- not always all alternative contexts are known in advance;
- often conditional planners are associated with *probabilistic planners* that plan only for the most probable contexts.¹⁸⁹

Contingency planners

Let's list some contingency planners.

- *Cassandra* deterministic contingency planner.
- *Buridan* builds plans that have a probability greater than a certain threshold:
 - CBURIDAN;
 - algorithm <https://www.aaai.org/Papers/AIPS/1994/AIPS94-006.pdf>
 - *action representation Draper et al. 1994*: a probabilistic model of action for least commitment planning with information gathering. In proceedings of the Tenth International Conference on Uncertainty in Artificial Intelligence, pp. 178-186 Seattle, WA Morgan Kauffman.

Systems interleaving planning and execution:

- IPFM;

¹⁸⁷Suppose you insert n sensing actions with 2 possible outcomes each, we would have 2^n plans, an exponential number!

¹⁸⁸Of which only one will be executed at the end!

¹⁸⁹See next section.

- SAGE;
- XII.

Reactive planning (Brooks - 1986)

Brooks had a simple intuition that shocked the community working on planning: why we should plan when can simply react? Plan in a world continuously changing is very difficult and unpredictable; so just react to the interactions with the world.

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

- they observe the world in the planning stage;
- they acquire unknown information;
- they monitor the implementation of actions and check the effects;
- they interleave planning and execution.

Pure reactive systems do not plan, they only react as triggers to world variations.

Pure reactive systems

They have access to a knowledge base that describes what actions must be carried out and under what circumstances. Choose one action at a time, without any lookahead activity.

A thermostat uses the simple rules:

1. if the temperature T of the room is K degrees above the threshold T_0 , turn the air conditioner on;
2. if the room temperature is below T_0 of K degrees, turn the air conditioner off.

Advantages.

- They are able to interact with the real system. They are robust in domains for which it is difficult to provide complete and accurate models.
- They do not use models, but perceive world changes. That's why they are also extremely fast in responding.

Downside.

- Their performance in predictable domains that require reasoning and deliberate is quite low (*e.g.* Chess) as they are not able to generate plans.

Hybrid systems

Modern responsive planners are *hybrid*, integrate a *generative approach* and a *reactive approach* in order to exploit the computational capacity of the first and the ability to interact with the system of the second thus addressing the problem of the execution.

A *hybrid planner*:

- generates a plan to achieve the goal;
- checks the preconditions of the action that is about to run, and the effects of the action that just executed;
- backtracks the effects of an action (importance of action reversibility) and reschedules in case of failures;
- corrects the plans if unforeseen external events occur.

Chapter 7

Practical implementations of Planning

7.1 Linear Planning: STRIPS

Short summary on STRIPS

Let's recap the main concepts about STRIPS.

- STRIPS is a *linear planner* based on the *Closed World Assumption* (everything written in the initial state is true, what is not written is false).
- STRIPS produces a completely ordered sequence of actions and it is based on two data structures: *current state*; *goal stack*; we mainly work on the *goal stack* that proceeds backward from the goal to the initial state and then we proceed forward every time we extract from the *goal stack* one action. Specifically the thing is the following: from the *goal stack* you can select a goal that you have to achieve, then the first thing you have to do is to check whether this goal is satisfied in the *current state* and if it is, you can remove it (possibly by performing some unifications meaning that if the goal contains variable then you can substitute these variables with constants); while if it not satisfied in the *current state* then you have to select one action that has among its effects the goal that we want to reach. Then you have to insert in the stack the action and the preconditions list (conjunction of all preconditions of the action). Then when you extract a conjunction, we need to lead the conjunction define one order in which you want to solve. When you extract instead an action, then this is the only time in which your state changes because if you have an action that you can pop from the stack it means that all preconditions are true in the current state and therefore you can execute the action. That's the idea. Remember to leave on the stack the conjunctions because when you have checked and achieved every single part of the **and**, the conjunctions, you have to control that all of them are still valid; why that? Because there are goals that are interacting and therefore a second goal can destroy a previous goal.

Exercise 1

Now we will see the following exercise; this is very similar to those that we've seen up to now, because basically we've the traditional `pickup`, `putdown`, `stack`, `unstack`, but in the `ontable` we've three positions `p1`, `p2`, `p3`, so we have an additional argument for `ontable` dedicated to the positions; and so also `pickup` and `putdown` have another argument `Pos` indicating the position. So, let's start.

Given the initial state (where `p1`, `p2`, `p3` are three positions on the table):

```
[ontable(a,p1), ontable(d,p3), on(c,d), clear(a),
 clear(c), empty(p2), handempty]
```

Actions are modelled as:

```
pickup(X,Pos)
PRECOND: ontable(X,Pos), clear(X), handempty
DELETE: ontable(X,Pos), clear(X), handempty
ADD: holding(X), empty(Pos)

putdown(X,Pos)
PRECOND: holding(X), empty(Pos)
DELETE: holding(X), empty(Pos)
ADD: ontable(X,Pos), clear(X), handempty

stack(X,Y)
PRECOND: holding(X), clear(Y)
DELETE: holding(X), clear(Y)
ADD: handempty, on(X,Y), clear(X)

unstack(X,Y)
PRECOND: handempty, on(X,Y), clear(X)
DELETE: handempty, on(X,Y), clear(X)
ADD: holding(X), clear(Y)
```

The goal is:

```
ontable(a,p2)
```

Describe how the algorithm STRIPS finds a plan. Show *only one path* from the root to the node.¹

Look at the figure below.

We start with the two data structure *State* (which represent the current state and that at the beginning coincides with the initial state) and the *Goal Stack*; in this case the *Goal Stack* is very simple with just one goal so it is easy to choose what to select.

¹Sometimes can also be asked to show where are *choice points*. So suppose that for example when you extract a goal that you want to reach you can have two actions able to reach the goal, so you choose one but you need to highlight that there you have a choice point, so in case the algorithm backtrack up to here then we would have another path to follow. Another choice point could be a choice in the order of the conjunctions.

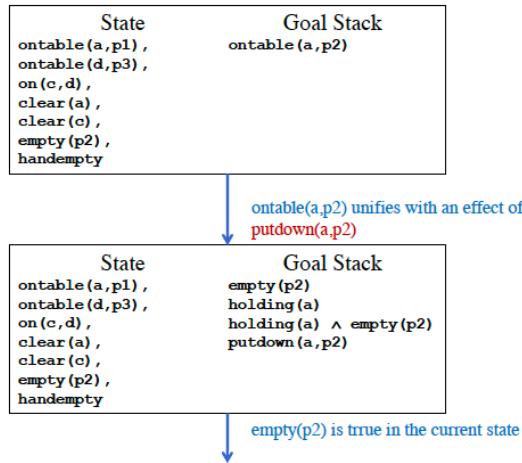


Figure 7.1

Then with the sentence:

```
ontable(a,p2) unifies with an effect of putdown(a,p2)
```

We deduce that goal `ontable(a,p2)` is one of the effect of the actions `putdown(X,Pos)` unifying `X=a, Pos=p2`; so we remove `ontable(a,p2)`, insert the action `putdown(a,p2)`, insert the and of the preconditions `holding(a) ∧ empty(p2)` and finally one possible order (in this case `empty(p2)` [top] first and `holding(a)` [bottom] second). And in this case if it is asked to show choiche points we should highlight that this is a choiche point because we could have the inverse `holding(a)` [top] first and `empty(p2)` [bottom] second.

Then with the sentence:

```
empty(p2) is true in the current state
```

You extract `empty(p2)` which is already true in the current state. So you simply remove it, you don't need to do any unification being no variables.

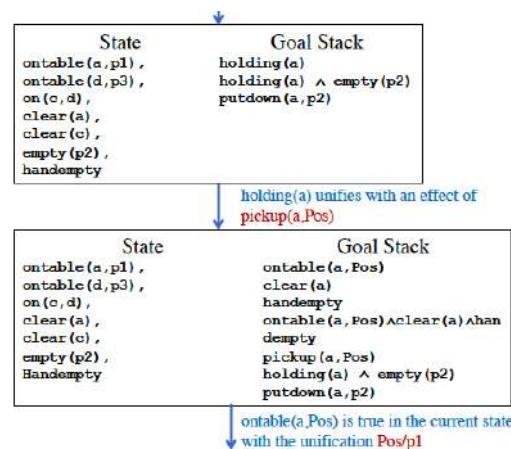


Figure 7.2

Looking above the sentence:

```
holding(a) unifies with an effect of pickup(a,Pos)
```

We extract `holding(a)` which is not true in the current state, so we extract `holding(a)` but we need to select basically one action that has `holding(a)` in its ADD LIST; here we can choose between 2 actions *i.e.* one is `pickup(X,Pos)` and the other is `unstack(X,Y)`.² In this case being humans we have in mind a meaning of the situation, semantics, so we choose `pickup` but remember always that the algorithm has no semantics at all but only syntax. So `pickup(a,Pos)` has three preconditions `ontable(a,Pos) ∧ clear(a) ∧ handempty` and then we choose one possible order: `ontable(a,Pos)` first, `clear(a)` second, `handempty` third. Another interesting thing to highlight is that you don't know where `a` is, so you leave `Pos` not unified at this stage, you know that you have to `pickup a` but from what position is something you decide later (when you will select the different conditions that have `Pos` in their arguments).

Then with the sentence:

```
ontable(a,Pos) is true in the current state with the
unification Pos/p1
```

We intend that only at this point we can unify `Pos` with a constant which is `p1` because is the current state that suggest us. So we remove `ontable(a,Pos)` from the Goal Stack and substitute `Pos` with `p1` in the other items.

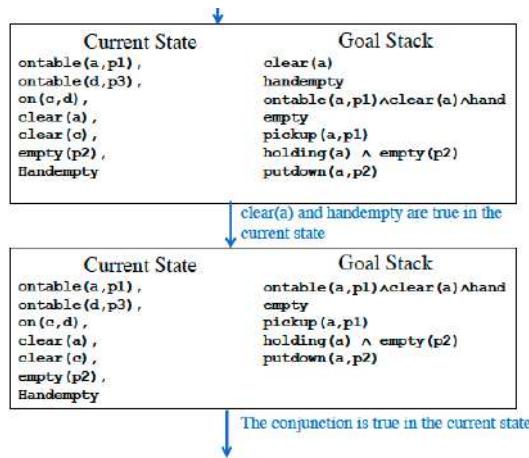


Figure 7.3

Then with the sentence:

```
clear(a) and handempty are true in the current state
```

We refer to the fact that we extract `clear(a)` which is true in the current state so we remove it from the Goal Stack. The same with the following `handempty`. Then with the sentence:

²So another choice point; if asked we need to highlight that here there is a choice point and that we could have chosen `unstack(X,Y)` with the unification `X=a`.

The conjunction is true in the current state

We refer to the fact that we remove the conjunction from the Goal Stack.

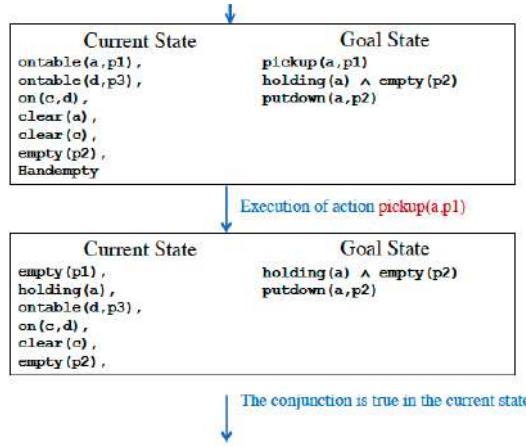


Figure 7.4

Then with the sentence:

Execution of action `pickup(a,p1)`

We refer to the fact that we can remove it. Note that this is the first and only time in which we can change the Current State (when – in general – we extract an action that is executable *i.e.* that has constants as parameters) and we move forward: so we modify the current state removing all the conditions that are in the DELETE LIST of `pickup(a,p1)` and adding all the conditions that are in its ADD LIST.

Then with the sentence:

The conjunction is true in the current state

We refer to the fact that we remove the conjunction and remains to execute the last executable action `putdown(a,p2)`.

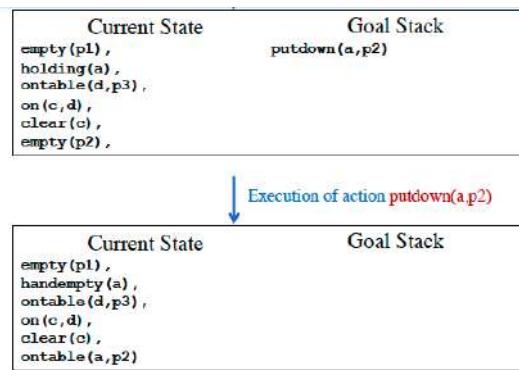


Figure 7.5

Then with the sentence:

```
Execution of action putdown(a,p2)
```

We refer to the fact that we have reached the goal that we wanted *i.e.* `ontable(a,p2)`.

7.2 Non linear Planning

Exercise 1

Let's see an exercise that is pretty the same of what seen before in the previous section, but this time solved with a *non linear planner*; the only thing that changes is that in the *initial state* positions p1, p2, p3 are occupied.

A *non linear planner* starts from an empty plan where you have a *starting action* (formally the START itself) that has as postconditions the list describing the initial state and a *final action* (formally the STOP itself) that has as precondition the *goal*; then you proceed backward from this final state: every time you select/achieve a goal that is not still open you enforce an establishment (*Modal truth criterion*) meaning that we find an action whose effects contain a condition that matches with the goal, so you insert the action considering also causal links controlling if the various causal links are threatening or not (causal links are not ordered). Remember that here during the plan construction the order in which you select the goals of the various steps is not relevant.

Let's start. Given the following initial state:

```
ontable(a,p1), ontable(d,p3), on(c,d), ontable(b,p2),
clear(a), clear(c), clear(b), handempty
```

And the goal:

```
ontable(a,p2)
```

Actions are described as follows:

```
pickup(X,Pos)
PRECOND: ontable(X,Pos), clear(X), handempty
DELETE: ontable(X,Pos), clear(X), handempty
ADD: holding(X), empty(Pos)

putdown(X,Pos)
PRECOND: holding(X), empty(Pos)
DELETE: holding(X), empty(Pos)
ADD: ontable(X,Pos), clear(X), handempty

stack(X,Y)
PRECOND: holding(X), clear(Y)
DELETE: holding(X), clear(Y)
ADD: handempty, on(X,Y), clear(X)

unstack(X,Y)
PRECOND: handempty, on(X,Y), clear(X)
DELETE: handempty, on(X,Y), clear(X)
ADD: holding(X), clear(Y)
```

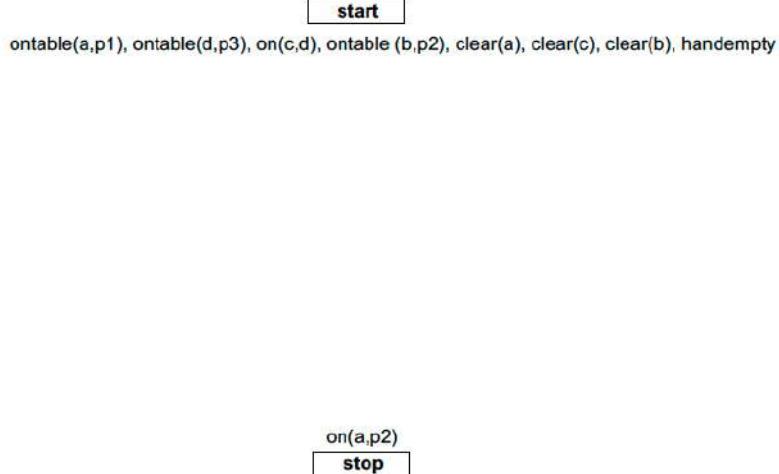


Figure 7.6

In the figure above there is an error since the goal (precondition of the formal action STOP) written is `on(a,p2)` but should be `ontable(a,p2)`.

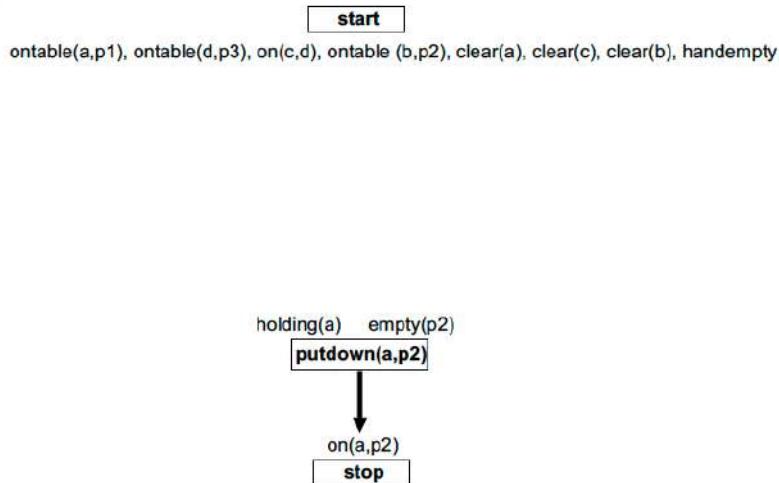


Figure 7.7

Looking at the figure above: the `putdown(a,p2)` has `ontable(a,p2)` as an effect so we choose it listing also its preconditions.

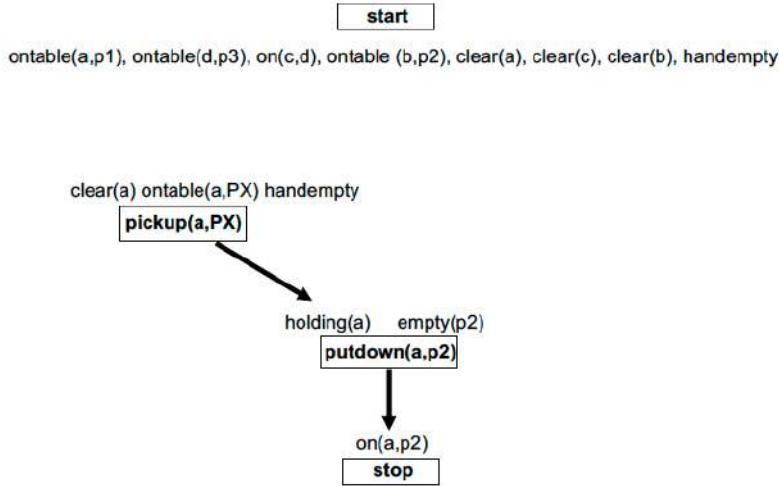


Figure 7.8

Looking at the figure above: to achieve `holding(a)` we basically follow the same path taken in STRIPS.

In the `pickup(a,PX)`, `PX` is a variable indicating the position in which `a` is; but we don't know yet where.

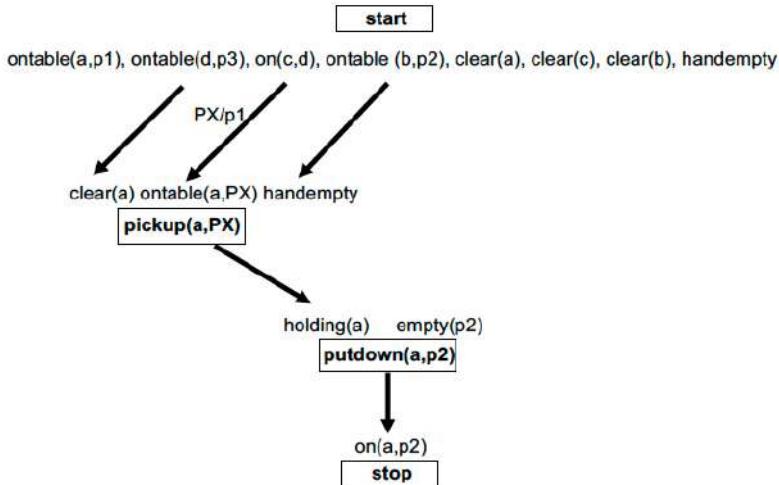


Figure 7.9

Looking at the figure above: in the *initial state* we have `clear(a)`, `ontable(a,p1)` so we unify `PX` with `p1` only at this point, and then we have another *causal link* that links the `START` with the `pickup(a,PX)` because of `handempty`. So we have 3 causal links here at this point. At this point consider that it is useless looking for threats since we have only a single chain and we cannot have threats if we don't have 2 actions that are not ordered (and here at this point are all ordered).

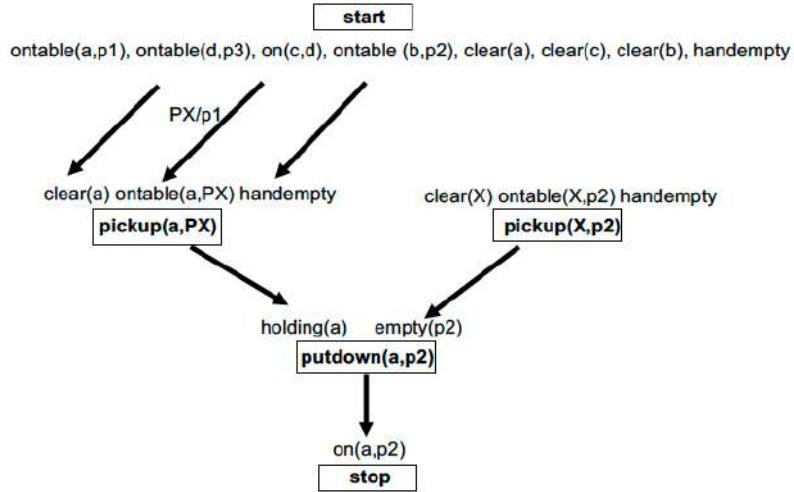


Figure 7.10

Looking at the figure above: now we have to achieve `empty(p2)`.

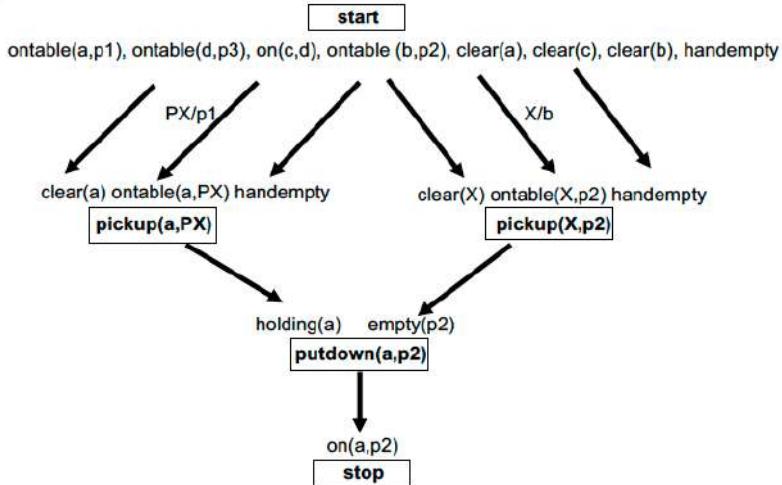


Figure 7.11

Establishment:³ all open goals (namely initial goals and action preconditions) have been achieved. Is the plan safe? We have to search for threats.

- The action `pickup(a,p1)` threatens the causal link:

– `<start, pickup(b,p2), handempty>`.⁴

³Remember that *establishment* is one of the five steps of the *modal truth criterion*.

⁴Remember an action cannot be threatened by another action. An action can threaten a causal link (why that? Because of the order).

- The action `pickup(b,p2)` threatens the causal link:

– `<start, pickup(a,p1), handempty>`.⁵

We decide to order them, and add a *white knight*.⁶

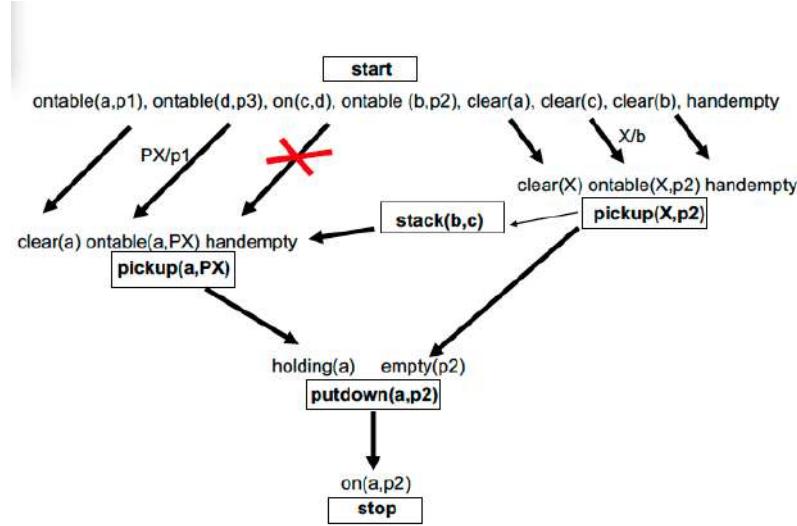


Figure 7.12

White Knight: so a *white knight* is an action that re-establish `handempty`; so we use `stack(b,c)`. Be carefull here: we decided to put `pickup(a,p1)` after the `pickup(b,p2)` basically removing the causal link `<start, pickup(a,p1), handempty>` because now this `handempty` is no longer brought to the action from the START (so from the formal action START but from `stack(b,c)`).

Notice and remember that the thinner arrow from `pickup(X,p2)` to `stack(b,c)` in the figure above indicates an *ordering constraint* (and not a causal link). So we ordered `pickup(b,p2)` and `pickup(a,p1)` and we have to introduce a *white knight*.

So:

1. `pickup(b,p2);`
2. `stack(b,c);`
3. `pickup(a,p1);`
4. `putdown(a,p2).`

⁵START provides `handempty` to `pickup(a,p1)` while the `pickup(b,p2)` negates `handempty`, so if we put `pickup(b,p2)` in between the START and `pickup(a,p1)` of course `handempty` is not longer true and viceversa.

⁶In general, when there are these crossed threats – meaning that one action is threatening a causal link and the same kind of action is threatening the other causal link – the only thing that you can do is adding a *white knight* (there is no way to solve with an ordering).

Exercise 2

Given the following initial state:

```
robot_at(loc_a), handempty, object_at(plate, loc_a),
object_at(cube, loc_b)
```

We have to reach the goal:

```
coloured(plate, blue), coloured(cube, red)
```

Actions are modelled as follows:

```
colour(Object, Location, Colour)
PRECOND: object_at(Object, Location), robot_at(Location),
robot_has(Colour)
ADD: coloured(Object, Colour)

go(X, Y)
PRECOND: robot_at(X)
DELETE: robot_at(X)
ADD: robot_at(Y)

pickColour(C)
PRECOND: handempty
DELETE: handempty
ADD: robot_has(C)

releaseColour(C)
PRECOND: robot_has(C)
DELETE: robot_has(C)
ADD: handempty
```

Solve the problem with the POP algorithm, identifying threats and their solution during the process.

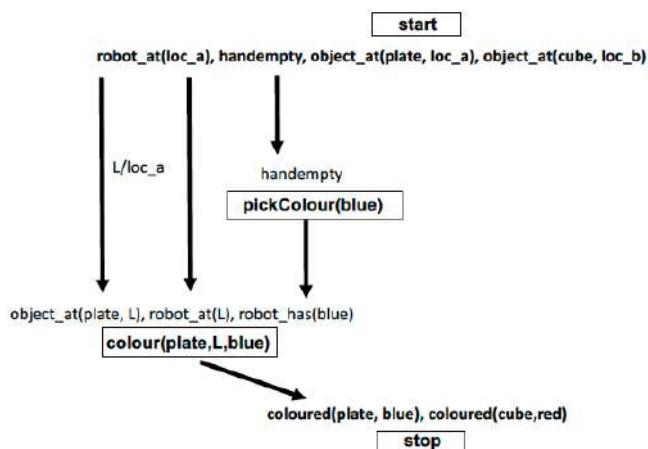


Figure 7.13

As did before, we start from an empty plan (`start`, `stop`) and we proceed backward. We select one of the items that are part of the goal and proceed backward as shown in the figure above.

Notice that regarding `object_at(plate, L)`, `robot_at(L)` here L indicates the same variable and we unify L directly from the `start` with `loc_a` because we have already these conditions in the *initial state*; two causal links.

Notice that regarding `robot_has(blue)` this is not already `true`, so we consider it as effect of the action `pickColour(blue)` which has `handempty` as unique precondition which is satisfied in the *initial state*.

Up to now, since we have only one chain, no threats; let's move on with the second chain.

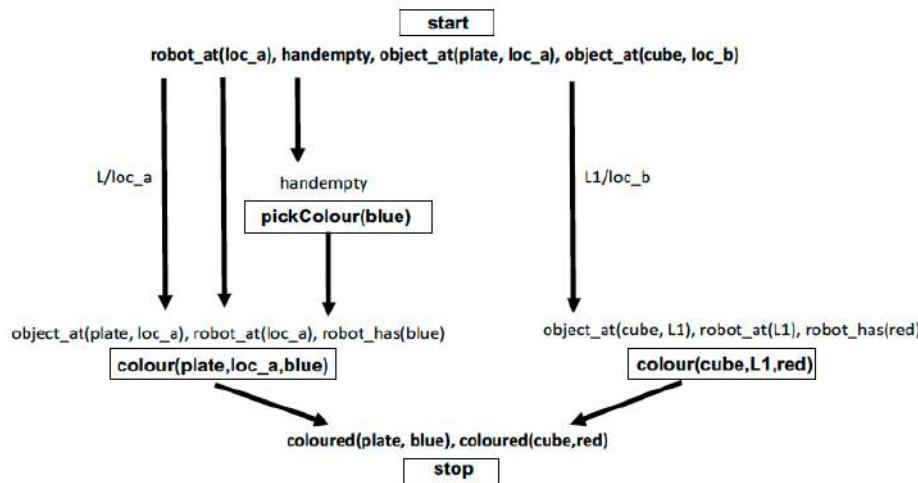


Figure 7.14

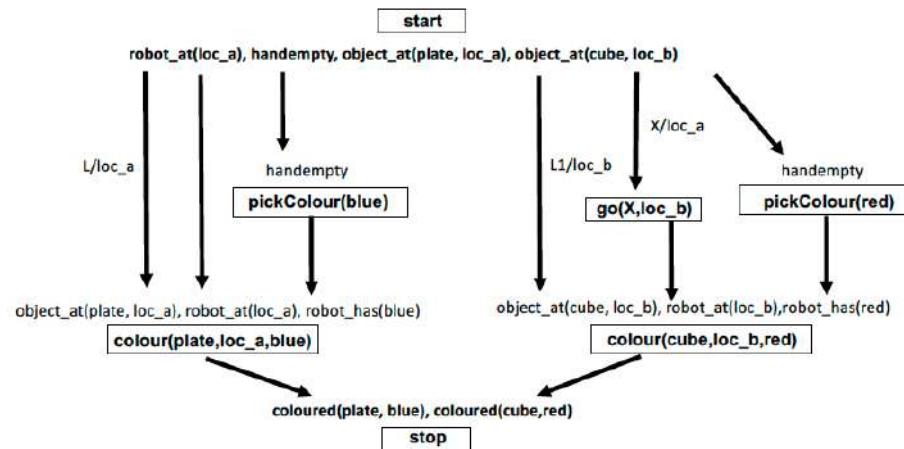


Figure 7.15

Now, in the plan all open goals are achieved, now we need to check for threats since it contains threats.

In the figure below we highlighted items of the various DELETE LIST that are important for threats.

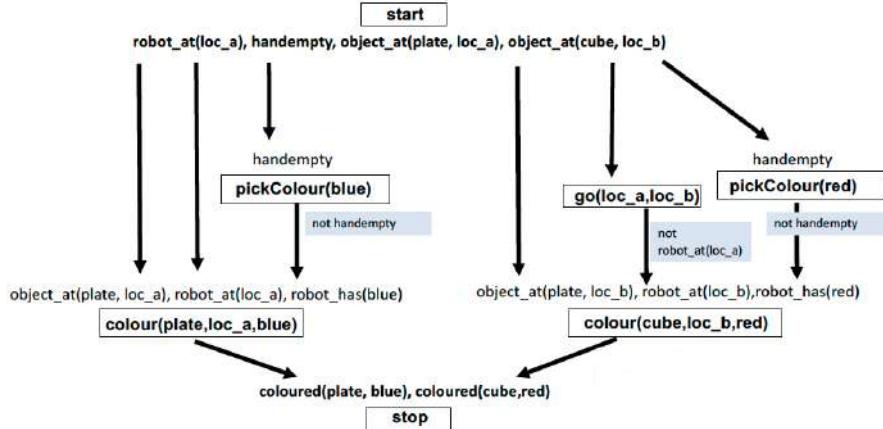


Figure 7.16

Let's list the threats.

- `pickColour(blue)` threatens the causal link:⁷
 - `<start, pickColour(red), handempty>`
- `pickColour(red)` threatens the causal link.⁸
 - `<start, pickColour(blue), handempty>`
- `go(loc_a, loc_b)` threatens the causal link.⁹
 - `<start, colour(plate, loc_a, blue), robot_at(loc_a)>`

Regarding the first two of the list we can follow the logic adopted in the previous exercise so we can solve inserting a *white knight*.

For the last one, the `go`, I can insert an ordering constraint: first I `colour` then I `go`, we can impose this ordering constraint.

Let's start with the third solving it with an ordering constraint.

In the figure below, notice and in general remember that the thinner arrow indicates an *ordering constraint* (and not a *causal link*).

⁷`pickColour(blue)` threatens `<start, pickColour(red), handempty>` because of `handempty`.

⁸The same of the previous note.

⁹`go(loc_a, loc_b)` threatens `<start, colour(plate, loc_a, blue), robot_at(loc_a)>` because of `robot_at(loc_a)`.

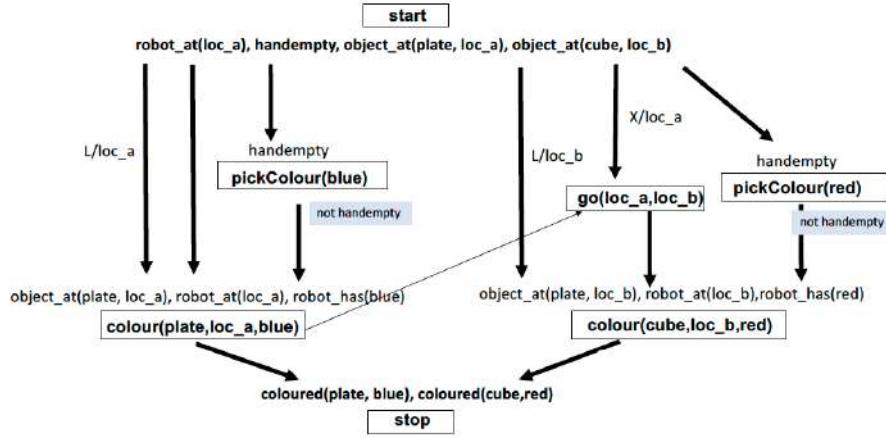


Figure 7.17

Now we order `pickColour(blue)` and `pickColour(red)` and we have to introduce a *white knight*.

So the `handempty` is not anymore justified by the formal action `start` but by the *white knight* `releaseColour(blue)`.

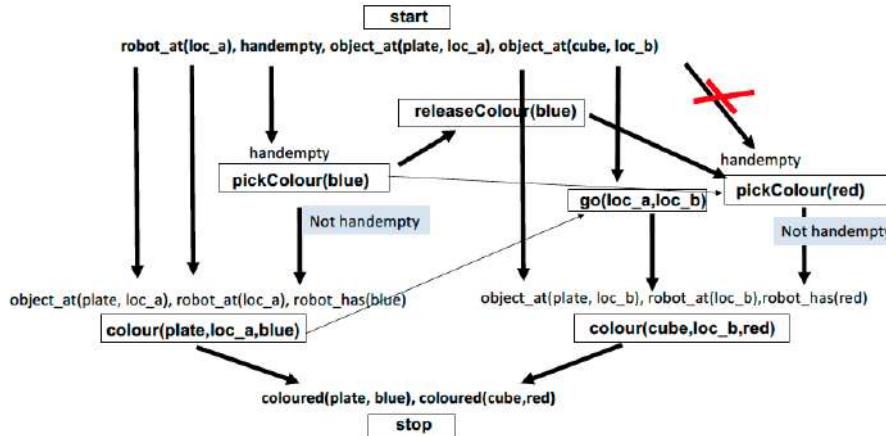


Figure 7.18

But we haven't finished yet, be carefull! We introduced the *white knight*, but we need to assure that it doesn't introduce threats. Every time we introduce something, potentially we could introduce threats, so always check!

Now `releaseColour(blue)` threatens the causal link:

- `<pickColour(blue), colour(plate, loc_a, blue), robot_has(blue)>`.¹⁰

¹⁰ `releaseColour(blue)` threatens the causal link `<pickColour(blue), colour(plate, loc_a, blue), robot_has(blue)>` because of `robot_has(blue)`.

So we order `colour(plate, loc_a, blue)` and `releaseColour(blue)`; if not ordered the `releaseColour(blue)` could possibly go between `<pickColour(blue)` and `colour(plate, loc_a, blue)`, but if so `releaseColour(blue)` will delete `robot_has(blue)` which is a precondition of `colour(plate, loc_a, blue)`; we put instead before `colour(plate, loc_a, blue)` because it has not a DELETE LIST so it has no conflicts with `releaseColour(blue)`.

Finally, to sum up:¹¹

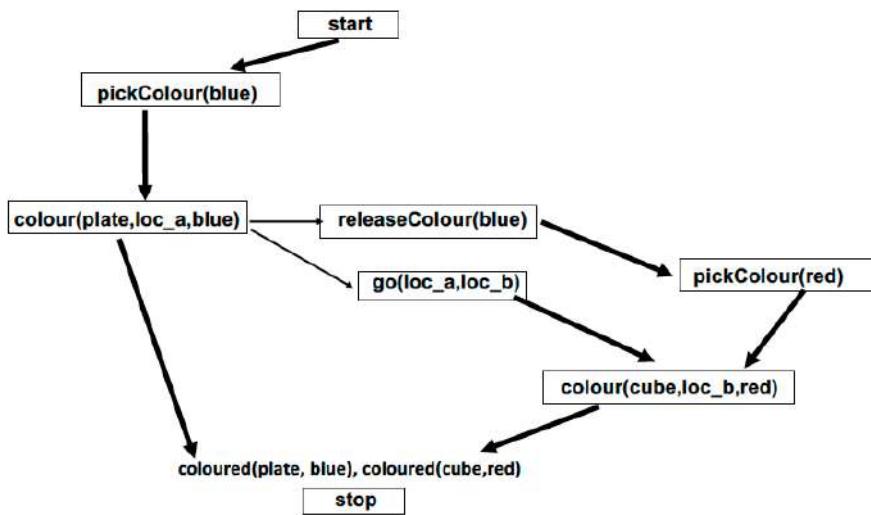


Figure 7.19

Exercise 3

Try to solve again the same exercise but this time using *graphplan*: so it is asked to write the first two levels of *graphplan* (one action level, one proposition level) and in addition you have to write which actions and which propositions are incompatible in the two levels. So let's formulate correctly the exercise.

Given the following initial state:

```
robot_at(loc_a), handempty, object_at(plate, loc_a),
object_at(cube, loc_b)
```

We have to reach the goal:

```
coloured(plate, blue), coloured(cube, red)
```

Actions are modelled as follows:

```
colour(Object, Location, Colour)
PRECOND: object_at(Object, Location), robot_at(Location),
robot_has(Colour)
```

¹¹You can avoid to graph this sum up in the exam, it is enough just to graph the entire plan as in the previous page.

```

ADD: coloured(Object, Colour)

go(X,Y)
PRECOND: robot_at(X)
DELETE: robot_at(X)
ADD: robot_at(Y)

pickColour(C)
PRECOND: handempty
DELETE: handempty
ADD: robot_has(C)

releaseColour(C)
PRECOND: robot_has(C)
DELETE: robot_has(C)
ADD: handempty

```

Show two levels of graphplan for this problem showing incompatible actions and incompatible propositions.

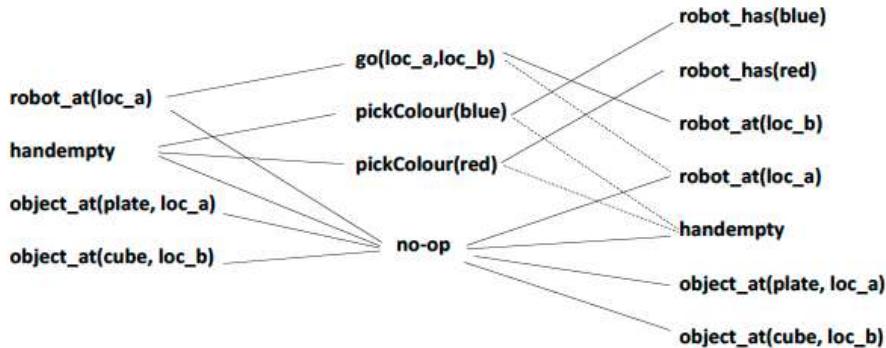


Figure 7.20

Starting from the left we can see three columns corresponding to the various levels.

- The initial level – let's say *level 0* – contains all the facts that are true in the initial state. Then the thing that we have to do is to check for each action and for each instantiation of the action with constants that we have in the description of our problem, we have to see whether their preconditions are true and, in case they are true, we can put the action with all the parameters assigned in the next level.
- In order to build this level let's start from the first action described by the problem *i.e.* `colour(Object, Location, Colour)` but since the precondition `robot_has(Colour)` is not true in the initial state we cannot insert the action due to the fact that we cannot execute it. Then we have the action `go(X,Y)`, the precondition is satisfied by `robot_at(loc_a)` but what is the second parameter Y? Remember that in *graphplan* we cannot

leave an action with variables. So which other location do we have in the initial state? we have `loc_b` and we insert it. But what would happen if we had other locations as `loc_c`, `loc_d`? We would have many instantiations of the action like `go(loc_a, loc_c)`. Then we have the `pickColour(C)`: the precondition `handempty` is satisfied. Regarding the colours (ADD) we ask ourselves: which colours are available in our domain? They are `blue` and `red`, so we can pick each of them letting to two instantiations. Then we do not insert `releaseColour(C)` since the precondition is not satisfied. In addition we have to remember to put the `no-op` operations (4 `no-op`) for each proposition true in the initial state, in order to bring them to the next level.

- Let's write now the second level: of course we can suddenly notice that the `no-op` simply transport all the propositions true in the initial state, so we simply re-write them in the list; in addition the `go(loc_a, loc_b)` deletes `robot_at(loc_a)` (dashed line) and enforces `robot_at(loc_b)`; the same logic for the other remaining actions.

Now remains to consider the *incompatibilities*:

- Incompatible actions:

Interference:

```

pickColour(blue) and pickColour(red)
pickColour(blue) and no op on handempty
pickColour(red) and no op on handempty
go(loc_a, loc_b) and no op on robot_at(loc_a)

```

- Incompatible propositions:

```

robot_at(loc_a) and robot_at(loc_b)
robot_has(blue) and handempty
robot_has(red) and handempty
robot_has(blue) and robot_has(red)

```

Let's start with analizing the actions: we have that `pickColour(blue)` requires `handempty` and the `pickColour(red)` deletes `handempty` and also viceversa, so this is the classical *interference* among two actions, so you cannot put in the same level, why? Because if one is executed after the other the precondition of the second is not guaranteed being deleted. Then the `pickColour(blue)` is also incompatible with `no-op` on `handempty` because clearly this `no-op` enforces `handempty` which is in the DELETE of `pickColour`; the same for `pickColour(red)`; notice that it is important to underline the incompatibilities with the `no-op`. Finally the `go(loc_a, loc_b)` is incompatible with the `no-op` on `robot_at(loc_a)` since is deleting such proposition.

Let's continue with the propositions: if we now bring forward the incompatibilities found in the action level then we can report on the proposition level which are the propositions that are incompatible. Let's see. Since `go(loc_a, loc_b)` is incompatible with the `no-op` on `robot_at(loc_a)` then the condition that is enforced by `go(loc_a, loc_b)` which is `robot_at(loc_b)` is incompatible with `robot_at(loc_a)`. For the same reason and logic the other incompatibilities. We finally notice that `robot_has(blue)` and `robot_has(red)` are also

incompatible at the same level since the only way to achieve them is through `pickColour(blue)` and `pickColour(red)` that are incompatible in the previous level.

Chapter 8

Constraints

8.1 Constraints satisfaction

Last topic of the course; this topic is very related with *search*, it comprehends a lot of algorithms that concern *search strategies*.

Constraints Satisfaction introduction

Many AI problems can be seen as *constraints satisfaction* problems.¹
Objective: to find a state of the problem that meets a given set of constraints.

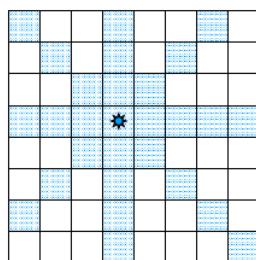


Figure 8.1

An example is the Eight Queens problem:

- given a chessboard (8×8) the problem consists in placing eight queens in order to avoid mutual attacks;

¹Basically the main feature of this problem is that you have to take decisions, but these decisions are *constraints*, the decision should be linked with a constraint. For example, I want to schedule two activities on time and one (the first listed) should come before the other (the second). So I can find many different assignments of these activities on time but the only that are feasible are those in which the first activity comes before the second. Or another example: you have to select a set of items to buy but you've a budget and the sum of their costs should be less or equal your budget. We can definitively model and solve these problems through *search strategies*: in every state of our problem you can build the solution constructively as in tree search and at each stage you have to take one decision, and one after the other at the end you've defined everything. But you cannot take every decision, you have to respect your constraint.

- the possible moves for the Queen are all the positions on the same row, column and diagonal.²

N Queens problem

The *N Queens problem* borns as a mathematical problem inspired by the game of chess (1884).

The known mathematician Gauss found 72 different solutions. The possible solutions are indeed 92.³

The problem can be generalized to a chessboard of $N \times N$ on which you must place N queens.

It has been proven mathematically that for every N greater than 3 exists a certain positive number of solutions; this number varies depending on N .



Figure 8.2

This problem can be modelled with variables, domains and constraints and solved through search strategies.⁴

²We've already seen, but basically given a chessboard 8×8 and wanting to place 8 queens avoiding that they attack each other I have to put them in a way that they don't stay on the same row or in the same column or in the same diagonal (both verse) since the possible moves of the queens are those listed above. We also saw that we can build the solution constructively, as in the *tree search* I have that the state of my problem is represented by the chessboard, at each step I place one additional queen and after 8 levels I'm done; another way I can solve this problem is through *local search metaheuristics*: so I place queens randomly and then I move them in the neighborhood so that I find a feasible solution. But now from this new topic we consider *three search algorithms*.

³If an 8×8 problem (as a normal chessboard).

⁴One possibility would be: how many decisions we have to take here? 8, so we can think at least 8 variables (maybe they can be more); so we can think, for example, that the variable of the queen number one that stays on row one as value can have the column, so if x_1 is the first variable referring to queen 1 in the row 1, then the value of this decision is the column where I put the queen (e.g. $x_1 = 1$). This could be a very good model in fact: how can you say for instance that the variables cannot stay on the same column? Simply saying that all the values assigned to the variables must be different from each other; then of course there are also some simple mathematical formula to avoid that two queens stay on the same diagonals (both diagonals indeed), we will see. There is another model where variables are not queens but cells of the chessboard: so how many variables? 64 (8×8) variables. What we should say of one cell? There is a queen or there is not a queen, so a binary information (either 0 or 1). How can you say that you don't want two queens on the same row? The sum of all cells in the same row is equal to 1. The same for the column. Then we will see there are two simple mathematical constraints that avoid diagonals. So we've provided two models meaning that the state of our search tree depends on our model of course because if we have the first model we have 8 levels of our search tree and in each level we place one queen and we basically decide to place a queen and then the other and so on. On the contrary if we

Modelling the N Queens problem

Model 1.

- The positions of the board are $N \times N$ represented by variables (8×8 , so 64 variables) x_{ij} .
- The instantiation of a variable x_{ij} to the value 1 indicates that this position is assigned to a queen, and if the value is 0 the position is free. Domain of possible values: $[0, 1]$.
- The constraints are that there cannot be two 1 simultaneously on the same row, column or diagonal.

Model 2.

- The eight queens are represented by the variables x_1, x_2, \dots, x_8 .
- The subscript refers to the column occupied by the corresponding queen.
- In the instantiation of a variable x_i the value k belonging to the set $[1, \dots, 8]$ indicates that the corresponding queen is placed on the k -th row of the i -th column.
- The variables have as a set of possible values for the integers between 1 and 8 which correspond to the rows occupied.

Constraints: two types

Let's consider *Model 2*.

- Constraints: those which bind the variables of the domain values and those who must prevent a mutual attack and which impose, then, relationships between the values assumed by the variables:
 - $1 \leq x_i \leq 8$ for $1 \leq i \leq 8$;
 - $x_i \neq x_j$ for $1 \leq i < j \leq 8$;
 - $x_i \neq x_j + (j - 1)$ for $1 \leq i < j \leq 8$;
 - $x_i \neq x_j - (j - 1)$ for $1 \leq i < j \leq 8$.
- The first constraint requires that the values of the variables of the problem are between the integers 1 and 8: *unary constraints*.⁵
- The next three constraints define relationships between the variables and, in particular, between two variables at a time: *binary constraints*.⁶
- The second requires that two queens are not positioned on the same line.
- The third and the fourth constraints concern the positions on the two diagonals starting from the initial box.⁷

use the 0, 1 variables for a cell, then our levels go up to 64. To give you an intuition the constraints can be used to reduce the number of paths to explore because not all of them are feasible. So if you use these constraints properly and actively you can avoid to exhaustively explore the entire search tree (thing that we've done up to now). This is called *constraint filtering* (remove values that are not feasible anymore from variables).

⁵Involves one variables.

⁶Involves two variables.

⁷Two queens cannot stay on the same diagonals (considering the two diagonals of the chessboard).

Scheduling as a CSP

Another example of *constraint satisfaction problem* (CSP) is *scheduling*, it is one of the most studied combinatorial optimization problem. Let's see how to model.

- *Scheduling*: assign tasks (with a certain duration) to resources at a given time.⁸ Resources can be shared.⁹
- *Variables*: start time of activities.
- *Domains*: possible start time of activities.¹⁰
- *Constraints*:
 - activities can be ordered:¹¹

```
Start1 + d1 <= Start2
```

 - the activities that use the same resource cannot overlap:¹²

```
Start1 + d1 <= Start2 or Start2 + d2 <= Start1
```

Map coloring

Map coloring is a very easy problem, there are many theorems on this problem. Suppose we need to color portions of a map, characterized by a number, in such a way that two contiguous regions are colored with different colors. Suppose we also have the colors red (r), green (g) and blue (b).

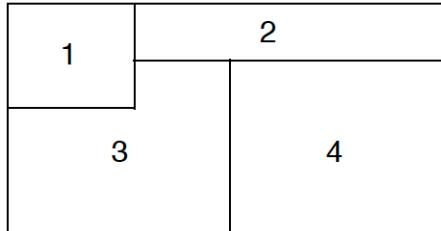


Figure 8.3

Referring to the figure above we need to take 4 decisions, and every decision has 3 possible values that are colours. This is really a toy problem (simple). Just to show how to model such a problem. But if you move to the *Graph coloring problem* where you have to color every node such that all adjacent nodes are colored with different colours (then of course the number of colours needed can be any depending on the clustering, the connectivity of the graph), this problem is a very famous problem from *combinatorial optimization* and it is very difficult to be solved. There are many algorithms for solving this problem.

⁸Like universitary lectures for instance.

⁹For example one machine works on a specific piece, on a specific product (unary) while other machines can work on three products simultaneously (their capacity is 3).

¹⁰Of course if you have a very long time horizon scheduling activity, e.g. 1 month, and the granularity of 15 minutes, the domains becomes quite big since you have to put a lot of start times.

¹¹For example, before drying a product you've to colour it.

¹²Below d_1 , d_2 indicates the duration.

Map coloring as CSP

Four colors are sufficient for each map (proved in 1976). Easy graph coloring problem variant where at most four regions are all related.

Schema:

- *variables*: regions;
- *domains*: colors;¹³
- *constraints*: adjacent regions must have different colors.

Criptoarithmetics

$$\begin{array}{r}
 \begin{array}{ccccc} S & E & N & D & + \\ M & O & R & E & = \\ \hline M & O & N & E & Y \end{array}
 \end{array}$$

Figure 8.4

We have the:

- *variables*: letters;
- *domains*: digits from 0 to 9;
- *constraints*: only one constraint, the showed sum ($D + E = Y$ or $D + E = 10 + Y$ etc.).

Objective: to determine a state in which each letter is a digit so that the initial constraints are met.

The numbers sum as illustrated by the problem.

Sudoku as a CSP

	9			7				
4	5	9	1					
3		1			2			
1		6		7				
	2	7	1	8				
5		4		3				
7		3		4				
8	2	4		6				
6		5						

Figure 8.5

Sudoku as a CSP:

- some boxes are already fixed, the other should be filled with numbers from 1 to 9;

¹³Domains means: which values can I assign to these variables?

- the table is divided into 9 quadrants of 3×3 boxes;
- each quadrant should contain all the numbers, with no repetitions;
- in addition, each horizontal row and each vertical line of the entire board must not contain repetitions;
- each box has a variable domain with from 1 to 9;
- you will see special constraints (`AllDifferent`) during other courses.

CSP more formally

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

- (x_1, x_2, \dots, x_n) decisions that we have to take;
- (D_1, D_2, \dots, D_n) domains of possible values;¹⁴
- a set of constraints.

A constraint $c(x_{i1}, x_{i2}, \dots, x_{ik})$ between k variables is a subset of the cartesian product $D_{i1} \times D_{i2} \times \dots \times D_{ik}$ that specifies which values of the variables are compatible with each other.

This subset must not be explicitly defined but is represented in terms of relationships.

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

*CSPs can be solved through search.*¹⁵

- *Initial state:* empty assignment [] .
- *Successor Function:*¹⁶ assigns a value to a variable not yet assigned; `fail` if there is none.
- *Goal:* the assignment is complete (all variables are assigned to a feasible value).

Main points:

1. same scheme for all CSPs;
2. limited depth n if n is the number of variables; uses *depth-first search*;¹⁷
3. the path is irrelevant;
4. problem with d^n leaves (if d is the cardinality of domains).¹⁸

¹⁴Each domain should have the discrete cardinality, finite (we are in the combinatorial world).

¹⁵In general can be solved through any kind of *search*, we will see only the case of *tree search*.

¹⁶*Successor Function* is the function which brings you from one state to the next one.

¹⁷ n represents the maximum depth. In general it is used *depth-first search*; but we can also use other searches.

¹⁸Which makes the problem an exponential problem.

Search tree

A possible search tree for a CSP is obtained after establishing an order for variables:¹⁹ each level of the tree corresponds to a variable and each node corresponds to a possible value assignment.

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

The search for a solution is equivalent to the exploration of the tree to find a leaf-solution.

In an n -variable problem in which all the domains have the same cardinality d , the number of leaves of the search tree is equal to d^n .

Example

For example, in a tree that is a problem of 10 variables and where each domain has cardinality 10 there are 10 billion leaves.

It is therefore evident that a smart tree exploration strategy is essential to find a solution to a complex problem in a reasonably short time (consistency techniques). Here 3^2 leaves.

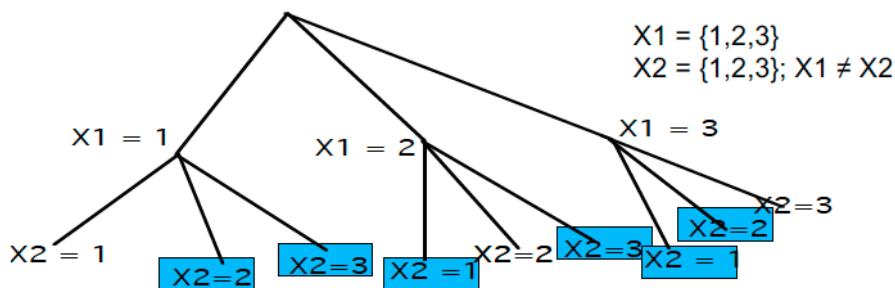


Figure 8.6

Looking at the figure above we have *one constraint* $X1 \neq X2$, two levels of expansions (as the number of variables) and in the second level we have two failures (since occurs that $X1 = X2$); finally all the blue rectangles represent solutions.

Propagation algorithms

So it is reasonable to understand that it is no efficient to explore the entire search tree finding all possible leaves and then filtering the good leaves from the bad ones; this technique – called *generate and test* – is not efficient with exponential problems. So, we can exploit *propagation algorithms* or *consistency techniques* that help to discover *searc trees* in a smarter way.

The propagation algorithms are smart search methods that exploit constraints to prevent failures rather than recover from failures have already occurred.

A priori pruning of the search tree.

¹⁹This would be an *heuristic* we will see.

Use constraints between the variables of the problem to reduce the search space before reaching the failure.

This eliminates subtrees that lead to a failure thus limiting unnecessary backtracking.

Two approaches

Given a CSP there are two possible approaches to its solution: one based on Consistency techniques and the other on Propagation algorithms.

Without loss of generality, we will refer, hereinafter, to CSP involving binary constraints (*i.e.* constraints which involve two variables).

- *Propagation Algorithms:*

- based on the propagation of constraints to eliminate a priori, *while searching*, portions of the search tree which would lead to a failure.

- *Consistency Techniques:*

- based on the propagation of constraints in order to derive a simpler problem than *original*.

A posteriori algorithms

The two techniques that use a posteriori constraints are:²⁰

- the Generate and Test (GT);
- the Standard Backtracking (SB).

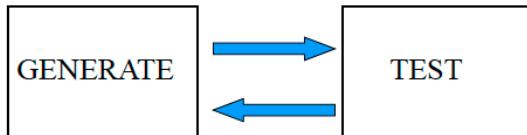


Figure 8.7

Propagation algorithms

The propagation algorithms are based on the inverse concept.

Techniques such *Forward Checking* (FC), and *Looking Ahead* (LA).²¹

A module propagates the constraints as much as possible (*constrain*); at the end of propagation either we have reached solution, or a failure or new information about the free variables is required (*generated*).

²⁰Up to now we've seen and used the *Generate and Test* technique: generate the all solutions and then check the leaves. A slightly better version of the previous *blind* method is the *Standard Backtracking*: you perform an assignment and as soon you perform the assignment you ask yourselves if the performed assignment is feasible, if yes you continue, if not you backtrack. These methods are called a *posteriori* methods because they use the constraints after that the assignment has been done, they test after the consistencies.

²¹We will see both *partial* and *full* versions. These techniques basically propagate the constraints

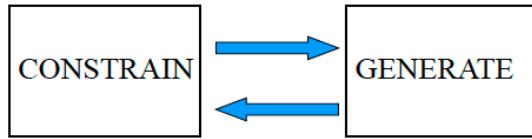


Figure 8.8

Tree Search

Consider a *depth-first search*.²² It assigns a variable at a time. At each step we either:²³

- find a solution;²⁴
- discover a failure;²⁵
- assign another variable.

The algorithm has three degrees of freedom:

- the choice of the variable ordering;
- the choice of the ordering of values to be assigned to the current variable;²⁶
- the propagation carried out in each node.²⁷

The first two relate to search heuristics.

The third degree of freedom is what differentiates the different strategies.

- No propagation algorithms:

- *Generate and Test*;
- *Backtracking Standard*.

- Propagation Algorithms

- *Forward Checking*;
- *(Partial and Full) Look Ahead*.

constraining the problem and then generate a solution. So they basically remove values that are not feasible and then regenerate a solution in the remaining part of the domain. And you do that during search.

²²Remember that depth-first explore always nodes at an higher depth; it could be a search which is not complete but in this case we're in a combinatorial world and therefore we have a finite number of solutions so we cannot have cycles, loops or whatever so *depth-first* can be complete since there are no loops, so it is complete.

²³Three possibilities.

²⁴You're in a solution node: you've assigned every variable and you're happy with your constraints.

²⁵You cannot go further, so you fail.

²⁶E.g. in the eight-queen problem when I decide how to place queens I can decide to go through lexicographic order (column 1 then columns 2 then column 3 and so on) but I can choose another order that starts with column 5.

²⁷We will see that we've three different algorithms for propagation of the constraints. In particular we also decide how much time you want to spend in every node to remove as much as possible not feasible values.

Generate and Test

The language interpreter develops and visits a decision tree covering it in depth by assigning values to variables without bothering to verify the consistency with any constraint.

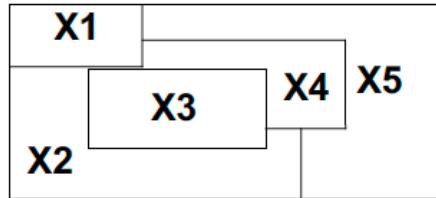


Figure 8.9

For example, for this instance of Map Coloring problem we have:

- *domain constraints*: $X_1, X_2, X_3, X_4, X_5 :: \{red, yellow, green, blue\}$;
- *topological constraints*: $X_1 \neq X_2, X_1 \neq X_3, X_1 \neq X_4, X_1 \neq X_5, X_2 \neq X_3, X_2 \neq X_4, X_2 \neq X_5, X_3 \neq X_4, X_4 \neq X_5$;
- we try all permutations of colors and then check if the constraints are satisfied;
- the search tree has 4^5 leaves.

Generate and Test for the 8 queens

Consider the problem of the eight queens: the variables involved in the problem require as a domain of definition, the integers between 1 and 8.

The Generate and Test assignes to the variables a permutation of the integers values in the domain.

The only constraints considered in Generate step are:

- $1 \leq X_i \leq 8$ for $1 \leq i \leq 8$;
- $X_i \neq X_j$ for $1 \leq i < j \leq 8$.

The second is due to the fact that any attempt consists of a permutation of the values belonging to the domains, and then each value assigned to the variables is different from others.

Generate and Test inefficiencies

Basic inefficiencies.

- The constraints are used to limit the space of solutions after the search is performed, therefore in an *a-posteriori* fashion.
- The number of possible permutations increases with the factorial of the number of terms to permute. If $n = 8$ we have a number of permutations equal to $8! = 40320$, if $n = 10$ then $n! = 3.6288$ million coming for $n = 20$ to orders of magnitude of 10^{18} and then the size is unacceptable for a search space.

Standard Backtracking

Although this technique is better than the previous one, it still uses constraints in an *a-posteriori* fashion:

- at each instantiation of a variable x_i , constraints involving x_i and previously instantiated variables are checked;
- therefore, the use of constraints is more effective than for *Generate and Test* as it does not continue searching in the branches that present contradictions.

A simple example of Standard Backtracking

Let's see a basic example, a simple algorithm.
First we place, then check; place first, check then.

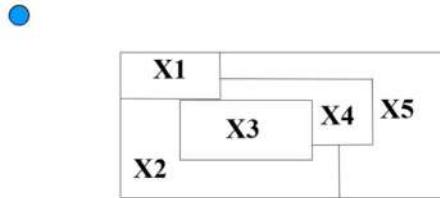


Figure 8.10

Place red in $X1$ first; then ask yourselves: it is consistent with the constraints?
Yes, so go on...

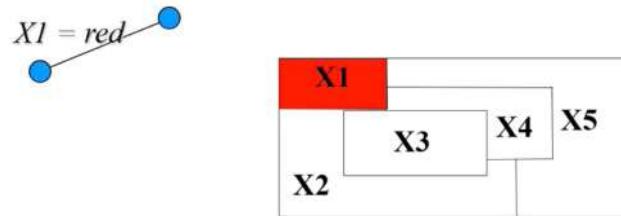


Figure 8.11

Place $X2 = \text{red}$ first; then ask yourselves: it is consistent with the constraints?

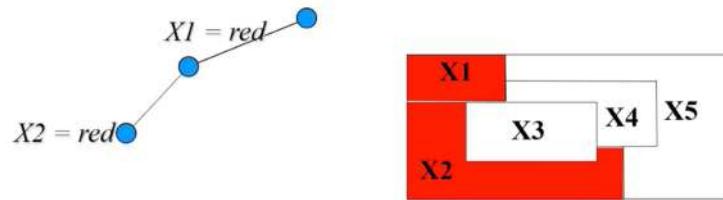


Figure 8.12

No!

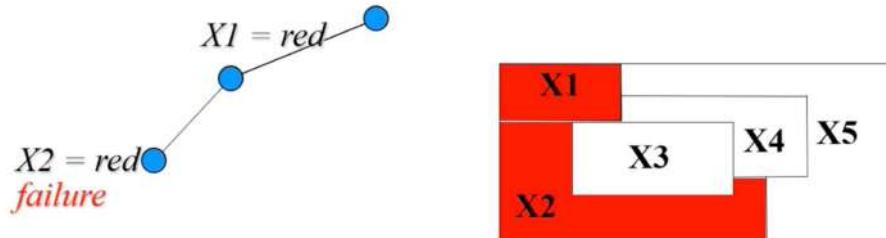


Figure 8.13

Place $X2 = \text{green}$; then ask yourselves: it is consistent with the constraints?
Yes, so go on.

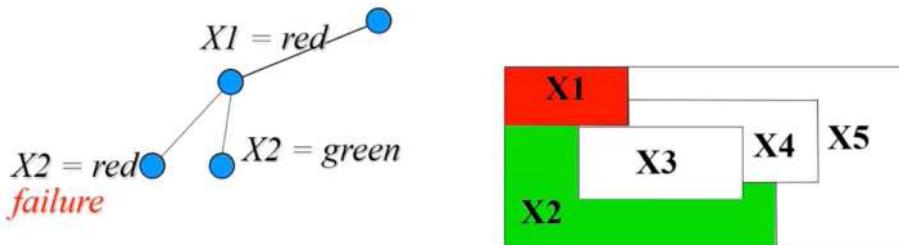


Figure 8.14

And so on...

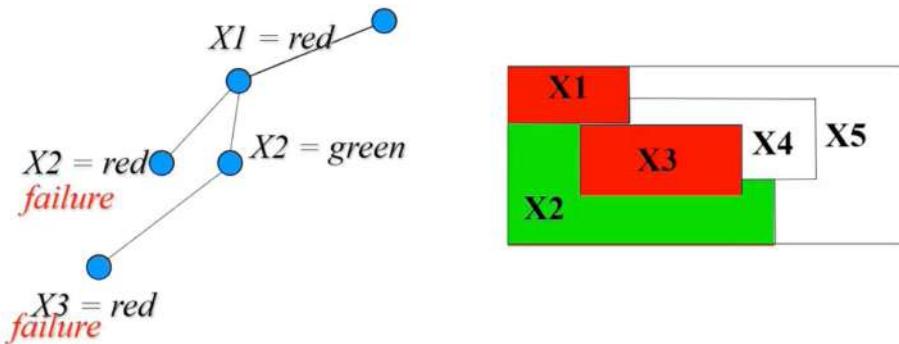


Figure 8.15

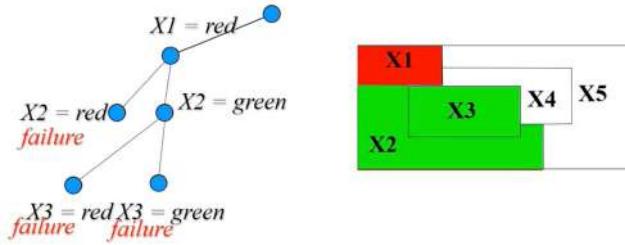


Figure 8.16

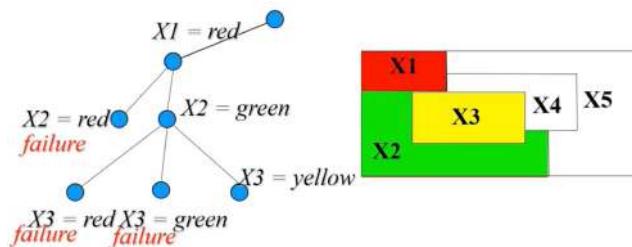


Figure 8.17

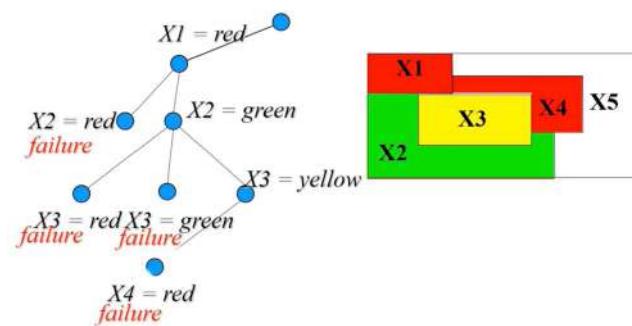


Figure 8.18

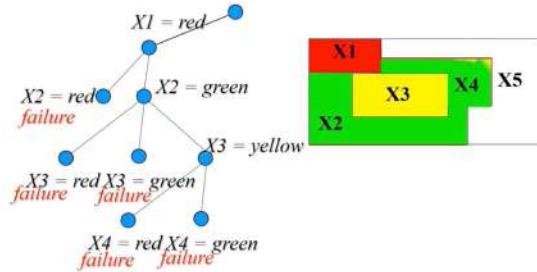


Figure 8.19

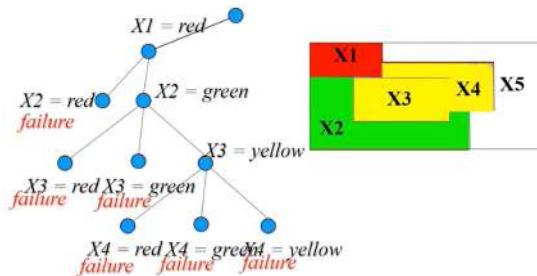


Figure 8.20

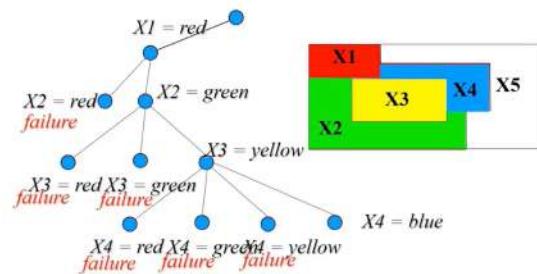


Figure 8.21

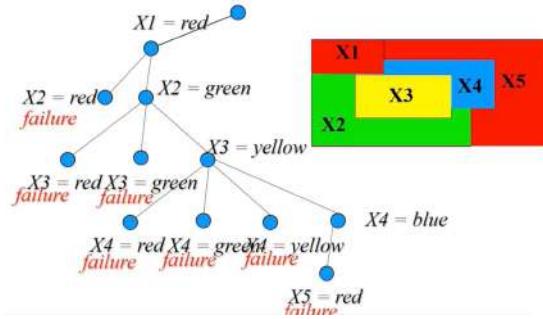


Figure 8.22

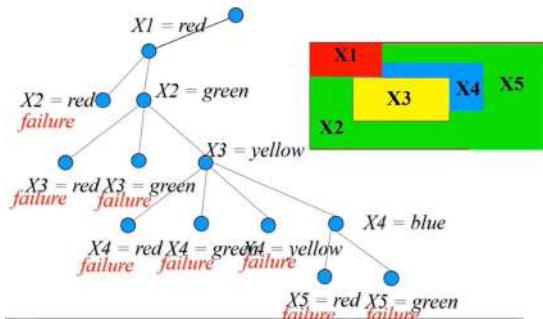


Figure 8.23

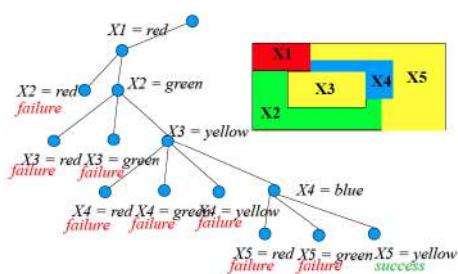


Figure 8.24

So even if is a small problem we had 8 failures!

8 queens Standard Backtracking

For the eight queens, the attempt made by *Generate and Test* to solve assigning all queens to a diagonal of the board would be blocked at the second instantiation. In fact, the constraint $x_i \neq x_j - (j - 1)$ for $1 \leq i < j \leq 8$ would be violated by the first two assignments replacing $x_1 = 1$ and $x_2 = 2$ in the above constraint would lead to $1 \neq 2 - (2 - 1)$ which leads to a contradiction.

The algorithm is then stopped and, with a backtracking, you try to assign to x_2 the value 3 with success and so on.

Standard Backtracking pseudo-code

Below we propose the *pseudo-code* of *Standard Backtracking*; do not learn in detail, just understand the concept!

```

function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
            if result  $\neq$  failure then return result
            remove { var = value } from assignment
    return failure

```

Figure 8.25

The constraints are used backward (*backward*) and lead to an effective reduction of the search space w.r.t. the one produced by *generate and test*.

However this reduction is done backward after the assignment.

Unlike the *Generate and Test* method, which leaves the constraint checking activity at the end of the instantiation of all the variables, *Standard Backtracking* checks their consistency at each instantiation.

Example: the eight-queens

In the solution of the 8 queens problem suppose we have already assigned six variables to values:

- $(X_1, X_2, X_3, X_4, X_5, X_6) = (1, 3, 5, 7, 2, 4)$.

The assignment $X_1 = 1$ is the first choice made.

The assignment $X_2 = 1$ would violate the constraint on the column.

The assignment $X_2 = 2$ would violate the constraint on the diagonal: $V_{k+1} \neq V_i - (k + 1 - i)$ to $1 \leq i \leq k$.

So the assignment $X_2 = 3$ is performed that successfully satisfies all the constraints:

- $3 \neq 1;$
- $1 \leq 3 \leq 8;$
- $3 \neq 1 + (2 - 1);$
- $3 \neq 1 - (2 - 1).$

For the third variable the value 1 violates the constraint on the row, the value 2 would violate the constraint on the diagonal with the second variable, in fact: $2 \neq 3 - (3 - 2)$ is not satisfied and value 3 violates the constraint on the row with the second variable.

Then we proceed with $X_3 = 4$ which violates the constraint on the diagonal with X_2 .

$X_3 = 5$ is compatible with all constraints:

- $5 \neq 1; 5 \neq 3; 1 \leq 5 \leq 8;$
- $5 \neq 1 + (3 - 1); 5 \neq 3 + (3 - 2);$
- $5 \neq 1 - (3 - 1); 5 \neq 3 - (3 - 2).$

We proceed to the instantiation of the fourth variable, and all its assignments to values 1, 2, 3, 4, 5, 6 violate all constraints.

$X_4 = 7$ is compatible with all constraints:

- $7 \neq 1; 7 \neq 3; 7 \neq 5; 1 \leq 7 \leq 8;$
- $7 \neq 1 + (4 - 1); 7 \neq 3 + (4 - 2); 7 \neq 5 + (4 - 3);$
- $7 \neq 1 - (4 - 1); 7 \neq 3 - (4 - 2); 7 \neq 5 - (4 - 3).$

In the same way we end up assigning $X_5 = 2$ and $X_6 = 4$.

For the last column, corresponding to the variable X_8 all assignments fail.

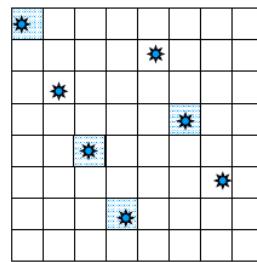


Figure 8.26

Therefore the algorithm should backtrack.

Limitations of Standard Backtracking

A pitfall of SB is that it uses constraints *a-posteriori*, i.e. after the instantiation is done.

Exploiting constraints involving free variables (still to be assigned variables) would detect this situation in advance thus avoiding expensive backtracking.

The idea behind a priori *propagation algorithms* is an active use of the constraints during search. They *prune* a priori the search tree. Each variable is associated with the set of values that are still consistent after each assignment.

These values are stored in *domains* that are reduced during search by removing those values that would lead to a failure.²⁸

Propagation algorithms

Essentially we have three algorithms:

- *Forward Checking (FC)*;
- *Partial Look Ahead (PLA)*;
- *Full Look Ahead (FLA)*.

They perform increasing amount of checks on free variables.

Forward Checking

Forward Checking is the most intuitive among the propagation algorithms.

- After each assignment of variable X_i ,²⁹ the forward checking propagates all constraints involving X_i considering also all variables that are not yet instantiated.
- This method is very effective especially when free variable domains are associated with a reduced set of allowable values and therefore are easily assignable.
 - If the domain associated to a free variable has only one value left in the domain it can be performed without any computational effort.
 - If a domain becomes empty the Forward Checking algorithm fails and backtracks.

²⁸The state of the search tree at some point contains the variables and the domains of the variables containing those values that are compatible with the values that have been previously selected. Think about the 8 queens problem. Initially the domain of a queen that we decide to start contains all possible values. After the placement of the first queen on the first row, all other queens see the value 1 removed from their domains, since the first row is not available for any other queens accordingly with the constraints; in this way constraints are active, but you have to remember that the state of the search tree has to contain the variables but also the domains that contain those values that are left, compatible after the propagation.

²⁹So the thing that you do is: you assign the variable X_i , then you say: in which constraints this X_i is involved? I will use these constraints to remove from the domains of the other variables values that are inconsistent with the assignments that you've done. So as soon as you've assigned one queen on the row 1 you have to remove from other queens (variables) the same row which is 1, but also all values of the diagonals. So you have to keep these domains always update during your search. When you backtrack you reinsert in the domains the values that you've pruned by your last decision.

- It is based on the observation that the assignment of a value to a variable has impact on all the available values for the free variables. In this way, the constraints act forward and reduce the space of the solutions before exploring it.

Forward Checking: a simple example

Let's see an example.



Figure 8.27

Suppose we have already assigned X_1 and X_2 as shown in the figure above (so we have 4 variables).

- The Standard Backtracking checks $c(X_2, X_1)$.³⁰
- The Forward Checking checks $c(X_2, X_3)$ for each value in D_3 and $c(X_2, X_4)$ for each value in D_4 ; the values of D_3 and D_4 that are incompatible with the current assignment of X_2 are deleted.³¹
- If the domain of a free variable becomes empty we have a *failure*.³²

Forward Checking: example

Elimination of prior inconsistent values from variable domains of the future.

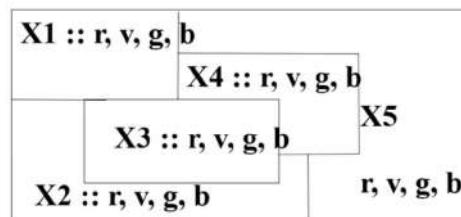


Figure 8.28

I assign X_1 with red and immediately I remove red from all adjacent zones (so remove the colour used for X_1 , see the red semi-cross).

³⁰Generate and Test cannot do anything with only 2 assignments.

³¹Shouldn't we do the same thing for X_1 ? Depends on the problem. It will be clearer ahead.
Let's see an example in the next section.

³²So we can fail before having completed the leaf.

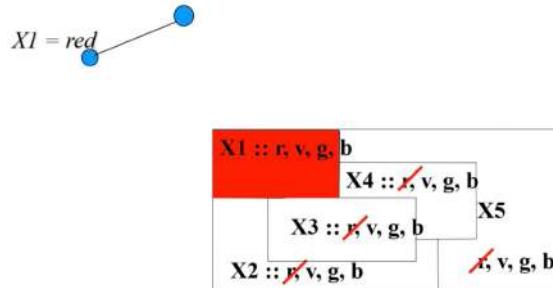


Figure 8.29

Now I proceed with assign X_2 with the following feasible value which is green (it is indicated in the figure with the letter 'v' which stands for the italian word 'verde' indicating the colour green) and immediately remove it from adjacent zones.

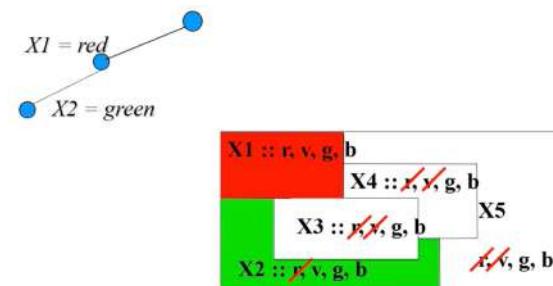


Figure 8.30

Then X_3 with yellow and remove yellow from X_4 (notice that in X_4 the only remaining value after this assignment is blue).

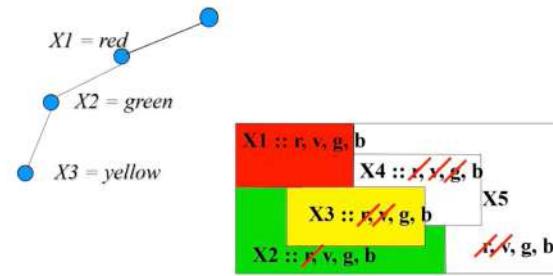


Figure 8.31

So I colour X_4 with blue and I remove it from X_5 which remains with only one value which is yellow.

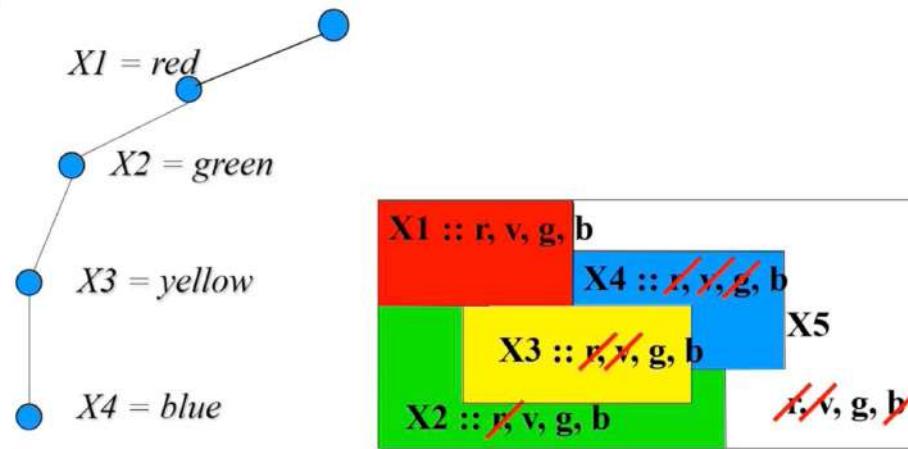


Figure 8.32

So I colour X_5 with yellow!

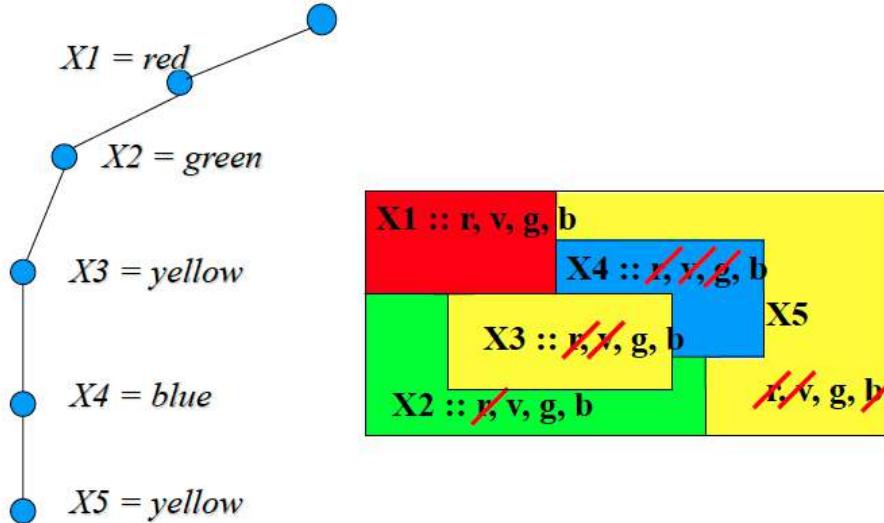


Figure 8.33

What do you notice in comparison to the same example resolved with *Standard Backtracking*? We have drastically reduced the search space, failures and backtracks (failures and backtracks are zeros) and so also *time*!

Forward Checking: 8 Queens (cont.)

We assign $X_1 = 1$. The FC removes value 1 from the domain of all variables and the value i from the domain of each X_i :³³

³³Basically as the figure below shows we remove the row, the column and the diagonal. Then what we do? We assign X_2 with the next available value which is 3.

- X_2 is associated with the domain $D_2 = (3, 4, 5, 6, 7, 8)$;
- X_3 is associated with the domain $D_3 = (2, 4, 5, 6, 7, 8)$;
- X_4 is associated with the domain $D_4 = (2, 3, 5, 6, 7, 8)$;
- X_5 is associated with the domain $D_5 = (2, 3, 4, 6, 7, 8)$;
- X_6 is associated with the domain $D_6 = (2, 3, 4, 5, 7, 8)$;
- X_7 is associated with the domain $D_7 = (2, 3, 4, 5, 6, 8)$;
- X_8 is associated with the domain $D_8 = (2, 3, 4, 5, 6, 7)$.

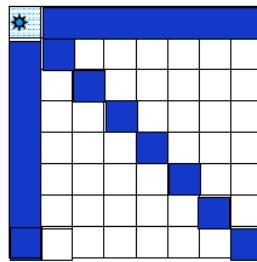


Figure 8.34

We then assign $X_2 = 3$. The domains of free variables become:³⁴

- X_3 is associated with the domain $D_3 = (5, 6, 7, 8)$;
- X_4 is associated with the domain $D_4 = (2, 6, 7, 8)$;
- X_5 is associated with the domain $D_5 = (2, 4, 7, 8)$;
- X_6 is associated with the domain $D_6 = (2, 4, 5, 8)$;
- X_7 is associated with the domain $D_7 = (2, 4, 5, 6)$;
- X_8 is associated with the domain $D_8 = (2, 4, 5, 6, 7)$.

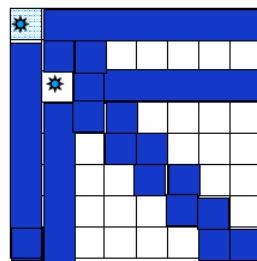


Figure 8.35

We assign value 5 to X_3 and propagate the constraints obtaining:

³⁴You remove the second column, the third row, and the 2 associated diagonals. Propagate the deletions on the other domains.

- X_4 is associated with the domain $D_4 = (2, 7, 8)$;
- X_5 is associated with the domain $D_5 = (2, 4, 8)$;
- X_6 is associated with the domain $D_6 = (4)$ ³⁵
- X_7 is associated with the domain $D_7 = (2, 4, 6)$;
- X_8 is associated with the domain $D_8 = (2, 4, 6, 7)$.

As soon as X_4 is assigned to value 2 we obtain a failure as the domain of X_6 becomes empty:

- X_5 is associated with the domain $D_5 = (4, 8)$;
- X_6 has domain D_6 empty;
- X_7 is associated with the domain $D_7 = (4, 6)$;
- X_8 is associated with the domain $D_8 = (4, 7)$.

The backtracking leads to the assignment $X_4 = 7$.

The algorithm narrows domains as follows:

- X_5 is associated with the domain $D_5 = (2, 4)$;
- X_6 is associated with the domain $D_6 = (4)$;
- X_7 is associated with the domain $D_7 = (2, 6)$;
- X_8 is associated with the domain $D_8 = (2, 4, 6)$.

Then $X_5 = 2$ leads to:

- X_6 is associated with the domain $D_6 = (4)$ ³⁶
- X_7 is associated with the domain $D_7 = (6)$;
- X_8 is associated with the domain $D_8 = (4, 6)$.

When assigning the two singleton variables we obtain a failure and backtrack again.

³⁵Note that the domain of variable X_6 contains only one value.

³⁶At this point we have that D_6 remains with only one value; theoretically this is already an assignment for variable X_6 but *Forward Checking* does not consider it as an assignment, it is not aware of that; so proceed with its instantiation. But we already know at this point that X_6 is already assigned and that we could already remove the value 4 from the other remaining domains (and that this lead to a failure). So we could understand the failure before *Forward Checking*. We can do more than what *Forward Checking* does. Note that if we reason on top of unassigned variables we can already infer something. If you just check the domains related to unassigned variables it is possible to do more than what *Forward Checking* does.

Look Ahead

The *Look Ahead* algorithms are algorithms that do more: they *reason* on top of those variables that are still free, for which a decision has not already been taken. While *Forward Checking* checks constraints between the variables that I've just instantiated and the future variables the *Look Ahead* does the *Forward Checking* steps plus something more: check between the variables that are still free, something more can be pruned. I can do this more in two ways: a *partial* or *full* way.

Beside checking the constraints with the current instantiated variable, look ahead also checks the non assigned variables.

Look Ahead checks the existence, in the domains associated with the non assigned variables, of values that are compatible with the constraints containing only non-instantiated variables.

It is verified the possibility of a future consistent assignment between (pairs of) free variables.

Partial Look Ahead (PLA) and *Full Look Ahead (FLA)*.³⁷

Suppose we have assigned variables from X_1 to X_{h-1} .³⁸

- *PLA*: constraint propagation containing the not yet assigned variable X_h with not yet instantiated ‘future’ variables, i.e. variables X_{h+1}, \dots, X_n ;
 - for each value in the domain of X_h it checks if in the domain of not yet assigned variables X_{h+1}, \dots, X_n there is a value compatible with it.³⁹
- *FLA*: constraint propagation containing the not yet assigned variable X_h with not yet instantiated variables $X_{k+1}, \dots, X_{h-1}, X_{h+1}, \dots, X_n$:⁴⁰
 - for each value in the domain of X_h it checks if in the domain of not yet assigned variable $X_{k+1}, \dots, X_{h-1}, X_{h+1}, \dots, X_n$ there is a value compatible with it.

A very basic example

A very basic example. Suppose to have the free variables (along with their domain) $X_1 :: [1, 2, 3]$, $X_2 :: [1, 2, 3]$ and the constraint is $X_1 < X_2$ and we have assigned X_0 .⁴¹

What do you think you can remove from the domains? Which is the propagation?

- Consider *PLA*:⁴² $\forall x_1 \in [1, 2, 3] \exists x_2 \in [1, 2, 3] : x_1 < x_2$? Yes, for 1, 2 but not for 3 so \Rightarrow remove it from $X_1 :: [1, 2, 3] \Rightarrow X_1 :: [1, 2] \therefore$ the propagation is $X_1 :: [1, 2], X_2 :: [1, 2, 3]$.

³⁷Which is the high level difference between the twos? Basically the PLA consider constraints in only one direction.

³⁸Consider: *Forward Checking* is already done, as said is always part of both PLA and FLA. Then consider to be in the middle of our search tree. I have not to consider anymore X_1 to X_{h-1} because with them I'm done, I've already propagated the constraints involving them. Now I'm at the point to assign the variable X_h (current variable).

³⁹For each value in the domain of X_h we basically check a *support* in the domain of the not yet assigned variables that satisfies the constraints; if there is such a support I will leave the accounted value in the domain of X_h , otherwise I remove it.

⁴⁰Perform the checks in the two directions: variable before h and after h .

⁴¹Note the symbol $::$ represents the specific notation for the domain.

⁴²So we take only the point of view of X_1 and compare it with the ‘future’ X_2 .

- Consider *FLA*:⁴³ first perform what done before; then also: $\forall x_2 \in [1, 2, 3]$ $\exists x_1 \in [1, 2] : x_1 < x_2$? Yes for 2, 3 but not for 1 so \Rightarrow remove it from $X_2 :: [1, 2, 3] \Rightarrow X_2 :: [2, 3] \therefore$ the propagation is $X_1 :: [1, 2], X_2 :: [2, 3]$.

FC vs PLA for Map Coloring

PLA does not increase more than FC the pruning of the tree in this case.⁴⁴

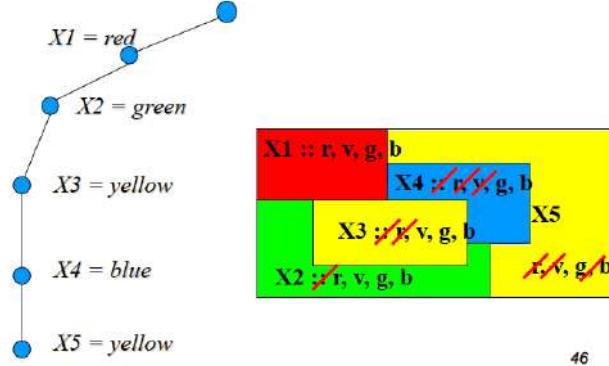


Figure 8.36

Consider the case we have only three domain values (r, v, g), the FC performs the following search:⁴⁵

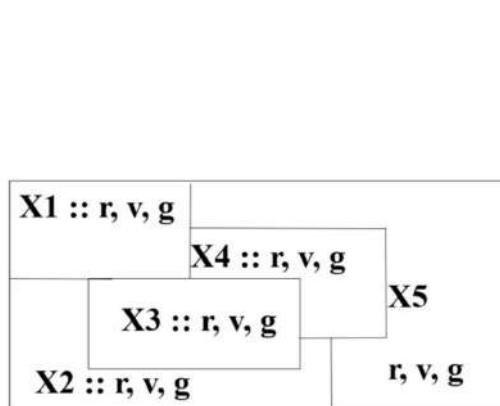


Figure 8.37

⁴³We also take the point of view of X_2 and compare it with the no yet instantiated X_1 . Remember that X_1 and X_2 are still free variables. We are not considering a variable that is instantiated.

⁴⁴When we have 4 colours. Remember that we said that PLA and FLA does *Forward Checking plus something*. So since we have seen that for this specific problem FC has no failures they are basically the same.

⁴⁵With only three colors we anticipate there is no solution. Anyway if you try before with FC and then with PLA we will see that PLA can do something better than FC. PLA can anticipate the failure. One typical exercise of the exam is just like this: given a specific problem it is asked to show the behaviour of both FC and PLA.

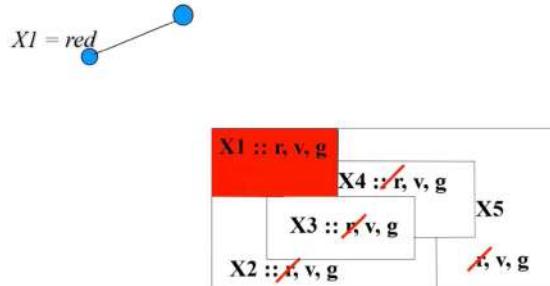


Figure 8.38

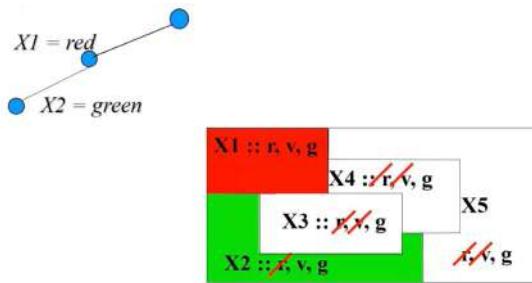


Figure 8.39

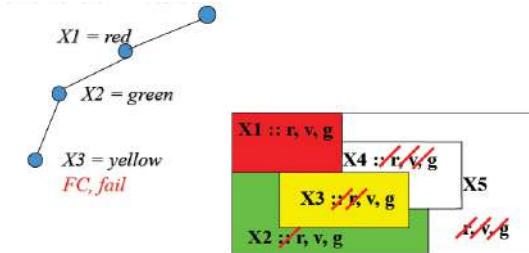


Figure 8.40

The domain of $X5$ becomes empty \implies fail! What we're seeing is a very little problem but you have to imagine that real problems can be very big! So saving one step here, anticipating the failures means saving a great number of steps and backtracking in real life problems.

The PLA anticipates the failure (of one level in this case).

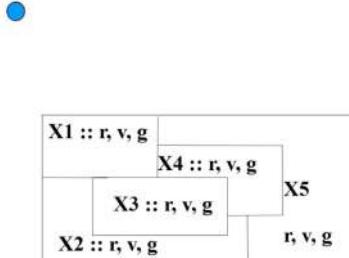


Figure 8.41

As soon as you assign the red you simply remove it from other domain variables.

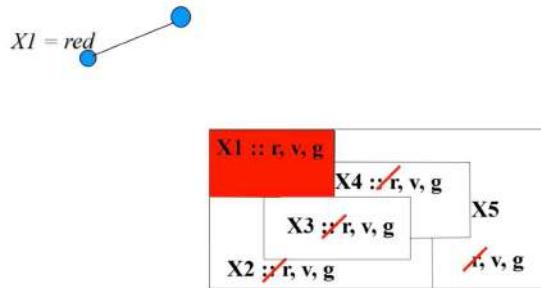


Figure 8.42

As soon as you assign $X2$ with green, you remove it from other domain variables, furthermore the $X3$ domain becomes empty because there is no value in the domain of $X4$ compatible with the g value for $X3$.

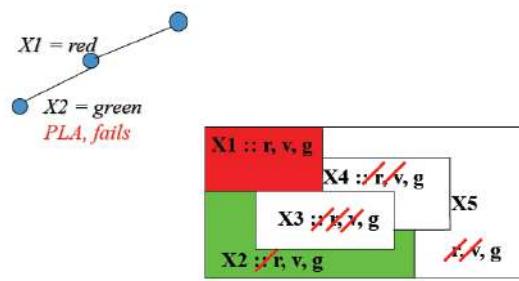


Figure 8.43

Look Ahead: example

Consider the example so structured: the assigned variable is $X0 = 0$; the free variables $X1, X2, X3$; the constraints $X0 < X1 < X2 < X3$ with domains for $X1, X2, X3 :: [1, 2, 3]$.



Figure 8.44

PLA, verification:

- for each value in D_1 checks if there exists at least one value in D_2 and at least one value in D_3 compatible (in case these values do not exist we delete the value from the domain D_1);
- for each value in D_2 checks if there exists at least one value in D_3 compatible (in case this value does not exist we delete the value from the domain D_2);
- propagation is: $X_1 :: [1, 2]$; $X_2 :: [1, 2]$; $X_3 :: [1, 2, 3]$.

FLA, beside the checks performed by PLA checks also:

- for each value in D_2 checks if there is at least a value compatible in D_1 , for each value in D_2 checks if there exists at least one value in D_3 compatible (in case this value does not exist we delete the value from the domain D_2);
- for each value in D_3 checks if there exists at least one value compatible in D_2 and at least one value in D_1 compatible (in case these values do not exist, we delete the value from the domain of D_3);
- propagation is: $X_1 :: [1, 2]$; $X_2 :: [2]$; $X_3 :: [3]$.

Note: one single iteration; we could do more!

PLA: the eight queens

Suppose you have assigned to the first three variables X_1, X_2, X_3 respectively the values 1, 3, 5.

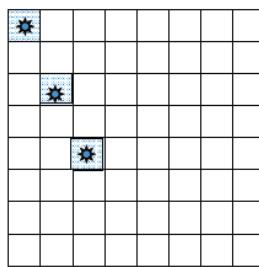


Figure 8.45

Domains remain the same of the FC:

- X_4 is associated with the domain $D_4 = (2, 7, 8)$;
- X_5 is associated with the domain $D_5 = (2, 4, 8)$;
- X_6 is associated with the domain $D_6 = (4)$;

- X_7 is associated with the domain $D_7 = (2, 4, 6)$;
- X_8 is associated with the domain $D_8 = (2, 4, 6, 7)$.

We consider for D_4 the value 2. The values 4 in D_5 , 6 in D_7 and 4 in D_8 are compatible with the value of 2 in D_4 . However, the value 4 in D_6 , is not compatible.⁴⁶ Value 2 is deleted from D_4 .

For values 7 and 8 instead we find a support in the domain of other variables. The domain associated with X_4 thus becomes: $D_4 = (7, 8)$.

Likewise, we can delete value 4 from D_5 as it does not have support in D_6 .

Value 8 in D_5 has support in the domain of future variables. So the domain associated with X_5 becomes: $D_5 = (2, 8)$.

The sets D_6 and D_7 are not modified.

Note that we have removed the value 2 from the domain X_4 that was instead assigned by FC leading for a failure.

FLA: the eight queens

Take the earlier example, suppose you have already propagated domains with the PLA technique. Now we *apply FLA*:

- X_4 is associated with the domain $D_4 = (7, 8)$;
- X_5 is associated with the domain $D_5 = (2, 8)$;
- X_6 is associated with the domain $D_6 = (4)$;
- X_7 is associated with the domain $D_7 = (2, 4, 6)$;
- X_8 is associated with the domain $D_8 = (2, 4, 6, 7)$.

Note that for the value 8 in D_5 , there is no value belonging to D_4 that satisfies the constraints imposed by the problem.

The PLA is not ‘aware’ of this inconsistency because it verifies the consistency of the values belonging to a domain D_i with the values belonging to the domains D_j only if $j > i$.

In the domain D_5 , we have only the value 2. The value 4 in D_6 is compatible with 2 in D_5 and with the values 7 and 8 in D_4 .

- X_4 is associated with the domain $D_4 = (7, 8)$;
- X_5 is associated with the domain $D_5 = (2)$;
- X_6 is associated with the domain $D_6 = (4)$.

Now consider the domain D_7 : the value 2 in D_7 is not compatible with 2 in D_5 and is eliminated. The same thing happens for 4 in D_7 incompatible with 2 in D_5 .

D_7 is therefore: $D_7 = (6)$.

The domain D_8 , already at this point in the computation, does not contain more values compatible with those of the previous domains. In fact the value 2 in D_8 is incompatible with 2 in D_5 , 4 in D_8 is incompatible with 4 in D_6 , 6 and 7 in

⁴⁶These values would stay on the same diagonal.

D_8 are incompatible with 6 in D_7 . The computation, therefore, fails without performing additional assignments.

The computational load due to additional verification of the consistency of constraints, needs to be balanced by a reduced search time. In the first levels of the tree sometimes FC is more effective.⁴⁷

Tree search: to recap

To recap: all constraint satisfaction problem are solved through tree search (at least for what concerns this course).

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

- find a solution;
- discover a failure;
- assign another variable.

The algorithm has three degrees of freedom:

- the choice in the ordering of variables;
- the choice in the ordering of values to be assigned to the current variable;
- the propagation carried out in each node.

The first two relate to search heuristics.

The third degree of freedom is what differentiates the different strategies.

- No propagation algorithms:
 - Generate and Test;
 - Backtracking Standard.
- Propagation Algorithms:
 - Forward Checking;
 - (Partial and Full) Look Ahead.

Search heuristics

While constraints propagation is performed by the solver, the order of variable selection and value selection are available to the programmer.

The heuristics can then act on these two degrees of freedom to try to ensure the achievement of a good solution in a reasonable time for even the most complex problems.

The heuristics can be classified into two types:⁴⁸

⁴⁷That's why in some situations, also if PLA and FLA seem to be in general more efficient of FC, FC is still used. Because otherwise one could ask: why do we take still into account FC (the less checks I do, faster I go!) if we have PLA and FLA? You always have to find the balance between *propagation* (PLA, FLA) and *search* (FC), the more you propagate \Rightarrow less you search; the less you propagate \Rightarrow the more you have to search.

⁴⁸In general the value selection heuristics is not so widely used while the variable selection heuristics are always used.

- *Variable Selection Heuristics:*

- determine what should be the next variable to instantiate. The two most commonly used heuristics are the *first-fail* (a. k. a. *MRV*: Minimum remaining Values) that chooses the variable with the smallest domain,⁴⁹ and *most-constrained principle* who chooses the variable appearing in the largest number of constraints. *Rationale: most difficult variables are assigned first.*

- *Value Selection Heuristics:*

- determine what value to assign to the selected variable. There are no general rules here but the rationale is try first those values that are most likely to succeed (*least constraining principle*).⁵⁰

Classification of heuristics

A further classification is as follows.

- *Static heuristics:*

- determine variables and values order before starting the search; this order remains unchanged throughout the research.

- *Dynamic heuristics:*

- select the next variable or value each time a new selection is done (so at every step of labeling).

Dynamic heuristics are potentially better (less backtracking). The computation of the perfect heuristic (which requires no backtracking) is a problem that has, in general, the same complexity of the original problem. We must therefore find a tradeoff.

Forward checking for 8 queens (first fail)

Let's solve an example with the *first fail* logic:

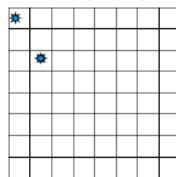


Figure 8.46

- X_3 is associated with the domain $D_3 = (5, 6, 7, 8);$
- X_4 is associated with the domain $D_4 = (2, 6, 7, 8);$

⁴⁹Smallest domain means you have less chance to assign these variables because if you have for example 2 values or 100 values then it's more difficult to assign the first one; it is easier to find a feasible value among 100 values rather than among 2 values.

⁵⁰Exactly the opposite of above: the rationale is that I want to find the most promising values.

- X_5 is associated with the domain $D_5 = (2, 4, 7, 8)$;
- X_6 is associated with the domain $D_6 = (2, 4, 5, 8)$;
- X_7 is associated with the domain $D_7 = (2, 4, 5, 6)$;
- X_8 is associated with the domain $D_8 = (2, 4, 5, 6, 7)$.

Notice that from D_3 to D_7 we have the same number of values for these domains (except for D_8), so let's follow the variable ordering starting from X_3 .

The FC search assigns $X_3 = 5$ and propagates the constraints obtaining:

- X_4 is associated with the domain $D_4 = (2, 7, 8)$;
- X_5 is associated with the domain $D_5 = (2, 4, 8)$;
- X_6 is associated with the domain $D_6 = (4)$;
- X_7 is associated with the domain $D_7 = (2, 4, 6)$;
- X_8 is associated with the domain $D_8 = (2, 4, 6, 7)$.

The domain of variable X_6 contains only one value and is chosen for the assignment:

- X_4 is associated with the domain $D_4 = (7, 8)$;
- X_5 is associated with the domain $D_5 = (2, 8)$;
- X_7 is associated with the domain $D_7 = (2, 6)$;
- X_8 is associated with the domain $D_8 = (7)$.

The domain of variable X_8 contains only one value and is chosen for the assignment and so on.

Consistency techniques

In contrast to propagation algorithms that propagate the constraints as a result of instantiations of variables involved in the problem, consistency techniques reduce the *original problem*⁵¹ by eliminating domain values that cannot appear in a final solution.

They can be applied statically or at every step of assignment (*labeling*⁵²) as powerful propagation techniques for the not yet instantiated variables.

All consistency techniques are based on a representation of the problem as a network (graph) of constraints.⁵³ The arcs can be oriented or non-oriented: for example, the constraint $>$ is represented by a directed arc, while the constraint \neq from a simple arc (undirected or doubly oriented).

⁵¹But in the real current constraint solvers they're used also during *search* (e.g. stop search at some point in a given step and consider the current situation as an original problem). They are in a way techniques that are stronger than *Full Look Ahead*.

⁵²With *labeling* we mean the assignment of a variable to a value ($X_1 = 4$).

⁵³A constraint problem can be represented as *constraint graph*: in a constraint graph we have that we have a node for each variable and you have an arc between two nodes if there is a constraint between the two.

Constraint graph

For each CSP exists a graph (constraint graph) in which the nodes represent variables and the arcs are the constraints:

- the binary constraints (R) connecting two nodes X_i and X_j ;
- the unary constraints⁵⁴ are represented by arcs that begin and end on the same node X_i .

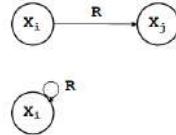


Figure 8.47

Example: Map coloring problem

Suppose we need to color portions of a floor, characterized by a number, in such a way that two contiguous regions are colored by different colors. Suppose we also have available the colors red (r), green (g) and blue (b).

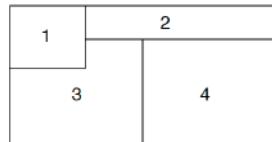


Figure 8.48

The constraint-graph is as follows. However, there are combinations of values not compatible with each other (e.g. $X_1 = r$, $X_2 = r$, $X_3 = r$, $X_4 = r$). There are several algorithms that implement different degrees of consistency.

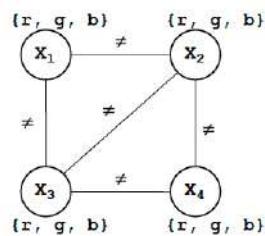


Figure 8.49

Looking at the figure above we can see both *unary constraint* (like $X_1 \leftrightarrow \{r, g, b\}$) and *binary constraint* (like $X_1 \neq X_3$). Furthermore:

⁵⁴For example in the eight queens problem each variable has the constraint to be between 1 and 8.

- X_1 is contiguous to X_2 and X_3 so we have the constraint that cannot be the same of X_2 and X_3 ;
- X_2 and X_3 are contiguous with all the others so they should be different with all the others;
- for X_4 the same logic of X_1 .

Node consistency

The purpose of building a graph is to define the *properties* of this graph: the first and the most basic one is called *node consistency*.

- *NODE-CONSISTENCY: consistency level 1.*
 - A node of a graph of constraints is consistent if for each value $X_{the} \in D_{the}$ the unary constraint on X_{the} is satisfied.

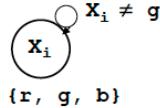


Figure 8.50

In the above example shown in the figure, the node is not consistent because the value $g \in D_i$ violates the unary constraint on X_i .

To make the node consistent it is necessary to eliminate from the domain of X_i the value g .

A graph is node consistent if all its nodes are consistent.

Arc consistency

The consistency of level 2 is obtained from a node-consistent graph and applies to arcs.

- *ARC CONSISTENCY: consistency level 2.*
 - an arc $a(i, j)$ is consistent if for each value $x \in D_i$ exists at least one value $y \in D_j$ such that the constraint between i and j is satisfied. y is called *support* for i .⁵⁵

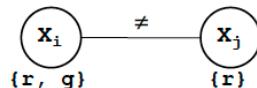


Figure 8.51

⁵⁵Note that arc $a(i, j)$ is in general different from $a(j, i)$; then depends on the relation that defines the arc and its symmetry.

The arc in the example above is not consistent because, considering the value $r \in D_i$, there is no value belonging to D_j which satisfies the constraint between them.

To make consistent the arc between X_i and X_j it is necessary to delete the value of r from the domain of X_i .

This value would not appear in any feasible solution.

A graph is arc consistent if all its arcs are arc consistent.

All this is similar to something that we've already seen; the check of arc consistency that we've done is the same we've seen for the FULL LOOK AHEAD: for each value in D_i we check a *support* in D_j , if there is no support we delete this value, so then can we say that with the FULL LOOK AHEAD we can achieve *arc consistency*? Yes it does this check, but it is not iterative, it does just one iteration. While to achieve *arc consistency* for the *entire graph* and reach an *arc consistent graph* you have to perform this check in an iterative way (see next section).

Iterative procedure

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

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

So this process should be repeated until the network reaches a stable configuration *QUIESCENT*.

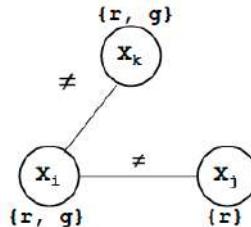


Figure 8.52

Consider the arc between X_i and X_k this constraint is arc consistent.⁵⁶

Now the constraint between X_i and X_j removes r from the domain of X_i ⁵⁷ which in turn requires the first constraint to be reconsidered.⁵⁸

Arc consistency is an iterative process that converges to a stable and arc consistent network.⁵⁹

⁵⁶We don't delete any items; because r has the support g and g has the support r for both variables.

⁵⁷The FULL LOOK AHEAD stops here while arc consistency says: we changed something in a variable, we need to re-consider all the constraints in which that variable is involved. The FULL LOOK AHEAD algorithm would lead to $X_k \leftrightarrow \{r, g\}$, $X_i \leftrightarrow \{g\}$, $X_j \leftrightarrow \{r\}$.

⁵⁸Instead the arc consistency would lead to $X_k \leftrightarrow \{r\}$, $X_i \leftrightarrow \{g\}$, $X_j \leftrightarrow \{r\}$.

⁵⁹So at the end we run this algorithm iteratively, of course it converges, and the convergence is proved to converge to an arc consistent graph. At the end of the iterative procedure, every value that you leave in a domain of a variable has a support in the domain of other variables that are involved with binary constraint with the first one. So, do you think that the values that we remove are proven infeasible? Yes, why we remove them? Just because they're infeasible. Do you think that the value that we leave there are all feasible? Not in general! We will see why in the next example.

Arc consistency: example (cont.)

Consider the constraints $X_1 < X_2 < X_3$ and the domains with $X_1, X_2, X_3 :: [1, 2, 3]$.

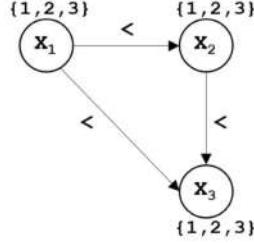


Figure 8.53

Before iteration.

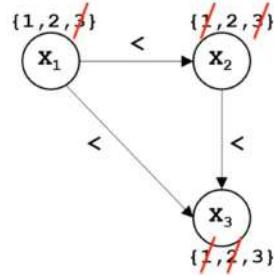


Figure 8.54

Looking at the figure above:

- if we consider arc $a(1, 2)$ we remove 3 from X_1 since there is no value $x_2 \in X_2 : 3 < x_2$;
- then we consider arc $a(2, 1)$ and we remove 1 from X_2 since there is no value $x_1 \in X_1 :: [1, 2]$ s.t. $x_1 < 1$;
- then arc $a(2, 3)$ and we remove 3 from X_2 since there is no value $x_3 \in X_3 : 3 < x_3$;
- then arc $a(3, 2)$ and we remove 2 from X_3 since there is no value $x_2 \in X_2 :: [2]$ s.t. $x_2 < 2$;
- for the same reason we remove 1 from X_3 ;
- comparing then $a(1, 3)$ and $a(3, 1)$ would not change the situation achieved
⇒ this is what we achieve with FULL LOOK AHEAD; just with the *first iteration*.⁶⁰

Second iteration.

⁶⁰We considered the comparing in this order: $[a(1, 2), a(2, 1), a(2, 3), a(3, 2), a(3, 1), a(1, 3)]$; should we obtain the same result for: $[a(3, 1), a(1, 3), a(3, 2), a(2, 3), a(2, 1), a(1, 2)]$? Check it (it should be yes!).

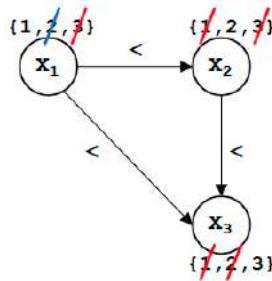


Figure 8.55

Looking at the figure above. There have been deletions in the first iteration, so there have been changes; the situation changed. Why cannot we perform another iteration on a situation that is different?

- Consider again arc $a(1, 2)$, we remove 2 from X_1 since there is no value $x_2 \in X_2 :: [2]$ s.t. $2 < x_2$;
- The other comparison would lead the situation unchanged.

Third iteration: *quiescence*.⁶¹

The *FLA* instead led to domains: $X_1 :: [1, 2]$, $X_2 :: [2]$, $X_3 :: [3]$. Less pruning vs *AC*, but lower computational cost (*FLA* is also called *AC1/2*).⁶²

Arc consistency

The arc consistency can be applied:

- before the search, as preprocessing to produce a simplified problem with the same solutions of the original one or;
- as propagation step (as done for Full Look Ahead) after each variable assignment, it is often called *Maintaining Arc Consistency* (MAC).⁶³

Many algorithms have been proposed. The most famous is called *AC – 3* (Mackworth, 1977):

- use a queue of arcs (*queue*), and cycles until it is empty;
- as soon as the domain of a variable X_i is reduced, we add in queue all arcs (X_k, X_i) for each variable X_k connected by an arc with X_i .⁶⁴

⁶¹At the end of this *arc consistency* also you achieve a network that is basically guaranteed to be *arc consistent*.

⁶²Basically because it is the same algorithm, but it is not iterative.

⁶³So *during search*: who does prevent us to run arc consistency after the assignment of a variable? The only difference – after the assignment – is that we have a new problem where *the assigned variable has a domain of only one value*.

⁶⁴Why are we guaranteed that this algorithm does not run indefinitely? Because the domains are discrete and finite.

Arc consistency vs Solved constraints

Let's make an in-depth and a precision. Suppose to have $X :: [3, 4]$, $Y :: [6, 7]$, $X < Y$; the constraint $X < Y$ is said to be *solved*; the same for $X \neq Y$: every pair I consider (*Cartesian Product*) is consistent/solved.

Now suppose to have $X :: [3, 4]$, $Y :: [3, 4]$, $X \leq Y$; it is *solved*? (Remember it is considered *solved* only if you pick any value from X and any for Y and they're okay [cartesian product].) No! Because if you take $4 \leq 3$ it is inconsistent with the constraint. But attention X and Y are *arc consistent*!

So *arc consistent* is different from to be *solved*: *solved* means that the *cartesian product* of the domains contain only feasible couples.

AC-3 algorithm (Mackworth, 1977)

Let's consider the *pseudo-code* of the *AC – 3* algorithm (Mackworth, 1977).

```

function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp
  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if RM-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue

function RM-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff remove a value
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy constraint( $X_i, X_j$ )
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed

```

Figure 8.56

Let's analyze the above code.

We have a *csp* which is the problem itself (we have variables, domains, constraints); we have a *queue* that is a queue of arcs and initially we have all constraints in the *queue*; we are considering binary constraints, they are arcs, not hyperarcs (we consider 2 variables, 2 nodes).

While *queue* is not empty I remove one arc (the first one), if the function *RM-INCONSISTENT-VALUES* return *true* – and this means basically to check if there is any deletion during the *arc consistency* validation between X_i and X_j – then reconsider the constraints with the neighborhoods after the deletion (of

the inconsistent values of X_i with X_j). Note that after eventual deletions the domain of X_i is changed, so also the relations with other change, note also that X_j can be reinserted in the queue if is in the neighborhoods (but notice that the order has been inverted (X_k, X_i) and not (X_i, X_k)).

Observation

Let's make an important question: do you think that in an *arc consistent graph* every value that is left in the domains of the variables is part of a feasible solution? Obviously the values that we've removed are infeasible with a solution, we are sure of that. But the ones that are left? Do you think they're feasible meaning that for sure they're part of a feasible solution? Or, alternatively: does an arc consistent network guarantee to contain a solution? No! Even if you have only binary constraints there may be inconsistencies of higher order. Let's see! Suppose we consider the network of constraints related to an instance of map coloring problem:

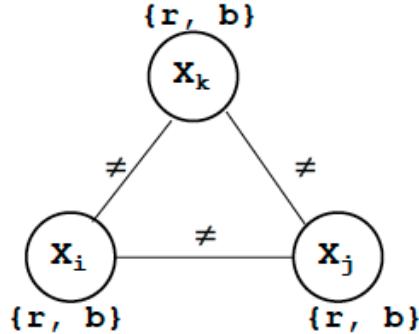


Figure 8.57

This network is arc-consistent: in fact, for each value of each domain, there is at least a value in every other domain that matches the existing constraint between the two nodes.⁶⁵

However, it is immediate to see that the network has no solution.⁶⁶

All values removed by arc-consistency are not part of any feasible solution BUT values that are left in the domains are not necessarily part of a consistent solution.

Path consistency: level 3

The consistency of level 3 is obtained starting from an arc-consistent graph.

- *PATH-CONSISTENCY: consistency level 3.*

⁶⁵For all r has the support of b and b has the support of r .

⁶⁶Clearly having three variables with only 2 values each, this is an over constraints problem, an infeasible problem on the context of map coloring problem! Indeed there are higher level consistencies and for understanding that this network does not have any solution you have to go up; so we have: node consistency for level 1 consistency; arc consistency for level 2 consistency; path consistency for level 3 consistency.

- A path between the nodes (i, j, k) is path consistent if, for every value $x \in D_i$, and $y \in D_j$ (that are node and arc consistent) there is a value $z \in D_k$ that satisfies the constraints $P(i, k)^{67}$ and $P(k, j)$.

Note: the consistency of the unary constraint $P(k)$ is guaranteed by the node consistency.

Path consistency: example 1 (cont.)

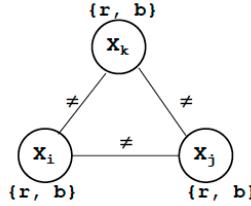


Figure 8.58

Additional data structures needed, for each pair of variables (for each arc) we need to store pairs of compatible values for which exist a support in the domain of any third variable (example: for the arc X_i to X_j , none of the pairs (r, b) and (b, r) has a support value in the domain X_k).⁶⁸

The network cannot be made path consistent thus leading to a failure.

By adding a color to the available ones: r, g, b .

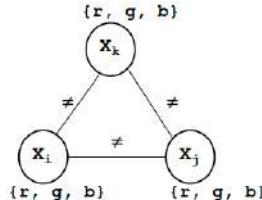


Figure 8.59

The network is PC.

Checking the consistency of level $n = 3$ for this network that has exactly $n = 3$ variables is equivalent to verify that there is a solution AND all values that are left in the domain are part of a consistent solution.

This happens because the level of consistency is equal to the number of variables.⁶⁹

K-Consistency

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

⁶⁷This indicating the binary constraint between i and k .

⁶⁸More formally: (X_i, X_j, X_k) path consistent if $\forall i \in D_i :: [r, b], \forall j \in D_j :: [r, b] \exists k \in D_k :: [r, b]$ s.t. $k \neq i \wedge k \neq j$. Counter example: $i := r, j := b$; if $k := r \implies r \neq r \wedge r \neq b$ which is not a true relation and the same if $k := b$.

⁶⁹This can be generalized in a principle, see next page!

solution, we need to apply k -consistency, *i.e.*:⁷⁰

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

In general, if a graph containing n variables is k -consistent with $k < n$, then search in the remaining space is needed.

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

Freuder in 1978 has defined a general algorithm to make a network k -consistent for any k .

However, making n -consistent a network of n variables has a complexity exponential in n (*the cost is the same as solving the original problem*).

Constraint solvers in practice

Constraint solvers are tools to solve CSPs that embed all techniques seen so far. Search and propagation. They typically use arc-consistency for the propagation of constraints and search.

They are based on the so called Constraint Programming languages.

The constraints are seen as software components that encapsulate a *filtering* algorithm. Very often the *filtering* algorithm making the propagation is not general purpose like those seen so far, but is based on the semantics of the constraint for efficiency reasons.

Example: $X :: [1..10]$, $Y :: [1..10]$, $X > Y$ we don't have to check all values in the two domains, but we just have to check out *bounds*. In particular, this constraint is AC if $\min(X) > \min(Y)$ and $\max(X) > \max(Y)$.

Example $X :: [1..10]$, $Y :: [1..10]$, $X \neq Y$. This constraint is always AC if the two domains contain more than one value. Therefore the constraint is propagated only when one of the two variables is instantiated to a value. This value is removed from the domain of the other.

A key feature of the constraint solver is the presence of n -ary⁷¹ constraints, also called GLOBAL constraints.

Also they embed a filtering algorithm.

Clearly to achieve the consistency of a n -ary constraint (Generalized Arc Consistency), in principle, you should apply the n -consistency, which has exponential complexity:

- however, there are particular constraints for which reaching the n -consistency has polynomial cost (*e.g.* AllDifferent);
- for others an approximation of Generalized Arc Consistency is achieved.

CSP and Optimization

We considered only CSPs in which variables have discrete domains. Most of these problems are NP-hard, that are problems for which has not yet been found,

⁷⁰For being sure that all the values that we leave in the domain are part of a feasible solution in a problem with k variables you have to achieve k -consistency (but k -consistency has an exponential complexity).

⁷¹Not only binary.

and probably does not exist, an algorithm able to find the solution in polynomial time in the size of the problem.

A Constraint Optimization Problem (COP) is a constrained problem in which an objective function is added. A COP is then formally described as a CSP whose purpose is not only to find a feasible solution, but the optimal solution according to a certain evaluation criterion.

The general algorithm to solve a CSP can then be used to solve any COP. In fact, after having described the problem in terms of variables, constraints and domains, just add a further variable that represents the objective function.

Whenever a CSP solution is found, a new constraint is added that ensures that any future solution has a better value of the objective function. This process continues until you can no longer find any solution.

The last solution found is the optimal solution.

8.2 Exercises on Constraints

Let's see some exercises on *constraints*. The following will be probably more elaborated in comparison to what proposed at the exam. In general there are two kinds of exercises, one where given a problem you have to create the model and then you solve it; the other the problem is given with the model and you have just to solve it.

Exercise 1

Consider the following constraints network:

- $X :: [3, 6, 10..15], Y :: [7..12], K :: [4..15], Z :: [5..10];$
- $Y = Z, Z < K, K > Y, X \leq Y.$

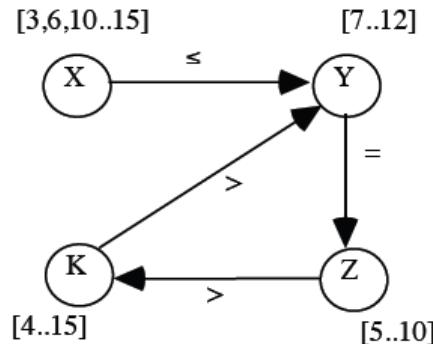


Figure 8.60

Apply arc-consistency.⁷² In addition discuss what happens by applying the arc-consistency to the same network if we add the constraint $X = K$.

Arc consistency leads to:

⁷²In the figure above don't read the arrows like should have the same verse of the constraints (e.g. $Z < K$). The arrow is just to say that there is a no bidirectional binary relation.

- $X = [3, 6, 10], Z = [7..10], K = [8..15], Y = [7..10]$.

Let's try to understand why we obtained the above solution; let's consider constraints in order.

- $Y = Z \rightarrow$ what happens? You have to find the *intersection* of the domains because to be equal a value should be in both the domains for having a *support*. So $Y :: [7..12] \cap Z :: [5..10] \implies Y = Z :: [7..10]$.
- $Z < K \rightarrow Z :: [7..10]$ (we should take into account the previous constraint and not consider the original domain) should be less than $K :: [4..15]$, so $Z :: [7..10] < K :: [4..15] \implies \min(Z) < \min(K) \wedge \max(Z) < \max(K)$ ⁷³ $\implies K :: [8..15]$.
- $K > Y \rightarrow$ nothing happen since already true $\min(Y) < \min(K) \wedge \max(Y) < \max(K)$.
- $X \leq Y \rightarrow \max(X) \leq \min(Y)$? NO!!!⁷⁴ We need to ask: has 3 at least one *support* in Y ? Yes! Has 6 at least one *support* in Y ? Yes! 10? Yes! 11? No because $\nexists y \in Y :: [7..10]$ t.c. $11 \leq y$ and so on...

Introducing the new constraint leads to a failure. Note the incremental computation.

Introducing the new constraint do you think we should restart from scratch or from the last point? From the last point! The addition of every constraint is incremental! Basically because the introduction of new constraints reduces further the domains, does not make them larger!

Anyway $X = K \implies X :: [10], K :: [10]$; then other constraints will be awaked: in particular – for instance – $Z :: [7..10] < K \implies Z :: [7..9]$; then $K > Y :: [7..10] \implies Y :: [7..9]$; then $X :: [10] \leq Y :: [7..9] \implies$ fails! $\nexists y \in Y$ s.t. $10 \leq y$.

Let's do something more: now we have achieved an arc-consistent network with the arc consistent domains that are what listed above.

So this is still the *original problem*, we have not yet done any *search*. So since we've reduced the domain we start from here.

Let's starting search! We start assigning X to 3.

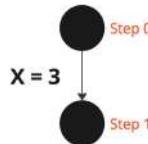


Figure 8.61

⁷³We can also avoid to write this, writing directly the result but is just to understand. You need to apply the definition, not always the formula like *min/max* applied above is equivalent to the definition which is the thing to apply.

⁷⁴Pay attention! Before reasoning in terms of *min/max* worked since we had contiguity in the values of the domains (in the sense that one was the successor of the other), here X jumps from 3 directly to 6 and from 6 to 10. Remember to apply the definition of *Arc-Consistency*, reason in terms of *support*.

What happens to the domains? Nothing since the only constraint involved is $X \leq Y$ and we have $3 \leq y \forall y \in Y$ (domains of Z and K are not involved). Now, between Z , K , Y I can decide to use either Z or Y because they have exactly 4 values in the domain while K has more than 4 values, so considering the *First-Fail* logic (also above we used it so we started from X since it had the lowest cardinality), so let's consider Z and (just following the order of the variables) we put $Z = 7$.

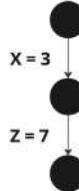


Figure 8.62

Now we have changes in the domains *i.e.* $Y :: [7]$ since it should be true the constraint $Y = Z$, while $K :: [8..15]$ remains unchanged since the constraints involving K are satisfied $\forall k \in K$.

So a possible solution is:

- $[X, Z, Y, K] = [3, 7, 7, 8]$.

Exercise 2

Given the following constraints:

- $A :: [1..4], B :: [1..4], C :: [2, 4, 6], D :: [1, 7, 9], E :: [2..5];$
- $A > D, A \neq B, B < C, C > D, E = A.$

Draw the graph corresponding to the constraint satisfaction problem⁷⁵ and apply arc-consistency. Show the search tree *to get to the first solution* using as a heuristic of the first-fail assignment of values to variables (MRV)⁷⁶ at each instantiation apply the arc-consistency to the remaining network.⁷⁷

Graph corresponding to the problem:

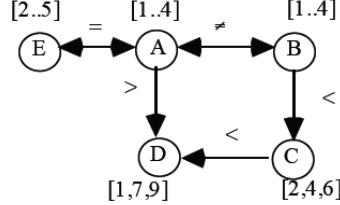


Figure 8.63

⁷⁵In general it is not asked at the exam.

⁷⁶Minimum remaining value.

⁷⁷Basically what we did before.

After the application of the arc-consistency to the original problem we obtain:⁷⁸

- $A :: [2..4]$, $B :: [1..4]$, $C :: [2, 4, 6]$, $D = 1$, $E :: [2..4]$.

Let's see the *search*:

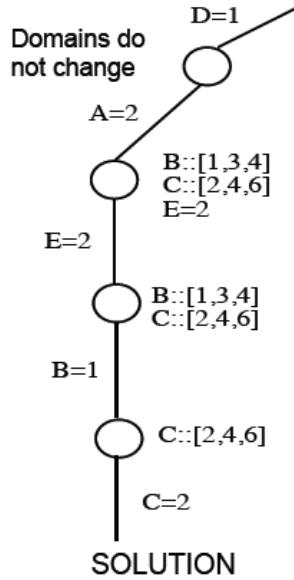


Figure 8.64

So looking at the figure above we have basically the following sequence:⁷⁹

0. arc-consistency;
1. $D = 1$;
2. arc-consistency (the same of step 0 so we have no changes because we reached $D = 1$ at the first application of arc consistency (level 0));
3. $A = 2$ (I choose A because of *first-fail* on the first value);
4. arc-consistency ($E = 2 = A$, from B remove 2 since $A \neq B$, C doesn't change since no links with A);
5. $E = 2$ (forced by step 4);
6. arc-consistency (the same as what done in step 4 so B, C unchanged);
7. $B = 1$ (since *first-fail*);
8. arc-consistency (constraints respected $\forall c \in C$);

⁷⁸So before starting search we apply arc-consistency which reduces the domains but still referring to the original problem. Then we start search making first assignment; then check again arc-consistency and so on...

⁷⁹For the variable selection we consider the *first-fail* principle, and for the value the *lexicographic* order (leftmost value in the domain).

$$9. C = 2.$$

We obtain the following *solution*:

- $A = 2, B = 1, C = 2, D = 1, E = 2.$

Exercise 3

Given a 4×4 checkerboard and 4 colors $[r, b, g, y]$, a color must be placed in each cell of the board so that each row, each column and the two diagonals of the board contain different colors.

Model the problem as a CSP, and solve it *until the first solution* via the forward checking technique with first-fail heuristic (also known as Minimum Remaining Values MRV).

Solution.

- Each cell of the checkerboard is a variable $X_{11} \dots X_{44}$.⁸⁰ The initial domains of the variables are composed of the four available colors.⁸¹
- The constraints are:
 - for all i $X_{ij} \neq X_{ik}$ for $j \neq k$;
 - for all i $X_{ji} \neq X_{ki}$ for $j \neq k$;⁸²
 - for each i and j $X_{ii} \neq X_{jj}$ with $i \neq j$;⁸³
 - for each i and j , $X_{i,4-i+1} \neq X_{j,4-j+1}$ with $i \neq j$.⁸⁴

Let's solve it by a forward checking; suppose this is a tree.

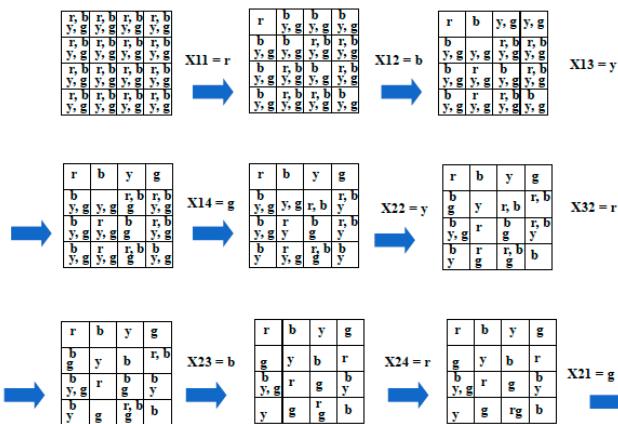


Figure 8.65

Looking at the figure above:

⁸⁰ $4 \times 4 = 16$ cells divided in rows and columns.

⁸¹ We can also consider numbers associated with colors. It's the same.

⁸² These first two constraints tell us that colors should be all different on each row and on each column.

⁸³ Condition for the principal diagonal.

⁸⁴ Condition for the other (secondary) diagonal (diagonal that pass on the center but from right to left).

- $X_{11} = r \leftrightarrow$ we make this assignment and we remove r from first column, first row, and principal diagonal;
- $X_{12} = b \leftrightarrow$ the same here and so on...
- $X_{14} = g \leftrightarrow$ see the next;
- $X_{22} = y \leftrightarrow$ since *first-fail* remember that we choose variable with the lowest cardinality (so after X_{14} we take X_{22} and not X_{21} which has cardinality $3 > 2$ of X_{22}); then following lexicographic order.

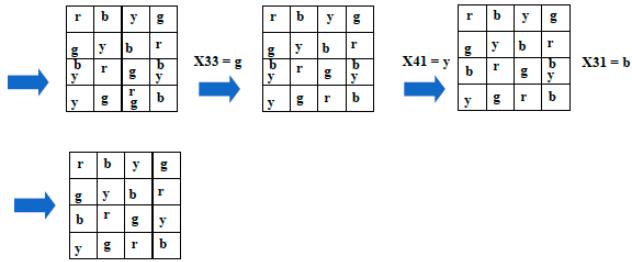


Figure 8.66

Appendices

Appendix A

Search strategy

Search strategies lab

Aim of this laboratory session:

1. assess the comprehension of the search strategies saw in the first two chapters;
2. learn to exploit existing libraries, in particular the AIMA implementation available on the net.

Content:

1. short recap of main search strategies;
2. few considerations on how to represent a state;
3. short introduction to the AIMA python library;
4. exercises.

Requirements

Requirements:

- Python 3 (correctly installed);
- editor for Python (*e.g.* Pycharm CE);
- (`git`-)clone the AIMA repository: <https://github.com/aimacode/aima-python>
 - if you don't have `git` installed, then it's time to install it;
- prepare a new Python project:
 - `virtual env` is suggested, but not mandatory;
- install module `numpy`;
- copy in your project the following files:
 - `search.py`¹

¹It contains implementation of the search strategy function.

– `utils.py`

- install **Jupyter**, there are some notebook already prepared in the **AIMA** repository.

Looking for solutions

Few concepts.

- **Expansion:**² given a state, one or more actions are applied, and new states are generated.
- **Search strategy:**³ at every step, let us choose which will be the next state to be expanded.⁴
- **Search tree:**⁵ the expansion of the states, starting from the initial state, linked to the root node of the tree.
- The leaves of the tree (the **frontier**) represent the set of states/nodes to be expanded.

Search strategies

Search strategies can be **Non-informed** or **Informed**.

Non-informed search strategies:⁶

- breadth-first (uniform cost);⁷
- depth-first;
- limited depth-first;
- iterative deepening.

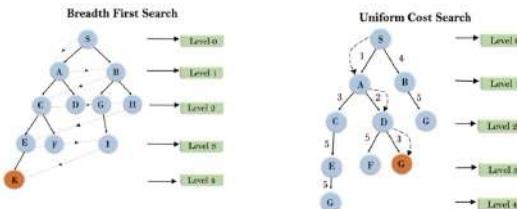


Figure A.1

²Concept implemented in the **AIMA** library.

³For example **breadth-first**.

⁴For example following an arbitrary choice or following an heuristic with an estimation of the distance from the goal; this determines the strategy.

⁵Structure of your search.

⁶The idea of **Non-informed** is that you can choose with... for example in the breadth-first you choose always the node with a left most choice between two nodes at the same depth, so you have not an information or knowledge about something in your tree, if you want to use information or knowledge of your tree you have to define a **heuristic** (**Informed** search strategy).

⁷There are other breadth-first with no uniform cost; we've seen only the uniform case.

Looking above we can see **Breadth First Search**, its most particular features are that is *complete* and *always the search is with less deep node*; the idea is you can have problem of memory (computational problem) because you have more than one path open *simultaneously* (main problem of the breadth-first).

We can solve this problem in some way with **Uniform Cost Search**, so if you have a cost associated to a part of a path you can choose the *min* cost path in your breadth first, the idea is that breadth first is always with 1 for each step of the search (cost equal to 1).

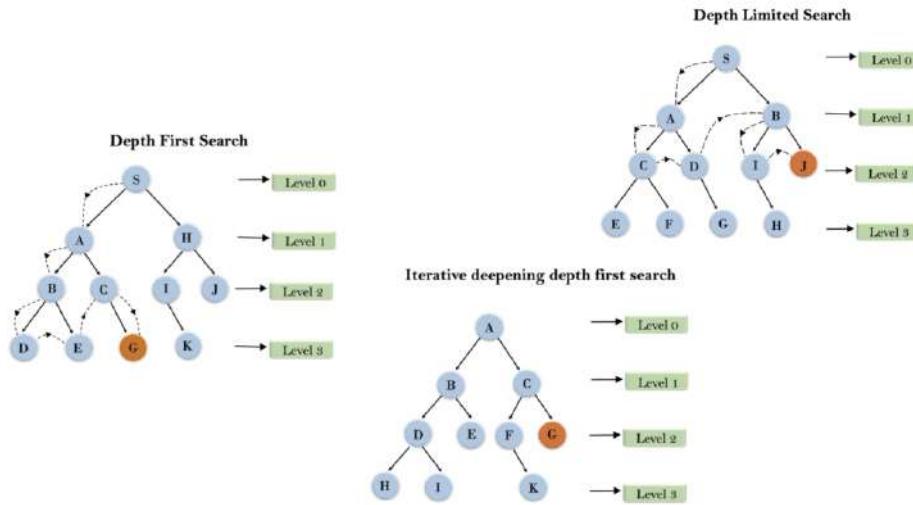


Figure A.2

Depth First Search is the other important **Non-informed** search strategy. The idea is that you always explore the deepest node so the difference with breadth-first is that you are more efficient but you're not guaranteed to find the complete solution. One more problem in the depth-first tree search is you can have *infinite loop* because you have a repetition (we will see this problem in the first exercise ahead).

You can try to improve your **Depth First Search** by imposing a *limit* with **Depth Limited Search** but also in this case we can have the problem that if the depth of the solution is higher than the limit you've not the completeness of your search strategy.

Finally, the **Iterative deepening** is a sort of **Depth First Search** that increase iteratively the limit of the search; so you have the advantages of the **Breadth First Search** and the **Depth First Search**.

Informed search strategies:⁸

1. **greedy** $\rightarrow f(n) = h(n);$

⁸The Informed search strategies that we've seen are based on the best-first search and are: **greedy** and **A***. The main difference with the Non-informed is that we associate a *knowledge* in the search strategies and in particular the main idea is to explore the closest nodes to the goal. With **greedy** the function for each node is only the *distance* you expanded the closest node to the goal. But with the **A*** you have a sum that is composed, so we have to consider two parts of your function.

2. A* $\rightarrow f(n) = g(n) + h'(n)$ where $g(n)$ is the depth of the node, and $h'(n)$ the estimated distance from the goal;
3. IDA*;
4. SMA*.⁹

Data structure for the search tree

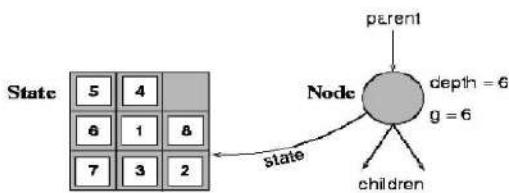


Figure A.3

Structure of a Node:

- the *state* (in the state space) to which this node corresponds;¹⁰
- the parent node;
- the action applied to get this node;
- the path cost for reaching this node.

The general search algorithm

The following is the general algorithm for all search strategies:

```

function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

```

Figure A.4

Let's analyze the general purpose code above.

⁹Then you can see other informed search strategies (IDA*, SMA*) but the most important for this course are the greedy and the A*. In general see the *Russel Norvig* textbook for a complete introduction to search strategies.

¹⁰The *state* is a representation of your node and in particular considering the implementation (with code, with AIMA library etc.) you have to decide how to represent this state so, for example, with a matrix, with a tuple, with a simple integer and so on... So you have to chose the representation of your state and you have to consider that you have a *Node* for each state with a possible parent node and children nodes; you have to apply an action on this node and then you have to build a path to the solution.

- In the first row we are essentially saying that we have a *problem* and a *strategy* to use and then we have to return a *solution* or a *failure*. We are going to see in the first exercise a function that define all the possible failures (better: all the possible *states* that represent *failures* on the search strategy). So we have to initialize the search tree using the *initial state* of the problem (a description of the initial state) and you have to do a *loop*.
- At the third row begins the *loop*; the idea is well explained in the loop body.
- In the sixth row – inside the loop body – is indicated to *choose a leaf node*; you have always to choose a *leaf node* for expansion. So, for example, if we have an *heuristic* we have to consider the rules of the heuristic itself.

First step: the problem definition

Essentially we have a description of a problem in a natural language and we have to understand the best representation of the initial state, the actions and the goal.

- How to represent a ‘*problem*’?
 1. Usually, represented by means of ‘*states*’, together with state operators (aka actions).
 2. There is a initial state.
 3. A goal to reach (a property that should hold, *i.e.* a state for which the property holds).
- How to represent a state?
 - Through data structures... but also how to represent a state with a *notation*; so we have to decide a convention (this is a difficult part).

The AIMA search library

The AIMA library provides two classes:

- **class Problem**: our problem instance will extend this class;
- **class Node**: we will just access it... (you can just access to this class and its methods);
- from the **search.py** code that contains the two classes mentioned above we have the following general comment.

....

Search (Chapters 3–4)

The way to use this code is to subclass Problem to create a class of problems, then create problem instances and solve them with calls to the various search functions.

....

The **class Problem** is abstract (*i.e.* you must implement some methods); consider the following comment.

```
"""
The abstract class for a formal problem. You should
subclass it and implement the methods actions and result
and possibly__init__, goal_test, and path_cost. Then
you will create instances of your subclass and solve them
with the various search functions.
"""
```

The constructor `__init__` of the **class Problem**

```
def __init__(self, initial, goal=None):
    """
    The constructor specifies the initial state, and
    possibly a goal state, if there is a unique goal.
    Your subclass's constructor can add other arguments.
    """
    self.initial = initial
    self.goal = goal
```

However, the state must be a tuple (why? Graph search, the possibility of computing the hash).

The **class Problem**

The following are the main methods of the class that must be implemented in the problem of your exercises.

```
def actions(self, state):
    """
    Return the actions that can be executed in the given
    state. The result would typically be a list, but if
    there are many actions, consider yielding them one at a
    time in an iterator, rather than building them all at
    once.
    """

def result(self, state, action):
    """
    Return the state that results from executing the given
    action in the given state. The action must be one of
    self.actions(state).
    """

def goal_test(self, state):
    """
    Return True if the state is a goal. The default
    method compares the state to self.goal or checks for
    state in self.goal if it is a list, as specified in the
    constructor. Override this method if checking against a
    single self.goal is not enough.
    """
```

Considering the `def actions(self, state)` we highlight that the `state` is an argument and the result would be a `list` (a list of actions); so in general you have to define this list of possible actions for each possible given state.

Let's implement these methods

Which are the search strategies that I can apply?

1. Breadth-first.
2. Depth-first.
3. Depth-bounded.
4. Iterated deepening.

What about the path of a solution?¹¹ The `class Node` keeps track of the path.

What about the Uniform Cost Strategy?

But we need to keep into account the *cost* of the actions that we use in each state.

We need to keep into account also the cost of the operators/actions that we applied to reach the goal.

You should implement the method.¹²

```
def path_cost(self, c, state1, action, state2):
    """
    Return the cost of a solution path that arrives
    at state2 from state1 via action, assuming cost c to
    get up to state1. If the problem is such that the
    path doesn't matter, this function will only look at
    state2. If the path does matter, it will consider c
    and maybe state1 and action. The default method costs
    1 for every step in the path.
    """
```

A default method is provided, which considers the constant cost 1 for every action.

Informed Strategies?

Remember the notion of heuristic: a function that estimates (with a certain error) the distance of a state from the goal.¹³

You should implement the method:

```
def h(self, node):
    """
    Returns the heuristic applied to the Node node
    """
```

¹¹We will see also that we can build a path for the solution. We consider both *path cost* and *path of a solution* in AIMA library.

¹²After that we will see tha first exercise this would be more clear. The important thing is to understand the concepts.

¹³It is difficult to establish and implementing a performant heuristic.

Summing up

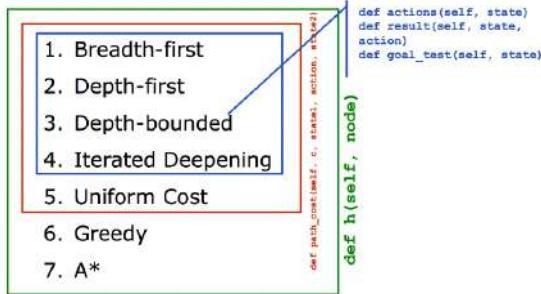


Figure A.5

Notice: applying intelligent strategies means a cost in terms of the knowledge you should provide... notice as going from 1. **Breadth-first** to 7. **A*** we need to define more methods.

This is one of the main idea of the AIMA library.

The library/module `search.py`

Given an implementation of the problem, provides the following functions/methods:

```

def breadth_first_tree_search(problem) :
    """
    Search the shallowest nodes in the search tree first.
    Search through the successors of a problem to find a
    goal. The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

def breadth_first_graph_search(problem):
    """
    """

def depth_first_tree_search(problem):
    """
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a
    goal. The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

def depth_first_graph_search(problem):
    """
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a
    goal. Does not get trapped by loops.
    If two paths reach a state, only use the first one.
    """

```

Note that we have two possibilities of implementation, *tree* or *graph*. The idea is that with:

- **tree_search** strategy we can have *infinite loop* because we can have repetition of the states;
- **graph_search** strategy we have the nodes of the already visited nodes and so we avoid repetition.

In the first exercise we will see an example where we can see an infinite loop with the **depth_first_tree_search** and by debugging this function in the **search.py** with a simple **print()** function we will see that we have a clear infinite loop. The repetition of the same four steps each time in the expansion.

Other function/methods:

```
def uniform_cost_search(problem):
    def depth_limited_search(problem, limit=50):
    def iterative_deepening_search(problem):
    def astar_search(problem, h=None):
    def hill_climbing(problem):
    def simulated_annealing(problem, schedule=exp_schedule()):
    def genetic_search(problem, fitness fn, ngen=1000,
                      pmut=0.1, n=20):
```

We will use in these cases the A* search (**astar_search**). So we have to define the problem (better: instance of the problem) and if we have an heuristic in the argument.

If we want to use the other search strategies we can see the material available on the net about AIMA library.

How to use it?

- Define the **class Problem**, representing your problem.
- Import the file **reporting.py** (utilities...).
- Looking for a solution?

```
myP = MyProblem(...)
soln = breadth_first_tree_search(myP)
path = path_actions(soln)
print(path)
print("Cost: ", soln.path_cost)
path = path_states(soln)
print(path)
```

This is a general structure, general schema. Remember this schema, generally we can always use it to solve exercises:

- with `myP = MyProblem(...)` \longleftrightarrow we instantiate our problem;
- with `soln = breadth_first_tree_search(myP)` \longleftrightarrow we obtain a solution;
- with `path = path_actions(soln)` \longleftrightarrow This method return the path of actions of a solution;
- with `soln.path_cost` \longleftrightarrow we have the cost of this solution;
- with `path_states(soln)` \longleftrightarrow we have the states of this solution.

- Want to compare strategies?
-

```
report([
    breadth_first_tree_search,
    breadth_first_graph_search,
    # depth_first_tree_search,
    depth_first_graph_search,
    astar_search],
    [myP])
```

In AIMA library we can also compare the strategies by using the method `report`; so with this method we can add a list of strategies and the important thing is to write the instance of the problem you want to solve with all these strategies (and we will see the differences).

During this lab session

We are going to.

1. Prepare a new python project.
2. Choose the first problem, define the state and the functions, and test the different strategies.
 - Missionaries and Cannibals.
3. Define the state and the functions, and test the different strategies for the second and the third problem.
 - U2.
 - Fill the 10×10 matrix.

A simple problem: Missionaries and Cannibals

A classical problem for search strategies is the *Missionaries and Cannibals* problem.

- 3 missionaries and 3 cannibals are on the same shore of a river, and want to cross such river. There is a single boat (on the same initial shore), able to bring around max two persons at a time.
- Should it happen that in *any* shore there are more cannibals than missionaries, the cannibals will eat the missionaries (failure states).¹⁴

¹⁴Constraints: so we can have *max* 2 individuals in the boat and, in any shore, we should have the same number of M (missionaries) and C (cannibals) otherwise C will eat M. While the *goal* is that at the end we should have all the 3 missionaries and 3 cannibals cross the river with the boat.

- Which state representation?¹⁵
 - State: a tuple of three numbers representing the number of missionaries, the number of cannibals, and the presence of the boat on the starting shore.
- The initial state is: $(3, 3, 1)$ where respectively we have $|M|$, $|C|$, *boat present* (while we use 0 to indicate boat not present in the initial shore; we consider this tuple formulation always referring to the initial shore as frame of reference).

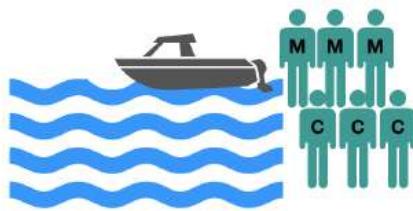


Figure A.6

- Actions: the boat cross the river with passengers:¹⁶
 - 1 missionary, 1 cannibal;
 - 2 missionaries;
 - 2 cannibals;
 - 1 missionary;
 - 1 cannibal.
- Goal: state $(0, 0, 0)$ (this, as already said, considering the initial shore; $C = 0$ in the initial shore, $M = 0$ in the initial shore, the boat is not in the initial shore but in the opposite).
- Path cost: the number of river crossings.¹⁷

A possible solution: Missionaries and Cannibals

Let's see how to structure a possible solution:

```
from reporting import *
```

```
class MC(Problem):
    """How to represent the state?
    Choice: state as a tuple, indicating
    the number of missionaries, the number of cannibals,
    and the presence of the boat on the starting river
```

¹⁵The interesting thing is that in our state representation we have to consider the situation in *both shores* because we could have a failure if in one shore $|C| > |M|$. So we should need to consider a combination of **tuple**.

¹⁶Otherwise we reach surely a failure state.

¹⁷We consider this; we will see the possibility to consider this heuristic.

Hence, the initial state is (3, 3, 1)
The goal state will be (0, 0, 0)"""

```
def __init__(self, initial, goal):
    """Constructor"""
    self.initial = initial
    self.goal = goal
    Problem.__init__(self, initial, goal)

def isValid(self, state):
    m, c, *_ = state
    if (m>0 and c>m) or ((3-m)>0 and (3-c)>(3-m)):
        return False
    else:
        return True
```

Remember to import `reporting.py` from my repository <https://github.com/giodesi/FundamentalsOfAI.git> and probably in `reporting.py` you have to import also the `math` module depending on the the Python version you are using. Remember that a *state* like the initial one (3, 3, 1) considers as point of reference always the starting river.

Considering the above function `isValid` we highlight that in general a function like this is always important to define because we need to define always a *failure state*. Note that the argument is the `state` we are considering and the logic condition that `return False` represents our *failure condition*. In the specific, analyzing the logic condition we have:

- ($m > 0$ **and** $c > m$) \longleftrightarrow positive number of missionaries and number of cannibals greater than number of missionaries (in the initial shore);
- $((3-m) > 0$ **and** $(3-c) > (3-m)$) \longleftrightarrow but the idea is that we can also have failures in the opposite shore at the same time so we have always to consider also the opposite shore.

Now we need to define the actions:

```
def actions(self, state):
    """The actions executable in this state."""
    if not self.isValid(state):
        return []

    m, c, b = state
    result = []
    if m > 0 and c > 0 and b:
        result.append('MC->')
    if m > 1 and b:
        result.append('MM->')
    if c > 1 and b:
        result.append('CC->')
    if m > 0 and b:
        result.append('M->')
    if c > 0 and b:
        result.append('C->')
    if (3-m) > 0 and (3-c) > 0 and not b:
```

```

        result.append('<MC')
if (3-m) > 1 and not b:
    result.append('<MM')
if (3-c) > 1 and not b:
    result.append('<CC')
if (3-m) > 0 and not b:
    result.append('<M')
if (3-c) > 0 and not b:
    result.append('<C')
return result

```

Let's analyze the code above:

- **return []** \longleftrightarrow if we have a failure state we need to return a void list of actions;
- **m, c, b = state** \longleftrightarrow assign to m, c, b the values that compone our state;
- **result = []** \longleftrightarrow start with a void list;
- **if m > 0 and c > 0 and b:** \longleftrightarrow if missionaries are greater than zero and cannibals are greater than zero and b = 1 (1 corresponds to **True**, while 0 to **False**) i.e. the boat is present \implies we can move one missionary and one cannibal ('MC->' is a string and represents the notation for this move; the string '->' means mooving from the initial shore to the second);
- **if m > 1 and b:** \longleftrightarrow if the number of missionaries is greater than 1 we can move 2 missionaries ('MM->');
- *etc.;*
- from top till **if c > 0 and b:** \longleftrightarrow convention fro moving people from the initial shore;
- regarding the other shore we have the second part of *actions* starting from the following;
- **if (3-m)> 0 and (3-c)> 0 and not b:** \longleftrightarrow note the notation regarding the opposite shore (3-m), (3-c); essentially with this condition we are saying that if in the second shore there are at least one missionary and one cannibal and there is the boat \implies then move 1 M and 1 C to the first shore (the string '<-MC' means mooving from the second to the initial shore);
- *etc.;*
- **return result** \longleftrightarrow finally we return the list of actions for the given state.

Then we need to implement also the **result** for each **action** applied in a given **state**:

```

def result(self, state, action):
    """The state that results from executing this
       action in this state."""
    m, c, b = state

```

```

if action == 'MC->':
    return (m - 1, c - 1, 0)
elif action == 'MM->':
    return (m-2, c, 0)
elif action == 'CC->':
    return (m, c-2, 0)
elif action == 'M->':
    return (m-1, c, 0)
elif action == 'C->':
    return (m, c-1, 0)
elif action == '<-MC':
    return (m+1, c+1, 1)
elif action == '<-MM':
    return (m+2, c, 1)
elif action == '<-CC':
    return (m, c+2, 1)
elif action == '<-M':
    return (m+1, c, 1)
elif action == '<-C':
    return (m, c+1, 1)
else:
    print("ERROR!!! ")
    return None

```

Let's analyze the code above:

- a possible doubt could be if makes sense to move the boat without people; it could be possible, this is not the unique way to solve.
- **elif action == '<-MC':** \longleftrightarrow if we move from the second shore to the first one we're adding people to the initial shore, so the state which take into account the initial shore will see an increment; furthermore we have 1 to indicate that the boat has returned.

We implemented the *actions* and the *results*. Now we have to implement the *goal* and the *heuristic* (*h* function); then we can run our script and see the solutions.

```

def goal_test(self, state):
    """Return True if the state is a goal.
    The default method compares the
    state to self.goal or checks for state
    in self.goal if it is a list, as specified
    in the constructor. Override this method if
    checking against a single self.goal
    is not enough."""
    if isinstance(self.goal, list):
        return is_in(state, self.goal)
    else:
        return state == self.goal

def h(self, node):
    m, c, b = node.state
    return m + c - b

```

Let's analyze the code above.

- node \longleftrightarrow is an instance of **class Node?** It should be.
- **h(self, node):** \longleftrightarrow the idea is that our node state is composed by m, c ,
b. We consider for example river crossing by using $m + c - b$ (distance from the goal); this is a possibility but is not the only one.
- Then we can implement also the heuristic function for using for example the A* search strategy that we will print in the **reporting**.

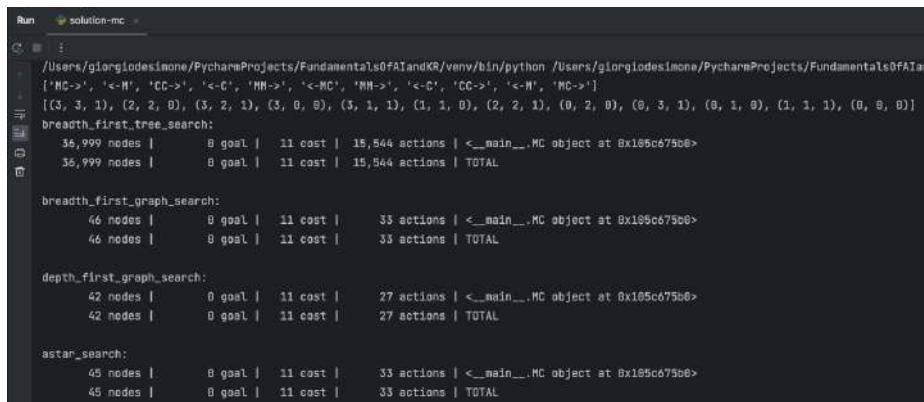
Finally, this is the script implementing the solution.

```
mc1 = MC((3, 3, 1), (0, 0, 0))
soln = breadth_first_tree_search(mc1)
# print("Done!!!")
path = path_actions(soln)
print(path)
path = path_states(soln)
print(path)

report([
    breadth_first_tree_search,
    breadth_first_graph_search,
    # depth_first_tree_search,
    depth_first_graph_search,
    astar_search
],
[mc1])
```

Let's analyze the code above:

- with AIMA library we can define these variables for one single strategy and then we can compare the result for all the strategies we list in the **report**;
- in the **report** we commented the **depth_first_tree_search** because we can see clearly a loop.



```
/Users/giorgiodesimone/PycharmProjects/FundamentalsOfFAIandKR/venv/bin/python /Users/giorgiodesimone/PycharmProjects/FundamentalsOfFAIandKR/solution-mc.py
[...]
breadth_first_tree_search:
  36,999 nodes |     0 goal |   11 cost |  15,544 actions | <__main__.MC object at 0x105c675b0>
  36,999 nodes |     0 goal |   11 cost |  15,544 actions | TOTAL

breadth_first_graph_search:
  46 nodes |      0 goal |    11 cost |      33 actions | <__main__.MC object at 0x105c675b0>
  46 nodes |      0 goal |    11 cost |      33 actions | TOTAL

depth_first_graph_search:
  42 nodes |      0 goal |    11 cost |      27 actions | <__main__.MC object at 0x105c675b0>
  42 nodes |      0 goal |    11 cost |      27 actions | TOTAL

astar_search:
  45 nodes |      0 goal |    11 cost |      33 actions | <__main__.MC object at 0x105c675b0>
  45 nodes |      0 goal |    11 cost |      33 actions | TOTAL
```

Figure A.7

Let's analyze the output above.

- Focusing on the first two lines of the output we can see the lists of actions and states printed. We have that we start with the state $(3, 3, 1)$ on which it is applied the action '**MC->**' which leads to the state $(2, 2, 0)$; then from the opposite shore we move only one missionary '**<-M**' and we obtain $(3, 2, 1)$ and so on.
- It is also interesting to see the computational results of each search strategy.
 - **breadth_first_tree_search**: great number of nodes explored, the cost is simply that of the **default** method which is 1 for each step (we have 11 steps in the **path_actions**) and we have a considerable number of actions.
 - **breadth_first_graph_search**: if we use *graph* we have no repetition because consider a list of visited nodes, so we decrease the number of nodes (we are more efficient) and also the actions.
 - **depth_first_tree_search**: we commented it because we have an infinite loop.
 - **depth_first_graph_search**: is more efficient than the corresponding **breadth_first** in the number of nodes and actions.
 - **astar_search**: we will see the debug of this search strategy ahead.
- Note also that in the output we have the expression **0 goal** because we are not considering a list of goals but only one; if, for example, we give a list of 6 goals in the output we can have **3 goal** meaning that with the strategy we reach 3 out of 6 goals.

```
[MC->, <-M, CC->, <-C, MM->, <-MC, MM->, <-C, CC->, <-M, MC->]
[(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]
breadth_first_tree_search:
1
<Node (3, 3, 1)>
dequeue([<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>])
2
<Node (2, 2, 0)>
dequeue([<Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>, <Node (3, 3, 1)>, <Node (3, 2, 1)>, <Node (2, 3, 1)>])
3
<Node (1, 3, 0)>
dequeue([<Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>, <Node (3, 3, 1)>, <Node (3, 2, 1)>, <Node (2, 3, 1)>])
4
<Node (3, 1, 0)>
dequeue([<Node (2, 3, 0)>, <Node (3, 2, 0)>, <Node (3, 3, 1)>, <Node (3, 2, 1)>, <Node (2, 3, 1)>, <Node (3, 3, 1)>])
5
<Node (2, 3, 0)>
35,999 nodes | 0 goal | 11 cost | 15,544 actions | <__main__.MC object at 0x10a067d10>
35,999 nodes | 0 goal | 11 cost | 15,544 actions | TOTAL
```

Figure A.8

In the output above we presented the first five steps of the **breadth_first_tree_search** (by printing them): we can see that you expand always the *less deep node*, so in this case the first on the left of the list, so the first step is from $(3, 3, 1)$ you go to the first of the list which is $(2, 2, 0)$, then you go to the first of the list of the second step which is $(1, 3, 0)$... but the problem in the tree search is that we can have repetition (in the image above we highlighted the repetitions, we returned to the initial state). So it is possible to create an *infinite loop*; in this case with **breadth_first_tree_search** we don't have an infinite loop but with **depth_first_tree_search** we have it, and so we commented it.

One idea to avoid this is to use the **graph_search**.

```

breadth_first_graph_search:
1
<Node (3, 3, 1)>
deque([<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>])
2
<Node (2, 2, 0)>
deque([<Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>, <Node (2, 3, 1)>, <Node (2, 3, 1)>])
3
<Node (1, 3, 0)>
deque([<Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>, <Node (3, 2, 1)>, <Node (2, 3, 1)>])
4
<Node (3, 1, 0)>
deque([<Node (2, 3, 0)>, <Node (3, 2, 0)>, <Node (3, 2, 1)>, <Node (2, 3, 1)>])
5
<Node (2, 3, 0)>
46 nodes |      0 goal |    11 cost |    33 actions | <__main__.MC object at 0x10a067d10>
46 nodes |      0 goal |    11 cost |    33 actions | TOTAL

```

Figure A.9

Seeing the above debugging of the first five steps of the `breadth_first_graph_search` we can notice that we don't have repetition.¹⁸

```

depth_first_graph_search:
1
<Node (3, 3, 1)>
[<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>]
2
<Node (3, 2, 0)>
[<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>]
3
<Node (2, 3, 0)>
[<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>]
4
<Node (3, 1, 0)>
[<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 2, 1)>]
5
<Node (3, 2, 1)>
42 nodes |      0 goal |    11 cost |    27 actions | <__main__.MC object at 0x10a067d10>
42 nodes |      0 goal |    11 cost |    27 actions | TOTAL

```

Figure A.10

The above output is the same of the previous in the sense that we don't have repetitions (`graph_search`), but being a `depth_first`. This time the idea is that you always expand the *last node i.e.* the deepest node of the tree, the last node of the list $(3, 3, 1) \rightarrow (3, 2, 0) \rightarrow (2, 3, 0)$ and so on.

```

astar_search:
1
<Node (3, 3, 1)>
[(5, <Node (1, 3, 0)>), (5, <Node (2, 2, 0)>), (5, <Node (3, 1, 0)>), (6, <Node (2, 3, 0)>), (6, <Node (3, 2, 0)>)]
2
<Node (1, 3, 0)>
[(5, <Node (2, 2, 0)>), (6, <Node (2, 3, 0)>), (5, <Node (3, 1, 0)>), (6, <Node (3, 2, 0)>)]
3
<Node (2, 2, 0)>
[(5, <Node (3, 1, 0)>), (6, <Node (2, 3, 0)>), (6, <Node (3, 2, 0)>), (6, <Node (3, 2, 1)>)] (6, <Node (2, 3, 1)>)
4
<Node (3, 1, 0)>
[(6, <Node (2, 3, 0)>), (6, <Node (2, 3, 1)>), (6, <Node (3, 2, 0)>), (6, <Node (3, 2, 1)>)]
5
<Node (2, 3, 0)>
45 nodes |      0 goal |    11 cost |    33 actions | <__main__.MC object at 0x10a067d10>
45 nodes |      0 goal |    11 cost |    33 actions | TOTAL

```

Figure A.11

Above in the A* output we have that each node is associated to a cost. Remember that we have the $f(n)$ function composed by the `(path_cost, h)`, specifically:

$$f(n) = n.\text{path_cost} + h(n)$$

¹⁸To see in general it is enough to add a `print()` function in your code.

Where:

- $n.\text{path_cost} \longleftrightarrow$ the default method costs 1 for every step in the path;
- $h(n) \longleftrightarrow m + c - b$.

For example, seeing the first row of the debugging output above, from the first step (`<Node (3, 3, 1)>`) we chose (`<Node (1, 3, 0)>`) (h is 4, furthermore from (`<Node (3, 3, 1)>`) we can reach this state with 1 step starting from the root, so $4 + 1 = 5$).

Furthermore, if we focus on the part of the output highlighted in blue above, for example, we are not able to reach this state with only 1 step starting from the root (simply because in $(3, 3, 1)$ we are in the initial shore [we have 1 as the third entry of the tuple] and also in $(3, 2, 1)$), so the only possibility is that we make 2 steps (initial shore \rightarrow second shore \rightarrow initial shore) [we have moved 1 cannibal, $3 + 2 - 1 + 2 = 6$].

```

solution-mc: ~
:
:
/Users/giorgiodesimone/PycharmProjects/FundamentalsOfAIandKR/venv/bin/python /Users/giorgiodesimone/PycharmProjects/FundamentalsOfAIandKR/solution-mc.py

[<MC->, '<-M', '<-CC->', '<-C', '<-MM->', '<-MC', '<-MM->', '<-C', '<-CC->', '<-M', '<-MC->']
[(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

breadth_first_tree_search:
    36,999 nodes |     0 goal |   11 cost | 15,544 actions | <__main__.MC object at 0x107bbb5b0>
    36,999 nodes |     0 goal |   11 cost | 15,544 actions | TOTAL

breadth_first_graph_search:
    46 nodes |     0 goal |   11 cost |      33 actions | <__main__.MC object at 0x107bbb5b0>
    46 nodes |     0 goal |   11 cost |      33 actions | TOTAL

depth_first_tree_search:

```

Figure A.12

Then (see above) if we try to run also the `depth_first_tree_search` without the initial comment # we can see that we crash our code when we start the `depth_first_tree_search` because we have an infinite loop.

```

depth_first_tree_search:
1
<Node (3, 3, 1)>
|<Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>|
2
<Node (3, 2, 0)>
|<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 3, 1)>|
3
<Node (3, 3, 1)>
|<Node (3, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 2, 0)>|
4
<Node (3, 2, 0)>
|<Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (2, 2, 0)>, <Node (1, 3, 0)>, <Node (3, 1, 0)>, <Node (2, 3, 0)>, <Node (3, 3, 1)>|
5
<Node (3, 3, 1)>

```

Figure A.13

In the `depth_first_tree_search` we chose always the last node, then... (in the `tree_search` we can have repetition) we can avoid this situation (see above) considering the `graph_search`.

Here ends the discussion on Missionaries and Cannibals problem. In the next section we quickly start a second problem.

A second problem: The U2 Bride

In the following problem we have constraints on time.

- U2 (4 components) are giving a concert in Dublin.
- There are still 17 minutes. Unfortunately, to reach the stage, the members of the band must cross a small, dark, and dangerous bridge... do not despair! They have a torch!!! (Only one).
- The bridge allows the passing of two persons at a time. The torch is mandatory to cross the bridge, and should be brought back and forth (cannot be thrown). All the members are on the wrong side of the bridge, far from the stage.
- Every member of the U2 walks at a different velocity, and they takes different time to cross the bridge.
 - Bono, 1 minute.
 - Edge, 2 minutes.
 - Adam, 5 minutes.
 - Larry, 10 minutes.
- If two members cross the bridge together, it will take them the highest time to cross it (*i.e.* the faster will walk at the velocity of the slower).
- For example: if Bono and Larry will cross together, it will take them 10 minutes to get over the bridge. If Larry brings back the torch, another 10 minutes will pass, and the mission will be failed.
- Think about an heuristic function for this problem, and try to solve it using A* over graphs. To limit the search space, suppose that the members move always in couple in one direction, and only one member brings back the torch.

This is a more complex problem (in comparison to the previous, because we have constraints also in the time for each member).

- How to represent the state?
- Choice: state as a list of lists, indicating who is in which shore, and the presence of the torch.
 - Initial state is [["Bono", "Edge", "Adam", "Larry"], [], 1], where:
 - * ["Bono", "Edge", "Adam", "Larry"] \longleftrightarrow initial shore;
 - * [] \longleftrightarrow second shore;
 - * 1 \longleftrightarrow torch is present in the initial shore.
 - Goal state will be [[], ["Bono", "Edge", "Adam", "Larry"], 0]

Consider the following words as suggestions, we are not going to discuss this exercise in this laboratory session, we just use it to give an example.

A suggestion for the heuristic function.

- At most, only two persons can move at a time. Let us group the members in couples. Sort the member in descending order on the base of the crossing time, and:
 - put in the first group the two slower members;
 - put in the second group the remaining members.
- Every group will move at the velocity of the slowest in the group. The heuristic function is the sum of the time required by each group still in the wrong side of the bridge.
- Is this heuristic function admissible? Will it find the best solution?

In any case you can find the implemented solution in the script `solution-u2.py` in my repository <https://github.com/giodesi/FundamentalsOfAI.git>

```
Cost: 17
[(['Edge', 'Bono'], 'Bono', ('Larry', 'Adam'), 'Edge', ('Edge', 'Bono'))
 [((('Larry', 'Adam', 'Edge', 'Bono'), (), 1), ((('Larry', 'Adam'), ('Edge', 'Bono')), 0),
   (('Larry', 'Adam', 'Bono'), ('Edge', ), 1), ((('Bono', ), ('Larry', 'Adam', 'Edge')), 0),
   (('Edge', 'Bono'), ('Larry', 'Adam'), 1), ((), ('Larry', 'Adam', 'Edge', 'Bono'), 0))]
```

Figure A.14

A third problem: Fill a square

We give just another exercise that you can try to solve.

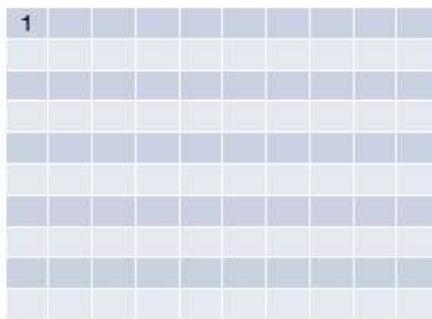


Figure A.15

- A matrix of 10×10 cells is given.
- In the initial state, all the cells are empty, except the left upper corner, that is initialized to 1.

- Problem: assign a consecutive value to all the cells, starting from 1 up to 100, with the following rules.
 - Starting from a cell with value x , you can assign value $(x + 1)$ to:
 - * empty cells that are two cells away in vertical or horizontal direction, or;
 - * one cell away in diagonal direction.
- If the matrix is empty, a cell has 7 possible “next cells” to be filled branching factor.
- When the matrix is partially filled, the number of free cell that are reachable decreases —> branching factor lowers.
- The depth of the search tree is 100...

Appendix B

Agent-based simulation

Introduction to Netlogo

Netlogo is a simple tool, we need to understand its proper programming language (a little bit different in comparison to other languages), but the tool itself is very simple having a very user friendly graphical interface.

So, a very important feature of *Netlogo* is that with a simple user interface you are able to define very complex systems.

Modeling complex systems

Programmable modeling environment for simulating natural and social phenomena:

- well suited for modeling complex systems evolving over time;
- hundreds or thousands of independent agents operating concurrently;
- exploring the connection between the micro-level behavior of individuals¹ and the macro-level patterns that emerge from the interaction of many individuals.

Easy-to-use application development environment:²

- creating custom models and quickly testing hypotheses about self-organized systems;³
- simple scripting language;⁴
- user-friendly graphical interface;

¹We can define very simple behaviour of a single agent; for example in Ant Colony a behaviour of a single ant could be: move to the right in order to search the nearest source of food.

²Wilensky, U. (1999). NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

³You can create different simulations and you are also able to define different parameters of your simulation because you can define a *slider*, for example, from 0 to 100 and if you relate this slider with a variable in your code you are able to change on the fly the value of this parameter to run different simulations; so you can also plot different values on your simulation and calculate average value and so on.

⁴Not really true because we have to understand some basic rules of the language.

- runs on JVM.⁵

Practical info

Download link:

- <http://ccl.northwestern.edu/netlogo/download.shtml>
- launch Netlogo through command line:⁶
 - `/{netlogo_download_folder}/netlogo.sh`

Online doc:⁷

- <http://ccl.northwestern.edu/netlogo/docs/>

Book for agent-based modeling (special focus on Netlogo):

- <http://www.intro-to-abm.com/>

History snapshot

Logo (Papert & Minsky, 1967):⁸

- theory of education based on Piaget's constructionism ('hands-on' creation and test of concepts);
- simple language derived from LISP;
- turtle graphics and exploration of 'micro-worlds'.

StarLogo (Resnick, 1991), MacStarLogo, StarLogoT:⁹

- agent-based simulation language.

NetLogo (Wilensky, 1999):

- further extending StarLogo (continuous turtle coordinates, cross-platform, networking, etc.).¹⁰

⁵JVM stands for *Java Virtual Machine*. We need to use *Netlogo 5.0.2* version that is able to run the code under the examples that we propose (2 files with the extension *.nlogo*); different versions of *Netlogo* differs a lot, we will see.

⁶This is the command for the version 5.0.2; for newest version is a little bit different.

⁷Netlogo is full of built-in functions and variables and if you read and learn this built-in items you are able to write in a single line of code very different and complex command for the simulation.

⁸First version.

⁹The problem is the portability, now with *Netlogo* we are able to run it through the JVM on the operating system.

¹⁰For the newest version there is also the possibility to import other languages (this feature is not valid for *Netlogo 5.0.2*).

The world of Netlogo

Netlogo is a 2-D world made of 4 kinds of agents.¹¹

- *Patches*: make up the background or ‘landscape’.¹²
- *Turtles*: move around on top of the patches.¹³
- *Links*: connect two turtles.¹⁴
- *The Observer*: oversees everything going on in the world.¹⁵

Graphical Interface - Controls

Controls allow to run and manage the flow of execution.

- Buttons:¹⁶ initialize, start, stop, step through the model:

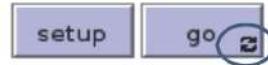


Figure B.1

- ‘Once’ button execute one action;
- ‘Forever’ button repeat the same action until pressed again.

- Functions with the name of the buttons specify the action executed on click.
- Command center:¹⁷ ask agents to execute specific commands ‘on the fly’.¹⁸

¹¹The following represent the general structure of Netlogo. In the graphical interface we will be able to see different pannels and on the right you can see a window of the world and in this window we can see the evolution of our simulation and all the agents that move in our world.

¹²The idea is that we have the background composed by agents called *patches*, so in Netlogo the background is composed by agents.

¹³On the top of patches we have turtles *i.e.* the agents that are able to move on the patches. So when we define the coordinates of turtles we can define absolute coordinates in the world but also relative coordinates based on the patches.

¹⁴Agents that are connections between 2 turtles.

¹⁵At the *top level* we have the *Observer* that is the most important agent; in most cases the *Observer* coincides with the user itself because the *Observer* is the only that is able to write commands on the fly. For example, if you want to change color of an agent we can write the command, run the simulation and the agent that we defined with coordinates or with an id changes its color.

¹⁶We can add in a very simple way the button to the G.I. (with the ‘+’ on the left). Most used buttons are the *setup* and *go* button. The *setup* is often related to a procedure or a function defined in your code, with the *setup* in general we set our environment: so, for example, we initialize our variables, we create our agents, we set initial coordinates for the agents and so on. With the *go* button we start the simulation; the *go* button can be ‘once’ *go* button (single press) or a ‘forever’ *go* button; for the ‘forever’ button we have to stop the simulation by pressing again the button. So, for example, if we want to see a simulation only for 10 seconds we have to stop the simulation if we have a ‘forever’ button.

¹⁷The command center is the command line.

¹⁸If, for example, we are the *Observer*.

Graphical Interface - Settings

Settings allow to modify parameters.

- Sliders: adjust a quantity from *min* to *max* by an increment.¹⁹

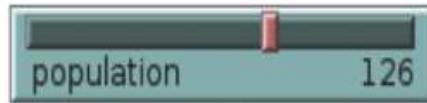


Figure B.2

— population = 126

- Switches: set a Boolean variable.²⁰



Figure B.3

— incentivi installazione? = false

- Choosers: set a value from a list.²¹

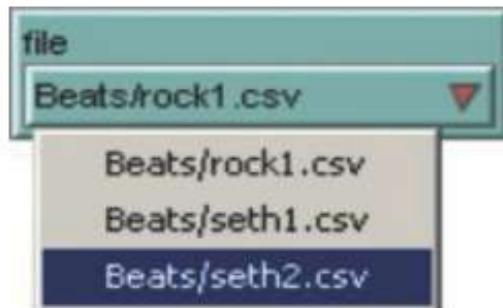


Figure B.4

— file = "Beats/seth2.csv"

¹⁹Usually is related to the population of our simulation. We associate the slider to a variable in our code (in this case `population`) and we can change this parameter also for parameter tuning.

²⁰In this case if you want to see the simulation with the possibility of `incentivi installazione?` for all the population you can set this variable to `ON`, otherwise to `OFF` and you can see different emerging behaviour.

²¹Good, for example, if you want to import data (*e.g.* from a .csv file) you can use ‘Choose’. But also if you want to write data on a file, to export data.

Graphical Interface - Views

Views allow to display information.

- Monitors display the current value of variables over time.



Figure B.5

- Plots display the history of a variable's value.²²

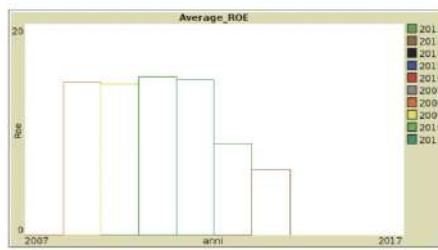


Figure B.6

- Output text areas, log text info.
- Graphic window, the main view of the 2-D Netlogo world.²³

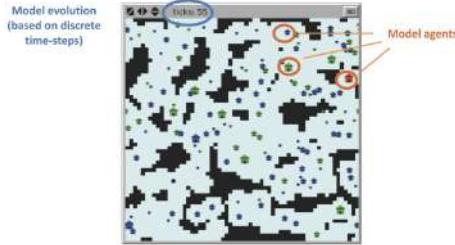


Figure B.7

²²You can plot the final result of your simulation but also the evolution of your simulation. We will see in the *Ant Colony* that we are able to plot the *evolution* of the food consumption with three lines of plots. And if you are able to see evolution with plot and/or monitors you can decide when stop the simulation with the ‘forever’ button.

²³Usually found on the right of our graphical interface. At the beginning is a total black window, but all the black pixel in the window are **patches** so are agents; you can also see that **patches** can change color (like blue or black) based on different values of the variables related to the simulation. You can see also the **turtles** also here we can change agent color on the base of values related to a variable, for example, `photovoltaic_adoption`; not only colors, we can change also the *dimension* of our agents with some logic. You can see also the variable related to the evolution of the model that is the **ticks**. The image is referred to a particular **tick** of the evolution of the simulation. Generally the initial **tick** is not initialized in Netlogo world, so you have to set the initial **tick** to 0 generally but also as you want with the **setup** button.

Programming Concepts - Agents

Agents²⁴ carry out their activity, all simultaneously.

- *Patches* don't move, form a 2D wrap-around grid, have *integer* coordinates (*pxcor*, *pycor*).²⁵
- *Turtles* move²⁶ on top of patches (not necessarily in their centre), have *decimal* coordinates (*xcor*, *ycor*)²⁷ and orientation (heading).
- *Observer* can create new turtles, can have read/write access to all the agents and variables.

Programming Concepts - Procedures & Functions

As in other programming we can define *procedures* or *functions*.

Commands (**to** keyword):²⁸

- action for the agents to carry out ('void' functions);
- example with 2 input arguments:²⁹

```
to draw-polygon [num-sides size]
  pd ;; pen down, draw
  repeat num-sides
    [fd size ;; forward 'size' steps
     rt (360 / num-sides)] ;;
  end
```

Reporters (**to-report** keyword):³⁰

- report a result value;
- example with 1 input argument:³¹

²⁴In Netlogo you can also define features and properties for all the agents based on the type of agent that you want to consider.

²⁵Coordinates have to be integers (**int**, **int**) otherwise you have an error in the command line.

²⁶Are able to move.

²⁷Continuous coordinates; note that there is no 'p'.

²⁸So in this slide we can see a '*Procedure*', the keyword is **to**, '*Procedures*' are 'void' functions.

²⁹In Netlogo we have a lot of built-in commands that you can use (without having to define always yourself). So in this case for the procedure **draw-polygon** you have to define input parameters that are **num-sides** and **size**. Then you can use the **pd** built-in command that allows you to put down a pen and draw: the notation **;;** is for comments. The **repeat** keyword is for *loops*: in this case we want to repeat the action for **num-sides** of times. Then you can write the block of your instructions inside the **[]** block in which we have the *forward* (**fd**) and *rotate* (**rt**) commands which as parameter has the degree you want to rotate your pen. Remember the **end** keyword for each procedure or function.

³⁰What we saw refers to 'procedure'; but you can also define 'functions'; the difference is the keyword **to-report** and not only **to**.

³¹For example if you want to define an **absolute-value** function. Here we can see the **if-else** implementation: we have **number >= 0** which is the condition; the first **[]** block executed if the condition is satisfied; the second **[]** block executed if the condition is not satisfied, so for the **else**.

```

to-report absolute-value [number]
  if-else number >= 0
    [report number]
    [report 0 - number]
  end

```

Primitives:

- *built-in* commands or reporters;³²
- some have an abbreviated form (`create-turtle` \longleftrightarrow `crt`).³³

Procedures:

- *custom* commands or reporters (user made).³⁴

Programming Concepts - Variables

Variables - places to store values.³⁵

- Global variables: only one value for the variable and every agent can access it.
- Turtle and Patch variables: each turtle/patch has its own value for every turtle/patch variable.
- Local variables: defined and accessible only inside a procedure (scope = narrowest square brackets or procedure itself).³⁶

Built-in variables:

- *E.g.* turtle variables: `color`, `xcor`, `year`, *etc.*
- *E.g.* patch variables: `pcolor`, `pxcor`, *etc.*

Custom variables:

- Defining global variables:³⁷
- ```
global [clock]
```
- Defining turtle/patch variables:<sup>38</sup>
- ```
turtles-own [energy speed]
patches-own [friction]
```

³²In the documentation you can find a lot of built-in commands, reporters (that are functions) or also variables (*e.g.* `pxcor`, `pycor`, `xcor`, `ycor`).

³³You can also have abbreviations, for example when you write the keyword `create` for agents you can also use the abbreviation `crt` (that are most used in the newest versions of Netlogo).

³⁴Obviously you can also define different functions or procedures based on your behaviour simulation.

³⁵In Netlogo we have variables based on the structure of Netlogo.

³⁶The Observer is able to see all the variables. We can have for example *local variables* for a particular agent, but this agent is able to see the variable of the agents that are under you and so on... this kind of logic.

³⁷In the square brackets the name of the variable; feature of Netlogo that is not present in modern languages like Python.

³⁸Keywords. You can define a list of turtles/patches variables.

- Defining local variables:³⁹

```
let variable value
```

– Creates a new local variable and gives it the desired value.⁴⁰

```
to swap-colors [t1 t2]
  let temp color-of t1
```

- Setting a variable value (after its definition):

```
set variable value
```

Programming Concepts - Ask

ask specify commands to be run by turtles or patches.⁴¹

- Asking all turtles:

```
ask turtles [ ... ]
```

- Asking all patches.

- Asking N turtles:⁴²

```
ask n-of N turtles [ ... ]
```

Observer code cannot be inside any **ask** block.⁴³

Programming Concepts - Variables

Setting variables.⁴⁴

- Setting the color of all turtles:

```
ask turtles [set color red]
```

- Setting the color of all patches:

```
ask patches [set pcolor red]
```

- Setting the color of the patches under the turtles:⁴⁵

```
ask turtles [set pcolor red]
```

³⁹Keyword **let**, the name **variable** of the variable, the value **value** of the variable.

⁴⁰Example to swap colors between 2 agents t1 and t2 you have to define a local variable **temp** and assigning it the color of t1 through the built-in command **color-of** that takes t1 and outputs its color. Read the documentation.

⁴¹It is a usefull command that you can use directly in you command center.

⁴²For asking to a subset of turtles we need to use the **n-of** keyword.

⁴³Because is the agent able to write an **ask** command. The commands inside the block here are only used at ‘turtles level’, ‘patches level’.

⁴⁴Very simple examples.

⁴⁵Note that this is a relative command: you can define the color to all the patches and you can merge it in the **ask** turtles, so you can ask turtles to set **pcolor** (that is referred to patches) red.

- Setting the color of one turtle (identify by `id`):

```
ask turtle 5 [set color green]
```

- Or:

```
set color-of turtle 5 green
```

- Setting the color of one patch (identified with coordinates):⁴⁶

```
ask patch 2 3 [set pcolor green]
```

Programming Concepts - Agent sets

Agent set, definition of a subset of agents (not a keyword).⁴⁷

- All blue turtles:

```
turtles with [color = blue]
```

- All blue turtles on the patch of the current caller (patch or turtle).⁴⁸

```
turtles-here with [color = blue]
```

- All turtles less than 5 patches away from the caller:

```
turtles in-radius 5
```

- The 4 patches to the east, north, west and south of the caller:

```
patches at-points [ [1 0] [0 1] [-1 0] [0 -1] ]
```

Using agent sets.

- Ask such agents to execute a command:

```
ask <agentset> [ ... ]
```

- Check if there are such agents:

```
show any? <agentset>
```

- Count such agents:

```
show count <agentset>
```

E.g. remove the richest turtle (with the maximum ‘assets’ value):⁴⁹

```
ask max-one-of turtles [sum assets] [die]
```

Where:

⁴⁶The world-center is in (0, 0).

⁴⁷We can define agent sets. Very important because you could have the necessity to ask something to a given set of agents (*e.g.* all turtles with color blue *etc.*).

⁴⁸`turtles-here` is a built-in command that does what the sentence to which we are referring describes; this is another relative command.

⁴⁹Example of a complex command in Netlogo.

- `max-one-of turtles [sum assets] ↔ agentset;`⁵⁰
- `[die]` ↔ command/action.⁵¹

Programming Concepts - Breeds

Breed, a ‘natural’ kind of agent set (other species than turtle):

```
breed [wolves sheep]
```

A new breed comes with automatically derived primitives:

```
create-<breed>
create-custom-<breed>
<breed>-here
<breed>-at
```

The breed is a turtle variable:⁵²

```
ask turtles 5 [if breed=sheep]
```

A turtle agent can change breed:

```
ask turtles 5 [set breed sheep]
```

Exercise 1 - Basic Ants Model

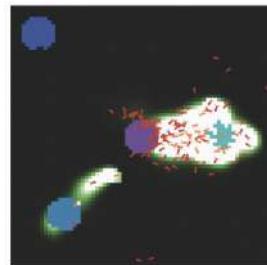


Figure B.8

Very simple model as a first ‘hands-on’ experience.

A colony of ants forages for food:

- Although each ant follows a set of simple rules,⁵³ the colony as a whole act in a sophisticated way.⁵⁴

⁵⁰The *max* among the turtles of the sum of assets that is a feature of the turtles in this particular problem; you sum all the assets of all the turtles and then the agentset is based on the turtle that has the maximum value of this sum.

⁵¹Then you can ask to this turtle to die.

⁵²If you define `breed [wolves sheep]`, breed that is wolves or sheep, you can define this type of agent set to all your turtles and then you can check if your turtle is breed 1 or 2 for example. In this case, for example, we’re asking to turtles 5 (5 is the *id*) if the breed of this turtle is equal to *sheep*.

⁵³Based on: move to the right, move to the left, follow the chemical trail if the value is greater than *n* etc.

⁵⁴The idea is that we want to study the emerging behaviour of a colony of ants for searching for food. We define a world with sources of foods, ants which are the simple local agents and you can see that the sources of food are different types of patches.

In the figure above is possible to see also the chemical trails left from the ants that are able to attract other ants of the colony in order to search the food, and also note the various colors present in the landscape; they are all patches that with colors represent the state of the simulation and change based on the evolution of the model. At the beginning you have basically all black patches plus the patches representing the full sources of food, then with the evolution you have to change for example the consumed parts of the food *etc.*⁵⁵

How to start

In order to start Netlogo 5.0.2⁵⁶ we need Java 8 installed.
Launch Netlogo through command line:

- /{netlogo_download_folder}/netlogo.sh

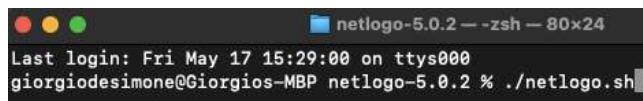


Figure B.9

In the following image is presented the initial *graphical interface*.

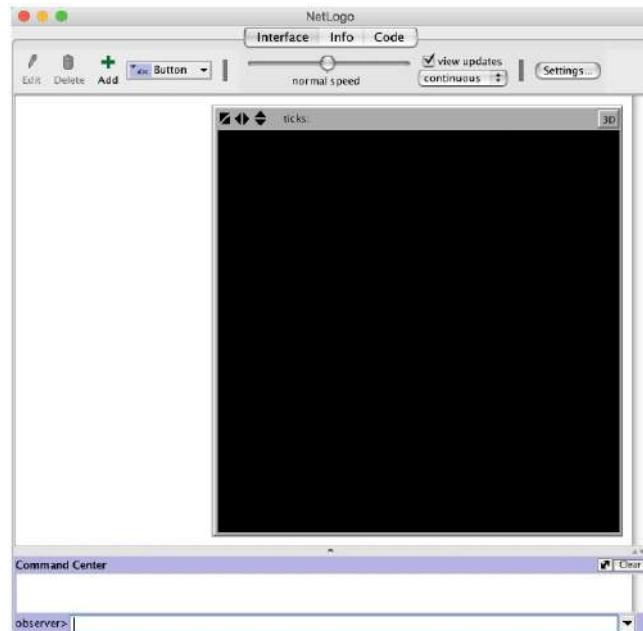


Figure B.10

At the top we can see three pannels:

⁵⁵Wilensky, U. (1997). Netlogo Ants model. <http://ccl.northwestern.edu/netlogo/models/ants> Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

⁵⁶You can download Netlogo 5.0.2 from <https://github.com/giodesi/NetLogo>

- **Interface** ↔ graphical interface;
- **Info** ↔ informations and documentations of the NetLogo version;
- **Code** ↔ this is probably the most important panel because is where you can see the code of your project; in this panel you can change all the relations between graphical elements and your code. In newest versions of Netlogo (e.g. 6.4.0) you are able also to add new panels (in this version is not possible).

At the left we can see the **Add** button:

- with this button you are able to add new graphical elements based on the list on the right of the button itself.

At the top center, just down the three panels described before you can set different speeds of your simulation with the slider **normal speed**.

Just down we can see the **ticks** that are not initialized.

Just down again, the black square representing the **World**.

Finally, at the bottom, down the **Command center**, there is the **observer**; generally we are the **observer**, we can write commands (e.g. `ask turtles set color blue`) and you can see the command executed in the simulation.

So, in our case, the next steps are:

1. open an existing model .nlogo;

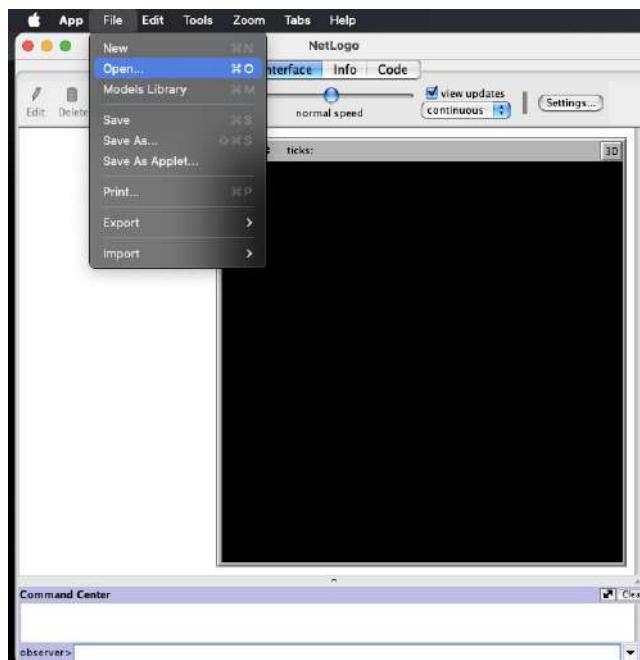


Figure B.11

2. choose the model of the first exercise i.e. `Es1_AntsModel.nlogo`⁵⁷

⁵⁷You can download it from <https://github.com/giodesi/NetLogo>

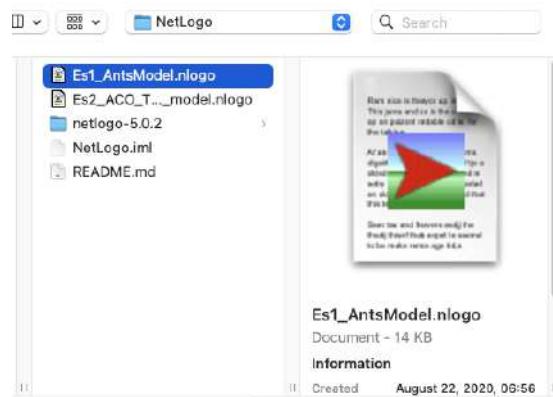


Figure B.12

Exercise 1 - Ants Model

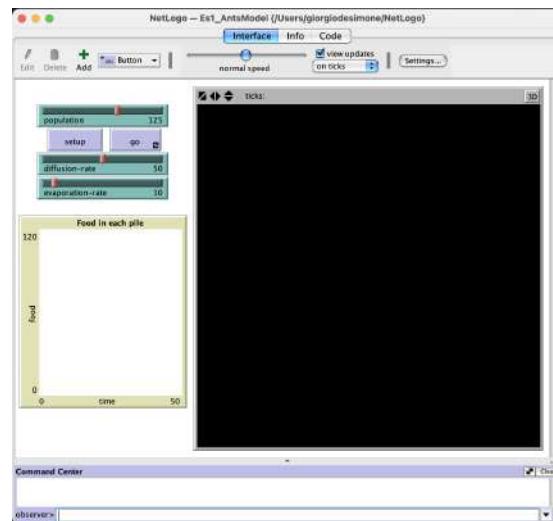


Figure B.13

Once opened the script `Es1_AntsModel.nlogo` we can see a more complex graphical interface arising (because in the *code* panel we have defined these graphical elements).

We can see.

- On the left a slider for the *population* that represents the number of ants.
- Below the *setup* and *go* buttons. With the *setup* button we are able to set our world, our agents, initialize the *ticks*, in general initialize all the variables. With the *go* ('forever') button we can wait till all the sources of food are consumed or we can stop during the execution of the simulation by pressing again the *go* button.

- Below the sliders `diffusion-rate` and `evaporation-rate`: we can set the parameters of the *diffusion rate* or the *evaporation rate* that are parameters that we define in our code; and these parameters are related to the *chemical trails* of the ants. You can modulate `diffusion-rate` and `evaporation-rate`.
- Below we can see a plot named `Food in each pile`: in this plot we are able to see the consumption of the three sources of food that we defined during the evolution of the system (basically we will see three different lines that are related to the sources of food that appear in our world once started the simulation).
- Finally, on the top, in the middle, we have the `view updates` option that is set `on-ticks`: so here the updates are based `on-ticks` but can be set also on `continuous` values.

Let's understand now the scope of this exercise:

- when an ant finds a piece of food, it carries the food back to the nest, dropping a chemical as it moves;
- when other ants 'sniff' the chemical, they follow the chemical toward the food;
- as more ants carry food to the nest, they reinforce the chemical trail.

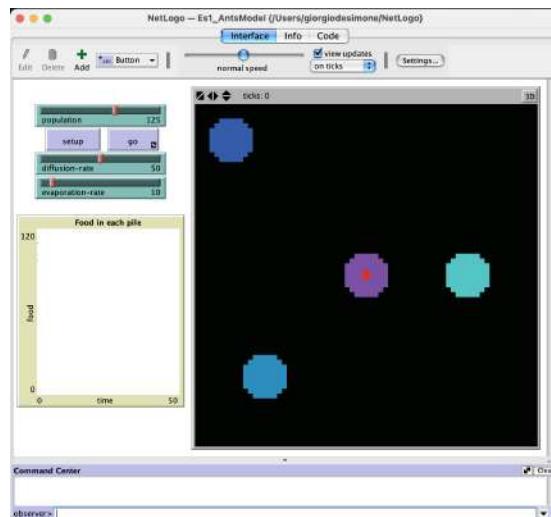


Figure B.14

Pressing the `setup` button – with the given initial parameters defined in the slider (they are initial parameters, you can change them) – you are able to see the emerging of graphical settings.

We have the violet nest at the center and three sources of food (colored circles) in different locations around.

Once we press the **go** button we can see ants exiting from the red circle inside the violet nest and exploring all around searching for food with the behaviour explained in the previous pages.

We can see that on the right of the nest, the light blue source of food is the nearest to the nest, so we expect that it is consumed first, then is consumed the second nearest, which is located in the bottom-left, then the third which is the blue one on the top-left; and this happens.

The previous description represents the *emerging general behaviour* that was not set by anyone; it simply emerges from the interaction of agents.

But attention, this is the emerging behaviour with these parameters set; changing them could change also the behaviour.

Exercise 1 - Model Usage

Usage.

- Click the **setup** button to set up the ant nest (in violet, at center) and three piles of food, then click the **go** button to start the simulation.
 - The chemical is shown in a green-to-white gradient.
- The **evaporation-rate** slider controls the evaporation rate of the chemical. The **diffusion-rate** slider controls the diffusion rate of the chemical.
- If you want to change the number of ants, move the **population** slider before pressing **setup**.

Exercise 1 - Code Analysis

Let's investigate the pieces of codes that compone the script.

```

NetLogo – Es1_AntsModel (/Users/giorgiodesimone/NetLogo)
Interface Info Code
Find... Check Procedures Indent automatically

patches-own [
  chemical      ; amount of chemical on this patch
  Food          ; amount of food on this patch (0, 1, or 2)
  nest?         ; true on nest patches, false elsewhere
  nest-scent    ; number that is higher closer to the nest
  food-source-number ; number (1, 2, or 3) to identify the food sources
]

;----;
;----;
;----;

to setup
  clear-all
  set-default-shape turtles "bug"
  crt population
  [ set size 2      ; easier to see
    set color red   ; red = not carrying food
  ]
  setup-patches
  reset-ticks
end

to setup-patches
  ask patches
  [ setup-nest
    setup-food
    recolor-patch
  ]
end

to setup-nest ; patch procedure
  ; set nest? variable to true inside the nest, false elsewhere
  set nest? (distancey 0 0) < 5
  ; spread a nest-scent over the whole world -- stronger near the nest
  set nest-scent 200 - distancexy 0 0
end

```

Figure B.15

Referring to the image above:

```
patches-own [
    chemical      ;; amount of chemical on this patch
    food          ;; amount of food on this patch (0,1,2)
    nest?         ;; true on nest patches, false elsewhere
    nest-scent    ;; number that is higher closer to the nest
    food-source-number ;; number (1,2,3) to identify the food sources
]
```

The first part is a definition of the **patches** variables; so with the keyword **patches-own** we define the variables.

```
; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;
;; Setup procedures ;;
; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;
```

Then we define the procedures related in this case to the **setup** button.

```
to setup
  clear-all
  set-default-shape turtles "bug"
  crt population
  [ set size 2          ;; easier to see
    set color red ]    ;; red = not carrying food
  setup-patches
  reset-ticks
end
```

The first **setup** is the list of functions and commands related to the setup button; the name is the same **setup** and if you define in the graphical interface the **setup** button, is automatic the relation with the **setup** procedure; then you can write built-in commands or custom functions or procedures:

- the first is **clear-all** which is a built-in command that is able to initialize the variables to 0, clear all the plots, clear all the world;
- then you can set the shape of your agents, in this case we use shape of ‘bug’ (built-in shape in Netlogo, but in newest versions you can import images and define new other shapes for the agents);
- then we create the population, so we define the properties for all the agents in our world: in the first instruction we set the size for each agent to 2 and in the second we set the color red; in this case **population** is related to the slider in the graphical interface, so if you set the slider to 1 you’re setting only 1 agent and so on;
- then we have the procedure **setup-patches** that is a custom procedure;
- then we have the **reset-ticks** that is the built-in command to reset for each **setup** the variable **ticks** to 0.

Note that in general we have here a sequential call for each command and procedure. So the order of the code is important in Netlogo.

```

to setup-patches
  ask patches
    [ setup-nest
      setup-food
      recolor-patch ]
end

```

`setup-patches` is a procedure that has other 3 procedures inside. Rememeber that patches are landscape agents, so in this case the patch can be: the nest, the food, or a green gradient color based for the chemical trail.

```

to setup-nest ;; patch procedure
  ;; set nest? variable to true inside the nest, false elsewhere
  set nest? (distancexy 0 0) < 5
  ;; spread a nest-scent over the whole world -- stronger near the nest
  set nest-scent 200 - distancexy 0 0
end

```

In the `setup-nest`:

- we use the boolean variable `nest?` giving it the boolean condition to be less than 5 distant from the center. So all the patches inside a circle of radius 5 are part of the nest;
- then we define the `nest-scent` that is the variable that we use to calculate the chemical trail that is stronger near the nest (see associated simple instruction); since the ants need to come back to the nest.

```

to setup-food ;; patch procedure
  ;; setup food source one on the right
  if (distancexy (0.6 * max-pxcor) 0) < 5
  [ set food-source-number 1 ]
  ;; setup food source two on the lower-left
  if (distancexy (-0.6 * max-pxcor) (-0.6 * max-pycor)) < 5
  [ set food-source-number 2 ]
  ;; setup food source three on the upper-left
  if (distancexy (-0.8 * max-pxcor) (0.8 * max-pycor)) < 5
  [ set food-source-number 3 ]
  ;; set "food" at sources to either 1 or 2, randomly
  if food-source-number > 0
  [ set food one-of [1 2] ]
end

to recolor-patch ;; patch procedure
  ;; give color to nest and food sources
  ifelse nest?
  [ set pcolor violet ]
  [ ifelse food > 0
    [ if food-source-number = 1 [ set pcolor cyan ]
      if food-source-number = 2 [ set pcolor sky ]
      if food-source-number = 3 [ set pcolor blue ] ]
    ;; scale color to show chemical concentration
    [ set pcolor scale-color green 0.1 5 ] ]
end

```

Figure B.16

```

to setup-food    ;; patch procedure
;; setup food source one on the right
if (distancexy (0.6 * max-pxcor) 0) < 5
[ set food-source-number 1 ]
;; setup food source two on the lower-left
if (distancexy (-0.6 * max-pxcor) (-0.6 * max-pycor)) < 5
[ set food-source-number 2 ]
;; setup food source three on the upper-left
if (distancexy (-0.8 * max-pxcor) (0.8 * max-pycor)) < 5
[ set food-source-number 3 ]
;; set "food" at sources to either 1 or 2, randomly
if food-source-number > 0
[ set food one-of [1 2] ]
end

```

We can distribute the numbers 1, 2, 3 to identify the different sources of food based on particular geometry which locate particular circles in the world landscape. Then, if the **food-source-number** is greater than zero we can set the patch as a source of food.

```

to recolor-patch    ;; patch procedure
;; give color to nest and food sources
ifelse nest?
[ set pc当地色 violet ]
[ ifelse food > 0
[ if food-source-number = 1 [ set pc当地色 cyan ]
  if food-source-number = 2 [ set pc当地色 sky ]
  if food-source-number = 3 [ set pc当地色 blue ] ]
;; scale color to show chemical concentration
[ set pc当地色 scale-color green chemical 0.1 5 ] ]
end

```

Then we can recolor patches:

- if is the nest, color it with violet;
- then we set other colors for food sources;
- at the end we can set the color **pc当地色** of our patches that are chemical trails based on the **scale-color** green and we define the parameters of this scale of colors (you can find more details for this command in the documentation).

So, **setup-nest**, **setup-food**, **recolor-patch** are procedures that we call in the **setup-patches** that we call in the **setup** procedure by pressing the **setup** button.

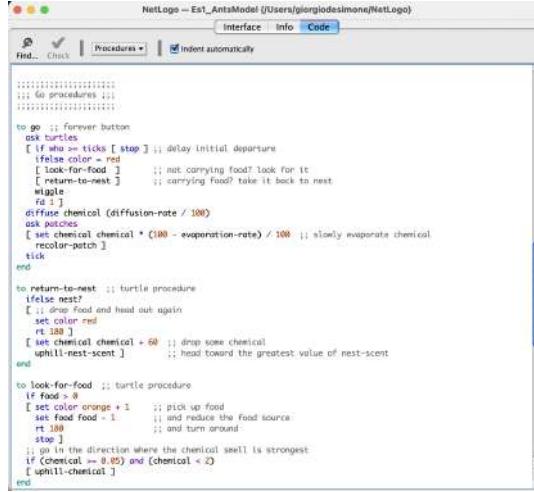


Figure B.17

```

to go ;; forever button
ask turtles
[ if who >= ticks [ stop ] ;; delay initial departure
  ifelse color = red
    [ look-for-food ] ;;; not carrying food? look for it
    [ return-to-nest ] ;;; carrying food? take it back to nest
    wiggle
    fd 1 ]
diffuse chemical (diffusion-rate / 100)
ask patches
[ set chemical chemical * (100 - evaporation-rate) / 100
;; slowly evaporate chemical
  recolor-patch ]
tick
end

```

The go procedure (started and ended by the forever button go):

- we have ants that at the beginning are red when we start the count of the ticks;
- then we have to consider 2 under procedures **look-for-food** in one direction and **return-to-nest** for the other;
- **wiggle** is a procedure to rotate ants;
- **fd** is to move agents;
- then we have to diffuse the chemical trail;
- at the end we have to ask patches to set the color of the chemical trail based on the position of the agents during the simulation; we can see that differently from the **nest-scent** (that is highest near to the nest) we set a slow evaporation for the chemical, evaporation rate that is relatively slow to try to reinforce the chemical trail for all the colony, for all the ants.

```

to return-to-nest  ;; turtle procedure
  ifelse nest?
    [ ;; drop food and head out again
      set color red
      rt 180 ]
    [ set chemical chemical + 60  ;; drop some chemical
      uphill-nest-scent ] ;; head toward the greatest value of nest-scent
  end

```

If we are in the nest set the color red and rotate of a certain degree, otherwise set the chemical to a particular value and the other function is to refresh the value of your gradient.

```

to look-for-food  ;; turtle procedure
  if food > 0
    [ set color orange + 1      ;; pick up food
      set food food - 1        ;; and reduce the food source
      rt 180                  ;; and turn around
      stop ]
    ;; go in the direction where the chemical smell is strongest
    if (chemical >= 0.05) and (chemical < 2)
      [ uphill-chemical ]
  end

```

If we are in the nest we have red ants, otherwise we have orange ants, we have essentially define this behaviour.

If `food > 0` so we are in a source of food we set color to orange and decrement food, rotate the ant and stop.

Then you have to change the value of the chemical and so call the procedure to update this new value.

```

NetLogo - Es1_AntsModel (Users/giorgiodesimone/NetLogo)
Interface Info Code

;; sniff left and right, and go where the strongest smell is
to uphill-chemical  ;; turtle procedure
  let scent-right chemical-scent-at-angle 0
  let scent-left chemical-scent-at-angle 45
  let scent-right nest-scent-at-angle 45
  let scent-left nest-scent-at-angle -45
  if (scent-right > scent-ahead) or (scent-left > scent-ahead)
    [ ifelse scent-right > scent-left
      [ rt 45 ]
      [ lt 45 ] ]
  end

;; sniff left and right, and go where the strongest smell is
to uphill-nest-scent  ;; turtle procedure
  let scent-right nest-scent-at-angle 0
  let scent-left nest-scent-at-angle 45
  let scent-right nest-scent-at-angle -45
  if (scent-right > scent-ahead) or (scent-left > scent-ahead)
    [ ifelse scent-right > scent-left
      [ rt 45 ]
      [ lt 45 ] ]
  end

to wobble  ;; turtle procedure
  rt random 40
  lt random 40
  if not com-move? 1 [ rt 180 ]
end

to-report nest-scent-at-angle [angle]
  let p patch-right-and-ahead angle 1
  if p = nobody [ report 0 ]
  report [nest-scent] of p
end

to-report chemical-scent-at-angle [angle]
  let p patch-right-and-ahead angle 1
  if p = nobody [ report 0 ]
  report [chemical] of p
end

```

Figure B.18

```

;; sniff left and right, and go where the strongest smell is
to uphill-chemical ;; turtle procedure
  let scent-ahead chemical-scent-at-angle 0
  let scent-right chemical-scent-at-angle 45
  let scent-left chemical-scent-at-angle -45
  if (scent-right > scent-ahead) or (scent-left > scent-ahead)
    [ ifelse scent-right > scent-left
      [ rt 45 ]
      [ lt 45 ] ]
  end

;; sniff left and right, and go where the strongest smell is
to uphill-nest-scent ;; turtle procedure
  let scent-ahead nest-scent-at-angle 0
  let scent-right nest-scent-at-angle 45
  let scent-left nest-scent-at-angle -45
  if (scent-right > scent-ahead) or (scent-left > scent-ahead)
    [ ifelse scent-right > scent-left
      [ rt 45 ]
      [ lt 45 ] ]
  end

```

You can also consider simple movements for the ants based on left and right rotation and go where the chemical is greater in comparison to other parts of the world. Very basic behaviour for all the ants.

```

to wiggle ;; turtle procedure
  rt random 40
  lt random 40
  if not can-move? 1 [ rt 180 ]
end

```

This is simply to rotate.

Finally the last pieces of code.

```

to-report nest-scent-at-angle [angle]
  let p patch-right-and-ahead angle 1
  if p = nobody [ report 0 ]
  report [nest-scent] of p
end

to-report chemical-scent-at-angle [angle]
  let p patch-right-and-ahead angle 1
  if p = nobody [ report 0 ]
  report [chemical] of p
end

```

Exercise 1 - Things to notice

Emerging (general) behaviour.⁵⁸

⁵⁸Modifying parameters and customizing the code you can change this behaviour.

- The ant colony generally exploits the food source in order, starting with the food closest to the nest, and finishing with the food most distant from the nest.
- It is more difficult for the ants to form a stable trail to the more distant food, since the chemical trail has more time to evaporate and diffuse before being reinforced.
- Once the colony finishes collecting the closest food, the chemical trail to that food naturally disappears, freeing up ants to help collect the other food sources:
 - the more distant food sources require a larger ‘critical number’ of ants to form a stable trail.
- The consumption of the food is shown in a plot:
 - the line colors in the plot match the colors of the food piles.

Exercise 1 - Model Extensions

Let's consider other possibilities.

1. Try different placements for the food sources:
 - (a) what happens if two food sources are equidistant from the nest?⁵⁹
2. In this project, the ants use a ‘trick’ to find their way back to the nest *i.e.* they follow the ‘nest scent’:
 - (a) real ants use a variety of different approaches to find their way back to the nest;
 - (b) try to implement some alternative strategies.
3. In the **uphill-chemical** procedure, the ant ‘follows the gradient’ of the chemical. That is, it ‘sniffs’ in three directions, then turns in the direction where the chemical is strongest:
 - (a) try variants of the **uphill-chemical** procedure, changing the number and placement of ‘ant sniffs’.

Exercise 2 - Ant Colony Optimization e TSP

Now we only introduce the TSP solution through the Ant Colony Optimization; you can find the solution in the Netlogo script named `Es2_ACO_TSP_model.nlogo`⁶⁰

- Goal: implementing the Ant System algorithm⁶¹ and use it to solve the Traveling Salesman Problem.⁶²

⁵⁹In this case, for example, you can try to define not only a random choice of the source of food but *e.g.* a choice based on a particular heuristic or function that you want to define.

⁶⁰You can download it from <https://github.com/giodesi/NetLogo>

⁶¹Dorigo, M., Maniezzo, V., and Colorni, A., The Ant System: Optimization by a colony of co-operating agents. IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics, Vol. 26, No. 1. (1996), pp. 29-41. <http://citeseer.ist.psu.edu/dorigo96ant.html>

⁶²This is a very common problem optimization and the idea is that you can relate ACO to TSP problem. In TSP there are nodes and arcs in a graph presentation *e.g.* we can relate nodes to cities. The TSP is a very complex problem, so just try to understand the logic behind; at this level it is not required to know how to solve this problem with Netlogo.

- Based on the observation of ants behaviour:
 - positive feed back based on pheromone tracks which reinforce the best solution components.

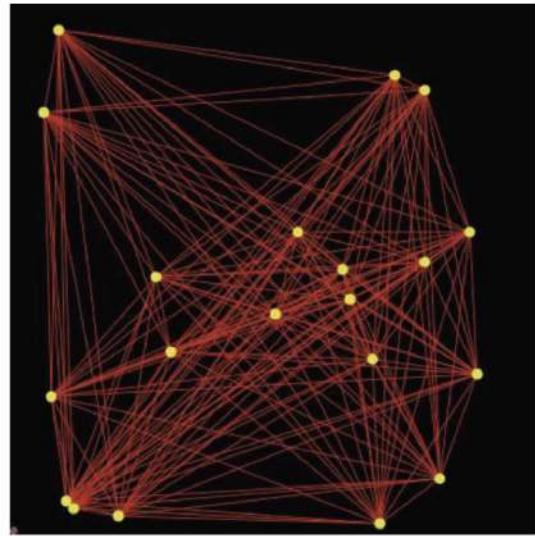


Figure B.19

Exercise 2 - Ant Colony Optimization

As you already know:

- ants leave a pheromone trail while going from the nest to food sources (and vice versa);
- ants tend to choose (with higher probability) routes with greater amount of pheromones;
- cooperative interaction which leads to an emergent behaviour, that is finding the shortest path.

Probabilistic model (pheromone model) used to recreate the pheromone trails left by ants.

Ants incrementally build the components of a solution.

Ants perform stochastic steps on a fully connected graph (construction graph). Constraints used to obtain a feasible solution.

Exercise 2 - ACO and TSP

A possible model for TSP:

- the graph nodes are the city to visit (the components of a solution);
- the edges are connections between the cities;

- a solution is a Hamiltonian circuit in the graph;
- constraints are used to avoid loops, so that an ant can visit a city exactly once.

Exercise 2 - Information sources

Edges or vertexes (or both) have two information:

- pheromone τ , which stands in for natural trail left by ants and represents the long term memory of ants in relation to the global search process;
- heuristic value η , *i.e.* the a priori knowledge on the problem.

Exercise 2 - ACO System

ACO System:

- the ants follow a path on the construction graph and build a solution;
- they used a transition (probabilistic) rule to choose the next node to visit;
- both pheromone and heuristic are taken into account;
- pheromone values are adjusted based on the quality of the solution found.

Appendix C

Graph-based Planning

Outline

Topic: planning based on graphs.

- Graphplan:
 - recall on graphic planning;
 - graphplan in action.
- PDDL modeling:¹
 - PDDL.
- SATPlan and Blackbox:²
 - planning as a satisfiability problem;
 - Blackbox in action.
- Fast Forward:³
 - Recalls plus FF in action.
- Exercises with PDDL.⁴

C.1 Graphplan

What is Graphplan?

Graphplan planners are based on STRIPS and the idea is that you're able to build really a planning graph based on an initial state, then you search in your space your graph solution and then you have to reach the final state which is the goal.

Main features.

¹It is the base of all planners that are based on graphplan.

²Other than the basic graph-plan we will see also SATPlan and Blackbox; Blackbox has different types of solvers inside, so you can choose the solver or the planner that you want to use in order to solve your problem.

³Fast Forward is based on heuristics, very efficient.

⁴PDDL is a common language for planning.

- Planner of the STRIPS type introduced by Blum and Furst (CMU) in 1995.
- Based on a data structure called planning graph.
- Planning as a search on the planning graph.
- It is an off-line planner.⁵
- It is complete and always produces the shortest possible plan.
- Produces partially ordered plans.
- The actions are represented as those of STRIPS with:
 - ADD LIST;
 - DELETE LIST;
 - PRECONDITIONS.
- A no-op action is implicitly defined that does not change the status.⁶
- State = objects + set of predicates.
- Objects have a type (typed)⁷

The structure of Graphplan is a planning graph, so you have a graph with nodes and arcs and you have to translate this representation in a correct notation for your problem.

More specifically.

- A planning graph is a direct level graph:
 - the nodes are grouped into levels;
 - the arcs connect nodes at adjacent levels.
- A planning graph alternates:
 - Proposition levels, containing proposition nodes;
 - Action levels, containing action nodes.
- Level 0 corresponds to the initial state (it is a proposition level).
- Arcs are divided into:
 - Precondition arcs (proposition → action);
 - Add arcs (action → proposition);
 - Delete arcs (action → proposition).

Below we can see a very basic but meaningful graphical example of the graphplan structure.

⁵It is based on the *Closed World Assumption*.

⁶Simply the action that from a state goes to the other state without any changes.

⁷Important feature of graphplan, we need to specify also the type of an object in graphplan: e.g. an object `home` has the type `place`.

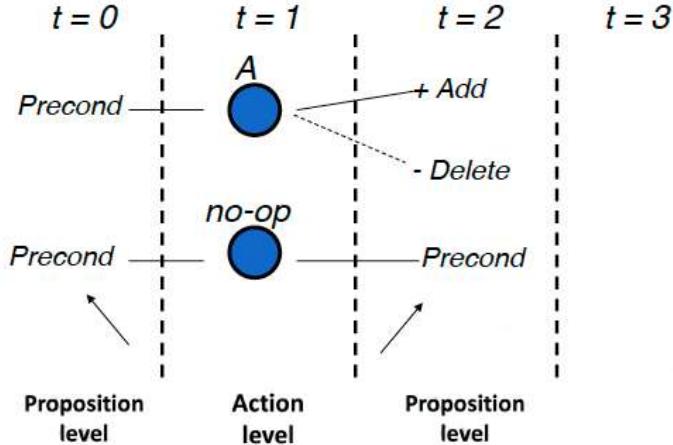


Figure C.1

Looking at the figure above focusing to the *Action level* remember that for each proposition you have also the same **no-op** operation that simply transport the proposition.

We highlight that the rules are simple but is very difficult to have a simple graphplan for problems that are not very very small, in the sense that we would reach a very big graph.

Continuing on listing the main key points.

- At a certain time step you can insert an action if its preconditions are present in the previous time step.
- **no-op** actions translate the propositions from one time step to the next.
- Each action level contains:
 - all actions that are applicable in that time step;
 - constraints that specify which pairs of actions cannot be performed simultaneously (inconsistencies or mutual exclusions).
- Each proposition level contains all literals that might result from any choice of actions in the previous time step including **no-op**.

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

- two actions can be inconsistent in the same time step;
- two propositions can be inconsistent at the same time step. In this case the actions/propositions are mutually exclusive.⁸

Main types of inconsistencies.

- *Inconsistent effects*: one action denies the effect of another.

⁸We will see that – while we need to identify these inconsistencies manually during the exam – the planner that we use in this section is able to identify automatically these inconsistencies.

- The action `move(start, dest)` has the effect `not at(start)` while the `no-op` action on `at(start)` has this effect.
- *Interference*: an action deletes a precondition of the other.
 - The action `move(start, dest)` has the effect `not at(start)` while the `no-op` action on `at(start)` has this as a precondition.
- *Competing needs*: two actions that have mutually exclusive preconditions.
 - The action `load(obj, means)` has as a precondition `not in(obj, means)` while the action `unload(obj, means)` has as a precondition `in(obj, means)`.⁹

Once the planning graph is built, we have to extract a *valid-plan*, i.e. a connected and consistent subgraph of the planning graph.

Features of a valid plan:

- actions in the same time step can be performed in any order (do not interfere);
- propositions at the same time step are not mutually exclusive;
- the last time step contains all the literals of the goal and these are not marked as mutually exclusive.

A classic example

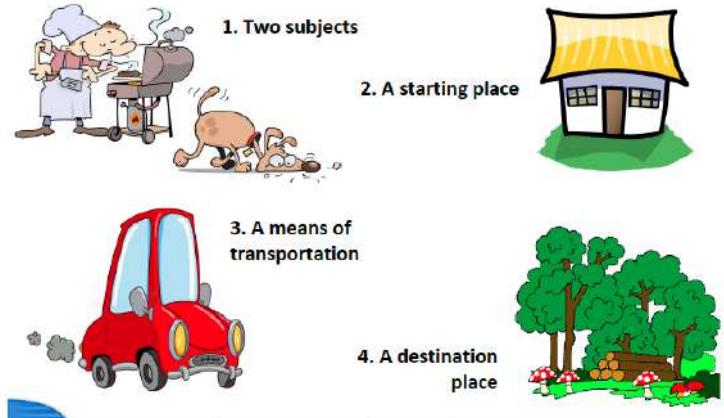


Figure C.2

So here we have:

1. two subjects: Bob and Jack;
2. a starting place: Home;
3. a means of transportation: Car;
4. a destination place: Mushrooms Wood.

⁹Obviously this referring that two actions are referring to the same objects `obj`.

Actions:

- MOVE (R , $PosA$, $PosB$):¹⁰
 - PRECONDITIONS: $at(R, PosA)$, $hasFuel(R)$.
 - ADD LIST: $at(R, PosB)$.
 - DELETE LIST: $at(R, PosA)$, $hasFuel(R)$.¹¹
- LOAD ($Object$, R , Pos):¹²
 - PRECONDITIONS: $at(R, Pos)$, $at(Object, Pos)$.
 - ADD LIST: $in(R, Object)$.
 - DELETE LIST: $at(Object, Pos)$.
- UNLOAD ($Object$, Pos):
 - PRECONDITIONS: $in(R, Object)$, $at(R, Pos)$.
 - ADD LIST: $at(Object, Pos)$.
 - DELETE LIST: $in(R, Object)$.

The idea is that you will have – for planning with automatic planners – the description of your problem but you have to translate this description in a notation that is useful for these planners, so in this case we will use the PDDL notation, so we have to translate these informations in PDDL notation.

Types, state, goals:

- *OBJECTS*:


```

cart: car (c)
% cart -> type
% c -> name of the variable
% car -> we could omit; we inserted just to
%         make clear that 'c' stands for car';
%         it is mandatory only to specify the
%         type and the name of the variable
objects: jack (j), bobby (t)
locations: home (h), mushrooms (m)
      
```
- *INITIAL STATE*:


```

at (j, h)
at (b, h)
at (c, h)
hasFuel (c)
      
```
- *GOAL*:


```

at (j, m)
at (b, m)
      
```

Some actions are omitted to make the graph more readable.

¹⁰Here R indicates a means of transportation.

¹¹We should insert the negation `not` before these actions but we simply list the actions because we are in the DELETE LIST.

¹²Load an object in a means of transportation in a particular position.

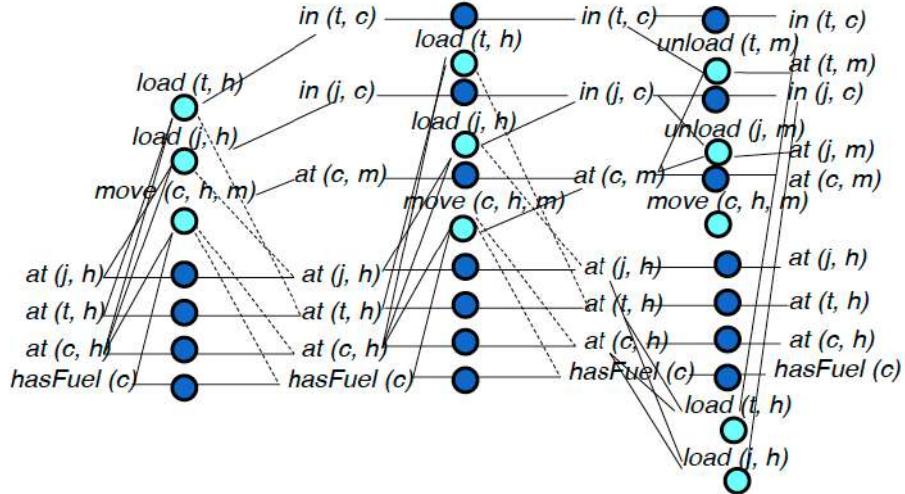


Figure C.3

Graphplan practical implementation

Starting a virtual machine or a local machine and supposing to have all installed and present on the machine.¹³

- Open the command line.
- Graphplan can run from the command line:¹⁴

```
info@info-VirtualBox:~$ graphplan -h
# or ./graphplan -h
# or sudo ./graphplan -h
# sudo for granting high permissions
```

- On your desktop¹⁵, you can find the *cart_example_gp* folder:

```
operator files: cart(gp).ops
Fact file: cart(gp).facts
```

As input files we need to give two types: a **.facts** file and a **.ops** file.

¹³If you run a Mac (Apple Silicon) follow the instruction on my repository <https://github.com/giodesi/Graphplan> to install graphplan compiling the code.

¹⁴If your local machine is a Mac set the PATH to the **.zshrc** user-specific configuration file which include environment variables, then you can launch the command from your user home directory or, as alternative, indicate always the full path in which is present the executable graphplan.

¹⁵Or in your local proper path.

In the `.facts` file you can find the description in pseudo-PDDL notation¹⁶ of the OBJECTS, INITIAL STATE and GOAL. For example the `cart.gp.facts`:

- OBJECTS:¹⁷

```
(home PLACE)
(mushrooms PLACE)
(car CART)
(jack CARGO)
(bobby CARGO)
```

- INITIAL STATE:¹⁸

```
(preconds
(at car home)
(at jack home)
(at bobby home)
(has-fuel car)
)
```

- GOAL:¹⁹

```
(effects
(at jack mushrooms)
(at bobby mushrooms)
)
```

In the `.ops` file you can find the description of the actions. For example the `cart.gp.ops`:

```
(operator
LOAD
(params
(<object> CARGO) (<cart> CART) (<place> PLACE))
(preconds
(at <cart> <place>) (at <object> <place>))
(effects
(on <object> <cart>) (del at <object> <place>)))

(operator
UNLOAD
(params
(<object> CARGO) (<cart> CART) (<place> PLACE))
(preconds
(at <cart> <place>) (on <object> <cart>))
(effects
```

¹⁶Truly the following is not the real PDDL notation, we will see the real PDDL after this example, because graphplan is based on a small different variant of PDDL notation.

¹⁷Below for example `home` indicates the name of the variable while `PLACE` indicates the type.

¹⁸For the initial state we have the keyword `preconds` and then we have to list the description of the initial state.

¹⁹For the goal we have the keyword `effects` and list the description of the goal.

```
(at <object> <place>) (del on <object> <cart>))

(operator
MOVE
(params
(<cart> CART) (<from> PLACE) (<to> PLACE))
(preconds
(has-fuel <cart>) (at <cart> <from>))
(effects
(at <cart> <to>) (del has-fuel <cart>) (del at <cart> <from>)))
```

Focusing on the first piece of code above notice the keyword **operator**, then we have the name of the action **LOAD**, then we have **params**, **preconds**, **effects** instead of **PRECONDITIONS**, **ADD LIST**, **DELETE LIST** seen before; finally notice that if we have a negative effect we have **del** which is the equivalent of the **not**.

Notice also that in the **<object> CARGO**, the **<>** brackets indicate the input of graphplan.

Let's implement the construction of the graph as exercise.

- Try to solve the problem with Graphplan.
- Gradually increase the level of output detail, using the **-i** option.
- Try to identify:
 - the graph gradually produced;
 - the lists of mutual exclusion.
- Try to disable the use of the mutual exclusions.

Construction of the graph...

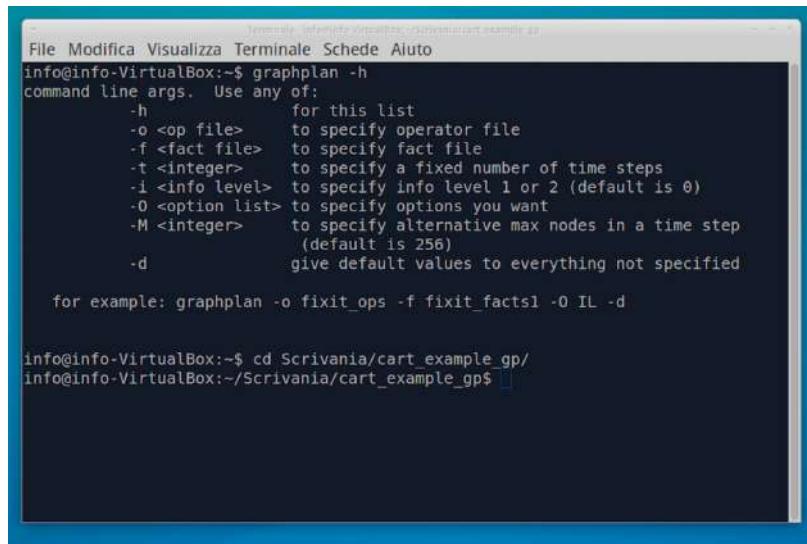


Figure C.4

In the figure below you can see that we use the following command to load the files:

```
graphplan -o cart.gp.ops -f cart.gp.facts
```

Then it is asked to choose the number of TIME STEPS that you want to use to find a valid plan; Here we can go on with the default option pressing ENTER or you can write a particular number of steps. If we use the minimum number of steps, so the default option, you can find the solution of this problem.

```
xubuntu1 [Running]
info@info-VirtualBox:~$ graphplan -h
File Modifica Visualizza Terminal Schede Auto
info@info-VirtualBox:~$ graphplan -h
command line args. Use any of:
  -h           for this list
  -o <op file>   to specify operator file
  -f <fact file>   to specify fact file
  -t <integer>    to specify a fixed number of time steps
  -i <info level>  to specify info level 1 or 2 (default is 0)
  -O <option list> to specify options you want
  -M <integer>    to specify alternative max nodes in a time step
                  (default is 256)
  -d           give default values to everything not specified

for example: graphplan -o fixit_ops -f fixit_facts1 -O IL -d

info@info-VirtualBox:~$ cd Scrivania/cart_example_gp/
info@info-VirtualBox:~/Scrivania/cart_example_gp$ graphplan -o cart.gp.ops -f ca
rt.gp.facts
LOAD
UNLOAD
MOVE
facts loaded.
number of time steps, or <CR> for automatic:
```

Figure C.5

In the figure below you can see that the solution is reachable in 3 steps:

Goals first reachable in 3 steps.

The goal is at time 4 joined with the description of our goals, so we have a valid plan:

```
goals at time 4:
  at_jack_mushrooms at_bobby_mushrooms
```

Then we have the description of each step with the actions and the total number of actions tried.

```
1 LOAD_jack_car_home
1 LOAD_bobby_car_home
2 MOVE_car_home_mushrooms
3 UNLOAD_jack_car_mushrooms
3 UNLOAD_bobby_car_mushrooms
0 entries in hash table,
2 total set-creation steps (entries + hits + plan lenght - 1).
5 actions tried
```

```

graphplan VirtualBox:~/Scrivania/cart_example.gps
graphplan VirtualBox:~/Scrivania/cart_example.gps graphplan -o cart_gp.ops -f cart_gp.facts
Usage: graphplan [options] <info-level>
  -o <op file>          to specify operator file
  -f <facts file>        to specify facts file
  -i <integers>          to specify a fixed number of time steps
  -l <levels>             to specify info level (0 or 2) (default is 0)
  -c <count>              to specify count (for testing)
  -m <integers>          to specify alternative max nodes in a time step
  -d                      give default values to everything not specified
  for example: graphplan -o first_ops -f first_facts -l 0 -c 250

cart_gp.facts
CLIPS
CLIPS
UNLOAD
MOVE
Goals
  time 0: 0 facts, or <0> for automatic
  time 1: 1 facts, 0 or <0> = min
  other: 
    'L' = Lower bound time needed by planning
    'B' = Build up to goals
    'H' = Hash table for memoization
    'E' = exclusive subsets
  time 1: 1 facts and 6 exclusive pairs.
  time 2: 3 facts and 12 exclusive pairs.
  time 3: 9 facts and 32 exclusive pairs.
  total set-creation time: 3 steps
  0 actions tried
  0.00 secs
graphplan VirtualBox:~/Scrivania/cart_example.gps graphplan -o cart_gp.ops -f cart_gp.facts

```

Figure C.6

If we increase the information level of the solution that we want to see we can have the description (see the figure below) of each time step and also the goals reached at each time-step adding the **-i <info-level>** option:

```
graphplan -o cart_gp.ops -f cart_gp.facts -i 1
```

```

Time: 1
LOAD(jack_car_home)
MOVE(car_home_mushrooms)
LOAD(bobby_car_home)

Time: 2
LOAD(jack_car_home)
MOVE(car_home_mushrooms)
LOAD(bobby_car_home)

Time: 3
UNLOAD(jack_car_mushrooms)
UNLOAD(bobby_car_mushrooms)
goals at time 4:
  at_jack_mushrooms_at_bobby_mushrooms

goals at time 3:
  on_bobby_car_at_car_mushrooms_on_jack_car

goals at time 2:
  on_jack_car_at_car_home_has-fuel_car_on_bobby_car

goals at time 1:
  at_bobby_home_at_car_home_has-fuel_car_at_jack_home

1 LOAD(jack_car_home)
1 LOAD(bobby_car_home)
2 MOVE(car_home_mushrooms)
1 UNLOAD(jack_car_mushrooms)
3 UNLOAD(bobby_car_mushrooms)
8 entries in hash table,
2 total set-creation steps (entries + hits + plan length - 1),
5 actions tried
0.00 secs
graphplan VirtualBox:~/Scrivania/cart_example.gps

```

Figure C.7

You can see that this time we obtain more informations (look at the central part of the figure above), but the solution is the same, the same valid plan.

Memoization

The concept of *memoization* is an important possibility in graphplan.

If at some step of the search, a subset of goals is not satisfiable, graphplan saves this result in a hash table. Whenever the same subset of goals is selected in the future will automatically fail.

```

1 LOAD_jack_car_home
1 LOAD_bobby_car_home
2 MOVE_car_home_mushrooms
3 UNLOAD_jack_car_mushrooms
3 UNLOAD_bobby_car_mushrooms
7 entries in hash table, 6 hash hits, avg set size 3.
15 total set-creation steps (entries + hits + plan length - 1).
17 actions tried
0.00 secs

```

Note the Hash Table entries

Figure C.8

C.2 Planning Domain Definition Language

PDDL

Other planners (*e.g.* Blackbox and FF) require that the files of the facts and actions are in *PDDL*.

PDDL stands for *Planning Domain Definition Language*.

Attempt to standardize a language for modeling planning problems:

- developed to support the International Planning Competition.

A *PDDL* definition consists of two parts, typically in separate files:

- a domain definition for predicates and actions;
- a problem file for the initial state and the goal specification.

PDDL - Domain

Let's see the approximate syntax of a domain definition:

```

(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  [(:types NAME_1 ... NAME_N)]
  (:predicates
    (PREDICATE_1_NAME ?A1 ?A2 ... ?AN))
  ...
  )

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
    ...
  )

```

Figure C.9

Looking at the figure above we have that:

- you have to define with the keyword `define` the domain name;
- then we have the `:requirements` section which is always the same for the requirements;
- then you have to define your types and your predicates;
- then in the second text block of the figure above are specified the actions indicating the action name, parameters, preconditions and effects.

```
(define (domain DOMAIN_NAME)
(:requirements [:strips] [:equality] [:typing] [:adl])
[(:types NAME_1 ... NAME_N)]
(:predicates
  (PREDICATE_1_NAME (?A1 ?A2 ... ?AN))
  ...
)
(:action ACTION_1_NAME           Object types that are part
  [:parameters (?P1 ?P2 ... ?PN)] of the domain
  [:precondition PRECOND_FORMULA]
  [:effect EFFECT_FORMULA] )
  ...
)
```

Figure C.10

```
(define (domain DOMAIN_NAME)
(:requirements [:strips] [:equality] [:typing] [:adl])
[(:types NAME_1 ... NAME_N)]
(:predicates
  (PREDICATE_1_NAME (?A1 ?A2 ... ?AN))           Types of predicate; eg
  ...
)
(:action ACTION_1_NAME                         at (home, car)
  [:parameters (?P1 ?P2 ... ?PN)]                )
  [:precondition PRECOND_FORMULA]
  [:effect EFFECT_FORMULA] )
  ...
)
```

Figure C.11

```
(define (domain DOMAIN_NAME)
(:requirements [:strips] [:equality] [:typing] [:adl])
[(:types NAME_1 ... NAME_N)]
(:predicates
  (PREDICATE_1_NAME (?A1 ?A2 ... ?AN)))
  ...
)
(:action ACTION_1_NAME           operators
  [:parameters (?P1 ?P2 ... ?PN)] Description
  [:precondition PRECOND_FORMULA]
  [:effect EFFECT_FORMULA] )
  ...
)
```

Figure C.12

PDDL - predicates and actions

Syntax for variables with type:

```
?<Var name> - <type>
```

Syntax of a formula (aggregation of predicates using logical operators):

```
(and PREDICATE_1 ... PREDICATE_N)
(or PREDICATE_1 ... PREDICATE_N)
(not PREDICATE)
```

Effects (Add and Delete lists are aggregated (the delete effects are preceded by a **not**):²⁰

```
:effect FORMULA
```

PDDL - .facts file

Structure of *.facts* file:

```
(define (<problem name>
  (:domain <reference domain>)
  (:objects
    {<object name> - <type>})
  )
  (:init
    {<predicate in the initial state>})
  )
  (:goal
    (and {<predicate in the final state>}))
  )
)
```

Input of a planner are two files:

- description of actions and types of objects (domain, *.pddl* file);²¹
- description of the initial state and the goal (*.facts* file).

C.3 SATPLAN & Blackbox

SATPLAN & Blackbox description

An observation:

- Graphplan looks for a valid plan by branching on sets of propositions;
- if the problem is simple, this saves several branches and a plan is found quickly;
- if the problem is complex, however, the number of sets on which to branch can become very big.

One idea (Kautz & Selman, '99):

²⁰In graphplan, remember we had the keyword **del**.

²¹In graphplan we had the operator *.ops* file.

- model the search for a valid plan as a SAT problem (SAT stands for satisfiability);
- Blackbox planner.

SATifiability problem: decide if a logical formula is satisfiable.
Declarative approach.

1. We model a problem by:

- logical variables (*0-1 domain*);
- operators **and**, **or**, **not**.

2. Using a solver *generic* to find an assignment of the variables that satisfies all the constraints.

Why *SAT*? Because there are *very efficient SAT solvers* that would become *usable as planners*.

Exercise

Start Blackbox.²²

- Suppose Blackbox is pre-installed on a virtual machine or on a local machine.²³
- Try to run it with:

```
info@info-VirtualBox:~$ blackbox
```

- Run the program using as input the two *.pddl* and *.facts* files that you have found in the `cart_example` folder:

```
blackbox -o <.pddl file> -f <.facts file>
```

Try to:²⁴

1. understand which method blackbox is using to find a valid plan (maybe try to vary the output level);
2. you can also employ blackbox by using different solvers for finding a valid plan (the online help does not say that, but you can also use `chaff`).

²²So we try to use `blackbox` and we will see that you're able to choose the planner that you want to use for solving your problem.

²³On a local Mac (Apple Silicon) it has been impossible to install – due to incompatibilities related to the architecture – try in the future following the infos in <https://henrykautz.com/satplan/index.htm> or in my repository <https://github.com/giodesi/BlackBox>.

²⁴These points are not mandatory, the main goal of talking about Blackbox is to see that exists a dedicated program able to find a valid plan; furthermore is that it is used the PDDL notation.

```

info@info-VirtualBox:~/Scrivania/cart_examples$ blackbox -o cart.pddl -f cart.facts
blackbox version 43
command line: blackbox -o cart.pddl -f cart.facts

Begin solver specification
  -maxint      0   --maxsec 10.000000  graphplan
  -maxint      0   --maxsec 0.000000  chaff
End solver specification
Loading domain file: cart.pddl
Loading fact file: cart.facts
Problem name: cart-example
Facts loaded.
time: 1, 7 facts and 6 exclusive pairs.
time: 2, 7 facts and 4 exclusive pairs.
time: 3, 9 facts and 12 exclusive pairs.
Goals first reachable in 3 steps.
64 nodes created.

#####
goals at time 4:
  at-crg_jack_mushrooms at-crg_bobby_mushrooms

-----
Invoking solver graphplan
Result is Sat
Iteration was 12
Performing plan justification:
  0 actions were pruned in 0.00 seconds

-----
Begin plan
1 (load jack car home)
1 (load bobby car home)
2 (move car home mushrooms)
3 (unload bobby car mushrooms)
3 (unload jack car mushrooms)
End plan

-----
5 total actions in plan
0 entries in hash table,
2 total set-creation steps (entries + hits + plan length - 1)
5 actions tried

#####
Total elapsed time: 0.00 seconds
Time in milliseconds: 0
#####
info@info-VirtualBox:~/Scrivania/cart_examples$ 

```

Figure C.13

Starting from the top we can see:

1. number of seconds of the default planner used which is **graphplan**;
2. after 10 seconds **blackbox** is able to use another planner that is a sub-planner called **chaff**;
3. then you can see that the goal is reached at the same time step of the previous example (because we're using **graphplan** also here);
4. then between the **Begin plan** and **End plan** clause you can see the valid plan.
5. Finally the total actions.

C.4 Fast Forward

FF: Fast Forward description

FF is an extremely efficient heuristic planner introduced by Hoffmann in 2000:

- Heuristic = at each state S is an evaluation of the distance from the goal using a heuristic function.

Basic operations:

1. starting from a state S , all successors S' are examined;
2. if you find a successor state S^* better than S , move to it and go back to step 1;
3. if no state is found with better evaluation, a complete search is performed (A^*), using the same heuristic.

FF: Fast Forward exercise

Start FF.

- Suppose Fast Forward is pre-installed on a virtual machine or on a local machine.²⁵
- Try to run it with:

```
info@info-VirtualBox:~$ ff -f <dataFile> -o <operatorFile>
```

- Run the program using as input the two *.pddl* and *.facts* files from the same classic example used with graphplan.

```
info@info-VirtualBox:~/Scrivania/cart_example$ ff -f cart.facts -o cart.pddl
ff: parsing domain file
domain 'CART' defined
... done.
ff: parsing problem file
problem 'CART_EXAMPLE' defined
... done.

Cueing down from goal distance: 5 into depth [1]
                                4           [1]
                                3           [1]
                                2           [1]
                                1           [1]
                                0

ff: found legal plan as follows

step   0: LOAD JACK CAR HOME
        1: LOAD BOBBY CAR HOME
        2: MOVE CAR HOME MUSHROOMS
        3: UNLOAD JACK CAR MUSHROOMS
        4: UNLOAD BOBBY CAR MUSHROOMS

time spent:  0.00 seconds instantiating 12 easy, 0 hard action templates
            0.00 seconds reachability analysis, yielding 9 facts and 12 actions
            0.00 seconds creating final representation with 9 relevant facts
            0.00 seconds building connectivity graph
            0.00 seconds searching, evaluating 6 states, to a max depth of 1
            0.00 seconds total time

info@info-VirtualBox:~/Scrivania/cart_example$
```

Figure C.14

Looking at the figure above we highlight:

²⁵The reference link provided is not available, so we are not able to access directly the code.

1. the two input files are the same given for `blackbox`;
2. here we have the distance from the goal for each action (being based on an heuristic);
3. we can see that in this case we have a *linear planner*, so we have one step for each single action; differently from `graphplan` where we had for the first steps two `load`.

References

GRAPHPLAN:

- <http://www.cs.cmu.edu/~avrim/graphplan.html>
- A. M. Blum and Furst, ‘Fast Planning Through Graph Analysis’, *Artificial Intelligence*, 90: 281-300 (1997).

Blackbox:

- <http://www.cs.rochester.edu/u/kautz/satplan/blackbox/index.html>
- SATPLAN: <http://www.cs.rochester.edu/u/kautz/satplan/index.htm>
- Henry Kautz and Bart Selman, ‘Unifying SAT-based and Graph-based Planning’, Proc. IJCAI-99, Stockholm, 1999.

Fast Forward:

- <http://members.deri.at/~joergh/ff.html>
- J. Hoffmann, ‘FF: The Fast-Forward Planning System’, in: AI Magazine, Volume 22, Number 3, 2001, Pages 57-62.

C.5 Exercises

Exercise 1

Modeling in the PDDL the *nine puzzle problem*.

- Try to solve it with the various planners.
- How to vary performances by *mixing a little*?

initial state			Goal		
7	3	4	1	2	3
2			4	5	6
8	6	5	7	8	

Figure C.15

The idea is that you have to define the name of the problem, the name of the domain (the name of the domain must be the same of the file name), then the objects. The idea is that *cells* are related to the position and the *piece* is what we have to move. The idea is that we have to move pieces from a cell to another, and you have to model the features of the cells: so, for example, there is a particular feature to model and also the predicates related to the possibility of a free cell. In general, finding a proper notation is the most difficult thing for this kind of problems.

Let's see a possible solution.

The first part of the *.facts* file – named `nine_puzzle.facts` – could be the following:

```
(define (problem puzzle9)
  (:domain nine_puzzle)
  (:objects
    c11 - cell
    c12 - cell
    c13 - cell
    c21 - cell
    c22 - cell
    c23 - cell
    c31 - cell
    c32 - cell
    c33 - cell
    p1 - piece
    p2 - piece
    p3 - piece
    p4 - piece
    p5 - piece
    p6 - piece
    p7 - piece
    p8 - piece
  )
)
```

Then we have the *.pddl* file – named `nine_puzzle.pddl` – that could be structured like so:

```
(define (domain nine_puzzle)
  (:requirements :strips :typing)
  (:types piece cell)
  (:predicates
    (at ?p - piece ?c - cell)
    (free ?c - cell)
    (nextc ?c1 ?c2 - cell)
  )
  (:action MOVE
    :parameters (?p - piece ?c1 ?c2 - cell)
    :precondition (and
      (at ?p ?c1) (nextc ?c1 ?c2) (free ?c2))
    :effect (and
```

```

(at ?p ?c2) (not (at ?p ?c1))
(free ?c1) (not (free ?c2)))
)
)

```

So we've our *.pddl* file that is composed in the first block by:

- the **domain**;
- the **requirements** (that are always the same);
- the **types** (in our case **piece** and **cell**);
- the **predicates**:
 - **at** \leftrightarrow *at* a piece in a particular cell (that is a position);
 - **free** \leftrightarrow a cell can be *free*;
 - **nextc** \leftrightarrow this is very important and indicates if two cells are next, because you can move between next cells.

In the second block we have the action **MOVE** which is based on the previous predicates and types, and it is composed by:

- **parameters** \leftrightarrow three parameters: one piece that you can move from a cell to another;
- **precondition** \leftrightarrow they are in *and* relation: the piece should be in the initial cell, the two cells are next, the final cell shoul be free (remember the PDDL notation with the *question mark*);
- **effect** \leftrightarrow also they are in the *and* relation: the piece is in the final cell, the piece is not any more in the initial cell, now the free cell is the first one while the second cell is not free anymore.

The second part of the *.facts* file could be the following:²⁶

```

(:init
(nextc c11 c12) (nextc c11 c21)
(nextc c12 c11) (nextc c12 c13) (nextc c12 c22)
(nextc c13 c12) (nextc c13 c23)
(nextc c21 c11) (nextc c21 c22) (nextc c21 c31)
(nextc c22 c21) (nextc c22 c12) (nextc c22 c23) (nextc c22 c32)
(nextc c23 c22) (nextc c23 c13) (nextc c23 c33)
(nextc c31 c21) (nextc c31 c32)
(nextc c32 c31) (nextc c32 c22) (nextc c32 c33)
(nextc c33 c32) (nextc c33 c23)
(at p7 c11) (at p3 c12) (at p4 c13)
(at p2 c21) (free c22) (at p1 c23)
(at p8 c31) (at p6 c32) (at p5 c33)
)

```

²⁶Second part of the *.facts* file which follows the first part; because in the *.facts* file you have to define the *objects* and the *types* and you have to describe the *initial state* and the *final state*.

7	3	4
2		1
8	6	5

Figure C.16

```
(:goal
  (and
    (at p1 c11) (at p2 c12) (at p3 c13)
    (at p4 c21) (at p5 c22) (at p6 c23)
    (at p7 c31) (at p8 c32) (free c33)
  )
)
)
```

1	2	3
4	5	6
7	8	

Figure C.17

Finally, using *Blackbox*, so the command:

```
blackbox -o <.pddl file> -f <.facts file>
```

The **nine_puzzle** solution is the following where we have 16 moves, the first is move **p1** from cell at position (2, 3) to cell at position (2, 2) and so on... so if we

well describe the problem, the planner will solve the problem.

```

Invoking solver graphplan
Result is Sat
Iteration was 1411
Performing plan justification:
  0 actions were pruned in 0.00 seconds

Begin plan
1 (move p1 c23 c22)
2 (move p4 c13 c23)
3 (move p3 c12 c13)
4 (move p1 c22 c12)
5 (move p2 c21 c22)
6 (move p7 c11 c21)
7 (move p1 c12 c11)
8 (move p2 c22 c12)
9 (move p4 c23 c22)
10 (move p5 c33 c23)
11 (move p6 c32 c33)
12 (move p8 c31 c32)
13 (move p7 c21 c31)
14 (move p4 c22 c21)
15 (move p5 c23 c22)
16 (move p6 c33 c23)
End plan

16 total actions in plan
288 entries in hash table, 135 hash hits, avg set size 12
438 total set-creation steps (entries + hits + plan length - 1)
374 actions tried

```

Figure C.18

Exercise 2

A building has 5 floors, 5 tenants and two lifts (currently at the ground floor and at the 4th floor).

Each tenant is at a starting floor and must go to another floor.

We need to find a plan that moves each tenant to its destination and that requires the minimum possible time (load/unload the lift takes the same time to move up one level).



Figure C.19

Model the problem – with the following specific description of the initial and final state – in PDDL and solve it (with Blackbox):

- the young Enrico is at the ground floor after a hard morning at work and is finally back home on the 3rd floor;
- Elena is at the ground floor should bring her shopping bags on the 2nd floor;
- After a family lunch on the 3rd floor, Ettore should go back to his workshop on the ground floor;
- Elvira and Ennio are on the 1st (to play cards) and on the 2nd (composing music) and should go back home on the 4th floor.

So the first part of the *.facts* file – named **elevators.facts** – could be:

```
(define (problem elevators_puzzle)
(:domain elevators)
(:objects
  f0 - floor
  f1 - floor
  f2 - floor
  f3 - floor
  f4 - floor
  ettore - person
  enrico - person
  elena - person
  elvira - person
  ennio - person
  e0 - elevator
  e1 - elevator
)
)
```

So by reading the previous text indications we modeled the following objects (*name*, *type*). Based on this description of objects is very simple to describe the initial state and the goal (we will see in the second part of the *.facts* file). The *.pddl* file – named **elevators.pddl** – could be:

```
(define
(domain elevators)
(:requirements :strips :typing)
(:types elevator person floor)
(:predicates
  (at_elv ?e - elevator ?f - floor)
  (at_prs ?p - person ?f - floor)
  (next_flr ?f1 ?f2 - floor)
  (in_prs ?p - person ?e - elevator)
)
)

(:action MOVE
  :parameters (?e - elevator ?f1 ?f2 - floor)
  :precondition (and (at_elv ?e ?f1) (next_flr ?f1 ?f2))
```

```

    :effect (and (at_elv ?e ?f2) (not (at_elv ?e ?f1)))
)

(:action LOAD
  :parameters (?e - elevator ?f - floor ?p - person)
  :precondition (and (at_elv ?e ?f) (at_prs ?p ?f))
  :effect (and (not (at_prs ?p ?f)) (in_prs ?p ?e))
)

(:action UNLOAD
  :parameters (?e - elevator ?f - floor ?p - person)
  :precondition (and (at_elv ?e ?f) (in_prs ?p ?e))
  :effect (and (not (in_prs ?p ?e)) (at_prs ?p ?f))
)

)

```

Finally, the first part of the *.facts* file could be:

```

(:init
  (next_flr f0 f1)
  (next_flr f1 f0) (next_flr f1 f2)
  (next_flr f2 f1) (next_flr f2 f3)
  (next_flr f3 f2) (next_flr f3 f4)
  (next_flr f4 f3)
  (at_elv e0 f2) (at_elv e1 f4)
  (at_prs ettore f3)
  (at_prs enrico f0)
  (at_prs elena f0)
  (at_prs elvira f1)
  (at_prs ennio f2)
)

(:goal
  (and
    (at_prs ettore f0)
    (at_prs enrico f3)
    (at_prs elena f2)
    (at_prs elvira f4)
    (at_prs ennio f4)
  )
)
)
```

So, invoking *Blackbox*:

```
blackbox -o <.pddl file> -f <.facts file>
```

We obtain the following:

```

Invoking solver graphplan
Result is Sat
Iteration was 17088
Performing plan justification:
  0 actions were pruned in 0.00 seconds

-----
Begin plan
1 (move e0 f2 f1)
1 (move e1 f4 f3)
2 (move e0 f1 f0)
2 (load e1 f3 ettore)
3 (load e0 f0 elena)
3 (move e1 f3 f2)
4 (move e1 f2 f1)
4 (move e0 f0 f1)
5 (move e1 f1 f0)
5 (load e0 f1 elvira)
6 (move e0 f1 f2)
6 (load e1 f0 enrico)
6 (unload e1 f0 ettore)
7 (unload e0 f2 elena)
7 (load e0 f2 ennio)
7 (move e1 f0 f1)
8 (move e1 f1 f2)
8 (move e0 f2 f3)
9 (move e0 f3 f4)
9 (move e1 f2 f3)
10 (unload e1 f3 enrico)
10 (unload e0 f4 elvira)
10 (unload e0 f4 ennio)
End plan
-----

23 total actions in plan
1123 entries in hash table, 1717 hash hits, avg set size 9
2849 total set-creation steps (entries + hits + plan length - 1)
4346 actions tried

#####
Total elapsed time:  0.03 seconds
Time in milliseconds: 33
#####

```

Figure C.20

Exercise 3

Consider Exercise 3 and Exercise 4 as an in-depth (not mandatory).
Two robots have to paint the tiles on a floor:

- each robot can move between adjacent tiles;
- each robot is loaded with a specific color and can use the color on the tile it is located to;
- robots cannot pass on a painted tile;
- the two robots can be on the same tile.

Note below that a robot can color a tile, even if the other robot is located there above at the same time.

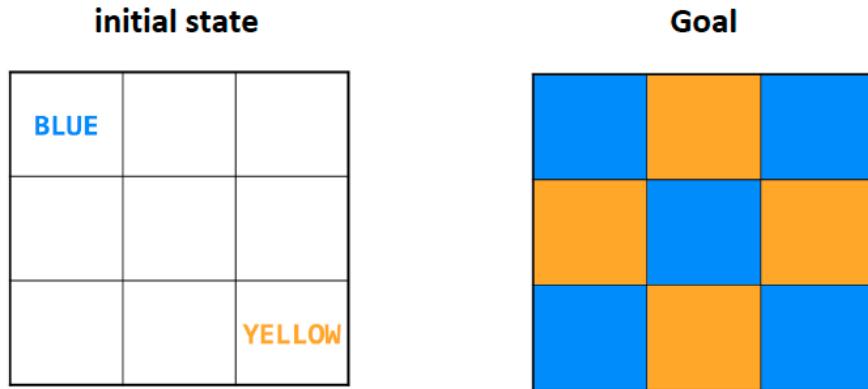


Figure C.21

The first part of the `.facts` file – named `painters.facts` – could be the following:

```
(define (problem painter_problem)
(:domain painters)
(:objects
  t00 - tile
  t01 - tile
  t02 - tile
  t10 - tile
  t11 - tile
  t12 - tile
  t20 - tile
  t21 - tile
  t22 - tile
  p0 - painter
  p1 - painter
  blue - color
  yellow - color
)
)
```

The `.pddl` file – named `painters.pddl` – could be the following:

```
(define
(domain painters)
(:requirements :strips :typing)
(:types painter tile color)
(:predicates
  (at ?p - painter ?t - tile)
  (clean ?t - tile)
  (colored ?t - tile ?c - color)
  (adj ?t1 ?t2)
  (loaded ?p - painter ?c - color)
)
(:action MOVE
```

```

:parameters (?p - painter ?t1 ?t2 - tile)
:precondition (and (at ?p ?t1) (adj ?t1 ?t2) (clean ?t2))
:effect (and (at ?p ?t2) (not (at ?p ?t1)))
)

(:action PAINT
  :parameters (?p - painter ?c - color ?t - tile)
  :precondition (and (at ?p ?t) (loaded ?p ?c) (clean ?t))
  :effect (and (not (clean ?t)) (colored ?t ?c))
)

)

```

The second part of the *.facts* file could be the following:

```

(:init
  (adj t00 t01) (adj t00 t10)
  (adj t01 t00) (adj t01 t02) (adj t01 t11)
  (adj t02 t01) (adj t02 t12)
  (adj t10 t00) (adj t10 t11) (adj t10 t20)
  (adj t11 t10) (adj t11 t01) (adj t11 t12) (adj t11 t21)
  (adj t12 t11) (adj t12 t02) (adj t12 t22)
  (adj t20 t10) (adj t20 t21)
  (adj t21 t20) (adj t21 t11) (adj t21 t22)
  (adj t22 t21) (adj t22 t12)
  (clean t00) (clean t01) (clean t02)
  (clean t10) (clean t11) (clean t12)
  (clean t20) (clean t21) (clean t22)
  (loaded p0 blue)
  (loaded p1 yellow)
  (at p0 t00)
  (at p1 t22)
)

(:goal
  (and
    (colored t00 blue) (colored t01 yellow) (colored t02 blue)
    (colored t10 yellow) (colored t11 blue) (colored t12 yellow)
    (colored t20 blue) (colored t21 yellow) (colored t22 blue)
  )
)
)
```

Exercise 4

Shape in the puzzle of PDDL *Hanoi Tower*:

- move a circle at a time, from one busbar;
- a smaller circle can never go below a larger one.

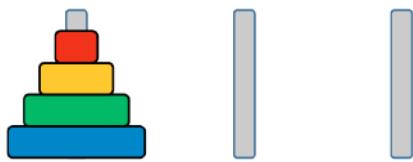


Figure C.22

And the Goal is:

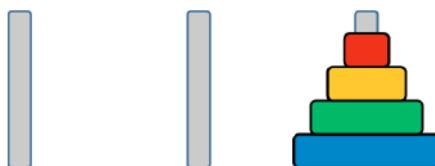


Figure C.23

Bibliography

- [1] S. J. Russel, P. Norvig, *Artificial Intelligence: A modern approach*, Prentice Hall, International edition.
- [2] R. J. Brachman, H. J. Levesque, *Knowledge Representation and Reasoning*, Elsevier, 2004.
- [3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (editors), *The description logic handbook: Theory, implementation, and applications*, Cambridge University Press New York, NY, USA, 2007.
- [4] Nils J. Nilsson, *Artificial Intelligence: A New Synthesis*, Morgan Kaufman, 1998.
- [5] M. Ginsberg, *Essentials of Artificial Intelligence*, Morgan Kaufman, 1993.
- [6] P. H. Winston, *Artificial Intelligence*, Addison-Wesley, 1992.