Chapter 2

Theoretical Frame

In this chapter, the main definitions of the theory of NP complete problems are presented, as well as definitions of the typing and set covering problems. In Section 2.2, is outlined the state of art of set covering problem, for example, using neural networks, and different algorithms tested so far. This work uses Logic Programming to precise the statements of the problem and to find a point of comparison of the solutions. Therefore, there is an introduction into Disjunctive Logic Programming and the DLV system in Section 2.3. These are specific definitions to the comprehension of this work. Some complementary bibliography is presented for finding more exhaustive definitions. Dynamic Programming definition is presented in Section 2.4.

2.1 Algorithms and Complexity

First of all, let us precise some concepts used along this thesis. It is important to know how these concepts will be understood.

Definition 2.1. [14] A problem will be a general question to be answered, given by a description of its parameters and a statement of what properties the answer or solution is required to satisfy.
**Definition 2.2.** [14] An instance of a problem is obtained by specifying particular values for all the problem parameters.

**Definition 2.3.** [14] Algorithms are general step-by-step procedures for solving problems. An algorithm is said to solve a problem $\Pi$ if that algorithm can be applied to any instance $I$ of $\Pi$ and is always guaranteed to produce a solution for that instance $I$.

The complexity of an algorithm is a very important concept in NP-completeness theory. Therefore, there are some algorithms that can be solvable in reasonable time, depending on the input, but some others can not. First of all, a precise definition of complexity is given.

**Definition 2.4.** [14] The time complexity function for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size.

More precisely, we say that:

**Definition 2.5.** [21] Let $f(n)$, $g(n)$ be two nonnegative functions defined for all positive integers.

1. It is said that
   \[
   f(n) = O(g(n))
   \]
   if there exist constants $c$ and $N$ such that $f(n) \leq c \cdot g(n)$ for all $n \in N$.

2. It is said that
   \[
   f(n) = \Theta(g(n))
   \]
   if there exist constants $c_1 \cdot c_2$ and $N$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \in N$. 
In general, $O-$ notation denotes an upper bound that may not be asymptotically tight. $\Theta-$ notation is used for denoting upper or lower bounds that are asymptotically tight. For example, if an algorithm needs $3n^2 + 5n$ steps, it runs in time $\Theta(n^2)$. Other bound definitions for functions are also explained in [21].

Some problems encountered in practice are solvable with an algorithm associated with a polynomial function. However, there are also many other problems where time is an exponential function of the input. This means that for some real problems, even a computer can spend too much time looking for the solution, even when the size of the input is not especially large, less than hundreds of thousands of elements.

There are two kinds of complexity. One is that inherent to the solution, when the output itself is required to be exhaustive and can not be bounded by a polynomial function. The other kind is that the problem is difficult, so exponential time is needed to find the solution.

In order to understand the explanation of NP complete problems, it is necessary to understand the Turing Machine. For a formal and deep explanation of the Turing Machine, consult [21] and [14]. To effects of this thesis, we can say that a Turing Machine is a control that reads a string and that has a transition function which works as the ‘program’ of the Turing Machine. The most important characteristics of the Turing Machine that must be understood are that:

1. It is capable of expressing any algorithm and simulating any programming language, and
2. It has a realistic function $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times D$, where $D = \{\leftarrow, \rightarrow, \cdot\}$. $Q$ is a state of the Turing Machine, $\Gamma$ is a set of symbols, which are read sequentially, and $D$, the direction, that indicates the next symbol to be read.

The nondeterministic Turing Machine is similar to the Turing Machine, except that
the transition function \( \delta \) is no longer a function, but rather a multi-valued relation. This means that, given a state and a set of symbols, the machine can, at each step, choose among several moves. This is an unrealistic model of computation. Therefore, we say that P problems are those solved in polynomial time by the deterministic Turing Machine and NP problems are those that can be solved in polynomial time in a nondeterministic Turing Machine.

Nowadays, the question whether \( P \neq NP \) remains unsolved. Several problems have been proved to be from one or another group, or they remain unclassified. However, there is an interesting characteristic about NP problems. They can be reduced to each other in polynomial time. This means that an instance of a problem \( \Pi \) can be transformed in polynomial time to an input of another problem \( \Psi \), and the algorithm to solve \( \Psi \) can be applied to \( \Pi \).

Set Covering Problem is one of the oldest and most studied problems in optimization. The decision problem associated was one of the first 21 problems shown to be NP-complete by Karpo in his paper “Reducibility among Combinatorial Problems” [15]. We present a formal definition as it is stated in [7]. Other definitions can be found in [14], [21], [5].

**Definition 2.6.** An instance of set covering problem consists of a finite set \( X \) and a Family \( F \) of subsets of \( X \), such that every element of \( X \) belongs to at least one element of \( F \). Therefore, we have:

\[
X = \bigcup_{S \in F} S
\]

We say that a subset \( S \in F \) covers its elements. The problem, then, is to find a subset \( C \subseteq F \) of minimal size, whose elements cover \( X \):

\[
X = \bigcup_{S \in C} S
\]

Any \( C \) that satisfies this equation, is said to cover \( X \).
2.2 State of the Art: Other solutions to the set covering

One of the proven most effective exact approaches to set covering is the general branch and bound algorithm, discussed in [23]. A variant of this algorithm can be found in [11]. However exact algorithms are not an option for large inputs in set covering problem. This means that more than 400 or 500 sets could become $2^{500}$ possibilities and the most of them should be analyzed. This is why polynomial algorithms are used, changing optimal solutions for efficient alternatives.

Many heuristics have been used to solve set covering. The most intuitive and first published approximation algorithm, according to [5], is the greedy heuristic. This algorithm was published with a worst-case analysis. The approximation ratio of the greedy algorithm was first defined as $\ln m + 1$, where $m$ is the size of base set $X$. This bound was improved in [20] to $\ln m - \ln \ln m + \Theta(1)$. This algorithm consists on selecting the set with the highest cardinality each time. Pseudo code versions of this algorithm are defined in [23].

Linear Programming is also a useful method to solve set covering. According to [7] and [23], there are many heuristics based on relaxing the solution and solving it as a linear programming problem. This means that a vector representing the elements of $F$ can take not only values in $\{0, 1\}$, but any real value in $[0, 1]$. A very good explanation about linear programming is found in [3].

A different approach to this problem is using pattern recognition. An evolutionary algorithm for applied set covering can be found in [12]. Some very interesting analysis comparing different algorithms is found in [5] and [3].
2.3 Logic programming

In this work, two of the algorithms were outlined in the languages of logic programming, before using a structured language. In this section some bibliography will be referenced about these items.

Logic programming is a different way of programming as compared to the conventional. The initial objective of logic languages was to have a code that was self-documented; this is, to understand what the program was doing with the information inside the code, reducing external documentation. It intended to support natural language processing. The main characteristic of logic programming is that the problem is described, instead of a sequence of steps to give a solution. An advantage of this approach is that it helps to find errors, or even better, to avoid them, and validate the program.

A clause in some way is a preposition, this is, a descriptive statement, which can be true or false. The clauses can be simple or composed by other clauses and binary operators like and, or, implies, not. A logic program is a set of clauses, and its work is to find a combination of values that render all this clauses true.

DLV is a disjunctive logic system that calculates stable models. This means that it uses deductive logic, and negation as failure, which means that every clause that is not said in the program is supposed to be false. The semantic associated to DLV is explained for the first time and in a complete way in [9]. Further information about this program can be found in [18] and [22]. Semantic models help to calculate different possibilities of true/false combinations of clauses, in order to avoid any kind of contradiction in the model.

XSB also called Prolog, aims to be a descriptive language, but it has still a small imperative content. Even though it has the same principles of logic, it works changing
clauses by functions and “calling” them. The way to operate is using queries as a way to interact with the program, instead of calculating models. It was selected because it manages lists in a natural way. More information about this language can be found in [10].

2.4 Dynamic programming

Dynamic programming is another strategy that solves problems combining the solution to sub problems. It is similar to the divide-and-conquer method. However, in some cases, sub problems are not independent. This means that they share sub problems. Dynamic programming solves sub problems only once and saves the answer. This approach is typically applied to optimization problems.

The first thing that must be done is to define the structure and characteristics of the solution. Then, a recursive function for getting the optimal solution at level \( n \) is defined. After that, the value of suboptimal solutions is computed, generally saving them in structures like arrays or matrices. After that, the solution is obtained from the previously defined function.

Further information about dynamic programming is found in algorithm books like [23]. Applications of this approach to set covering are explained in [5], citing to [8].