Chapter 3

Algorithms Design

An exact algorithm that uses a branch and bound technique was implemented to solve the set covering problem. Two polynomial algorithms were also designed and tested. In this chapter, design of these algorithms will be presented. A DLV specification will be presented in the first section. The exact algorithm will be explained in Section 3.2. Next two sections detail main ideas about the polynomial algorithms. The first polynomial algorithm uses a greedy approach and the other uses a dynamic programming technique. Finally, in last section, some other details about implementation are explained.

3.1 DLV specification

In this section a specification of the problem using DLV is shown. DLV is a disjunctive datalog system. It is useful to specify properties of a problem and to allow the inference engine to solve it. NP complete problems are in general efficiently solved with logic programming.

Some variables will be defined to explain the program. Let $< X, F >$ be an instance of the set covering problem. Let X be the set of elements, and F be a collection of subsets, $S_i \in F$ where $S_i \subseteq X$.

First, it is need to define a range of constants that have the property of being subsets of the set S, this is, the indexes of S_i.

```
i_subs(N) :- N>=1, number_of_subsets(X), N <= X, #int(N).
```

Then two subsets are defined, one of all the elements $x \in X$ and the other of the elements that have been already included in the solution.

```
tot1(X) :- in(R), s(R,X).
```

```
tot(X) :- i_subs(R), s(R,X).
```

It is asked for the inclusion of a variable subset R in the solution, using a disjunctive clause.

```
in(R) ∨ - in(R) :- i_subs(R).
```

Then it is defined that there is no solution if all elements x are not included in it.

```
:- tot(X), not tot1(X).
```

Finally, the best solution is looked for, the minimum number of sets.

```
:∼ in(_).
```

It means that DLV has to look for a solution with a minimum number of predicates "`in`"; in other words, with a minimum number of subsets in the solution.

This program is a small specification and it has many advantages. It is easy to find mistakes on it, and it can be checked in efficiency and effectiveness against the implemented algorithms.

The input has the next format:

```
s(S, E).
```

Where `S` is the number of subset, `E` is the number of element and `s` is the predicate which means: *E belongs to S*.

Example 3.1. Let X be the set $\{a, b, c, d, e, f\}$. Let F = $\{S_1, S_2, S_3, S_4\}$. Let $S_1 = \{a, b, c\}$, $S_2 = \{a, d\}$, $S_3 = \{b, e\}$ and $S_4 = \{c, f\}$.

This example is coded in DLV as follows:

```
s(1,a).
s(1,b),
s(1,c).
s(2,a).
s(2,d).
s(3,b).
s(3,e).
s(4,c).
s(4,f).
```

The output will be:

```
Best model: {in(2), in(3), in(4)}
Cost ([Weight:Level]): <[3:1]>
```

It means that sets S_2, S_3 and S_4 are an optimal solution with a cardinality of 3 elements.

3.2 Exact algorithm

An exact algorithm was implemented in C++. As the problem is NP complete, there are no different complex algorithms. An algorithm with a general branch and bound is used to solve set covering. It is outlined in [23]. This strategy assures the exactness of the algorithm. A recursive strategy is used in order to reduce the complexity of the implementation.

The main idea of the algorithm is to eliminate a set and recursively find the minimum size. Later, it adds this set to the solution and gets the minimal size. Both sizes are

compared and the function returns the set with the best solution.

It can be shown that this algorithm is Θn^2. The time grows exponentially; consequently, the time to solve a big input will be excessive. In order to reduce it some strategies are used to avoid recursive calls to the function.

The first prune is to determine a starting maximum bound with the help of a greedy approximation algorithm in order to follow only those branches that promise better results. This bound is updated as soon as a better solution is found during the run.

Also, on each iteration, two intuitive pruning rules are applied. These prunes will be explained below. They are used until no further pruning can be used. In this case, recursion is applied.

The algorithm works as follows:

[fontadjust, mathescape] function (integer) set$_c over(collectionXi, collectionwitness, setpartial_set$,
unique$_e lement_p rune(Xi) doXi \leftarrow$ X$_m odified; X_m odified \leftarrow$ subset$_p runing(Xi); Xi \leftarrow$ X$_m odified; X_m o$
unique$_e lement_p rune(Xi, partial_set, witness); while(X_m odified <> Xi)$

//set Selection set $S_i \leftarrow$ selected$_s et(X_m odified)$;

//Eliminate the set from the list of missing sets Xi $\leftarrow$ Xi S_i

//Add the set and check if solution is found partial$_s et = partial_s et \cup S_i$ if(S $\subseteq$ partial$_s et) return partial_s ize + 1$;

//Find the best solution without this set size1 $\leftarrow$ setCover(Xi, partial$_s et\ Si, S, partial_s ize$)

//Find the best solution with this set partial$_s ize \leftarrow$ partial$_s ize + 1 if((partial_s ize +$
$1) > maxSize) size2 \leftarrow$ set$_c over(Xi, witness, partial_s et, S, maxSize, partial_s ize)$

if(size2 ¿ size1) witness $\leftarrow$ witness $\cup \{S_i\}$ return size2 else return size1

The functions `subset_pruning()` and `unique_element_prune()` are explained in Sections 3.2.1 and 3.2.2 respectively.

The function `selected_set()` selects a subset to be proved. This set can be selected

by different strategies: randomly selected, the first one or the last one of the list. It can use a more sophisticated strategy; for instance, the set that has more elements or more intersections with other sets. In the implementation the first set of the list is selected. A more sophisticated strategy was not implemented because exact algorithm is going to be used for small inputs.

First of all, a function that gets the greedy solution of the problem is called. The function returns a bitmap array of the selected sets. Actually, all the implementations use a bitmap array to present the solution. The reason for this is that it is easier to check whether a set is inside the solution, outside the solution, or just has not been processed. Other advantage is that the positions in the array, representing the sets, can be directly accessed; there is no need of a sequential access.

Greedy function returns a number representing the maximum cardinality of the solution. The recursive function uses this size when it adds a new set, knowing beforehand which branches have not optimal solution.

The recursive function previously outlined receives the following parameters:

- **Sol**. It comes from solution. A value of 1 is set in this variable if the function got a solution in this iteration.
- **Next**. It is a pointer to the next subset that has not been processed in the solution.
- **Bsets**. Bitmap of subsets. These are arrays where the best solutions are saved. It has an efficient access to any position.
- **Subu**. Union of subsets that has been already included in the solution. It will help to avoid recursive calls when a solution has been found.
- **MaxS**. Maximum size of the solution. Since in the second part of the algorithm

a set is added to the solution, this bound has to be checked in order to reduce some recursive calls. When a better solution is found, this bound is updated.

- **Size**. It has the cardinality of the partial solution in this iteration.

The output of the function is the minimum size found, taking the partial solution as described before.

3.2.1 Subsets pruning

It is explained now how this pruning rule works.

Lemma 3.1. *Let $< X, F >$ be an instance of set cover. Let S_i and S_j be subsets of the base set X. This is, $\{S_i, S_j\} \subseteq F$. It can be shown that if $S_i \subseteq S_j$ then S_i can be eliminated as if it were not part of an optimal solution. This is because if S_i were part of any solution, there would be other solutions equal or better with the subset S_j. Proof of this is trivial.*

Corollary 3.1. *Let $C' \subseteq F$ be a collection of subsets which are necessarily added to the solution. Let $X' \subseteq X$ be the set of elements covered by C'. Notice that this is not exactly a set covering problem because there is a set of subsets which is fixed. However, Lemma 3.1 can still be applied as follows: It can be shown that if $(S_i \backslash X') \subseteq (S_j \backslash X')$ then, again, S_i is not part of the solution. The proof of this is similar to a proof of Lemma 3.1.*

It is important here to remark that this does not mean that S_1 could not be part of an optimal solution. Therefore, if all the optimal solutions are serached, follow a path where S_1 is part of the solution, is needed. For example:

Example 3.2. Let $< F = \{S_1, S_2, S_3, S_4\},\ X = \{a, b, c\} >$ be an instance of set covering. It is defined $S_1 = \{a, b\}, S_2 = \{c\}$ and $S_3 = \{b, c\}$. There are two optimal

solutions: $C_1 = \{S_1, S_2\}$ and $C_2 = \{S_1, S_3\}$. However, cardinality of the best solution can be found eliminating S_2 because it is a subset of S_3.

This prune is implemented using a simple cycle for each set remaining in the list of unprocessed sets. For every set not added to the solution, the algorithm will select subsets of the set and will add them to the list of subsets that are not in the solution. The function uses the same parameters explained in the previous section.

3.2.2 Unique element subset

This pruning rule adds necessary subsets to the solution.

Lemma 3.2. *Let $< X, F >$ be an instance of set cover. Let S_1 and S_2 be subsets of the base set X, $\{S_1, S_2\} \subseteq F$. Let $x \in X$ be an element of the base set. Let, furthermore, $x \in S_1$. If there is no subset S_2 such that $x \in S_2$ and $S_1 \neq S_2$, then can be said that S_1 is necessarily part of the solution in this branch, because x must part of the union of subset which will be covered by the solution. The proof of this is also trivial.*

A corollary is defined as in the previous section:

Corollary 3.2. *Let $C' \subseteq F$ be a collection of subsets which are necessarily removed from the solution. Let $X' \subseteq X$ be the set of elements covered by C'. After this restriction, the algorithm could have "unique elements" which belong to only one subset of the remaining subsets in F. The same rule of Lemma 3.2 can be applied to these subsets. A proof of this is analogous to a proof of the referred lemma.*

The corollaries give the possibility of applying the pruning rules at any level of the algorithm, as it was defined in the exact algorithm proposed in the beginning of Section 3.2.

The algorithm used for this pruning rule is presented here:

[fontadjust, mathescape] unique$_e$lement$_p$rune($collectionXi, setpartial_set$)$U\leftarrow \emptyset$ U2 $\leftarrow \emptyset$ For each $X \in Xi$ R $\leftarrow$ X partial$_s$et//$Step1U\leftarrow$ U R // Step 2 R $\leftarrow$ R U2 // Step 3 U $\leftarrow$ R $\cup$ U // Step 4 U2 $\leftarrow$ R $\cup$ U2 // Step 5

Lemma 3.3. *Let n be the number of elements. If the algorithm is applied to every set $S_i \in Xi$, it can be proven that at the end of any iteration, $U2$ is the set of all elements which appear at least once. Moreover, U is the set of unique elements, elements that have appeared exactly once.*

Proof. It will be proven by induction over the cardinality of Xi.

Base case. Following the steps of the algorithm:

$U \leftarrow \emptyset$

$U2 \leftarrow \emptyset$

1. $R \leftarrow S1 \backslash \emptyset; R = S1$
2. $U \leftarrow \emptyset$
3. $R \leftarrow R$
4. $U \leftarrow R; U = S1$
5. $U2 \leftarrow R; U = S2$

U and $U2$ have the properties enunciated before.

An element X with different properties will be taken to demonstrate that properties remain.

Case 1. If $X \in U$, and $X \notin R$, then X is unique, so X must be in U and $U2$ at the end of the iteration.

$X \in U$. It can be seen that as $X \notin R$, $X \in U$ after steps 2 and 4.

$X \in U2$. If $X \in U$ then, by hypothesis of induction, $X \in U2$ in the beginning. $U2$ is modified in step 5, however, no elements are removed because it is a union operation.

Case 2. If $X \in U$, and $X \in R$, then X is not unique, so X must not be in U at the end of the iteration. It can be seen as in the previous case that X remains belonging to $U2$.

After step 1, $X \in U$, but after step 2 $X \in U$. $X \in U2$ so $X \notin R$ by step 3. Therefore, $X \notin U$ after step 4 and 5.

Case 3. If $X \notin U$, and $X \in R$.

a) X is a unique element. Then $X \in R$ until step 4. After it, $X \in U$, and by step 5 $X \in U2$ too.

b) X is not a unique element, this is, $X \in U2$. Then, after step 3, $X \notin R$, so $X \notin U$ but $X \in U2$, as it was supposed to be. □

Now, it is necessary to find the subsets containing unique elements $x \in U$. These sets will be added to the solution.

Pruning functions use also the parameters for exact algorithm. However, they use two more flags:

- **EndF**. A value of 1 indicates that the brand has ended, this means that the algorithm has found a solution or there is no solution.
- **Sol**. This flag indicates if there is or not a solution.

3.3 Polynomial Algorithm using a Greedy Approach

Greedy algorithm is one of the best polynomial approaches found for set covering. Its approximation ratio is bounded by the function $\ln m - \ln \ln m + \Theta(1)$. The following

pseudocode is the general algorithm. The function *cardinality(S)* gets the number of elements in a set.

[fontadjust, mathescape] Greedy-Set-Covering(set X, collection F) C $\leftarrow \emptyset$ while(C $\neq$ X) max = 0 foreach S_i in F do card = cardinality(S_i) if (card¿max) Temp $\leftarrow S_i$ max = card /* The subset with maximum cardinality is added to the solution and remove it from the collection of subsets */ C = C $\cup \{Temp\}$ F = F $\setminus \{Temp\}$ return C

As it can observed, if the sets have the same size, greedy algorithm selects the first set found. Therefore, it breaks ties with alphanumeric order.

The greedy algorithm was implemented to define a maximum limit of sets in order to reduce recursion calls. The first implemented polynomial algorithm is a greedy variant. It uses two different strategies, which will be explained throughout this section.

The first strategy is the use of an exact algorithm at the lower level. It increments the exactness of the greedy at the end, where more ties can be found in the moment of selecting the set with the highest cardinality.

Therefore, the general algorithm stays as follows:

1. Apply general greedy in order to find the minimal bound.
2. It is defined a minimum limit in order to call the exact algorithm.
3. If the limit is bigger than some constant k, apply a greedy strategy on each iteration until this limit is reached. If it is not, go to the next step.
4. Apply the exact algorithm to the rest of sets.

Let n be the number of subsets obtained in the initial greedy. It is defined a constant $c' = 2 \times (\log_2 n)$. For example, if there are 500 subsets obtained, the exact algorithm after $c' = 2 \times (\log_2 500)$ is called; this is, 18 elements. In order to use improve exactness of the algorithm, the constant c is defined as $c = max(k, c')$ where k is a number, the

number of subsets, such that exact algorithm still finds the solution in a reasonable time.

This constant was defined in order to conserve the property of the algorithm of being polynomial, since branch and bound is exhaustive and $\Theta(2^n)$. Therefore, this algorithm is $\Theta(n^2 + 2^{2\times(\log_2 n)}) = \Theta(n^2)$, this is, n^2. Note that if the constant that multiplies the logarithm is increased, the order (Θ) of this algorithm also increases. For example, if it has $2^{5\times\log_2 n}$, the algorithm increases to Θn^5. Although it is still polynomial, it was preferred to have at most the same complexity.

This approach improves the greedy algorithm solutions. However, it has, of course some error margin compared against the exact algorithm. Hence, there will be presented the second strategy used to reduce error in the polynomial algorithm. It consists into taking pairs of sets instead of single sets on each iteration.

Let X be a collection of subsets $S_n \subseteq S$. Pairs of this subsets are selected, this is $< S_i, S_j >$. This pairs are elements of $S \times S$. Greedy algorithm is applied over this pairs. This means that it will select the pair with the highest cardinality until the set S is completely covered. This algorithm also breaks ties using alphanumeric order. The strategy of using pairs diminishes the probability of selecting a local best set. It is Θn^4 (remember that greedy approach is Θn^2 and combining all the subsets in pairs is also Θn^2). Intuitively, it gives a better warranty than greedy algorithm.

However, sometimes when the exact solution is an odd number, this algorithm has some trouble on getting it, as a consequence of selecting one set instead of two. The greedy algorithm there works better than greedy-by-pairs. The problem outlined can be easily solved if both strategies are combined. In this way, there are no disadvantages of using a greedy approach by pairs. Other strategy that can be used to solve this problem is to check redundancy on sets, or use pruning rules. It helps if pruning is used only the first time but some instances get worst solutions if the pruning functions are

applied at every iteration.

3.4 Dynamic programming approach

The knapsack problem is one of the best solved NP complete problems. It is explained and analyzed in [23]. There is an algorithm to solve it that gets the best solution in a reasonable time for the average case. Problems of this kind are called semi polynomial because they are polynomial according to other variables of the input. The algorithm uses a recursive function when it is outlined. At this point, correctness can be checked. This would represent a problem; however, the partial solutions of the function can be calculated previously, converting the time of solution, from exponential to polynomial.

An algorithm using the same approach was designed to solve set covering. It is defined with to variables, n and m, where n is the total number of subsets of a list, and m is the number of subsets supposed to be minimal. The goal is to find a function that maximizes the number of elements using m subsets. This function is similar to that one of the knapsack problem. Whether the set is added or not, is decided in each function. Let *dsc* be the function for the algorithm. Let *card* be a function that obtains the cardinality of any set. It can be defined as follows:

$$dsc(n,m) = \begin{cases} 0 & \text{if } m = 0, \text{ or } n = 0, \\ \max(\text{dsc}(n-1,m)), & \\ \text{card } (n \cup dsc(n-1,m-1)) & \text{Otherwise} \end{cases} \tag{3.1}$$

where function max is defined as:

$$max(a,b) = \begin{cases} a & \text{if } a \geq b \\ b & \text{if } b > a \end{cases} \tag{3.2}$$

The problem is solved in polynomial time. However, there is an error bound over the optimal value. One of the reasons, in addition to the NP complete nature of the problem, is that the maximum value is not always precise.

This algorithm was first defined in XSB language. The complete set is defined:

```
seti(X,L) :- setof(Y, s(X,Y), L).
```

Then, the base cases of the function are defined:

```
setc(N,0, []).
setc(0,M, []).
```

Then, the recursive function is defined:

```
setc(N,M, Out) :- seti(N, S1), N1 is N-1, setc(N1, M,S2), N> 0, M> 0,
                  M1 is M-1, setc(N1,M1,S3), union(S1,S3,S4),
                  maxc(S2,S4, Out).
```

It means "Maximal output of N sets using M sets is Out. S1 is defined as the list of element of the set N. S2 is defined as the elements of N-1; this is, when the set has not been added yet. S3 is defined as the maximum set of elements until N-1 taking M-1 sets. S4 is the union of S1 and S3, this is, the largest set including N. Therefore, Out is defined as the largest set between S2 and S4". It can be easily mapped to the function 3.1.

It is important the definition of the function which gets the maximum between two sets:

```
maxc(S1,S2,S1) :- length(S1, N), length(S2,M),  N == M.
maxc(S1,S2,S1) :- length(S1, N), length(S2,M),  N > M.
maxc(S1,S2,S2) :- length(S1, N), length(S2,M),  M > N.
```

The sets operation of "union" has also to be defined:

```
union([],X,X):- !.
union(X,[],X):- !.
union([X|Y], Z, [X|W]) :- not member(X,Z), union(Y,Z,W).
union([X|Y], Z, W) :- member(X,Z), union(Y,Z,W).
```

The input is the same than in DLV program. It is defined as `s(N,E)` where N is the number of set and E is the element belonging to this subset. XSB is a language that manages queries. In this case, the query would be:

```
? dsc(4,3,Out)
```

The output format is the maximum set found, which means that it is a list of elements.

```
Out = { a,b,c,d,e }
```

However, this program is used only for some small examples, where the number M of minimal subsets is known, or at least there is an approximate idea. The real program is not that simple. It fills a table, using i, j as the parameters n and m are used in the algorithm. The program stops when it is in the set N and the set saved in the table is the same than universal set. The column j is returned as the result (M).

A table is used for saving the results. By the nature of the function, only the previous line of results is necessary. Therefore, the table has only two rows and uses a module 2 operation to access the rows. This has the purpose of saving memory space, because sometimes the input can be really big. A complete table of the cardinalities is used, however, in order to help in the recovery of the best sets defined in the solution.

The witness recovery is just a pointer to the matrix that backs down the rows. It checks if there is any difference between two rows i and $i-1$ of the matrix. If they are different, it means that the row i was important and was included in the solution. It is

important then to follow the correct path. The column to start is the column M, called j, of the algorithm. For example, if the algorithm returns a best solution of 5 elements, and the number of sets is 20, it starts jumping back and it stops when it arrives to the first column.

3.5 General structure of the implementation

The software will be divided in different sections, with the goal of integrating these modules and exchange them to do the proofs.

3.5.1 Methodology

To develop the software, a simple cascade method of software engineering consisting on four steps was followed:

1. The first step was to analyze the problem. This is the shortest step because the typing problem and the conversion to set covering had already been done. On the other side set covering is a well known problem, and lots of literature about its description, definition, and even algorithms and proofs can be found. This part was outlined in Chapter 1.

2. The second part was to select out and design algorithms of those that have proven to be good.

3. The third step was the implementation of the algorithms. These steps, 2 and 3, are explained in Chapter 3.

4. Finally, they were tested. It was necessary to define the variables to be observed. In this case, the most important parameters are time, size of input and exactness

in the case of polynomial algorithms. There were two kinds of input. The first one was the random input, generated using a program that creates instances of set covering. The second type is input from a real database of instances of typing sequences, which helps to validate the results and to observe how different algorithms work on this kind of information.

There are different modules for the program. For example, it is important to have modules for reading input, as its format could vary. There is also used a module for set operations, like union and intersection. There is also a set of variables called "state of solution" defined for every iteration in greedy or in the exact algorithm. It is useful also in prunes.

As the main program, there is a module that can call the algorithms once or many times, in order to run many inputs and to work with them. The output is the running time of the entries and some other statistics like prunes, ties (in greedy algorithm) and some other stuff, useful to find some properties and conclusions of the algorithms. All the algorithms and libraries were programmed in C++.

A library of set manipulation was taken from [2]. This library implements set operations in a very efficient way. It uses bit level operations and logic operations. It was used as a module of the system. The advantage of bit operations is that logic operations, meaning set operations, increased the velocity of the set operations, crucial for a good performance of the algorithms. The operations found in the library were:

intersect: Finds an intersection between two sets.

difference: Finds the difference between two sets.

unite: Finds the union between two sets.

add_element: Adds an element to a given set.

remove_element: Removes an element from a given set.

clear_set: Removes all elements from a set.

print_set: Prints the elements of a set to the standard output.

isin: Tests if an element belongs to a given set.

Some other functions were added to the library using the same structure of char array and bits:

cardinality: Finds the number of elements of a given set.

is_empty: Tests if a given set is the empty set $\emptyset$.

subset: Tests if a set is subset of another given set.

copy_to: Returns a set with the same elements than a given set.

As it the problem was designed using a structured approach instead of object oriented programming. The diagrams and Documentation the software can be seen in Appendix A, where Data Flow Diagrams are presented, as well as function call diagrams and a class diagram. The class diagram only presents the general structure of the software for it only shows how the libraries of the methods are used. User's Manual of the software is found in Appendix B.

3.5.2 Input and output of algorithms

The input read has to be transformed to a set of character arrays. Up to now it has been implemented only in one format. First, two numbers are read: $setsize$ and $nofssets$, the total number of elements, this is $|X|$, and the number of subsets. The sets are read as pairs of numbers $< x, y >$ where x is the number of set, $0 < x < nofssets$, and y is the number of element, $0 < y < setsize$.

The output is the minimal number of subsets found by the algorithm. The list of subsets selected as part of the solution is also printed, this is, a list of numbers x, $0 < x < nofssets$, representing the number assigned to the subsets.