The Greedy Algorithms Class: Formalization, Synthesis and Generalization

B. Charlier[1]

bc@info.ucl.ac.be

Université catholique de Louvain
Département d'Ingénierie Informatique
Place Sainte Barbe 2
B-1348 Louvain-la-Neuve
Belgium

September 1995

Abstract

On the first hand, this report studies the class of *Greedy Algorithms* in order to find an as systematic as possible strategy that could be applied to the specification of some problems to lead to a correct program solving that problem. On the other hand, the standard formalisms underlying the *Greedy Algorithms* (matroid, greedoid and matroid embedding) which are dependent on the particular type *set* are generalized to a formalism independent of any data type based on an algebraic specification setting.

[1]Supported by the National Fund for Scientific Research (Belgium)

Contents

1 Introduction

Solving a problem with a problem dependent technique is good but solving it by using a technique which can be applied to many other different problems is much better. Such techniques are very useful and one has always been looking for them. Indeed, in many different textbooks about algorithms (e.g. [HS78, PS82, BB87, MS90, CLR90]), some special chapters are dedicated to those techniques: *Branch-and-Bound*, *Dynamic Programming*, *Divide-and-Conquer* and in almost all of them *Greedy Algorithms* . Each of these techniques can be applied to several problems. All the problems that can be solved by a particular technique form a *class* of problems. Among the different textbooks referencing the *Greedy Algorithms* class, many of them give a vague informal english definition of what greedyness is. Some of them give informal conditions that help to identify the *Greedy Algorithms* class more precisely. Finally, a very few (e.g. [MS90]) ponder on the question of formalizing the concept of greedyness. The approach developped in all these textbooks often focus on the way to apply the different techniques to a practical problem.
A complementary approach consist of investigating the classes of problems in order to formalize the features underlying the problems of the class. Some works have already been done for the *Greedy Algorithms* class some decades ago [Edm71] and more recently, some new very interesting results have come up [Hel89b, KLS91, HMS93].
The purpose of our report consist of combining the advantages of both approaches: we want to study how it is possible to get a program, based on the greedy technique, (as it's done in the first approach) for a problem lying in the *Greedy Algorithms* class while taking into account the formalizations of this problem thanks to the results of the second approach.
We can refine our main purpose into three subgoals:

1. to give to the readers an overview of the two approaches (it consists more or less to Section 2 and part of Section 4 for the first approach and to Sections 3 and 5 for the second one);

2. to reach our main purpose in a **systematic** way by using a synthesis process (see Section 4);

3. to find, among the second approach, some generalizations of the formalisms that exist and that all possess a similar limitation: being heavily dependent on the particular data type *set*. This subgoal consist of finding a new formalism which would be independent of any particular data type (see Section 6);

As the second subgoal is central in the following, we are now going to introduce briefly the general field in which we want to handle it.
Program synthesis is the process of obtaining **automatically** a program from a given formal specification.
Up to now, one does not know any technique that can be used by anyone in order to synthesize fully automatically a program from the specification of any problem. Rich and Waters [RW88] proposed three ways to simplify this idealistic technique:

- to replace the "fully automatical" constraint by a "semi automatical" one;
- to focus on a particular domain of interest instead of any possible specification;
- to let the system be used only by specialists instead of by anyone.

A very interesting approach to program synthesis consists of focusing on a particular *class* of problems that all can be solved by using a same general technique. This approach has two major advantages:

- as already mentioned, almost all textbooks about algorithms express the different classes that have already been identified. That work can thus be reused by analyzing all these already well-known classes;
- the fact that we only look to a particular domain (second simplification of Rich and Waters) is intrinsic to this approach and consists of a possible answer to the too numerous detractors of program synthesis. Indeed, theoretically, we know that undecidable problems exist and therefore, program synthesis cannot be defined on these particular problems. However, it is not because a technique cannot be used generally that we can deduce that it cannot be used very usefully for a whole class of particular cases.

Pioneering work in program synthesis among classes of problems has been done mainly by D. Smith. He already studied the following classes: *Divide-and-Conquer* [Smi85a, Smi85b, Smi87a], *Problem Reduction Generators* [Smi91] abstracting general Branch-and-Bound and Dynamic-Programming, *Global Search* [Smi87b] abstracting bactrack and Branch-and-Bound and *Local Search* (hillclimbing) [Low91] with M. Lowry (see Appendix).
D. Smith inserted those different classes into a taxonomy [Smi93]. He also showed in [Smi93] how it is possible to use this taxonomy in order to carry out the synthesis process incrementally.
Our second subgoal could thus be split into two:

- to study the *Greedy Algorithms* class uniformly with all the other classes studied by D. Smith;
- to insert this class into the already defined hierarchy.

There already exist some work about the *Greedy Algorithms* class. Indeed, R. Bird and O. de Moor, studied the classes of *Greedy Algorithms* [Bir92a, Bir92b], *Dynamic Programming* [dM94] as well as their combination [BdM92, BdM93]. Their work is obviously closer to formal calculation of programs than to program synthesis. Let us also mention P. Helman who discovered that the classes *Greedy Algorithms* , *Dynamic Programming* and *Branch and Bound* are all based on a similar strategy using dominance relations [Hel89b, Hel89a] but who did not investigate at all the synthesis process for solving problems among those classes.
Section 2 presents informally what are the algorithms that we consider greedy and will also give several examples.
Section 3 introduces the notion of *matroid*[2] which is the most well-known setting in which the *Greedy Algorithms* have been studied.
Section 4 shows the results of our studies about the synthesis of greedy programs for a given problem specification that is underlain by a matroid.
Section 5 generalizes the matroid formalism to two other formalisms: greedoid[3] and matroid-embedding.
Section 6 generalizes the matroid formalism, dependent on the data type *set*, to a new formalism independent of any data type.

[2]This word is derived from the word "matrix".
[3]This word is a synthetic blending of "matroid" and "greedy" [KLS91].

2 The greedy principle

Our purpose in this section is to satisfy the reader's intuition about the greedy algorithms. We first present an informal definition of greedyness and show how this definition can be applied to some examples.
The *Greedy Algorithms* class is intended to cover optimization problems.
We say that an algorithm is greedy (i.e. is based on the greedy principle) when the two following conditions are simultaneously satisfied:

- the algorithm has to construct a solution **incrementally** (i.e. in a sequence of stages);
- at each stage of the algorithm, the **best possible local choice** is made.[4].

The algorithm will stop when there is no more possible choice.
We could sum up these conditions by saying that, in a greedy algorithm, *a global optimum is iteratively constructed thanks to a succession of local optimization.*
This informal definition is intended to give a first description of which necessary conditions are needed to obtain a greedy algorithm. Moreover, it was deliberate to present it as weak as possible in such a way that it can be applied to as many cases as possible. The reader will understand it more easily by reading the following examples.
Let us also note that, although the greedy principle might look very simple, it has already been applied for solving many different problems among which some were non trivial at all.

Example 1 (Heaviest Forest Problem)
The problem is to find the heaviest forest of a graph. A forest is a set of arcs which does not contain any circuit. The weight of a forest is defined as the sum of the weight of all the arcs contained in the forest. Kruskal [Kru56] invented a greedy algorithm to tackle this problem:

- the forest is obtained by adding, at each stage, an arc to the set of arcs previously chosen;

[4]This is the reason why the greedy principle can be seen as an instance of the famous principle "Think globally, act locally"

- at each stage, the arc added is heaviest among all the arcs which have not yet been considered and which do not create any circuit if they are added to the set of arcs previously chosen.

Kruskal's algorithm was the first greedy algorithm to be found[5] and was historically very helpful in order to discover the whole *Greedy Algorithms* class. Figure 2 represents the different stages of Kruskal's algorithm over the graph of Figure 1 which is taken from [PS82].

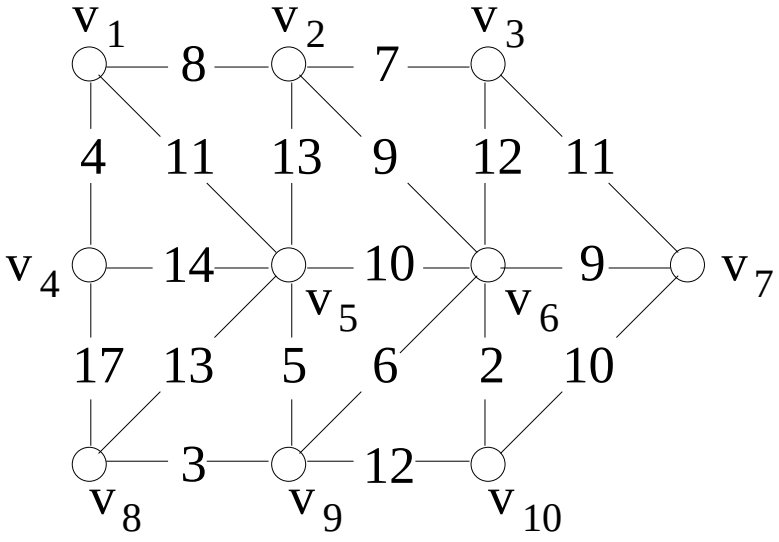

Figure 1: A graph.

Example 2 (Heaviest Matching Problem)
The problem is now to find the heaviest matching of a graph. A matching is a set of arcs such that there are no two arcs incident on the same node. Once again, the weight of a matching is defined as the sum of the weight of all the arcs contained in the matching.
We can easily think to a greedy algorithm to cope with this problem:

- the matching is obtained by adding, at each stage, an arc to the set of arcs previously chosen;

[5]It dates back to 1956

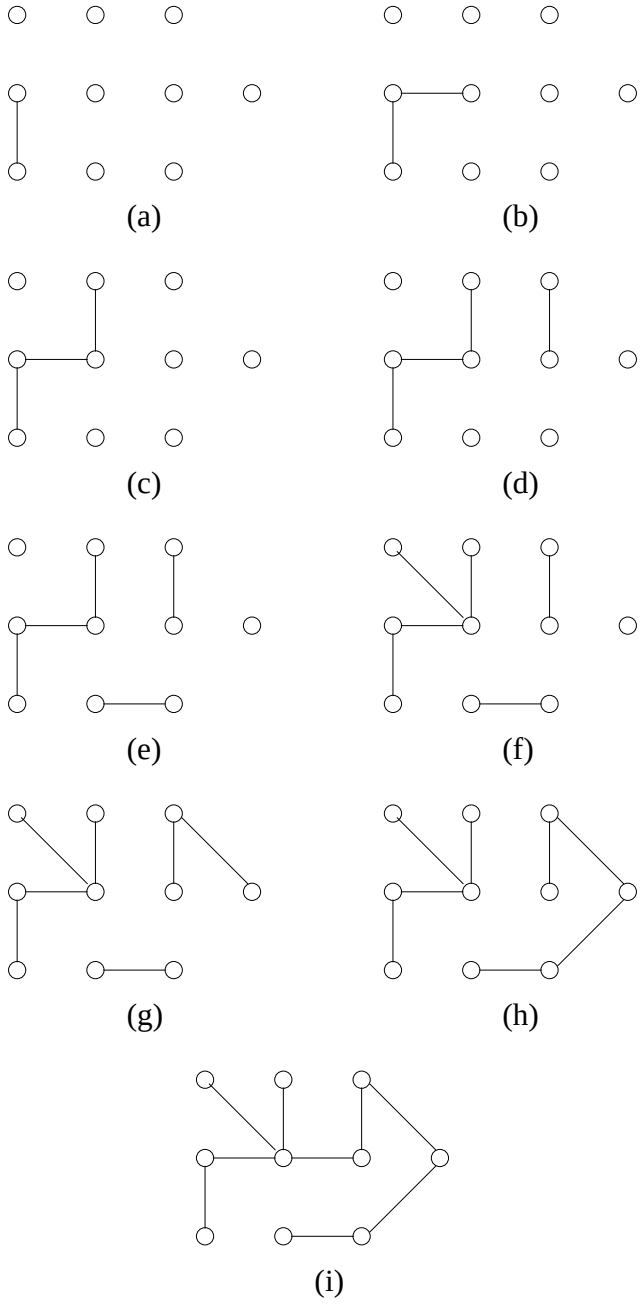

Figure 2: Stages of Kruskal's algorithm.

- at each stage, the arc added is heaviest among all the arcs which have not yet been considered and which are not incident to any node on which an arc among the set of arcs previously chosen is incident.

Figure 3 contains a graph with both its matching found thanks to the greedy algorithm and its optimal matching.

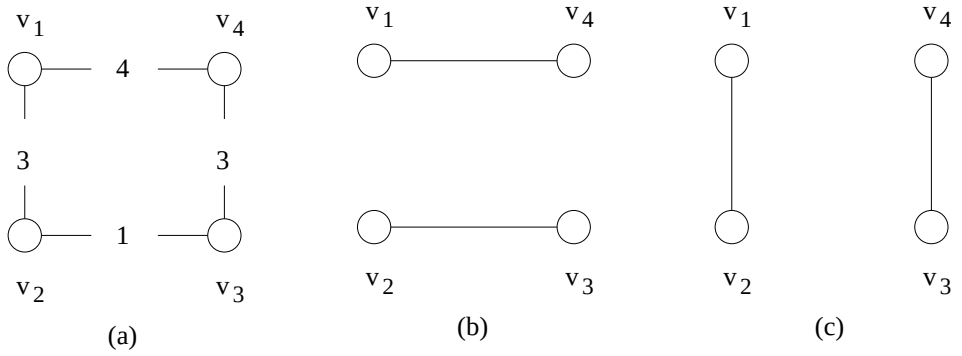

Figure 3: A graph (a), the matching obtained by the greedy algorithm (b) and the optimal matching (c).

As the two previous problems look similar and are treated very similarly thanks to a greedy algorithm, we could expect the algorithm to find the optimal solution in both cases or in neither of them. However, it can be surprisingly proved that the heaviest forest is solved optimally while the heaviest matching is not (as Figure 3 shows). The next section develops a theory which will allow us:

- to understand why the two problems aren't both solved optimally;
- to formalize and study *Greedy Algorithms* thanks to the set systems setting.

3 Matroid theory

3.1 Introduction

This theory was invented by Whitney in 1935[6] as an abstraction of the algebraic theory of linear independence. Later on, matroid theory has been shown to have links with many other theories.
Let us consider a particular problem of linear independence. Among a set of vectors of the same size and for each of which a weight is associated, let us find the lightest (or heaviest) set of independent vectors[7].
An obvious greedy algorithm to be applied to this problem would be:

- at each stage, a vector is added to the set of vectors already chosen;
- the vector added is the lightest among all the vectors which have not yet been considered and which are not a linear combinaison of the set of vectors already chosen.

Example 3 (Lightest Basis Problem)

Name	e_1	e_2	e_3	e_4
Value	$\begin{pmatrix} 1 \\ 0 \\ 3 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 4 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 3 \\ 3 \end{pmatrix}$
Weight	6	2	8	4

If the vectors are sorted in increasing weight, we get: $e_2 \rightarrow e_4 \rightarrow e_1 \rightarrow e_3$.
At the first stage, the algorithm includes e_2 (the lightest vector) in the matching. As $e_4 = 3 * e_2$, the second stage adds e_1 and not e_4 to the matching. The algorithm stops at that time because both e_4[8] and e_3 are linear combinaisons of e_1 and e_2. Indeed, we have: $e_4 = 3 * e_2$ and $e_3 = e_1 + e_2$.

[6] Matroid theory is thus anterior to Kruskal's algorithm.

[7] The weight of a set of vectors is the sum of the weights of all the vectors in the set.

[8] The fact of considering at this step element e_4 (which has already been considered at the previous step) is not a mistake. Of course, it's not useful in this particular problem but we shall see later that it could be crucial for some other problems.

It's very interesting to notice that this greedy algorithm is very similar to the one used to solve the heaviest forest problem. Moreover the lightest basis problem is solved optimally as well. It's at first sight very suprising to notice that this way of solving problems can be applied to two problems that are so different.

3.2 Formalization

We are now going to give a setting which will allow us to formalize some greedy algorithms. Let us first notice that in the three problems that were presented, the purpose was to find a particular subset S of a whole set E of objects:

- a subset of arcs among the set of all arcs of a graph;
- a subset of vectors among a set of vectors.

Each subset of E is either feasible or not. The subset which we are looking for is an optimal[9] one among all the feasible subsets of E.
The following definition is intended to represent the set E as well as all the feasible subsets of E.

Definition 1
$(E, \mathcal{F})$ is called a **set system** if:

- E is a finite set;
- $\mathcal{F}$ is a collection[10] of subsets of E (i.e. $\mathcal{F} \subseteq 2^E$).

Definition 2
If $X \subseteq E \ \wedge \ X \in \mathcal{F}$ then X is called **independent**.
If $X \subseteq E \ \wedge \ X \notin \mathcal{F}$ then X is called **dependent**.

[9] We thus need to have a cost function which assigns a value to each subset of E. We shall call it *weight* in the following. Let us also note that in the three previous problems, the weight of a subset of objects was obtained by summing the weight of all the objects in the subset. In this particular case, the cost function is called a linear cost function. We shall come back to this precise point later and see that there exist other possible cost functions.

[10] We use the term collection synonymously with the term set only in the case in which the elements of the set are sets themselves. Indeed, an expression such as "a collection of sets" is clearer than "a set of sets".

This notion of independence corresponds to our previous notion of feasibility. Due to history, many terms used in the matroid theory comes from linear algebra (e.g. independence, basis) and also from graph theory (e.g. circuit). We are now going to axiomatize some properties that have to be satisfied by $\mathcal{F}$ in order to form a matroid. Let us first note that, by convention, all the free variables in the axioms should be considered as implicitely universally quantified.

Axiom 1 (Trivial axiom)
$\emptyset \in \mathcal{F}$

Axiom 2 (Heredity axiom)
$X \in \mathcal{F} \ \wedge \ Y \subseteq X \Longrightarrow Y \in \mathcal{F}$

The trivial axiom is not of prior importance in this theory but makes the algorithm easier. Its sole[11] usefulness consists of preventing $\mathcal{F}$ from being empty therefore:

- a solution for any combinatorial problem (see Section 3.3) associated to any matroid can always be found;
- the initialization of the partial solution of the greedy algorithm by the empty set is always valid.

On the other hand, the heredity axiom is crucial and is a strong axiom. We shall see in Section 5.1 that other theories not requiring it can be created and used to formalize the greedy algorithm.
The heredity axiom translates the fact that the once a set is feasible, all the sets obtained by discarding some elements from it (i.e. its subsets) are also feasible.

Definition 3
A set system satisfying both the trivial and heredity axioms is called a **hereditary** set system[12].

Axiom 3 (Augmentation axiom)
$X, Y \in \mathcal{F} \ \wedge \ |X| = |Y| + 1 \Longrightarrow \exists x \in X \setminus Y \ : \ Y \cup \{x\} \in \mathcal{F}$

[11] Indeed, as soon as $\exists X \in \mathcal{F} \ : \ X \neq \emptyset$, the heredity axiom implies the trivial axiom.

[12] Let us only mention that the heredity set systems look similar to the notion of ideal and filter in lattice theory [DP90].

The augmentation axiom is very strong but is a necessary condition as we shall see in the following definition:

Definition 4
An hereditary set system satisfying the augmentation axiom is called a **matroid**.

In fact, in the case of a hereditary set system, there is an equivalent axiom to the augmentation axiom which is often easier to use: for any subset E' of E, all maximal independent subsets of E' have the same cardinality. Formally,

Axiom 4 (Cardinality axiom)
$$E' \subseteq E \ \wedge \ \{X, Y\} \subseteq \{Z | Z \in \mathcal{F} \ \wedge \ Z \subseteq E' \wedge \nexists W \in \mathcal{F} \ : \ W \subseteq E' \ \wedge \ Z \subset W\} \implies |X| = |Y|$$

Both the augmentation and the cardinality axioms seem to characterize a property:

- for which the **size** of the subsets plays a critical role;
- which is not based on some precise objects but more on any object of a whole large congruent class of objects. This kind of property might look intuitively strange but if we consider the lightest basis problem for example, the property of linear independence behaves exactly like that: in order to generate a plane, **any** pair of non colinear vectors are needed (whatever what they are exactly) and this constraint leaves quite a great choice...

Definition 5
A maximal independent subset X of E (i.e. $Z \in \mathcal{F} \ \wedge \ \nexists Y \in \mathcal{F} \ : \ X \subset Y$) is called a **basis**.

From now on, $\mathcal{B}$ will denote the set of all bases.
Let us now come back to the three problems we already mentioned, transform them in set systems $(E, \mathcal{F})$ and analyze if they satisfy all the axioms we mentioned.

1. For the heaviest forest problem, E represents the set of all the arcs of the graph and $\mathcal{F}$ represents the collection of all the forests over that graph.

(a) The empty set of arcs is trivially a forest.

(b) A subset of a forest is still a forest.

(c) Graph theory teaches us that the largest forests of a graph have all the same size (see [CLR90]).

$(E, \mathcal{F})$ is thus in this case a matroid.

2. For the heaviest matching problem, E represents the set of all the arcs of the graph and $\mathcal{F}$ represents the collection of all the matchings over that graph.

   (a) The empty set of arcs is trivially a matching.

   (b) A subset of a matching is still a matching.

   (c) The augmentation axiom isn't fulfilled as we have easily shown on the example in Figure 4.

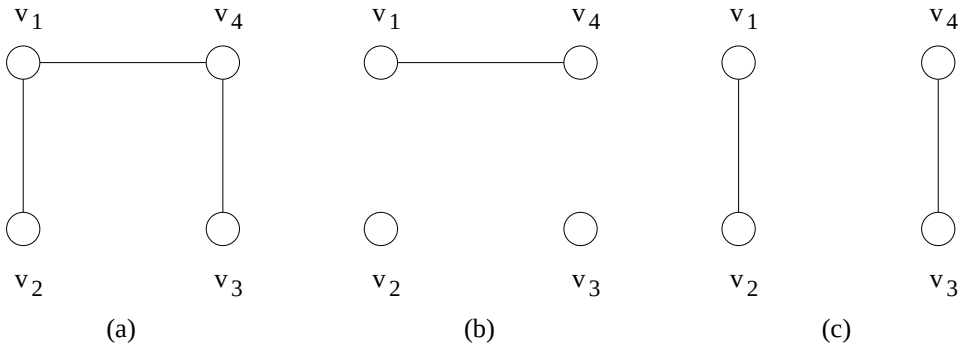

Figure 4: A graph (a) and two matchings (b) and (c) which do not satisfy the augmentation axiom if we choose the matching of (c) for X and of (b) for Y.

$(E, \mathcal{F})$ is **not** a matroid in this case but only a hereditary set system.

3. For the lightest basis problem, E represents the set of all the vectors and $\mathcal{F}$ represents the collection of all the independent subsets of that set.

(a) The empty set of vectors is trivially an independent set of vectors.

(b) A subset of an independent set of vectors is still an independent set of vectors.

(c) Linear algebra teaches us that the largest independent subsets of a set of vector are all of the same size [MB67].

$(E, \mathcal{F})$ is thus in this case a matroid.

3.3 Relation between matroid and greedy algorithm

As we have already said, if we give a cost assignement to the elements of E:

$$weight : E \rightarrow \mathbb{R}$$

it will induce a linear cost function over all the subsets of E:

$$weight^* : 2^E \rightarrow \mathbb{R} : weight^*(X) = \sum_{x \in X} weight(x)$$

We can now define the combinatorial problem which is associated to a set system $(E, \mathcal{F})$:

$$\text{Find } z \text{ such that: } z \in extrema(weight^*, \mathcal{B})$$

where *extrema* is defined as follows:

$$extrema(f, S) = \{y \mid y \in S \ \wedge \ \forall w \in S \ : \ f(y) \preceq f(w)\}$$

where $\preceq$ will always in the following stand either for $\leq$ or $\geq$. If we know a priori that $\preceq$ is instanciated to $\leq$ (resp. $\geq$), *extrema* will be called *lightest* (resp. *heaviest*).
In words, this combinatorial problem could be expressed by: *find the heaviest (or lightest) maximal independent set.* Let us remark that we always use the words weight, heavy,light... to refer to the cost of objects and we use the words maximal or minimal to refer to the cardinality of subsets.
Let us now express formally the greedy algorithm that is intended to solve the combinatorial problem associated to a set system $(E, \mathcal{F})$.

Definition 6
$EXT(X)$ represents the set of all feasible extensions of a subset X of E. Formally, $EXT(X) = \{x | x \in E \setminus X \ : \ X \cup \{x\} \in \mathcal{F}\}$.

The Greedy Algorithm (1)
begin
  $Solu := \emptyset$
  while $EXT(Solu) \neq \emptyset$ **do**
  begin
    $e \in extrema(weight, EXT(Solu))$
    $Solu := Solu \cup \{e\}$
  end
end

Let us remark that the relation replacing $\preceq$ in the definition of *extrema* that appears in the program should be the same that the one that appears in the specification of the problem. Indeed, if the problem is looking for an heaviest (resp. lightest) subset, each step of the algorithm will choose an heaviest (resp. lightest) element to include into that subset. In other words, the global and the local optimizations are expressed with the same order relations. This constraint will possibly be relaxed in our generalization of the greedy algorithm (see Section 6).

Theorem 1 (Edmonds 1971)
The greedy algorithm optimizes all linear cost functions over a hereditary set system $(E, \mathcal{F})$ if and only if $(E, \mathcal{F})$ is a matroid [Edm71].

Theorem 1 allows us to understand why the greedy algorithm did work on the heaviest forest problem as well as the lightest basis problem but did not on the heaviest matching problem: the formers could be formalized as matroids but the latter could not !
The reason why Theorem 1 is of great importance is due to the fact that among hereditary set systems, matroids (or more precisely the augmentation axiom or the cardinality axiom) represent an *exact* characterization of the

fact that a greedy algorithm finds a global optimimum for all linear weight function.

3.4 Some remarks

The most important remark to make is that the algorithm presented in Section 3.3 is not exactly the well-known original one. We choose however to present that algorithm for two main reasons:

- it's more general than the one which is often shown and we didn't want to start by a too particular case for us to understand the general algorithms in the next sections more easily;
- it's closer to the intuition and to our comments about the three examples we already presented.

However, the original algorithm is so widespread that we cannot fail to present it. In fact, it's a particular case of our more general algorithm that can be used when the heredity axiom is satisfied.

The Greedy Algorithm (2)
begin
  $Solu := \emptyset$
  while $E \neq \emptyset$ **do**
  begin
    $e \in extrema(weight, E)$
    $E := E \setminus \{e\}$
    if $Solu \cup \{e\} \in \mathcal{F}$ **then** $Solu := Solu \cup \{e\}$
  end
end

Instead of calculating $EXT(Solu)$ at each step as it's done in the first algorithm, the second one considers all the elements of E in nonincreasing order of weight and all the elements that can be added to the partial solution (Solu) are indeed added. The algorithm works because when an element of E is discarded (i.e. is not included in the partial solution), by the contrapositive of

the heredity axiom, we know that we won't have to attempt to include that element once again in the partial solution later in the process.
The second algorithm has drastically much better performances than the first one.
Indeed, the loop is executed at most $|E|$ times for both algorithms but each operation in the loop of the second one can be considered to take a constant time[13] (once the elements of E have been sorted according to their $weight$ order which has a complexity of $O(|E|log|E|)$) so the final complexity of the second algorithm is $O(|E| * log|E|)$ but the first algorithm has to compute $EXT(Solu)$ which is proportional to $|E| - |Solu|$ and its complexity is then $O(|E|!)$.
In Figure 5, we represented all possible executions of the second greedy algorithm over a set of four elements: $\{e_1, e_2, e_3, e_4\}$ whose weight order is: $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$.
At each step, two conditions are satisfied:

1. the partial solution is represented by the union of all the elements in the nodes of the path from the root of the tree to the element the algorithm is "looking at";

2. the algorithm goes down of one level in the tree: it "looks at" the leftmost son of the element it was "looking at" previously such that if that son is added to the partial solution, the new partial solution is still independent;

The algorithm starts looking at the root of the tree which represents the empty set. By the trivial axiom, the empty set is always independent. The algorithm finishes when the node which is looking at doesn't have any son satisfying the above second condition.
What is really interesting to notice in the tree of Figure 5 is the fact that for any node in the tree, all its left brothers don't appear in the substree of that node. This property is due to the heredity axiom. In fact, we can notice that the subtree of a node is the same as the subtree of the father of that node without that particular node, its subtree and all its left brothers as well as

[13]This hypothesis is quite strong for the test $Solu \cup \{e\} \in \mathcal{F}$ which might takes more than a constant time. However, the complexity of that test is often polynomial and then our conclusion comparing the two complexity is still valid.

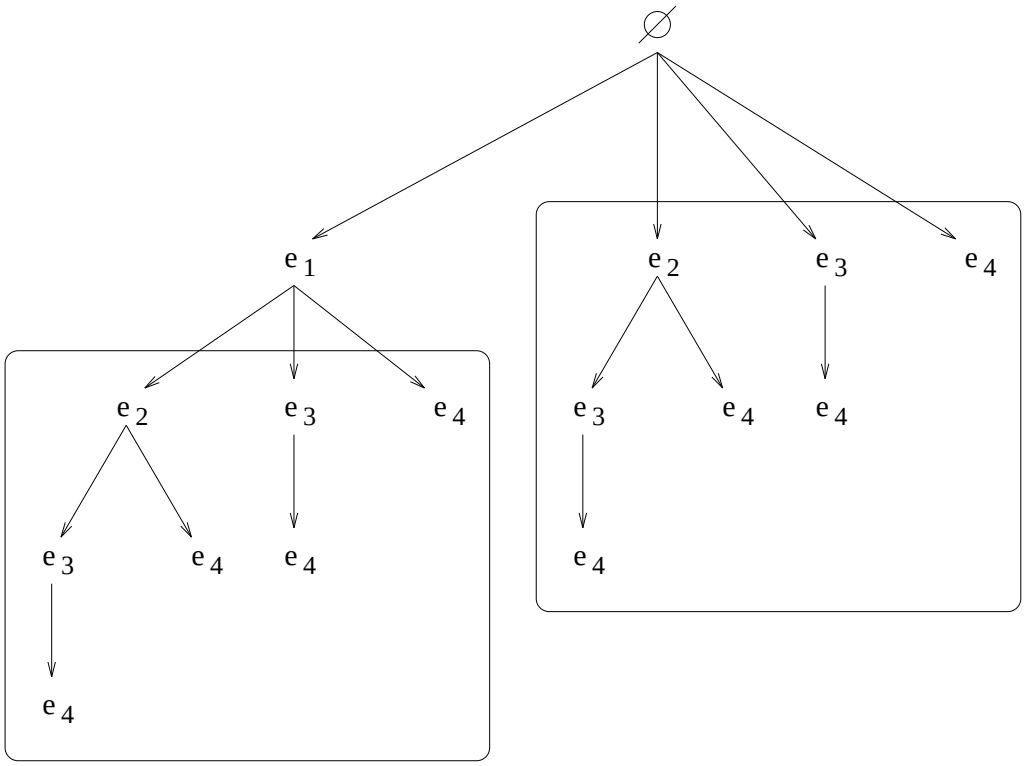

Figure 5: Representation of the execution of a greedy algorithm over a hereditary set system.

their subtrees. That fact can easily be verified in Figure 5 and an example of it is shown by the two rectangles.
The second important remark, comes from the fact that the combinatorial problem we are dealing with has not to be confused with the following one:

$$\text{Find } z \text{ such that: } z \in extrema(weight^*, \mathcal{F})$$

which looks for the heaviest (or lightest) independent subset and not, as before, for the heaviest (or lightest) maximal independent subset.
If the weight is restricted to positive values (i.e. $weight : E \rightarrow \mathbb{R}^+$) than the two problems:

Find z_1 such that: $z_1 \in heaviest(weight^*, \mathcal{B})$

Find z_2 such that: $z_2 \in heaviest(weight^*, \mathcal{F})$

are linked by the fact that: $weight^*(z_1) = weight^*(z_2)$ and as $\mathcal{B} \subseteq \mathcal{F}$, z_1 is a potential solution to the second problem as well. We consider problems only expressed with *heaviest* because the problem: Find z such that: $z \in lightest(weight^*, \mathcal{F})$ would have a trivial solution: $z = \emptyset$.
If the weight is not restricted at all, then the problem of finding the heaviest independent subset in $(E, \mathcal{F})$ can be solved by finding the heaviest basis of another set system $(E', \mathcal{F}')$ where E' is equal to E after having extracted all its elements of negative cost and $\mathcal{F}'$ is equal to the set of all the subsets of $\mathcal{F}$ that contain only elements from E'. Another way to solve this problem is to adapt one of the two greedy algorithms that we presented. Hereafter follows the first greedy algorithm modified in order to tackle this problem:

The Modified Greedy Algorithm
begin
  $SOLU := solu := \emptyset$
  while $EXT(solu) \neq \emptyset$ **do**
  begin
    $e \in heaviest(weight, EXT(solu))$
    $solu := solu \cup \{e\}$
    if $weight^*(SOLU) \geq weight^*(solu)$ **then** $SOLU := solu$
  end
end

The first two greedy algorithms were progressing towards the final solution modifying the variable *Solu* at each step. This algorithm carries out the same operations but *solu* is not equal to the final solution at the end of the algorithm. The final solution is equal to the best value taken by *solu* during the execution of the algorithm. The purpose of the variable $SOLU$ is to contain that value when the algorithm finishes. Of course, in the case of

a matroid, this algorithm is not very useful compared to the first solution we gave in the previous paragraph. Indeed, we know that at each time an element of positive (resp. negative) weight is added to the partial solution, the new partial solution obtained is always better (resp. worse). Moreover, as the heredity axiom would be verified, the e chosen at each step would be in nonincreasing order (if $\preceq$ is instanciated to $\geq$). Thus we could know that once we have found an element of negative weight as a result of a local optimization, the algorithm should be stopped because the partial solution cannot be improved any more. However, we wanted to show this modified greedy algorithm because it will be used later to solve similar problems for an extension of matroids (see Section 5.1).
We can notice that the two alternative solutions that we presented to solve our problem of finding the heaviest independent subset when the weight is not restricted at all are very similar.
As we have just shown that the problem of finding an extremal element over the set $\mathcal{F}$ can always be solved by resolving a related problem of finding an extremal element over the set $\mathcal{B}$, there is no restriction of considering only the combinatorial problem we defined.
Moreover, if we restrict the weight to have only positive values, we don't loose any generality. Indeed, if there are some elements with negative weight, it's always possible to add a positive value to the weight of all the elements of E such that it would be positive for all of them. As we know that all the bases have the same number of elements, that process is equivalent to adding the same value to the weight of all the bases and doesn't change their relative order. Thus, in doing so, the greedy algorithm will find the same solution (or an equivalent one !).
A third remark comes from the fact that the greedy algorithms we presented are sometimes called *best-in* greedy algorithms (see [KLS91]) because they start with the empty set and augment it at each step with the *best* local choice which keeps the partial solution independent if added to it. However, there is a dual strategy, which is called the *worst-out* greedy algorithm. It consists of starting with the whole set of objects and deleting at each step the worst possible local choice as long as the intial set still contains a basis. The widespread use of the first algorithm compared to the second one comes from the fact it's often much easier to verify that an independent set to which we add an element stays independent than to test that a set containing a basis still contains one when we delete one of its element.

Eventually, when we tackle an optimization problem in which we look for an heaviest object, we want to emphasize the importance for the reader to detect the difference between the cases in which we could also have done the dual (look for a lightest object) and the case in which looking for an heaviest object was the only possible situation. This remark is even more important because, from now on, by default, the future optimization problems will look for an heaviest object when the dual is also possible. It will be mentioned when we use the dual or when only one of them is possible.

4 Synthesis of greedy algorithms

4.1 Introduction

In this section, we are going to investigate how it's possible to help a user in the design of a greedy algorithm solving a given formal specification.
Our approach will be similar to Smith's one in the way to:

- characterize the class of *Greedy Algorithms* as Smith already did for *Divide-and-Conquer* [Smi85a, Smi85b, Smi87a], *Problem Reduction Generators* [Smi91], *Global Search* [Smi87b] and with Lowry for *Local Search* (see Appendix);
- formalize the synthesis process [SL90, Smi92, Smi93].

Two interesting consequences of this study would be:

1. to create a framework which could help in synthesizing greedy algorithms;
2. to improve Smith's hierarchy of classes [Smi93] by inserting the *Greedy Algorithms* class and comparing it to some others (see Appendix).

A specification[14] will be represented as follows:

Spec *SpecName*

Imports

[14] In the sequel, we shall use the terms *specification* and *theory* synonymously.

$Spec_1, Spec_2, ...$

Sorts

$S_1, S_2, ...$

Operations

$O_1 : ...$

$O_2 : ...$

Axioms

$Ax_1 : ...$

$Ax_2 : ...$

End-Spec

in which $SpecName$ is the name of the specification, $Spec_i$ represent other specifications used in the $SpecName$ specification, S_i are the sorts (i.e. the types) which are useful to express the signatures of the operations O_i. Finally, the axioms Ax_i are intended to define some operations and to characterize properties of others.
As any problem consists of a set of possible inputs $x \in D$ that satisfy a particular condition called the *input condition* $I(x)$ and of a set of outputs called the *feasible solutions* $z \in R$ such that some condition called the *output condition* $O(x, z)$ holds, any problem specification will contain at least the following format:

Spec $Basic - Problem - Theory$

Imports

Sorts

D, R

Operations

$I : D \rightarrow Boolean$

$O : D \times R \rightarrow Boolean$

Axioms

End-Spec

Of course, this is the most abstract interesting problem. All pratical problems

will add sorts, operations and axioms to the $Basic - Problem - Theory$[15]. We associate to this $Basic - Problem - Theory$ another specification called $Basic - Program - Theory$ which includes a function f that **solves** the problem expressed by $Basic - Problem - Theory$: for any data x satisfying the input condition, it finds a feasible solution. Formally,

Spec $Basic - Program - Theory$

Imports

Sorts
D, R

Operations
$I : D \rightarrow Boolean$
$O : D \times R \rightarrow Boolean$
$f : D \rightarrow R$

Axioms
$\forall x \in D : I(x) \Longrightarrow O(x, f(x))$

End-Spec

An identical way to express that $Basic - Program - Theory$ more like a program is the following:

function $f(x : D) : R$
where $I(x)$
returns $(z \mid O(x, z))$
$= Body(x)$

in which $Body(x)$ represents the program which is executed to compute $f(x)$. When the problem we deal with is an optimization problem, we could also use $Basic - Problem - Theory$ and $Basic - Program - Theory$. However, we think that it's much clearer to use two other theories $Basic - Optimization - Problem - Theory$ and $Basic - Optimization - Program - Theory$ in which the feasibility condition (which is basically an absolute condition) is included in the operation O and the optimality condition (which is basically a relative condition) is included in a new operation *cost*.

[15] The practical problem is then called an *extension* of the $Basic - Problem - Theory$.

The structure of $Basic-Optimization-Problem-Theory$ is equivalent to the structure of $Basic-Problem-Theory$ extended by a cost structure:

1. Cost domain: C
2. Cost ordering: $\preceq : \ C \times C \rightarrow Boolean$
3. Cost function: $cost : D \times R \rightarrow C$

as well as the axioms of a total order (reflexivity, antisymmetry, transitivity and totality) for the cost ordering.
Thus, we have:

Spec $Basic-Optimization-Problem-Theory$

Imports

Sorts
D, R, C

Operations
$I : D \rightarrow Boolean$
$O : D \times R \rightarrow Boolean$
$\preceq : C \times C \rightarrow Boolean$
$cost : D \times R \rightarrow Boolean$

Axioms

End-Spec

And the associated $Basic-Optimization-Program-Theory$ can be expressed (in the program like version) as follows:

function $f(x : D) : R$
where $I(x)$
returns $(z \in extrema(\lambda y.cost(x, y), \{z \mid O(x, z)\}))$
$= Body(x)$

Typically, C will be instanciated by $\mathbb{R}$ or $\mathbb{R}^+$ and $\preceq$ by $\leq$ or $\geq$. So in order to shorten the specifications in the sequel, C and $\preceq$ will be omitted.

Definition 7
A program f is called **consistent** when it satisfies the axiom of the $Basic - Program - Theory$.

Sections 4.2 and 4.3 will respectively investigate how:

- to characterize the essence of the *Greedy Algorithms* class;
- to use this charactarization in order to help the synthesis of a greedy program that would solve a problem given by its formal specification.

4.2 An algorithm theory for the *Greedy Algorithms* class

Let us first remind the reader that in this section, we restrict our study to problems which are underlain by a matroid.
An algorithm theory is an abstract program theory (see p.24) for which the function f solving the problem is given as a program using executable operations as well as operations among the ones of the specification. Such algorithm theory is equivalent to a program scheme: a program based on the algorithm can be obtained by instanciating the operations of the specification of the algorithm theory. If all those instanciations are executable, the associated algorithm of the theory then becomes a program as it is expressed only by executable operations.
In the algorithm theory we construct in this section, we want to make the matroid structure apparent: we thus have to find a representation for both E and $\mathcal{F}$ of the set system $(E, \mathcal{F})$. Let us first recall that the combinatorial problem associated with a matroid for which each element has a positive weight is to find an optimal subset in $\mathcal{F}$(which will be trivially also a basis). As $\mathcal{F}$ corresponds to the set of all independent (or feasible) subsets of E, it could be expressed thanks to the operation O which characterize all the feasible solutions of a problem: $\mathcal{F} = \{z \mid O(x, z)\}$.
As it is not possible to find E thanks to the operations of the $Basic - Optimization - Problem - Theory$ we add a new operation called Get_E which takes a data from the input domain and is intended to return the set E of all the objects that can be contained in the elements of $\mathcal{F}$.
Let us also note that we know in advance that the type of the result of the greedy algorithm (represented by R in the $Basic - Problem - Theory$) is a "set of something" where something is the type of the ground objects. So

we decided not to use R as a sort but to use the type of the ground objects that we call R' instead.
Here is now the matroid specification:

Spec $Matroid$

Imports

$Boolean, set$

Sorts

D (input domain)
R' (domain of the ground objects)

Operations

$I : D \rightarrow Boolean$
$O : D, set(R') \rightarrow Boolean$
$Get\_E : D \rightarrow set(R')$
$weight : D, R' \rightarrow \mathbb{R}^+$

Axioms

(Get_E(x) is a superset of all feasible solutions)

$$\forall(x : D)\ \forall(y : set(R'))\quad I(x)\implies [O(x,y)\implies y \subseteq Get\_E(x)]$$

(Trivial axiom)

$$\forall(x : D)\quad I(x)\implies O(x,\emptyset)$$

(Heredity axiom)

$$\forall(x : D)\ \forall(y,z : set(R'))\quad I(x)\implies [z \subseteq y\implies (O(x,y)\implies O(x,z))]$$

(Augmentation axiom)

$$\forall(x : D)\ \forall(y,z : set(R'))\quad I(x)\implies [O(x,y)\ \wedge\ O(x,z)\ \wedge\ |y|+1 = |z|)\implies (\exists(e : R')\ e \in z \setminus y\ :\ O(x, y \cup \{e\})]$$

End-Spec

The first axiom comes from the addition of the operation Get_E and the last three correspond to the matroid definition (see p.4).

It's obvious (see Section 3.2) that we could have replaced the augmentation axiom by the following cardinality axiom:

$$\begin{array}{l} \forall(x:D)\ \forall(w,y,z:set(R'))\quad I(x)\implies \\ [w\subseteq Get_E(x)\ \wedge\ \{y,z\}\subseteq\{v|O(x,v)\ \wedge\ v\subseteq w\ \wedge \\ \qquad\qquad \nexists u\ :\ O(x,u)\ \wedge\ u\subseteq w\ \wedge\ v\subset u\}]\ \implies |y|=|z| \end{array}$$

and we would have obtained an equivalent theory.
The heredity axiom expresses the fact that O is antimonotonic (i.e. monotonic decreasing) with respect to $\subseteq$.
This theory refers implicitly to the matroid $(Get\_E(x), \{z \mid O(x,z)\})$ for any data $x \in D$ satisfying the input condition I.
In order to have an algorithm theory of matroid, we must add to the previous specification an abstract greedy program and prove that it's consistent. These two tasks are done Theorem 2. The reader should note that the program given in Theorem 2 corresponds to the second greedy algorithm mentioned (see Section 3.4, p.17) written in a functional recursive way.

Theorem 2
In a matroid theory, the following greedy recursive program specification is consistent.

function $F(x:D):set(R')$
  where $I(x)$
  returns $(z \mid z \in extrema(\,weight^*,\ \{y \mid O(x,y)\}\,))$
  $=\ F_G(x, Get\_E(x), \emptyset)$

function $F_G(x':D, GE:set(R'), solu:set(R')):set(R')$
  where $I(x')\ \wedge\ GE\cap solu=\emptyset\ \wedge\ O(x', solu)$
  returns $(solu\cup z \mid z\in extrema(weight^*,$
    $\{y \mid O(x', solu\cup y)\ \wedge\ y\subseteq GE\ \}\,))$
  $=$ **if** $GE=\emptyset$
    then $solu$
    else let $lc\in extrema(weight, GE)$
      if $O(x', solu\cup\{lc\})$
      then $F_G(x', GE\setminus\{lc\}, solu\cup\{lc\})$
      else $F_G(x', GE\setminus\{lc\}, solu)$

where the first argument of $weight$ and $weight^*$ has been discarded for clarity reasons (e.g. $weight$ stands for $\lambda v.weight(x,v)$) and where $weigh^*$ is defined by: $weight^* : D, set(R') \rightarrow \mathbb{R}^+ \ : \ weight^*(x, \{e_1, \ldots, e_n\}) = \sum_{i=1}^{n} weight(x, e_i)$

Proof

The main challenge is to show the consistency of F_G:

$$\begin{array}{l} \forall (x' : D)(GE, solu : set(R'))\ I(x') \wedge \\ \quad GE \cap solu = \emptyset \ \wedge \ O(x', solu) \Longrightarrow [F_G(x', GE, solu) = solu \cup z \ \wedge \\ \qquad z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \ \})] \end{array} \quad (1)$$

If we assume the consistency of F_G, the one of F is easy to prove:

$$\forall (x : D)) \quad I(x) \Longrightarrow F(x) \in extrema(weight^*, \{y \mid O(x,y)\})$$

By definition of $F(x)$, proving:

$$F(x) \in extrema(weight^*, \{y \mid O(x,y)\})$$

is equivalent to proving:

$$F_G(x, Get\_E(x), \emptyset) \in extrema(weight^*, \{y \mid O(x,y)\})$$

As we assume (1) and as $I(x) \ \wedge \ Get\_E(x) \cap \emptyset = \emptyset \ \wedge \ O(x, \emptyset)$ holds, we have :

$$\begin{array}{ll} & F_G(x, Get\_E(x), \emptyset) = \emptyset \cup z \ \wedge \\ & \quad z \in extrema(weight^*, \{y \mid O(x, \emptyset \cup y) \ \wedge \ y \subseteq Get\_E(x) \ \}) \\ \Longleftrightarrow & \quad \forall s : s \cup \emptyset = s \\ & F_G(x, Get\_E(x), \emptyset) = z \ \wedge \\ & \quad z \in extrema(weight^*, \{y \mid O(x,y) \ \wedge \ y \subseteq Get\_E(x) \ \}) \\ \Longleftrightarrow & \quad \text{By the first axiom: } O(x,y) \ \wedge \ y \subseteq Get\_E(x) \Longleftrightarrow O(x,y) \\ & F_G(x, Get\_E(x), \emptyset) \in extrema(weight^*, \{y \mid O(x,y) \ \}) \end{array}$$

To establish the consistency of F_G, we shall split the proof into three cases. The F_G program is indeed doing a different action in three disjoint cases. With each of these cases, we associate below the body that is indeed executed:

$$
\begin{array}{lll}
1 & GE = \emptyset & \longrightarrow \quad F_G(x', GE, solu) = solu \\
2 & GE \neq \emptyset \ \wedge \ \neg O(x', solu \cup \{lc\}) & \longrightarrow \quad F_G(x', GE, solu) = \\
 & & \qquad F_G(x', GE \setminus \{lc\}, solu) \\
3 & GE \neq \emptyset \ \wedge \ O(x', solu \cup \{lc\}) & \longrightarrow \quad F_G(x', GE, solu) = \\
 & & F_G(x', GE \setminus \{lc\}, solu \cup \{lc\})
\end{array}
$$

The proof will be made by induction on the number (n) of elements of the second argument of F_G (i.e. GE).

1 Base case: $n = 0$

It corresponds to the first case of our three cases proof.
We have to show that if we assume $GE = \emptyset$ as well as the antecedent in (1), the consistency is fulfilled:

$$
\begin{array}{ll}
 & F_G(x', \emptyset, solu) = solu \cup z \ \wedge \\
 & \qquad z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq \emptyset \ \}) \\
\Longleftrightarrow & \quad \forall s : s \subseteq \emptyset \Longleftrightarrow s = \emptyset \\
 & F_G(x', \emptyset, solu) = solu \cup z \ \wedge \\
 & \qquad z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y = \emptyset \ \}) \\
\Longrightarrow & \quad \text{By the antecedent of (1), } O(x', solu) \text{ holds} \\
 & F_G(x', \emptyset, solu) = solu \cup z \ \wedge \ z \in extrema(weight^*, \{\emptyset\}) \\
\Longrightarrow & \quad \text{By definition of } extrema \\
 & F_G(x', \emptyset, solu) = solu \cup z \ \wedge \ z \in \{\emptyset\} \\
\Longleftrightarrow & \quad \forall s, s' : s \in \{s'\} \iff s = s' \\
 & F_G(x', \emptyset, solu) = solu
\end{array}
$$

2 Induction case: $n > 0$

This case correspond to the last two cases of our three cases proof. We shall assume that F_G is consistent when $|GE| = n - 1$. Once again, this case splits into 2 subcases, whether $O(x', solu \cup \{lc\})$ holds or not.

2.1 $\neg O(x', solu \cup \{lc\})$ holds.

By definition of F_G, proving:

$$F_G(x', GE, solu) = solu \cup z \ \wedge \\ z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \ \})$$

is equivalent to proving:

$$F_G(x', GE \setminus \{lc\}, solu) = solu \cup z \ \wedge \\ z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \ \})$$

By indcution hypothesis, we know that:

$$F_G(x', GE \setminus \{lc\}, solu) = solu \cup z \ \wedge \\ z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \setminus \{lc\} \ \})$$

Thus if we prove:

$$\{y \mid O(x', solu \cup y) \wedge y \subseteq GE \setminus \{lc\} \ \} = \{y \mid O(x', solu \cup y) \wedge y \subseteq GE \ \} \quad (2)$$

then, we shall have proved the consistency of F_G in case 2.1.

$$\{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \ \}$$

$=$ Split into two cases whether $lc \in y$

$$\{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \setminus \{lc\} \ \} \\ \bigcup \\ \{y' \mid O(x', solu \cup \{lc\} \cup y') \ \wedge \ y' \subseteq GE \setminus \{lc\} \ \}$$

$=$ By the contrapositive of heredity axiom and $\neg O(x', solu \cup \{lc\})$

$$\{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \setminus \{lc\} \ \} \bigcup \emptyset$$

$=$ $\forall s : s \cup \emptyset = s$

$$\{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \setminus \{lc\} \ \}$$

2.2 $O(x', solu \cup \{lc\})$ holds.

By definition of F_G, proving:

$$F_G(x', GE, solu) = solu \cup z \ \wedge \\ z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \ \})$$

is equivalent to proving:

$$F_G(x', GE \setminus \{lc\}, solu \cup \{lc\}) = solu \cup z \ \wedge \\ z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \ \})$$

By indcution hypothesis, we know that:

$$F_G(x', GE \setminus \{lc\}, solu \cup \{lc\}) = solu \cup \{lc\} \cup z \ \wedge$$

$$z \in extrema(weight^*, \{y \mid O(x', solu \cup \{lc\} \cup y) \ \wedge \ y \subseteq GE \setminus \{lc\}\})$$

Let's first prove that:

$$\exists w \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE\}) : lc \in w \qquad (3)$$

Let's suppose the opposite of this thesis:

$$\forall w \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE\}) : lc \notin w$$

Let us note by: $\{e_1, \ldots, e_m\}$ an element w satisfying:

$$w \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE\})$$

We can apply the augmentation axiom for the two follwing sets: $solu \cup \{lc\}$ and $solu \cup \{e_1, e_2\}$ to find a set $solu \cup \{lc, e_i\}$ (with i=1 or 2) such that $O(x', solu \cup \{lc, e_i\})$ holds. And we can iteratively apply the augmentation axiom to get, after $m-1$ applications, a set $solu \cup w \cup \{lc\} \setminus \{e_j\}$ (with $1 \leq j \leq m$) which satifies $O(x', solu \cup w \cup \{lc\} \setminus \{e_j\})$ and $w \cup \{lc\} \setminus \{e_j\} \subseteq GE$ and such that $weight^*(w \cup \{lc\} \setminus \{e_j\}) \geq weight^*(w)$ by definition of lc. Therefore, we have constructed an element of the set:

$$extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE\})$$

and we have a contradiction with the opposite of your thesis.
Thus, we have the following:

$$extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE\})$$

$=$ Split into two cases whether $lc \in y$ and definition of $extrema$

$$extrema(weight^*, \; extrema(weight^*, \{y' \cup \{lc\} \mid O(x', solu \cup \{lc\} \cup y') \wedge y' \subseteq GE \setminus \{lc\}\}) \; \bigcup \; extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE \setminus \{lc\}\}))$$

$\supseteq$ By the proof of (3) and by the fact that $weight$ is positive

$$extrema(weight^*, \{y' \cup \{lc\} \mid O(x', solu \cup \{lc\} \cup y') \wedge y' \subseteq GE \setminus \{lc\}\})$$

Therefore, your induction hypothesis implies your thesis:

$$F_G(x', GE \setminus \{lc\}, solu \cup \{lc\}) = \ solu \cup \{lc\} \cup z' \ \wedge$$

$$z' \in extrema(weight^*, \{y' \mid O(x', solu \cup \{lc\} \cup y') \ \wedge \ y' \subseteq GE \setminus \{lc\}\})$$

$\Longrightarrow$ By the previous inclusion and by rewriting $\{lc\} \cup z'$ as z

$$F_G(x', GE, solu) = solu \cup z \ \wedge \\ z \in extrema(weight^*, \{y \mid O(x', solu \cup y) \ \wedge \ y \subseteq GE\})$$

□

We already mentioned (see p.19) that, if the function *weight* is positive, then a solution of the problem of finding an heaviest maximal independent set is always also a solution of the problem of finding an heaviest independent set. Let us notice that theorem 2 as well as all the examples that we shall present in Section 4.4 are formalized as finding an heaviest independent set (which we know that will also be a basis as *weight* is always constrainted to be positive). This was done for clarity reasons. However, we could obviously have a similar theorem (and similar examples) in which we **explicitely** mention that we are looking for an heaviest (or lightest) basis in which case, the *weight* function don't have to be positive any more.

4.3 The synthesis of greedy programs

The purpose of this section is to show how it is possible to use the algorithm theory developped in the previous section in order to help the synthesis of greedy programs from formal specifications of problems underlain by the structure of a matroid.
Smith explained how this purpose can be achieved [Smi92, Smi93]. We shall shortly explain his method and we recommend the interested reader to refer to [Smi93] for further details.
The method is based on category theory [BW90, Poi92]. Its basic idea is represented in Figure 6. In that graph, the nodes and the arcs represent respectively theories and morphisms. A morphism translates the language (i.e. the sorts and the operations) of one theory into the language of another theory in such a way that it preserves the theorems (and thus all the axioms). Let us make the remark that the identity morphism is always a possible choice between a theory and another one which is one of its extension[16].

[16] An extension of a theory contains at least all its sorts, operations and axioms and might contain others.

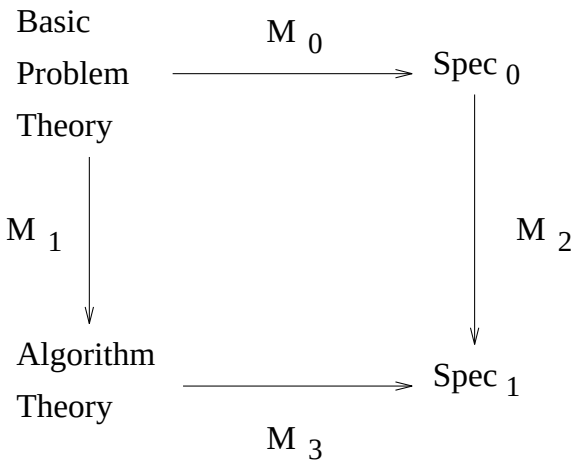

Figure 6: Program synthesis.

Figure 6 depicts a situation in which, we want to apply a method (contained in the algorithm theory) to a particular problem specified in $Spec_0$.
As $Basic - Problem - Theory$ (see Section 4) contains only what all specifications should contain. The two morphisms M_0 and M_1 are trivial: D, R, I and O are simply translated to their obvious equivalent in the two other specifications: $Spec_0$ and the algorithm theory.
The synthesis process would be finished if we find a theory ($Spec_1$) satisfying the two following conditions:

- it is an extension of the orginial specification ($Spec_0$) and moreover it uses only objects from $Spec_0$;
- there is a morphism (M_3) between the algorithm theory and $Spec_1$.

Indeed, finding such a theory means that we would have classified the problem represented by $Spec_0$ as an instance of the algorithm theory.
The synthesis is achieved by a two step procedure:

1. we first create equivalent classes for all the objects (sort or operation) which are the translation of the same initial object by the transitive closure of the morphisms. This is done by constructing the pushout of

the diagram containing the three theories $Basic-Problem-Theory$, algorithm theory and $Spec_0$ as well as the two morphisms M_0 and M_1 [Smi93];

2. we then construct a morphism from the pushout obtained in the first step to $Spec_1$ which contains expressions using objects only from $Spec_0$. Thus, all the symbols of the algorithm theory are expressed using objects from $Spec_0$.

Thanks to the morphism M_3, we know that $Spec_1$ satisfies the translation of the axioms of the algorithm theory. Thus, as M_3 translates the objects of the algorithm theory in the language of $Spec_0$, we can instanciate the algorithm associated to the algorithm theory in order to obtain a program expressed in the language of the intial specification and following the algorithm of the algorithm theory.
In [Smi93], Smith explains how it is possible to do the synthesis process incrementally while taking advantage of its hierarchy of algorithm theories. It consists of applying iteratively the same method as we did where $Basic-Problem-Theory$, $Spec_0$ and algorithm theory become respectively the algorithm theory, $Spec_1$ and a more specific algorithm theory (an extension of the previous one).
In order to carry out the second step of the procedure presented above, we first express the pushout (the result of the first step) as a system of equations and then solve it by treating the symbols of the algorithm theory as variables to which some expressions of $Spec_0$ will be assigned. In [Smi92], Smith studied different basic techniques to construct morphisms (e.g. unskolemization, connections,...).
As we shall use unskolemization in the sequel, we briefly mention how it is defined in [Smi92]. First, let us recall that skolemization is the process of replacing an existentially quantified variable z by a Skolem function over the universally quantified variables whose scope includes z. Unskolemization is the exact inverse process: given a formula containing identical simple occurences of operation symbol g, say $g(x)$, g is replaced by a fresh existentially quantified variable in the scope of x.
Let us consider the two following formulas:

$$\exists(s,t)\ \forall(u,v)\ \exists(w)\ \forall(x)\ \exists(y)\ :\ P(s,t,u,v,w,x,y) \tag{4}$$

$$\exists(s)\ \forall(u,v)\ \forall(x)\ :\ P(s,b,u,v,f(u,v),x,g(u,v,x)) \tag{5}$$

From formula 4 to formula 5, we skolemized the variables t, w and y and conversely from formula 5 to formula 4, we unskolemized b, f and g.

4.4 Examples

4.4.1 Heaviest Forest Problem

In this section, we are going to show how to use the theory developped in Sections 4.3 and 4.2 to the synthesis of a program solving the heaviest forest problem (see Example 1).
The first task to achieve is to specify the problem we want to solve. In order to make the following developments easier, we specify the problem by giving the translation of the sorts D, R and the operations I, O and *cost* of the $Basic - Optimization - Problem - Theory$ (see Figure 7).
Let us first give some remarks about some notations used in this specification that will also be valid for the sequel of this report:

- we use angle brackets ($\langle$ and $rangle$) to denote a tuple;
- we denote by e_i the ith projection of the tuple e and therefore e is equivalently noted $\langle e_1, \ldots, e_n \rangle$;
- we shall often use the only element of a 1-uple to denote that 1-uple: x will denote $\langle x \rangle$.

The domain D is represented as a set of triples, each of which containing three data: the nodes of the graph, the arcs and their weight. The range R is the set of arcs. I contains the type constraints associated with the elements of the domain D. In the expression of O, the first predicate ($nocircuit(S)$) expresses the fact that the set S of arcs don't contain any circuit and the second predicate $nopath(S, a, b)$ means that there is no path in S linking a and b. We think that defining O in a predicative recursive way makes the following easier.

[17] In this recursive definition as well as in all the following ones, we shall suppose that $e \notin S$

$$
\begin{array}{lll}
\boldsymbol{D} & \mapsto & \{\langle N, A, co\rangle \mid \ N \in set(Node) \ \wedge \ A \in set(Node \times Node) \ \wedge \\
 & & \qquad co \in map(Node \times Node, \mathbb{R}^+)\} \\
\boldsymbol{R} & \mapsto & set(Node \times Node) \\
\boldsymbol{I} & \mapsto & \lambda\langle N, A, co\rangle.(\forall a \in A : a_1 \in N \ \wedge a_2 \in N) \ \wedge \ domain(co) = A \\
\boldsymbol{O} & \mapsto & \lambda\langle N, A, co\rangle, A'.nocircuit(A') \ \wedge \ A' \subseteq A
\end{array}
$$

$$
\begin{array}{lcl}
nocircuit(\emptyset) & = & true \\
nocircuit(\{e\}) & = & true \\
nocircuit^{17}(S \cup \{e\}) & = & nocircuit(S) \ \wedge \ nopath(S, e_1, e_2) \\
 & & \\
nopath(\emptyset, a, b) & = & true \\
nopath(\{e\}, a, b) & = & \{e_1, e_2\} \neq \{a, b\} \\
nopath(S \cup \{e\}, a, b) & = & nopath(S, a, b) \ \wedge \\
 & & nopath(\{e\}, a, b) \ \wedge \\
 & & e_1 = a \implies nopath(S, e_2, b) \ \wedge \\
 & & e_1 = b \implies nopath(S, e_2, a) \ \wedge \\
 & & e_2 = a \implies nopath(S, e_1, b) \ \wedge \\
 & & e_2 = b \implies nopath(S, e_1, a)
\end{array}
$$

$$
\begin{array}{lll}
\boldsymbol{cost} & \mapsto & \lambda\langle N, A, co\rangle, A'.co(A') \\
 & & \\
\boldsymbol{Get}_E & \mapsto & \lambda\langle N, A, co\rangle.A
\end{array}
$$

Figure 7: Specification of the heaviest forest problem classified as an instance of the *Greedy Algorithms* class.

The next task consists of finding an extension of this specification which is such that there is a morphism between the *Greedy Algorithms* theory and the extension. We show in Figure 8 how the theory developped in Section 4.3 applies to the heaviest forest problem.
In Figure 8, we notice that the translation of D, R, I, O and *cost* in the *Greedy Algorithms* theory and in the heaviest forest problem can easily be unified.
After this unification, there are two more steps that have to be carried out in order to achieve the classification of the problem as an instance of the algorithm theory:

- to find a translation of all the other untranslated symbols of the al-

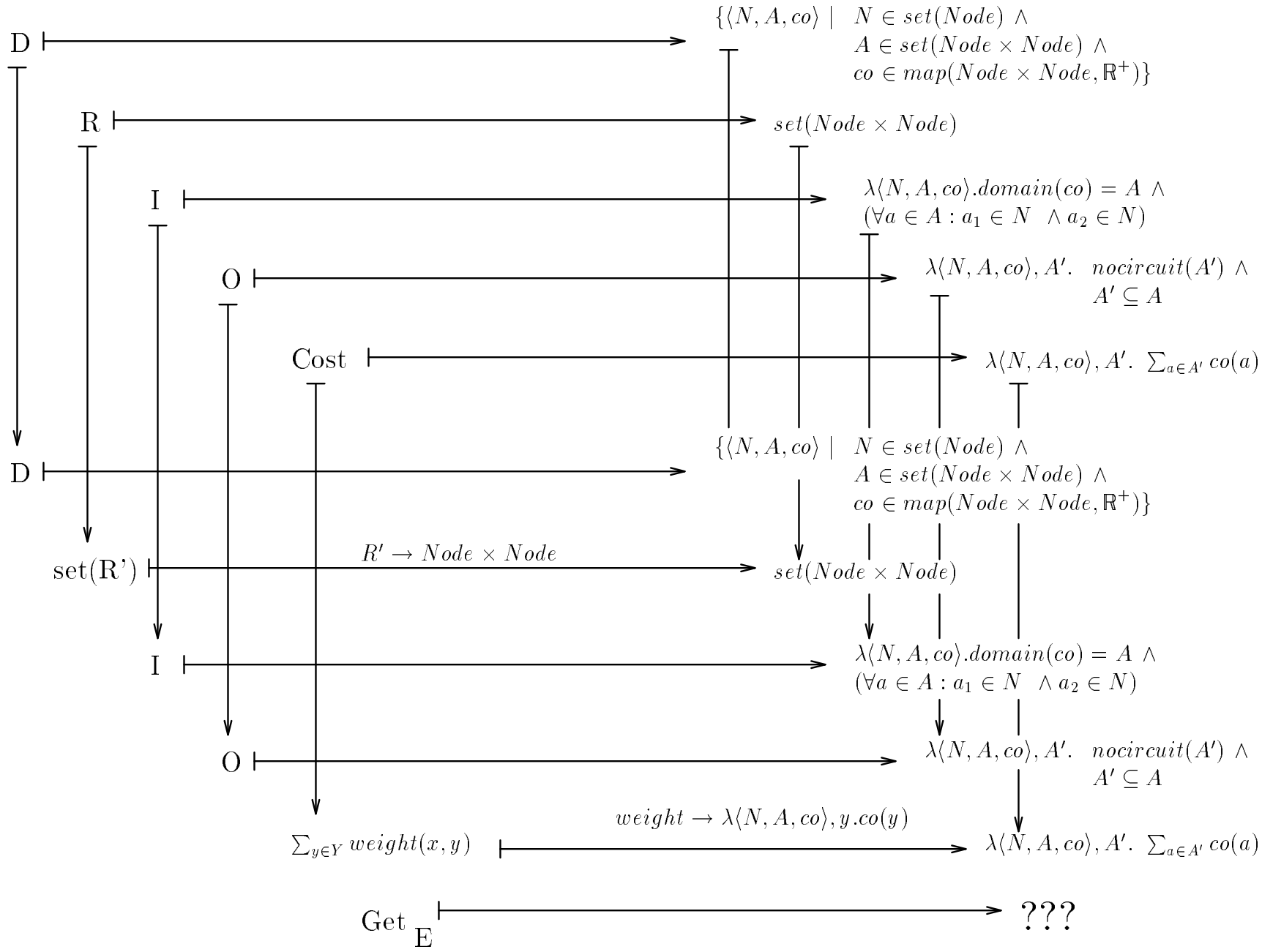

Figure 8 : Classifying the HFP as a GA.

gorithm theory (there is only Get_E involved in the case of the *Greedy Algorithms* theory);

- to verify that the axioms of the algorithm theory are indeed translated to theorems in the theory of our problem.

We shall start with the second step and we shall notice that the first one will be solved as a side effect.
In order to shorten the following expressions, we denote by starred symbols (resp. expressions) of the *Greedy Algorithms* theory, the translation of those symbols (resp. expressions) into the problem theory.

1. The first axiom:

   $$\forall(x : D), \forall(y : set(R')) \quad I(x) \implies [O(x, y) \implies y \subseteq Get\_E(x)]$$

   is equivalent to:

   $$\forall(x : D), \forall(y : set(R')) \quad [I(x) \wedge O(x, y)] \implies y \subseteq Get\_E(x)$$

   and can be translated to:

   $$\forall(x : D^*), \forall(y : set(Node \times Node)) \\ [I^*(x) \wedge y \subseteq A \wedge nocircuit(y)] \implies y \subseteq Get\_E^*(x)$$

   in which A stands for x_2 and $Get_E^*(x)$ is unknown (see step 1).

   Using the unskolemization (see p. 4.3), we get:

   $$\forall(x : D^*), \exists(g : set(Node \times Node)), \forall(y : set(Node \times Node)) \\ [I^*(x) \wedge y \subseteq A \wedge nocircuit(y)] \implies y \subseteq g$$

   And this axiom is trivially satisfied by choosing $g = A$. Thus, thanks to this process, we have killed two birds with one stone: we have found a translation for $Get_E(x)$ and we have verified that the first axiom was indeed translated to a theorem. In fact, it would be more accurate to say that we chose the translation of $Get_E(x)$ in order to fulfill the translation of the axiom.

2. The trivial axiom:

   $$\forall(x : D) \quad I(x) \implies O(x, \emptyset)$$

is translated to:

$$\forall(x : D^*) \quad I^*(x) \implies \emptyset \subseteq A \ \wedge \ nocircuit(\emptyset)$$

which is trivially satisfied by the definition of $nocircuit$.

3. The heredity axiom:

   $$\forall(x : D)\,\forall(y, z : set(R'))\ \ I(x) \implies [z \subseteq y \implies (O(x, y) \implies O(x, z))]$$

   is translated to:

   $$\forall(x : D^*)\ \forall(y, z : set(Node \times Node)) \quad I^*(x) \implies$$
   $$[z \subseteq y \ \wedge \ y \subseteq A \ \wedge \ nocircuit(y)] \implies [z \subseteq A \ \wedge \ nocircuit(z)]$$

   which is trivially satisfied by transitivity and by the definition of $nocircuit$.

4. The translation of the augmentation axiom or of the cardinality axiom is not easy to verify. There are both correct as it is proved in many textbooks such as in [CLR90]. However, their proof use some extra notions about graph theory and look very much domain dependent. Thus the proof will be omitted.

In the two following examples, we shall see that it is possible to find some examples for which the augmentation axiom (or the cardinality axiom) can be proved using domain independent techniques.
Let us note that, in order to get a greedy program that solves the heaviest forest problem, we only have to instanciate the symbols D, R, I, O and Get_E that appear in the program scheme of theorem 2 by their translations we found. For clarity reasons, we shall not carry out this obvious instanciation for this example as well as for the following.

4.4.2 A local feature example

Given a directed graph $D = (G, A)$ and a weight for each $a \in A$, we want to find the heaviest B such that $B \subseteq A$ and no two arcs of B have the same head. At first sight, this problem is very similar to the matching problem. However, it is underlain by a matroid and the matching problem is not (see Section 2).

We first specify this problem as it is done in Figure 9. Then, we try to unify the translations of the same symbol in *Greedy Algorithms* and in this problem theory (this step is exactly the same than the one done for the heaviest forest problem). Eventually, we verify that the four axioms of the *Greedy Algorithms* theory are translated to theorems and find at the same time a translation for $Get_E(x)$.

$$
\begin{array}{lcl}
\boldsymbol{D} & \mapsto & \{\langle N, A, co\rangle \mid \ N \in set(Node) \ \wedge \ A \in set(Node \times Node) \ \wedge \\
 & & \qquad co \in map(Node \times Node, \mathbb{R}^+)\} \\
\boldsymbol{R} & \mapsto & set(Node \times Node) \\
\boldsymbol{I} & \mapsto & \lambda\langle N, A, co\rangle.(\forall a \in A : a_1 \in N \ \wedge a_2 \in N) \ \wedge \ domain(co) = A \\
\boldsymbol{O} & \mapsto & \lambda\langle N, A, co\rangle, A'.notwosamehead(A') \ \wedge \ A' \subseteq A \\
 & & \begin{array}{lcl}
notwosamehead(\emptyset) & = & true \\
notwosamehead(\{e\}) & = & true \\
notwosamehead(S \cup \{e\}) & = & notwosamehead(S) \wedge \\
 & & \qquad noonehead(S, e_2) \\
 & & \\
noonehead(\emptyset, a) & = & true \\
noonehead(\{e\}, a) & = & e_2 \neq a \\
noonehead(S \cup \{e\}, a) & = & noonehead(S, a) \wedge \\
 & & \qquad noonehead(\{e\}, a)
\end{array} \\
\boldsymbol{cost} & \mapsto & \lambda\langle N, A, co\rangle, A'.co(A') \\
 & & \\
\boldsymbol{Get_E} & \mapsto & \lambda\langle N, A, co\rangle.A
\end{array}
$$

Figure 9: Specification of the local feature example classified as an instance of the *Greedy Algorithms* class.

1. The first axiom:

   $$\forall(x : D), \forall(y : set(R')) \quad I(x) \implies [O(x, y) \implies y \subseteq Get\_E(x)]$$

   is translated to:

   $$\forall(x : D^*), \forall(y : set(Node \times Node)) \\ [I^*(x) \ \wedge \ y \subseteq A \ \wedge \ notwosamehead(y)] \implies y \subseteq Get\_E^*(x)$$

   Using once again the unskolemization technique, we get:

$$\forall(x : D^*), \exists(g : set(Node \times Node)), \forall(y : set(Node \times Node))$$
$$[I^*(x) \ \wedge \ y \subseteq A \ \wedge \ notwosamehead(y)] \implies y \subseteq g$$

And this axiom is trivially satisfied by choosing $g = A$.

2. The trivial axiom:

$$\forall(x : D) \quad I(x) \implies O(x, \emptyset)$$

is translated to:

$$\forall(x : D^*) \quad I^*(x) \implies \emptyset \subseteq A \ \wedge \ notwosamehead(\emptyset)$$

which is trivially satisfied by the definition of *notwosamehead*.

3. The heredity axiom:

$$\forall(x : D)\forall(y, z : set(R')) \quad I(x) \implies [z \subseteq y \implies (O(x, y) \implies O(x, z))]$$

is translated to:

$$\forall(x : D^*) \ \forall(y, z : set(Node \times Node)) \quad I^*(x) \implies$$
$$[z \subseteq y \ \wedge \ y \subseteq A \ \wedge \ notwosamehead(y)] \implies$$
$$[z \subseteq A \ \wedge \ notwosamehead(z)]$$

which is trivially satisfied by transitivity and by the definition of *notwosamehead*.

4. The augmentation axiom:

$$\forall(x : D) \ \forall(y, z : set(R')) \quad I(x) \implies$$
$$[O(x, y) \ \wedge \ O(x, z) \ \wedge \ |y| + 1 = |z|) \implies$$
$$(\exists(e : R') \ e \in z \setminus y \ : \ O(x, y \cup \{e\})]$$

is translated to:

$$\forall(x : D^*) \ \forall(y, z : set(Node \times Node)) \quad I^*(x) \implies$$
$$[\ (\quad y \subseteq A \ \wedge \ notwosamehead(y) \quad \wedge$$
$$z \subseteq A \ \wedge \ notwosamehead(z) \quad \wedge |y| + 1 = |z|) \implies$$
$$(\exists(e : Node \times Node) \ e \in z \setminus y \ : \ y \cup \{e\} \subseteq A \wedge notwosamehead(y \cup \{e\}))]$$

The part $y \cup \{e\} \subseteq A$ of the consequent is trivially satisfied by the

antecedent and by choosing e such that $e \in z \setminus y$. The consequent we still have to prove thus becomes:

$$(\exists e : Node \times Node)\ e \in z \setminus y \ : \ notwosamehead(y \cup \{e\})$$

which reduces, by the antecedent of the augmentation axiom, to:

$$(\exists e : Node \times Node)\ e \in z \setminus y \ : \ noonehead(y, e_2) \qquad (6)$$

In order to prove 6, we rewrite the two predicates used to define the output condition O as follows:

$$\begin{array}{lcl} notwosamehead(y) & \Longleftrightarrow & |\{a_2 | a \in y\}| = |y| \\ noonehead(y, e) & \Longleftrightarrow & e_2 \notin \{a_2 | a \in y\} \end{array}$$

Therefore, the following conjunction from the antecedent of the augmentation axiom:

$$notwosamehead(y) \ \wedge \ notwosamehead(z) \ \wedge \ |y| + 1 = |z|$$

is translated to:

$$|\{a_2 | a \in y\}| + 1 = |\{a_2 | a \in z\}|$$

and thus the formula:

$$\exists e \in z \setminus y : e_2 \notin \{a_2 | a \in y\}$$

is satisfied and we are done because this formula is equivalent to 6.

This example shows that the fourth axiom is easily proved whenever we tackle a problem whose underlying matroid is such that an independent subset can be characterized by a feature that has to be different for all its elements. When this is possible, we always know that there is an element in z for which no element in y has the same feature as $|z| = |y| + 1$. So that element can be added to y while keeping the fact that the feature is different for all the elements in the new set. In the previous example, that feature was the head of an arc. The reason why we call this kind of problems a local feature problem comes from the fact that in order to have a feature which is different for all the elements of any subset, the feature has to have something to do with the element in itself (which is local) and not with the whole subset (which is global).

4.4.3 Sequencing Problem

The sequencing problem can be expressed as follows:

> A certain number of jobs have to be processed by a single machine. All the jobs require one hour processing time. To each job i is assigned a deadline d_i and a penalty w_i which must be paid if the job is not completed by its deadline. What ordering of the jobs minimizes the total penalty costs ?

We simplify the sequencing problem by trying to find among the sets of jobs for which there is an ordering such that all the jobs can be processed by their deadline the one that maximizes the total penalty costs. If we have a solution for this simplified problem, we can easily find a solution for the initial problem. Indeed, it's easy to prove that when a set of jobs is such that there exists an ordering for which all the jobs can be processed by their deadline, the schedule of the jobs in order of nondecreasing deadline is such an ordering [CLR90]. So once we have a solution of the simplified problem, we find a solution of the initial one by sorting the jobs in order of nondecreasing deadline and then concatenating in any order to that sequence all the jobs that were not contained in the initial solution (i.e. all the jobs which are not completed by their deadline).
Moreover, it is obviously equivalent to minimize the total penalty costs or to maximize the total penalty costs of the set of jobs that can be processed by their deadline.
We specify the simplified problem in Figure 10.
The unification step leads to the two following assignments:

$$\begin{array}{lcl} R' & \rightarrow & \mathbb{N} \\ weight & \rightarrow & \lambda\langle n, d, w\rangle, y.w(y) \end{array}$$

Next we verify the axioms:

1. The first axiom:

   $$\forall(x : D), \forall(y : set(R')) \quad I(x) \implies [O(x, y) \implies y \subseteq Get\_E(x)]$$

   which is translated to:

   $\forall(x : D^*), \forall(y : set(\mathbb{N}))$

$$
\begin{array}{lcl}
\boldsymbol{D} & \mapsto & \{\langle n, d, co\rangle \mid n \in \mathbb{N} \ \wedge \ d \in map(\mathbb{N}, \mathbb{N}) \ \wedge \ w \in map(\mathbb{N}, \mathbb{R}^+)\} \\
\boldsymbol{R} & \mapsto & set(\mathbb{N}) \\
\boldsymbol{I} & \mapsto & \lambda\langle n, d, w\rangle.n \geq 1 \ \wedge \ domain(d) = \{1, ..., n\} \ \wedge \\
 & & \qquad\qquad domain(w) = \{1, ..., n\} \\
\boldsymbol{O} & \mapsto & \lambda\langle n, d, w\rangle, Y.Y \subseteq \{1, ..., n\} \ \wedge \ processible(d, Y) \\
 & & \begin{array}{lcl}
processible(d, \emptyset) & = & true \\
processible(d, \{e\}) & = & d(e) > 0 \\
processible(d, S \cup \{e\}) & = & processible(d, S) \ \wedge \\
 & & \forall j \in \{d(e), ..., Max(d, e, S)\} : \\
 & & |\{s \mid s \in S \ \wedge \ d(s) \leq j\}| < j \\
Max(d, e, S) & = & max(d(e), max\{d(e') \mid e' \in S\})
\end{array} \\
\boldsymbol{cost} & \mapsto & \lambda\langle n, d, w\rangle, y.w(y) \\
 & & \\
\boldsymbol{Get}_E & \mapsto & \lambda\langle n, d, w\rangle.\{1, ..., n\}
\end{array}
$$

Figure 10: Specification of the sequencing problem classified as an instance of the *Greedy Algorithms* class.

$$[I^*(x) \ \wedge \ y \subseteq \{1, ..., n\} \ \wedge \ processible(d, y)] \implies y \subseteq Get\_E^*(x)$$

Using once again the unskolemization technique, we get:

$$\forall(x : D^*), \exists(g : set(\mathbb{N})), \forall(y : set(\mathbb{N}))$$
$$[I^*(x) \ \wedge \ y \subseteq \{1, ..., n\} \ \wedge \ processible(d, y)] \implies y \subseteq g$$

And this axiom is trivially satisfied by choosing $g = \{1, ..., n\}$.

2. The trivial axiom:

$$\forall(x : D) \quad I(x) \implies O(x, \emptyset)$$

is translated to:

$$\forall(x : D^*) \quad I^*(x) \implies \emptyset \subseteq \{1, ..., n\} \ \wedge \ processible(d, \emptyset)$$

which is trivially satisfied by the definition of *processible*.

3. The heredity axiom:

$$\forall(x : D)\forall(y, z : set(R')) \quad I(x) \implies [z \subseteq y \implies (O(x, y) \implies O(x, z))]$$

is translated to:

$$\forall(x : D^*)\ \forall(y, z : set(\mathbb{N}))\quad I^*(x)\ \Longrightarrow$$
$$[z \subseteq y\ \wedge\ y \subseteq \{1, ..., n\}\ \wedge\ processible(d, y)]\ \Longrightarrow$$
$$z \subseteq \{1, ..., n\}\ \wedge\ processible(d, z)$$

which is trivially satisfied by transitivity and by the definition of *processible*.

4. The augmentation axiom:

$$\forall(x : D)\ \forall(y, z : set(R'))\quad I(x)\ \Longrightarrow$$
$$[O(x, y)\ \wedge\ O(x, z)\ \wedge\ |y| + 1 = |z|)\ \Longrightarrow$$
$$(\exists(e : R')\ e \in z \setminus y\ :\ O(x, y \cup \{e\})]$$

is translated to:

$$\forall(x : D^*)\ \forall(y, z : set(\mathbb{N}))\quad I^*(x)\ \Longrightarrow$$
$$[(\ \ y \subseteq \{1, ..., n\}\ \wedge\ processible(d, y)\quad \wedge$$
$$z \subseteq \{1, ..., n\}\ \wedge\ processible(d, z)\quad \wedge |y| + 1 = |z|)\ \Longrightarrow$$
$$(\exists(e : \mathbb{N})\ e \in z \setminus y\ :\ y \cup \{e\} \subseteq \{1, ..., n\}\ \wedge\ processible(d, y \cup \{e\}))]$$

The part $y \cup \{e\} \subseteq \{1, ..., n\}$ of the consequent is trivially satisfied by the antecedent and by choosing e such that $e \in z \setminus y$. The consequent we still have to prove thus becomes:

$$(\exists e : \mathbb{N})\ e \in z \setminus y\ :\ processible(d, y \cup \{e\})$$

which reduces, by the antecedent of the augmentation axiom, to:

$$(\exists e : \mathbb{N})\ e \in z \setminus y\ :$$
$$\forall j \in \{d(e), ..., Max(d, e, y)\}\ :\underbrace{|\{s \mid s \in y\ \wedge\ d(s) \leq j\}|}_{(a)} < j \quad (7)$$

with $Max(d, e, y) = max(d(e), max\{d(e') \mid e' \in y\})$

In order to prove 7, we choose e satisfying:

$$e \in highest(d, \{x | x \in z \setminus y\})$$

$processible(d, z)$, appearing in the antecedent of the augmentation axiom, implies:

$$\forall j \in \{d(e), ..., Max(d, e, z \setminus \{e\})\} \ : \underbrace{|\{s \mid s \in z \setminus \{e\} \ \wedge \ d(s) \leq j\}|}_{(b)} < j$$

Therefore, by definition of e and by the fact that $|z \setminus \{e\}| = |y|$, we have:

$$\forall j \in \{d(e), ..., Max(d, e, z \setminus \{e\})\} \ : (a) \leq (b) < j \qquad (8)$$

Moreover, if we notice that $Max(d, e, z \setminus \{e\}) = max\{d(e') \mid e' \in z\}$, then we have trivially:

$$\forall j \in \{Max(d, e, z \setminus \{e\}), ..., Max(d, e, y)\} \ : (a) \leq |z| - 1 = (b) < j \quad (9)$$

As a consequence of 8 and 9, the verification of the formula 7 is finished.

This example shows another general way to verify the augmentation axiom: whenever an e satisfying all the constraints can be easily described a priori, the verification often becomes much easier.

4.4.4 Lightest basis problem

This problem can be formulated as: given a matrix A with which is associated a weight for each of its column, let us find the heaviest linear independent subset of columns of A.
The specification of the problem is represented in Figure 11 in which we represented by $A(\cdot, i)$ the ith column of the matrix A and by $\{A'_1, \ldots, A'_{|A'|}\}$ the set A', second parameter of the output condition O.
In this example, the unification step as well as the verification of the translation of the first axiom as well as the trivial axiom can be done easily. However, the heredity and the augmentation axioms seem very hard to prove. This is due to two different reasons:

- the intrinsic difficulty of the problem in itself;
- the way the operation O was expressed. Indeed, O, as specified in Figure 11, is not directly computable as it was in the previous examples.

$$
\begin{array}{lcl}
\boldsymbol{D} & \mapsto & \{\langle m,n,A,co\rangle \mid \ m \in \mathbb{N} \ \wedge \ n \in \mathbb{N} \ \wedge \ A \in map(\mathbb{N}\times\mathbb{N},\mathbb{R}) \ \wedge \\
 & & \qquad co \in map(\mathbb{N},\mathbb{R}^+)\} \\
\boldsymbol{R} & \mapsto & set(map(\mathbb{N},\mathbb{R})) \\
\boldsymbol{I} & \mapsto & \lambda(m,n,A,co). \ \ m \geq 1 \wedge n \geq 1 \ \wedge \ \forall a \in A : a_1 \in \{1,...,m\} \ \wedge \\
 & & \qquad a_2 \in \{1,...,n\} \ \wedge \ domain(co) = \{1,...,n\} \ \wedge \\
 & & \qquad \nexists i,j \in \{1,...,n\} \ : \ i \neq j \ \wedge \ A(.,i) = A(.,j) \\
\boldsymbol{O} & \mapsto & \lambda(m,n,A,co),A'. \ \ A' \subseteq \{A(\cdot,i) | 1 \leq i \leq n\} \ \wedge \\
 & & \qquad \bigwedge_{j=1}^{m} \sum_{i=1}^{|A'|} \alpha_i A'_i(j) = 0 \iff \bigwedge_{k=1}^{|A'|} \alpha_k = 0 \\
\boldsymbol{cost} & \mapsto & \lambda(m,n,A,co),a.co(i) \text{ where } i \text{ such that } A(\cdot,i) = a \\
 & & \\
\boldsymbol{Get}_E & \mapsto & \lambda(m,n,A,co).\{A(\cdot,i) \mid 1 \leq i \leq n\}
\end{array}
$$

Figure 11: Specification of the lightest basis problem classified as an instance of the *Greedy Algorithms* class.

This last reason has another much more important consequence: the body of the greedy program associated to the algorithm theory (see p.28) makes an explicit reference to O in the condition of the second "if-then-else". Thus the program can only be executed if we know a way to compute O. This is the main reason why defining O in a predicative and recursive way very useful in the three previous examples. Another reason comes from the fact that it eases very much the verification of the trivial and heredity axioms.
When O is specified in a non processible way, a by-product of the synthesis of a greedy program consists of finding a program processing O. Therefore, it would be very nice to add this theory of greedy programs synthesis to an already existing environment which could allow the synthesis of programs of other classes. As we are close to Smith's work, we think that the best choice would be the KIDS[18] [Smi90] system (or a descendant) [BGG+94].
Even if the specification of O is processible, the use of KIDS would be very useful because it contains several tools which could be of great help during our synthesis process. For instance, finite differencing [PK82, Smi90] simplification and partial evaluation would be helpful in order to transform the expression $O(x', solu \cup \{lc\})$ from the greedy program code (see p.28) into

[18]Kestrel Interactive Development System.

more efficient expressions.
And last but not least, KIDS covers many different classes of algorithms but not yet the *Greedy Algorithms* class.
In conclusion, although we haven't synthesized a full program solving the lightest basis problem, the work we did in this section is not negligible: we synthesized a program solving the initial problem but which is not executable by the time we don't have a way to compute O. Our thought process could be seen in this case as a reduction of problems into simpler ones.

4.5 Conclusion

We have shown in the previous examples that our theory could help in the synthesis of greedy programs solving problems defined by a formal specification.
The synthesis process consists of four steps:

1. unifying the translations of the symbols of the Basic-Optimization-Problem-Theory in the *Greedy Algorithms* theory with the ones in the problem specification. In doing so, we consider the symbols of the *Greedy Algorithms* theory as variables to which we want to assign expressions of the problem specification;

2. finding a translation for all the untranslated symbols of the *Greedy Algorithms* theory;

3. verifying that the axioms of the *Greedy Algorithms* theory are indeed translated to theorems in the problem theory;

4. instanciating the symbols of the algorithm associated to the *Greedy Algorithms* theory by their translations found in the first two steps.

The first step was easy to achieve in all our examples. This is always the case when the specification of the problem is in a "good" form which means that:

- the output domain has to be $set(\cdots)$;

- the cost of a potential solution (a set of elements) has to be expressed has the sum of the costs of its elements.

In fact, we simplified the sequencing problem in order to satisfy this first condition. Indeed, the initial problem was looking for an object of type $seq(\cdots)$ which cannot be unified with $set(\cdots)$. We shall consider (see Section 6) another much more general alternative to solve such problems in which the output domain is different than $set(\cdots)$.
We solved the second step in our four examples by applying the unskolemization technique to the first axiom. As a consequence, the first axiom was trivially verified.
We have to admit that finding a translation for $Get_E(x)$ was greatly facilitated by the fact that the specification of O contains explicitly the obvious candidate for $Get_E(x)$. When such a candidate is not explicitly mentioned, we would have to infer it from $I(x) \wedge O(x, y)$. Once again, in such a case, a system like KIDS, having some tools to help such inference would be very useful.
In order to achieve the third step, we noticed that the trivial and heredity axioms are often easy to verify, which is not at all the case for the augmentation axiom or for the cardinality axiom. As we already said, the verification of the trivial and heredity axiom was facilitated by the fact that the main predicate in O was often defined recursively. Verifying the augmentation or the cardinality axiom seemed to be the most difficult step of the synthesis process. In fact, we didn't identify a general procedure to apply to all problems in order to verify any of the two axioms. However, we showed in Sections 4.4.2 and 4.4.3 that there exist some procedures that can be applied to several problems. If those procedures cannot be applied to a problem 4.4.1, then the verification has to be problem dependent.
Eventually, the fourth step is direct as soon as the first three are over.
Interesting considerations about the possible representations of a set system are done in [MS90]. Indeed, if we represent a set system by the collection of all the feasible sets, it has the major drawback to require a very large space. If we represent a set system by a *feasibility oracle* (i.e. a decision procedure which, given any subset, decides whether or not it is feasible), it's proved in [MS90] that there is no polynomial-time algorithm to decide whether or not an accessible set system is a matroid. However, in this report, we used a nice time/space compromise: we formalized a set system by a predicate expressing whether or not a subset is feasible. This predicate is similar to a feasibility oracle for the space complexity. However, in order to verify whether the set system is a matroid or not, we do not check the axioms for

all possible subsets but we try to infer from the expression of the predicate that the axioms are indeed correct.
Some interesting questions to investigate in the future would be:

- to find a domain independent proof for the augmentation/cardinality axiom of the heaviest forest problem of Section 4.4.1;
- to find as general procedures as possible to verify the augmentation axiom or the cardinality axiom in order to cover as many problems underlain by a matroid theory as possible;
- to find a subclass of the matroid theory in which the axioms would be easier to satisfy and thus programs easier to synthesize;
- to tackle the lightest basis problem by specifying it in another way;
- to look how it would be possible to tackle problems for which the cost of a subset is not expressed as the sum of the costs of its elements;
- to ponder on how to use and combine different algorithm classes in a synthesis process.

Finally, we want to say that the way the specification is expressed can greatly facilitate (or not) the synthesis process. This was already noticed when we wanted to find a translation for $Get_E(x)$ and also when we wanted to verify the axioms, we saw that it heavily depends on the way O was defined.
So, as we don't live in an ideal world, saying that: *"The closer to the theory the specification is, the easier the synthesis process is"* is far from being completely wrong.

5 First generalization of matroid theory

When we formalized the matroid theory (see Section 3.2), we mentioned that the heredity axiom was very strong. Therefore, an obvious generalization of matroid theory would come from the weakening of its axioms (the heredity axiom in particular). In the two following sections, we shall consider two different ways to do so.

5.1 Greedoids

Korte and Lovász [KL81, KL84] defined greedoids, a generalization of matroids, in the early eighties: a greedoid is a relaxation of a matroid for which the heredity axiom is replaced by the accessibility axiom.

Axiom 5 (Accessibility axiom)
$X \in \mathcal{F} \ \wedge \ X \neq \emptyset \Longrightarrow \exists x \in X : X \setminus \{x\} \in \mathcal{F}$

Definition 8
A set system satisfying both the trivial and accessibility axioms is called an **accessible** set system.

It's important to notice that the accessibility axiom is the weakest version of the heredity axiom that we can use in order to formalize an underlying theory to the *Greedy Algorithms* class. Indeed, the first condition of the greedy philosphy (see Section 2) was that a solution had to be constructed incrementally in a succession of stages. Obviously, if a set X doesn't satisfy the accessibility axiom, it cannot be constructed incrementally and it couldn't be found by the greedy algorithm.

Definition 9
An accessible set system satisfying the augmentation axiom (see Section 3) is called a **greedoid**.

Let us note that greedoids can be defined in different ways as it is explained in [KLS91]. We recommend all the interested readers to refer to [KLS91] to get some more information about greedoids. For instance, we can learn that if we define antimatroids by abstracting the combinatorial properties of convexity, as matroids were defined by abstracting the properties of linear independence, we surprisingly notice that antimatroids are also generalized by greedoids.

Example 4 (Heaviest Tree Problem)
The problem is to find an heaviest tree in a connected graph. A tree is a forest for which every pair of nodes of the tree is connected by a path belonging to the tree. The weight of a tree is defined as the sum of the weight of all the arcs contained in the tree. Let us note that this problem is very close to the heaviest forest problem (see Example 1). In fact, if the

graph is connected and if the weight of all the arcs is positive then an heaviest forest of a graph is obviously an heaviest tree, thus the two problems become equivalent. Therfore, both have often been called the maximum spanning tree problem. However, as the heaviest forest and the heaviest tree problems can solve different problems, we prefer to consider them separatly. Prim [Pri57] invented a greedy algorithm to tackle this problem:

- the final tree is obtained by adding, at each stage, an arc to the set of arcs previously chosen;
- at each stage, the arc added is heaviest among all the arcs which have not yet been considered, which do not create any circuit if they are added to the set of arcs previously chosen and which are connected to the graph composed of the set of arcs previously chosen.

Let us formalize the heaviest tree problem using the set system formalization. In this case, E would obviously represent the set of all the arcs of the graph. If we choose $\mathcal{F}$ as the collection of all the trees of that graph, $(E, \mathcal{F})$ is not a greedoid because the augmentation axiom is violated. Indeed, let us consider the graph represented in Figure 1 (see Section 2); if we take as X the two arcs weighting respectively 17 and 4 and as Y the arc weighting 2, the antecedent of the augmentation axiom (see p.12) is satisfied but the consequent is not. The solution is to change the $\mathcal{F}$ we chose into the collection of all the trees of the graph which all contains a given node. With such a definition, the trivial, accessible and augmentation axioms are all easily proved and this problem is thus underlain by a greedoid.
This choice of a particular node of reference in order to define $\mathcal{F}$ may seem rather arbitrary. However, in Section 5.2, we shall see another generalization in which the choice of a collection of all the trees of the graph for $\mathcal{F}$ could be valid.
Figure 12 represents the different stages of Prim's algorithm (found in 1957) over the graph of Figure 1 in which v_1 is the node that has been chosen to be contained in any of the elements (trees) of $\mathcal{F}$. As the intitial graph is connected and as the weight of all its arcs is positive, the heaviest forest found by Kruskal's algorithm (see Figure 2) and the heaviest tree found by Prim's have exactly the same weight[19].

[19] The reason why both results are identical comes from the fact that there's only one heaviest forest/tree.

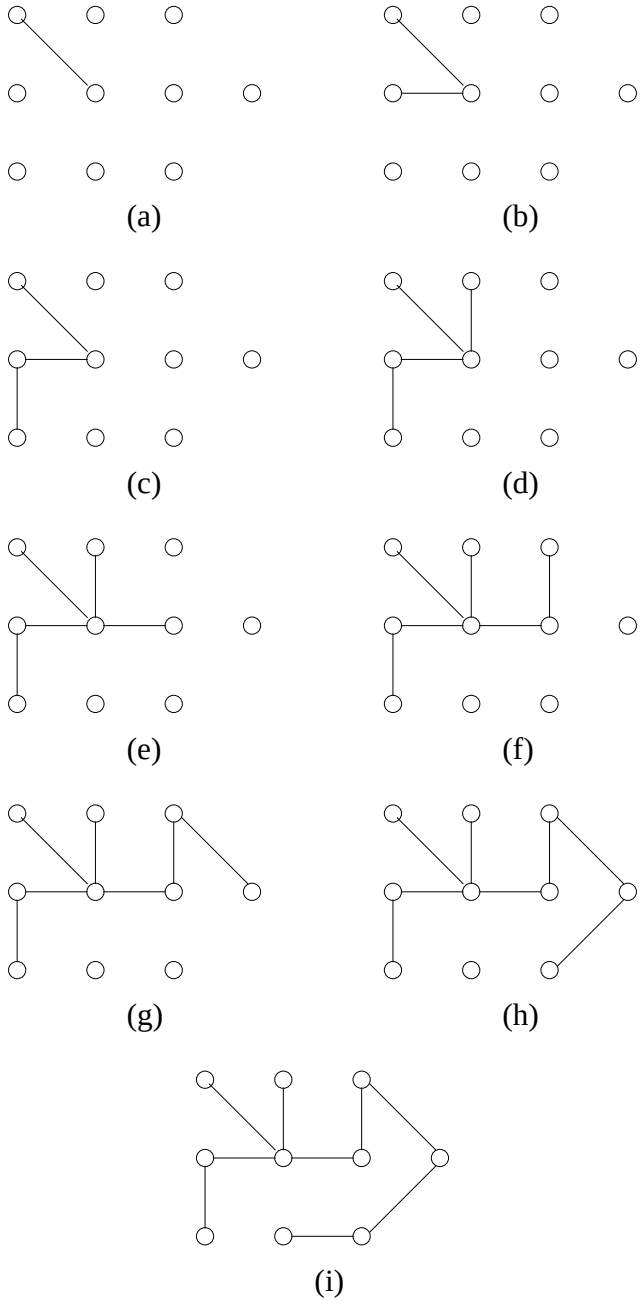

Figure 12: Stages of Prim's algorithm.

Let us notice that Prim's algorithm is nothing else but an instanciation of the first greedy algorithm (see p.16). Let us also recall that we cannot use in this case the second greedy algorithm (see p.17) because it supposed that the heredity axiom was fulfilled which is not the case here.
We represented in Figure 5 all possible executions of the second greedy algorithm over a set of four elements: $\{e_1, e_2, e_3, e_4\}$ whose weight order was $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$. Similarly, we represent in Figure 13 all possible exe-

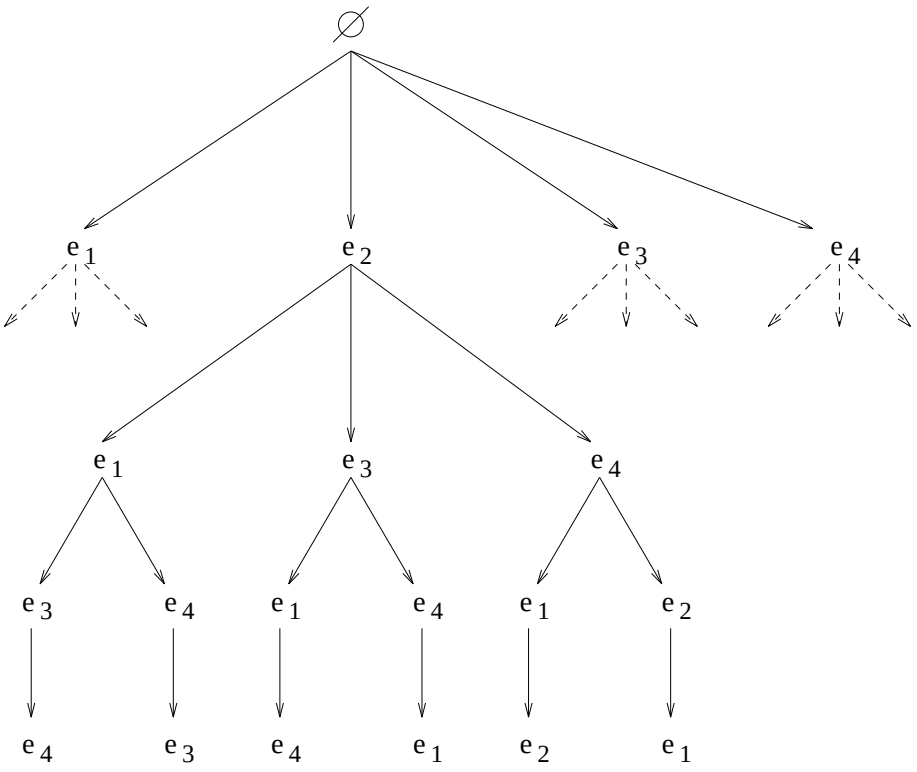

Figure 13: Representation of the execution of a greedy algorithm.

cutions of the *general* (i.e. the first) greedy algorithm over the same set of elements. All the comments already done in order to understand Figure 5 (see p.19) are still valid here. In Figure 13, we meant by dotted arrows the fact that there exist subgraphs which are not drawn in the figure by lack of place. Indeed, the subgraph of e_1 (resp. e_3, e_4) was omitted but is exactly

the same than the one of e_2 in which all occurences of e_1 (resp. e_3, e_4) are replaced by e_2. The great difference between Figure 13 and Figure 5 is that in the former, at any node of the tree, its children are all the elements but the ones which are already in the partial solution (see p.18) associated to that node. On the other hand, in the latter, as soon as one node was considered and not chosen, it didn't appear any more in the subgraphs of its right brothers. It's interesting to compare in the two figures the subgraph of element e_2 of the first level of the tree and to notice the drastic difference of its size[20]. At the look of those two figures, it is clear why the performance of the second greedy algorithm was far better than the one of the first. However, as we shall see in the following, even the general algorithm does not solve all optimization problems associated to greedoids.
Although the greedy algorithm works on the particular example of the heaviest tree problem, it is not a general fact. Indeed, in [KLS91], an exact characterization of greedoids is given: a set system is a greedoid if and only if the greedy algorithm maximizes a certain class of weight functions. Unfortunately, this class does not include linear functions. However, in [KLS91], the linear functions have been analyzed and it has been proved that the greedy algorithm optimizes all linear weight function over a greedoid $(E, \mathcal{F})$ if and only if $(E, \mathcal{F})$ obeys to the strong exchange axiom.

Axiom 6 (Strong exchange axiom)
$$X \in \mathcal{F} \ \wedge \ B \in \mathcal{B} \ \wedge \ X \subseteq B \ \wedge \ x \in E \setminus B \ \wedge \ X \cup \{x\} \in \mathcal{F} \Longrightarrow$$
$$\exists y \in B \setminus X : X \setminus \{y\} \in \mathcal{F} \ \wedge \ B \setminus \{y\} \cup \{x\} \in \mathcal{F}$$

So the reason why the greedy algorithm worked for the greedoid associated to the heaviest tree problem comes from the fact that its formalization under a set system $(E, \mathcal{F})$ (see p.53) satisfied the strong exchange axiom.
Let us also note that if we want to solve the problem of maximizing a (not necessarily non-negative) linear weight function over **all** the independent sets of a greedoid (and not only the maximal ones), we can think to apply the algorithm which had to be used to tackle a similar problem (see p.20) (in which *heaviest* and $\geq$ should be replaced respectively by *extrema* and $\preceq$) but for a matroid instead of a greedoid. In fact, if the modified greedy algorithm is seen as solving a succession of problems of finding the heaviest maximal independent subset whose size is inferior or equal to k (the algo-

[20] This difference is even greater for e_3 and even more for e_4.

rithm starts with $k = 0$ and k would be incremented at each step), the solution to the initial problem is obviously equal to the heaviest among the solutions to the different problems tackled successively. In order to satisfy this condition, the greedy choice should transform the solution to any of the problems (characterized by some k) into the solution of the successor of this problem (characterized by $k + 1$). In [KLS91], we can find the definition of some particular greedoids (the Gauss greedoid) whose exact characterization is precisely that the modified greedy algorithm should find an heaviest independent subset for any given linear weight function.
It's important to have in mind that the problem of maximizing any linear weight function over the independent sets of some greedoid is $\mathcal{NP}$-complete as the Steiner problem can be reduced to it (see [KLS91]).
Let us now come back to the remark saying that greedoids might be defined in several ways (see p.52). In this section, we defined greedoids as a particular set system. They can also be defined as a particular case of word system (or language) which corresponds to an ordered version of set system.

Definition 10
Given a finite ground set E, we denote by E^* the set of all **words** (or strings or sequences) $x_1x_2\cdots x_k$ of elements $x_i \in E$ for $1 \leq i \leq k$.

Definition 11
A **simple** word does not contain any repeated element.

Definition 12
$(E, \mathcal{L})$ is called a **simple word system** or **simple language** if:

- E is a finite set;
- $\mathcal{L}$ is a collection of simple words over E ($\mathcal{L} \subseteq E^*$).

This way of defining greedoids has the great advantage of refering to an order. Therefore, it is possible to distinguish the two following objects:

1. add to the empty set the element x_1 and then x_2 and eventually x_3;
2. add to the empty set the element x_3 and then x_2 and eventually x_1.

In some cases, this is not really useful (e.g. the heaviest forest problem (see p.36)) because the solution of the problem is of type *set* but in other cases (e.g. the sequencing problem (see p.44)), this advantage can be useful (and even sometimes essential) to solve the problem. We shall come back later to this general idea of trying to avoid the limitation coming from the specific type of the solution of a given problem by giving a generalize theory independent of any particular data type (see Section 6).
We saw that for hereditary set systems, matroids were an exact characterization of the fact that the greedy algorithm optimizes all linear weight function (see p. 16). As opposed, we do not have a similar exact characterization for accessible set systems. Indeed, greedoids are too general because we had to add the strong exchange axiom (see p.56) as a necessary condition for the greedy algorithm to optimize all linear weight function. They are also too constraining because there exist some problems which are not underlain by a greedoid and for which the greedy algorithm optimizes any linear weight function. For instance, the first formalization of the heaviest tree problem (see p.53) was not a greedoid but though the greedy algorithm could solve it. In Figure 14, we represent the different stages of the generalized Prim's algorithm (i.e. the greedy algorithm run over the first formalization of the heaviest tree problem) over the graph given in Figure 1.
Our purpose is now to find a structure which is the exact characterization in accessible set systems of the fact that the greedy algorithm optimizes all linear weight function. This is developped in the following section.

5.2 Matroid embeddings

All this section is very much based on Helman's work [HMS93] and on the fifth chapter (The Greedy Method) of [MS90].
As we want to find an exact characterization and as the greedoids were both too general and too constraining, we have to find axioms that both restrict and generalize those of greedoids.
One of the problems we have with accessible set systems comes from the fact that they can contain an independent set which is included in a basis but from which that basis cannot be reached any more. This situation is depicted in Figure 15.a which is taken from [MS90] and in which a set system $(E, \mathcal{F})$ with $E = \{a, b, c, d\}$ and $\mathcal{F} = \{\emptyset, \{a\}, \{b\}, \{a, c\}, \{b, d\}, \{a, b, c\}, \{a, b, d\}\}$ is sketched.

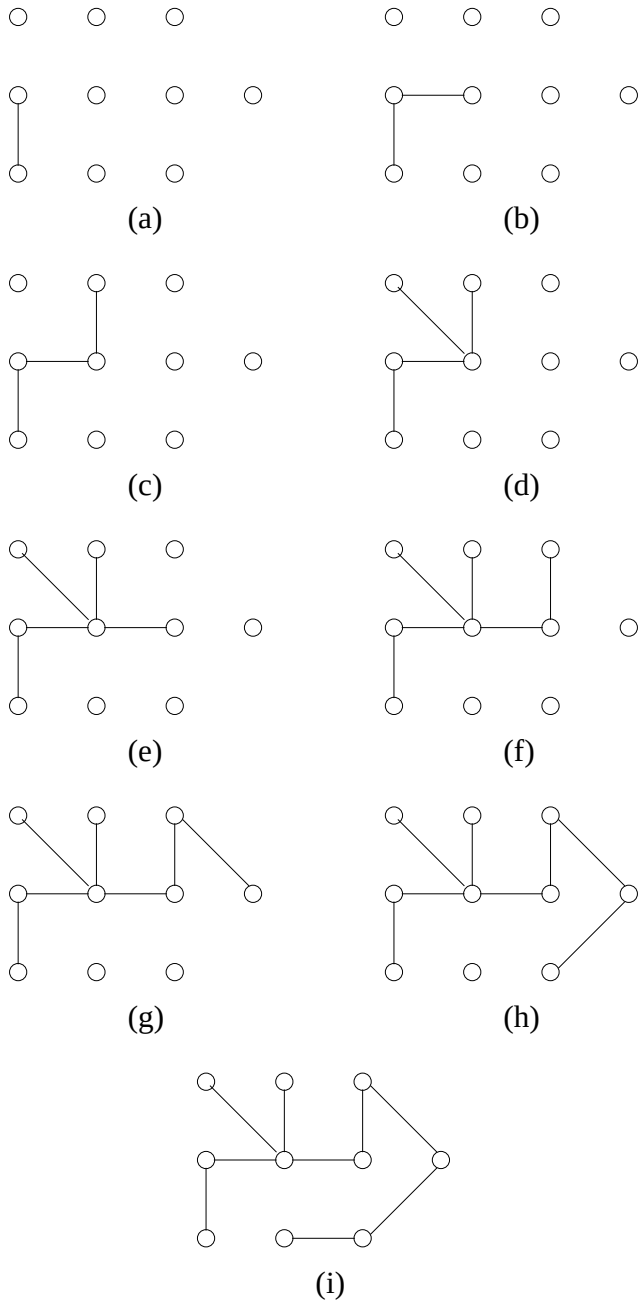

Figure 14: Stages of Generalized Prim's algorithm.

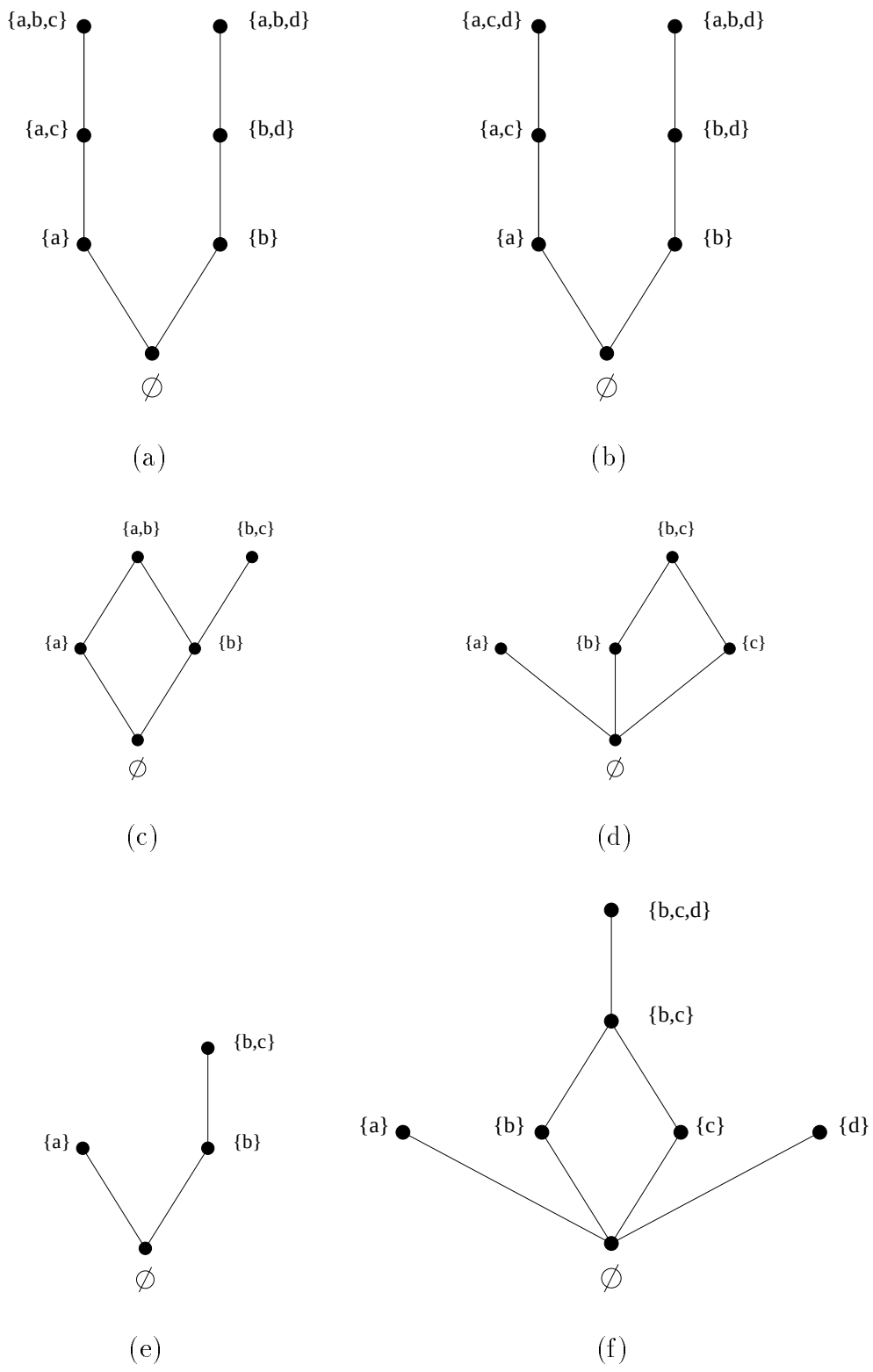

Figure 15: Representation of different accessible set systems.

Indeed, if $weight(a) < weight(b)$ then the greedy algorithm will choose the right part of the tree but if moreover $weight(d) < weight(c)$, this choice won't lead to the optimum as $weight^*(\{a,b,d\}) < weight^*(\{a,b,c\})$. The problem, in this example, comes from the fact that, once we have chosen the set system $\{b\}$, we cannot reach the optimal basis $\{a,b,d\}$ any more, although $\{b\}$ is a subset of it.
The traditional solution to that kind of problem was to include the augmentation axiom (as it's done for matroid and greedoid). However, this was a very strong axiom as it forces that for **every** couple of independent sets of inequal size (X and Y), it is possible to transform the smallest set (let us call it Y) into a new set (Y') of the same size than X by adding successively to Y elements from $X \setminus Y$ while keeping as invariant that all the sets obtained to reach Y' from Y are also independent. This was indeed a very strict way to solve the problem. Instead, we can use the following extensibility axiom which exactly means what we wanted:

Axiom 7 (Extensibility axiom)
$X \in \mathcal{F} \ \wedge \ B \in \mathcal{B} \ \wedge \ X \subset B \Longrightarrow \exists y \in B \setminus X : X \cup \{y\} \in \mathcal{F}$

As the augmentation axiom implies the extensibility axiom, all greedoids satisfy the extensibility axiom (see Figure 16). Therefore, we have already found an axiom generalizing the definition of greedoid (see p.52). We are now going to give two other axioms which will restrict it.
Let us first define a notion on which the two following conditions for a set system to be an exact characterization will be based.

Definition 13
The **hereditary closure** of a set system $(E,\mathcal{F})$ is equal to the set system $(E,\mathcal{F}^*)$ with $\mathcal{F}^* = \{Y \subseteq X | X \in \mathcal{F}\}$.

It's clear that the hereditary closure of any set system is itself a hereditary set system.
After having seen an axiom generalizing greedoid, let us now turn to some other conditions restricting it. If we want the greedy algorithm which is based on locality (see the second condition of our greedy philosphy in Section 2) to lead to a global optimum, we have to include an axiom like the following one:

Axiom 8 (Closure-congruence axiom)
$X \in \mathcal{F} \ \wedge \ x, y \in EXT(X) \ \wedge \ S \subseteq E \setminus (X \cup EXT(X)) \implies$
$X \cup \{x\} \cup S \in \mathcal{F}^* \iff X \cup \{y\} \cup S \in \mathcal{F}^*$

Let us emphasize the fact that the consequent of the closure-congruence axiom is expressed in the hereditary closure and not in the original set system. This axiom is hardly intuitively linked to the greedy algorithm. However, we can consider it as a way to make independent the local choice at one step to all the future choices made at the next steps. In other words, the choice made at one step should prevent the algorithm to include into the partial solution only elements that could also have been chosen at the same step.
As it is done in [MS90], we want to illustrate the closure-congurence axiom by re-expressing its application on the particular problem of the heaviest tree problem (which was solved by the generalized Prim's algorithm). Let us recall that this problem was formalized by a set system $(E, \mathcal{F})$ in which E represents the set of all the arcs of a given connected graph G and $\mathcal{F}$ includes all the trees of G. The bases of $(E, \mathcal{F})$ are the spanning trees of G. For any feasible set (i.e. a tree of G) X, we can partition the set of all the arcs E into four distinct categories :

1. the set of arcs in X;
2. the arcs not in X which have their two endpoints already included in the tree X. None of those arcs could be added to X to form a feasible set;
3. the arcs not in X which have exactly one of their endpoints in the tree X. The set of all these arcs is equivalent to $EXT(X)$;
4. the arcs not in X whose none of their two endpoints is in the tree X.

For this problem, the closure-congruence axiom can be translated to : for any tree X, if x and y are two arcs which have both exactly one endpoint in X and if S is a collection of arcs of the second or of the fourth categories, the set $X \cup \{x\} \cup S$ is a forest of G (i.e. is contained in a tree of G as G is connected) iff $X \cup \{y\} \cup S$ is also a forest of G. In fact, if S contains arcs only of the fourth category which form a forest, then both $X \cup \{x\} \cup S$ and $X \cup \{y\} \cup S$ are forests of G. Otherwise (i.e. if S contains a cycle or at least

one arc from the second category), both $X \cup \{x\} \cup S$ and $X \cup \{y\} \cup S$ aren't forests of G (i.e. they both contain a cycle).
Eventually, let us remark that every heredity set system is closure-congruent as if S is different than the empty set, both sides of the equivalence of the closure-congurence axiom are false and if S is the empty set, they are both true.
We show in Figure 16 a Venn diagram representing the relationships among the heredity, augmentation, extensibility and closure-congruence axioms. We also give a refering example (see Figure 15) for each different classes that appear in that diagram.

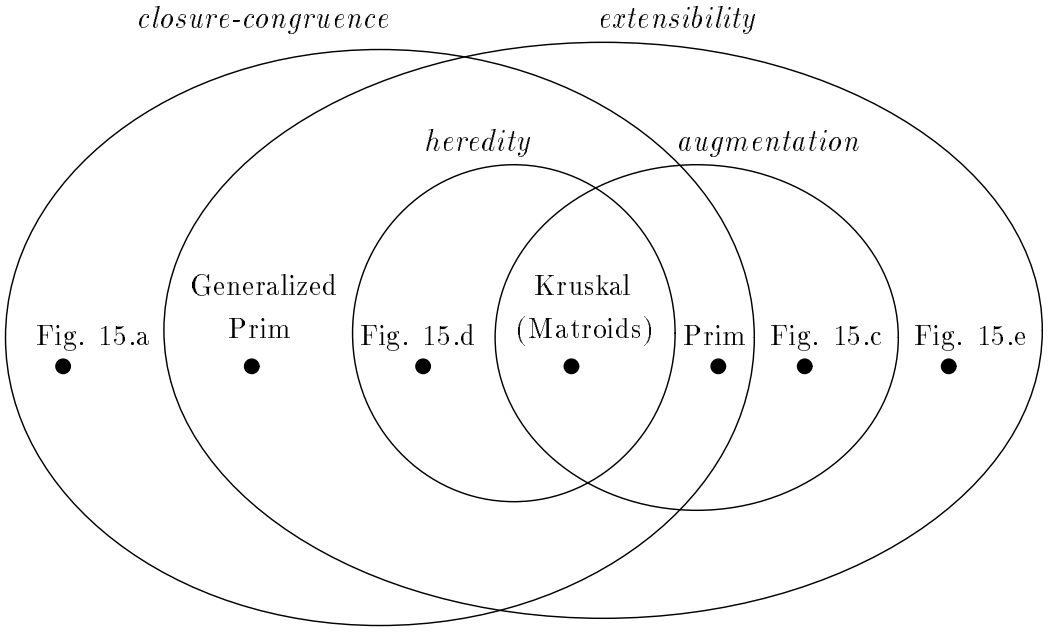

Figure 16: Relationships among accessible set systems.

Accessible set systems satisfying both the extensibility and the closure-congruence axioms are not all such that the greedy algorithm optimizes every linear weight function. Indeed, hereditary set systems fulfill those two axioms and we recall that we only want matroids to be included into our characterization. Therefore, we need some other necessary condition, which

is expressed in Theorem 3.

Theorem 3 (Helman)
For a given accessible set system $(E, \mathcal{F})$, if the greedy algorithm optimizes every linear weight function, then the hereditary closure of $(E, \mathcal{F})$ is a matroid. [Hel89b]

Among hereditary set systems, theorem 3 excludes all the set systems which are not matroids (the hereditary closure of a hereditary set system is equal to the set system itslef.).
In fact the three conditions we have just developped are necessary and sufficient conditions for the exact characterization as theorem 4 expresses.

Definition 14
An accessible set system satisfying both the extensibility and the closure-congruence axioms and whose hereditary closure is a matroid is called a **matroid embedding**.

As it is expressed in [MS90], definition 14 is well formed: the three conditions are independent (i.e. no two conditions imply the third one). To see this, we reprensented in Figure 17 a Venn diagram in which there is an example for every category.

Theorem 4 (Helman 1993)
The greedy algorithm run over an accessible set system $(E, \mathcal{F})$ optimizes all linear weight functions if and only if $(E, \mathcal{F})$ is a matroid embedding. [MS90, HMS93]

In [HMS93], the largest class of functions to which theorem 4 applies is identified. Moreover, an exact characterization of greedy structures for another class of weight function than the linear one (the bottleneck functions[21]) is given.

[21] For bottleneck functions, the weight of an independent subset of a set system $(E, \mathcal{F})$ is defined by: $weight^* : 2^E \rightarrow \mathbb{R} : weight^*(X) = \min_{x \in X} weight(x)$

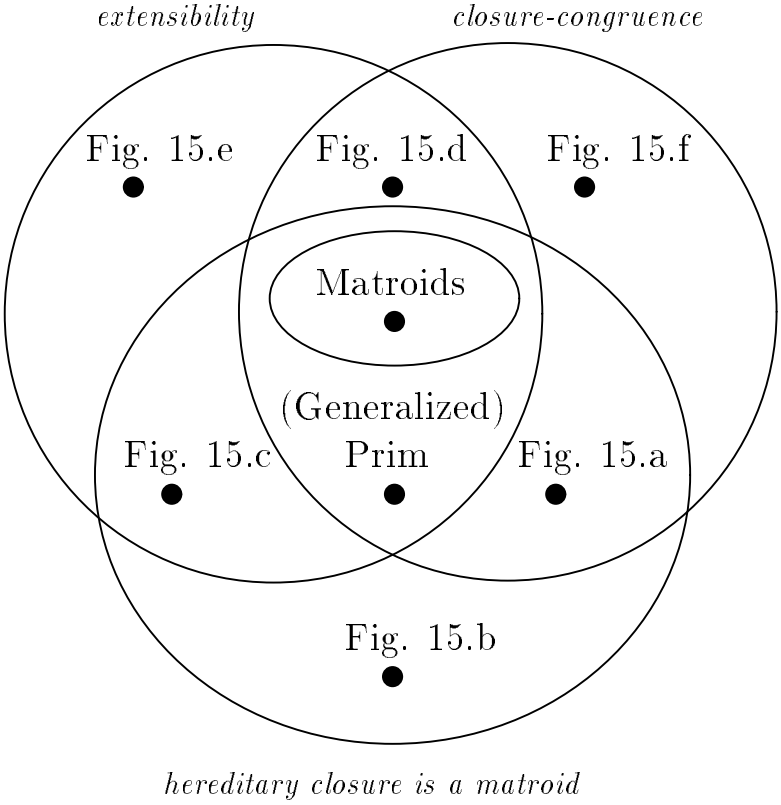

Figure 17: Independence beteween the three conditions of the exact characterization.

6 Second generalization of matroid theory

6.1 Introduction: the Huffman's algorithm example

All the formalizations of the greedy algorithm we have seen up to now were based on **set** systems. Therefore, they are heavily dependent on the particular data type *set*. This fact is a major drawback when we want to deal with problems which are difficult to be **naturally** expressed by returning a set of objects. When we tackled the sequencing problem (see p.44) which was intended to return an ordering of objects, we met this problem because

we had to transform the problem in an equivalent one which would return a set (see Sections 4.4.3 and 4.5). In this particular case, it was not too difficult because the data types *set* and *sequence* are not too different[22] but there exist other problems for which this transformation would be very difficult (or maybe even impossible). For instance, let us present the Huffman's algorithm for data compression[23].
The problem is the following: we have a text composed of different characters that appear with different frequencies. We would like to find for each character a sequence of bits that would represent it univoquely and that is called the *code* of the character. The set of codes for all the characters is called a binary character code. The binary character code we are looking for must be such that the resulting total length of the coded text (i.e. the text in which every character is replaced by its code) is minimal. We present in Figure 18, an example taken from [CLR90] in which the initial text is

	a	b	c	d	e	f	Total
Occurence	45	13	12	16	9	5	100
Fixed-length-code	000	001	010	011	100	101	300
Variable-length-code	0	101	100	111	1101	1100	224

Figure 18: A character-coding problem.

constituted of 100 characters including a, b, c, d, e and f whose number of occurence in the text is given in the first line of Figure 18. In the next two lines, we give two examples of codes, the first of which is such that the length of the codes for all the characters are exactly the same. As we have to represent six characters, we need at least 3 bits. Therefore, for this first example,

[22]Indeed, both data types fit into the Boom hierarchy [Mee86]. The properties of the underlying binary operator, on which is based this hierarchy, has to be associative for the data type *sequence* and associative, commutative and indempotent for the data type *set* [Jeu92].

[23]We haven't proved that it's impossible to formalize this problem by a set system. However, we know that it's very difficult and even if it's possible, the resulting formalization will certainly be more awkward and much trickier than the standard Huffman's algorithm based on the data type *binary tree*.

the length of the whole text would be of 300 bits. The last line represent a code which doesn't satisfy the previous condition of equal length for all the codes. The suppression of this constraining condition is very useful in order to solve optimally our problem. Indeed, it is intuitively clear that it would be optimal to assign the shortest codes to the most frequent characters.
We want to represent the binary character code we find thanks to a binary tree. The set of all characters would consist of all the leaves of the binary tree. Let us first note that the binary trees we use are non standard for different details:

- only the leaves have an associated information which is a character and the internal nodes do not have any associated information;
- each node has either two child or no children at all (i.e. there is no node with exactly one children)[24].

As optimal codes always satisfy this last condition, it's not constraining at all. For any non leaf node, the arc leading to its left (resp. right) child is labelled by a *0* (resp. *1*). For each character, its associated code is obtained by concatenating the labels associated to all the arcs of the unique path starting from the root of the binary tree and arriving at the leaf representing that character (Figure 19.f depicts the binary tree representing the variable-length code of Figure 18).
Therefore, only prefix codes (i.e. codes in which no code for a character is the prefix of the code for another character) can be represented by a binary tree. Nevertheless, it can be proved that there is always at least one prefix code among all the optimal codes. Thus any optimal prefix codes are optimal codes as well. Moreover, prefix codes are very useful because they simplify very much both encoding and decoding. Huffman [Huf52] discovered a very simple way to find optimal prefix codes that have been called since then *Huffman codes*. At each step of the algorithm, two binary trees from a set of binary trees are chosen and merged to form a new one. Their choice is such that their associated frequencies are minimal for all the binary trees in the

[24]This constraint prevents us to represent trees with one single leaf and thus we cannot use the Huffman's algorithm to find the optimal code for a text containing only one character. Obviously, this problem is not very intersting but, as opposed to what is usually done in the literature, we prefer to mention this tiny restriction to be fully correct.

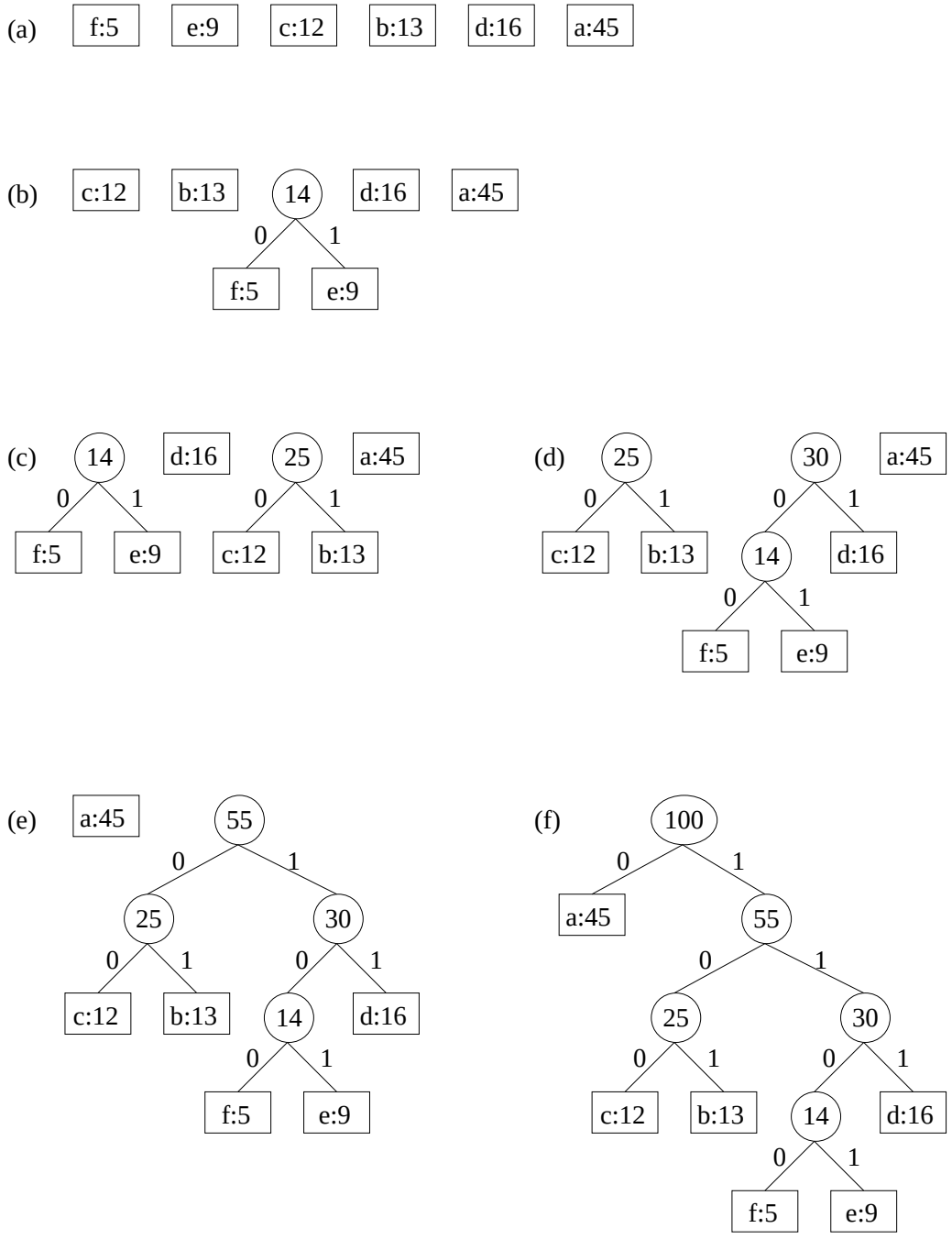

Figure 19: Stages of Huffman's algorithm.

set. The frequency of a binary tree (that we shall call $weight$) is defined as the sum of the frequencies of all the characters associated to its leaves. It is noteworthy that a problem so different than all the problems underlain by a matroid, greedoid or matroid embedding we have seen also fulfill exactly the two conditions of the greedy principle (see Section 2). We show in Figure 19 the different steps of the following Huffman's code algorithm over the text represented in line one of Figure 18:

The Huffman Greedy Algorithm
begin
  $Solu := \{leaf(c) | c \in C\}$
  while $|Solu| > 1$ **do**
  begin
    $e_1 \in extrema(weight, Solu)$
    $e_2 \in extrema(weight, Solu \setminus \{e_1\})$
    $Solu := Solu \cup \{merge(e_1, e_2)\} \setminus (\{e_1\} \cup \{e_2\})$
  end
end

In the Huffman algorithm, C represents the set of all the characters used in the original text, $leaf(object)$ creates a binary tree consisting of only one node, a leaf, whose associated information is $object$, $weight$ is defined hereafter and the relation $\preceq$ in the definition of $extrema$ is instanciated to $\leq$ (i.e. we want to find an oject minimizing function $weight$).
Let us notice that in Figure 19, we have added some information to the binary trees which do not appear identically in the binary trees used in the Huffman algorithm:

- the leaves contain both a character and its associated frequency and in the Huffman algorithm, in contains only a character and its frequency is given by the function $freq$;
- the internal nodes contain the associated frequency of the subtree rooted at that node and in the Huffman algorithm, that value is obtained thanks to the function $weight$.

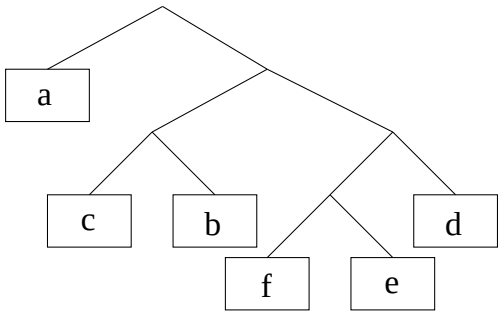

Figure 20: Representation of a "real" binary tree of the Huffman algorithm.

We represented in Figure 20 the binary tree of Figure 19.f as used in the Huffman algorithm. We also want to remark that the first line of the Huffman algorithm (i.e. $Solu := \{leaf(c) | c \in C\}$), corresponds to the initialization step and its result is represented in Figure 19.a.
We easily calculate that the Huffman codes represent approximatively a savings of 25% compared to the fixed-length code represented in the second line of Figure 18. In fact, savings from 20% to 90% are usual, depending on some features of the original text. As we have already mentionned, the more irregular the frequencies of the characters of the original text are, the more interesting to use the Huffman's code algorithm is.
Eventually, we want to make one remark about Huffman's algorithm because it's the first time we meet a problem for which we deal with a non linear weight function and moreover the local and global optimizations look quite different. Indeed, locally, we find at each step two binary trees with the least $weight$ (frequence) defined as:

$$weight(T) = \sum_{e \in L(T)} freq(e)$$

where T represents a binary tree, $L(T)$ the set of all characters associated with the leaves of T and $freq(e)$ the number of occurence of character e in the initial text.
As opposed, globally, the problem is to find a tree T whose $weight^*$ is minimal and is defined by:

$$weight^*(T) = \sum_{e \in L(T)} freq(e) * d_T(e)$$

where $d_T(e)$ represents the depth in the tree T of the leaf to which the character e is associated and $weight^*(T)$ represents exactly the length of the text coded thanks to the codes associated to T. In Figure 19, the value of $weight(T)$ is represented by the number in the root of T and the value of $weight^*(T)$ by the sum of the numbers associated to all non leaf nodes.
To sum up, the existence of an algorithm such as the Huffman's code algorithm which satisfies the greedy philosphy without being **naturally** underlain by a matroid, greedoid or matroid embedding structure leads us to look for a generalization of the previous structures which would be independent of the data type *set*.
We are also confirmed in this direction by previous work [KL81, Hel89b] which expressed the desire to use other more general settings than set systems.

6.2 Formalism for the generalization

As we want to find a formalization in which we get rid of the constraint of the data type *set*, the best solution would be to express the formalization independently of any particular data type.
In fact, the use of algebraic specification is ideal for that purpose without begin an absolute necessity (i.e. we could have tried to use other settings). Indeed, it allows to express some schemes thanks to sorts and signatures without directly referencing to the different types of the objects in use. Therefore, a sort could stand for any particular data type.
Let us make a brief review of the algebraic concepts that will be needed in the sequel. The interested reader could refer to [EM85] for a deep introduction and to [Smi83] to find how algebraic specifications was already used in order to express the structure of divide-and-conquer algorithms.
Let S represent a nonempty set of symbols called sorts. We distinguish a sort $(\hat{s})$ of S that we call the *principal sort*.
An S-sorted set X is an S-indexed family of sets $\{X_s\}_{s \in S}$.
Let X be a S-sorted set and $w = w_1 w_2 \dots w_n$ an element of S^*, we denote by X^w the cartesian product $X_{w_1} \times X_{w_2} \times \dots \times X_{w_n}$. Letting ϵ denote the empty string, X^ϵ represents the set consisting of the 0-tuple: $\{\langle\rangle\}$.

An *S-sorted signature* Σ is an $S^* \times S$-sorted set $\{\Sigma_{w,s}\}_{w \in S^*, s \in S}$. $\Sigma_{w,s}$ is a set containing the **symbols** of operations with argument sort w and with range sort s. The constants of a sort s can be considered as a special kind of operators on that sort: a constant is nothing else than a function without argument. Therefore, the symbols of the constants of the sort s consist of the set $\Sigma_{\epsilon,s}$.
A *signature* $\Sigma = \langle S, \{\Sigma_{w,s}\}_{w \in S^*, s \in S} \rangle$ is a pair formed by a set of sorts S and by a S-sorted signature.
A *$\hat{s}$-oriented S-sorted signature* Σ is a S^*-sorted set $\{\Sigma_w\}_{w \in S^*}$. Σ_w is a set containing the symbols of operations with argument sort w and with **range sort** $\hat{s}$.
A *$\hat{s}$-oriented signature* $\Sigma = \langle S, \{\Sigma_w\}_{w \in S^*} \rangle$ is a pair formed by a set of sorts S and by a $\hat{s}$-oriented S-sorted signature.
Let Σ be a S-sorted signature (resp. $\hat{s}$-oriented S-sorted signature). A Σ-algebra A consists of:

- a S-sorted set $\{A_s\}_{s \in S}$ where the sets A_s are called the carriers (or base sets or domains) of A;
- for each operation symbol f such that $f \in \Sigma_{w,s}$ (resp. $f \in \Sigma_w$), a function f_A such that $f_A : A^w \rightarrow A_s$ (resp. $f_A : A^w \rightarrow A_{\hat{s}}$).

$A_{\hat{s}}$ is called the principal carrier of A.
In the sequel, we shall usually represent by $\langle \{C_1, \ldots, C_k\}, \{f1, \ldots, fr\} \rangle$ any Σ-algebra A. In that representation, $C_1, \ldots, C_k$ denote the carriers of A and $f1, \ldots, fr$ its operators.
From now on, by default, Σ will denote a $\hat{s}$-oriented S-sorted signature which can be equally represented by the set of operator symbols $\{\sigma 1, \ldots, \sigma r\}, r \geq 1$, where $\forall i \; 1 \leq i \leq r$, σi has type $\langle wi, \hat{s} \rangle$ where $wi \in S^*$ and $wi = wi_1, \ldots, wi_{n_i}$ with $n_i \geq 0$.
Let us now see an example (which were inspired by [ST95]) illustrating all these previous definitions.

Example 5
The set of sorts will be the following: $S = \{shape, suit\}$.

Let us define a $S^* \times S$-indexed set Σ by:

$$\begin{array}{ll} \Sigma = & \{\Sigma_{\epsilon,shape}, \Sigma_{\epsilon,suit}, \Sigma_{shape,shape}, \Sigma_{shape\ suit,suit}\} \text{ in which:} \\ & \Sigma_{\epsilon,shape} = \{box\}, \\ & \Sigma_{\epsilon,suit} = \{hearts, spade\}, \\ & \Sigma_{shape,shape} = \{bagify\} \text{ and} \\ & \Sigma_{shape\ suit,suit} = \{f\}. \end{array}$$

Σ represents a S-sorted signature in which box is a constant of the sort $shape$ and $hearts$ and $spade$ for the sort $suit$. $\langle S, \Sigma \rangle$ is a signature. Next we present a Σ-algebra A:

$$\begin{array}{ll} A_{shape} = \{\square, \triangle\} & box_A = \square \\ A_{suit} = \{\clubsuit, \heartsuit, \spadesuit\} & hearts_A = \heartsuit \\ & spade_A = \spadesuit \\ & boxify_A : \begin{cases} \square \longmapsto \square \\ \triangle \longmapsto \square \end{cases} \end{array}$$

f_A :

f_A	$\clubsuit$	$\heartsuit$	$\spadesuit$
$\square$	$\clubsuit$	$\spadesuit$	$\heartsuit$
$\triangle$	$\heartsuit$	$\spadesuit$	$\spadesuit$

If we want to represent A with $\langle \{C_1, \ldots, C_2\}, \{f1, \ldots, f5\} \rangle$, it's easily possible by choosing:

$$\begin{array}{ll} C_1 = A_{shape} & f1 = box_A \\ C_2 = A_{suit} & f2 = hearts_A \\ & f3 = spade_A \\ & f4 = boxify_A \\ & f5 = f_A \end{array}$$

If we keep the same S as before and if we choose the sort $suit$ to be the principal sort (i.e. $\hat{s}$), we can easily define a $\hat{s}$-oriented S-sorted signature Σ' as follows:

$$\Sigma' = \{\Sigma'_\epsilon, \Sigma'_{shape\ suit}\} \text{ in which: } \begin{cases} \Sigma'_\epsilon = \{hearts, spade\} \\ \Sigma'_{shape\ suit} = \{f\} \end{cases}$$

We can also represent Σ' by the set of its r (i.e. 3) operator symbols $\sigma' i$:

i	σi	wi	ni
1	$hearts$	ϵ	0
2	$spade$	ϵ	0
3	f	$shape\ suit$	2

In the sequel, we consider S a set of sorts in which $\hat{s}$ is the principal sort, Σ, a $\hat{s}$-oriented S-sorted signature represented by the set of operator symbols $\{\sigma 1, \ldots, \sigma r\}$ and A a Σ-algebra.
Let us give other examples much closer to our interest domain:

Example 6
$S = \{c, \hat{s}\}$ and $\Sigma = \{\sigma 1, \sigma 2\}$ with:

i	σi	wi	ni
1	$\sigma 1$	ϵ	0
2	$\sigma 2$	$c\ \hat{s}$	2

We can associate to Σ several Σ-algebras such as A or B:

- $A = \langle \{\mathbb{N},\ set(\mathbb{N})\}, \{empty,\ with\} \rangle$
- $B = \langle \{\mathbb{R},\ list(\mathbb{R})\}, \{nil,\ cons\} \rangle$

Example 7
$S = \{c, \hat{s}\}$ and $\Sigma = \{\sigma 1, \sigma 2, \sigma 3\}$ with:

i	σi	wi	ni
1	$\sigma 1$	ϵ	0
2	$\sigma 2$	c	1
3	$\sigma 3$	$\hat{s}\ \hat{s}$	2

We can associate to Σ a Σ-algebra A' such as:
$A' = \langle \{object,\ bintree(object)\}, \{empty,\ leaf,\ merge\} \rangle$.

In Example 6, the carrier $A_{\hat{s}}$ (resp. $B_{\hat{s}}$) of the first (resp. second) Σ-algebra basically represents the data type *set* (resp. *list*). Indeed, the fact that we have $set(\mathbb{N})$ (resp. $list(\mathbb{R})$) is not important: we could have used without any difference $set(object)$ (resp. list(object)) instead, whatever *object* would be. Moreover, $\sigma 1_A$ and $\sigma 2_A$ (resp. $\sigma 1_B$ and $\sigma 2_B$) are constructors of the corresponding data type as their range is exactly $A_{\hat{s}}$ (resp. $B_{\hat{s}}$). The other carriers than $A_{\hat{s}}$ (resp. $B_{\hat{s}}$) represent the other types than the data type represented by $A_{\hat{s}}$ (resp. $B_{\hat{s}}$) which are used by the constructors.
So from now on, we shall explicitely refer to a $\hat{s}$-oriented signature $\Sigma = \langle S, \Sigma' \rangle$ where $\Sigma' = \{\sigma 1, \ldots, \sigma r\}$. The data type of the object which is constructed by the greedy algorithm we want to formalize is abstracted by $A_{\hat{s}}$. More generally all the sets $\{A_s\}$ for $s \in S$ representing a set of objects will all also be seen as **types**. Finally, $\{\sigma 1_A, \ldots, \sigma r_A\}$ abstracts a particular set of constructors of that data type.

6.3 A first generalizing step

As matroids, greedoids and matroid embeddings are all based on set systems and therefore on the data type *set*, we can easily express the greedy algorithms underlain by any of these structures without explicit reference to the data type set but by only using its underlying $\hat{s}$-oriented signature Σ (see Example 6).
Refering to Example 6, the first greedy algorithm (see p.16) would be reexpressed as:

Abstracted Greedy Algorithm
begin
  $Solu := \sigma 1_A$
  while $EXT(Solu) \neq \emptyset$ **do**
  begin
    $e \in extrema(weight, EXT(Solu))$
    $Solu := \sigma 2_A(e, Solu)$
  end
end

where $EXT(X) = \{x | x \in E \setminus X \ : \ \sigma 2_A(x, X) \in \mathcal{F}\}$.

This intermediate algorithm is abstracted from the data type *set* and the last traces from that data type it contains are located in the definition of EXT. In the following, the idea would be to find another version without **any** reference to a particular data type.
Refering to Example 7, the Huffman algorithm (see p.69) can also be reexpressed as:

Abstracted Huffman Algorithm
begin
  $Solu := \{\sigma 2_{A'}(c) | c \in C\}$
  while $|Solu| > 1$ **do**
  begin
    $e_1 \in extrema(weight, Solu)$
    $e_2 \in extrema(weight, Solu \setminus \{e_1\})$
    $Solu := Solu \cup \{\sigma 3_{A'}(e_1, e_2)\} \setminus (\{e_1\} \cup \{e_2\})$
  end
end

in which some constructors of the *bintree* data type have been replaced by their associated operators σ.
As we want to find an algorithm that would generalize both the the abstracted greedy algorithm and the abstracted Huffman algorithm, we shall first slightly modify the latter in order for both of them to have an exact similar shape.
We shall transform the sequence of the two statements:

$$e_1 \in extrema(weight, Solu)$$
$$e_2 \in extrema(weight, Solu \setminus \{e_1\})$$

representing the choice of two optimal objects into an equivalent one single statement representing the choice of only one object which is a couple containing as its i-th element a possible e_i ($i = 1$ or 2):

$$e \in extrema(weight^2, Solu \times Solu)$$

in which $weight^2$ could be defined as

$$weight^2 : \ bintree \times bintree \to \mathbb{R} \ :$$
$$weight^2(X, Y) = \begin{cases} 2 * weight(X) + weight(Y) & \text{if } X \neq Y \\ +\infty & \text{if } X = Y \end{cases}$$

This choice of $weight^2$ could seem a bit tricky but we gave its definition to show that the transformation of the two statements into a single statment was thoroughly possible. However, in the following, we might not give explicitely the $weight$ function but explain in some other way what the optimal tuple should be (by constructing each of its element for instance).
An algorithm scheme generalizing both the abstracted greedy and the abstracted Huffman algorithms could be expressed as:

Generalized Greedy Algorithm Scheme
begin
  $Solu$:= INITIALIZATION
  while CONDITION **do**
  begin
    $e \in extrema(weight,$ EXPRESSION1)
    $Solu$:= EXPRESSION2
  end
end

We would like in the following to find abstract expressions for which INITIALIZATION, CONDITION, EXPRESSION1 and EXPRESSION2 stand for.
The greedy and the Huffman algorithms are similar and complementary at the same time. Let us emphasize both their similarities and their differences which are important for their generalization:

- they are both based on the greedy principle;
- they can be expressed very similarly;
- $Solu$ represents only **one** element (i.e. a set) in the abstracted greedy algorithm and a set of **several** elements (i.e. trees) in the Huffman algorithm;
- they are both based on **one** basic constructor:
  - $with : A_c \times A_{\hat{s}} \rightarrow A_{\hat{s}}$ (see Example 6) for the greedy algorithm;
  - $merge : A_{\hat{s}} \times A_{\hat{s}} \rightarrow A_{\hat{s}}$ (see Example 7) for the Huffman algorithm;

- at each step of the algorithm, a new object of $A_{\hat{s}}$ is constructed by applying the basic constructor to subobjects belonging in some given sets:

  - the function *with* for the greedy algorithm, has as a first argument an element from the set that we always called E and as a second argument was an element represented $Solu$[25];
  - the function *merge* for the Huffman algorithm, has its two arguments which are elements from $Solu$.

  Therefore, the final solution found by the algorithm is obtained only by using elements that belong to the given sets;

- when the algorithm ends, the final solution is represented by $Solu$:

  - for the greedy algorithm, $Solu$ is the final solution;
  - for the Huffman algorithm, $Solu$ is a singleton whose single element is the final solution.

  Therefore, at any step of the algorithm, a necesseray condition for the algorithm to stop is that $Solu$ represents only one element (i.e. the final solution). Otherwise, if it represents several elements, the algorithm will not stop at that step and **all** these elements will have to be used in order to construct the final solution;

- each element represented by/contained in $Solu$ is optimal for some associated problem:

  - at step k of the greedy algorithm, $Solu$ represents the optimum of the same problem but for which the initial set system $(E, \mathcal{F})$ is replaced by: $(E, \mathcal{F}')$ in which $\mathcal{F}' = \{X | X \in \mathcal{F} \ \wedge \ |X| \leq k\}$;
  - for the Huffman algorithm, each element in $Solu$ is a tree which is the solution of the initial problem for which the number of characters has been reduced: all the characters that do not appear in any leaf of that tree have not been considered;

[25] **The** element would also be correct as $Solu$ represents one single element.

Let us now sum up our ideas for the generalization:

- for each problem that we want to solve by our generalized greedy algorithm, we consider that we know a Σ-algebra A that underlies it by the fact that the carriers A_s (for $s \in S$) represent some types (i.e. sets of objects) and the operators $\{\sigma 1_A, \ldots, \sigma r_A\}$ represent some constructors (whose domain is a cartesian product of the defined types) of the data type symbolized by $A_{\hat{s}}$. Among those constructors, one of them is identified (σz_A with $1 \leq z \leq r$) as the one being applied at each iteration of the algorithm;

- for each type A_s (for $s \in S$), we have a set B_s of objects of that type (i.e. $B_s \subseteq A_s$). Let us notice that $B_{\hat{s}}$ was simulated in the previous algorithm by $Solu$[26];

- at each iteration σz_A can be applied only to objects that belong to $\{B_s\}_{s \in S}$. These objects are extracted from their respective sets and the object created is added to $B_{\hat{s}}$;

- the algorithm finishes with $B_{\hat{s}}$ being a singleton and whose single element is the final solution. Therefore, each element in $B_{\hat{s}}$ as been used to construct an optimal object and can be considered as an optimal subobject which can also be considered as optimal for a modified version of the intial problem.

The objects of $\{B_s\}_{s \in S \setminus \{\hat{s}\}}$ from which objects in $B_{\hat{s}}$ are constructed are called ground objects.
With all these new considererations, we turn back to our previous goal which was to find expressions translating INITIALIZATION, CONDITION, EXPRESSION1 and EXPRESSION2 of the generalized greedy algorithm scheme (see p.77) in order to obtain the generalized greedy algorithm.
First, let us focus on the expression:

$$EXT(X) = \{x | x \in E \setminus X \ : \ \sigma 2_A(x, X) \in \mathcal{F}\}$$

of the abstracted greedy algorithm (see p.75) for which we said that it contains the last traces of the data type set in that algorithm.

[26] For the abstracted Huffman algorithm, we have: $B_{\hat{s}} = Solu$ and for the abstracted greedy algorithm, we had:$B_{\hat{s}} = \{Solu\}$

Let us recall that X was always instanciated by $Solu$, the partial solution being constructed and thus the only element of $B_{\hat{s}}$.
Let us start by considering the sub-expression $\sigma 2_A(x, X)$ in which $\sigma 2_A$ was the basic constructor (i.e. the constructor applied at each step) of the abstracted greedy algorithm. It's thus now called σz_A. Constructor $\sigma 2_A$ was applied to the pair x (an element of $E \setminus X$) and $Solu$. In our generalization, we know that E and $Solu$ became respectively B_c and **the** element of the singleton $B_{\hat{s}}$. So as we generalized $\sigma 2_A \ : \ A_c \times A_{\hat{s}} \rightarrow A_{\hat{s}}$ by $\sigma z_A \ : \ A^{wz} \rightarrow A_{\hat{s}}$ and as we know that we can construct objects only by using elements from $\{B_s\}_{s \in S}$, σz_A can only be applied to tuples from B^{wz}. In the set system formalism, we could always use the collection $\mathcal{F}$ as a feasibility test (i.e. $x \in \mathcal{F}$ iff x is a feasible solution) but in the generalization, we don't have access to that collection any more. We thus suppose that we have a predicate $feasible$ which returns true if and only if its argument is a feasible solution. In the future, we shall analyze some properties that $feasible$ has to satisfy as $\mathcal{F}$ verified some axioms in the matroid/greedoid/matroid-embedding formalism. So, we now have translated the expression:

$$\{x | x \in E \ : \ \sigma 2_A(x, X) \in \mathcal{F}\}$$

to

$$\{x | x \in B^{wz} \ : \ feasible(\sigma z_A(x))\}$$

The former expression differs from $EXT(X)$ only by the fact that $x \in E$ instead of $x \in E \setminus X$ which expresses the fact that the element x has to belong to the initial set of ground objects E but cannot have been already chosen in order to create an object (of $B_{\hat{s}}$). In the set system formalism, it was very easy to express such a fact because the constructed object (i.e. X) was expressed as the set of all the (ground) elements of the set E chosen until then. In order to achieve the same purpose in our generalization, we create an $S \setminus \{\hat{s}\}$-indexed family of functions $\{elements\}_{s \in S \setminus \{\hat{s}\}}$ defined as follows:

$$\forall s \in S \setminus \{\hat{s}\} \ : \ elements_s \ : \ A_{\hat{s}} \rightarrow A_s \ : \ \forall i \in \{1, ..., r\} \ :$$

$$elements_s(\sigma i_A(e)) = \begin{cases} \bigcup\limits_{\substack{1 \leq j \leq n_i \\ wi_j = s}} \{e_j\} \ \cup \bigcup\limits_{\substack{1 \leq j \leq n_i \\ wi_j = \hat{s}}} elements(e_j) & \text{if } wi \neq \epsilon \\ \emptyset & \text{if } wi = \epsilon \end{cases}$$

with the intended following semantics: $elements_s(x)$ returns the set of all the ground elements of A_s that have been used in order to create x.
The previous definition of $elements_s$ is easily understood if we notice that the first union includes all the ground elements of the type A_s and the second union includes recursively all the elements of that type that were used in order to construct the objects of the type $A_{\hat{s}}$, themselves used to construct the object x. All the ground objects of type A_t (with $t \neq s$) are obviously not taken into account at all.
To be completely correct, the previous definition of $elements$ should be equivalent whatever the σi which is "chosen" to reach an object. For instance, if we associate to the set data type the constructors $empty$, $with$ and $union$. The previous remark impose that:

$$elements(with(4, \{1,2,3\})) = elements(\{1,4\}, \{2,3\})$$

We can now define a $S \setminus \{\hat{s}\}$-indexed family of functions $\{Elements_s\}_{s \in S \setminus \{\hat{s}\}}$ that computes **all** the ground elements that have already been used in order to obtain all the elements in $B_{\hat{s}}$:

$$\forall s \in S \setminus \{\hat{s}\} \ : \ Elements_s \ : \ 2^{A_{\hat{s}}} \rightarrow A_s \ : \ \forall i \in \{1, ..., r\} \ :$$

$$Elements_s(Y) = \bigcup_{x \in Y} elements_s(x)$$

Finally, we can ease the following by extending the $S \setminus \{\hat{s}\}$-indexed family of functions $\{Elements\}$ to a S-indexed family by adding the equality: $Elements_{\hat{s}} = \emptyset$.
Therefore, if we define $EXT(\{X_s\}_{s \in S})$ by:

$$\{x | x \in (X \setminus Elements(X_{\hat{s}}))^{wz} \ : \ feasible(\sigma z_A(x))\}$$

then $EXT(\{B_s\}_{s \in S})$ represents the exact translation of the expression $EXT(Solu)$ of the abstracted greedy algorithm. Let us note that the argument of EXT is from now on an S-indexed family of sets and was in the abstracted greedy algorithm a set. However, we notice that the expression $EXT(X)$ of the abstracted greedy algorithm, used E as an implicit argument. Thus, all the arguments of the abstracted greedy algorithm (i.e. E and $Solu$) correspond exactly to the specialization of the S-indexed family of sets $\{B_s\}_{s \in S}$.

There is one last feature that we have to add to EXT in order to be also used in the Huffman algorithm: in a tuple of arguments containing two (or more) arguments of the same type, we want to have *different* values for all the arguments of the same type. So, we add this last constraint in the definition of $EXT(\{X_s\}_{s\in S})$ to get its final expression:

$$\{x|x \in (X \setminus Elements(X_{\hat{s}}))^{wz} \ : \ feasible(\sigma z_A(x)) \wedge \\ (\forall i,j : 1 \leq i,j \leq n_z \ \wedge \ wz_i = wz_j \ \wedge \ i \neq j \Longrightarrow x_i \neq x_j)\}$$

We could ponder on the solution to take when we would have to tackle a problem in which some identical elements of the same type could coexist with each other. The easiest solution would be to use a family of **bags** instead of the family of sets $\{B_s\}_{s\in S}$. This solution has the great advantage of not changing anything to all our previous developments.
Let us finally answer our initial purpose of finding translations to the four untranslated expressions of the generalized greedy algorithm scheme:

- INITIALIZATION basically depends on the particular problem we are solving. Thus, we won't give a particular translation for the initialization of $B_{\hat{s}}$ (i.e. previously $Solu$). We shall come back later to a possible formalization of this step of initialization;
- CONDITION is replaced by: $EXT(\{B_s\}_{s\in S}) \neq \emptyset$;
- EXPRESSION1 is translated to: $EXT(\{B_s\}_{s\in S}$;
- $Solu :=$ EXPRESSION2 is abstracted by: $B_{\hat{s}} := B_{\hat{s}} \cup \sigma z_A(e)$.

The generalized greedy algorithm scheme can now be re-expressed as:

Generalized Greedy Algorithm (1)
begin
  $B_{\hat{s}} :=$ INITIALIZATION
  while $EXT(\{B_s\}_{s\in S}) \neq \emptyset$ **do**
  begin
    $e \in extrema(weight, EXT(\{B_s\}_{s\in S}))$
    $B_{\hat{s}} := B_{\hat{s}} \cup \sigma z_A(e)$
  end
end

Let us note that there is a difference between $B_{\hat{s}}$ and B_s (with $s \in S \setminus \{\hat{s}\}$): the latter is consider as a given data that the algorithm will use and the former is not a data at all: it is initialized in the beginning of the algorithm and its final value corresponds to the solution of the problem.
Let us also remark that the function *weight* used in this first generalized greedy algorithm is different from the ones (of the same name) of the abstracted greedy and Huffman algorithms. Indeed, for the abstracted Huffman algorithm for instance, it corresponds precisely to the function $weight^2$ [27].
The use of functions $Elements_s$ to define EXT in algorithm 6.3 may look a bit awkward if we explain the basic idea why they were introduced. In fact, the algorithm starts with some given sets B_s (with $s \in S \setminus \{\hat{s}\}$) containing all the ground objects that can be used in order to construct the final solution. Once some ground objects have been used, they cannot be used any more in the future. This feature is at the origin of the introduction of the functions $Elements_s$[28]. There is a much more intuitive way to take this feature into account (based on the difference between the first and the second greedy algorithms (see p.16 and p.17)): each time a ground object e is chosen in a set B_s (with $s \in S$), it will be deleted from that set for future potential choice (i.e. $B_s := B_s \setminus \{e\}$).
The first generalized greedy algorithm can thus be re-expressed as a second one:

Generalized Greedy Algorithm (2)
begin
  $B_{\hat{s}} :=$ INITIALIZATION
  while $EXT(\{B_s\}_{s \in S}) \neq \emptyset$ **do**
  begin
    $e \in extrema(weight, EXT(\{B_s\}_{s \in S}))$
    $B_s := B_s \setminus \{e_i | 1 \leq i \leq n_z \ \wedge \ wz_i = s\} \ \ \forall s \in S$
    $B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma z_A(e)\}$
  end
end

[27] It's interesting to note that the definition of function $weight^2$ already prevented the algorithm from choosing twice the same element. However, as this constraint has been added to EXT, we could simplify $weight^2$.

[28] We shall see in the future that these functions have another utility in the expression of the invariant of the algorithm.

in which $EXT(\{X_s\}_{s\in S})$ is defined as:

$$\{x|x \in X^{wz} \ : \ feasible(\sigma z_A(x)) \wedge (\forall i,j : 1 \leq i,j \leq n_z \ \wedge \ wz_i = wz_j \ \wedge \ i \neq j \Longrightarrow x_i \neq x_j)\}$$

So this second generalized greedy algorithm in which, at each step, useful elements are chosen and are deleted from their corresponding sets, can be seen as a compromise between the first greedy algorithm in which, at each step, useful elements are chosen but not deleted (explicitely) from their corresponding sets and the second greedy algorithm in which, at each step, (non necessarily useful) elements are chosen and deleted from their corresponding sets.

We now investigate how it's possible to specialize this second generalized greedy algorithm to the abstracted greedy and Huffman algorithms (see p.75 and p.69).

For the abstracted greedy algorithm, we have the following translations:

- INITIALIZATION $\leftrightarrow \ \sigma 1_A$
- $weight$ is defined as:

  $$weight : A_c \times A_{\hat{s}} \rightarrow A_{\hat{s}} : weight(x,Y) = weight'(x)$$

  in which function $weight'$ is the function $weight$ in the abstracted greedy algorithm defined on the ground elements. The fact that function $weight(x,Y)$ is independent of its second argument is due to the fact that the set $B_{\hat{s}}$ is a singleton and therefore, there is no choice for Y;
- $feasible(X) \ \leftrightarrow \ X \in \mathcal{F}$
- $EXT(\{B_s\}_{s\in S}) \leftrightarrow \{x|x \in B_c \times B_{\hat{s}} \ \wedge \sigma 2_A(x) \in \mathcal{F}\}$

Instanciation of the Second Generalized Greedy Algorithm (1)
begin
  $B_{\hat{s}} := \sigma 1_A$
  while $\{x|x \in B_c \times B_{\hat{s}} \ \wedge \sigma 2_A(x) \in \mathcal{F}\} \neq \emptyset$ **do**
  begin
    $e \in extrema(weight, \{x|x \in B_c \times B_{\hat{s}} \ \wedge \sigma 2_A(x) \in \mathcal{F}\})$
    $B_c := B_c \setminus \{e_1\}$

$B_{\hat{s}} := B_{\hat{s}} \setminus \{e_2\}$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 2_A(e)\}$
end
end

which is easily proved equivalent to the first greedy algorithm.
For the abstracted Huffman algorithm, we have the following translations:

- INITIALIZATION $\leftrightarrow$ $\{\sigma 2_{A'}(c) | c \in C\}$
- $weight \leftrightarrow weight^2$
- $feasible \leftrightarrow true$
- $EXT(\{B_s\}_{s \in S}) \leftrightarrow \{x | x \in B_{\hat{s}} \times B_{\hat{s}} \ \wedge \ x_1 \neq x_2\}$

With these instanciations, algorithm the second generalized greedy algorithm becomes:

Instanciation of the Second Generalized Greedy Algorithm (2)
begin
$B_{\hat{s}} := \{\sigma 2_{A'}(x) | x \in B_c\}$
while $\{x | x \in B_{\hat{s}} \times B_{\hat{s}} \ \wedge x_1 \neq x_2\} \neq \emptyset$ **do**
begin
$e \in extrema(weight^2, \{x | x \in B_{\hat{s}} \times B_{\hat{s}} \ \wedge x_1 \neq x_2\})$
$B_{\hat{s}} := B_{\hat{s}} \setminus \{e_1, e_2\}$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 3_{A'}(e)\}$
end
end

which is easily proved equivalent to the Huffman algorithm (see p.69).
Let us recall the fact that we could use a simplified definition of $weight^2$.
Something that has still to be done is to associate some necessary conditions (on $weight$, $feasible$, INITIALIZATION) that would guarantee the second generalized greedy algorithm to work. It would even be better to find an exact characterization on the set of all the structures on which the generalized greedy algorithm solves optimally the associated optimization problem (as it was done both for hereditary set systems and for accessible set systems respectively with matroid and matroid embedding).

6.4 A second generalizing step

An interesting idea would be to generalize the second generalized greedy algorithm to a similar algorithm in which we could use **all** the constructors associated to the data type and not only **one** basic constructor (which we called σz_A in Section 6.3). This idea is presented in the third generalized greedy algorithm:

Generalized Greedy Algorithm (3)
begin
$B_{\hat{s}}$:= INITIALIZATION
do $Condition1 \ \wedge EXT(1,\{B_s\}_{s\in S}) \neq \emptyset \ \rightarrow$
$e \in extrema(weight1, EXT(1,\{B_s\}_{s\in S}))$
$B_s := B_s \setminus \{e_i | 1 \leq i \leq n_1 \ \wedge \ w1_i = s\} \ \ \forall s \in S$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 1_A(e)\}$
$\vdots$
$\square$ $Conditionk \ \wedge EXT(k,\{B_s\}_{s\in S}) \neq \emptyset \ \rightarrow$
$e \in extrema(weightk, EXT(k,\{B_s\}_{s\in S}))$
$B_s := B_s \setminus \{e_i | 1 \leq i \leq n_k \ \wedge \ wk_i = s\} \ \ \forall s \in S$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma k_A(e)\}$
$\vdots$
$\square$ $Conditionr \ \wedge EXT(r,\{B_s\}_{s\in S}) \neq \emptyset \ \rightarrow$
$e \in extrema(weightr, EXT(r,\{B_s\}_{s\in S}))$
$B_s := B_s \setminus \{e_i | 1 \leq i \leq n_r \ \wedge \ wr_i = s\} \ \ \forall s \in S$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma r_A(e)\}$
od
end

in which we used the syntax of guarded command programs:

do $B_1 \ \rightarrow \ S_1$
$\square$ $B_2 \ \rightarrow \ S_2$
$\vdots$
$\square$ $B_n \ \rightarrow \ S_n$
od

which can be interpreted as follows: at each iteration of the loop, all the guards B_i (with $1 \leq i \leq n$) are evaluated and if they are all false, the loop

is skipped otherwise, an index j (with $1 \leq j \leq n$) is chosen such that B_j is true and the associated code S_j is executed.
This third generalized greedy algorithm is closely related to the second one (see p.83): we repeated for each constructor of the data type an adaptation of a copy of the code that we used in the second generalized greedy algorithm for the particular constructor that was only used in that algorithm (i.e. σz_A). We said "adaptation" because in the second generalized greedy algorithm, the particular code had explicit reference to the single constructor σz that was used and we had to modify those reference so that the code could be used for any constructor. For constructor σ_k, the code from the second generalized greedy algorithm:

$$\begin{array}{l} e \in extrema(weight, EXT(\{B_s\}_{s\in S})) \\ B_s := B_s \setminus \{e_i | 1 \leq i \leq n_z \ \wedge \ wz_i = s\} \ \ \forall s \in S \\ B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma z_A(e)\} \end{array}$$

in which $EXT(\{X_s\}_{s\in S})$ is defined as:

$$\begin{array}{lll} \{x | x \in X^{wz} & : & feasible(\sigma z_A(x)) \wedge \\ & & (\forall i,j : 1 \leq i,j \leq n_z \ \wedge \ wz_i = wz_j \ \wedge \ i \neq j \Longrightarrow x_i \neq x_j)\} \end{array}$$

was replaced by:

$$\begin{array}{l} e \in extrema(weightk, EXT(k, \{B_s\}_{s\in S})) \\ B_s := B_s \setminus \{e_i | 1 \leq i \leq n_k \ \wedge \ wk_i = s\} \ \ \forall s \in S \\ B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma k_A(e)\} \end{array}$$

in which $EXT(k, \{X_s\}_{s\in S})$ is defined as:

$$\begin{array}{lll} \{x | x \in X^{wk} & : & feasible(\sigma k_A(x)) \wedge \\ & & (\forall i,j : 1 \leq i,j \leq n_k \ \wedge \ wk_i = wk_j \ \wedge \ i \neq j \Longrightarrow x_i \neq x_j)\} \end{array}$$

Let us note that the function $weight$ as well has to be adapted for each constructor ($weightj$ for σ_j). Indeed, the local optimization associated to a constructor is most probably different than for another constructor[29].

[29] When the construtors have different domains, this probable feature becomes a certainty.

We also added some predicate expressions ($Conditionk$) that are intended to be necesseray conditions to allow the code associated to the application of constructor σ_k to be applied.
It's obvious to notice that the third generalized greedy algorithm is a generalization of the second one. Indeed, if we impose:

$$Conditionz = true \bigwedge \bigwedge_{\substack{1 \leq i \leq r \\ i \neq z}} Conditioni = false$$

the third generalized greedy algorithm becomes equivalent to the second one. This generalization has practical interest if we notice that both instanciations of the second generalized greedy algorithm (see p.85 and p.85) can be re-expressed more uniformly by instanciating the third generalized greedy algorithm.
This new expression of both algorithms is more uniform for two main reasons:

- a single constructor σz don't have to be "chosen" any more but all the constructors play a similar role;
- the initialization step is now solved by an instanciation of the greedy principle associated to one particular constructor[30].

Let us now give a version of both instanciations of the second generalized greedy algorithm as instanciations of the third one.

Instanciation of the Third Generalized Greedy Algorithm (1)
begin
$B_{\hat{s}} := \{\}$

[30] We could already notice that in the first (resp. second) instanciation of the second generalized greedy algorithm, the initialization step was associated to constructor $\sigma 1_A$ (resp. $\sigma 2_{A'}$).

do $Condition1 \ \wedge \ \{\langle\rangle\} \neq \emptyset \ \rightarrow$
$e := \langle\rangle$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 1_A(e)\}$
$\square$ $Condition2 \ \wedge \ \{x | x \in B_c \times B_{\hat{s}} \ \wedge \sigma 2_A(x) \in \mathcal{F}\} \neq \emptyset \ \rightarrow$
$e \in extrema(weight, \{x | x \in B_c \times B_{\hat{s}} \ \wedge \sigma 2_A(x) \in \mathcal{F}\})$
$B_c := B_c \setminus \{e_1\}$
$B_{\hat{s}} := B_{\hat{s}} \setminus \{e_2\}$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 2_A(e)\}$
od
end

Constructor $\sigma 1$ has $\{\langle\rangle\}$ as domain. This very simple domain causes many possible simplifications. $EXT(1, \{B_s\}_{s \in S})$ becomes $\{\langle\rangle\}$ and thus the condition $EXT(1, \{B_s\}_{s \in S}) \neq \emptyset$ is always satisfied. Moreover, we don't have to define a particular translation for $weight1$ as there is only one element in the set in which we want to find an optimum. This is the reason why instead of the expression $e \in extrema(weight1, EXT(1, \{B_s\}_{s \in S}))$, we have $e := \langle\rangle$.
Another important characteristic is provided by the expressions $Conditioni$ $(i = 1, 2)$. Their translation would define the control flow for the loop. We could choose the following translations:

- $Condition1 \ \leftrightarrow \ B_{\hat{s}} = \emptyset$
- $Condition2 \ \leftrightarrow \ B_{\hat{s}} \neq \emptyset$

which means that, at the first iteration, $\sigma 1$ will be chosen and at each iteration later, it will be $\sigma 2$. This often corresponds to the fact that a first step is assigned to the initialization process. Indeed, in this particular example, the expression $B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 1_A(e)\}$ under condition $B_{\hat{s}} = \emptyset$ can be simplified to $B_{\hat{s}} := \{\sigma 1_A(e)\}$ which is exactly the initialization in the first initialization of the second generalized greedy algorithm (see p.85). This fact shows that the initialization process can often be seen as a particular instanciation of the greedy principle.
Finally, let us notice that $Condition2$ (i.e. $B_{\hat{s}} \neq \emptyset$) could be simplified to $true$ as we have the following implication:

$$\{x | x \in B_c \times B_{\hat{s}} \ \wedge \ \sigma 2_A(x) \in \mathcal{F}\} \neq \emptyset \implies B_{\hat{s}} \neq \emptyset$$

Here follows a version of the second instanciation of the second generalized greedy algorithm (see p.85), as an instanciation of the third generalized greedy algorithm.

Instanciation of the Third Generalized Greedy Algorithm (2)
begin
$B_{\hat{s}} := \{\}$
do $Condition1 \ \wedge \ \{\langle\rangle\} \neq \emptyset \rightarrow$
$e := \langle\rangle$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 1_A(e)\}$
$\square$ $Condition2 \ \wedge \ B_c \neq \emptyset \rightarrow$
$e \in extrema(weight2, B_c)$
$B_c := B_c \setminus \{e\}$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 2_A(e)\}$
$\square$ $Condition3 \ \wedge \ \{x | x \in B_{\hat{s}} \times B_{\hat{s}} \ \wedge x_1 \neq x_2\} \neq \emptyset \rightarrow$
$e \in extrema(weight3, \{x | x \in B_{\hat{s}} \times B_{\hat{s}} \ \wedge x_1 \neq x_2\})$
$B_{\hat{s}} := B_{\hat{s}} \setminus \{e_1, e_2\}$
$B_{\hat{s}} := B_{\hat{s}} \cup \{\sigma 3_{A'}(e)\}$
od
end

In this second instanciation of the third generalized greedy algorithm, $weight2$ represents any function having A_c as domain (i.e. the particular oder in which the elements from B_c are chosen is quite unimportant) and $weight3$ corresponds to the function $weight^2$ (see p.76).
As for the data flow, we could choose the following translations:

- $Condition1 \ \leftrightarrow \ B_c = \emptyset \ \wedge \ B_{\hat{s}} = \emptyset$
- $Condition2 \ \leftrightarrow \ B_c \neq \emptyset$
- $Condition3 \ \leftrightarrow \ B_c = \emptyset \ \wedge \ |B_{\hat{s}}| > 1$

which means that in almost all cases (i.e. if there is at least one character), the second constructor is executed during a certain number of steps and the third constructor at each later step and if there is no character in the initial file, then there is only one iteration at which the first constructor is applied.

Let us note that the initialization of the second instanciation of the second generalized greedy algorithm (see p.85) is now carried out, in almost all cases, by successive applications of the second constructor and, in a very particular case, by the application of the first one. So, in this case as well, the intialization became a particular case of of the greedy principle for some constructor. But moreover, in this example, we notice that:

- different initializations (depending on some initial conditions) are treated uniformally;
- an initialization step can be in fact achieved thanks to the execution of several steps of a loop, as it's done with successive applications of the code associated to the second constructor. This remark is coherent with the one statement initialization step of the second instanciation of the second generalized greedy algorithm because that single statement was in fact already hidding a loop.

The simplifiations in the code associated to the first constructor have already been explained for the same identical constructor in the first instanciation of the third generalized greedy algorithm.
Moreover, as it was done in the first instanciation of the third generalized greedy algorithm, $Condition3$ can be simplified to $B_c = \emptyset$ because the other part in the condition is implied by $EXT(3, \{B_s\}_{s \in S})$.
The instanciation of the third generalized greedy algorithm to our two main examples raises some open questions:

- the third generalized greedy algorithm allows some non-determinism which was not used in both instanciations (all the different conditions $Conditioni$ were always mutually exclusive). Does there exist any example in which non-determinism could be usefully used ?
- the third generalized greedy algorithm uses $B_{\hat{s}} :=$ INITIALIZATION as initialization and in both instanciations, it has been replace by $B_{\hat{s}} := \{\}$. Is it a general rule ? Of course an initialization always uses some constructors to create an object of type $A_{\hat{s}}$ and therefore we could think that the associated code of that constructor could perform the initialization. However, it's hard to be intuitively convinced that each initialization can be expressed by the schema which is associated to each constructor;

- the most important question is probably: given the formal specification of a problem, which data type do we choose in order to solve that problem and moreover, for this particular data type which set of constructors should we use. Indeed, the data type as well as the set of constructors chosen could influence the efficiency of the resulting algorithm but also simply the own existence of such an algorithm. For instance, for the data compression problem, the choice the binary tree data type was not a priori evident;

- a very important question is to find the translations of the conditions $Conditionj$ (with $1 \leq j \leq r$) strong enough for the associated code to be defined and useful to be applied and weak enough to allow the entire loop to find a global solution before exiting the loop;

- it would also be very useful to develop a design tactic in order to rationally help the process of synthesizing a program, based on the third generalized greedy algorithm, solving a given problem;

- it would be nice to find necessary conditions on some parts of the third generalized greedy algorithm (the set of constructors, the functions $weighti$, the expressions $Condition_i$ and so on) in order to guarantee that the algorithm is fully correct (i.e. it eventually stops and indeed finds an object which is globally optimal). If those conditions are also sufficient, then we would have found an exact characterization of the structures on which the third generalized greedy algorithm is correct;

7 Conclusion

Let us first recall that we have already given some conclusions at the end of some sections. Thus, we shall focus here on the main conclusions concerning the different goals that we proposed in the introduction.
First, we wanted to help the design of greedy programs from formal specifications. We couldn't find a fully automatic procedure in order to synthesize greedy programs. However, we could identify a sequence of four different steps in the synthesis process. Some of these steps were solved automatically and some others were easily solved semi-automatically. In the step consisting of verifying that the axioms of the *Greedy Algorithms* theory are translated to

theorems in the problem theory, it was easy for all axioms but one (the augmentation or the cardinality axiom) which was quite hard to verify. We tried to find some general techniques that can be applied to tackle this problem. We found two different techniques that can be applied to several problems but not all.

In the future, we would like to systematize even more the design process of greedy algorithms by tackling the main problem bottleneck which seemed to consist of verifying the augmentation or the cardinality axiom. Moreover, we would like to implement our design process in a synthesis environment such as KIDS [Smi90] and to apply it to a larger sample of examples in order to validate practically our results. For instance, we would like to classify the "matrix chain product problem" as a greedy algorithm altough it's very usually located in the dynamic programming chapter in the textbooks about algorithms (e.g. [BB87, CLR90]). We would like also to broaden our study by dealing with the synthesis process of greedy algorithms from specifications of problems underlain not only by a matroid but also by a greedoid or by a matroid embedding.

Secondly, we generalized the set system formalism (on which are based matroid, greedoid and matroid embedding) that is dependent on the particular data type *set* in another formalism which is independent of any data type and which is based on the algebraic setting. A first generalization was obtained as an abstraction of both the original greedy algorithm and the Huffman algorithm. Then, a second generalization was introduced. This last generalization has the advantage of considering all the constructors of the data type uniformly and the intialization step becomes an instanciation of a greedy process.

In the future, we shall try to find some necessary conditions over the different expressions of the generalized greedy algorithm (e.g. $weighti$, CONDITIONI, σi_A) that would guarantee the correctness of the algorithm. Then, we would like to find as many examples of algorithms that can be expressed either as our first or second generalization. Later, we would also like to study the synthesis process of greedy programs based on these two generalizations.

Other more general works that could be done in the future consist of:

- considering that the greedy principle can also be used as a heuristic to find efficiently an approximation of a global optimum;
- looking for subclasses of existing theories (matroid, greedoid, matroid

embedding and the two generalizations) among which the synthesis process would be easier than in the whole class;

- finding the analogy of our two generalizations (which are "matroid oriented") for greedoid and matroid embedding;

- formalizing some applications of the greedy principle that often appears embedded within other algorithms. For instance, Bland's anticycling rule [Mor94] ("whenever there is a choice of entering or leaving variable, we always choose the one with smallest subscript") is often added to the simplex algorithm to prevent cycles;

- applying the *Greedy Algorithms* to some (a priori) non-optimization problems (i.e. problems in which the purpose is to find a feasible solution): for instance, we think that the way to get out of a two dimensional maze found by the greeks: "You must always follow the wall located at your right (resp. left)" could be seen as an application of the greedy principle. Therefore, the *Greedy Algorithm* would have some links with the backtracking technique and thus maybe some with *Global Search*. Another well-known example is the problem of "coin changing" which is related to greedy algorithms [Wri75, CK76].Others examples may be found in [AM84].

- combining different classes of algorithms (such as *Dynamic Programming* and the *Greedy Algorithms* which as already been studied [BdM92, BdM93]) could be very interesting because on the first hand the synthesis of some algorithms might require to use different classes at different steps of the process and on the other hand there certainly exist some "hybrid" algorithms based partly on several classes without belonging to one of them;

- to study thoroughly the similarities and differences between the *Greedy Algorithms* and *Local Search* that were shortly presented in the Appendix. For instance, we shall try to investigate if the local optimization done by the *Greedy Algorithms* cannot be expressed on the resulting objects that could be created at each step (as it's done in *Local Search*) instead on the subobjects (as it's done actually);

- considering the greedy principle as a second order theory (a control theory) which can be applied to the many algorithm theories that are based on the fact that the final solution is reached incrementally. For instance, if we would apply the greedy principle on the classes: *Global Search*, *Local Search* or *Problem Reduction Generators* (see Appendix), it would more or less lead repsectively to the following program schemes: Global Search without backtracking, steepest descent (or ascent) and Problem Reduction Generator with the possibility to choose an optimal reduction without having to investigate all possible reductions. It would be a very interesting and useful work to formalize the greedy principle as such a second order theory;

- studying other possible applications of the greedy algorithm. For instance, the simulated annealing approach which generalizes the greedy principle by the possibility of probabilistically doing a local choice which is not the best one. Another direction would be to ponder on the possible use of the greedy philosphy for parallel algorithms [AM84].

Acknowledgments

To be written !

Appendix: Short description of the different algorithm theories

The basic idea of *Global Search* is to consider a **set** containing (possibly implicitely) at least **all** the candidate solutions of the problem. This set is then recursively split into subsets. For each of these subsets, we either try to find the optimal[31] solution it contains or, if it is not possible, we look for some general features valid for all the solutions of the subset. Basically, these features consist of upper or lower bounds on the value of the solutions. A subset is not split any more either when we have found its optimal solution or when we know (thanks to the features derived for instance) that it cannot contain an optimal solution of the initial problem. The process is finished when no sets remain to be split.
Local Search is based on the existence of a neighborood function which associates to each candidate solution a set of candidate solutions that are called its neighboors. The idea of the *Local Search* philosophy is to start to search from an arbitrary candidate solution and to travel in the space state from neighboors to neighboors until a candidate satisfying some condition is reached. As this condition is often expressed as being better than all its neighboors, the *Local Search* strategy obviously lead to a local optimum but not always to a global optimum. Compared to *Global Search*, the *Local Search* strategy differs by the main fact that it considers one single solution at a time and not sets of solution as it is the case for *Global Search*.
We can easily notice that the *Local Search* and the *Greedy Algorithms* techniques look very similar. We could give two principles for the Local Search philosophy as we did (see Section 2) for the greedy philosophy:

- the algorithm **finds** the final solution after a sequence of steps;
- at each step, a **local choice** is made.

The difference for the first principle comes from the fact that, in the greedy philosophy, the algorithm does more than finding the solution, it **constructs** it ! The second principle of the greedy philosophy can be seen as a particular case as the one of the local search philosophy: the local choice done is always the **best** one in the greedy algorithms.

[31] With respect to the solutions in the subset only.

We can also notice other differences between these two strategies:

- *Local Search* improves the weight of the partial solution at each iteration which is not the case for the *Greedy Algorithms* which choose the best neighboor (which is not necessary better than the previous partial solution);

- the local optimization in *Local Search* is defined on the feasible solutions and in the *Greedy Algorithms*, it is defined on the subobjects used in order to form the new feasible solutions;

Although *Local Search* and the *Greedy Algorithms* may look a priori very similar, we think that it might be so for some particular view of the *Greedy Algorithms* restricted to matroid for instance but it is certainly no more the case for the different generalizations we made.
Therefore, the two different philosphies the we recall:

- to travel in the space state from feasible solutions to feasible solutions (*Local Search*);

- to construct incrementally a feasible solution (*Greedy Algorithms*);

are basically different (although quite similar).
The *Divide and Conquer* technique first decomposes a large problem into several subproblems, then looks for solutions of these subproblems and eventually compose the results of the subproblems in order to get a solution of the initial problem. When the generated subproblems are small enough, they can be solved by any brute force strategy otherwise, the decomposition is often applied recursively.
Dynamic Programming is similar to *Divide and Conquer* because its idea is to reduce the initial problem in subproblems from which, once solved, the solution of the initial problem will be easily found. However, in this case, there are often different possible reductions for a problem and we do not know a priori which one leads to the optimal solution. Therefore, all the different reductions are carried out and once they are all solved, we can find which potential solution each reduction would lead to. Thus, by chosing the best one among these potential solutions, we find the optimal solution of the initial

problem. As different reductions often generate identical (small) subproblems, the direct application of this technique is very inefficient. To avoid this drawback, the small subproblems are often first solved adn then composed in order to solve greater problems and so on... So *Dynamic Programming* is often implemented by an ascendant method, as opposed to *Divide and Conquer* which is often implemented by an descendant method. Another difference comes from the fact that *Divide and Conquer* does only useful job: solving all the reduced subproblems is necessary in order to find the solution of the initial problem. As opposed, for *Dynamic Programming*, among all the different possible reductions, only one should be sufficient in order to solve the initial problem. However, we do not know a priori which one it is and we are forced to try all possible reductions which implies that a lot of unuseful job is carried out.

Problem Reduction Generators is basically similar to *Dynamic Programming.*

References

[AM84] R. Anderson and E.W. Mayr. Parallelism and greedy algorithms. Technical Report STAN-CS-84-1003, Dept. of Computer Science, Stanford University, 1984.

[BB87] G. Brassard and P. Bratley. *Algorithmique. Conception et analyse*. Masson, 1987.

[BdM92] R.S. Bird and O. de Moor. Between dynamic programming and greedy : data compression. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, 1992.

[BdM93] R.S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller H. Partsch S. Schuman, editor, *Formal program development: IFIP TC2/WG 2.1 State of the Art Report*, volume 755 of *LNCS*, pages 43–61. Springer-Verlag, 1993.

[BGG+94] L. Blaine, L.M. Gilham, A. Goldberg, R. Jüllig, J. McDonald, and Y.V. Srinivas. Slang user manual (unpublished), 1994.

[Bir92a] R.S. Bird. Two greedy algorithms. *Journal of Functional Programming*, 2(2):237–244, 1992.

[Bir92b] R.S. Bird. Unravelling greedy algorithms. *Journal of Functional Programming*, 2(3):375–385, 1992.

[BW90] M. Barr and C. Wells. In C.A.R. Hoare, editor, *Category Theory for Computing Science*. Prentice Hall International, 1990.

[CK76] L. Chang and J. Korsh. Canonical coin changing and greedy solutions. *JACM*, 23(3):418–422, 1976.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[dM94] O. de Moor. Categories, relations and dynamic programming. *MSCS*, 4:33–70, 1994.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.

[Edm71] J. Edmonds. Matroids and the greedya algorithm. *Math. Programming*, 1:127–136, 1971.

[EM85] H. Ehrig and B. Mahr. In W. Brauer G. Rozenberg A. Salomaa, editor, *Fundamentals of Algebraic Specifications 1. Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[Hel89a] P. Helman. A common schema for dynamic programming and branch and bound algorithm. *Journal of the ACM*, 36(1):97–128, 1989.

[Hel89b] P. Helman. A theory of greedy structures based on k-ary dominance relations. Technical Report CS89-11, Dept. of Computer Science, University of New Mexico, 1989.

[HMS93] P. Helman, B.M.E. Moret, and H.D. Shapiro. An exact characterization of greedy structures. *SIAM J. Disc. Math.*, 6(2):274–283, 1993.

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Pitman Publishing Limited, 1978.

[Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[Jeu92] J. Jeuring. *Theories for algorithm calculation*. Ph.D. Thesis, Rijksuniversiteit Utrecht, 1992.

[KL81] B. Korte and L. Lovász. Mathematical structures underlying greedy algorithms. In *Fundamentals of Computation Theory*, volume 177 of *LNCS*, pages 205–209. Springer-Verlag, 1981.

[KL84] B. Korte and L. Lovász. Greedoids and linear objective functions. *SIAM J. Alg. Discrete Meth.*, 5:229–238, 1984.

[KLS91] B. Korte, L. Lovász, and R. Scharder. In R.L. Graham B. Korte L. Lovász, editor, *Greedoids*, volume 4 of *Algorithms and Combinatorics*. Springer-Verlag, 1991.

[Kru56] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proceedings of the American Mathematical Society*, volume 7, pages 48–50, 1956.

[Low91] M.R. Lowry. Automating the design of local search algorithms. In M.R. Lowry R.D. McCartney, editor, *Automating Software Design*, pages 515–546. MIT-Press, 1991.

[MB67] S. McLane and G. Birkhoff. *Algebra*. The Mc Millan Company, 1967.

[Mee86] L. Meertens. Algorithmics – towards programming as a mathematical activity. In J.W. de Bakker M. Hazewinkel J.K. Lenstra, editor, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North-Holland, 1986.

[Mor94] P. Morris. In J.H. Ewing F.W. Gehring P.R. Halmos, editor, *Introduction to Game Theory*, Universitext. Springer-Verlag, 1994.

[MS90] B.M.E. Moret and H.D. Shapiro. *Algorithms form P to NP. Design & Efficiency*, volume I. The Benjamin/Cummings Publishing Company, 1990.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[Poi92] A. Poigné. Basic category theory. In S. Abramsky D.M. Gabbay T.S.E. Maibaum, editor, *Handbook of Logic in Computer Science*, volume 1 Background: Mathematical Structures of *Oxford Science Publications*. Clarendon Press, 1992.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[RW88] C. Rich and R.C. Waters. Automatic programming : myths and prospects. *IEEE Computer*, 21(8):40–51, 1988.

[SL90] D.R. Smith and M.R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2-3):305–321, 1990.

[Smi83] D.R. Smith. The structure of divide and conquer algorithms. Technical Report NPS 52-83-002, Naval Postgraduate School, 1983.

[Smi85a] D.R. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5(1):37–58, 1985.

[Smi85b] D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[Smi87a] D.R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987.

[Smi87b] D.R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, 1987.

[Smi90] D.R. Smith. Kids - a semi-automatic program developement system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[Smi91] D.R. Smith. Structure and design of problem reduction generators. In B. Möller, editor, *Constructing Programs from Specifications*, pages 91–124. North-Holland, 1991.

[Smi92] D.R. Smith. Constructing specifications morphisms. Technical Report KES.U.92.1, Kestrel Institute, 1992.

[Smi93] D.R. Smith. Classification approach to design. Technical Report KES.U.93.4, Kestrel Institute, 1993.

[ST95] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development.* Cambridge University Press, (to appear) 1995.

[Wri75] J.W. Wright. The change-making problem. *JACM*, 22(1):125–128, 1975.