Hitch hiker's Guide to ESQIMO

Computational model for analogy solving in IQ-tests
May 1998, ESQIMO version 2.0

Erika Valencia

erika@lri.fr
Off 109, LRI, ura 410,
Bât 490,Université Paris-XI,
91405 Orsay Cedex
France
+33 1 69 15 42 25
http://www.lri.fr/~erika/HTML/esqimo.html

Abstract

ESQIMO is a computational model for analogy solving based on a *topological* formalism of representation. The source and the target analogs are represented as *simplexes* and the analogy solving is modeled as a topological *deformation* of these simplexes along a *polygonal chain* and according to some constraints. We apply this framework to the resolution of IQ-tests typically presented as *"given A, B and C, find D such that A is to B what C is to D"*.

Keywords : Knowledge Representation, Diagrammatic Reasoning, Simplicial complexes, Analogy solving, CAT.

Introduction

ESQIMO is a system that automatically solves the following type of IQ-tests: *Given three figures A, B and C, find a figure D such that D is to C what B is to A.* These IQ-tests are a paradigmatic example of *analogy solving* problems.

ESQIMO is thus a computational model for analogy solving. The framework used by ESQIMO to solve analogies is called *CAT* for Combinatorial Algebraic Topology. More precisely, ESQIMO models knowledge with *simplicial complexes* and reasoning with topological tools that are applicable to them. For instance, the resolution of an analogy is modeled as the topological transformation of simplexes along polygonal chains into a complex. This model is part of a larger research program, called *UTopoIa* [1] that proposes topological models for diagrammatic reasoning.

ESQIMO can be considered at different levels. It can be seen as an IQ-test solver and you can try many different tests and compare the results with yours. You can even create your own universes of elements to express new IQ-tests with other properties. The first chapter of this guide includes a few examples of resolution to make you understand ESQIMO's philosophy.

If you want to see ESQIMO as a model for analogy solving [VG98], you may want to read chapter 3. This chapter presents ESQIMO's algorithm for analogy solving on the basis of a topological representation of knowledge. Many heuristics have been implemented at different levels of the algorithm and are discussed. You can change them to test the differences.

ESQIMO is more than *yet another computational model for analogy*, it is one application of a larger framework developed in the UTopoIa project. UTopoIa aims to propose topological models in the field of diagrammatic representation and reasoning. If you are interested in how and why we propose such a framework for diagrammatic reasoning, you may want to read chapter 2. We briefly introduce our system of knowledge representation based on simplicial complexes. Simplicial complexes are an abstract structure introduced in algebraic topology and you will hear about simplicial complexes all along this guide so you may need its rigorous definition for a better understanding.

Finally, ESQIMO has a material incarnation as a running program, and you may be interested in its structure. In that case, you have to read chapter 4. ESQIMO 1.0 has been implemented in the MATHEMATICA[2] programming language and the version 2.0 that is referred to in this guide has been implemented in the CAML programming language. The library developed for the manipulation of simplicial complexes have been reused for other applications and is presented in details so that you can use it too for other developments.

[1] See [GV98] or `http://www.lri.fr/~erika/HTML/UTopoIa.html` for more details on the UTopoIa project.

[2] This version is also available at `http://www.lri.fr/~erika/HTML/esqimo.html` and the technical report of my DEA training period is a good manual to this version.

Contents

Chapter 1

Using ESQIMO

1.1 Copyright and Installation

ESQIMO is Copyrighted by Erika Valencia from the UTopoIa project at LRI laboratory, Université Paris-Sud, Orsay, France. UTopoIa web site is:
http:/www.lri.fr/˜erika/HTML/UTopoia.html

ESQIMO's sources are freely available for download at:
http://www.lri.fr/˜erika/HTML/esqimo.html

If you are having any problem using ESQIMO, you can contact its developer at *erika@lri.fr* but there is no warranty for the program. ESQIMO can not be used for commercial purposes. If you desire to propose a new version or new modules for ESQIMO, you must inform Erika Valencia at *erika@lri.fr* and make sure that these changes lie within ESQIMO's philosophy. For any new version, the sources must be freely available too and include this copyright.

ESQIMO has been developed with the Caml Light system, compiler and linker version 0.74. For further information, you can see the CaML web site at:
http://www.pauillac.inria.fr/caml/index-fra.html

Downloading the ESQIMO system leaves you with a compressed archive file `Esqimo-2.0.tar.gz`. Once decompressed and extracted, you get a directory called `Esqimo-2.0` containing the following files:

- `esq-kernel.ml` and `.mli`, `.zi` interfaces: the ESQIMO kernel ;
- `complex-lib.ml` and `.mli` , `.zi` interfaces: libraries for the manipulation of lists, chains, simplexes and complexes (these concepts are introduced in chapter 2 and the corresponding `CaML` types are presented in chapter 4);
- `univers-geo.ml` and `.mli` , `.zi` interfaces: universe of the elements of IQ-tests manipulated by ESQIMO (these elements are geometrical and they are introduced in chapter 3);
- `Makefile` and `.Makefile.Dep`.

Once esqimo has being compiled with the make command, a Caml light interface is automatically launched. If esqimo is already compiled, you can launch it by typing esqimo directly.

```
sun7j 40 > make
camlmktop -o esqimo  complex-lib.zo esq-kernel.zo univers-geo.zo
camllight esqimo
>       Caml Light version 0.74
#
```

The Camllight interface prompt is #. In this interface, all the commands you will use are in fact CaML functions. But don't worry, you don't need to know CaML to use ESQIMO, all the useful functions are detailed in this guide and are mainly ESQIMO functions. All you need to be careful about is syntax. In particular, do not mix upper and lower case letters or underscore and dash line. Finally, every line of command, even if it is with ESQIMO functions must finish with a double semicolon (like this: ;;).

To use ESQIMO, you must first open the univers-geo, esq-kernel, complex-lib, sort, set files and libraries with the instructions:

```
##open "sort";;
##open "set";;
##open "univers-geo";;
##open "complex-lib";;
##open "esq-kernel";;
#
```

Once these files are opened, you can start to ask for IQ-tests resolutions. All the examples presented in what follows take elements from the universe univers-geo already implemented. You can look into this file to see how the elements are defined.

1.2 Examples of resolution

1.2.1 Presentation of an IQ-test

Consider that we want to model an IQ-test that is presented to a subject with the sentence:

Given three figures A, B and C, find a fourth figure D such that it is to C what B is to A.

The three figures A, B and C are presented and usually five solutions are suggested and the subject has to choose the *right*[1] one. Here, we give no solutions to choose and ESQIMO has to build one from scratch. We call this kind of IQ-test, *unsupervised*. In the univers-geo universe, IQ-tests are given with geometrical figures. We have only twelve figures that materialize all the possible combinations of

[1] See section 1.2.3 for a discussion on the existence of a *right* solution.

seven properties. The set of properties is P and the set of elements is E (see also figure 1.1):

$$P = \{\text{round, square, triangle, black, white, small, big}\} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$$

$$E = \{e_1, ..., e_{12}\}$$

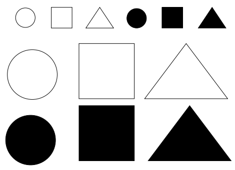


Figure 1.1: Elements of the universe Ω of ESQIMO, respectively called e_1 to e_{12} starting from the top left element

To see the composition of one element in terms of properties, simply type:

```
#elements e1;;
- : int list = [1; 5; 6]
```

Here, the element `e1` has the properties `[1; 5; 6]`, the property `1` corresponds to roundness, the property `5` to a white color, and the property `6` to a small size. This element is thus a small white circle. More details about the geometrical world of ESQIMO are given in chapter 3. For simplicity, we gave the corresponding drawing for each test.

1.2.2 A Simple Example to Use the Interface

In this first example, we will try to ask ESQIMO to solve the IQ-test on figure 3.6. We will define the figures A, B and C, and we expect ESQIMO to answer the figure D seen on figure 3.6.

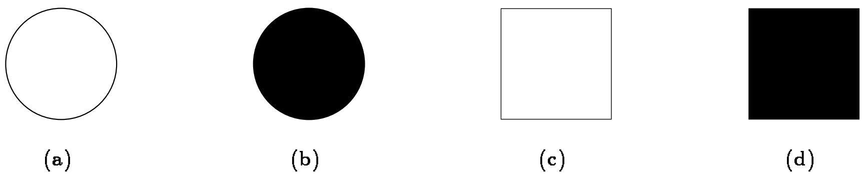


Figure 1.2: Figures A, B and C are defined and we expect ESQIMO to answer the figure D. The transformation that we expect ESQIMO to find could be that the color white is replaced by black, and shapes and sizes are left unchanged

We want to define A as a white small circle, that is: $A = \{e1\}$. So we write in the Camllight interface:

```
#let A=e1;;
A : int t = <abstr>
```

And we can verify that it is indeed composed of the same properties than e_1:

```
#elements A;;
- : int list = [1; 5; 6]
```

The same way, we define the figures B and C and verify their properties:

```
#let B=e4 and C=e2;;
B : int t = <abstr>
C : int t = <abstr>
#elements B;;
- : int list = [1; 4; 6]
#elements C;;
- : int list = [2; 5; 6]
```

The figure B has indeed the properties of small (6), circle (1) and black (4) while C has the properties of small (6), square (2) and white (5). To ask ESQIMO to solve this test we must use the function `Resoud-simple`, more details about the definition of this function are given in chapter 4. To use this function, you have to specify first the name of the *complex representing all the possible figures* of your universe and then respectively the figures A, B and C. We call *complex representing all the possible figures* the universe of all the figures ESQIMO is supposed to know to solve the analogies. They are representative of all the properties we chose. In `univers-geo`, this complex is called `omega`. It is simply defined as containing all the elements `e1` to `e12` of E. To see the elements of `omega` just type:

```
#map elements (elements omega);;
- : int list list =
 [[1; 4; 6]; [1; 4; 7]; [1; 5; 6]; [1; 5; 7]; [2; 4; 6]; [2; 4; 7];
  [2; 5; 6]; [2; 5; 7]; [3; 4; 6]; [3; 4; 7]; [3; 5; 6]; [3; 5; 7]]
```

Indeed, if you only type:

```
#elements omega;;
- : int t list =
 [<abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>; <abstr>;
  <abstr>; <abstr>; <abstr>; <abstr>]
```

You will only get a list of abstract structures (in fact simplexes) but without knowing what they contain. When you use the command `elements`, you ask for the `list` of all the elements of a set, when applied to a simplex, you get its elements(its properties), when applied to a complex you get other sets inside. A complex is a set of sets. This remark gives you an intuition of what are simplicial complexes, you will discover them in more details in chapter 2. Now we can ask ESQIMO to solve our test:

```
#let D=resoud_simple omega A B C;;
- : int t = <abstr>
```

The structure of the solution is a simplex, to see the properties that composes D, we type:

```
#elements D;;
- : int list = [2; 4; 6]
```

The solution proposed by ESQIMO is composed of the properties `[2; 4; 6]`, that is a black small square, which corresponds to the element `e6` expected.

In conclusion to this example, we see that ESQIMO decomposes each figure in properties and transforms the property that changed between A and B. Here he took off the property `5` and added the property `4`. This transformation can be considered from different points of view that are discussed in chapter 2 and that lead to different heuristics in the algorithm. An interpretation of the transformation used here could be *take off the color white, put a color black, leave the rest unchanged.* Finally you may find that a resolution is pretty slow but there are many topological operations performed so don't worry, it is absolutely normal[2].

1.2.3 Different Solutions are Acceptable

We performed a non representative series of tests on 10 humans in [Val97]. If we ask different humans to solve the same IQ-tests, we find that there are many solutions that are justifiable in terms of a *right* transformation. In principle, there are no *good* or *bad* solution.

In traditional IQ-tests, there *is* a good solution in the sense that we give five solutions[3] and for four among them it is impossible to find a correct justification. It is because the test is *made to test* this specific capacity of humans to *detect* a good transformation when presented that we consider that there is a good solution. It is good by opposition or by contrast to the others that are *impossible.*

But if no choices are given, many solutions are *possible* and thus can be considered as acceptable. In this case, it is the simplicity or naturality of a transformation that can be different from a person to another. It just depends on the point of view.

For example, transforming A into B in the preceding example (see figure 3.6) can be considered as "*transforming a white circle into a black circle only if it is a circle*", or "*transforming everything white into black*", or "*changing the color*", or etc. Depending on the way you express the transformation, it will give you different solutions. Applying the first formulation of the transformation to a white square, leaves the square unchanged, applying the second one transforms it into a black square, applying the third transforms it into a white square. But if it was applied to a black square, the two first formulations would leave it black and the last one would transform it into a white square, and so on.

1.2.4 More Examples

It is possible to use the following function to put back in correspondence one simplex with its name in the set E:

```
#let nom_de =
        function [1;5;6] -> "e1"
```

[2] Optimizations are under study.

[3] Five for example, detect the ANALOGY system in [Eva68] for such a protocol

```
    | [2;5;6] -> "e2"
    | [3;5;6] -> "e3"
    | [1;4;6] -> "e4"
    | [2;4;6] -> "e5"
    | [3;4;6] -> "e6"
    | [1;5;7] -> "e7"
    | [2;5;7] -> "e8"
    | [3;5;7] -> "e9"
    | [1;4;6] -> "e10"
    | [2;4;6] -> "e11"
    | [3;4;6] -> "e12"
    | _ -> "inconnu";;
```

This function has to be defined in the `univers-geo.ml` file since it is universe dependent. For that reason it is not included into the function `resoud_simple`. here is a first example:

```
#nom_de(elements(resoud_simple omega e1 e5 e4));;
- : string = "e5"
```

As shown on figure 1.3, we propose a an example of the transformation, the couple *little white round* transformed into a*little black square* and we ask ESQIMO to apply it to a *little black round*. The property of whiteness is replaced by blackness and the property of roundness is replaced by squareness in the pair $A - B$. Therefore, when applied to C, we get around changed into a square and the color is left black. We can say that any color is turned into black or that the color is changed only if it was white. In this case, ESQIMO uses a heuristic that consists in turning any color to black. To see if the shape would have been anyway by a square, we ask the same

Figure 1.3: Color is turned to black and shaped is replaced by square

test but with C being a triangle, we show the result on figure 1.4:

```
#nom_de(elements(resoud_simple omega e1 e5 e3));;
- : string = "inconnu"
```

Here, the element proposed by ESQIMO as a solution is said to be `inconnu` which means *unknown*. This occurs each time that we have an element that do not correspond to a defined element $e_i \in E$. But to see why ESQIMO answered that, we look at the components of this solution:

```
#elements(resoud_simple omega e1 e5 e3);;
- : int list = [2; 3; 4; 6]
```

This solution contains 4 properties! two of them are properties concerning the shape, in other terms, this solution is a little black triangle-square!! ESQIMO doesn't know that if one figure has already one property of shape, then it is impossible to have another one, at least for us. This is an interesting property of ESQIMO and depending on the domain, it can be suitable or not to keep it. In our domain, we would probably prefer to get rid of these *monsters*, and it is possible by using a different heuristic in the application of the transformation to C. A larger discussion about these heuristics, their choices, their implications and implementation is presented in chapter 2.

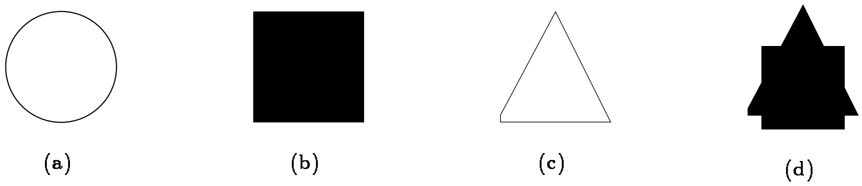


Figure 1.4: Color is turned to black and shaped is replaced by square

1.2.5 Examples with Composed Figures

Until now we have always used the resolution function `resoud_simple`, which means *simple resolve*. We used it because all the preceding examples where *simple* examples in the sense that they contained only one element each. Let's look at composed examples where the figures may contain more than one element, we will use the function `resoud_compose`.

In the composed case, there is one more step before the resolution of the analogy. Since the figures contain more than one element, they are no longer represented as one simplex but as a set of simplexes. Thus, transforming A into B is now transforming a set of simplexes into another. We do such a thing by pairing the simplexes of A with simplexes of B and then we apply a simple resolution to each pair. But the things are also more complicated when C is also composed, because each pair-transformation has to be applied to each simplex of C, by another pairing process. These steps are detailed in chapter 3.

We can illustrate this phenomena of pairing by the following example. Two different elements, `e1` and `e2` of the figure A, are transformed into the same element `e2` repeated twice. The system will pair these elements by two, the first of figure A with the first of figure B for example and consider separately the two *local* transformations. We use the following instructions:

```
#map nom_de (map elements (resoud_compose [e1;e2] [e2;e2] [e1;e1]));;
- : string list = ["e2"; "e1"]
```

Where `map` is a function that distributes the next function `nom_de` over all the elements of the list. We first apply `elements` to all the simplexes of the list and then apply `nom_de` to have directly their names. The solution proposed by ESQIMO is also made of two elements.

Let's call the first element of A, A_1, and the second, A_2. Likewise, we have B_1 and B_2. We suppose that ESQIMO pairs A_1 with B_1 and A_2 with B_2. Then it gets two transformations, say T_1 and T_2, where $T_1(A_1) = B_1$ and $T_2(A_2) = B_2$. When applied to C, also composed of two elements C_1 and C_2, we get $T_1(C_1) = D_1$ and $T_2(C_2) = D_2$. We see that in this example $C_1 = C_2$ but that $D_1 <> D_2$, which means that $T_1 <> T_2$. Indeed, T_1 transforms the element `e1` into `e2` and T_2 transforms `e2` into himself, that is T_2 is the *identity*. Thus, we naturally get $T_1(C_1) = T_1(\texttt{e1}) = \texttt{e2}$, and since T_2 is the identity, when applied to `e1`, it leaves `e1` unchanged. This is illustrated by figure 1.5.

Figure 1.5: Two transformations are considered separately because the elements are paired together before the resolution in the composed case.

There are different pairing heuristics that are presented in chapter 3. It is because we represented the composed cases by sets of simplexes that we can pair them together before the resolution. Another solution would have been to represent the composed figures by complexes of simplexes and instead of pairing the simplexes we would have directly transformed the complexes. This solution is even more *topological* but is much harder to implement. Indeed, the construction of such a complex should use relative relationships between the elements in a figure, but we have to decide which ones. This is also discussed in chapter 3.

Here is one more example of a resolution in a composed case with an expected solution. We can again see how ESQIMO paired the simplexes together, the first with the first, applied to the first, and the second with the second applied to the second. This is simple because we have exactly the same number of elements in each figure and we get also two elements for D. See figure 1.6 for the illustration of this example:

```
#map nom_de (map elements (resoud_compose [e1;e2] [e7;e5] [e3;e1]));;
- : string list = ["e9"; "e4"]
```

Figure 1.6: Two transformations are considered separately because the elements are paired together before the resolution in the composed case.

If the number of elements is different between $A$and B, another strategy of pairing has to be used in order to correctly create a third element for D. The following example is illustrated on figure 1.7:

```
#map nom_de (map elements (resoud_compose [e1;e5] [e4;e2;e8] [e3;e4]));;
- : string list = ["e6"; "e1"; "e7"]
```

Figure 1.7: Two transformations are considered separately because the elements are paired together before the resolution in the composed case.

1.3 Creating Other Universes for ESQIMO

A universe for ESQIMO is a set of elements E, a set of properties P, and the complex defining their relationships[4]. The construction of such a complex is presented in chapter 2. For now, we will just see how to technically add other universes.

1.3.1 Definition

You can choose to type these definition directly on the `Camllight` interface or in a separate file. If you create a separate file, you will have to change your `Makefile` so that it compiles and links your new file. This procedure will be explained in the next section.

To create a new universe you must choose the nature of this universe. It can be another geometrical world or a linguistic one in the sense that you use words, or in the sense that you use letters, it can be numerical, etc.

When the nature is set, you have to select a small number of properties adapted to this world and define the set P. I say a small number because you certainly won't want to define a huge complex for the properties of your universe. If you want a numerical world, you can choose for example the classical properties of multiple of 3, non-multiple of 3, odd, even, etc. These properties will have to be numbered for example from 1 to 4 in order to define the simplexes of each element you will allow in your universe. For the geometrical universe, as we said before, the properties were numbered from 1 to 7. Here we have:

$$P = \{mult\_3\, non\_mult\_3, odd, even\} = \{p_1, p_2, p_3, p_4\}$$

Then, you must define the simplexes, that is the figures that ESQIMO will be allowed to use to solve the analogies. In the geometrical universe, the simplexes allowed were `e1` to `e12`, defined by:

```
let e1 = creer_simplexe[1;5;6]
...
and e12 = creer_simplexe[3;4;7];;
```

[4] We recall that this complex corresponds for example to `omega` in the `univers-geo` universe

By analogy with these definitions, we define our simplexes with the function `creer_simplexe`, we can, for example, define the three first integers:

```
let one = creer_simplexe [2;3]
and two = creer_simplexe [2;4]
and three = creer_simplexe [1;3];;
```

If you try to represent the ten first integers you will see that some of them have the same representation and will be treated as equivalent with only these four properties[5]. After that, you will have to define the complex of the allowed simplexes by analogy with the definition of omega in the `univers-geo.ml` file:

```
let omega = creer_complexe[e1;e2;e3;e4;e5;e6;e7;e8;e9;e10;e11;e12];;
```

we write for example:

```
let numbers = creer_complexe[one;two;three];;
```

You can try a universe that is not complete, I mean, that does not have one element to represent each combination of the properties. You can try to play with the different heuristics to see what happens by reading chapters 3 and 4.

1.3.2 Changes in the Makefile

All the preceding definitions can be included in a new file named `univers-num.ml` for example. On the top of this file you must include the libraries, so that the `univers-num.ml` file looks like:

```
(* ********************************************************** *)
(* ********************************************************** *)
(* univers-num.ml :    numerical universe for ESQIMO          *)
(* your name                                                  *)
(* Creation:                                                  *)
(* Last modified :                                            *)
(* ********************************************************** *)
(*  ESQIMO                                                    *)
(* ********************************************************** *)
#open "complex-lib";;
#open "set";;
#open "esq-kernel";;
(* ********************************************************** *)
(*                  Numerical universe                        *)
(* ********************************************************** *)
let one = creer_simplexe [2;3]
and two = creer_simplexe [2;4]
and three = creer_simplexe [1;3];;
let numbers = creer_complexe[one;two;three];;
(* ********************************************************** *)
```

[5] This numerical example is also evoked in chapter 2, to explain how to represent the ten first integers as simplexes using the CAT framework.

Before the first use, you will have to compile it separately to generate the `.mli` file. Moreover, this output will have to be copied into a file that you will call `univers-num.mli`, type for example:

```
camlc -i univers-num.ml > univers-num.mli
```

The output is in our example:

```
value one : int t;;
value two : int t;;
value three : int t;;
value numbers : int t t;;
```

Now that you have one `.ml` and one `.mli` file, you can compile them, type:

```
camlc univers-num.ml
camlc univers-num.mli
```

If no syntax error is reported, you can change the `Makefile` so that it compiles and links your new files too. At the line that says `New Universe`, you will find:

```
# New Universe

SRC_INTERFACES=\
complex-lib.mli\
esq-kernel.mli
univers-geo.mli

SRC_ML=\
complex-lib.ml\
esq-kernel.ml\
univers-geo.ml
```

That you will only have to replace by:

```
# New Universe

SRC_INTERFACES=\
complex-lib.mli\
esq-kernel.mli
univers-geo.mli
univers-num.mli

SRC_ML=\
complex-lib.ml\
esq-kernel.ml\
univers-geo.ml
univers-num.ml
```

Before re-compiling again your system. When a new `Camllight` interface is launched, don't forget to `#open` your new file too!

Chapter 2

The Concepts of ESQIMO

2.1 Presentation

As we said in the introduction, ESQIMO can be considered at different levels. We already presented it as a simple IQ-tests solver in the first chapter. In this chapter, it is viewed as an application of the CAT framework studied in the UTopoIa project [GV98].

Topology studies objects and properties that are invariant under continuous deformations. Combinatorial topology focuses on the study of finite sets of objects satisfying some spatial relations, and algebraic topology develops the application of algebraic tools to topological problems. The combinatorial algebraic topological (**CAT**) approach is thus attractive for constructivist models and applications.

We will use the CAT framework in ESQIMO at two levels:

- knowledge representation ;
- reasoning.

Of course these two domains are widely interdependent and it is natural to consider them together and model them into the same framework.

We will discuss here the properties of a knowledge representational system based on simplicial complexes. This system is an extension of Q-Analysis introduced by Atkin [Atk74] to represent binary relations. Then we will talk about reasoning into this system with topological operations. Some topological tools are presented that we already use or would like to use. In the particular case of analogy solving, we use only polygonal chains. Finally, we discuss the notion of similarity of figures and configurations in a topological model.

2.2 Knowledge Representation with Simplicial Complexes

In Cognitive Sciences, there are many structures used in Knowledge Representation for computational models. Among the most widely spread are graphs like semantic networks and more precisely conceptual graphs.

2.2.1 Diagrammatic Representations

Diagrammatic reasoning is a field of research investigating the use of spatial relations for knowledge processing. This include knowledge representations, retrieval processing, inference making, etc. Several issues are addressed in these active fields (see [GNC95] for an excellent introduction).

Diagrammatic reasoning is the use of spatial relationships (neighborhood, border, dimension, path, hole, ...) to represent and structure knowledge. Objects and relationships implied by diagrammatic representation have a pre-metric, discrete and finite nature even if some *continua* are involved.

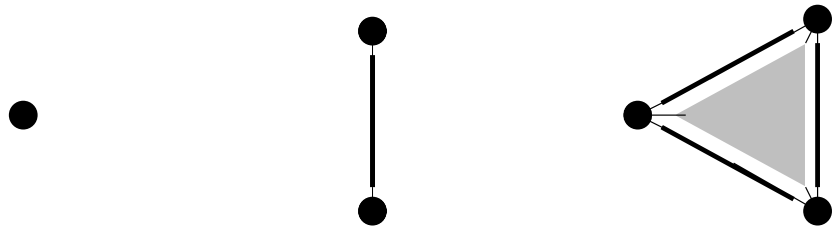
Current logical approaches to diagrams suffer from a "loss of graphicality" [LP97]. The intuitive representation and the direct expressiveness that makes diagrams so appealing for knowledge representation and processing must be preserved. So, we advocate the use of combinatorial algebraic topology for such a linguistic support.

2.2.2 Simplicial Complexes

Simplicial complexes are not unknown in Computer Science, they are used for the construction of 3D objects, for the representation of spatio-temporal constraints. Simplicial complexes are topological abstract structures that generalize the notion of graph. Indeed, all complexes of dimension less than 2 are graphs. The following definition are standard in algebraic topology.

Definition 1 *An abstract simplicial complex is a couple* (V, K) *where* V *is a set of elements called vertices of the complex and* K *is a set of finite parts of* V *such that if* $s \in K$*, then all the parts* $s' \subseteq s$ *belongs also to* K*. The elements of* K *are called abstract simplexes. The dimension of a simplex* s *is equal to* $Card(s) - 1$*. The dimension of a complex is the dimension of its biggest simplex [Hen94, HY88].*

Definition 2 *The couple* $G = (S, A)$*, is an* oriented graph*, where* S *is the set of its vertices, and* $A \subseteq S \times S$ *the set of its edges.*



(a) 0-simplex (b) 1-simplex (c) 2-simplex

Figure 2.1: Geometrical representation of p-simplexes for p varying from 0 to 2.

A p-simplex s is noted: $s = \langle v^0 v^1 ... v^p \rangle$, where $v^i \in V$, the figure 2.1 shows the geometrical representation of 0, 1 and 2-simplexes. A complex is a set of sets closed for the inclusion and the intersection. Thus, simplicial complexes are particularly

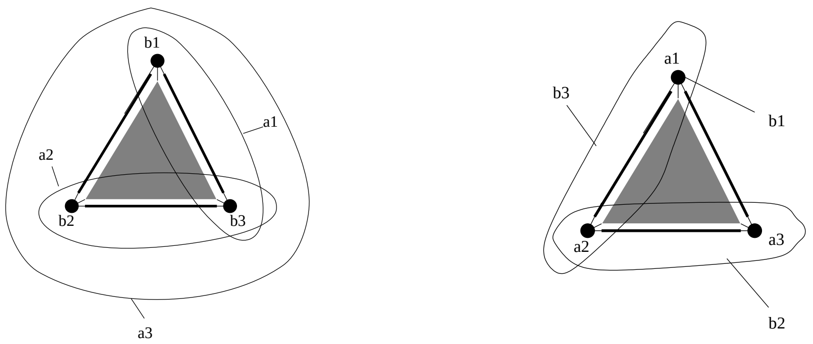
λ	a_1	a_2	a_3
b_1	1	0	0
b_2	0	1	1
b_3	1	1	0

Table 2.1: Incidence matrix associated with λ.

attractive to generalize semantic networks by keeping the possibility to express hierarchies like in a relational graph (a hierarchical structure is highly recommended and trees are often not sufficient for that [Hir97]).

2.2.3 Representation of Binary Relations: Q-Analysis

Atkin already proposed to represent a binary relation λ between two sets with a simplicial complex: it is the **Q-Analysis** [Atk77, Atk81, Joh91a]. Q-Analysis have been used to model traffics [Joh91b], interactions between agents [Dor86, PLC91, Cas94, chap. 8], position analysis at chess [Atk76] and social relations [Atk77, Gou80, Cas94]. Let Λ be the incidence matrix of a binary relation $\lambda \subset A \times B$. Let $a \in A$, and the set B_a of $b_i \in B$ such that $(a, b_i) \in \lambda$. The set B_a can be directly read from Λ, as the a-column (see table 2.1). We represent the elements b_i of B_a as vertices and a as a simplex build on these vertices. The dimension of the simplex S_a representing a depends on the number of elements in B_a. The whole matrix Λ can then be represented as a simplicial complex containing all the simplexes representing each element $a_i \in A$, we note it $K_A(B, \lambda)$ (see figure 2.2.3).



(a) Simplicial representation of λ taking b_i as vertices and a_i as simplexes

(b) Dual simplicial representation of λ taking a_i as vertices and b_i as simplexes

Figure 2.2: Simplicial representation of a binary relation λ. On the left hand, we have $\lambda(a_1) = \{b_1, b_2\}$. So we represent a_1 as a 1-simplex, b_1 and b_2 being its two vertices. On the right hand, we have $\lambda(b_3) = \{a_1, a_2\}$, so b_3 is represented as a 1-simplex where a_1 and a_2 are the vertices.

Likewise, we can represent Λ^{-1} with the dual simplicial complex $K_B(A, \lambda^{-1})$. In this case, the elements a_i are taken as vertices and the elements b_i are represented as simplexes (see figure 2.2.3). We say that $K_A(B, \lambda)$ and $K_B(A, \lambda^{-1})$ are conjugates, they contain the same information but present it in a different and complementary way. On the complex $K_A(B, \lambda)$ we can study the connectivities of the elements of A, on the complex $K_B(A, \lambda^{-1})$, we study the elements of B.

Now, consider two simplexes σ_1 and σ_2, we can study their possible connectivity.

Definition 3 *Two simplexes σ_1 and σ_2 are connected by a chain if there exists a finite sequence of simplexes $\sigma_{a_1}, \sigma_{a_2}, ..., \sigma_{a_n}$ such that:*

$$\begin{array}{ll} (1) & \sigma_{a_1} \leq \sigma_1 \\ (2) & \sigma_{a_n} \leq \sigma_2 \\ (3) & \sigma_{a_i} \quad et \quad \sigma_{a_{i+1}} \text{ partagent une face.} \end{array}$$

We say it is a q-connectivity chain, where q is the smallest a_1

We illustrated this notion by the figure 2.3 where $\langle x_1, x_2, x_3, x_4 \rangle$ and $\langle x_2, x_4, x_5 \rangle$ are 1-connected by the 1-simplex $\langle x_2, x_4 \rangle$.

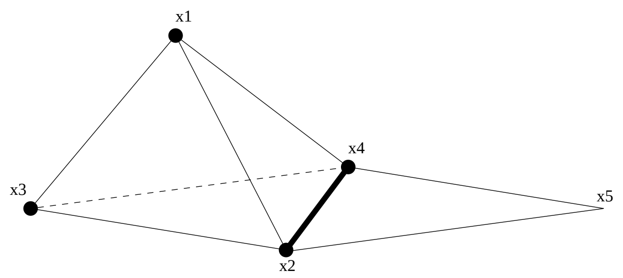


Figure 2.3: A 3-simplex and a 2-simplex 1-connected

For each simplex of a given complex, there are particular values of q-connectivities:

- q_{min}: smallest connectivity of this simplex to any other simplex of the complex;
- q_{max}: biggest connectivity, less or equal to the dimension of the simplex itself.

Finally, we can define the concept of *eccentricity*:

Definition 4 *The eccentricity of a simplex σ_p is given by $\epsilon(\sigma_p) = \frac{q_{max}(\sigma_p) - q_{min}(\sigma_p)}{q_{min}(\sigma_p)+1}$. This measure is infinite when a simplex is totally disconnected from a complex.*

The fact of identifying the q-connected parts of a complex K, with $q \in [0, N]$ and where $N = dim(K)$ gives a partitioning of K.

Let γ_q be an equivalence relation over K for a given q such that: $(\sigma_p, \sigma_q) \in \gamma_q$ iff σ_p is q-connected to σ_q. The equivalence classes under γ_q are member of the quotient group $\frac{K}{\gamma_q}$. The cardinality of $\frac{K}{\gamma_q}$, noted Q_q is equal to the number of q-connected components of K. The Q-Analysis of K consists in finding the numbers $Q_0, Q_1, Q_2, ..., Q_N$ with $N = dim(K)$.

2.2.4 Extended Q-Analysis

We extend the Q-Analysis to allow the representation of sets of predicates as a simplicial complex too. The idea, which is very simple, is to take a set of predicates $P = \{p_1, p_2, ..., p_n\}$ and represent the binary relation $\lambda \subset A \times P$ such that $(a_i, p_j) \in \lambda$ if $p_j(a_i)$ holds.

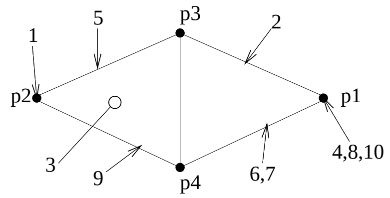


Figure 2.4: Dual complex associated with $\lambda \subset \{1,2,3,4,5,6,7,8,9,10\} \times \{p_1, p_2, p_3, p_4\}$ where we can see that the integers 4, 8 and 10 are identical with respect to these criteria.

	p_1	p_2	p_3	p_4
1	0	1	0	0
2	1	0	1	0
3	0	1	1	1
4	1	0	0	0
5	0	1	1	0
6	1	0	0	1
7	0	1	1	0
8	1	0	0	0
9	0	1	0	1
10	1	0	0	0

Table 2.2: Incidence matrix associated with λ in the numbers example.

Take for example the set of integers $A = \{1,2,3,4,5,6,7,8,9,10\}$ and the set of predicates $P = \{p_1, p_2, p_3, p_4\} = \{\text{parity}, \text{oddity}, \text{primality}, \text{multiple of 3}\}$. The incidence matrix of λ is then obviously the one given on table 2.2. We can represent the dual complex of λ, each element $a_i \in A$ being a simplex build with vertices $p_i \in P$. This dual representation enlighten the fact that elements $4, 8, 10$ have exactly the same representation when taking these few predicates. A representation based upon simplicial complexes associates the same simplex to elements of A that cannot be distinguished. In other words, two elements will be separated only if there is at least one predicate that allows the differentiation. The same situation occurs with the dual complex.

Two simplexes that have a smaller k-simplex in common are said to share a k-face. In terms of representation, it means that *they have k features in common.* As Freska emphasized it, we call here for the use of discriminating features rather than for precise characterization in terms of universally applicable reference system [Fre97].

We can say that the identity of an element is represented by the features he shares with others and also by the ones that are specific to it [HE97]. In our example of the ten first integers, we can analyse the complex of figure 2.4 to see that 2 and $6, 7$ share one $0-$face that is represented by the elements $4, 8, 10$. We see also that 9 and 5 share also a $0-$face but not the same one, that 3 contains 5 since it is one of its faces, etc... But one more interesting analysis would be to consider the dual complex of properties represented on the numbers, we get:

$$p_1 = \{2, 4, 6, 8, 10\}$$

$$p_2 = \{1, 3, 5, 7, 9\}$$

$$p_3 = \{2, 3, 5, 7\}$$

$$p_4 = \{3, 6, 9\}$$

and we can see that p_2 and p_3 share a 2−face composed on the numbers 3, 5, 7 and that no property contains any other one. This can be usefull to choose a good base of properties,...

2.3 Reasoning on a Topological Representation

If the knowledge representation level is modeled as a topological structure such as simplicial complexes, we intuitively want to model the operations made on that representation as topological operations. Indeed, the knowledge is represented as a structure and the reasoning is the way we use, manipulate, transform or compute *on* and *with* it.

2.3.1 Topological Tools for Reasoning

Among the topological operations that we can perform on simplicial complexes are:

- Q-Analysis;
- polygonal chains;
- homotopy;
- homology.

Q-Analysis has already been presented, the homotopy and homology tools will be used for further applications of the CAT framework in the UtopoIa project. But for now, we only present *polygonal chains* since it is the only tool usedin ESQIMO.

Definition 5 *Let $\alpha = (\sigma_0, \sigma_1, ..., \sigma_n)$ be a sequence of simplexes belonging to a complex K. It is called a polygonal chain of origin σ_0 and end σ_n if for all couple (σ_i, σ_{i+1}), $\sigma_i \cap \sigma_{i+1} \neq \emptyset$. The dimension of α is the smallest dimension of $\sigma_i \cap \sigma_{i+1}$.*

Any p-simplex is p-connected to himself with a 0-chain.

2.3.2 Topological Model for Similarity and Analogy

Analogy solving is often regarded as a fundamental cognitive process involved in several tasks such as problem solving, explanation, learning and decision making.

The study of analogies is thus a very wide and active field of investigations in which the main different streams are [Hal89]:

- importance of analogy solving in human behaviors;
- possible structures to support the representation of analogies;
- role of analogies in learning processes;
- computational models for the detection, the formation and the solving of analogies.

The work we present here belongs to the latter approach.

Going back from Aristotle we can now find many definitions in which analogy entails certain inferences based on the similarity of qualities, of relations or of structures [Hof95]. More generally, an analogy is a *similarity* between two couples of items [GI92]. The exact nature of the similarity involved in analogy processes is still widely discussed, but people generally split the process of analogy solving into four steps:

- selection of a possible source analog;
- mapping between the source and the target analogs;
- analogical transfer;
- subsequent learning.

As outlined by Holyoak and Thagard, the core of analogical thinking lies in the process of mapping [HT89]; this is precisely the step on which we focus here.

Gentner has emphasized the importance of structural constraints in determining the correspondences between two analogs. We know that semantic and pragmatic constraints are also to be taken into account, for the sake of simplicity, we start studying only the structural constraints.

In a computational approach, the mapping is performed on a representational structure that has to be adapted to satisfy the structural constraints given by Gentner [HT89]. In the CAT framework, we will use the concept of*topological equivalence* to model a topological similarity. Two objects are topologically equivalents if we can obtain the second by simply applying a continuous deformation to the first.

Definition 6 *A continuous deformation is a deformation where the neighborhood of any point is preserved. The structure of the space is preserved.*

This comparison only concern topological properties such as form, border, holes, dimension, and not metrical properties. Thus, our measure of similarity can not use any metric. A transformation that preserves the structure of simplicial complexes is a simplicial application.

Chapter 3

Analogy solving with ESQIMO

In this chapter, we consider ESQIMO as a model for analogy solving. we recall the four basic steps involved in an analogy solving and we present ESQIMO's algorithm based on the CAT framework.

3.1 Modeling an Analogy Solving Process

The task is to answer a typical IQ-test by giving an element called D such that it completes a four-term analogy with three other given elements A, B and C: "find D such that it is to C what B is to A". This kind of analogy solving has already been studied by Evans [Eva68], but in our work the solution has to be build from scratch, it is *non supervised.* This four-term analogy solving is usually decomposed into four steps [Eva68]:

- *Step 1*: find the possible transformations T_{AB} between A and B.
- *Step 2*: find the possible transformations T_{AC} between A and C.
- *Step 3*: apply T_{AB} to C only on a domain determined with T_{AC}.
- *Step 4*: verify the symmetry by applying T_{AC} to B.

To solve a four-term analogy, we propose to represent each figure by a simplex and the relation between the first two figures by a polygonal chain into the problem space represented as a complex[1]. Building the fourth figure from the third will thus be deforming this third figure according to the precedent path.

3.2 The Geometrical World of ESQIMO

Usually, IQ-tests are given in terms of geometrical elements so that they can express many different properties at the same level and still stay simples. We chose a geometrical universe similar to the one investigated in [WS90] of twelve basic elements, as shown on figure 3.1:

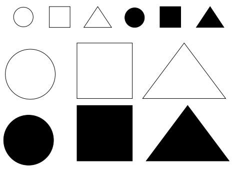
$$E = \{e1, ..., e_{12}\}$$

[1] The definition of these concepts are given in chapter 2

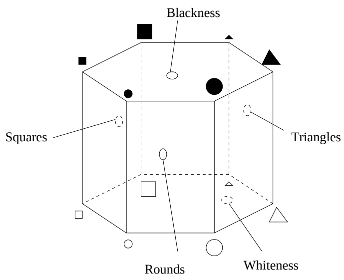
These elements are all the possible combinations of the seven properties (or predicates):

$$P = \{p_1, ..., p_7\} = \{\text{round, square, triangle, white, dark, big, small}\}$$

Each element is defined with three properties, one of shape, one of size and one of color.



(a) Elements of the universe Ω of ESQIMO, respectively called e_1 to e_{12} starting from the top left element



(b) A 2D view of the dual complex $K'(Omega)$, the elements of E are the vertices and the properties $p_i \in P$ are simplexes of $K'(\Omega)$. Notice that the 6-simplex representing the property of blackness is normally 5-dimensional

Figure 3.1: Elements manipulated by ESQIMO and their representation as a simplicial complex.

These two sets are the only knowledge used by ESQIMO to solve the tests. We can represent this knowledge with a simplicial complex $K(\Omega)$ or its conjugate $K'(\Omega)$ (see figure 3.1) by representing the binary relation $\lambda \subset A \times P$ such that $(a_i, p_j) \in \lambda$ if $p_j(a_i)$ holds. The complex $K'(\Omega)$ is then the space of the problem in which ESQIMO solves analogies by deforming simplexes into others.

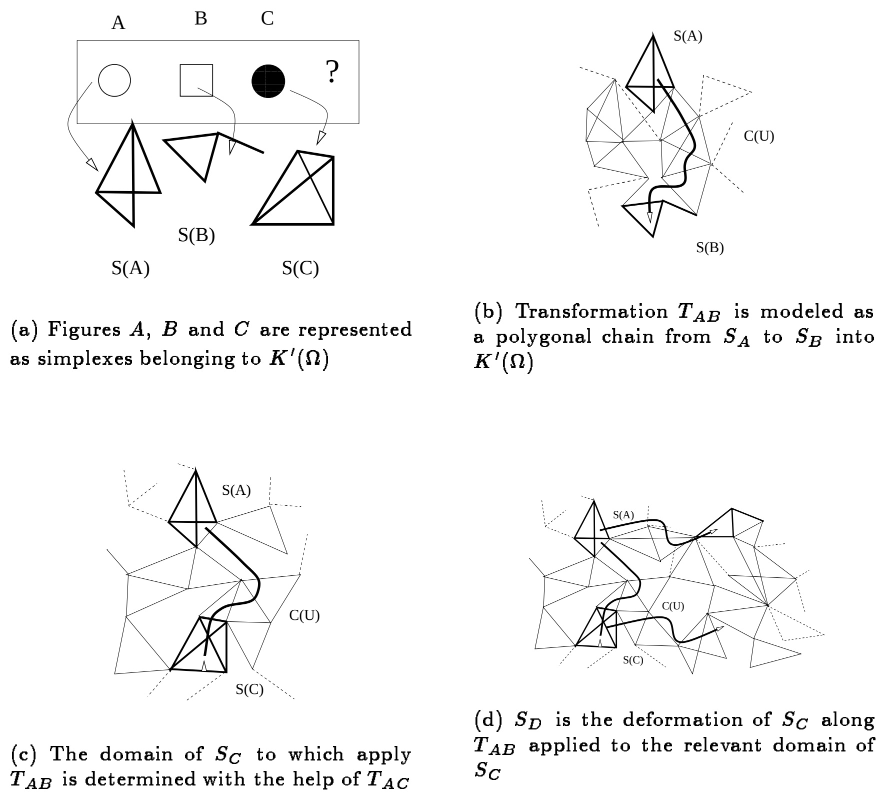
3.3 Algorithm

3.3.1 Representing the Problem

When a problem is presented, each figure A, B and C is composed of one or more elements $e_i \in E$. Each element e_i can be represented as a simplex of $K(\Omega)$ by simply considering the binary relation $\lambda \subset E \times P$ such that $p_j(e_i)$ holds. Then an element e_i will always be a 2-simplex, since it is always defines with three properties[2].

We call a *simple figure*, a figure composed of only one element, it will be represented as a simplex. We call *a composed figure*, a figure that contains more than one element, it will be represented with a set of simplexes. The problem is now to find a relation between the (set of) simplex(es) representing A and the (set of) simplex(es) representing B and apply it to the (set of) simplex(es) representing C. Note that the representations of A, B and C are all included into the complex $K'(\Omega)$.

[2]Except for monsters presented in the examples.



(a) Figures A, B and C are represented as simplexes belonging to $K'(\Omega)$

(b) Transformation T_{AB} is modeled as a polygonal chain from S_A to S_B into $K'(\Omega)$

(c) The domain of S_C to which apply T_{AB} is determined with the help of T_{AC}

(d) S_D is the deformation of S_C along T_{AB} applied to the relevant domain of S_C

Figure 3.2: Four steps of ESQIMO's algorithm to solve IQ tests in the case of simple figures A, B and C respectively represented as the simplexes S_A, S_B and S_C.

3.3.2 Case of Simple Figures

In *step 1*, we model the transformation T_{AB} as a polygonal chain connecting S_A to S_B in $K(\Omega)$. An elementary step linking S_i to S_{i+1} in a chain is interpreted as an elementary transformation $T_{S_i,S_{i+1}}$. A polygonal chain from S_A to S_B is then a transformation of A into B given by: $T_{S_l,S_B} \circ \ldots \circ T_{S_A,S_1}$.

If there exists several chains connecting S_A to S_B, then we say that there are several possible relations between A and B. We can choose to minimize the number of possible solutions, by giving a higher priority to polygonal chains that are short and of higher dimension, that corresponds to choose a transformation that requires less steps and that preserves more properties. This is comparable with selecting a *best* solution according to some measure of satisfaction or of naturality of the transformation like in [Eva68].

In *step 2*, to apply T_{AB} to S_C we have to extend the domain of T_{AB}, and so extend T_{AB} to T'_{AB} such that $T'_{AB}(S_C) = S_D$ and $T'_{AB}(S_A) = S_B$ (simplicial application [Hen94, HY88]). There are different possible strategies to determine the domain of $S(C)$ on which we can apply T_{AB}:

- Let p be the set of properties added to S_B with respect to S_A, let m be the set of properties retired from S_B with respect to S_A. The first heuristic simply adds the properties of p without caring if they were already present[3] and subtracts any element of m that is present in S_C.

- In the second heuristic, only the elements of m a subtracted if present in S_C.
- The third strategy transforms anything into S_B, we take here into consideration the possible *constant* transformation that have been observed in empirical data (see chapter ?? for empirical results).
- The fourth heuristic changes S_C only if it the same figure as S_A. Here the transformation obtained by studying the pair $A - B$ is not extended, the definition domain is strictly restricted to S_A.
- We consider here more *a priori* knowledge about the number of arguments to have a stable solution, not a monster.

These five strategies have been implemented and the first is applied by default. You can change the default strategy or implement others, see chapter 4. Then, *step 3* consists in building the corresponding simplex for D. No *step 4* have been implemented to verify the symmetry until now.

3.3.3 Case of Composed Figures

For composed figures, before *step 1*, we pair the simplexes of $\{S_A\}$ with those of $\{S_B\}$ and look for transformations between the simplexes of each pair. The transformation T_{AB} is then the parallel application of the transformation found for each pair.

There are many possible pairings leading to different or to the same solution. The only constraint we need is that all the vertices and faces of S_B are paired with vertices from S_A, all the possibilities are illustrated on figures 3.3 3.4 3.5:

- If $Card(S_A) = Card(S_B)$ (see figure 3.3(a)) we pair one simplex of S_A with one vertex of S_B, any permutation is acceptable since the order of the simplexes in S_A and S_B brings no information.
- If $Card(S_A) \sup Card(S_B)$ (see figure 3.4(b)) some simplexes of S_A will have to be paired with simplexes of S_B already paired. In this case, we pair all the extra simplexes of S_A with the last paired simplex of S_B. A more intelligent strategy would be to recognize the elements, pair first those that are identical between A and B and pair the others on the left simplexes.
- If $Card(S_B) \sup Card(S_A)$ (see figure ??), to crate more elements during the transformation, we pair the extra simplexes of S_A with the last simplex of S_A. The same remark as in the preceding case holds.

The transformations can be of several types: destruction, creation, metamorphosis, division, junction (like in the changes introduced by Hornsby [HE97]). A pairing of a vertex of $S(A)$ with $\emptyset$ means its destruction, the pairing of a vertex of $S(A)$ with one vertex of $S(B)$ means its transformation and the pairing of a vertex from $S(A)$ with several vertices of $S(B)$ means its duplication with transformation.

[3] It is impossible to have twice the same property, adding a property already there is just like doing nothing

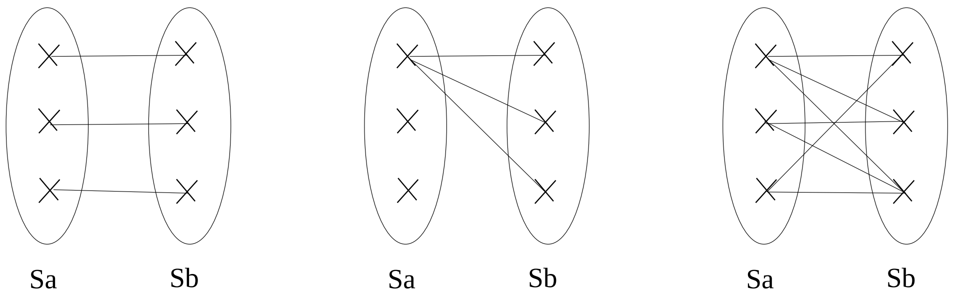


Figure 3.3: Allowed pairings between two sets of simplexes when $Card(S_A) = Card(S_B)$

Figure 3.4: Allowed pairings when $Card(S_A) \sup Card(S_B)$

Figure 3.5: Allowed pairings when $Card(S_B) \sup Card(S_A)$

3.4 Discussion

Many choices made in ESQIMO's algorithm can be discussed. In fact, they can be seen as additional strategies parameterizing the ESQIMO kernel.

3.4.1 The Description of the Properties

We describe the properties of each figure in terms of predicates can be a problem for properties such as position. Indeed, we use only *true/false* properties and not multi-valuated properties. We would like for example to express things like *form* with values going from *blue* to *red*. We also want to express the negation of a predicate. Since now, we only introduced a predicate p and a predicate $!p$ (not p) like *black* and *white*, but there is no information that links them together.

If we want to constraint the elements to have only one color, one shape, one size, we should consider the complex of colors, the complex of shapes and the complex of sizes and require any figure to have one slot in each complex. This point of view is not simplistic because it introduces too much granularity in the representation while no new concept is considered.

Another problem occurs when we want to express relative relationships, we could give each possible position a predicate that could be true or false but this is a heavy solution. In fact, we should represent composed figures with complexes and not with sets of simplexes. First, this would be even a more topological representation, and second, we could use the relative properties to build these complexes. This solution in under study and will probably be implemented in the next version.

3.4.2 Association of a Transformation with a Polygonal Chain

The way we associate a transformation to a given polygonal chain is not unique. In particular, our transformations could be called $0-$degree since they preserve the minimum of topological properties along a chain. The next step consists in pairing higher-order structures between the sets of simplexes.

The correspondence between finding a polygonal chain that connects two simplexes and finding a relations between two figures has to be studied in more details. Indeed, when the figures are represented as simplicial complexes, there is a translation of the information from one form to another. The information concerning the properties of the figures is now represented as edges connecting vertices, where the vertices are the properties. Finding a chain that connects two simplexes is then finding which edges to move on to go from one figure to the other. But there is still the operation of transforming back this path into the vocabulary of the properties.

More precisely an edge connecting two vertices can be interpreted in different ways. When moving from one vertice to the other along this edge, we can say that the first property is lost an the second is won. We can say that the property of whiteness for example is lost while the property of blackness is won. But if we consider more implicit knowledge, we can say that the property of *color* is changed from the *value* white to the value black. considering the properties of color, shape, etc... would lead to a different knowledge representation.

However, here is one of the big dilemma of ESQIMO, choosing the simplicity and allowing some weird elements to appear, or choose exactitude and allow some complicated structures to appear and hide the simple concepts that we first wanted to illustrate.

3.4.3 The Determination of the Domain of C

The way we determine the domain of S_C on which to apply T_{AB} can also lead to different strategies depending on whether we consider only the intersection between S_A and S_C or the whole S_C:

(a) (b) (c) (d)

Figure 3.6: In this example, no constraint has been imposed to the number of values for the shape

- The different heuristics to apply a transformation to C consider the constant transformation, where the domain is everything, the most restrained transformation where the domain is strictly A and intermediate domains.
- In the composed figures case, another pairing is necessary between the elements of S_C and the many transformations we found between S_A and S_B after the first pairing.

3.4.4 Measure of a *Best* Solution

The measure of satisfaction to select a *best* solution is here to take the shorter and wider polygonal chain between the two complexes. Other measures of satisfaction can be tested.

Note that our formalization of IQ-test problems does not depend on their geometrical nature. Indeed, only the representational level is based on topology while the objects manipulated by the system could have been non geometrical. We could, for example try ESQIMO on verbal IQ-tests more like in the Copycat system [Hof84].

3.5 Comparison with other models

Corresponding to different theories of analogy, different computational models have been developed to model analogy solving. Some of them require a very sophisticated representation of the knowledges, others use very simplified ones. Some give many solutions, others select only a best one; some compute individual analogies, others compute domain to domain analogies [Hof95].

A number of models have involved connectionist and hybrid architectures. Among them, ARCS (Analog Retrieval by Constraint Satisfaction) has been developed by Thagard and Holyoak [THNG90]. Their main idea was to simultaneously satisfy the structural, semantic and pragmatic constraints and then select the best solution by putting them into a competition. We can hardly compare this system to ours since we only tried to satisfy the structural constraint. Moreover, ARCS is principally build to evaluate domain-domain mappings and we only studied intra-domain analogies.

ARCS is based upon a neuronal network architecture, which, again cannot be compared with our representational formalism in terms of performance. The purpose of ESQIMO does not lie in its performance but in its cognitive plausibility, its possible use for cognitive purposes as a computational model and also as an exploration of a new representational structure using topology.

SME is another system proposed by Falkenhainer to illustrate Gentner's theory for analogy [FFG89]. Gentner gives two rules to proceed the mapping between the source and the target. These rules only depend of syntactic properties of the knowledge representation and are proceeded with respect to the relations and not the attributes of the objects [Gen83]. In that sense, the SME system can be compared with ESQIMO since we also focused our work on the structural constraint. However, the representational structure used in SME is propositional, even if these propositions are included into little nets. Our simplicial structure is not opposed to theirs but only suggest that the spatial properties in such a net could (and should) also be used.

The ANALOGY system [Eva68, Ha91] is much more closer to ESQIMO in the sense that it deals with almost exactly the same cognitive task. Anyway, this system is also much more developed than our prototype since it uses more properties such as relative position of elements in the figures. However, a bigger difference is that ANALOGY selects a best solution among a choice of five candidates, which makes the problem to be supervised. ANALOGY is therefore a system that finds the best transformation, and so the best justification between five possible transformations. Anyway, the way this best solution is found implies a measure of similarity, notion that we somehow also needed to introduce when we selected the shorter and wider polygonal chain for our transformation of complexes.

As we can see from these three comparisons, our contribution lies principally in the search for a new representational structure to model analogy. This search for a new structure can be justified in terms of naturality of a spatial representation [Arn69, Gar97, GNC95] but also has to be seen as an intermediate, or hybrid structure between a purely symbolic and a purely analogical approach.

Chapter 4

The ESQIMO Program

4.1 Structure of the ESQIMO 2.0 system

The ESQIMO kernel is programmed in the file `esq-kernel.ml`[1] and specified in the `.mli` file. A complete library of manipulation of lists, simplicial complexes and chains (see types below) is programmed in the `complex-lib.ml` file and specified in the `.mli` file. One universe of IQ-test is already programmed, it is a geometrical universe and the filename is `univers-geo.ml`, specified in the `.mli` file.

The library `complex-lib.ml` is reusable for other applications based on simplicial complex (like hierarchy restructuring, concepts extraction in the little red riding hood Tale, hypertexts analyzing, etc in the *UTopoIa* project).

The figure 4.1 shows the dependencies that exist between the files. You must be careful with these dependencies when adding new files to your system in particular when defining new universes (see chapter 1).

[1] All the names of the functions are in French.

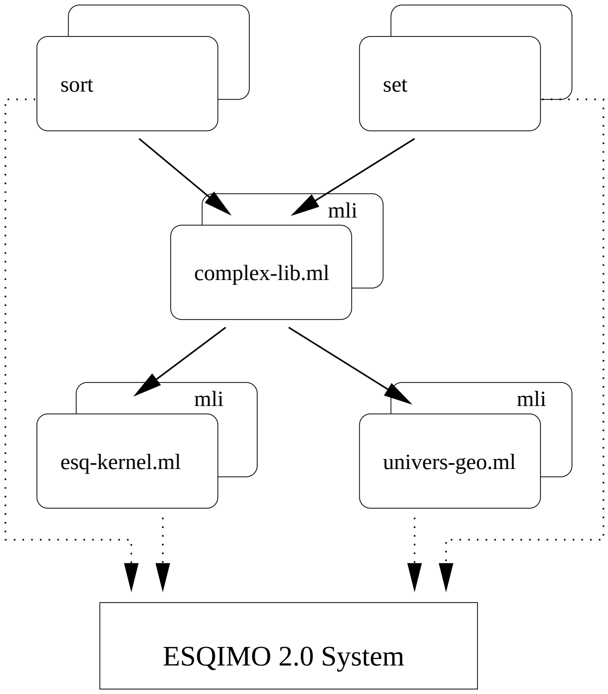


Figure 4.1: Dependencies between the three main files of ESQIMO's system

4.2 The Library `complex-lib`

The interface to the library `complex-lib.ml` is completely specified below to facilitate its reuse for other applications. We present first the types defined to represent the structures manipulated by ESQIMO.

4.2.1 Types and Data Structure

We will use the existing types of `list` and `set` of `CaML`. The lists are written between brackets, `[x;y;...]`. We will use the existing libraries for lists and sets but we added some other functions for the manipulation of lists.

ESQIMO uses also a representational level based on simplicial complexes, we thus introduce the types corresponding to these structures:

```
type 'elt Splx == 'elt set__t;;
type 'elt Cplx == ('elt Splx) set__t;;
```

Polygonal chains will be manipulated with the type `Chaine` defined as a list of simplexes:

```
type 'elt Chaine = cChaine of ('elt Splx) list;;
```

We also add an exception that characterizes the errors in ESQIMO's algorithm.

```
exception ErrorEsqimo;;
```

4.2.2 Operations on Type `list`

Some extra functions have been implemented for the manipulation of lists, like their concatenation `concatene`, sorting `trie_list`, transformation into a set `list_to_ens` , selection of special elements. The complete list of these functions and the types required are given below:

concatene

Concatenation of two lists and returns a list:

```
value concatene : 'a list -> 'a list -> 'a list;;
```

fin

Gives the last element of a list:

```
value fin : 'a list -> 'a;;
```

select

Selects the list of elements of the list for which the predicate p holds:

```
value select : ('a -> bool) -> 'a list -> 'a list;;
```

list_select

Selects the i^{th} element of the list:

```
value list_select : int -> 'a list -> 'a;;
```

sous_list

Selects the sub-list of the list starting from the beginning until the i^{th} element included:

```
value sous_list : int -> 'a list -> 'a list;;
```

add_list

Adds an element at the beginning of the list:

```
value add_list : 'a -> 'a list -> 'a list;;
```

mem_list

Tests the inclusion in the list:

```
value mem_list : 'a -> 'a list -> bool;;
```

list_to_ens

Transforms a list into a set:

```
value list_to_ens : 'a list -> 'a t;;
```

trie_list

Sorts the list by number and suppresses duplicates:

```
value trie_list : 'a list -> 'a list;;
```

4.2.3 Operations on Type Splx

A simplex can be created, sorted or displayed. One can add a vertice or take it off, test whether a vertice is member of a simplex, ask for the dimension or the list of vertices or the number of vertices of the simplex. The complete list of these functions and the types required are given below:

creer_simplexe

Creates a structure of type `Splx` with a list of vertices:

```
value creer_simplexe : 'a list -> 'a t;;
```

ajoute_simplexe

Adds a new vertice and creates the new simplex:

```
value ajoute_simplexe : 'a t -> 'a list -> 'a t;;
```

enleve_simplexe

Suppresses one vertice of a simplex and creates the new simplex:

```
value enleve_simplexe : 'a t -> 'a -> 'a t;;
```

dim

Gives the dimension of a simplex:

```
value dim : 'a t -> int;;
```

sommets_splx

Gives the list of vertices of a simplex:

```
value sommets_splx : 'a t -> 'a list;;
```

nb_sommets_splx

Gives the number of vertices of a simplex:

```
value nb_sommets_splx : 'a t -> int;;
```

dans_splx

Tests whether a vertice, an edge or a face is contained into a simplex:

```
value dans_splx : 'a t -> 'a t -> bool;;
```

voir_liste_splx

Display of a list of simplexes:

```
value voir_liste_splx : 'a t list -> 'a list list;;
```

voir_appariements

Display of a list of pairings between two sets:

```
value voir_appariements : 'a t list list -> 'a list list list;;
```

4.2.4 Operations on Type Cplx

creer_complexe

Creates a complex from a list of simplexes:

```
value creer_complexe : 'a t list -> 'a t t;;
```

value dans_cplx

Tests whether a simplex belongs to a complex:

```
value dans_cplx : 'a t -> 'a t t -> bool;;
```

elements_cplx

Gives the list of the elements of a complex:

```
value elements_cplx : 'a t t -> 'a list list;;
```

sommets_cplx

Gives the vertices of a complex:

```
value sommets_cplx : 'a t t -> 'a list list;;
```

nb_sommets_cplx

Number of vertices of a complex:

```
value nb_sommets_cplx : 'a t t -> int;;
```

compare_cplx

Returns true if the two complexe are the same:

```
value compare_cplx : 'a t -> 'a t -> bool;;
```

4.2.5 Operations on Type `Chaine`

One can add one element to a `Chaine`, test if an element is part of it, take the minimum of its elements and take its tail.

fin_chaine

Gives last element of a chain, it is a simplex:

```
value fin_chaine : 'a Chaine -> 'a Splx;;
```

add_chaine

Adds a simplex at the end of a polygonal chain:

```
value add_chaine : 'a t -> 'a Chaine -> 'a Chaine;;
```

chaine_length

Gives the number of simplexes in a chain :

```
value chaine_length : 'a Chaine -> int;;
```

dans_chaine

Tests whether a simplex belongs to a chain:

```
value dans_chaine : 'a Chaine ->
      (('b -> 'b t -> bool) -> 'a Splx list -> 'c) -> 'c;;
```

min_chaine

Gives the chain if minimal dimension between two chains:

```
value min_chaine : 'a Chaine -> 'a Chaine -> 'a Chaine;;
```

voir_chaine

Display of a chain of simplexes:

```
value voir_chaine : 'a Chaine -> 'a list list;;
```

4.2.6 Operations for Connectivity

One can ask if there is a connection between two elements in a complex, or ask for the list of neighbors of an element.

connectivite

If a connectivity of dimension more than the first integer given as argument exists, returns the simplex representing this intersection, else returns the empty set:

```
value connectivite : int -> 'a t -> 'a t -> 'a t;;
```

voisins

Neighbors of degree q of a simplex into a complex, the first argument s the complex,: the second q.

```
value voisins : 'a t t -> int -> 'a t -> 'a t t;;
```

4.3 ESQIMO's Kernel

Most of the functions of ESQIMO's kernel use the library `complex-lib.ml`. We present the principal functions corresponding to fundamental steps of the algorithm. You can play with these functions and test the different heuristics evoked in chapter 3.

4.3.1 Determination of a Polygonal Chain between two Simplexes

meilleur_chemin

Determines a *best* chain between two simplexes using two criteria: minimal length (`court_chemin`) and maximal dimensionality (`large_chemin`). Takes a complex C and two simplexes S_1 and S_2 and returns the polygonal chain with type `Chaine`. C is the complex where the polygonal chain has to be found, S_1 is the starting simplex and S_2 the arrival:

```
value meilleur_chemin : 'a t t -> 'a t -> 'a t -> 'a Chaine;;
```

court_chemin

Determines the shortest polygonal chain between two simplexes for a given degree of connectivity. Takes a complex C, an integer q and two simplexes S_1 and S_2 and returns the polygonal chain with type `Chaine`. C is the complex where the polygonal chain has to be found, q is the degree of connectivity, S_1 is the starting simplex and S_2 the arrival:

```
value court_chemin : 'a t t -> int -> 'a t -> 'a t -> 'a Chaine;;
```

large_court_chemin

Determines the polygonal chain of higher dimension among the shortest chains between two simplexes. It starts with a minimal value of the dimension and increases this value until no chain is found between the simplexes.The last dimension successfully tested is the higher dimension. Takes a complex C, an integer q_0 and two simplexes S_1 and S_2 and returns the polygonal chain with type `Chaine`. C is the complex where the polygonal chain has to be found, q_0 the starting dimension to search for a shortest chain, S_1 is the starting simplex and S_2 the arrival:

```
value large_chemin : 'a t t -> int -> 'a t -> 'a t -> int;;
```

chemins

Determines the set of polygonal chains linking two simplexes in a complex. Takes a complex C, an integer q, an integer l, a first simplex S_1 and a second simplex S_2 and returns a set of chains of type `'a Chaine t`. C is the complex where to find the chains, q the dimension and l the length, S_1 the starting simplex and S_2 the arriving simplex:

```
value chemins : 'a t t -> int -> int -> 'a t -> 'a t -> 'a Chaine t;;
```

voir_chemins

This function is used to *see* inside the structure of a set of polygonal chains. It is used to see the result of the function `chemins`:

```
value voir_chemins : 'a Chaine t -> 'a list list list;;
```

4.3.2 Translation of a Polygonal Chain into a Transformation

When a polygonal chain is found between two simplexes, we consider it as a transformation and apply it.

pas_elementaires

Transforms a polygonal chain into a succession of elementary transformations of simplexes. The output is a list of transformations, and each transformation is a list of two elements where the first element is the simplex that is transformed into the second element. You can see the list of elementary transformations with `voir_pas_elem`:

```
value pas_elementaires : 'a Chaine -> 'a Splx list list;;
value voir_pas_elem : 'a t list list -> 'a list list list;;
```

inv_tr, plus_tr, moins_tr

Three modes of interpretation of a transformation for its further application. The first `inv_tr` takes into account only the properties that are left invariant in the transformation, the second `plus_tr` look which property has been added and the last one which has been suppressed:

```
value inv_tr : 'a t -> 'a t -> 'a list;;

value plus_tr : 'a t -> 'a t -> 'a list;;

value moins_tr : 'a t -> 'a t -> 'a list;;
```

4.3.3 Application of a Transformation

We present here the heuristics to apply the previous transformations. The transformations are delivered as a list of couples of simplexes. We have to take into account different domains of the figure S_C. Four heuristics have been implemented to apply the transformation.

applique_hi

All the heuristics take three simplexes and return a list. The three simplexes are S_A S_B and S_C, the output is given as a list to see the properties:

```
value applique_h1 : 'a t -> 'a t -> 'a t -> 'a list;;
value applique_h2 : 'a t -> 'a t -> 'a t -> 'a list;;
value applique_h3 : 'a -> 'b t -> 'c -> 'b list;;
value applique_h4 : 'a t -> 'a t -> 'a t -> 'a list;;
```

The first heuristic adds any property of S_B that is not yet in S_C and takes off any property of S_C that is not in S_B. The second, only takes off the properties not included in S_B. In the third, everything is transformed into S_B, it is the constant transformation. In the last one, we consider the properties of S_C that are in common with those of S_A before changing them.

applique_h

This function sets the heuristic to choose and the heuristic by default:

```
value applique_h : 'a list -> 'a t list -> 'a list;;
```

4.3.4 Pairing of Simplexes in the Composed Case

apparie_eq, apparie_sur_un, apparie_de_un, apparie_sup, apparie_inf

Five functions of pairing have been implemented to cover all the cases mentioned in chapter 3. All these functions take two lists of simplexes corresponding to the set of S_A and the set of S_B and returns a list of couples of simplexes. Each couple is a list of length 2 and represent one pairing between one simplex of S_A and one simplex of S_B:

```
value apparie_eq : 'a list -> 'a list -> 'a list list;;
value apparie_sur_un : 'a -> 'a list -> 'a list list;;
value apparie_sup : 'a list -> 'a list -> 'a list list;;
```

The first function `apparie_eq` deals with the case where the two sets of simplexes have the same number of elements. The second one `apparie_sur_un` deals the case where several elements of the first set of simplexes are paired with only one element of the second set. the function `apparie_de_un` does the reverse, it pairs only one element of the first simplex with many of the second. The function `apparie_sup` deals with the case where the first set has more elements than the second, and `apparie_inf` with the case where the first set has less elements.

apparie

Choice of the heuristic of pairing and heuristic by default:

```
value apparie : 'a list -> 'a list -> 'a list list;;
```

applique_app

Applies a pairing to two sets of simplexes. Takes one complex C, two lists of lists of simplexes and returns a list of simplexes. The complex C is the universe, the two first lists are respectively the pairings between S_A and S_B and the pairings between S_A and S_C. The list returned is a distribution of the first pairings to the seconds. It means that we distribute the elementary transformations over the elements of S_C.

```
value applique_app : 'a t t -> 'a t list list ->
                     'a t list list -> 'a t list;;
```

traite_cas_compose

Deals with the total pairing process when the three figures are composed. Pairs the sets corresponding to A and B, finds a pairing between A and C and applies the pairings. Takes two lists of simplexes and returns a list of lists that is the distribution of the first pairing over the second.

```
value traite_cas_compose : 'a list -> 'a list -> 'a list list;;
```

It uses intermediate functions:

```
value trouve_app_de :  'a -> 'a list list -> 'a;;
value traite_un_app : 'a t t -> 'a t list list ->
                      'a t list -> 'a t;;
```

The first one finds the brother of an element x in a pair among a list of pairings. Takes an element x and a list of pairings and returns an element y which is the brother of x. The second function takes these pairs one by one and applies the function `resoud_simple` to each of them. Takes the complex C that is the universe, one series of pairings and the list of simplexes representing S_C.

4.3.5 Resolution

resoud_compose, resoud_simple

Functions for the complete resolution of an analogy when the figures are already presented as simplexes or sets. `resoud_simple` deals with simple cases where the figures are only represented by one simplex each. Takes a complex C and three simplexes S_A, S_B and S_C and returns a simplex S_D:

```
value resoud_simple : 'a t t -> 'a t -> 'a t -> 'a t -> 'a t;;
```

A schematic view of the function `resoud_simple` is presented on figure 4.2 where we can the in and out types as well as the internal calls for functions.

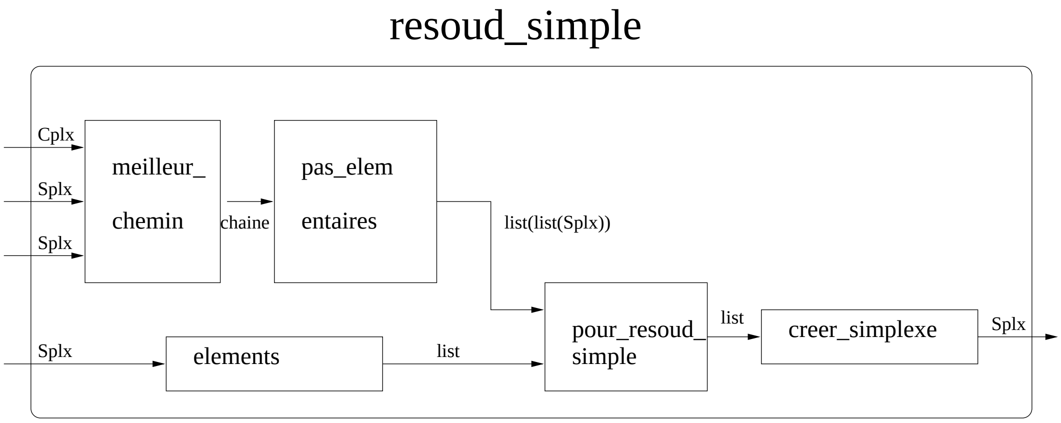


Figure 4.2: Specification of the function `resoud_simple` that solves simple case tests.

The function `resoud_compose` deals with composed cases and calls the pairing functions.

```
value resoud_compose :  'a t t -> 'a t list ->
                        'a t list -> 'a t list -> 'a t list;;
```

A schematic view of the function `resoud_compose` is presented on figure 4.3 where we can the in and out types as well as the internal calls for functions. The call for the function `resoud_simple` is done inside the function `applique_app` for each pair. The pairs are determined by the functions before `applique_app`.

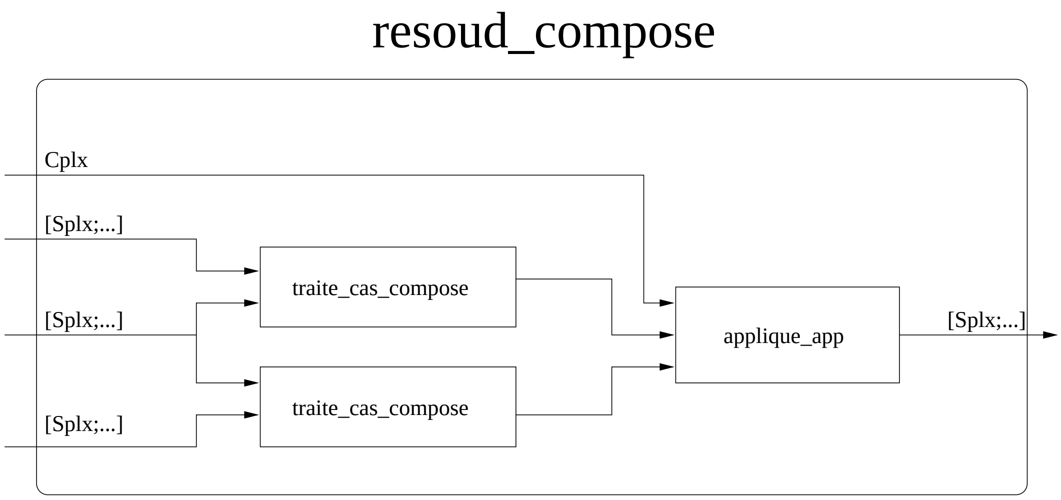


Figure 4.3: Specification of the function `resoud_compose` that solves composed case tests.

Bibliography

[Arn69] Rudolh Arnheim. *Visual Thinking.* University of California Press, 1969. trad. française : "La pensée visuelle", Champs-Flammarion, 1976.

[Atk74] R. H. Atkin. *Mathematical Structure in Human Affairs.* Heinemann, 1974.

[Atk76] Atkin & al. Fred CHAMP positional chess analyst. *International Journal of Man-Machine Studies*, 8:517–529, 1976.

[Atk77] R. H. Atkin. *Combinatorial Connectivities in Social Systems.* Verlag, 1977.

[Atk81] R. H. Atkin. *Multidimensional Man.* Penguin, 1981.

[Cas94] J. L. Casti. *Complexification.* Harper Perennial, 1 edition, 1994.

[Dor86] P. Doreian. Analysing overlaps in food webs. *J. Soc. & Biol. Structures*, 9:115–139, 1986.

[Eva68] Thomas G. Evans. A program for the solution of a class of geometric analogy intelligence-test questions. In *Semantic Information Processing*, chapter 5, pages 271–353. The MIT Press, 1968.

[FFG89] B. Falkenhainer, K. D. Forbus, and D. Gentner. Structure mapping engine. *Artificial Intelligence*, 41(1):1–63, November 1989.

[Fre97] Christian Freska. Spatial and temporal structures in cognitive processes. In C. Freska, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science*, volume 1337 of *LNCS*. Springer-Verlag, 1997.

[Gar97] H. Gardner. *Les formes de l'intelligence.* Éditions Odile Jacob, Paris, April 1997.

[Gen83] Dedre Gentner. Structure-mapping a theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, April-June 1983.

[GI92] M.-D. Gineste and B. Indurkhya. Modèles mentaux, analogie et cognition. In *Les modèles mentaux, approche cognitive des représentations*, chapter 6, pages 143–170. masson, Paris, 1992.

[GNC95] Janice Glasgow, N. Hari Narayanan, and B. Chandrasekaran. *Diagrammatic reasoning : Cognitive and Computational Perspectives.* AAAI Press/MIT Press, 1995.

[Gou80] P. Gould. Q-analysis, or a language of structure : An introduction for social scientists, geographers and planners. *International Journal of Man-Machine Studies*, 13:169–199, 1980.

[GV98] Jean-Louis Giavitto and Erika Valencia. Combinatorial algebraic topology for dagrammatic reasoning. Technical Report 1165, LRI, ura 410, april 1998.

[Ha91] J. P. Haton and al. *Le raisonnement en intelligence artificielle*. InterEditions, 1991.

[Hal89] Rogers P. Hall. Computational approaches to analogical reasoning : A comparative analysis. *Artificial Intelligence*, 39(1):39–120, may 1989.

[HE97] Kathleen Hornsby and Max J. Egenhofer. Qualitative representation of change. In D.Lukose, H.Delugach, M. Keeler, L. Searle, and J. Sowa, editors, *Spatial Information Theory*, volume 1257 of *LNCS*. Springer-Verlag, 1997.

[Hen94] M. Henle. *A combinatorial introduction to topology*. Dover publications, New-York, 1994.

[Hir97] Stephen C. Hirtle. Representational structures for cognitive space : Trees, ordered trees and semi-lattices. In D.Lukose, H.Delugach, M. Keeler, L. Searle, and J. Sowa, editors, *Spatial Information Theory*, volume 1257 of *LNCS*. Springer-Verlag, 1997.

[Hof84] D. R. Hofstadter. The Copycat Project: an experiment in nondeterminim and creative analogies. A.I. Memo 755, MIT Artificial Intelligence Laboratory, Cambridge Massachusetts, 1984.

[Hof95] Robert R. Hoffman. Monster analogies. *AI Magazine*, 16(3):11–35, 1995.

[HT89] K. J. Holyoak and P. Thagard. Analogical mapping by constraint satisfaction. *Cognitive Science*, 13(3):295–355, July-september 1989.

[HY88] J. G. Hocking and G.S. Young. *Topology*. Dover publications, New-York, 1988.

[Joh91a] J. Johnson. *The mathematical revolution inspired by computing*, chapter The mathematics of complex systems, Johnson J. and Loomes M. eds., pages 165–186. Oxford University Press, 1991.

[Joh91b] J. Johnson. *Transport Planning and Control*, chapter The dynamics of large complex road systems, Griffiths J. ed., pages 165–186. Oxford University Press, 1991.

[LP97] Oliver Lemon and Ian Pratt. Spatial logic and the complexity of diagrammatic reasonning. *Graphics and Vision*, 6(1):89–109, 1997.

[PLC91] J. Pimm, J. Lawton, and J. Cohen. Food web patterns and their consequence. *Nature*, 350:669–674, 25 April 1991.

[THNG90] P. Thagard, K. J. Holyak, G. Nelson, and D. Gochfeld. Analog retrieval by constraint satisfaction. *Artificial Intelligence*, 46(3):259–310, December 1990.

[Val97] Erika Valencia. Un modèle topologique pour le raisonnement diagrammatique. Rapport pour le DEA Sciences Cognitives, LIMSI. See also `http://www.lri.fr/~erika/Pro/erika.html`, August 1997.

[VG98] Erika Valencia and Jean-Louis Giavitto. Algebraic topology for knowledge representation in analogy solving. In AISB ECCAI, editor, *European Conference on Artificial Intelligence (ECAI98)*, pages 88–92, Brighton, UK, 23–28 August 1998. Christian Rauscher.

[WS90] S. H. Weber and A. Stolcke. l_0 : A testbed for miniature language acquisition. *International Computer Science Institute*, 1990.