L3-Internship

Formal Proof for Gene Clustering

13/04/13 — 5/07/13
Université Pierre et Marie Curie, Paris

Author:
Mathias Fleury

Supervisors:
Bin-Minh Bui-Xuan[†]
Frédéric Peschanski[‡]

UPMC
Laboratory LIP 6
APR Team
[†] buixuan@lip6.fr
[‡] frederic.peschanski@lip6.fr

31st December 2013

Abstract

We present a formal proof of UNO and YAGIURA's algorithm [16], using the theorem proving assistant COQ. This is a fundamental combinatorial algorithm which operates over permutation. Direct applications include computational biology (e.g. heuristics in estimating the genealogical distance between two species), text processing (e.g. generating the synchronous context-free grammar of two sequences with many-to-many alignment links [18]), the problem has been generalized to d permutations by HEBER and STOYE [10], . . .

This text is part of an internship project which took place at LIP6, Université Pierre & Marie CURIE, Paris, under the joint supervision of B.M. BUI-XUAN and F. PESCHANSKI.

Acknowledgment

I would like to thank my two supervisors Bin and Frédéric. The first one support my repetitive questions about UNO and YAGIURA's algorithm, even if he was not able to concern me about a few other arithmetical questions.

I would like to thank also the PhD students, in whose office I worked. They made me discover the work of a PhD student.

1 Introduction

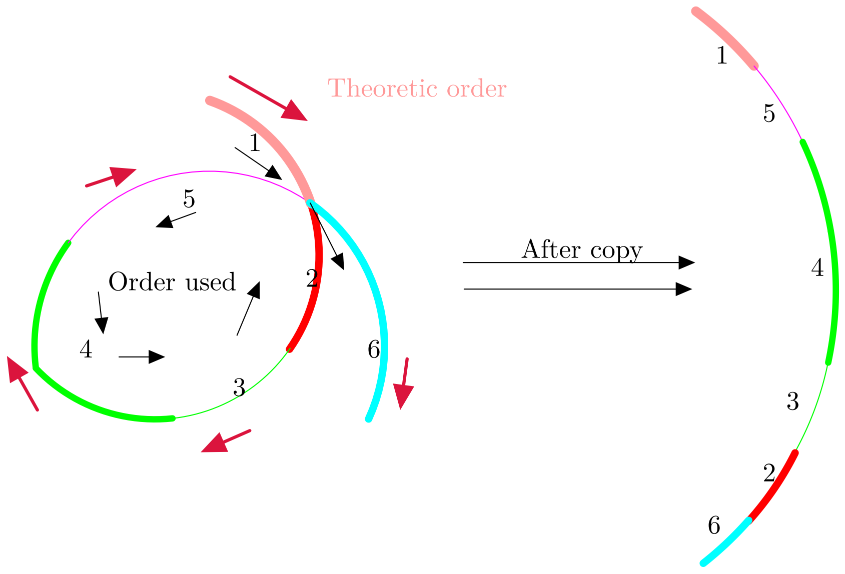
It is a fundamental task in computational biology to classify species, with respect to common ancestors such as in cladistics and taxonomy. For instance we would like to know the age of the last common ancestor of turtles and birds or the age of genetic 'Adam' and 'Eve'. The DNA can help determining this age, because if you know how fast changes happen and how many changes there are between them, you can guess an estimation. Some models have been described to simplify the problem, but the main idea is to work directly on genes and not on the succession of the DNA–bases (A, T, C and G[1]) since the DNA–base do change while there are different versions of genes, the alleles. One of the most important mechanisms that changes the order of the genes happens during the DNA duplicate process: since the DNA double helix is very long (more than one meter in one cell nucleus), while copying, there are loops which are sometimes copied in the wrong order. This can be seen on figure 1a on the following page. Such changes can be composed as shown in the figure 1b. The minimal number of reversals needed to go from one sequence to another is called *sorting by reversal* (SBR). The SBR problem is known to be NP-hard (if gene can appear more than once, even only twice[3]) or only polynomial if we know the direction of genes [2] (remark that the mentioned direction is reversed when the order changes in a loop). There are also approximation algorithms [11].

One way to simplify the problem is to look for the common intervals – sets of number that are consecutive in both. As we work with genes repeated only once, we consider the DNA as a permutation: we can consider, without loss of generality, that one of the two permutations, is the identity. Then a common interval is a set $\mathcal{S}$ such that there exists m and n with $\mathcal{S} = \sigma(\llbracket m; n \rrbracket)$. The splitting into intervals can help find the SBR (see figure 2 on page 4).

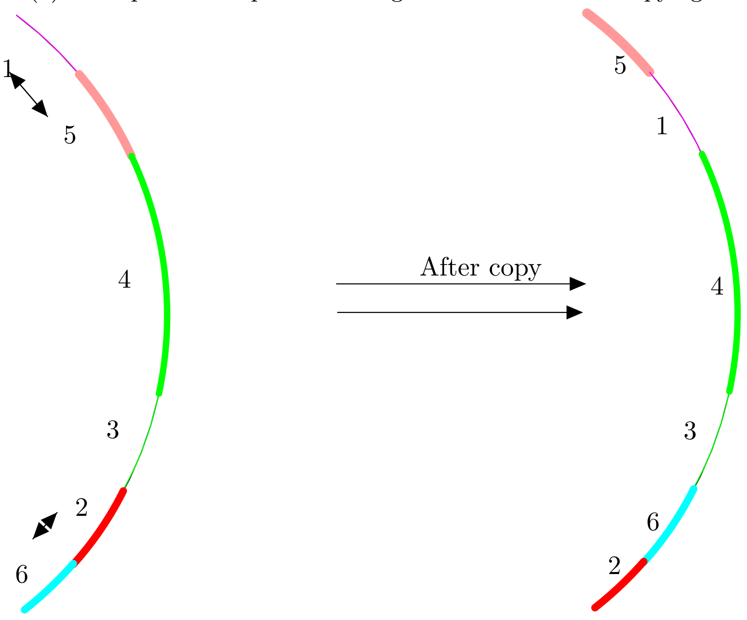
The notion of common intervals links to the deep theory of modular decomposition in graph theory. Indeed, if we draw the permutation as in the figure 3a on page 5, there are intersections: These intersections are the links of the permutation graph (figure 3b). A module of a graph is a subset of the nodes such that "compressing" them into one node does not change the view of the other: on the figure 3b, if we compress the nodes 6 to 8 into a node $6-7-8$, we do not change the ways the other nodes see the group. More formally a module is a subset X of the nodes such that all members of X have the same neighborhood outside X. Solving the modular decomposition problem in linear time has been a major challenge of the past decades, with more than ten papers in less than twenty years: [12], [13], [5], [9], [6], [1], [15]. It took over 10 years to go from "Linear-Time Modular Decomposition" to "Simple, linear-time modular decomposition". It can be shown that each common interval as $\{6; 7; 8\}$ is a module of the permutation graph. The reciprocal property does not hold, but in case of overlap-free common intervals (and strong modules), it is true. In fact the algorithm we will see later is a way to encode a graph in a linear way and to find the modular decomposition.

This report is organized as follows. In section 2, we present how to formalize the problem and the algorithm to find the common intervals. In section 3, we focus on the proof in Coq.

[1]adenine (A), cytosine (C), guanine (G), thymine (T), but it does not matter fur computer-scientist, even if it does for biologist



(a) Example of a loop and a change of the order while copying



(b) After two others inversions (look at the two dark arrows).

Figure 1: Example of gene reordering

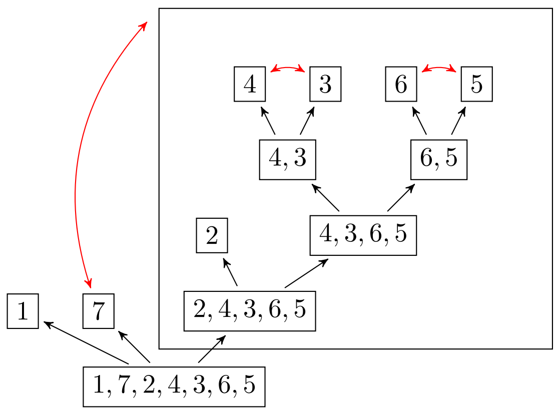


Figure 2: Decomposition of a permutation into some intervals: Red arrows show the elements to permute in order to find $[\![1;7]\!]$. There are intervals that are not shown as $\{2;4;3\}$

2 Formalization and algorithm

As we want to prove the correctness of the algorithm, we have to choose a proving assistant. We have chosen to use Coq. It is a theorem proving assistant: It can certify proofs, with user's help since logic is not decidable (although some parts such as Presburger arithmetic are). It is based on the idea of ML-language with dependent types (Calculus of Constructions or Coc, therefore the name) and written in Objective Caml. Several important proofs have been shown: the four color theorem [8] proven by Georges Gonthier [2], also the classification of finite simple groups, the proof of the Feit and Thompson theorem [7] (a preliminary theorem for the decomposition of finite simple groups). Another example is the CompCert compiler: it is a proven compiler of a subset of the C-language (Xavier Leroy). It is guaranteed, that if the program is correct (in the sense of the C-semantic), then the assembler instructions has the same behavior: More precisely the generated program is guaranteed to have one of the possible behaviors (since there are *undefined behavior* where everything can happen).

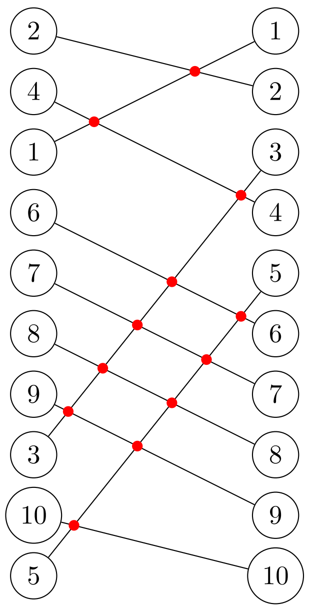
2.1 Problem description

2.1.1 Unsigned case

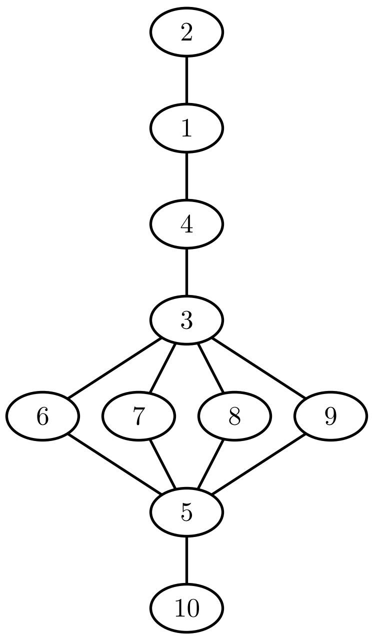
As we have seen in the last section, we consider the genes as unique, therefore we can number then as shown in the following table :

Reference species	1234567
Second species	1724365

[2]Who deeply changes the way Coq works with his extension `Ssreflect`: for example "2=3" has type Boolean, while it has type proposition in the normal Coq

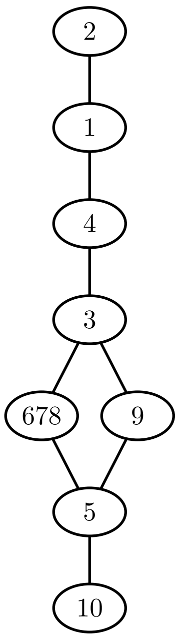


(a) Example of two corresponding sets, each intersection is marked by a red circle



(b) Permutation graph generated: each intersection becomes a link.

Figure 3: Example of a permutation and its given permutation graph



(a) The permutation where three nodes are compressed

The problem is formalized by considering that we have a permutation. In the example, it is:

i	1	2	3	4	5	6	7
σ	1	7	2	4	3	6	5

On this case, the common intervals are images of an interval by a permutation. In the example described above, as

$$\mathcal{S} := \{4; 3; 6; 5\} = \sigma\left(\llbracket 4; 7 \rrbracket\right)$$

the set $\mathcal{S}$ is an interval. Notice that both the whole set and singletons are common intervals.

More generally, we can find the distance between two permutations by renumbering, which means composing by a permutation in order to go to the last case: It is the same as searching for the distance between $\mathrm{Id} = \sigma_1 \circ \sigma_1^{-1}$ and $\sigma = \sigma_2 \circ \sigma_1^{-1}$.

2.1.2 Signed Case

If we want to work with the genes' direction, the permutation is said to be "signed": We consider one direction to be positive and the other as negative. We could have for example:

i	1	2	3	4	5	6	7
$\sigma(i)$	$+1$	-7	-2	$+4$	-3	-6	$+5$

This formalization is given as indication and won't be used further, as it is beyond the scope of our internship, but [2] explains the algorithm to find the transformation between the permutations.

2.2 Algorithm

Now that we have seen how to describe this problem formally, we will present a few algorithms to find common intervals.

2.2.1 Naively

The most simple method is to test every possible link between both permutations and testing if it is an interval, by verifying if all values between minimum and maximum are included in it. This method has a complexity of $\mathcal{O}(n^5)$, and is therefore polynomial[3].

2.2.2 A better method

The main idea of the characterization is to find a link between the length of the interval and the values inside. As there are no duplicate, we have the following property:

$$I \text{ is an interval} \Leftrightarrow \max I - \min I + 1 = |I|$$

[3] The polynomial complexity is not contradicting with the NP-completeness describe above: we are not speaking from the same problem (sorting by reversal distance and finding common intervals).

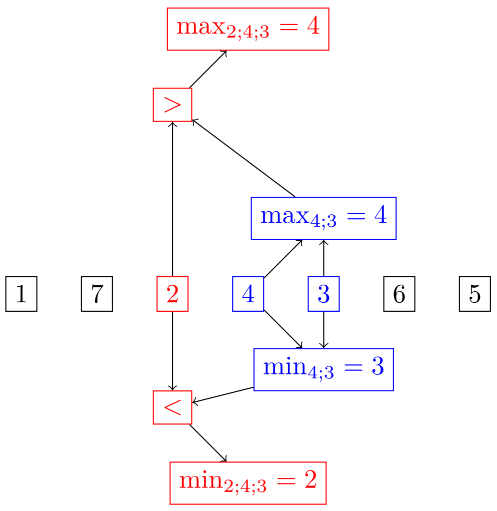


Figure 5: Quadratic algorithm: reevaluation of minimum and maximum after adding one element (here 2 in red), is simple since we have only to test whether $2 < 4 = \max$ or $2 > 3 = \min$. In fact only one case is possible: One test is enough.

where $|I|$ is the length of the interval.

To take an example $I = \{7; 5; 4; 6\}$, there are at most $\max I - \min I + 1 = 7 - 4 + 1 = 4$ elements for $|I| = 4$ places. Therefore I is an interval.

2.2.3 Using it

The first idea is to have two indexes i and j ($j \leqslant i$) that we change to test every interval. This algorithm has a complexity $\mathcal{O}\big(n^2\times$ evaluation of maximum and minimum $\big)$, i.e.:

1. if we have to reevaluate the maximum and minimum at every test, the complexity is $\mathcal{O}(n^3)$, since evaluating minimum and maximum is $\mathcal{O}(n)$
2. hopefully, as we add only value each time, reevaluation is a $\mathcal{O}(1)$, since only two comparisons are necessary (see figure 5 on the current page)

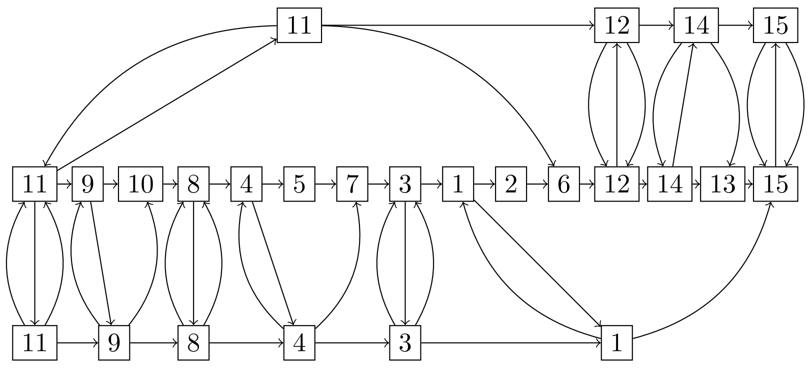
2.3 To a linear algorithm

The first version of the algorithm was published in 2000 by Uno and Yagiura [16], both bio-computer scientists. The complexity of that algorithm is $\mathcal{O}(n+K)$ where K is the number of common intervals ($K = \mathcal{O}(n^2)$ and it is reached for example for $[\![1; n]\!]$). Their proof was quite unclear, therefore it lasted five years until their paper was accepted.

Their algorithm was enhanced by Bui Xuan [17], to have a linear complexity in $\mathcal{O}(n)$, without a dependence of the number of intervals.

The algorithm is quite complex since it uses three single-linked lists [4] (see

[4]the original article speaks about doubly-linked lists, that's better (and easier to program), but not necessary



interval list: {11,15}; {11,13}; {11,12}; {11,6}; {11,8}; {11,10}; {11,11}; {9,6}; {9,8}; {9,10}; {9,9}; {10,10}; {8,6}; {8,8}; {4,6}; {4,5}; {4,4}; {5,5}; {7,7}; {3,2}; {3,3}; {1,2}; {1,1}; {2,2}; {6,6}; {12,15}; {12,13}; {12,12}; {14,15}; {14,13}; {14,14}; {13,13}; {15,15}

Figure 6: Example of the quiet complex data structure.

figure 6 on this page). We will first discuss the general principles of the algorithm.

2.3.1 Main Idea

The main idea of the algorithm is to work on the variations of the max and min functions. We remember those variations in three lists:

i) one containing the variations of the maximum function;

ii) the potential list where we maintain the elements of the permutation;

iii) one containing the variations of the minimum function.

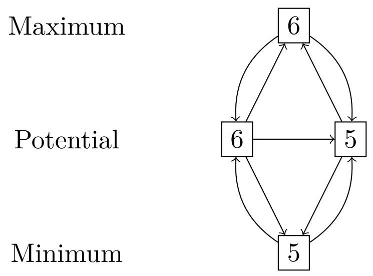
We maintain also pointers between the lists in order to find the correspondence easily.

We have to maintain this structure and its properties, when a number is added. After that, we can watch, if it is (or not) an interval. Thanks to the pointers, finding the maximum and the minimum costs $\mathcal{O}(1)$.

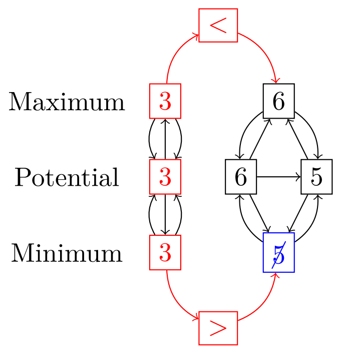
As the above description is very difficult to understand, we will now go throw an example. First of all, we will see an algorithm, which manipulate the same structure than that of the linear algorithm, but whose complexity is $\mathcal{O}(n^2)$.

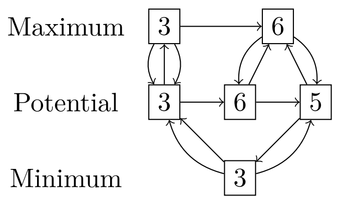
2.3.2 Example, quadratic algorithm

Maintaining the structure when adding an element We illustrate it on an example. We start from the structure below:

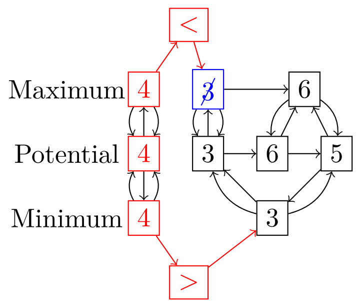


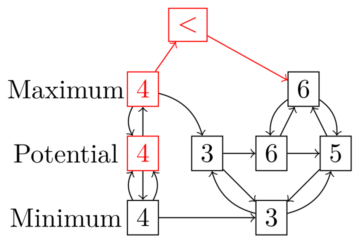
We want to insert a new value 3. We insert 3 in every of the three lists. Then we compare: In the minimum list, since $3 < 5$, we have to remove 5. In the maximum list, as $3 < 6$, we do not have to remove 6. After that, we have to update the value the pointers point on: In the minimum list 3 is the minimum of all three elements, so the right pointer of the value 5 that we removed, become the right pointer of the value 3. It is clear that 3 is the minimum of the values 3 to 6. In the maximum list, we only have to add 3 in the list. 6 is the maximum of the values 6 to 5 and 3 is the maximum of the single value 3.



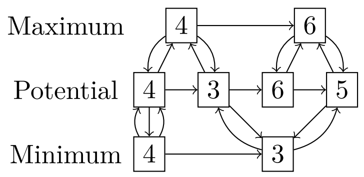


Now we want to insert 4. To do so, in the minimum list, we test whether $4 > 3$ or not. As it is, the minimum of the part of list (that is covered by the minimum values 4 and 3) does not change, and we only have to insert 4 in the minimum list. In the maximum list, as $4 > 3$, 4 is the maximum of the part covert by the two maximum values 4 and 3: We have to remove 3, from the maximum list.



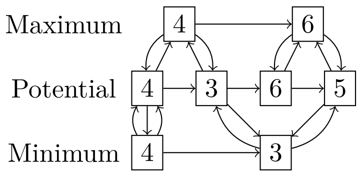


We do a new comparison: As $4 < 6$, we just have to insert 4 in the maximum list. 4 is the maximum of the part covered by the value (aka 4 and 3), and 6 is the maximum of 6 and 5).

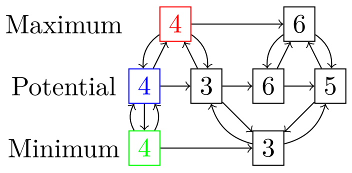


Searching for the common intervals After every insertion, we go through the potential list to find the common intervals. To watch for the maximum (resp. minimum) of the list, we begin with the element we have just added, and then we check: If there is a pointer, we choose the value pointed; else we keep the maximum (resp. minimum) computed before.

We begin with our three-list structure. We begin with the first value of the potential list, here 4. There are two pointers starting from the considered element: One to the maximum value 4 and one to the minimum value 4. We test with our characterization seen before: $4 - 4 + 1 = |\{4\}|$, so it is an interval.

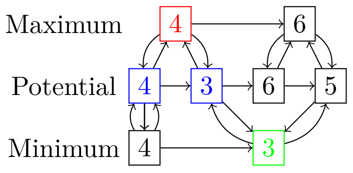


Found interval : $\emptyset$



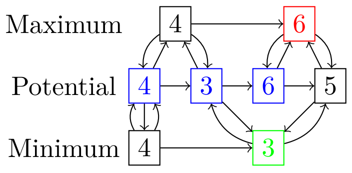
Found interval : $\{\{4\}\}$

Now we add the next value of the potential list, here 3: We now consider the set $\{4; 3\}$. the maximum does not change (4) and the minimum does (3). Using the characterization, we see that it is an interval. After that we consider the set $\{4; 3; 6\}$. As 6 as no pointer to a minimum value, we do not have to change the minimum value we have seen before (3), whereas the maximum changes and become 6. Using the characterization, we can see that it is not an interval.



Found interval :

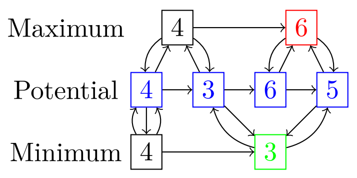
$\{\{4\}; \{4; 3\}\}$



Found interval :

$\{\{4\}; \{4; 3\}\}$

We do the same process as before, and see that $\{4; 3; 6; 5\}$ is a set. As 5; was the last element is our potential list, we can stop the process.



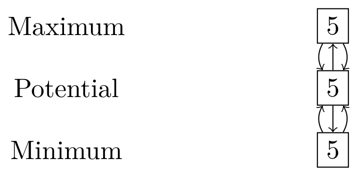
Found interval :

$\{\{4\};\{4;3\}\};\{4;3;6;5\}\}$

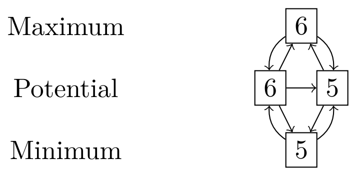
2.3.3 The linear version: Removing elements in the potential list

The main idea is the same as in the previous section, but we can remove some elements of the potential list. If we remove n maximums (resp. minimums), then we remove the $(n-1)$ corresponding values in the potential list. Moreover, we test the intervals and stop when reaching a set which is not an interval.

We start with the same example as before: When we inserts 6, there is no change like the algorithm described above.



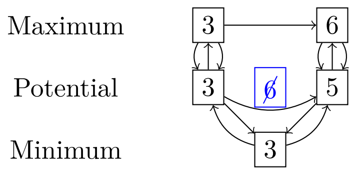
$\{5\}$



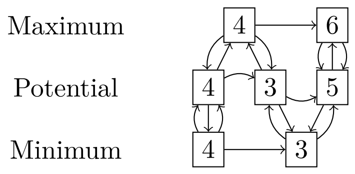
$\{5\};\{6\};\{6;5\}$

On the left figure below, when we add 3, we remove 6 (blue), because of the update of the minimum: the value 5 is “eaten”.

On the right figure, this step of the algorithm is the same than the algorithm described above. As the inserting of a value and the searching of common intervals are the same than the previous algorithm, there are not described.

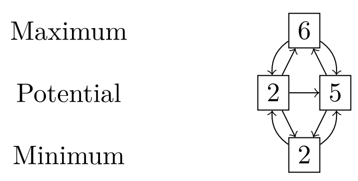


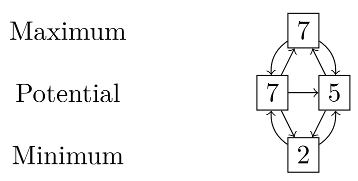
$\{5\};\{6\};\{6;5\};\{3\}$



$\{5\};\{6\};\{6;5\};\{4\}$
$\{4;\ldots;3\};\{4;\ldots;5\}$

On the left figure, we want to insert 2, as it is the minimum of the whole list, all the values of the minimum list are “eaten”. Therefore we remove all values of the potential list except the last one. As we can see, searching for the common intervals is much easier and it is shorter than testing all possible common intervals. we can also notice, that values in the maximum list, have not to be in the potential list anymore.





$\{5\}; \{6\}; \{6; 5\}; \{4\}$
$\{4 \ldots 3\}; \{4 \ldots 5\}; \{2 \ldots 6\}$

$\{5\}; \{6\}; \{6; 5\}; \{4\}$
$\{4 \ldots 3\}; \{4 \ldots 5\}; \{2 \ldots 6\}$
$\{7 \ldots 5\}$

This algorithm seems very surprising, but it was the aim of the internship to show that it is indeed correct.

3 Formal proof in Coq

3.1 Proof of the first quadratic algorithm

3.1.1 Technical choices

Normally, the algorithm is implemented with arrays, which is impossible directly in Coq: it is a pure functional language without side-effects. We have chosen to work with lists, manipulated as arrays: we have worked on the position inside a list (it is the function called `list_at` ℓ i: it gives access to i-th element of the list), and not on the inductive principles on lists.

That has changed the way we worked: In Coq, all functions have to terminate and we cannot work on the lists property. The reason why every function have to terminate is easy: all proofs are conducted by induction. `Fixpoint` is the keyword to explain to Coq that a function is recursive.

```
Fixpoint fonction_qui_ne_finit_pas (x:nat)
   := fonction_qui_ne_finit_pas x
```

This function has type `nat -> A` where `A` can be any type. So we can choose `A=False`. After that the proofs become easy, as every proposition can be derived from `False` and we get `False` without hypothesis. The logical system becomes less interesting.

To simplify the termination problem and in order to make Coq to accept the definitions; instead of manipulating two indices i (beginning index) and j (ending index), we will manipulate j and $\delta = j - i$. The proof that the function stops, will be the structural decreasing of delta (it means, delta decreasing until 0, which is a terminal case). This structural decreasing is automatically detected by Coq: There is nothing special to do.

To make the link between our program and the one working on arrays, we should develop a memory model, where arrays would be represented as contiguous cells, but it is not the most important point when discussing correctness (not complexity).

We use Coq's libraries and Pierce files in his book *Software foundation* [14], where a function is defined and is able to give labels to subgoals in proofs (useful to structure induction proofs).

3.1.2 Idea of the proof

To prove the correctness of the algorithm, We have formalized the equivalence between the program evaluating maximum and minimum at each step. Proving that this program is correct is easy, because the definition of the intervals is $\max \ell - \min \ell + 1 = |\ell|$ and we try every combinations. We have chosen that definition of interval because working with permutations was far more difficult than initially expected.

3.1.3 Organization of the proof

The proof is divided into a few files:

- `list.v`: it contains a few properties of lists, and the definition of maximum and minimum (and we prove that `find_max` and `find_min` have the expected properties)
- `algon3_boucle_interieur.v` contains the definition of the inner loop of the algorithm, that means that at final index fixed, we begin to vary the initial index. It contains also a few proofs.
- `algon3_boucle_exterieur.v` contains the definition of the outer loop and the proof
- the more efficient version of the algorithm is divided into two files:
  1. `algon2_boucle_interieur.v` shows the equivalence of both inner loops: with the same arguments, they produce the same output.
  2. `algon2_boucle_exterieur.v` is the whole program (the equivalence is shown).

An extract of the pdf file generated by `coqdoc` can be find in the annex (see **??** on page ??).

3.2 Linear algorithm

3.2.1 Pointers

One of the main difficulties of the algorithm is to implement pointers: in Coq, there exists no notion of native pointer. We have chosen to remember the positions in the list (in the reverse order to avoid renumbering at each step: 0 on the tail and $|\ell|$ on the head). We were also able to remove a few pointers by using lists instead of double linked lists.

To prove the algorithm with real pointers, we should develop a memory model, where we make a link between the position in the list and the memory: for example instead of remembering that the pointer points to a the third element of the list, it points to an address (like 00FF74). But it won't change the correctness part of the proof: it is another level of abstraction, but it relies on the proof, if we prove a bijection with the basic pointer operation.

3.2.2 Termination

Principle Since we manipulate three lists (internally described by a type), we often must prove that at least one is decreasing each time we make a recursive call. As Coq is not able do detect it automatically, we have to prove it "by hand", with the help of the keyword `Function`: We must add two arguments, one is the decreasing function and the other is the argument of the function that is decreasing. When trying to define the function, it generates a goal corresponding to the termination proof: the function is defined only when the termination is proven. In our case the function is the length of one of the lists: therefore we have only to prove that $|\ell| < |a :: l| \Longleftrightarrow |\ell| < |\ell| + 1$, which is true by theorem `lt_n_Sn`.

Example First we need a type:

```
(*the type LLL is defined as a function (uyl) with three
arguments: three integer lists:*)
Inductive LLL : Type :=
uyl : list nat → list nat → list nat → LLL
```

Then we use a function that will extract the second list of a LLL-element, by working only on the second list, and making it decreasing

```
Function test (lll:LLL) {measure get_length} :=
match lll with
    |uyl lu ly ll ⇒
       match ly with
          | [] ⇒ []
          | a::ly' ⇒ a::(test (uyl lu ly' ll))
       end
end.
```

This code generates a theorem to prove:

```
(*First all the used variables*)
∀ lll lu ly ll a,
(*then there are the assigned values*)
ly = a::ly'⇒ lll = uyl lu (a::ly') ll ⇒
(*The decreasing we want to prove.*)
get_length (uyl lu ly' ll) <
                        get_length (uyl lu (a::ly') ll)
```

After simplification, we have to prove that (we wrote $|\cdot|$ instead of `get_length`):

```
|lu|+|ly|+|ll| < |lu|+S(|ly|)+|ll|
(* (S n) is the successor of n*)
```

which can be solved by the `Omega`-tactic (that implements a method to solve a goal in Presburger arithmetic).

3.2.3 Proofs

To prove the quadratic algorithm (the one where we do not remove elements in the potential list), our idea was to demonstrate that the elements are the maximum and the minimum. But the proof becomes very complicated because of the pointers; we were not able to finish it, although we tried to specify the function used. The work we have done, represents more than ten thousand lines of code.

Conclusion

We present formal theory proving tools to be used under Coq and subsequently use them in order to prove algorithms from Uno and Yagiura seminal paper [16]. It would have been interesting to simplify the proofs with parts in $\mathcal{L}$tac (the tactic language in Coq[5]) to find what is really important and what is less useful. The last linear algorithm is left unproven due to complexity of the underlying data structure, but the foundations for such a proof are now established.

It would be interesting to combine our work with tools in order to achieve Uno and Yagiura's algorithm. Proving this algorithm would act as a major step toward a formal proof of modular graph decomposition algorithm (in the case of permutation graph): the modular graph decomposition can be divided into two parts, first Uno and Yagiura's algorithm and then a second part to conclude.

References

[1] Luca Aceto et al., eds. *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*. Vol. 5125. Lecture Notes in Computer Science. Springer, 2008. ISBN: 978-3-540-70574-1.

[2] David A. Bader, Bernard M.E. Moret and Mi Yan. 'A Linear-Time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study'. In: vol. 8. 2001, pp. 483–491.

[3] Guillaume Blin, Cedric Chauve and Guillaume Fertin. *The breakpoint distance for signed sequences (extended abstract)*. 2004.

[4] Adam Chlipala. *Certified Programming with Dependent Types*. `http://adam.chlipala.net/cpdt/`. 2013.

[5] Elias Dahlhaus, Jens Gustedt and Ross M. McConnell. 'Efficient and Practical Algorithms for Sequential Modular Decomposition'. In: *J. Algorithms* 41.2 (2001), pp. 360–387.

[6] Elias Dahlhaus, Jens Gustedt and Ross M. McConnell. 'Efficient and Practical Modular Decomposition'. In: *SODA*. 1997, pp. 26–35.

[7] Georges Gonthier. 'Engineering mathematics: the odd order theorem proof'. In: *POPL*. 2013, pp. 1–2.

[8] Georges Gonthier. 'The Four Colour Theorem: Engineering of a Formal Proof'. In: *ASCM*. 2007, p. 333.

[5] See an introduction in [4]

[9] Michel Habib, Fabien de Montgolfier and Christophe Paul. 'A Simple Linear-Time Modular Decomposition Algorithm for Graphs, Using Order Extension'. In: *SWAT*. 2004, pp. 187–198.

[10] Steffen Heber and Jens Stoye. 'Finding all common intervals of k permutations'. In: *In Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001*. Springer Verlag, 2001, pp. 207–218.

[11] Petr Kolman. 'Approximating reversal distance for strings with bounded number of duplicates'. In: *In Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science (MFCS), volume 3618 of Lecture Notes in Computer Science*. 2005, pp. 580–590.

[12] Ross M. McConnell and Fabien de Montgolfier. 'Linear-time modular decomposition of directed graphs'. In: *Discrete Applied Mathematics* 145.2 (2005), pp. 198–209.

[13] Ross M. McConnell and Jeremy Spinrad. 'Linear-Time Modular Decomposition and Efficient Transitive Orientation of Comparability Graphs'. In: *SODA*. 1994, pp. 536–545.

[14] Benjamin C. Pierce et al. *Software Foundations*. `http://www.cis.upenn.edu/~bcpierce/sf`. Electronic textbook, 2012.

[15] Marc Tedder et al. 'Simple, linear-time modular decomposition'. In: *CoRR* abs/0710.3901 (2007).

[16] Takeaki Uno and Mutsunori Yagiura. 'Fast Algorithms to Enumerate All Common Intervals of Two Permutations'. In: *Algorithmica* 26 (2000), p. 2000.

[17] Binh-minh Bui Xuan, Michel Habib and Christophe Paul. 'Revisiting T. Uno and M. Yagiura's algorithm'. In: *Proc. 16th International Symposium on Algorithms and Computation, in Lecture Notes in Comput. Sci.* Springer, 2005, pp. 146–155.

[18] Hao Zhang, Daniel Gildea and David Chiang. 'Extracting synchronous grammar rules from word-level alignments in linear time'. In: *In Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*. 2008.