Documentation of the C code used to calculate the path persistent homology (aka pph) up to diagrams of dimension 0 and 1

Rafael Polli Carneiro

Contents

1 Initiall Observations

Be warned that henceforth we will admit the following conventions:

i) Whenever talking about vector spaces, namelly V, we have that the field acting onto V is going to be $\mathbb{Z}_2$;

ii) Every vector space here has finite dimension;

2 How The Program is structured

Everything here, concerning the `C` code, is splitted into the two folders: *headers*, *src*. Inside headers one can find the data types which are used to perform the main algorithm, as well the respectivelly functions that operate with these data types. Now, *src* is merely the `C` implementation of such functions, or methods if someone prefers to call them. The `C` code called *main.c*, runs the algorithm desired to be calculated.

3 Headers

The folder Headers has the following tree structure:

```
headers
│
├── definitions.h
├── basis_of_vector_space.h
├── persistent_path_homology.h
└── Tp.h
```

Figure 1: tree headers.

Now let's describe each file contained at `headers`.

3.1 *definitions.h*

Here, basic, yet fundamental data types are defined. To begin with, we have the following constants:

- `TRUE` or `FALSE`;
- `MARKED` or `NOT_MARKED`. Here, these flags address whether or not an element of a vector space, more precisely, an element of its basis, is marked of not.
- `EMPTY` or `NOT_EMPTY`. We say that the ith element of an array can be empty or not, meaning that nothing has been allocated at its memomery space, if it is empty, and the contrary otherwise;
- `SORTED` or `NOT_SORTED`. ???;

3.1.1 Data types

The data types are:

- `boolean` = TRUE or FALSE;

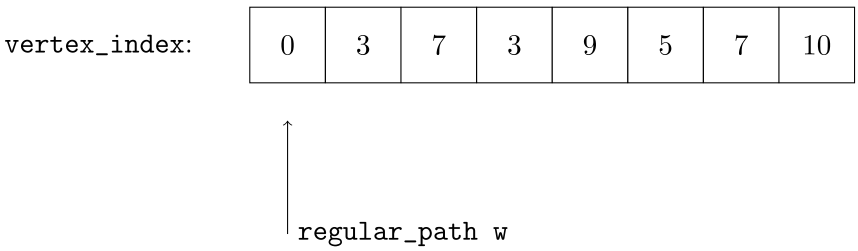

Figure 2: Representation of a regular path $w = [0, 3, 7, 3, 9, 5, 7, 10]$.

- `vertex_index` $= 0, 1, 2, \ldots, N$. Given a graph (G, V), each vertex (or node if you prefer to call it), will be enumerated by integers ≥ 0;
- `base_index`. Let V be a vector space. Then, considering $\beta \subseteq V$ as its base, with $\beta = \{v_0, v_1, \ldots, v_N\}$, we say that $0, 1, 2, \ldots$ will be elements of the `base_index`;
- `vector_index`. Given a enumeration of vectors, `vector_index` stores the indexes of such enumeration. For instance, let the vectors $v_0, v_1, v_2, \ldots$. Then the indexes $0, 1, 2, \ldots$ are elements of `vector_index`.
- `dim_path`. Let (G, A) be a graph. Then $w = [a_0, a_1, \ldots, a_N]$ is called a regular path in G if $\forall i \in \{0, 1, \ldots, N\}$ the following holds

  $$a_i \in G \quad \text{and} \quad (a_0, a_1), (a1, a_2), \ldots, (a_{N-1}, a_N) \text{are edges of the graph.}$$

  The regular path w, in this case, is said to have dimension N. Consequently, `dim_path` will be the data type responsible of representing all dimension of possible regular paths being used in the future;
- `dim_vector_space`. The meaning of this data type can be understood immeadiately;
- `regular_path`. A regular path is exactly what was described up abpve, that is, $w = [a_0, a_1, \ldots, a_N]$ is a regular path when each a_i is a node, forall i, and every pair $(a_0, a_1), (a_1, a_2), \ldots, (a_{N-1}, a_N)$ is an edge. It is natural to represent a regular path as an array and, in fact, that is the way they will be represented in here. Thus, the type `regular_path` is, as expected, a pointer to `vertex_index` types. See Figure 2 for a visual explanation.
- `vector`. Considering V a vector space with basis $\beta \subseteq V$, then any element of V can be seen as its coordinates. For instance, let $w = (0, 0, 0, 0, 1, 1, 0) \in V$. The data type `vector` is, as `regular_path`, a pointer to `boolean`, which are elements of $\mathbb{Z}_2$. For large graphs, and due to the fact that the array vector is filled with many zeros (in my

analysis), this data type is not the best to be used. I WILL CHANGE IT FOR A LIST!!!

3.1.2 Functions

Finnaly, the following functions, down below, will implement some routines over the data types defined into `definitions.h`. They are:

are_ these_ regular_ paths_ the_ same

Parameters : `regular_path`,`regular_path`, `dim_path`.

Objective : Check if two regular paths w_1, w_2 are equal. In case positive it returns TRUE, and FALSE otherwise.

Return : `boolean`.

`is_this_path_a_regular_path`

Parameters : `regular_path`, `dim_path`.

Objective : It checks if an the array $w = [a_0, a_1, \ldots, a_N]$ is a regular path, that is, the pairs $(a_0, a_1), (a_1, a_2), \ldots, (a_{N-1}, a_N)$ are, indeed, edges of the graph. This function is important specially when taking a border operator of a regular path which, in most cases, the outcome can incorporate non regular paths. It returns TRUE if w is a regular path, and FALSE otherwise.

Return : `boolean`.

`is_this_vector_zero`

Parameters : `vector`, `dim_vector_space`.

Objective : It checks if a vector v of a vector space is zero, returning TRUE if positive and FALSE otherwise

Return : `boolean`.

`sum_these_vectors`

Parameters : `vector`, `vector`, `dim_vector_space`.

Objective : Let w_1, w_2 two elements of a vector space. Then this function returns a vector with the value equal to the sum $w_1 + w_2$.

Return : `vector`.

3.2 basis_of_vector_space.h

This header will provide us an abstraction of vector spaces which are spanned by a collection of regular paths. Bare in mind that in our context we have a sequence of sets of regular paths, indexed by $i \in \mathbb{R}_{\geq 0}$, given by

$$R_i^p = \{[a_0, a_1, \ldots, a_p]; f(a_0, a_1) \leq i, f(a_1, a_2) \leq i, \ldots, f(a_{p-1}, a_p)\},$$

where $a_0, a_1, \ldots, a_p$ nodes of a graph and $f : G \times G \to \mathbb{R}_{\geq 0}$ a matrix. This provide a sequence

$$\mathbb{R}_i \subseteq \mathbb{R}_j,$$

whenever $0 \leq i \leq j$. This is called a filtration, and it will provide us times which homological features appears and dissapear. With that in mind, for every regular path $w = [a_0, a_1, \ldots, a_p$, we define

$$\text{allowtime}(w) := \inf\{i \in \mathbb{R}_{\geq 0}; f(a_0, a_1) \leq i, f(a_1, a_2) \leq i, \ldots, f(a_{p-1}, a_p)\}$$

and

$$\text{entrytime}(w) := \min\{\text{allowtime}(w), \text{allowtime}(\partial(w))\};$$

with f that matrix associated with the graph and ∂ the border operator.

3.2.1 Data Types

With these operators in mind we start by the following data types:

- `tuple_regular_path_double`. Given a base β of regular paths, then `tuple_regular_path_double` is a structure composed by the ith regular path of the base β and by the allow time of such regular path. See Figure 3.

- `base`. This is a structure that will be responsible to store all elements of a base which are regular paths of same dimension. Thus `base_matrix` is a pointer to `tuple_regular_path_double` where the regular paths must have the dimendion given by `dimension_of_the_regular_path`. The dimension of the vector space spaned by the base `base_matrix` is stored at `dimension_of_the_vs_spanned_by_base`. Finally, `marks` is an array of the same size of `base_matrix` where it shows if the ith element of the base is marked or not. See Figure 3. Notice that in the Figure 3 only `base_matrix` is represented, while the other features are left inside the center dots (this is done beacuse `base_matrix` is the object that should be paid more atention).

- `collection_of_basis`. This is a structure which can be seen as the union of every base indexed by the dimension of the paths that generates each vector space. Here `max_of_basis` counts how many basis do we have, taking into consideration that we start counting from zero. See Figure 3.

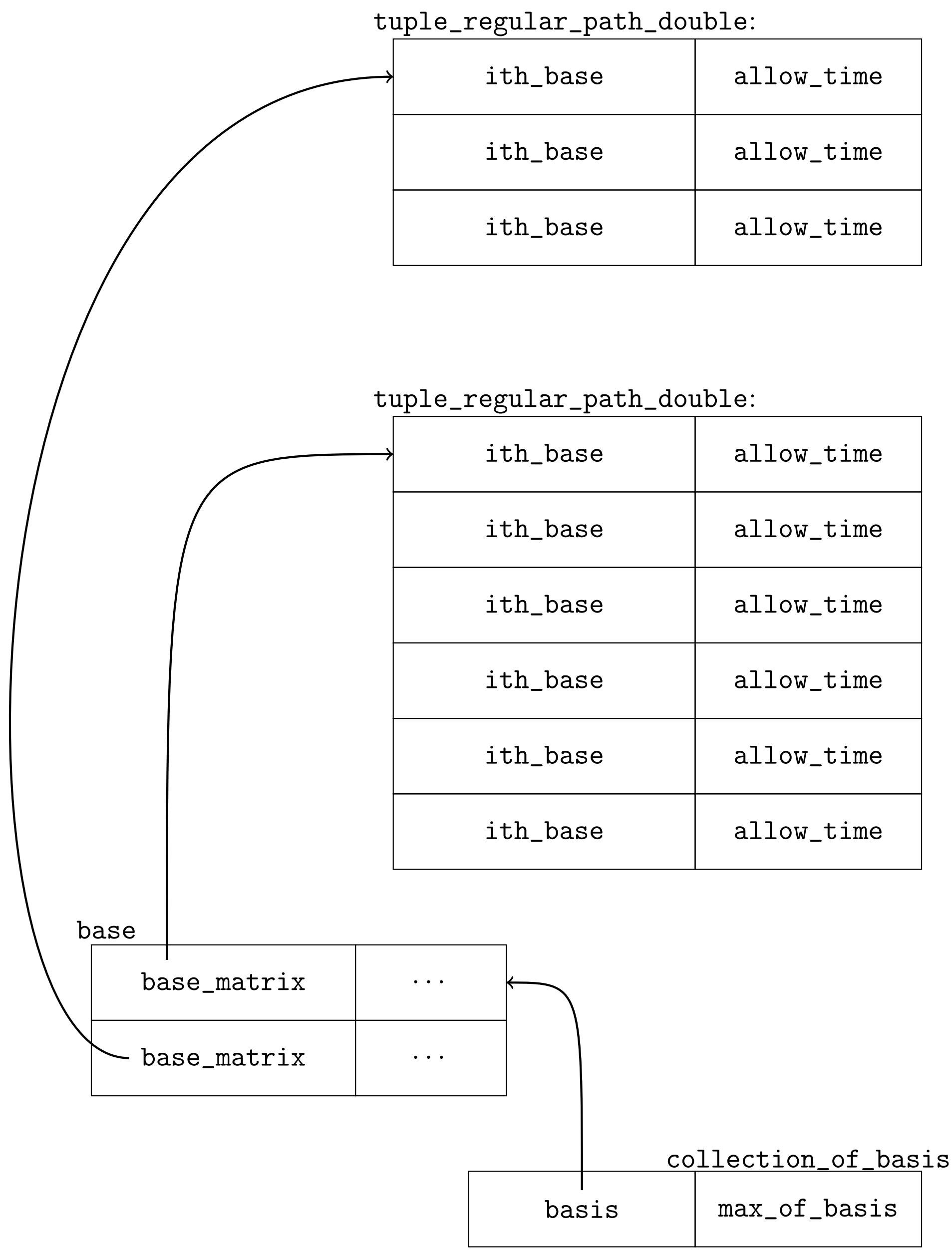

Figure 3: Visual reresentation of the data types defined at the file `basis_of_vector_space.h`.

3.2.2 Functions

We start with the two functions responsible for allocating all possible regular paths into their respectively basis:

alloc_ all_ basis

Parameters : `unsigned_int`, `unsigned_int`, `double**`.

Objective : Given an integer N, this function will allocate $N+1$ basis, taking as reference the amount of nodes of the graph and its network weight matrix.
(Obs: the function `generating_all_regular_paths_dim_p` is used)
Also, regular paths of dimension 0 are stored by this function.

Return : `collection_of_basis*`.

generating_ all_ regular_ paths_ dim_ p_ version2

Parameters : `collection_of_basis*`, `double**`.

Objective : Given a `collection_of_basis*` this function will all regular paths of dimension 1 and 2 into the structures which represent the basis of the vector spaces spanned by such regular paths. *generating_ all_ regular_ paths_ dim_ p_ version2* will be used by the function above.

Return : `void`.

Now, in order to the main algorithm to work we need to be able to "mark" a vector basis. The first thing that I do is to initialize arrays where their index are in correspondence with the index of the `base_matrix`. Then, a function is definined so, in the runtime, it is possible to mark a given vector basis.

initialize_ Marking_ basis_ vectors

Parameters : `collection_of_basis`

Objective : For each dimension of regular paths up to `max_of_basis`, an array, namely `marks`, of size `dimension_of_the_vs_spanned_by_base` is created so we will be able to say if a vector basis is marked or not

Return : void

marking_ basis_ vectors

Parameters : `collection_of_basis`, `dim_path`, `base_index`

Objective : Mark a vector of a basis where all regular paths have dimension equal to a `dim_path`.

Return : void

When we have all basis of regular paths allocated in memory we need to sort each base by the allow time of its elements. In other words, let

$\beta = \{a_o, a_1, a_2, \ldots, a_p\}$ be the base of the vector space spanned by regular paths of dimension N. Then we need to ensure that

$$\text{allowtime}(a_0) \leq \text{allowtime}(a_1) \leq \text{allowtime}(a_2) \leq \cdots .$$

This is done by the function

sorting_ the_ basis_ by_ their_ allow_ times

> **Parameters** : `collection_of_basis`
>
> **Objective** : Sort each element of a `base_matrix` by the allow time of each `tuple_regular_path_double`.
>
> **Return** : void

which uses the function:

compareTuple

> **Parameters** : `tuple_regular_path_double`, `tuple_regular_path_double`
>
> **Objective** : This function induces a partial order into the space of regular paths by compairing allow times.
>
> **Return** : `int`

to induce a partial order in the space of regular paths of same dimension.

Another important function is the one resposible to calculate the allow time of any regular:

allow_ time_ regular_ path

> **Parameters** : `network_weight**`, `regular_path`, `dim_path`
>
> **Objective** : It calculates the allow time of a `regular_path` with dimension `dim_path`.
>
> **Return** : `double`

Last but not least, the getters and setters! Their meaning are straightforward.

get_ dimVS_ of_ ith_ base

> **Parameters** : `collection_of_basis**`, `dim_path`
>
> **Objective** : It returns the dimension of the vector space spanned by regular paths of dimension `dim_path`
>
> **Return** : `dim_vector_space`

set_ dim_ path_ of_ ith_ base

> **Parameters** : `collection_of_basis**`, `dim_path`

Objective : It sets the dimension of the regular paths that are going to be used

Return : `void`

set_dimVS_of_ith_base

Parameters : `collection_of_basis**`, `dim_path`, `dim_vector_space`

Objective : It sets the dimension of the vector space spanned by regular paths of dimension `dim_path` to the values of `dim_vector_space`

Return : `void`

get_path_of_base_i_index_j

Parameters : `collection_of_basis**`, `dim_path`, `base_index`

Objective : It returns a regular path of dimension `dim_path` which has the index of `base_index`

Return : `regular_path`

is_path_of_dimPath_p_index_j_marked

Parameters : `collection_of_basis**`, `dim_path`, `base_index`

Objective : It returns true if a regular path of dimension `dim_path` is marked and false otherwise

Return : `boolean`

3.3 `Tp.h`

Here we have the main structure responsible to take care with the gaussian elimination.

3.3.1 Data Types

The data types in here are going to be something equal to the ones found to allocate the basis above.

- `T_p_tuple`. A structure rpresenting a tuple (v, e, m) where v is a `vector` whose coordinates represent the regular paths that decomposes v. Also, e is the `double` storing an entry time and m is a `boolean` which informs if this tuple is empty or not.

- `T_p_tuple_collection`. A collection of `T_p_tuple` which are indexed by a base of regular paths of the same dimension that spans a vector space of dimension `size`

- `T_p`. A collection of all `T_p_tuple_collection` up to `max_of_Tp`, which will be equal to 1.

Check Figure 4 to understand these data types and notice that they follow the same pattern of the Figure 3.

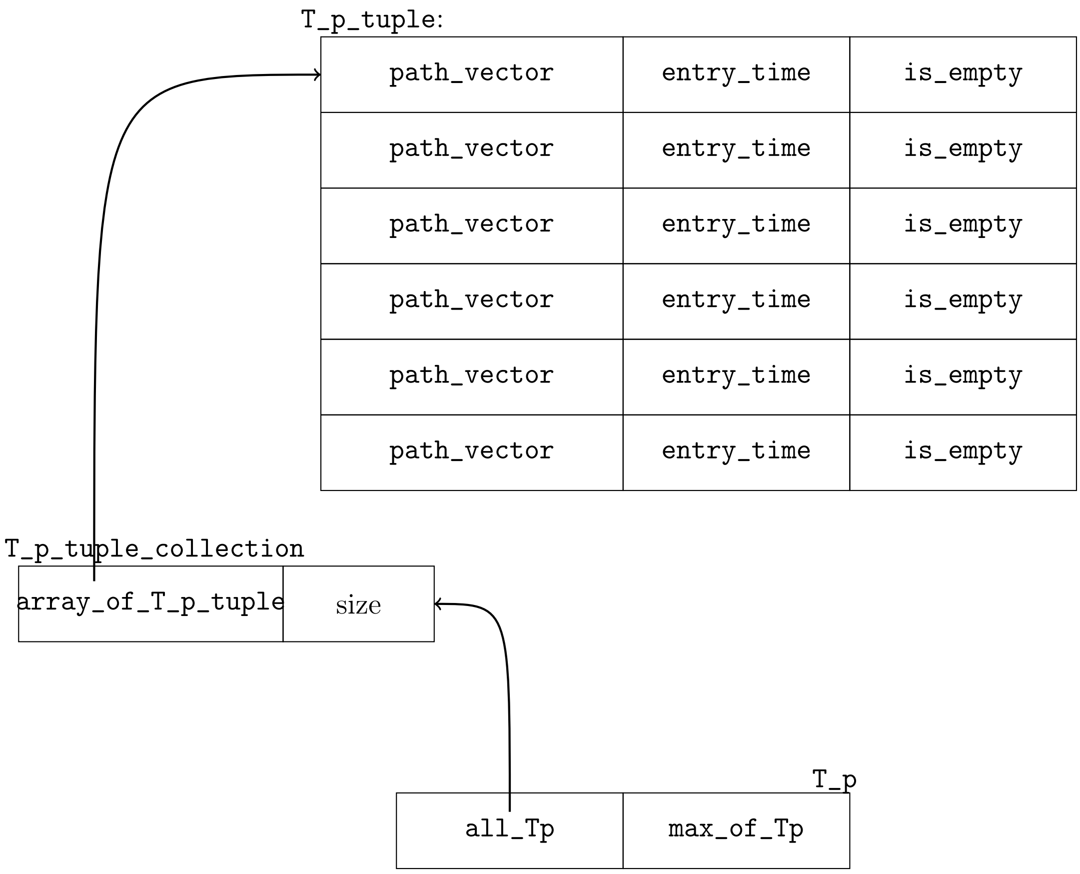

Figure 4: Visual reresentation of the data types defined at the file `Tp.h`.

3.3.2 Functions

alloc_ T_ p

Parameters : `collection_of_basis`.

Objective : Function responsible to allocate all `T_p` structure into memory

Return : `T_p*`.

Setters and getters.

set_ T_ p_ pathdim_ i_ vector_ j

Parameters : `T_p*`, `dim_path`, `vector_index`, `vector`, `double`.

Objective : Allocate the `vector` at position `(T_p->all_Tp)->array_of_T_p_tuple + vector_index` as well it stores an entry time. Whenever this function is called this `vector_index` is set as not empty.

Return : `void`.

is_ T_ p_ pathdim_ i_ vector_ j_ empty

Parameters : `T_p*`, `dim_path`, `vector_index`.

Objective : It checks if `(T_p->all_Tp)->array_of_T_p_tuple + vector_index` is empty

Return : `boolean`.

get_ Tp_ vector_ of_ pathdim_ i_ index_ j

Parameters : `T_p*`, `dim_path`, `vector_index`.

Objective : It returns the `vector` at `(T_p->all_Tp)->array_of_T_p_tuple + vector_index`

Return : `vector`.

get_ Tp_ et_ of_ pathdim_ i_ index_ j

Parameters : `T_p*`, `dim_path`, `vector_index`.

Objective : It returns the entry time of `(T_p->all_Tp)->array_of_T_p_tuple + vector_index`

Return : `double`.

3.4 `persistent_path_homology.h`

This can be considered as the main file inside this folder. It contains, finally, the algorithm we wish to run.

3.4.1 Data Types

The data types in here are the ones responsible to store the intervals of the path persistent homology. All this info will be kept in a list.

First we have the list

- `_Pers_interval_p`. A structure containing the interval of the path persistent homology and a pointer to the next element of the list
- `root`. The root of the list.

Now the structure that has many lists.

- `Pers`. A collection of lists

3.4.2 Functions

First we start by the functions operating with lists. Here the points of the diagrams, the intervals, are stored.

alloc_ Pers

Parameters : `dim_path`

Objective : Function responsible to allocate all path persistent diagrams up to dimension 1

Return : `Per*`.

add_ interval_ of_ pathDim_ p

Parameters : `Pers`, `dim_path`, `double`, `double`.

Objective : Store the path persistent interval of dimension `dim_path` into `Pers`

Return : `void`.

print_ all_ persistent_ diagrams

Parameters : `Pers`

Objective : Print all intervals of all diagrams of persistence

Return : `void`.

Two functions to calculate the allow time and the entry time of any vector:

allow_ time_ vector

Parameters : `double**`, `collection_of_basis`, `vector`, `dim_path`, `dim_vector_space`

Objective : It calculates the allow time of a vector. The vector is an element of the vector space spanned by regular paths of dimension `dim_path`

Return : `double`.

entry_ time_ vector

Parameters : `double**`, `collection_of_basis`, `vector`, `dim_path`, `dim_vector_space`

Objective : It calculates the entry time of a vector. The vector is an element of the vector space spanned by regular paths of dimension `dim_path`

Return : `double`.

Finally, the main functions:

BasisChange

Parameters : `collection_of_basis`, `T_p*`, `double**`, `vector`, `dim_path`, `double*`, `unsigned int*`

Objective : It calculates the gaussian elimination interactively

Return : It returns a `vector` and, by reference, it returns the entry time and the maximum index by the two last pointers on the parameters

ComputePPH

Parameters : `unsigned int`, `double**`, `unsigned int`.

Objective : It calculates the path persistent homology and it returns the respectivelly collection of lists that store the intervals

Return : `Per*`.