# DÉTAILS SUR LA BRANCHE ZIGZAG-FILTRATION-SIMPLEX-TREE DE GUDHI-DEVEL

CLÉMENT MARIA

## 1. Calcul rapide de la star d'un simplexe

L'algorithme suivant est décrit dans l'article sur le Simplex tree. Il part du principe suivant : pour un simplexe s, dont l'étiquette maximale d'un de ses sommets est u, l'ensemble de ses cofaces (s compris) dans le simplex tree est représenté par un ensemble de sous-arbre, dont les racines ont pour étiquette u. L'algorithme consiste à trouver les racines de ces sous-arbre, et à traverser tous leurs noeuds.

Pour l'implémentation :

### 1.1. Relier les nœuds avec même étiquette.

Il faut relier dans une liste $L\_u$ tous les noeuds de l'arbre avec étiquette $u$ (et ce pour toute étiquette $u$) pour une recherche rapide des candidats pour être racine. Cette propriété est maintenue à tout moment dans la construction du Simplex_tree. C'est implémenté en donnant :

- La possibilité de maintenir deux "boost hooks" pour que tous les Node avec la même Simplex_key soient reliés dans une même liste intrusive. L'implémentation se fait par la création de deux classes :

  `Hooks_simplex_base_dummy` (vide), et `Hooks_simplex_base_link_nodes` (contenant un boost intrusive list member hook),

  et d'un std::conditional pour choisir celle que l'on veut (appelée Hooks_simplex_base), et d'un héritage de Node (Simplex_tree_node_explicit_storage.h) depuis Hooks_simplex_base.

  Ce mécanisme est commandé par la nouvelle entrée "static const bool link_nodes_by_label = true" dans les modèles de SimplexTreeOptions.
- Si "link_nodes_by_label == true", il faut également maintenir les listes intrusives. Cela est fait par l'introduction de deux structures :

  nodes_by_label_dummy (empty, pour le cas "link_nodes_by_label == false"), et nodes_by_label_intrusive_list (qui contient une map: étiquette u -¿ pointeur vers une liste intrusive contenant tous les noeuds d'étiquette u).

  Un std::conditional sur Options::link_nodes_by_label permet de choisir quel type on veut, et on stocke une telle structure comme membre de la classe Simplex_tree:

  Nodes_by_label_data_structure nodes_by_label_;

  Au passage, l'utilisation de structure intrusive m'oblige à travailler avec des Node, contrairement à des Simplex_handle (i.e., itérateurs de value_type pair¡Simplex_key,Node¿). J'ai donc du ré-implémenter les fonctions "self_siblings", "dimension", et "has_children" en prenant un Simplex_key et un Node en entrée (à la place du Simplex_handle habituel).

  Finalement, l'insertion et la suppression d'un noeud entraînent des changements plus grand dans le Simplex_tree, notamment le fait que les intrusives listes décrites plus haut sont modifiée. J'aimplémenté ça par l'ajout de deux fonctions membres :

void update_simplex_tree_after_node_insertion(Simplex_handle sh) et void update_simplex_tree_before_
sh),

qui, d'une part, met à jours toutes les structures additionnelles une fois que le simplex a été inséré dans un Siblings (et a désormais un Simplex_handle), et, d'autre part, prépare le complexe à la suppression d'un simplex avant d'être enle vé de son Siblings (et donc avec toujours un Simplex_handle valide). Ces fonctions gèrent les listes intrusives, et seront utiles pour implémenter la théorie de Morse discrète (j'en parlerai plus bas).

b/. J'ai implémenté de nouveaux itérateurs (dans le fichier Simplex_tree/Simplex_tree_star_simplex_iterators.h pour calculer la star d'un simplexe. Précisémment :

class Simplex_tree_optimized_cofaces_rooted_subtrees_simplex_iterator -¿ pour un simplex sigma donné, d'étiquette maximale u, implémente un itérateur parcourant tous les noeuds du simplex tree d'étiquette u (en utilisant les listes intrusives) avec un filtre (filter iterator) pour ne garder que les cofaces de sigma.

class Simplex_tree_optimized_star_simplex_iterator -¿ iterateur sur les cofaces d'un simplex sigma (sigma compris), qui utilise un Simplex_tree_optimized_cofaces_rooted_subtrees_simplex_iterator pour trouver les sous-arbres contenant les cofaces, et parcourt ces sous-arbres.

Dans Simplex_tree.h, il nous faut le bon type pour Star_simplex_iterator. J'ai donc ajouté les typedef :

typedef Simplex_tree_optimized_star_simplex_iterator¡Simplex_tree¿ Optimized_star_simplex_iterator;
typedef boost::iterator_range¡Optimized_star_simplex_iterator¿ Optimized_star_simplex_range;

et un std::conditional sur Options::link_nodes_by_label pour definir le typedef Star_simplex_range

J'ai également un constexpr(Options::link_nodes_by_label) dans

Star_simplex_range star_simplex_range(const Simplex_handle simplex)

car les implémentations sont différentes.

À noter que, pour le moment, cet itérateur calcule toutes les cofaces (la star) et pas celles d'une dimension donnée.

2/. Construction de filtration zigzag de flag complexes (partie combinatoire).

Étant donnée un range d'insertions et de suppressions de sommets et d'arêtes (i.e., une filtration zigzag du 1-squelette), faire les expansions et suppressions appropriées dans le Simplex_tree à la volée.

Modifications du code :

- possibilité d'utiliser une boost::container::map pour stocker les Node d'un Sibling, à la place d'une flat_map qui est moins dynamique (on fait beaucoup d'insertions et de suppressions de simplexes dans une filtration zigzag). De plus, mon implémentation des filtrations zigzag requiert que les Simplex_handle (ici des map::iterator donc) restent valide pendant toute la période où un simplex est dans le complexe (ce n'est pas le cas dans les flat_map qui "bougent" en mémoire).

Cette possibilité est donnée par la nouvelle entrée "static const bool simplex_handle_strong_validity = true;" dans les modèles de SimplexTreeOptions, et par un std::conditional pour le "typedef Dictionary".

- nouvelles fonctions membres :

a/. flag_add_edge -¿ permet d'ajouter un sommet ou une arête, et de faire l'expansion du complexe jusqu'à la dimension maximale autorisée. Tous les nouveaux simplexes sont enregistrés dans un vecteur. Le Simplex_tree est modifié.

flag_add_edge dépend de plusieurs fonctions "zz_punctual_expansion, zz_local_expansion, zz_siblings_expansion", qui implémentent différents cas d'expansion locale, et zz_intersection qui fait des intersections de Siblings.

b/. flag_lazy_remove_edge -¿ enregistre (sans modifier le Simplex_tree) toutes les cofaces d'un sommet ou d'une arête qu'on veut supprimer.

flag_lazy_empty_complex -¿ la même chose, mais pour tout supprimer.

c/. De nouveaux itérateurs de filtration zigzag dans le fichier Simplex_tree/Simplex_tree_zigzag_iterators.h

Dans Simplex_tree.h, cela se traduit par l'introduction d'un type "Zigzag_filtration_simplex_iterator" et "Zigzag_filtration_simplex_range", et "Filtration_simplex_range" est choisi en fonction d'un std::conditional sur Options::is_zigzag (nouvelle option).

L'interface pour accéder à la filtration est toujours avec

filtration_simplex_range(...)

mais dépend désormais d'un linear_indexing_tag (filtration standard) ou d'un zigzag_indexing_tag.

Remarque : linear_indexing_tag/zigzag_indexing_tag et bool is_zigzag sont redondants dans les Options. Je propose d'utiliser un constexpr(Options::is_zigzag) (que je trouve plus lisible) pour filtration_simplex_range (et les autres fonctions où linear_indexing_tag est utilisé), et de supprimer ces indexing_tag. Qu'en pensez-vous ? Certaines fonctions, comme "Simplex_handle simplex(Simplex_key idx)", qui n'ont pas de version en persistence zigzag, pourraient utiliser un "std::false_type" en input, auquel on appelerait "simplex(key, Options::is_zigzag)". Je pense que ça déclenchera toujours une erreur à la compilation ?

3/. La partie géométrique de la construction de filtration zigzag : les oscillating Rips

4/. La théorie de Morse discrète :

J'ai implémenté une bonne partie des mécanismes, mais il me manque le calcul de bord dans un complexe de Morse. Je décris les mécanismes :

La volonté de faire la théorie de Morse discrète est donnée par Options::store_morse_matching.

- la possibilité de stocker un Simplex_handle dans chaque Node, pour représenter le simplex avec lequel on forme une paire de Morse. Implémenté par les deux classes

Pairing_simplex_base_dummy (vide) et Pairing_simplex_base_morse (contient un membre Simplex_handle) dont Simplex_tree_node_explicit_storage hérite.

- des méthodes "paired_with, critical, assign_pairing, make_critical, is_paired_with" dans la classe Simpex_tree pour atteindre l'information au sujet des paires de Morse.

- un appel à "Discrete_morse_theory¡Simplex_tree¿().compute_matching" quand on insère un ensemble de simplexe avec la même valeur de filtration (par exemple dans flag_add_edge). Cette méthode calcul un appariement de Morse (c'est dans le paquetage Discrete_morse_theory) et stocke l'information dans les noeuds du simplex tree.

- une mise à jour des noeuds dans les fonctions :

void update_simplex_tree_after_node_insertion(Simplex_handle sh) -¿ un nouveau simplex est critique par défaut,

void update_simplex_tree_before_node_removal(Simplex_handle sh) -¿ en enlevant un simplex non-critique, il faut rendre le simplex, avec lequel il forme une paire de Morse, critique.

Il y a également de nouveaux fichiers :

Point_cloud.h implémente des méthodes simples sur les nuages de points : chercher et éliminer les doublons, perturbations aléatoires, vérifier que deux paires de points (distincts) x,y,z,t ne soient pas à la même distance, i.e., d(x,y) != d(z,t).

Cette dernière méthode peut paraître étrange, mais en réalité elle est utile pour qu'un "farthest point sampling" d'un nuage de points soit unique une fois qu'on a choisi le premier point. (utilisé dans ma construction de l'oscillating Rips zigzag filtration).

src/Zigzag_persistence/ avec le fichier Zigzag_persistence.h Implémente la persistence zigzag.

data/points/S3_in_R4_2k_uniform.off Un échantillonage uniform de 2000 points sur la 3-sphère unité dans R4. C'est un cas intéressant pour la persistence zigzag qui permet de reconstruire et d'inférer l'homologie de 3-variétés.