


Université de Montpellier

Master de Bioinformatique

Projet d'Assemblage

Auteurs :
Mickael Coquerelle & Loik Galtier & Raphael Ribes

Professeure :
Annie Chateau

Contents

1 Etape de conception

1.1 Stratégies vues en cours *(Exercice 1)*

Nous avons vu en cours que la recherche d'une superchaîne optimale contenant tous les mots d'une famille $\mathcal{F}$ est un problème NP-difficile. Par conséquent, nous utilisons des approches heuristiques : celles-ci fournissent une solution, mais sans garantie d'optimalité. Trois méthodes ont été étudiées et développées durant le semestre. Comme demandé, voici une brève synthèse de notre compréhension de ces dernières.

1.1.1 Approche intuitive : la méthode gloutonne

Cette première approche repose sur les choix locaux optimaux (maximisation des *overlaps*), mais sans assurance d'optimalité globale. On va comparer nos mots deux à deux pour identifier leurs chevauchements. Sélectionner le plus long et fusionner les deux mots concernés (créer des contigs). On répéte ainsi l'opération jusqu'à obtenir la superchaîne $\mathcal{S}$. On stock nos *overlaps* dans une matrice $\mathcal{M}$ de nxn séquences. La stratégie construite en TD est en annexe (algo glouton).

L'algorithme glouton, bien que rapide avec une complexité de $\mathcal{O}(n^3+n^2L)$, est une heuristique purement locale pose un soucis dans son manque de garantie d'optimalité globale. Pour atténuer cette limitation, nous explorons une approche plus systématique basée sur les graphes de chevauchement.

1.1.2 Approche par graphe de chevauchement

Cette seconde manière d'appréhender notre problème repose sur une modélisation globale des relations entre les mots. Cela va nous permettre qui mieux explorer les chevauchements et d'éviter "les pièges des choix" locaux. Avant de rechercher notre super séquence, il est nécessaire de construire le graphe. Chaque sommet du graphe représente un mot de $\mathcal{F}$, et chaque arête est pondérée par la longueur de *l'overlap*(algo 4 ANNEXE)

Pour cet algorithme (inspiré fortement de la correction du TD), l'initialisation des sommets et des arêtes est immédiate (évidemment en temps constant $\mathcal{O}(1)$). Ensuite, pour chaque paire de mots $(F[i], F[j])$, l'algorithme calcule la longueur du chevauchement maximal k. Cette opération est effectuée en $\mathcal{O}(L)$, où L est notre longueur moyenne des mots (comme pour le glouton). Si un chevauchement est détecté ($k > 0$), une arête pondérée est ajoutée entre les sommets correspondants. Cette stratégie suggère une complexité temporelle globale de $\mathcal{O}(n^2L)$, car on doit parcourir toutes nos paires de mots, et chaque calcul de chevauchement est en $\mathcal{O}(L)$. La complexité spatiale est $\mathcal{O}(n^2)$, car on stocke au plus n^2 arêtes dans E. A titre d'exemple on aura pour des mots, si $F = \{$"BANANE", "ANE", "NEANT"$\}$, le graphe de chevauchement G contiendra les arêtes suivantes : ("BANANE", "ANE", 3), ("ANE", "NEANT", 2), et ("BANANE", "NEANT", 2).

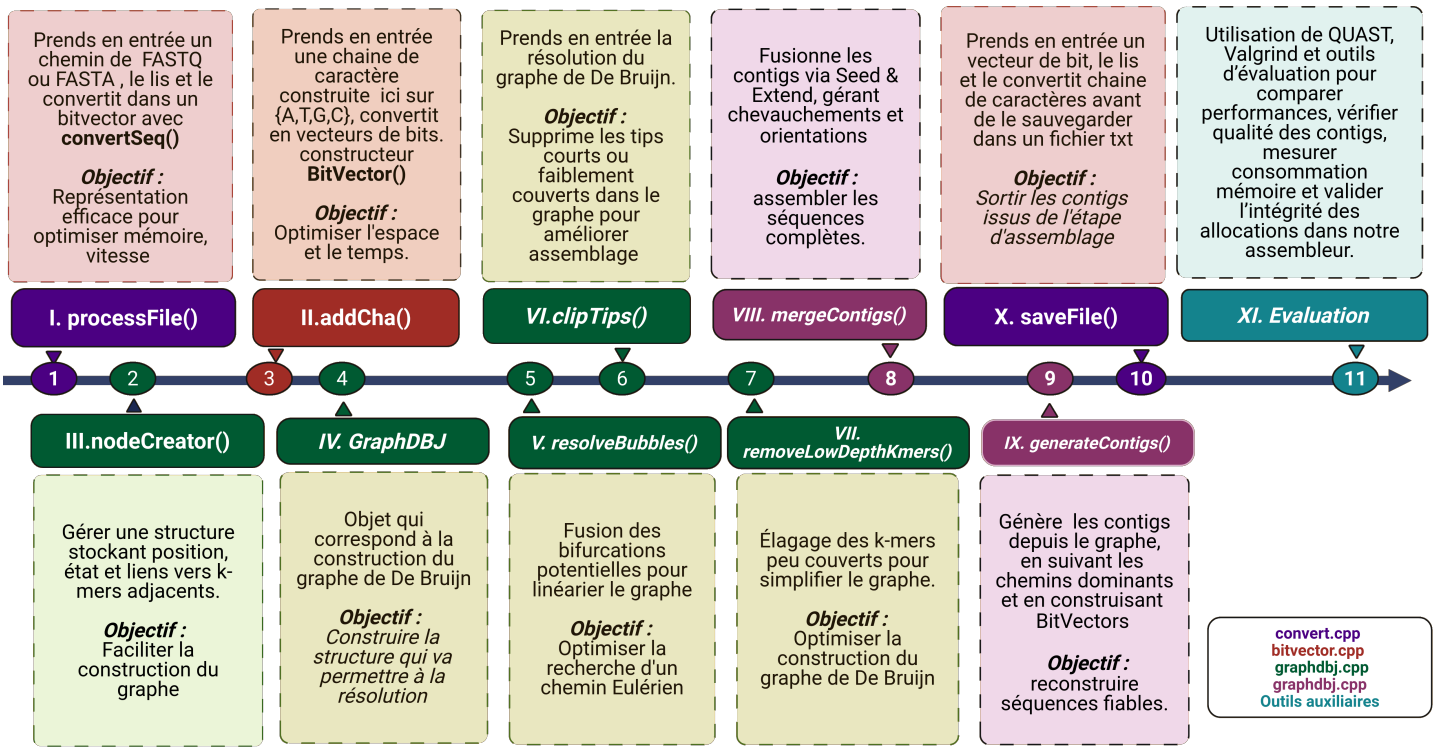
Ceci étant fait, l'étape suivante consiste à chercher notre chemin qui va maximiser la somme des pondérations de nos arêtes. On aura ici l'avantage (à l'inverse de glouton) de considérer l'entièreté des relations possibles entre les mots, ce qui, *à priori*, améliore la qualité de notre solution $\mathcal{S}$. (ALGO 5 ANNEXE)

Cet algorithme recherche un chemin maximal dans le graphe de chevauchement pour construire une superchaîne. Il commence par identifier un sommet de départ, généralement celui avec le moins d'arêtes entrantes, afin de minimiser les risques de cycles. Ensuite, il construit un chemin en choisissant à chaque étape l'arête de poids maximal vers un sommet non visité. Cette approche permet d'explorer les chevauchements de manière plus globale que la méthode gloutonne. La complexité temporelle est $\mathcal{O}(n^2L + n^3)$, car la recherche des arêtes de poids maximal peut nécessiter jusqu'à $\mathcal{O}(n^3)$ opérations dans le pire cas. La complexité spatiale reste $\mathcal{O}(n^2)$ pour stocker le graphe.

1.1.3 Notre choix le graphe de De Bruijn : Avantages & inconvénients (Exercice 2 & 5)

Durant les enseignements, nous avons explorés plusieurs "paradigmes" (Gloutonne, OLC & graphe de De Bruijn), après discussions et analyse des avantages et inconvénients de chaque méthode, nous avons retenu le graphe de De Bruijn pour sa capacité à représenter les relations entre kmers dans de (très) grandes collections de *short reads*, tout en réduisant la complexité liée à l'identification de nos chevauchements exacts [2, 3, 4, 5].

1.2 Explication des grandes étapes *(Exercice 2)*



1.2.1 Le langage choisi :

Il existe de nombreux langages de programmation, tels que le Python, le Go ou encore le Pascal. Mais pour notre projet, le choix s'est tourné vers le C++. Bien sûr, Python aurait été une possibilité, moins performant mais plus facile à intégrer et à manipuler. Dans le cadre d'un assembleur, qui peut gérer des fichiers imposants et complexes, un langage efficace est nécessaire. Le C++ est l'un des langages les plus performants après le C, qui permet un contrôle total sur la mémoire et est particulièrement efficace et direct.

1.2.2 Le module `bitvector.cpp`

Comment allons-nous représenter et stocker nos lectures ? Considérant que les technologies modernes de séquençage posent, par le volume de données, des contraintes importantes en mémoire et en performance lors de l'assemblage, il nous a paru pertinent d'utiliser des `BitVectors`. Parmi les avantages de cette structure, on trouve la "compacité" mémoire, i.e. au regard de notre alphabet $\Sigma = \{A, T, G, C\}$, une de nos séquences $s = s_1 s_2 \dots s_n$, avec $s_i \in \Sigma$ dans notre `FASTA`, peut être codée sur 2 bits par nucléotide via une fonction d'encodage, :

$$\text{encodage} : \Sigma \to \{00, 01, 10, 11\} \; ici \; |\Sigma| = 4 \quad \text{et} \quad \log_2 |\Sigma| = 2. \mid b = \text{enc}(s_1)\,\text{enc}(s_2)\,\dots\,\text{enc}(s_n)$$

Nous avons vu durant les enseignements (DevOps, Algorithmique du texte, ...), que la représentation classique de s est d'un octet par nucléotide (nt). Cela représente un stockage théorique en mémoire de 8n bits. Notons qu'un `BitVector` quant à lui repose sur l'utilisation d'un tableau de blocs de taille fixe (typiquement des entiers non signés de 64 bits). Chaque élément encode une portion de s, et la classe garantit que l'écriture comme la lecture se font en temps constant $\mathcal{O}_{(1)}$ grâce à des opérations bit-à-bit (*bitwise*). Cette approche réduit significativement l'empreinte mémoire : là où une chaîne classique occupe $8n$ bits pour n nucléotides. Notre représentation compacte utilise $2n$ bits, auxquels nous le verrons, on ajoute un surcoût négligeable pour les opérations de masquage et de décalage nécessaires à la manipulation interne des blocs. Ainsi , on peut stocker 4 fois plus de donnée :

$$\text{Gain} = \frac{8n}{2n} = 4,00 = 400\%$$

Une réduction d'un facteur 4 de la mémoire utilisée pour stocker nos lectures, c'est particulièrement pertinent dans un contexte où des millions de lectures doivent être simultanément maintenues en mémoire, notamment lors de la construction d'un graphe de De Bruijn dense.

De plus, l'utilisation d'une seul variable permet de diminuer la quantité de flag mémoire utilisé, diminuant davantage la quantité de mémoire utilisé.

D'un point de vue pratique, la gestion du stockage relève de la classe `bitvector`, qui implémente trois constructeurs. Le premier, permet d'instancier la structure minimale. Le second, `bitvector(sizeElement)`, autorise la définition explicite de la granularité, en fixant la taille de chaque élément dans les blocs de bits. Enfin, le constructeur `bitvector(sizeElement, dataSize)` préalloue l'espace nécessaire pour un nombre donné de bits ; cette stratégie optimise les performances lorsque la taille finale est connue à l'avance, en limitant les réallocations successives.

Du reste, nous avons implémenter un certain nombres de fonctions pour alimenter les bitvectors et maintenir leurs cohérence interne. La fonction `reserve()`, anticipe l'allocation en calculant le nombre de blocs nécessaires. la fonction `push_back()` insère un bit à la fin du vecteur (en allouant un nouveau bloc si nécessaire). On à également des fonctions utilitaires comme `clear()` nous permet de réinitialiser proprement l'objet ou `size()` pour avoir accès au nombre de bits stockés (pour borné les boucles de parcours). Enfin, `setSizeElement()` et `getSizeElement()` gèrent la "granularité" des éléments empaquetés dans les blocs. Sur cette premier partie de la classe nous avons le "coeur mécanique" du `BitVector`. On trouve dans la classe également des fonctions interactives comme la conversion des bits gérée par `to_vector()`, les mécanismes d'initialisations tels que `createListBit()`, `addCha()`. Quant à la lecture d'un bit à un index donné est effectué par `readBitVector()`. Après avoir présenté le `BitVector` intéressons nous à la classe `graphdbj`.

1.2.3 La classe `convert.cpp`

La classe `convert` exploite les fonctions de `bitvector` pour lire les fichiers `FASTA/Q`, encoder les séquences en bits et enregistrer les positions de fin de lecture. La fonction principale est `processFile()`:

Algorithm 1: processFile(filename)

Entrée: Fichier FASTA `filename`
Sortie: `bitVector` rempli ; `endPos` mis à jour
Complexité: Temps : $\mathcal{O}(n)$, où n est le nombre total de caractères du fichier

Début Ouvrir le fichier `filename`
Si *le fichier ne peut pas être ouvert* **Alors**
Retourner Erreur
FinSi
Reinitaliser `bitVector` et `endPos` ($\mathcal{O}(1)$)
totReadSize ← 0
totReadNum ← 0
cSeq ← ""
bitVector`.reserve(totReadSize * 2)` (réservation initiale)
endPos`.reserve(totReadNum)` (capacité vide)
TantQue ligne $\mathcal{O}(n)$
Si *ligne vide* **Alors**
Continuer
Si *ligne[0]* = *">"* **Alors**
totReadNum ← totReadNum + 1
Si *cSeq non vide* **Alors**
convertSeq(cSeq) ($\mathcal{O}(L)$)
cSeq ← ""
FinSi
FinSi
Sinon
Supprimer les " " ($\mathcal{O}(L)$)*
totReadSize ← totReadSize + longueur(ligne)
cSeq ← cSeq + ligne ($\mathcal{O}(L)$)
FinSi
FinTantQue
Si *cSeq non vide* **Alors**
: `convertSeq`(cSeq) ($\mathcal{O}(L)$)
FinSi
Fermer le fichier
Fin

L'approche que nous avons retenue privilégie un parcours linéaire, permettant un usage minimal de mémoire et une conversion différée des séquences via le tampon `cSeq`. Remarquons, que nos lignes ne commençant pas par '>' sont logiquement considérées comme des (fragments) de séquence. Les espaces sont éliminés, puis la longueur totale est mise à jour et la ligne est concaténée au tampon `cSeq`. Ces opérations sont linéaires en la taille des

lignes. Si une séquence reste non traitée à la fin du fichier (cas le plus courant en `FASTA`), elle est convertie une ultime fois avant que le fichier ne soit fermé. La fonction `convertSeq` présentée dans l'algorithme ci-dessous permet de passer chaque caractères en bits:

Algorithm 2: convertSeq(sequence)

Entrée: Séquence nucléotidique `sequence` de longueur L
Sortie: Bits ajoutés à `bitVector` ; fin de lecture ajoutée à `endPos`
Complexité: Temps : $\mathcal{O}(L)$

Début
Si *sequence vide* **Alors Retourner**
Pour *chaque nucléotide c dans* `sequence` $(\mathcal{O}(L))$
`bitVector`.addCha(c)
FinPour
Ajouter `bitVector`.size() à `endPos`
Fin

La complexité temporelle totale est en $\mathcal{O}(n)$, où n est le nombre de caractères du fichier. Chaque caractère est lu et traité une seule fois, les réservations sont amorties en temps constant, aucune opération quadraqtique n'est nécessaire. Cette propriété fait de `processFile` un parseur `FASTA` intéressant pour des fichiers de grande taille.

1.2.4 La classe `graphdbj.cpp`

La construction (dans les grandes lignes) :

Dans cette partie, nous revenons sur les choix conceptuels de notre assembleur, l'objectif étant de répondre, nous l'espérons, au plus près des attendus du TP. Les différentes classes décrites s'articulent autour de la classe `GraphDBJ`, qui constitue le coeur du programme : création du graphe, simplification et recherche de chemin(s). Le constructeur du *graphe de De Bruijn*, dans sa version synthétique :

Algorithm 3: constructGraph(converter, k)

Entrée: Objet `Convert` contenant les lectures ; taille des k-mers `k`
Sortie: Graphe de De Bruijn avec noeuds et arêtes
Complexité: Temps : $\mathcal{O}(N \cdot k)$, où N est la somme des longueurs des lectures

Début
Si $2 \leq k \leq 32$ **Sinon**
Retourner Erreur $(\mathcal{O}(1))$
bv ← `converter`.getBitVector()
read_ends ← `converter`.getEndPos()
current_read_start ← 0
Pour end_pos **Dans** read_ends
read_len ← $(end\_pos - current\_read\_start)/2$
Si $read\_len \geq k$ **Alors**
Pour i = 0 **allant de** read_len - k
u_fwd ← `extractKmerValue`(bv, current_read_start + 2*i, k-1)
v_fwd ← `extractKmerValue`(bv, current_read_start + 2*i + 2, k-1)
u_rev ← `getReverseComplement`(u_fwd, k-1)
v_rev ← `getReverseComplement`(v_fwd, k-1)
Ajouter `arête`(u_fwd → v_fwd) , `arête`(v_rev → u_rev) $(\mathcal{O}(1)$ amorti$)$
current_read_start ← end_pos
Fin

Le choix de représenter les noeuds comme des $(k-1)$-mers et les arêtes comme des transitions de taille $k+1$ s'appuie sur la formalisation du graphe étudiée en cours. Cette approche facilite également la détection d'erreurs structurées (tips, bulles), contrairement à l'approche par *overlap* (cf. annexe), plus coûteuse. Dans l'algorithme `constructGraph`, on effectue une conversion des k-mers en valeurs entières via `extractKmerValue()`. Cela a été l'une des premières difficultés rencontrées, notamment pour éviter les « débordements » et garantir la cohérence entre les brins *forward* et *reverse*. Dans la deuxième structure itérative, on génère simultanément le complément inverse en ajoutant les deux arêtes, comme discuté lors de la dernière séance de TP. Cela permet d'obtenir un graphe navigable quel que soit le sens de lecture. Cet aspect est fondamental pour de vrais jeux de données. Ensuite, les noeuds et les arêtes sont exportés en GFA, permettant de représenter le graphe de contigs assemblés avec leur couverture (que nous exploitons dans les outils auxiliaires nous le verrons). Ces explications générales étant posées, nous vous proposons de rdétailler dans la suite de cette section certains aspects qui nous paraissent pertinents.

Simplification du Graphe :

En générant nos contigs, nous nous sommes aperçus d'un certain nombre de problèmes à gérer pour obtenir des résultats réalistes, tant en quantité qu'en qualité des contigs. Dans ce sens, nous avons implémenté plusieurs

méthodes pour nettoyer le graphe afin d'éliminer les éléments qui nous paraissaient, *a priori*, artefactuels. Cette étape (appliquée dans le `main`) est orchestrée séquentiellement par deux fonctions de la classe `GraphDBJ`, jusqu'à stabilisation ou jusqu'à l'atteinte du nombre maximal de passes `max_passes_pop`.

Le premier problème qui nous parait essentiel à aborder est la **suppression des *Tips***. Cela revient à dire : exclure les chemins terminaux courts dans le graphe. Ils sont générés, d'après la littérature, majoritairement par des erreurs de séquençage. Pour chaque noeud terminal $v \in V$, on remonte le chemin $\mathcal{T} = (v_1, v_2, \ldots, v_n)$ jusqu'au noeud d'ancrage a, où se produit une bifurcation ou jusqu'à un un premier seuil de longueur matérialiser dans le code par $n \leq$ `TOPO_MAX_LEN`. Les *tips* de cette taille seront alors considéré comme des erreur et déconnectés. Viens ensuite une phase de "rédemption" ou les *tips* en dessous du seuil $n \leq$ `RCTC_MAX_LEN`, pour **Ratio de Couverture de Tip-To-Core** (RCTC) et avec une couverture moyenne inférieure à celle du k-mer d'ancrage sont déconnecté. Cette approche élégante est tirée de l'algorithme de `minia`[1], nous avons décidé de l'implémenter dans notre algorithme. Plus formellement, on calcule alors la "couverture moyenne du tip", pour ensuite la comparer avec la couverture du nœud d'ancrage Coverage(a). Nous avons choisi de supprimé le tip si [1]:

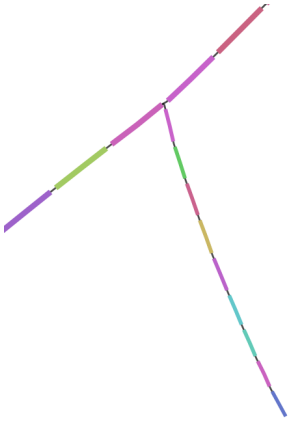


Figure 1::
Illustration d'un tip dans le graphe d'assemblage.

$$n \leq \texttt{TOPO\_MAX\_LEN et } (n \leq \texttt{RTC\_MAX\_LEN} \quad \&\& \quad \text{Coverage}(a) > \bar{c}_T) \mid \bar{c}_T = \frac{1}{n}\sum_{i=1}^{n} \text{Coverage}(v_i)$$

En pratique, on utilise `disconnectNodes(a, v)` pour couper la branche et marque tous les noeuds du tip comme `should_removed = true`. Ce choix permet de retirer les chemins artefactuels tout en préservant les structures linéaires réelles, plus longues et mieux couvertes. Le second problème rencontré dans les graphes de *De Bruijn* est celui des bulles.

Figure 2:: Illustration d'une bulle avec `Bandage`

Elles vont apparaître lorsqu'un nœud source s se divise en deux chemins parallèles avant reconverger vers un nœud d'arrivée t . Le motif résultant dans sa grade majorité à des variations biologiques et/ou erreur de séquençage. Si elles ne sont pas résolue elles entraînent une fragmentation artificielle des *contigs*. D'un point de vue formel, pour notre $G = (V, E)$, une bulle est définie par deux chemins : Soient $s, t \in V$. Considérons l'ensemble $\mathcal{P}(s, t)$ des chemins simples allant de s à t. Nous nous intéressons à deux éléments distincts $\mathcal{P}_1, \mathcal{P}_2 \in \mathcal{P}(s, t)$ vérifiant $\mathcal{P}_1 \neq \mathcal{P}_2$ et $\mathcal{P}_1 \cap \mathcal{P}_2 = \{s, t\}$.

On dois définir également la couverture cumulée $\mathcal{C}$ du chemin $\mathcal{P}$, on aura ainsi une résolution de la bulle en sélectionnant $\mathcal{P}_{\max}$ et à supprimer l'autre chemin :

$$C(\mathcal{P}) = \sum_{v \in \mathcal{P}} \text{Coverage}_v \; ; \quad \mathcal{P}_{\max} = \arg \max_{\mathcal{P} \in \{\mathcal{P}_1, \mathcal{P}_2\}} C(\mathcal{P}) \; ; \quad \{\mathcal{P}_1, \mathcal{P}_2\} \setminus \{\mathcal{P}_{\max}\}$$

La méthode `resolveBubbles()` fait ce travail de détection pour chacun des noeuds possèdants deux voisins qui finissent par converger vers un noeud commun et la profondeur $\mathcal{P}$est limitée par `SEARCH_DEPTH_FACTOR`). L'algorithme compare $\mathcal{C}$ de $\mathcal{P}_1$et$\mathcal{P}_2$, la branche la moins couverte est marquée à travers `removed = true`, puis `disconnectNodes()`.

Génération des Contigs Bruts :

Après avoir construit notre graphe et effectué les étapes de simplification (*Tip, bulles*,suppression de nos chemins artefactuels), il nous reste à générer nos contigs, i.e les chemins linéaires maximaux P ou aucune "ambiguité topologique" n'est présente. Formellement pour tout noeuds intermédiaires i, de degrès entrant $\deg^-$ et sortant $\deg^+$:

$$\mathcal{P} = (v_1, v_2, \ldots, v_m) \mid \forall\, i \in \{2, \ldots, m-1\},\ \deg^-(v_i) = \deg^+(v_i) = 1.$$

Finalement si on se rapporte à ce que l'on à vu en cours, on est en droit de dire que ces chemins sont des zones du génome ou l'assemblage est déterministe. Le challenge algorithmique se réparti en trois aspects : détecter les bons points de départ, prolonger le chemin tant que la structure reste univoque et enfin reconstruire la séquence nucléotidique à partir de nos $(k-1)$-mers encodés dans les noeuds de G. Pour construire cela on considère que chaque noeud v est un point de départ du contig si : $\deg^-(v) \neq 1 \quad$ ou $\quad \deg^+(v) \neq 1$ en imposant $\deg^+(v) \geq 1$. À partir d'un tel nœud v_0, on étend le chemin tant que : $\deg^-(v_i) = \deg^+(v_i) = 1$. Le contig sera maximal quand on atteint un noeud v_m tel que $\deg^+(v_m) \neq 1$. Considérant ces règles, il nous faut reconstruire la séquence en concaténant le $(k-1)$mer du premier noeud, puis à chaque arc $(v_i \to v_{i+1})$, le dernier nucléotide du $(k-1)$-mer de v_{i+1}. Ainsi, pour un chemin de longueur m, la longueur du contig est logiquement : $(k-1)+(m-1)$.

L'ensemble de cette logique est implémentée dans la méthode `generateContigs()`, dont nous vous proposons l'algorithme avec uniquement les grandes étapes de la fonction, à la page suivante. Il parcourt l'ensemble des noeuds du graphe simplifié (plus de *tips,bubbles*), pour déterminer ceux qui peuvent initier un contig. Pour chaque noeud candidat `startNode`, on vérifie qu'il n'a pas déjà été visité (`visited`), pour éviter des contigs dupliqués. On test ensuite notre condition topologique $\deg^-(v) \neq 1 \quad$ ou $\quad \deg^+(v) \neq 1$, ce qui va correspondre à soit un noeud de départ potentiel, soit un noeud sans parent, soit une bifurcation (réelle car restante). Ensuite on initialise notre contig avec le $(k-1)$-mer du nœud de départ avec (`addKmerToBitVector()`). Puis on fait un parcours linéaire du graphe en suivant les arêtes sortantes tant que chaque noeud intermédiaire satisfait un chemin simple, $\deg^- = \deg^+ = 1$. En cas de bifurcation, on à fait un choix heuristique basé sur la couverture (`COVERAGE_RATIO`), en s'inspirant du fonctionnement de `minia` et de notre stratégie de résolution des *bubbles*, pour prolonger le contig uniquement si un chemin dominant se dégage. Enfin, La reconstruction de la séquence repose sur le codage des $(k-1)$-mers,fonction `kmerToString()`. Lors du parcours, seul le dernier nucléotide du nœud suivant est ajouté au contig, conformément à la définition du graphe de De Bruijn vu en cous et à la logique d'extension décrite dans le pseudocode. Cette stratégie évite toute redondance (*à priori*). Pour finir on encapsule notre contig dans une structure qui contient: la séquence complète, la couverture moyenne du chemin, la longueur totale et un identifiant unique:

Algorithm 4: generateContigs() (Simplifié)

Entrée: Graphe de $G = (V, E)$, k, COVERAGE_RATIO, MAX_CONTIG_LEN
Sortie: Liste de contigs en BitVector.
Complexité: Temps : $\mathcal{O}(|V| + \sum_{v \in V} \deg^+(v))$

```
Début
    contigs ← liste vide
    visited ← Dictionnaire pour tous les noeuds
    Pour chaque startNode Dans V
        Si startNode.removed ou visited[startNode] Alors
            Continuer
        FinSi
        Si startNode.parents.empty() ou startNode.parents.size() > 1 Alors
            currentContig ← BitVector vide
            addKmerToBitVector(currentContig, startNode.p, k-1)
            visited[startNode] ← true
            curr ← startNode
            TantQue sanity_check < MAX_CONTIG_LEN
                next ← Choisir enfant unique ou (dominant selon COVERAGE_RATIO)
                [...]
                Si next = null ou next.removed ou visited[next] Alors
                    Sortir
                FinSi
                Ajouter Dernier nucléotide de next à currentContig
                visited[next] ← true
                curr ← next
            Ajouter currentContig à contigs
            FinTantQue
        FinSi
    FinPour
Fin
```

Fusion des contigs :

Nous avons constatés que malgré tout ce travail de simplification, les contigs obtenus avec `generateContigs()`, restent très "fragmentés". A l'aide de *bandage* et après discussions, nous pensons que ce problème vient en grande partie d'une orientation inversé de certains contigs (*reverse complément*). Mais aussi à des chevauchements partiels non détectés lorsque nous parcourons le graphe. La fonction `mergeContigs()` sert à "recoudre" ces derniers pour produire des séquences plus longues et cohérentes en prenant garde que l'on ne perde pas d'information. Simplement, on peut distinguer deux grandes phases. La phase d'inclusion (*Containment*) qui élimine les contigs totalements inclus dans d'autres, et la phase d'extension (*seed & extend*) qui cherche à fusionner les contigs restants en fusionnant les chevauchements internes et en gérant l'orientation. Soit notre ensemble de contigs C tel que : $\mathcal{C} = \{C_1, \ldots, C_n\}$ et k la taille du k-mer utilisé pour l'indexation.

Dans la phase d'inclusion (*Containment*), chaque contig C_j est comparé à tous les autres contigs C_i $(i \neq j)$ pour

détecter s'il est entièrement contenu dans un contig plus long, on marque C_j comme absorbé si :

$$\text{absorbed}[j] = \text{true} \iff \exists i \neq j,\ C_j \subseteq C_i \text{ (avec une tolérance d'erreur } \epsilon_{\text{contain}})$$

Ce qui est intéressant ici, c'est que tous les nucléotides de C_j correspondent, à une petite marge d'erreur près, à une sous-séquence d'un autre contig C_i. Dans ce cas que, C_j est considéré redondant et n'est plus conservé pour la phase suivante. Dans la phase d'extension (*Seed & Extend*), pour chaque contig maître C_m, on recherche un contig candidat C_c possédant un k-mer *seed* s qui correspond à une sous-séquence de C_m :

$$\exists s \in C_c : s = C_m[\text{pos} : \text{pos} + k - 1] \implies C_m \leftarrow C_m[0 : \text{align\_start} - 1] \oplus C_c$$

où $\oplus$ représente la concaténation après redimensionnement du contig maître pour remplacer la "pointe" éventuellement erronée par la séquence propre du candidat. Le contig candidat C_c est alors marqué comme absorbé. Cette recherche est effectuée à la fois sur C_m dans le sens direct et sur son complément inverse C_m^{RC} afin de gérer les contigs qui auraient été assemblés dans la direction opposée.

Ces contigs sont ensuite exploités par la fonction `exportToGFA()` (que nous ne détaillons pas ici, car nous devons faire des choix) qui produit un fichier GFA conforme où chaque contig apparaît comme un segment annoté par sa couverture. Cette représentation permet de visualiser efficacement l'assemblage, notamment avec des outils tels que `Bandage`.

1.3 Outils auxiliaires *(Exercice 7)*

`Quast` (QUality ASsessment Tool), dans un premier temps, nous permet d'évaluer la qualité d'un assemblage à l'aide d'un ensemble de statistiques standardisées (longueur totale, taux d'erreurs, fragments mal alignés, etc.). Bien que l'outil puisse fonctionner en mode de novo, sans référence, nous avons choisi d'utiliser ici la séquence de référence fournie pour la mitochondrie du varan de Komodo. Ce choix garantit une évaluation plus robuste en ancrant nos comparaisons sur un génome connu, ce qui maximise la pertinence

Dans un second temps, nous avons exécuté `minia`, un autre algorithme d'assemblage à base de graphe de Bruijn, afin d'obtenir un assemblage alternatif. Cet assemblage sert de point de comparaison directe avec celui obtenu précédemment.

Enfin, nous avons utilisé `D-genies`, qui génère un dot-plot visualisant le degré d'identité entre deux séquences. Cet outil complète l'analyse en offrant une représentation intuitive de la synténie, des éventuelles réarrangements, et des discordances locales entre notre assemblage principal et celui obtenu avec `minia`. L'approche par dot-plot complète notre stratégie puisqu'elle combine des métriques numériques (via Quast) et une inspection visuelle qualitative (via D-genies), ce qui nous à permis de valider nos choix d'implémentations après avoir identifier de grande divergences dans la nature et le nombre de contigs obtenu (dans nos premières exécutions du programme).

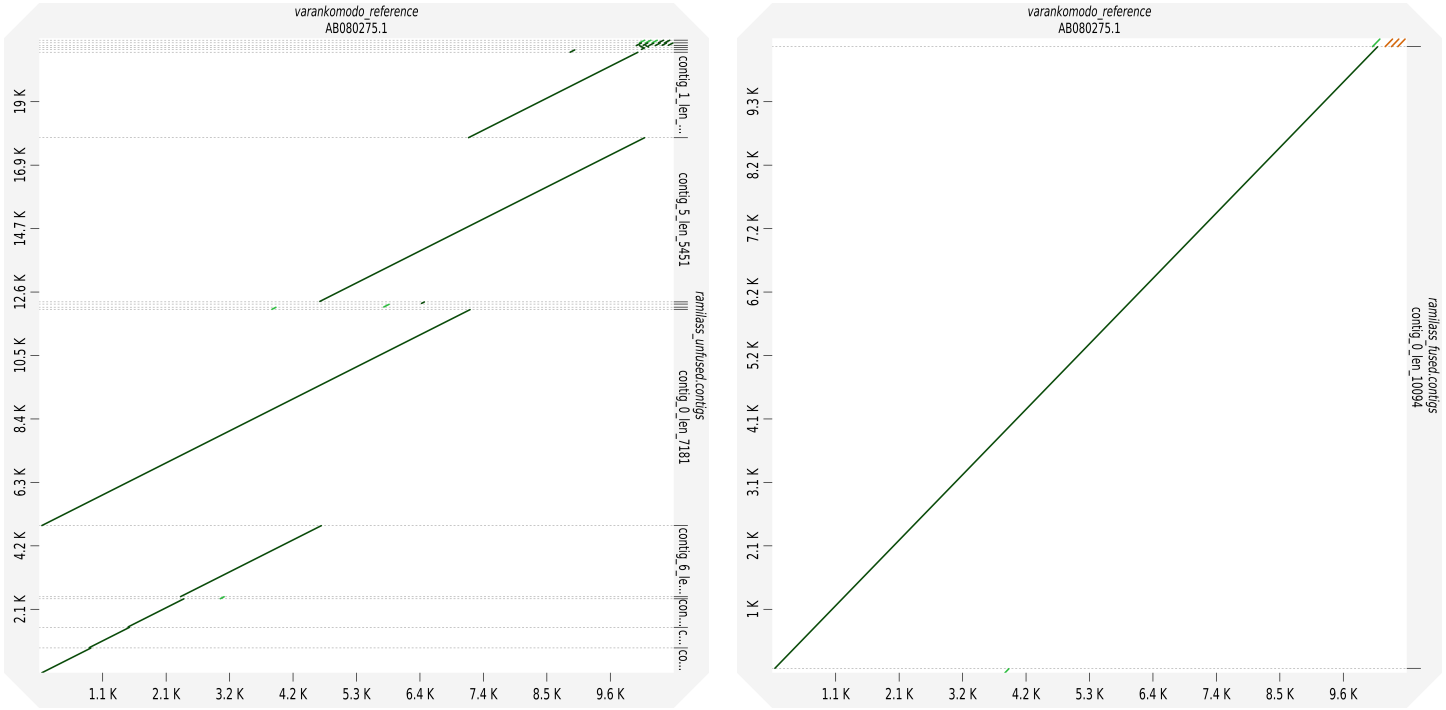
2 Résultats & discussion *(Exercice 6 & 7)*

2.1 Impact des étapes de simplification

Sans l'étape de fusion présentée plus haut, le graphe produit de nombreux contigs courts, localement cohérents mais fragmentés, et souvent recouvrants. La fusion rassemble ces segments et fait apparaître un contig unique, plus représentatif de la séquence attendue.

Dans notre cas d'étude, l'élagage des *tips* et la résolution des bulles n'ont eu (finalement) qu'un impact minimal : ils réduisent le graphe de 147 à 146 contigs, qui est dans notre cas, un contig compris dans un plus grand. Pour cette instance du problème, ces étapes de simplification ne sont donc pas déterminantes. Pour autant, on pourrait aller plus loin en examinant un plus grand nombre de cas limites et en soumettant des fichiers de lecture de plus grande taille afin de vérifier si cette observation se généralise. C'est un point que nous n'avons malheureusement pas eu le temps d'explorer de manière satisfaisante.

Inversement, la génération des chemins linéaires suivie de la fusion par "Deep Seeding" suffit à retrouver un contig continu. Cette étape corrige naturellement les fragments qui se chevauchent, sans dépendre des heuristiques de nettoyage du graphe. Pour garantir une évaluation honnête, nous avons désactivé l'élagage des *tips* et la résolution des bulles durant les benchmarks. Ces opérations prennent trop de temps pour un résultat similaire dans notre cas biologique. Leur retrait permet donc de mesurer uniquement l'effet réel du couple "génération + fusion". Ce que vous pouvez constater dans la figure 3 ci-dessous :



(a) Ramilass Contigs sans fusion **(b)** Ramilass Contigs avec fusion

Figure 3:: Comparaison des alignements avec (a) et sans (b) fusion par rapport à la référence Varankomodo

2.2 (Evaluation quantitative et qualité de l'assemblage *(Exercice 7)*

La comparaison entre `Minia` et `Ramilass` révèle que ce dernier génère un contig principal plus long et plus proche de la référence (cf. table 1). Bien que légèrement plus déréglé par les erreurs (≈ 89/100 kbp), le profil GC reste très proche de la référence (44.03%), ce qui signe une reconstruction biologique cohérente. Le taux d'erreur correspond à un score Phred d'environ 30, un seuil généralement considéré comme satisfaisant pour l'assemblage.

Table 1:: Statistiques d'assemblage : Minia vs Ramilass Fuse

Statistiques	Minia	Ramilass
Référence : longueur = 10 624 bp, GC = 44.14%		
Nombre de contigs (≥ 0 bp)	4	3
Nombre de contigs (≥ 1000 bp)	1	1
Longueur totale (bp)	10 187	10 301
Plus grand contig (bp)	9 936	10 094
GC (%)	44.05	44.03
Mismatches / 100 kbp	0	89.16
Indels / 100 kbp	0	9.91

Les mesures de performance confirment l'intérêt du modèle proposé (cf. table 2). L'assembleur s'exécute plus rapidement (0,32 s) et consomme beaucoup moins de mémoire (34 Mo) que `Minia` (249 Mo) ou `SPAdes` (69 Mo). Cette efficacité découle d'une représentation compacte par *bitvectors*, d'un traitement minimal des simplifications et d'une fusion basée sur les *seeds* plutôt que sur une reconstruction complète du graphe. La faible variabilité des mesures montre un comportement stable et reproductible, assurant un assemblage proche de la référence tout en minimisant les coûts computationnels.

Table 2:: Performances : temps d'exécution et mémoire (table 3)

Outil	Temps (s)	Mémoire (MB)
Minia	0.59 ± 0.02	249.7 ± 1.2
Ramilass	0.32 ± 0.00	34.2 ± 0.2
SPAdes	4.93 ± 0.10	69.9 ± 0.2

Nous rajouterons avoir utilisé `Valgrind` pour vérifier l'absence de fuites mémoire. Les résultats confirment que toutes les allocations sont correctement libérées, cet outil découvert en DevOps nous à paru approprié dans le contexte de ce projet, exemple pour une instance de test : total heap usage: 215,529 allocs, 215,529 frees.

2.3 Limites identifiées & perspectives d'améliorations *(Exercice 6)*

Nous vous listons/proposons quelques unes des critiques, assurément, non exhaustives de notre travail, ci-dessous.

Prise en charge d'un alphabet limité : tout d'abord, rappelons que la gestion des fichiers se limite aux nucléotides A, T, C et G. Les lettres dégénérées du standard `IUPAC` telles que N, R ou Y sont ignorées, ce qui peut poser problème dans le cadre de fichiers de qualité médiocre, hétérogènes.

L'apport de données pairées : Une autre limite est les modalités de génération de nos chemins linéaires dans le graphe qui peut conduire à la formation de contigs chimériques. L'intégration de données pairées pourrait limiter ce phénomène et améliorer la résolution de structures complexes. En effet, nous avons remarquer que des amas complexes de *tips* et bulles apparaissent (Figure 5), difficiles à résoudre sans heuristique fine.

La perte d'information haplotypique : Nous savons également que pour des génomes polyploïdes, en particulier homozygotes, la fusion des contigs, dans la majorité des assembleurs *de novo*, tend à condenser plusieurs haplotypes en un seul, masquant de la diversité génétique, notre outil ne fera pas exception à cette limite.

Un compromis qualité/vitesse: La stratégie que nous avons adoptée de fusion "par pas de k-mers" plutôt qu'un nucléotide par nucléotide sacrifie légèrement la qualité pour gagner en vitesse, accélérant l'assemblage certes, mais pouvant introduire de petites erreurs de jonction.

La scalabilité reste, selon nous, le principal facteur limitant. Bien que l'outil soit performant sur de petits virus ou bactéries, il est probable qu'il échoue sur un génome humain (3×10^9 bases). Trois goulots majeurs ont été identifiés : la taille et le coût mémoire de la structure `Noeud` (**??**), la fragmentation due aux millions de micro-allocations via `new`, et la double indirection introduite par `std::vector<Noeud*>`. Cette dernière entraîne un accès indirect aux données : il faut d'abord lire le vecteur pour récupérer l'adresse du nœud enfant (`Noeud*`), puis suivre ce pointeur pour accéder aux champs de l'objet. Ce schéma, classique en C++, provoque une surcharge mémoire et dégrade l'efficacité pour de grands volumes de données.

2.4 Un brin de complexité

Notre programme se décompose en plusieurs étapes successives dont la complexité globale dépend principalement de la taille de nos lectures, du nombre de k-mers et du nombre de contigs. Nous pensons que la lecture du fichier `FASTA` et la construction du `BitVector` se font en temps linéaire par rapport au nombre total de nucléotides, $O(N)$ car chaque nucléotide est traité exactement une fois.

La création du graphe de kmers via `Graphdbj` suit également une logique linéaire, $O(N)$, puisque proportionnelle au nombre de k-mers générés. Les étapes de simplification, `resolveBubbles` et `clipTips`, ont une complexité $O(P \cdot (V + E))$, avec P le nombre de passes et V, E le nombre de noeuds et d'arêtes, car chaque passe examine potentiellement dans le pire des cas tous les éléments du graphe brut. Ensuite, la comparaison des kmers entre lectures, effectuée par `CompareKMers`, est en théorie quadratique en nombre de lectures et linéaire en taille des k-mers, $O(R^2 \cdot k)$, puisqu'on doit comparer chaque k-mer à tous ses compères. Mais en pratique on peut dire que seule la portion initiale des k-mers est réellement comparée pour la fusion, justifiant notre choix de ne pas considérer toutes les positions possibles dans le calcul théorique. La fusion des contigs, toujours via `CompareKMers`, est quadratique en nombre de contigs, $O(C^2 \cdot k)$. Enfin, l'export des contigs en FASTA ou GFA via `Convert` est proportionnel à la somme des longueurs des contigs, $O(S)$. En combinant les étapes dominantes que nous venons d'expliciter, la complexité théorique dans le pire des cas est donc :

$$O(N) + O(N) + O(P(V + E)) + O(R^2k + C^2k) + O(S) = O(N + P(V + E) + R^2k + C^2k + S)$$

En synthèse, nous proposons comme ordre de grandeur de Landau pour la complexité temporelle de notre algorithme :

$$\boxed{O_{\max} = O(\max(R^2k, C^2k))}$$

Ajoutons que $(R^2 \cdot k)$ domine si le nombre de lectures est grand et que k est conséquent. Mais que $(C^2 \cdot k)$ domine si le nombre de contigs est grand et que k est conséquent, donc l'ordre de grandeur synthétique global n'est pas tout à fait $\mathcal{O}(N)^2$.

2.5 Conclusion générale

Lors de ce TP nous avons réussit a réaliser un petit assembleur pour permettant d'assembler des *short reads* d'individus haploïde ou d'ADN mitochondriaux/chloroplastique. Ce dernier utilise des méthodes bien documenté dans la littérature ainsi que des optimisations en mémoire originales ce qui lui permet d'être a vitesse équivalente, moins gourmand en mémoire et assemblant des contigs plus long que `minia` pour notre problème biologique. Nous

avons entièrement décrit le fonctionnement de notre logiciel avec la description de ses complexités en temps et en mémoire ainsi que la détermination de ses limitations.

Vous retrouverez l'entièreté du code source sur ce répertoire Github (https://github.com/MickaelCQ/RaMiLass). Le projet a été entièrement conçu en privilégiant la reproductibilité, d'où l'intégration de Pixi et la possibilité d'exécuter notre outil dans un conteneur Singularity/Apptainer.

A Annexes

Etat de l'art : algorithme Glouton *(Exercice 1)*

Algorithm 5: Assemblage_Glouton()

Entrée: Famille de mots F de taille n, où L est la longueur moyenne des mots
Sortie: Chaîne de caractères S assemblée
Complexité: Temps : $\mathcal{O}(n^3 + n^2 L)$, Espace : $\mathcal{O}(n^2)$
Début
  $S \leftarrow$ "" ; $M \leftarrow$ tableau d'entiers de taille $n \times n$ $(\mathcal{O}(n^2))$
  Pour i allant de 0 à $n-1$ **Faire** $(\mathcal{O}(n^2 L))$
    Pour j allant de 0 à $n-1$ **Faire** $(\mathcal{O}(nL))$
      $M[i][j] \leftarrow$ longueur du chevauchement maximal entre $F[i]$ et $F[j]$ $(\mathcal{O}(L))$
    FinPour
  FinPour
  $m \leftarrow 0$
  TantQue $m < n-1$ **Faire** $(\mathcal{O}(n))$
    $imax \leftarrow 0$; $jmax \leftarrow 0$; $max \leftarrow M[0][0]$
    Pour i allant de 0 à $n-1$ **Faire** $(\mathcal{O}(n^2))$
      Pour j allant de 0 à $n-1$ **Faire** $(\mathcal{O}(n))$
        Si $max < M[i][j]$ **Alors**
          $imax \leftarrow i$; $jmax \leftarrow j$; $max \leftarrow M[i][j]$
        FinSi
      FinPour
    FinPour
    $S \leftarrow S + F[imax][0 : \text{longueur}(F[imax]) - max]$ $(\mathcal{O}(L))$
    $S \leftarrow S + F[jmax]$ $(\mathcal{O}(L))$
    Pour i allant de 0 à $n-1$ **Faire** $(\mathcal{O}(n))$
      $M[imax][i] \leftarrow -1$; $M[i][jmax] \leftarrow -1$
    FinPour
    $m \leftarrow m + 1$
  FinTantQue
  Retourner S
Fin

Etat de l'art : construction du graphe de chevauchement *(Exercice 1)*

Algorithm 6: Build_Graphe_Chevauchement()

Entrée: Famille de mots $\mathcal{F}$, où chaque F_i est un mot

Sortie: Graphe de chevauchement $G = (V, E)$, où V est l'ensemble des sommets (chaque sommet représente un mot de F) et E est l'ensemble des arêtes pondérées par la longueur du chevauchement

Complexité: Temps : $\mathcal{O}(n^2 L)$, Espace : $\mathcal{O}(n^2)$ où L est la longueur moyenne des mots

Début

$V \leftarrow F$

$E \leftarrow$ ""

Pour i allant de 0 à $n-1$ **Faire** $(\mathcal{O}(n^2 L))$

Pour j allant de 0 à $n-1$ **Faire** $(\mathcal{O}(nL))$

Si $i \neq j$ **Alors**

$k \leftarrow$ longueur du chevauchement maximal entre $F[i]$ et $F[j]$ $(\mathcal{O}(L))$

Si $k > 0$ **Alors**

$E \leftarrow E \cup \{(F[i], F[j], k)\}$

FinSi

FinSi

FinPour

FinPour

Retourner $G = (V, E)$

Fin

Algorithm 7: Assemblage_Graphe_Chevauchement()

Entrée: Famille de mots $\mathcal{F}$; graphe de chevauchement $G = (V, E)$ construit à partir de $\mathcal{F}$
Sortie: Chaîne de caractères S assemblée
Complexité: Temps : $\mathcal{O}(n^2L + n^3)$, Espace : $\mathcal{O}(n^2)$
Début
" Initialisation"
$S \leftarrow$ ""
Visite $\leftarrow$ tableau de booléens de taille n, initialisé à FAUX
chemin $\leftarrow$ liste vide
degré_entrant $\leftarrow$ tableau d'entiers de taille n, initialisé à 0
Pour chaque arête $(u, v, k) \in E$ **Faire**
degré_entrant$[v] \leftarrow$ degré_entrant$[v] + 1$
FinPour
$u \leftarrow$ sommet avec le plus petit degré_entrant
TantQue $u \neq$ AUCUN **Faire**
chemin.$ajouter(u)$
Visite$[u] \leftarrow$ VRAI
$v \leftarrow$ sommet non Visite tel que $(u, v, k) \in E$ et k est maximal
$u \leftarrow v$
FinTantQue
Si chemin.taille > 0 **Alors**
$S \leftarrow F$[chemin[0]]
Pour i allant de 1 à chemin.taille $- 1$ **Faire**
$k \leftarrow$ longueur du chevauchement entre F[chemin$[i-1]]$ et F[chemin$[i]]$
$S \leftarrow S + F$[chemin$[i]][k:]$
FinPour
FinSi
Retourner S
Fin

Figure 4:: Etat de l'art :Algorithme de construction du graphe de chevauchement, comme étudié en cours

Spécifications du Système pour les benchmarks

Catégorie	Détail
Système	
Système d'Exploitation	**Ubuntu 24.04.3 LTS**
Noyau (Kernel)	**6.14.0-36-generic**
Processeur (CPU)	
Modèle	**AMD Ryzen 7 7730U with Radeon Graphics**
Architecture	x86_64
Cœurs physiques | Threads	8 cœurs | 16 threads
Fréquence Max.	4547 MHz
Mémoire Vive (RAM)	
Taille Totale	15 GiB
Graphique (GPU)	
Contrôleur | Modèle	**Advanced Micro Devices, Inc. [AMD/ATI]**
Produit (Puce Intégrée)	**Barcelo (AMD Radeon Graphics)**

Table 3:: Spécifications du Système - Ubuntu

Exemple d'une structure complexe : cas limite de notre algorithme *(Exercice 6)*

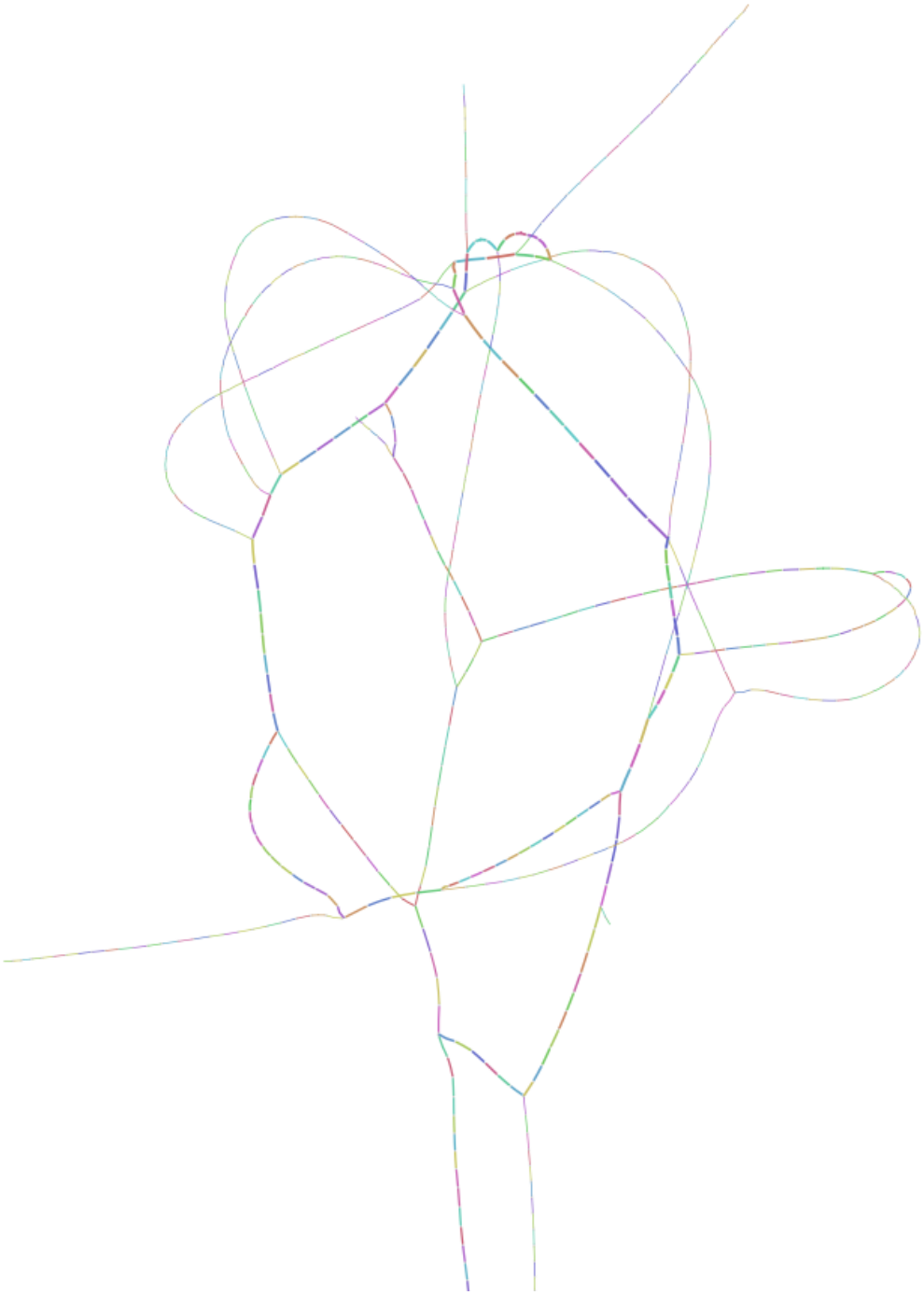


Figure 5:: Visualisation `Bandage` d'une structure non résolue par l'algorithme.

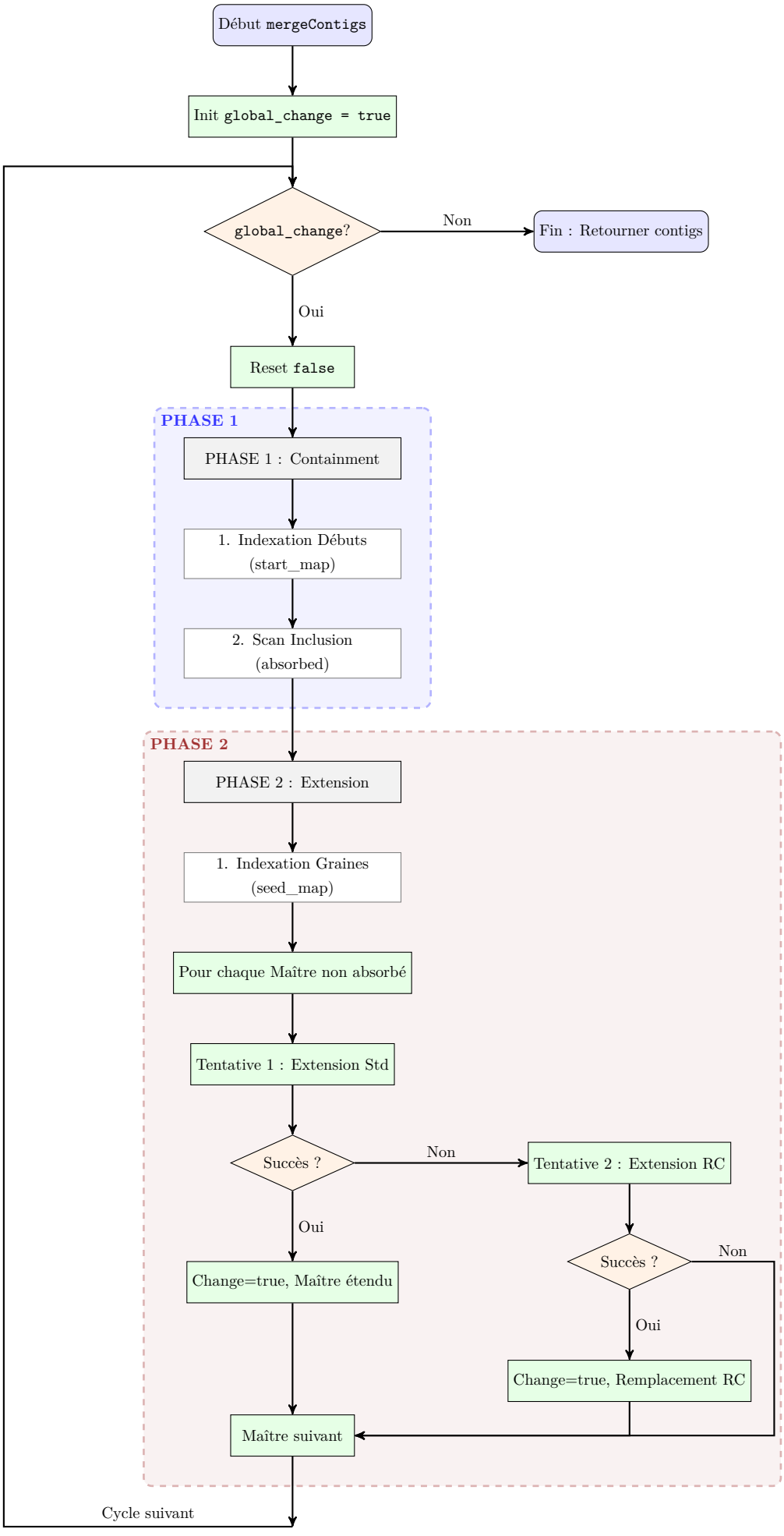

Début mergeContigs
Init global_change = true
global_change?
Non
Fin : Retourner contigs
Oui
Reset false
PHASE 1
PHASE 1 : Containment
1. Indexation Débuts (start_map)
2. Scan Inclusion (absorbed)
PHASE 2
PHASE 2 : Extension
1. Indexation Graines (seed_map)
Pour chaque Maître non absorbé
Tentative 1 : Extension Std
Succès ?
Non
Tentative 2 : Extension RC
Oui
Succès ?
Non
Change=true, Maître étendu
Oui
Change=true, Remplacement RC
Maître suivant
Cycle suivant

Table 4:: Répartition du temps d'exécution optimisé de l'Assembleur GraphDBJ.

Étape	Temps (ms)	Pourcentage du Total
Création du Graphe	175	53.8%
Fusion des Contigs	100.5	30.9%
Écriture Fichiers	36	11.1%
Génération Contigs	9.5	2.9%
Lecture FASTA	4.5	1.4%
Simplification	0	0.0%
TOTAL	**325.5**	**100.0%**

B Références

Bibliographie

[1] Rayan Chikhi and Guillaume Rizk. "Space-efficient and exact de Bruijn graph representation based on a Bloom filter". In: *Algorithms for Molecular Biology* 8.1 (Jan. 2013), p. 22. ISSN: 1748-7188. DOI: 10.1186/1748-7188-8-22. URL: https://almob.biomedcentral.com/articles/10.1186/1748-7188-8-22.

[2] Michael S. Waterman Pavel A. Pevzner Haixu Tang. "An Eulerian path approach to DNA fragment assembly". In: *Genetics* 98.17 (Aug. 2001). ISSN: 9748-97537. DOI: 10.1073/pnas.171285098. URL: https://doi.org/10.1073/pnas.171285098.

[3] Patterson M Rizzi R Beretta S. "Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era". In: *Quant Biol* 7.16 (Dec. 2019), pp. 278–292. ISSN: 278-292. DOI: 10.1007/s40484-019-0181-x. URL: https://doi.org/10.1007/s40484-019-0181-x.

[4] Chen-Fu Liao Yao-Ting Huang. "Integration of string and de Bruijn graphs for genome assembly". In: *Bioinformatics* 32.9 (Dec. 2016), pp. 25–37. ISSN: 1301-1307. DOI: 10.1093/bioinformatics/btw011. URL: https://doi.org/10.1093/bioinformatics/btw011.

[5] Desheng Mu Zhenyu Li Yanxiang Chen. "Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph". In: *Briefings in Functional Genomics* 11.2 (Dec. 2012), pp. 25–37. ISSN: 25-37. DOI: 10.1093/bfgp/elr035. URL: https://doi.org/10.1093/bfgp/elr035.

Webographie :

QUAST : https://github.com/ablab/quast
D-GENIES : https://dgenies.toulouse.inra.fr/
Minia : https://gatb.inria.fr/software/minia/
SPades : https://github.com/ablab/spades