Cours HAU902I : Bioinformatique avancé

Projet : Algorithme d'assemblage | Fait par : Conceptia Dagba Allade ; Hermine Kiossou ; Homero Sanchez

Algorithme d'Assemblage

L'algorithme d'assemblage repose sur le principe de reconstruire une séquence d'ADN ou d'ARN à partir de fragments courts issus du séquençage. L'objectif est d'exploiter les chevauchements entre ces reads pour retrouver, autant que possible, la séquence originale. Cette étape, réalisée entièrement in silico, suit immédiatement le séquençage d'un organisme, d'une population clonale (comme une culture bactérienne) ou d'un mélange complexe.

Dans ce projet, nous avons réimplémenté un assemblage de type OLC (Overlap–Layout–Consensus). Cette approche nous a permis de suivre en détail chaque phase de l'algorithme (du calcul des chevauchements à la construction du consensus) et d'évaluer son fonctionnement sur des jeux de données réels.

1. Overlap-Layout-Consensus

Dans cette méthode, on exploite le calcul des chevauchements pour construire un graphe permettant d'identifier les lectures qui se chevauchent (overlap), de les organiser en une séquence continue (layout) puis de corriger les erreurs afin d'obtenir une séquence consensus.

- La phase **Overlap** consiste à calculer les chevauchements optimaux entre les reads par **alignement semi-global**, à construire progressivement un graphe orienté reliant les reads par des arêtes pondérées selon la qualité du chevauchement, et à exclure les reads entièrement contenus dans d'autres. On peut coder le graph de chevauchement sous la forme de **matrice d'adjacence** où la case **M[i][j]** stocke le poids du chevauchement du read i avec le read j.
- La phase **Layout** vise à trouver un ordre optimal des reads en résolvant un problème de type TSP, que l'on peut aborder soit par des méthodes exactes (comme la programmation dynamique ou le branch and bound), soit par des heuristiques plus rapides (glouton, plus proche voisin, k-opt, Lin Kernighan, etc.), en adaptant le problème asymétrique en un TSP symétrique.
- La phase **consensus** consiste à effectuer un alignement multiple des séquences pour en obtenir une version optimisée en score, mais comme l'alignement exact devient rapidement impraticable au-delà d'une dizaine de séquences, on utilise généralement des méthodes heuristiques.

2. Avantages et Inconvénients

a. Avantages

- Très adapté aux longues lectures : Cette approche fonctionne particulièrement bien avec les longues lectures.
- Assemblage généralement plus continus : Les longues lectures peuvent traverser des zones répétées du génome sans se casser en morceaux, ce qui permet de reconstruire des séquences plus longues et avec moins de coupures.
- Modèle explicite de l'assemblage : Le graphe OLC offre une vision claire des chevauchements entre les lectures et facilite les étapes de correction ou de vérification.
- Résistant aux erreurs systématiques : Les erreurs aléatoires des lectures longues sont largement corrigées lors de la phase de consensus final.

---

b. Inconvénients

- Coût computationnel élevé : Comparer toutes les lectures entre elles demande beaucoup de temps et de mémoire, ce qui nécessite des optimisations.
- Peu adapté aux très grands jeux de données à courtes lectures : Avec de très nombreuses lectures courtes, l'OLC devient trop lourd et les graphes de Bruijn sont bien plus efficaces.
- Sensibilité aux régions très répétées : Certaines répétitions complexes peuvent créer des ambiguïtés dans le graphe, même avec des lectures longues.
- Pipeline plus complexe : Le processus OLC comporte plusieurs étapes distinctes qu'il peut être plus difficile de paramétrer et d'optimiser que dans un pipeline de Bruijn.

3. Pipeline de l'outil d'assemblage OLC

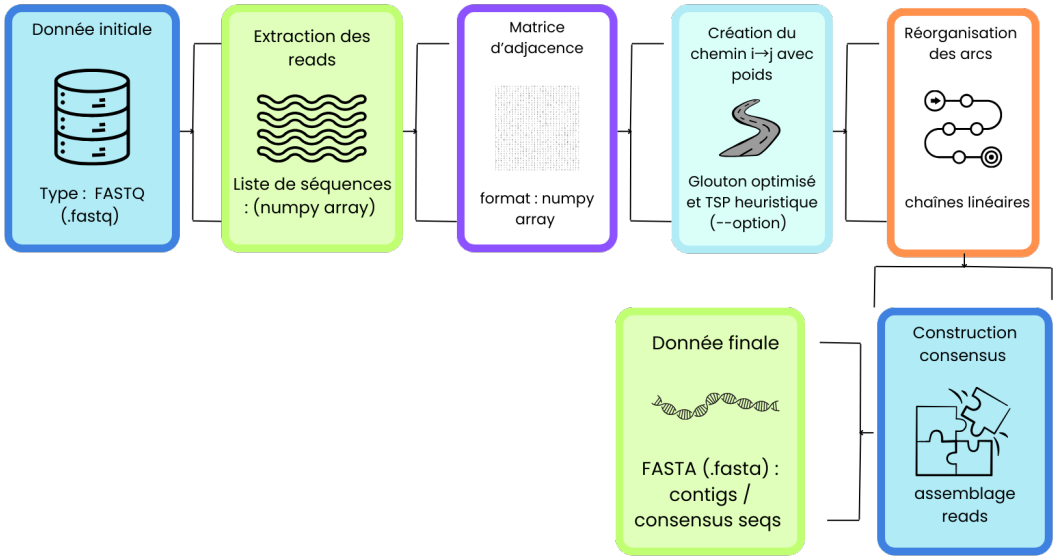

Figure 1: Pipeline OLC

4. Choix du langage de programation

Pour l'implémentation, nous avons retenu Python comme langage principal. Ce choix s'explique d'abord par sa rapidité de développement : Python permet de prototyper, tester et modifier l'algorithme d'assemblage de manière flexible, ce qui est essentiel dans un projet où plusieurs étapes (overlap, layout, consensus) doivent être ajustées progressivement.

Python bénéficie également d'un écosystème scientifique très mature, particulièrement adapté aux besoins de ce projet. Par exemple :

- NumPy permet de manipuler efficacement des matrices, ce qui est crucial pour la construction et l'exploitation de la matrice de chevauchement. Ses tableaux optimisés en C sont nettement plus performants que les listes natives de Python lorsqu'il s'agit d'opérations répétitives et intensives.
- La bibliothèque python_tsp met à disposition des heuristiques TSP déjà optimisées, ce qui nous a permis d'intégrer une solution de layout alternative sans devoir réimplémenter manuellement un solveur

complet.

- Python dispose également de nombreuses autres librairies utiles (gestion de fichiers, analyse de séquences, visualisation), permettant d'envisager facilement des extensions futures.

Enfin, Python offre une lisibilité élevée, ce qui facilite la compréhension et la maintenance du code, notamment dans un contexte collaboratif. Son adoption massive en bio-informatique en fait un choix naturel et cohérent avec les outils du domaine.

5. Algorithmes correspondant à chacune des étapes

a. Extraction des reads :

```
Algorithme ExtractionReads_FastQ
Entrée :
   fichier_fastq : chemin vers le fichier FASTQ
Sortie :
   Reads : liste de séquences

 Début
   Ouvrir fichier_fastq en lecture;
   Reads <- [];
   longueur_ref <- 0;
   compteur <- 0;

   Tant que fichier n'est pas fini faire          // Chaque read = 4 lignes dans FASTQ
      ignorer_ligne();                           // Header
      sequence <- lire_ligne();
      ignorer_ligne();                           // Ligne "+"
      ignorer_ligne();                           // Qualité

      Si compteur == 0 alors
         longueur_ref <- longueur(sequence);
      Sinon Si longueur(sequence) != longueur_ref alors
         ERREUR : "Read de taille différente détecté";
      Fin Si

      Ajouter sequence à Reads;
      compteur <- compteur + 1;
   Fin Tant Que

  Fermer le fichier;
  Retourner Reads;
Fin
```

b. Matrice d'adjacence

```
Algorithme overlap
Entrée :
   A, B : chaînes de caractères
Sortie :
   o : Entier, longueur du plus long suffixe de A qui est un préfixe de B

Début
   o <- 0;
   lenA <- longueur(A);
```

```
   lenB <- longueur(B);
   max_possible <- min(lenA, lenB);

    // on compare le suffixe de longueur k de A avec le préfixe de longueur k de B
   Pour k allant de 1 à max_possible faire
      Si sous_chaine(A, lenA - k, lenA) = sous_chaine(B, 0, k) alors
         o <- k;
     Fin Si
  Fin Pour

  Retourner o;
Fin



Algorithme matrice_adjacence
Entrée:
   Reads: Liste de chaînes de caractères, de taille n
Sortie:
   M: matrice d'adjacence des chevauchements (Matrice d'entiers n×n)

Début
   Pour i allant de 0 à n - 1 faire
       Pour j allant de 0 à n - 1 faire
           Si i = j alors
              M[i][j] ← -1;
           Sinon
              M[i][j] ← overlap(Reads[i], Reads[j]);
           Fin Si
        Fin Pour
    Fin Pour

   Retourner M
Fin
```

c. Recherche du chemin hamiltonien :

```
Algorithme Glouton_Layout_Matrice_Optimisé
Entrée :
    M : matrice d'adjacence des chevauchements (Matrice d'entiers n×n)
    len_read : longueur des reads (entier)
Sortie :
    chemin : Liste de triplets [i, j, poids] représentant les arcs du chemin hamiltonien
Début
    chemin ← liste vide;

    // Suivi des degrés pour construire un chemin sans cycles
    degre_sortant ← liste de n zéros;
    degre_entrant ← liste de n zéros;

    arcs_rejetes_degre ← 0;
    arcs_rejetes_cycle ← 0;

    max_val ← -1;
```

```
    // les chevauchements plus courts (7% de la longueur du read) sont pas consultés
    Tant que len(chemin) < n - 1 ou max_val > len_read * 0.07 faire

       max_val ← -1;
       i_max ← 0;
       j_max ← 0;

       Pour i de 0 à n - 1 faire
           Pour j de 0 à n - 1 faire
                Si i != j et M[i, j] > max_val alors
                  i_max ← i;
                  j_max ← j;
                  max_val ← M_copy[i, j];      // Chevauchement maximal
                Fin Si
           Fin Pour
       Fin Pour

       // Vérification des conditions d'ajout
       Si max_val > len_read * 0.07 alors :

          // Vérifier les contraintes de degré
          Si degre_sortant[i_max] >= 1 ou degre_entrant[j_max]>= 1 alors
             arcs_rejetes_degre ← arcs_rejetes_degre + 1;
          // Vérifier si on crée un cycle
          Sinon Si CREER_CYCLE(chemin, i_max, j_max) = VRAI alors :
             arcs_rejetes_cycle ← arcs_rejetes_cycle + 1;
        Sinon
             // Ajouter l'arc au chemin
             Ajouter [i_max, j_max, max_val] à chemin;
             degre_sortant[i_max] ← degre_sortant[i_max] + 1;
             degre_entrant[j_max] ← degre_entrant[j_max] + 1;
         Fin Si

       Fin Si

       // Supprimer les arcs liés à i_max et j_max
       Pour k de 0 à n - 1 faire
           M[i_max, k] ← -1;
           M[k, j_max] ← -1;
       Fin Pour

  Fin Tant que

   Retourner chemin;
Fin
```

d. Consensus :

```
Algorithme reoganiser_chemin
Entrée
    chemin : liste de triplets [i, j, poids] représentant des arcs
Sortie
    chemin_organisé : Liste de chaînes, où chaque chaîne est une liste ordonnée d'arcs
```

```
Début
   Si chemin est vide alors
      Retourner liste vide
   Fin Si

   // Construction du graphe
   successeurs ← dictionnaire vide
   predecesseurs ← ensemble vide
   tous_reads ← ensemble vide

   Pour chaque arc dans chemin faire
      i ← arc[0]
      j ← arc[1]
      poids ← arc[2]

      successeurs[i] ← (j, poids)
      Ajouter j à predecesseurs
      Ajouter i à tous_reads
      Ajouter j à tous_reads
   Fin Pour

   // Identification des points de départ
   points_depart ← tous_reads - predecesseurs

   // Construction des chaînes
   chaines ← liste vide

   Pour chaque depart dans points_depart faire
      chaine ← liste vide
      courant ← depart
      visite ← ensemble vide

      // Suivre la chaîne tant qu'il y a un successeur
      Tant que courant inclu dans successeurs et courant  pas inclu dans visite faire
          Ajouter courant à visite
          suivant, poids ← successeurs[courant]
          Ajouter [courant, suivant, poids] à chaine
          courant ← suivant
       Fin Tant que

       Si chaine n'est pas vide alors
          Ajouter chaine à chaines // Chaine trouvé
       Fin Si
   Fin Pour

   Si chaines est vide alors
      Retourner liste vide
   Fin Si

   Retourner chaines
Fin
```

```
Algorithme consensus
Entrée
   Reads : tableau de séquences (chaînes de caractères)
   chemin_brut : liste de triplets [i, j, poids] représentant des arcs
Sortie
   Liste de séquences consensus (chaînes de caractères)

Début
   Si chemin_brut est vide alors
      Afficher "ERREUR: Chemin vide"
      Retourner liste vide
   Fin Si

   // Réorganisation du chemin
   toutes_les_chaines ← reorganiser_chemin(chemin_brut)
   Si toutes_les_chaines est vide alors
      Afficher "ERREUR: Impossible de réorganiser"
      Retourner liste vide
   Fin Si

   seqs ← liste vide

   // Construction du consensus pour chaque chaîne
   Pour index de 0 à |toutes_les_chaines| - 1 faire
       chaine_ordonnee ← toutes_les_chaines[index]
       Premier assemblage i0, j0, p0 ← chaine_ordonnee[0]
       seq ← Reads[i0] + Reads[j0][p0:]

       // Assemblages suivants
       Pour k de 1 à |chaine_ordonnee| - 1 faire
          i, j, p ← chaine_ordonnee[k]
          seq ← seq + Reads[j][p:]
       Fin Pour

       Ajouter seq à seqs
   Fin Pour

   Retourner seqs
Fin
```

5. Évaluation de l'assembleur sur le génome mitochondrial du varan de Komodo

Le programme a été testé sur les séquences du génome mitochondrial du varan de Komodo. Le temps moyen d'exécution observé est de **4 minutes et 13,55 secondes**. Les difficultés rencontrées sont :

- Gestion et manipulation de grandes séquences d'ADN.
- Optimisation de l'algorithme pour limiter le temps de calcul.
- Gestion des chevauchements complexes entre reads pour un assemblage fiable.

6. Analyse de l'assemblage avec QUAST

Métrique	OLC_Result	Minia	Observations
Nombre de contigs	30	1	OLC fragmente l'assemblage
Longueur totale	24 835 bp	9 936 bp	OLC génère des duplications (ratio 2,374×)
Contig le plus long	2 233 bp	9 936 bp	Minia reconstruit la séquence complète
N50	818 bp	9 936 bp	Contiguïté excellente pour Minia
Couverture du génome	98,3%	93,5%	OLC couvre mieux mais avec redondance

ci après les résultats de QUAST :

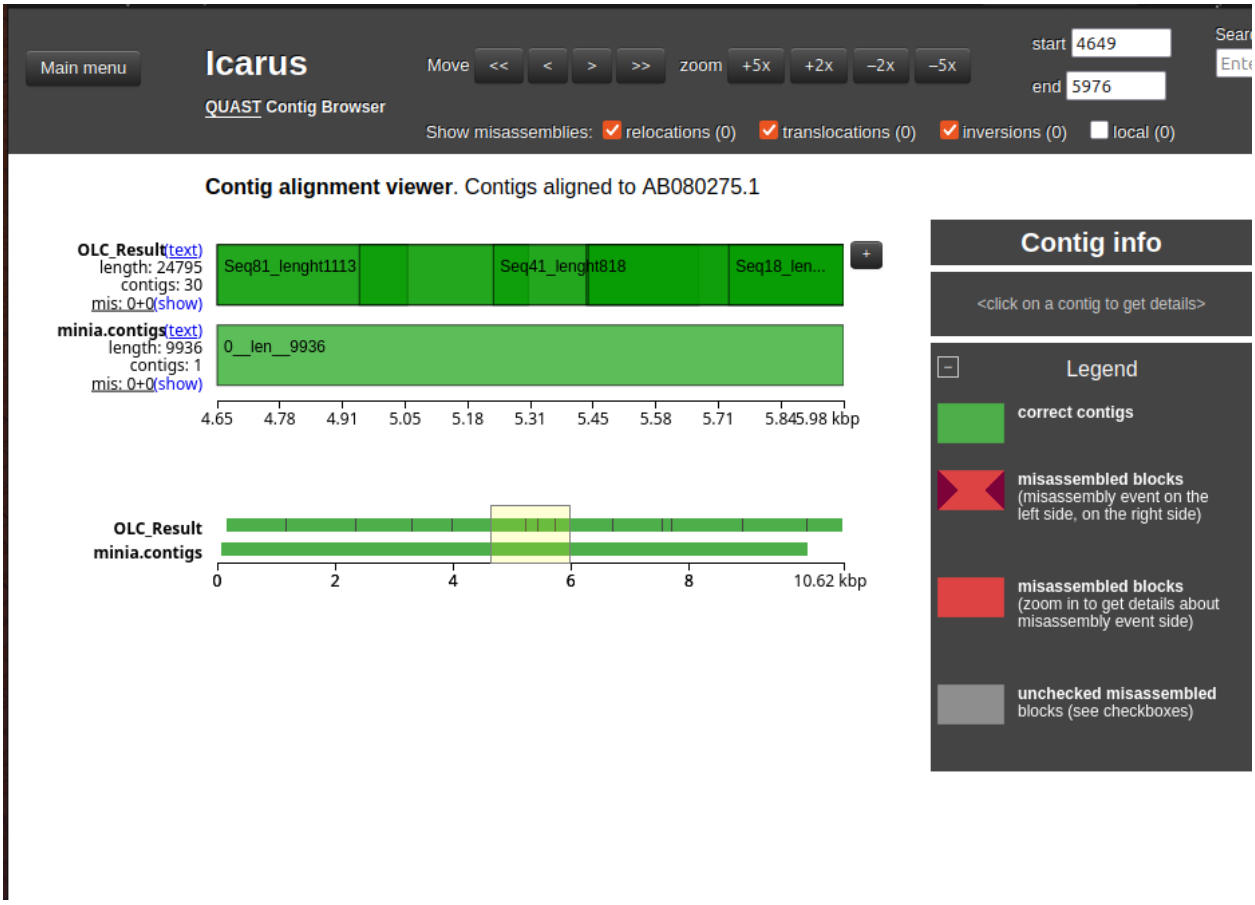

Figure 2: Résultat Quast

a. Qualité globale :

Minia produit un assemblage nettement supérieur, avec **1 seul contig de 9 936 bp** couvrant **93,5 %** du génome de référence (10 624 bp). En comparaison, notre assembleur OLC génère **30 contigs** totalisant **24 835 bp** avec une couverture de **98,3 %**.

---

b. Forces et faiblesses de l'assembleur OLC :

Points positifs :

- Aucun misassemblage structurel détecté.

- Couverture du génome légèrement meilleure que celle de Minia.
- Visualisation Icarus confirme des alignements corrects avec des blocs verts.
- L'algorithme glouton identifie correctement les chevauchements et respecte l'ordre des reads.

Points faibles :

- Fragmentation excessive : 30 contigs au lieu d'une séquence unique.
- Duplication des régions : ratio de duplication de 2,374, gonflant artificiellement la longueur totale.
- Faible contiguïté : N50 de 818 bp.
- Erreurs de séquence : 75 mismatchs détectés (302,48 pour 100 kbp).
- Incapacité de fusionner les chaînes disjointes en un contig unique, limitant l'utilité biologique de l'assemblage.

Conclusion

L'algorithme glouton OLC montre une bonne couverture et des chevauchements correctement identifiés, mais il échoue à produire un assemblage contigu et fiable. En revanche, Minia, basé sur un graphe de De Bruijn, produit un contig unique, sans duplication et plus exploitable biologiquement.