DNA fragment assembly: graph structures and chloroplast genome scaffolding

Victor Epain

▶ To cite this version:

Victor Epain. DNA fragment assembly: graph structures and chloroplast genome scaffolding: Comparative analyses, formulations and implementations. Bioinformatics [q-bio.QM]. Université de Rennes, 2023. English. NNT : 2023URENS083 . tel-04357206

HAL Id: tel-04357206
https://inria.hal.science/tel-04357206

Submitted on 21 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

COLLEGE DOCTORAL BRETAGNE | MATHS, TELECOMS INFORMATIQUE, SIGNAL SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Victor EPAIN

Assemblage de fragments ADN : structures de graphes et échafaudage de génomes de chloroplastes

Analyses comparatives, formulations et implémentations

Thèse présentée et soutenue à Centre Inria de l'Université de Rennes, le 27 novembre 2023
Unité de recherche : Inria / IRISA UMR 6074

Rapporteurs avant soutenance :

Éric ANGEL	Professeur des universités, IBISC, Université Paris-Saclay
Annie CHÂTEAU	Maitresse de conférence - HDR, LIRMM, Université de Montpellier

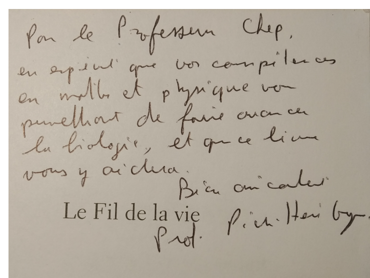
Composition du Jury :

Président :	Élisa FROMONT	Professeure des universités, IRISA, Université de Rennes
Examinateurs :	Éric ANGEL	Professeur des universités, IBISC, Université Paris-Saclay
	Annie CHÂTEAU	Maitresse de conférence - HDR, LIRMM, Université de Montpellier
	Élisa FROMONT	Professeure des universités, IRISA, Université de Rennes
	Camille MARCHET	Chargée de recherche CNRS, CRIStAL, Lille
	Mathias WELLER	Professor, Institut für Softwaretechnik und Theoretische Informatik, Berlin
	Dominique LAVENIER	Directeur de recherche CNRS, IRISA, Rennes
Dir. de thèse :	Rumen ANDONOV	Professeur des universités, IRISA, Université de Rennes
Co-dir. de thèse :	Jean-François GIBRAT	Directeur de recherche, INRAe, Université Paris-Saclay

REMERCIEMENTS

En fonction des personnes que je remercie, ces mots sont écrits en français ou en anglais / Depending on whom I am addressing, I am writing these acknowledgements in French or in English.

Merci à Annie Château et Éric Angel qui ont évalué et validé ce manuscrit pour me permettre de soutenir. Vos remarques positives et de fond m'ont encouragé pour l'oral et m'ont permis de bien préparer les questions. Merci à Elisa Fromont d'avoir présidé le jury, composé des rapporteurices, de Camille Marchet et de Mathias Weller. Je les remercie tou·te·s de s'être déplacé·es à Rennes pour assister en présence à ma soutenance, en particulier Annie qui venait de Montpellier, Mathias de Berlin, Camille de Lille et Éric de Paris. Nos échanges ont été plaisants, et je peux affirmer que ma soutenance a été un très beau moment de ma vie.


Pour le Professeur Chep,
en espérant que vos compétences
en maths et physique vous
permettront de faire avancer
la biologie, et que ce livre
vous y aidera.
Bien amicalement
Le Fil de la vie
Prof.

Lors de ma dernière année de prépa maths-physique au Lycée Dupuy de Lôme de Lorient (2017), j'ai assisté à une conférence donnée par Pierre-Henri Gouyon pour introduire son livre Le Fil de la vie, La face immatérielle du vivant, coécrit avec Jean-Louis Dessalles et Cédric Gaucherel. Leur livre illustrait entre autres comment les mathématiques et les modèles de la physique du signal permettaient d'appréhender des logiques et des tendances biologiques. À l'issue de la conférence, P-H Gouyon m'encourageait à revenir vers la biologie (voir son autographe en figure ci-contre).

J'ai ainsi souhaité continuer en master bioinformatique à Rennes, et la transition était toute trouvée : une licence d'informatique à l'Université de Rennes 1. Rumen Andonov m'a d'abord accueilli pour un stage de deux mois. À ce moment-là, Sébastien Letort me poussait à la rigueur même si je programmais en `Python`, chose que j'ai gardé, merci à toi. Rumen et Dominique Lavenier m'ont de nouveau intégré dans l'équipe GenScale pour le stage de M1. Cette fois-ci, nous nous sommes envolés pour Los Alamos, première fois que je prenais l'avion, première fois que je traversais l'Atlantique. Merci Rumen pour ce mois de cohabitation, de travail et de voyages. Thank you Hristo Djidjev for having welcomed us into your laboratory (LANL) and for the valuable discussions about graph partitioning. Jamais deux sans trois, mon dernier stage d'étude s'est fait… à GenScale !

Rumen et Dominique m'ont alors présenté Jean-François Gibrat avec qui nous avons préparé le sujet d'une thèse que nous souhaitions soumettre à concours. À l'origine, le titre était : *Développement de méthodes efficaces, précises et conviviales pour corriger, assembler et aligner des lectures issues des technologies de séquençage troisième génération.*

Merci à Olivier Dameron, Annabelle Monnier et Sophie Schbat d'avoir soutenu ma candidature. Merci aux membres du CSID, Guillaume Fertin et Christophe Klopp, pour les discussions scientifiques et leur vérification en faveur du bon déroulement de la thèse. Rumen, tu m'as toujours dit : « l'important, c'est que tu te fasses plaisir dans ta recherche ». Merci infiniment à toi, Dominique et Jean-François pour votre confiance, les très nombreuses discussions scientifiques, et de m'avoir permis d'exercer mes premiers pas de chercheur librement.

Durant mes deux premières années de thèse, j'ai eu l'occasion et la chance d'enseigner dans les matières qui me tenaient à cœur : méthodes algorithmiques sur les graphes et programmation linéaire. Merci Rumen, Kerian Thuillier et Arthur Gontier d'avoir été une super équipe pédagogique. Merci aux étudiant·es pour leurs retours alors que l'on enseignait en hybride, puis à distance confiné·es : leurs messages m'ont confirmé ma volonté d'enseigner. Merci à Pauline Hamon-Giraud pour son travail à nos côtés en stage de M1.

Thank you, Gunnar Klau and Sven Schrinner from the HHU ALBI team in Düsseldorf, for welcoming me during two months while we were threatened by the lockdown due to COVID-19, for your valuable discussions about integer linear programming and problem complexity.

Merci beaucoup Nicolas Buton, Garance Gourdel, Matthieu Bouguéon et Lucas Robidou d'avoir formé l'équipe de doctorant·es que nous avons été : nous garderons de très bons souvenirs de la réalisation de notre court métrage pour Sciences en Cour[t]s, de l'organisation du même festival l'année suivante, de la gestion de l'association Nicomaque. Merci pour votre soutien au quotidien.

Merci à Clara Delahaye, Olivier Dennler, Grégoire Siekaniec et Téo Lemane, nos ainé·es, de nous avoir accompagné·es dans la recherche, dans Sciences en Cour[t]s et Nicomaque.

Elisa Fromont, merci pour ta confiance et de m'avoir laissé la liberté de ton pour l'organisation des séminaires annuels DKM 2023. Un grand merci également à Marie Farge pour nos discussions concernant l'éthique de la publication scientifique

et tes encouragements pour la suite.

Merci Ambre Ayat pour ton aide dans l'organisation et l'animation de l'atelier éthique dans l'ESR. Merci Bernard Friot et Enka Blanchard pour les échanges autour du productivisme dans la recherche et les moyens d'en sortir.

To my (the best) office colleagues, Nicolas Guillaudeux, Khodor Announsh and Arya Kaul: thank you for the discussions and your cheerfulness.

Un merci chaleureux aux personnes des équipes Symbioses, pour les discussions sérieuses aux rires béats – parfois les deux en même temps, à table comme aux pauses : merci à Samuel, François C., Konogan, Meven, Pierre, Stéphanie, Catherine, Émeline, Claire, Emmanuelle, Jacques, Jeanne, Roland, Sandra, Baptiste, Victor, Camille, Riccardo, Olivier B., Sarah, Thomas R., Thomas C., Gildas, Gaëtan, Grégoire P., François M., Karel, Julien, Florian. Merci Marie et Gaëlle pour votre travail de support quotidien à la recherche et les rigolades. Laurence, Sawako, Charles, Anne-Claire et Maeva de la cafétéria de l'Inria, et à l'équipe du restaurant de CentraleSupélec : merci pour votre bonne humeur de tous les jours.

Merci à mes ami·es de Lorient et à ma famille pour leur soutien, leurs encouragements et leur patience quand je m'exerçais à la vulgarisation de sujet de thèse.

Merci aux ami·es du master bioinformatique pour cette amitié que nous avons su garder malgré les éloignements, et pour la plupart, malgré les thèses que vous prépariez aussi. Je souhaite que l'on puisse exercer nos recherches en assumant nos responsabilités éthiques.

RÉSUMÉ EN FRANÇAIS

Séquençage ADN et Assemblage de génomes

La molécule d'ADN est le support universel de l'information génomique chez tous les organismes vivants. Étudier *in silico* l'information qu'elle porte permet de mettre en exergue l'effet de son expression sur le fonctionnement des organismes, de les classifier, de connaitre l'histoire qui lie les êtres vivants entre eux. La molécule d'ADN est double brin. Chaque brin est un long mot formé de quatre lettres, les nucléotides : A, C, G, T. Les deux brins sont complémentaires : un nucléotide A fait face à T, C fait face à G — et vice-versa. Les nucléotides vont par paires, appelées comme *paires de bases* (pb). De plus, les deux brins sont transcrits dans des sens opposés. Ils sont par conséquent *inverses-complémentaires*. Pour connaitre sa composition en nucléotides, la molécule d'ADN est fragmentée en plusieurs morceaux chevauchants lors du *séquençage*. Ces fragments, appelés *lectures* sont *assemblées* pour obtenir la séquence nucléotidique d'un seul brin, l'autre brin s'obtenant en inversant et en complémentant la séquence du premier.

Les technologies de séquençage ne permettent toujours pas de séquencer en une seule fois la totalité d'une molécule ADN quand elle excède une certaine taille. À la fin des années 70, les recherches de Sanger permettent de séquencer des fragments dont les tailles des lectures approchent le millier de paires de bases. Ce processus est toutefois long et couteux. Les technologies de séquençage connaissent un développement important lors de la première décennie du 21e siècle. Des millions de lectures sont produites en une seule exécution de séquençage. Elles sont courtes (de l'ordre de la centaine de paires de bases), et leur taux d'erreur de séquençage atteint au mieux 0,1 %, un ordre de grandeur en dessous de celui des technologies de Sanger. Depuis 2010, une troisième génération de technologies produit des longues lectures (de l'ordre de la dizaine de milliers de paires de bases) mais fortement bruitées (de quelques pourcents).

L'assemblage de génomes ou assemblage de fragments est le processus visant à reconstruire les plus longues sous-séquences des molécules ADN séquencées (sinon leur séquence entière) à partir de leurs lectures. Plusieurs facteurs rendent le problème de l'assemblage de fragments non trivial. Les erreurs de séquençage obligent à la correction au préalable des lectures, sinon à la production d'une séquence consensus à partir des lectures assemblées. Un plus grand nombre de courtes lectures augmente la consommation en mémoire et en temps des algorithmes d'assemblage. Les fragments séquencés proviennent de copies des deux brins. Par conséquent, chaque lecture doit être considérée sous ses deux orientations exclusives :

soit sa séquence inchangée participe à l'assemblage (orientation *directe*), soit sa séquence inverse-complémentaire (orientation *inverse*), sinon aucune des deux. Le plus difficile étant la résolution des régions répétées dans les génomes. Les occurrences d'une région répétée se distinguent en fonction des régions adjacentes. Plus les lectures sont longues, plus il est probable que les occurrences et leurs régions adjacentes soient couvertes. À l'inverse, si aucune lecture ne couvre entièrement les occurrences et leurs adjacences, leur ordre ne pourra pas être inféré. La résolution des régions répétées est ainsi fonction de la longueur des lectures et des longueurs des régions répétées – par exemple longues de plusieurs milliers de paires de bases dans le génome du mais.

Deux grandes étapes composent l'assemblage de fragments. (i) L'assemblage des lectures à partir de leurs chevauchements (le début d'une lecture s'aligne sur la fin d'une autre). Il en résulte un ensemble de plus longues séquences, les *contigs*, qui peuvent se retrouver en plusieurs copies dans le génome. (ii) L'*échafaudage* des contigs (*scaffolding* en anglais) consistant à les orienter et les ordonner pour former des *échafaudages* (*scaffolds* en anglais) grâce à des données supplémentaires. Les contigs d'un échafaudage peuvent être éloignés par une distance nucléotidique.

La thèse présentée ici s'est concentrée sur deux problématiques. La première porte sur l'analyse et la comparaison de structures de graphes utilisées tout au long des étapes de l'assemblage de fragments. Le deuxième consiste en une formulation originale du problème de l'échafaudage dédié aux génomes d'un organisme particulier, les *chloroplastes*, et à sa résolution.

Graphes de fragments ADN

Un graphe est une structure mathématique utilisée pour rendre compte de relations entre des objets. Ces derniers sont représentés par des *sommets* reliés par des *arêtes* (*graphe non orienté*) ou par des *arcs* (*graphe orienté*). L'assemblage des lectures s'appuie sur les chevauchements entre elles. Ainsi, il existe un lien permettant de passer d'une première lecture orientée vers une deuxième. De même, l'échafaudage des contigs dépend des liens entre deux contigs orientés. Ces liens proviennent souvent de paires de lectures séparées par une distance (connue ou non) s'alignant sur deux contigs (*paired-end reads* en anglais). Pour généraliser, nous utilisons le terme générique *fragment*. Dans le cas de l'assemblage de lectures, il désigne les lectures. Dans celui de l'échafaudage, il désigne les contigs. Dans ces deux cas, un graphe permet de représenter les liens entre deux fragments orientés.

Trois structures de graphes émergent de la littérature. La première représente chacune des deux orientations de chaque fragment par un sommet. Parce que les liens sont des paires ordonnées de fragments orientés, ils définissent alors des arcs dans un graphe orienté (GO). Kececioglu (1991) l'employa pour la première fois. Les

objets y semblent redondants, car les sommets vont par pairs (deux orientations), ainsi que les arcs (les liens peuvent aussi être munis d'une orientation). Myers (1995) propose de fusionner les deux orientations d'un fragment en un seul sommet, ainsi qu'un lien et son inverse en une seule arrête. L'arrête doit alors être pondérée par un couple d'orientations. La première correspond à celle du premier fragment, par exemple celui d'identifiant lexicographiquement inférieur. La seconde orientation est celle du fragment défini comme étant le deuxième. Le graphe est *biorienté* (GB). Enfin, pour leur méthode d'échafaudage, Huson et al. (2002) simplifient la représentation en associant deux sommets pour chaque fragment, un pour chacune des deux extrémités d'un fragment : la tête et la queue. Ces deux sommets sont reliés par une arête (*arrête-fragment*). Passez de la tête à la queue correspond à choisir le fragment inverse. Ici, les liens sont des arêtes reliant l'extrémité d'un fragment à un autre (*arrêtes-liens*). Un chemin valide dans ce graphe non orienté (GN) commence et termine par une arête-fragment et alterne entre les deux types d'arêtes.

L'emploi de l'une ou l'autre structure de graphe varie entre les méthodes pour les étapes d'assemblage de fragments. Bien que leurs différences de conceptions soient parfois décrites, ces structures n'ont jamais été comparées théoriquement ni en pratique. Aucune base d'implémentation pour GO, GB et GN pour comparaison n'a donc été proposée. En réponse, nous avons proposé des implémentations s'appuyant sur des listes d'adjacence qui tirent bénéfice de la symétrie impliquée par l'inverse-complémentarité des deux brins. Les implémentations sont comparées en terme d'usage théorique en mémoire, et au travers du coût temporel des algorithmes pour réaliser des opérations élémentaires : itérer sur les liens, ajouter et supprimer un fragment ou un lien.

Bien que les coûts des implémentations ne diffèrent que de facteurs linéaires, nous concluons sur quelques recommandations. En particulier, nous avons mis en évidence que GB est à privilégier si la mémoire fait défaut, puisque deux fois moins de pointeurs sont requis, ou quand il s'agit de supprimer des fragments du graphe. L'implémentation ne décrivant que les successeurs dans GO est la meilleure pour itérer sur les voisins d'un fragment orienté, ajouter ou supprimer un lien. Une implémentation de GO, qui ne garde que les voisins des fragments orientés directs, est disponible pour `Python3` sur PyPI sous le nom de `revsymg`.

Échafaudage de génomes de chloroplastes

Résultat d'endosymbioses entre des bactéries et des cellules eucaryotes, les chloroplastes sont des organites de cellules de plantes et d'algues. Ils produisent des molécules carbonées à partir du CO2 dans l'air grâce au processus biologique de la photosynthèse. Les chloroplastes possèdent leur propre génome en plus de celui de la

plante (dans le noyau) et de celui des autres organites (comme des mitochondries). Les plantes sont des êtres pluricellulaires, et chacune des cellules de leurs feuilles possèdent plusieurs chloroplastes.

La structure et la répartition du génome des chloroplastes sont particulières. La structure la plus étudiée est un génome circulaire quadriparti, séparée en quatre régions : une Longue Région Unique (LRU), une Courte Région Unique (CRU) et une paire de régions inverse-complémentaires (RI). Chaque chloroplaste possède plusieurs copies de son génome, qui peuvent différer en formes. La présence des RI entraîne l'inversion-complémentaire d'une des régions uniques lors de la réplication qui s'opère sur les deux brins à partir d'un point d'origine. Ainsi, des *haplotypes structuraux* réversibles coexistent dans un même chloroplaste.

Plusieurs approches ont déjà été proposées pour l'assemblage spécifique d'organites et de chloroplastes. Certaines sont des suites (*pipeline* en anglais) de méthodes génériques appliquées sur des données de séquençage filtrées (Ankenbrand et al., 2018). D'autres profitent de la faible taille des génomes de chloroplastes et ont développé des algorithmes de graine-et-extension (*seed-and-extend* en anglais) visant un assemblage circulaire (Coissac et al., 2016 ; Dierckxsens et al., 2017). Jin et al. (2020) estiment les multiplicités des contigs afin qu'elles soient au plus proche des observations des alignements des lectures sur les contigs et que le graphe de contigs multipliés contienne un chemin circulaire. Les chemins circulaires représentant des RI identiques sont marqués comme solutions. Seule cette approche et celle de Andonov et al. (2019) (un échafaudeur appliqué sur des données chloroplastiques) tiennent compte de solutions multiples correspondant à des haplotypes structuraux. Toutes ces approches ne modélisent pas explicitement l'assemblage ou l'échafaudage d'un génome de chloroplaste (et de ses différentes formes), mais procèdent à des filtres a priori et a posteriori des assemblages.

Ici, nous proposons de nous appuyer sur les caractéristiques des structures de leur génome pour formuler l'échafaudage de génome de chloroplaste en un problème d'optimisation combinatoire. Cette formulation se centre sur la reconstruction des régions répétées et des régions uniques, pour ensuite déduire les différentes formes de génomes. Nous démontrons que ce problème est $\mathcal{NP}$-complet. Nous le décomposons en trois sous-formulations (pour les RI, RU, et régions répétées directes), et pour chacune nous proposons un modèle linéaire en nombre entier. Ces modèles sont implémentés dans un unique programme qui les hiérarchise. Nos résultats sont encourageants sur des données synthétiques, choisies en fonction de la variété des difficultés génomiques structurales qu'elles présentent. La méthode est disponible sous le module PyPI `Python3` `khloraascaf`, et les résultats sont reproductibles (voir https://khloraascaf-results.readthedocs.io/en/latest/).

Conclusions et perspectives

L'assemblage de fragments est un problème abstrait difficile à formaliser dans le cas général. Découpé en sous-étapes, leurs approches de résolution dépendent de la nature des données en entrée et des spécificités des génomes des organismes à assembler. Les problèmes d'optimisation combinatoire sous-jacents gagnent en pertinence lorsque leur modélisation rend compte de ces complexités.

Le graphe est une structure mathématique utilisée tout le long du processus d'assemblage de fragments : des lectures jusqu'aux échafaudages. Trois structures émergent de la littérature. Bien qu'elles soient comparées dans leur conception abstraite, elles n'avaient jamais été comparées sur la base d'une implémentation de graphe. Nous proposons dans cette thèse une base commune de leur formalisation en proposant une implémentation sous la forme de listes d'adjacences. Le cout de stockage en mémoire et le coût temporel des algorithmes d'itération sur les données, d'ajout ou de suppression d'informations sont calculés pour chacune des implémentations. Une des implémentations est disponible via un module `Python3`.

Nous envisageons d'implémenter les graphes sous `Rust` ou sous un langage de programmation compilé équivalent pour comparer les implémentations expérimentalement et confirmer les calculs théoriques. De même, puisque le choix d'une structure de graphe influence les modélisations ultérieures pour l'assemblage de fragments, nous souhaitons proposer, pour chaque structure, des modèles linéaires en nombres entiers de recherches de chemins ou de couvertures, les analyser et les comparer en théorie et en pratique.

Nous nous sommes ensuite penchés sur le problème de l'échafaudage dans le contexte des génomes de chloroplastes. La communauté travaille depuis maintenant une vingtaine d'année sur le problème de l'échafaudage dans le cas général, et depuis une dizaine d'années particulièrement sur les génomes de chloroplastes. Les méthodes diffèrent grandement sur les heuristiques utilisées ainsi que sur les stratégies de sous-échantillonnage des données concernant les outils les plus complets. L'intérêt du génome de chloroplaste pour son assemblage réside dans sa structure génomique particulière et dans l'existence de plusieurs formes au sein d'un même organite. Sa petite taille de génome se traduit par un jeu de données dont la taille permet de résoudre le problème de l'échafaudage par des méthodes exactes. Ainsi, nous posons une formulation du problème que nous résolvons au moyen d'une décomposition en sous-problèmes, modélisés et résolus par programmation linéaire en nombres entiers. À travers elle, nous couvrons de nombreuses spécificités du génome de chloroplaste, de l'agencement des régions génomiques jusqu'à l'existence d'haplotypes structuraux. Notre méthode produit des solutions multiples corrélées avec le phénomène d'inversion-complémentaire de certaines régions au cours de la réplication ADN. Enfin, nous avons implémenté ces

approches en un module `Python3`, testé sur des données synthétiques aux difficultés hétérogènes. Les résultats sont reproductibles.

L'échafaudage n'étant qu'une sous-étape du processus d'assemblage, nous souhaitons injecter notre approche dans une suite permettant d'assembler les génomes de chloroplastes à partir des lectures. Dans un souci de comparaison pertinente, nous envisageons d'injecter notre partie dans la suite `GetOrganelle`, à l'endroit où l'on identifie une équivalence d'objectif, entrées et sorties. Cela nous permettra de nous comparer avec l'état de l'art. Les chloroplastes ne sont pas les seuls organites en présence dans les cellules de plantes. Les mitochondries par exemple partagent des caractéristiques génomiques communes avec celles des chloroplastes. Nous envisageons d'investiguer dans quelle mesure notre approche peut-être adaptée pour l'assemblage de génomes mitochondriaux.

CONTENTS

Remerciements i

Résumé en français v

Séquençage ADN et Assemblage de génomes v
Graphes de fragments ADN vi
Échafaudage de génomes de chloroplastes vii
Conclusions et perspectives ix

Contents xi

List of Figures xvii

List of Tables xix

I Introduction 1

1 DNA overview 2
1.1 A recipe for cellular mechanisms 3
1.2 On the motivations for knowing the whole DNA sequence 3
1.3 Various molecular conformations structure the genome 4
2 Chloroplast genome 4
2.1 Genome division 5
2.2 Repeats and single-copy regions 5
2.3 Structural haplotypes 7
2.4 Genome evolution 8

3 DNA sequencing 8
3.1 The Shotgun sequencing approach 9
3.2 First generation: Sanger and BAC technologies 9
3.3 Second generation: high-throughput sequencing 10
3.4 Third generation: single molecule run time 11
3.5 Supplementary sequencing data 12
4 The challenges of fragment assembly 13
4.1 Read sequence alignment 13
4.2 Unknown fragment orientations 14
4.3 Sequence similarity: single-copy or repeat? 14
4.4 True sequence divergences or sequencing errors? 15
5 Fragment assembly approaches 15
5.1 The Shortest Common Superstring 15
5.2 Overlap-Layout-Consensus 17
5.3 De Bruijn Graph approach 18
5.4 Breaking down the fragment assembly problem 20
6 Addressed research topics 21
6.1 Graph structure for read assembly and scaffolding stages 22
6.2 Scaffolding of chloroplast structural haplotypes 22

II State-of-the-art 23

1 Graph structure for fragment assembly 24
1.1 Notations and fundamental definitions 24
1.2 Directed graph (DG): oriented fragments based 27
1.3 Bidirected graph (BG): oriented walk based 30
1.4 Undirected graph (UG): tail-head fragments based. 33
2 Scaffolding the contigs 37
2.1 Scaffolding input data 38
2.2 Subsampling the input data 40
2.3 Orienting the contigs 44
2.4 Ordering the oriented contigs 46
2.5 Orienting and ordering the contigs simultaneously 47
2.6 Solving approaches 49
3 Chloroplast genome assembly 51
3.1 Chloroplast sequence extraction 51
3.2 Chloroplast reads assembly 53

3.3 Chloroplast scaffolding 54
3.4 Chloroplast assembly validation 55

III Fragment graph implementations and comparison 57

1 Implementations 58
1.1 Directed graph (DG): oriented fragments based 59
1.2 Bidirected graph (BG): oriented walk based 66
1.3 Undirected graph (UG): tail-head fragments based 68
1.4 Fragment graph map 70
2 Algorithms for DGS, DGF and BGU 72
2.1 Subfunctions 72
2.2 Iterating over the predecessors 73
2.3 Iterating over the successors 74
2.4 Adding a vertex 75
2.5 Adding an edge 76
2.6 Deleting a vertex 78
2.7 Deleting an edge 81
3 Time costs 82
4 Memory and time cost comparisons 84
5 Conclusions and perspectives 85

IV Global exact optimisations for chloroplast structural haplotype scaffolding 87

1 Introduction 88
1.1 Chloroplast genome specificities 88
1.2 State-of-the-art 89
1.3 Our approach 89
2 Input data and notation 90
2.1 Set of contigs $\mathcal{C}$ 90
2.2 Set of links $\mathcal{L}$ 91
2.3 Mathematically defining genomic regions 91
3 Chloroplast scaffolding problem formulations 92
4 Graph and repeated fragment sets 96
4.1 Graph structure 97

4.2 Repeated fragment sets . . . 98
5 Integer Linear Programming (ILP) formulation . . . 102
5.1 Circuit constraints . . . 103
5.2 Repeated regions constraints . . . 104
5.3 Fixing regions constraints . . . 107
5.4 Speed-up constraints . . . 108
5.5 Scaffolding problems ILP . . . 108
6 Hierarchical problem succession . . . 109
7 From an ILP solution to a genome structure . . . 110
8 Multiple genome forms . . . 114
9 $\mathcal{NP}$-completeness . . . 117
10 Numerical results . . . 121
10.1 Complexity validation on artificial data . . . 122
10.2 Synthetic chloroplast input data . . . 123
11 Conclusion . . . 130
12 Discussion and perspectives . . . 131

V Conclusions and perspectives 133

Fragment graph . . . 134
Thesis contribution . . . 134
Short-term future work . . . 135
Long-term future work . . . 135
Chloroplast genome scaffolding . . . 136
Thesis contribution . . . 136
Short-term future work . . . 137
Long-term future work . . . 138

Bibliography 139

Appendix A1

1 Repeated fragment set functions . . . A1

2 Reduction of the repeated fragment sets A2
2.1 Repeated fragment set reductions A2
2.2 Pairs of repeated fragment set reductions A5
2.3 Adjacent repeated fragment set reductions A8
3 Metrics A17
3.1 Quast metrics A17
4 Supplementary results A18
4.1 v1 scaffolding benchmark A18
4.2 v2 scaffolding benchmark A20

Acronyms G1

Symbols G3

Computational terms G7

Glossary G11

LIST OF FIGURES

I Introduction 2

1 DNA double-strand molecule 3
2 Different DNA conformations 4
3 Simplified DNA distribution in a plant's cell. 5
4 DNA repeats. 6
5 Common chloroplast genome's architectures. 7
6 Inverted repeats causing structural haplotypes: flip-flop inversion during the DNA replication. 8
7 The shotgun sequencing approach. 10
8 Sequencing error types. 11
9 Paired-end sequencing. 12
10 Sequence alignments. 14
11 Unknown fragment's orientation. 15
12 Impact of the genome repeats on the fragment sequencing. 16
13 OLC versus DBG fragment assembly approaches. 19

II State-of-the-art 26

14 Examples of links 26
15 Fragment and link sets with functions 27
16 Link cases in DG. 28
17 A path in DG. 29
18 Link cases in BG. 31
19 A path in BG. 33
20 Link cases in UG. 35
21 A path in UG. 36
22 Overview of all the fragment graph structures. 37

23 Scaffolding input data. 41
24 Bidirected and undirected graph cycles. 45

III Fragment graph implementations and comparison 59

25 DGA implementation. 60
26 DGS implementation. 62
27 DGF implementation. 64
28 BGU implementation. 67
29 UGA implementation. 69
30 Graph structures and their implementations. 70

IV Global exact optimisations for chloroplast structural haplotype scaffolding 93

31 Repeat degeneration and region orders. 93
32 Chloroplast repeat scaffolding. 96
33 $MDCG$ example. 97
34 Repeated fragment sets illustration for two contigs c and d. 99
35 Adjacent repeated fragment sets examples. 101
36 Non-exhaustive illustrations for authorised and forbidden order cases for two repeated fragments $((i, j), (k, l)) \in PRepF$. 105
37 Extracting the genome architecture in $MDCG$ from a $\mathcal{CHSP}$ solution. 111
38 Region graph for the toy example. 115
39 From a digraph G for $LPSTP$ to a digraph G' for $\mathcal{IRP}$. 120
40 Solver running time distributions for perfect artificial data. 123

LIST OF TABLES

II State-of-the-art 38

1 Non-exhaustive categorised list of methods and approaches for the scaffolding stage. 39
2 Scaffolding input data properties. 40
3 Chloroplast genome assembly approaches. 52

III Fragment graph implementations and comparison 82

4 Calculus detail of basic operations costs. 83
5 Algorithmic costs for subfunctions. 83
6 Algorithmic costs of iterating over the neighbours. 83
7 Algorithmic costs of adding a vertex or an edge. 84
8 Algorithmic costs of deleting a vertex. 84
9 Algorithmic costs of deleting an edge. 84
10 Comparison between DGS, DGF and BGU. 85

IV Global exact optimisations for chloroplast structural haplotype scaffolding 90

11 Toy example of input data. 90
12 ILP sets and functions corresponding table. 105
13 Problem code combinations 110
14 `Gurobi` solver metrics on perfect artificial growing data. 124
15 `Gurobi` solver metrics on noisy artificial growing data. 125
16 Sequence and `Quast` metrics for the initial synthetic data version. 128

17 Sequence and Quast metrics for the modified synthetic data version. 130

Appendix A18

18 Benchmark 3 v1 contig Quast A18
19 Benchmark 3 v1 ILP stats A19
20 Benchmark 3 v2 ILP stats A20

I INTRODUCTION

Joe Hisaishi. (1999). Inner Voyage [Song]. On *The Universe Within, Vol.1 & 2*. PONY CANYON

In this chapter

1 DNA overview 2
- 1.1 A recipe for cellular mechanisms . . . 3
- 1.2 On the motivations for knowing the whole DNA sequence . . . 3
- 1.3 Various molecular conformations structure the genome . . . 4

2 Chloroplast genome 4
- 2.1 Genome division . . . 5
- 2.2 Repeats and single-copy regions . . . 5
- 2.3 Structural haplotypes . . . 7
- 2.4 Genome evolution . . . 8

3 DNA sequencing 8
- 3.1 The Shotgun sequencing approach . . . 9
- 3.2 First generation: Sanger and BAC technologies . . . 9
- 3.3 Second generation: high-throughput sequencing . . . 10
- 3.4 Third generation: single molecule run time . . . 11
- 3.5 Supplementary sequencing data . . . 12

4 The challenges of fragment assembly 13
- 4.1 Read sequence alignment . . . 13
- 4.2 Unknown fragment orientations . . . 14
- 4.3 Sequence similarity: single-copy or repeat? . . . 14
- 4.4 True sequence divergences or sequencing errors? . . . 15

5 Fragment assembly approaches 15
- 5.1 The Shortest Common Superstring . . . 15
- 5.2 Overlap-Layout-Consensus . . . 17
- 5.3 De Bruijn Graph approach . . . 18
- 5.4 Breaking down the fragment assembly problem . . . 20

6 Addressed research topics 21
6.1 Graph structure for read assembly and scaffolding stages 22
6.2 Scaffolding of chloroplast structural haplotypes 22

This chapter presents general and basic knowledge on the subject of *genome* assembly. Deoxyribonucleic acid (DNA) and genomic information studies have known great development since the 20th century. They impact many areas, from ecological understanding to economics, through agronomics and medicine.

We first give a quick overview on what DNA is and the information the molecule holds in Section 1. Section 2 describes the particularities of the *chloroplast* genome we exploit to formalise their assembly in Chapter IV. Then, we describe *sequencing* methods to obtain a DNA sequence through its fragmentation (Section 3) and the challenges it raises (Section 4). Section 5 gives the background of the approaches enabling to retrieve the sequence from *fragments*. Finally, we address the topics this thesis focuses on in Section 6.

1 DNA overview

DNA molecules are the support of the genomic information contained in each cell of living organisms. Studies in DNA have been developed since the beginning of the 50s, when X-ray pictures shot by Gosling while he was supervised by Franklin (Franklin and Gosling, 1953) lead to the double-strand model proposed by Watson and Crick in 1953. Each *strand* is a sequence of four *nucleotides*, adenine (A), cytosine (C), guanine (G) and thymine (T) that are organic molecules considered as elementary subunits.

We denote by $\Sigma_{nuc} = \{A, C, G, T\}$ the nucleotide alphabet, where A and T are complementary nucleotides as well as C and G. Each strand has a reading direction from the defined 5' extremity to the 3' one. One strand is the complement of the other, and the 5' 3' extremities are reversed. As an example, if $AATGCCA$ is a strand (or a DNA sequence), then $TGGCATT$ is its reverse-complement as illustrated in Figure 1. For the sake of clarity, the reverse-complement word is shortened to the *reverse*.

As the DNA molecule is double-stranded, the nucleotides are said to be paired, and the length of a DNA molecule is measured in base pairs (bp). The DNA molecules among the living organisms vary in forms and in length: from millions of base pairs for bacteria (Trevors, 1996) to billions for eukaryotes (Kidwell, 2002). This double-strand specificity contributes to chemical stability as well as enabling replication and repairing mechanisms. These mechanisms are the key to sharing genomic information through generations.

$$\begin{array}{ccc} 5' & \longrightarrow & 3' \\ & AATGCCA & \\ & TTACGGT & \\ 3' & \longleftarrow & 5' \end{array}$$

Figure 1 – DNA double-strand molecule
5' and 3' extremities give the reading direction of each DNA strand, illustrated by the two arrows. One strand is the reverse-complement (or the reverse) of the other. Thus, $AATGCCA$ is the reverse of $TGGCATT$, and vice-versa. This DNA molecule's length is $7\,\mathrm{bp}$.

1.1 A recipe for cellular mechanisms

DNA operates as both an archive and a recipe for cellular machinery, through hierarchical processes of synthesis. The DNA molecule can be divided into gene regions separated by intergenic regions. Particularly, enzymes read the DNA molecule from the 5' extremity to the 3' one, and synthesise RNA single-strand molecules from the genes (transcription stage). RNA molecules are then translated to proteins that can form protein-complexes performing different functions such as the catalytic function of enzymes. Especially during the translation stage of the RNA (denoted mRNA, standing for "messenger RNA") the nucleotide alphabet is translated to the amino-acid alphabet, where three consecutive nucleotides correspond to one amino-acid. Specific molecular biology research fields focus on each molecule and the synthesis processes. Various types of RNA cover different functions: messenger, participating in the translation stage, silencing genes etc.

1.2 On the motivations for knowing the whole DNA sequence

Uniquely studying proteins is insufficient to understand all the cellular mechanisms and their influences on phenotypes. Studying how DNA transitions into mRNA or tRNA completes knowledge of the DNA's 3D behaviour, helping researchers to understand life processes. Similarly, gene sequence analysis alone cannot fully explain the functioning of living organisms. Whole genome sequencing provides supplementary results compared to protein, gene or RNA studies. The interested reader is referred to the review by Rice and Green (2019) from which the following derives.

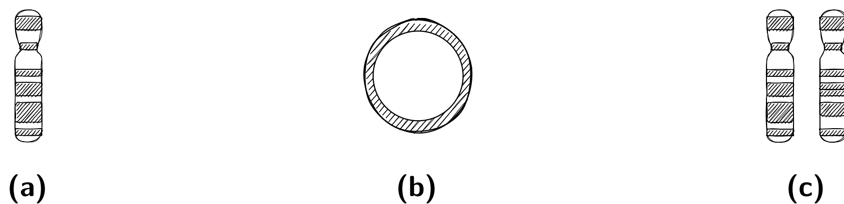
Without specifically studying the function of the genomic regions, a whole genome can serve as a reference sequence on which other sequences are aligned or mapped in order to infer history, timing or location of events based on edition distances. Gene expression (reading a gene results in the production of a mRNA) can be affected over large genomic regions: a gene can be silenced or over-expressed due to inhibitor/enhancer genomic regions. Indeed, while it is expected that gene expression is affected by genomic regions near to the gene in the DNA sequence, these

regions can also be near in space because of DNA molecular folding mechanisms. Some phenotypes are multifactorial corresponding to polygenic traits (e.g. gene co-expression), and it can be valuable to know the location order of the gene of interest. Also related to gene expressions, during the meiosis stage (cell division of germ cells for the production of gametes) *chromosome* recombination sometimes occur. Finally, changes in chromosomes can result in the emergence of new species.

Therefore, having the complete genome sequence influences biochemistry, immunology, evolution and ecological sciences.

1.3 Various molecular conformations structure the genome

The DNA or genomic information is differently organised across living organisms. The genomic information can be organised along one molecule or split into several ones. It can also be multiplied in the genomic container (e.g. in the cell's nucleus for eukaryotes). When multiple copies coexist in the container, they can be grouped: they are paired for humans (diploid species), or grouped by more than two, e.g. by 6-uplet as for the wheat (hexaploid). The DNA molecules supporting the whole or a part of the genomic information also differ in topology: they can be linear (as for humans) or circular (as for the *Escherichia coli* bacteria). Figure 2 illustrates divers DNA molecule conformations.



(a) (b) (c)

Figure 2 – Different DNA conformations
Each figure represents chromosomes, which are compressed DNA (double-stranded) molecules. $a\text{x} = b$ means that b chromosomes are grouped in a-uplet: **(a)** haploid linear DNA $1\text{x} = 1$; **(b)** haploid circular DNA $1\text{x} = 1$; **(c)** diploid linear DNA $2\text{x} = 2$.

2 Chloroplast genome

Chloroplasts are plant and algae cell *organelles*. They are the result of endosymbiosis between bacteria and eukaryotic cells (Gould, 2012). They produce carbon compounds from CO2 in the air through the photosynthesis biological process. *Mitochondria* are other organelles in endosymbiosis with plants and algae. In

1986, the first chloroplast genomes were sequenced and assembled. Those are chloroplasts of a liverwort, *Marchantia polymorpha* (Ohyama et al., 1986), and of a tobacco plant, *Nicotiana tabacum* (Shinozaki et al., 1986). Knowing the sequence of organelle genomes allows performing ecological, phylogenetic and evolutionary studies (Daniell et al., 2016; Sun et al., 2020; Long et al., 2023).

2.1 Genome division

Plants are multicellular organisms. As eukaryotes, the genomic information of the plants is hosted in the nucleus of their cells, noted nucDNA. Chloroplasts and mitochondria, possess their own genomic material (respectively cpDNA and mtDNA), and each plant's cell can host several organelles possessing the same genomic material. Furthermore, each chloroplast has multiple copies of its genome (Bendich, 1987; Kobayashi et al., 2002). The number of copies have been shown to decrease with the leaf age (Kumar et al., 2014). Figure 3 schematises the genome distribution in a plant cell.

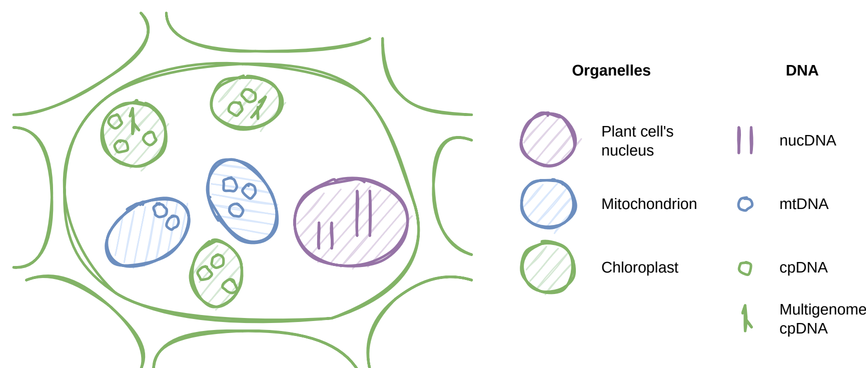


Figure 3 – Simplified DNA distribution in a plant's cell.
The grey octagon container represents a plant's cell. The purple, blue and green oval shapes are respectively the plant cell's nucleus, mitochondria and chloroplasts. There are several mitochondria and chloroplasts in a same cell. Each of them possesses its own genomic material: in the nucleus, for this artificial example, the plant is diploid ($2x = 4$); each mitochondrion has several copies of its genome; each chloroplast has multiple forms of its own genome, linear, circular and multigenomes.

2.2 Repeats and single-copy regions

Genomic regions can be classified according to whether their sequence is found approximately elsewhere in the genome, regardless of their functions. In this case,

the classification depends on two main factors:

— the minimum length of a sequence to consider it as a repeat (otherwise, a sequence of size one could be said to be repeated);

— an acceptable number of word editions (such as substitutions, insertions and deletions) to pass from one sequence to another.

The difficulty with these factors lies in the fact that they are left to the discretion of the researchers analysing them, and therefore suffer from a certain subjectivity. For the chloroplast we will focus on two types of repeats.

Direct Repeat (DR) The sequences are highly similar;

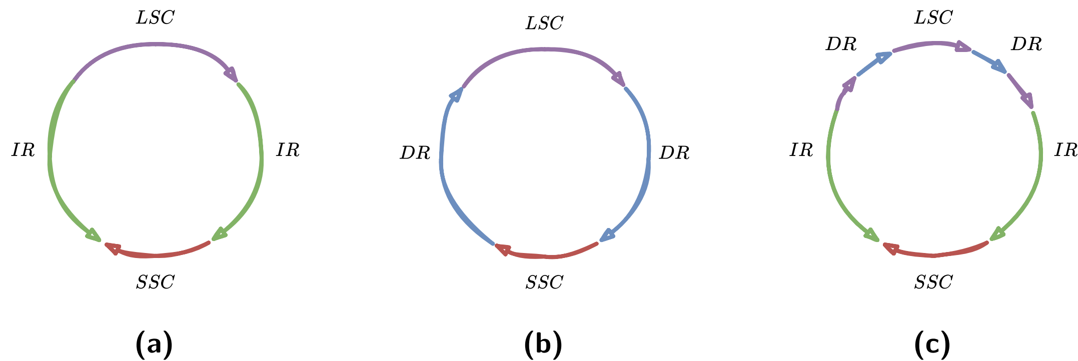
Inverted Repeat (IR) One sequence is the reverse of the other.

Figure 4 gives two examples of perfect repeats.

(a) Direct Repeat (DR) **(b)** Inverted Repeat (IR)

Figure 4 – DNA repeats.
A blue subsequence is the reverse-complement of a green subsequence, i.e. $GCAA$ is the reverse of $TTGC$. **(a)** $GCAA$ is direct repeated in the top strand, thus $TTGC$ is direct repeated in the bottom strand. **(b)** $GCAA$ is inverted repeated in the top strand, thus $TTGC$ is inverted repeated in the bottom strand.

One of the most studied chloroplast genome architectures is the circular DNA molecule whose quadripartite sequence includes a pair of highly similar (or identical) reversed nucleotide subsequences (Inverted Repeat (IR)) separated by a Long Single-Copy (LSC) and a Short Single-Copy (SSC) sequences (Bock and Knoop, 2012). Figure 5a illustrates IR-quadripartite chloroplast genome. For example, Thode et al. (2021) show that IRs in *Mikania* plastomes have length around 25 kbp. However, an example of the difficulties in discriminating what is a repeat is highlighted by Kim and Lee (2005). The latter studies quadripartite chloroplast genomes described as above, where inversions can be found in LSC because of small IRs. The structures are still described as quadripartite because the IRs in the LSC are smaller than the IRs considered in the structure. Quadripartite structures with Direct Repeat (DR), generally shorter than the IRs, can also be found (Palmer, 1985), as well as more complex structure involving both type of repeats (Tsai and Strauss, 1989). Figure 5 summarises the main chloroplast genome structures found in the literature.



(a) **(b)** **(c)**

Figure 5 – Common chloroplast genome's architectures.
The most studied chloroplast genome form is circular and very often quadripartite. For each of the figures (a), (b) and (c), coloured arrows represent nucleotide sequences. LSC and SSC stand for Long Single-Copy and Short Single-Copy (purple and red), respectively. They correspond to regions (subsequences) that are not repeated in the genome. On the opposite, IR and DR stand for Inverted Repeat and Direct Repeat (green and blue), respectively. **(a)** This architecture is the most common one and is defined as a quadripartite architecture. The two green IR arrows face each other that illustrates one is the reverse-complement sequence of the other; **(b)** the two blue DR arrows are in the same direction that illustrates both have the same nucleotide sequence; **(c)** the two types of repeat can simultaneously exist in the chloroplast genome, and DRs are shorter than IRs.

2.3 Structural haplotypes

The molecular form of the copies of chloroplast genomes differ (Bendich, 2004), but the circular form has been mainly studied. For example Seyer et al. (1981) give an electron micrograph of a circular cpDNA from *Nicotiana tabacum*. Although the circular structures are a minority, and multigenomes can be found (linear genomes combined in a tree-like structure due to recombination-dependent replication), the circularity property is less sensitive to virus interactions and thus is dominantly transmitted to the next generation (Bendich, 2004).

For this reason, in this manuscript we focus on the chloroplast circular genome copies. During the DNA molecular replication of one cpDNA molecule, flip-flop inversions can occur so that the resulting replicated molecules' form differs. This transformation is due to the presence of IRs, where the DNA subsequence between them is reversed. Inversions during the replication phase were also studied for bacteria, and they are considered as reversible operations. The copies of a genome, exact or obtained by inversions, are denoted as *structural haplotypes* (Palmer, 1983; Deng et al., 1989; Wang and Lanfear, 2019). Figure 6 schematises a flip-flop

inversion for a chloroplast genome with IRs.

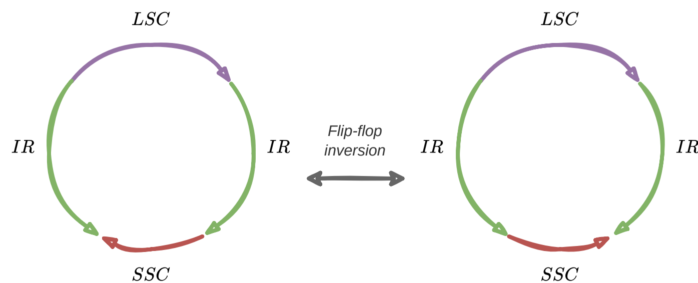


Figure 6 – Inverted repeats causing structural haplotypes: flip-flop inversion during the DNA replication.
During the DNA replication of the chloroplast genome, one of the region between the inverted repeats can be reversed (here the red region SSC). This provokes the existence of several forms of the genome in the same chloroplast (*heteroplasmy*).

2.4 Genome evolution

Chloroplasts and mitochondria in endosymbiosis with eukaryotic cells have gradually become the organelles of plant cells. As a result, organelles no longer need specific genomic traits, leading to a loss of genes and a reduction of their genome size (Xiao-Ming et al., 2017). Horizontal transfers (a transfer of genes not by heredity) were also studied from the plant to its organelles.

3 DNA sequencing

DNA sequencing aims to obtain the nucleotide sequence of a DNA molecule for downstream analyses. The sequence of only one DNA strand is sufficient because retrieving its complementary is immediate (c.f. Figure 1). Since the 1960s, it has been necessary to sequence subpart of molecules, corresponding to a few nucleotides. Although fragment lengths in 2023 could exceed the megabase range, sequencing the entire DNA molecule of most living organisms in one run remains unfeasible.

In some papers presenting sequencing results, the term sequencing refers both to the genome fragmentation and the assembly of the fragments. Here we separate the two procedures. Sequencing is the process of fragmenting DNA and obtaining their sequences using a *sequencer*. The *fragment assembly* solves the fragment puzzle with computational methods.

In the late 1960s, the available techniques were restricted to employing digestion enzymes which split the sequence at specific base positions. As a result, they output very short fragments, denoted as digests, sometimes of length two and generally limited to length eight. Sequencing DNA molecules of several kilobases was therefore impossible, and the digest sequencing technology was limited to sequence protein or RNA sequences.

3.1 The Shotgun sequencing approach

Since the 1970s, sequencing technologies have known continuous development and fast improvement leading to various data types. In the following, a *read* refers to a sequence obtained by these technologies, while a fragment refers to a partial or a whole sequence of a DNA molecule. The main approach to sequence a genome is proceeding to a shotgun sequencing: the reads are obtained from the sequencing of fragments randomly sampled from copies of the genome. This strategy was firstly proposed by Staden (1979) and is illustrated in Figure 7. It is based on the fact that a fragment from a copy of the genome may *overlap* fragments from other copies. Since the reads are subsequences of the fragments, they also overlap, enabling to retrieve a continuous sequence.

The next sections summarise the main sequencing technologies, inspired by the reviews written in Mehdi et al. (2017) and Pervez et al. (2022).

3.2 First generation: Sanger and BAC technologies

In the late 1970s Sanger's research succeeded in sequencing much longer fragments (Sanger et al., 1977). Their length increases from 8 bp up to one thousand (400 bp to 900 bp). The process is denoted by Sequencing By Synthesis (SBS): it consists in completing a single-strand DNA molecules from a fragment (a template) with denatured nucleotides. Each nucleotide addition is marked along a gel thanks to electrophoresis. The gel can be seen as an $n \times 4$ boolean matrix, where n is the fragment's length and each of the four columns corresponds to one nucleotide. If there is a band at the cell (i, j), then there is the jth nucleotide at the ith position of the fragment.

The Bacterial Artificial Chromosome (BAC) benefits from a bacterial DNA molecule as a vector to amplify inserted fragments of interest (O'Connor et al., 1989). Venter et al. (1996) suggested sequencing them at both ends. Thanks to adaptors at both extremities of the fragments, two reads are sequenced from the 5' extremity of the two complementary strands (Denoted as paired-end data, in opposition to single-end, c.f. Figure 9). As the length of the vector is known, BAC technology provides an approximated distance between the two paired-read.

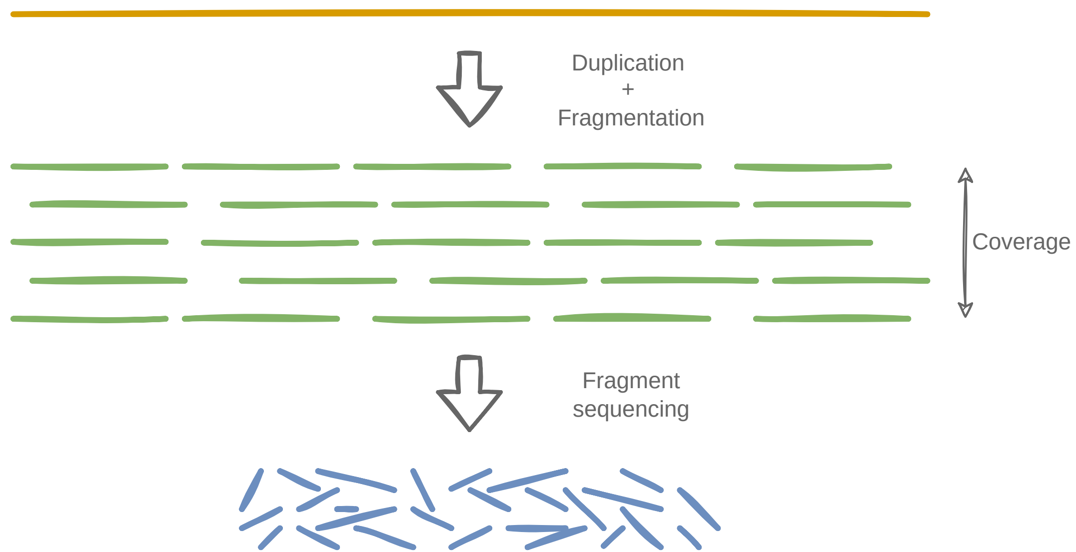


Figure 7 – The shotgun sequencing approach.
From the bottom to the top of the figure: the reads (in blue) are sequenced from longer fragment (in green) randomly sampled from copies of the genome (in orange). Based on the overlapping reads, the assembly process aims to retrieve the genome. The coverage counts the average number of times a base in the genome is covered by the reads.

Although the fragments are long and the accuracy can reach 99.99 %, the Sanger technology suffers from low throughput and high costs. Since the beginning of the 21$^{\text{th}}$ century it has been supplanted by high-throughput technologies, even if it is still used for validating DNA sequence and for target resequencing.

3.3 Second generation: high-throughput sequencing

As mentioned above, the first decade of the 21$^{\text{st}}$ century witnessed the emergence of new sequencing technologies. Refered to as Next Generation Sequencing technologies (NGS), they produce millions of short reads at low prices allowing the sequencing of numerous organisms.

Illumina sequencing (2000) The Illumina/Solexa sequencing technology is also an SBS technology. Adaptors are attached to the extremities of the double-strand fragments. The two strands are separated, the adaptors find their complementary on a flow cell, and the single-strand fragments are then amplified by Polymerase Chain Reaction (PCR). As modified nucleotides complete the strands, a laser emits

$ACGTCAT$ $ACGTCAT$ $ACGTCAT$

$ACGTCGT$ $ACGTCAAT$ $ACGTC_T$

(a) (b) (c)

Figure 8 – Sequencing error types.
Three types of errors can occur during the sequencing of DNA fragments. For each subfigure, the top sequence is the original/template sequence and the bottom sequence is the sequenced one. **(a)** Substitution: one nucleotide is changed to another one; **(b)** insertion: there is one extra nucleotide; **(c)** deletion: there is one missing nucleotide. The first error type is someone noted as a mismatch, and the two latters are categorised as indels.

a light signal that is detected by a camera and interpreted by a computer. The best machines are able to sequence millions of reads at a reduced cost, by providing a 0.1 % to 1 % error rate in favour of substitutions (Figure 8a). However, the read lengths are in the range of 100 bp to 300 bp.

454 sequencing (2005) Similarly to Illumina technology, the 454 sequencing is labelled SBS, it needs PCR amplification and nucleotide additions are detected by light emission. It produces reads with lengths from 100 bp up to 700 bp with a 1 % error rate mainly caused by homopolymers errors. Homopolymers are repetitions of the same nucleotide, and their misdetection leads to insertions and deletions (indels) in the reads (Figures 8b and 8c).

Ion Torrent (2011) SBS technology, it measures the PH variation to detect nucleotides. It produces reads with lengths from 200 bp up to 400 bp, with 1 % error type mainly due to indels.

Paired-end data Analogously to the BAC technology, it is possible to sequence short reads by pairs. For Illumina, the distances are short (100 bp – 300 bp), while for mate-pair data (circularised fragments generating paired reads, Korbel et al. 2007), they reach the kilobases at the expense of accuracy.

3.4 Third generation: single molecule run time

The Third Generation Sequencing technologies (TGS) refer to Single Molecule Run Time technologies (SMRT) that have emerged in the last decades. They produce fewer but longer reads without PCR. The procedure is shorter and less costly.

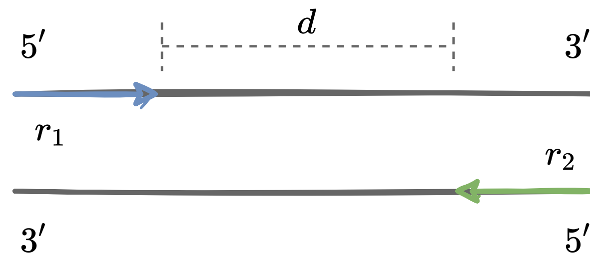


Figure 9 – Paired-end sequencing.
The two main grey lines are the two strands of a fragment. The two reads r_1 and r_2 are sequenced by pairs distanced by d nucleotides (inner distance).

Although they first suffered from relatively high error rate with indel error types, these technologies have made significant advances in lowering the error rate in recent years.

Pacific biosciences SMRT sequencing (2010) Each DNA fragment passes into a cell that contains a DNA polymerase. When the single-strand fragment passes through the polymerase, fluorescent labelled nucleotides complete the strand and release light signals. The average read length is 10 kbp and can reach 60 kbp. The first error rate was measured to 13 % but more recent HiFi technology would decrease the rate to 0.5 % (according to PacBio, Hon et al. 2020).

Oxford nanopore sequencing (2014) In Oxford Nanopore Technology (ONT), the DNA fragments pass though a pore (made by a protein complex). The passage of the molecule through the pore generates a variation in the ionic current. The variations are recorded and translated to the nucleotide alphabet. The average read length is 10 kbp and can reach 150 kbp, and even 4 Mbp for PromethION machine. Because controlling the speed at which the fragment enters the pore is challenging, the initial error rates are approximately 12 %. According to the firm, they can obtain reads with less than 1 % error rate. However, Delahaye and Nicolas (2021) measure it to 5 % – 8 % but remark that the base-caller (the software interpreting ionic signals) is upgrading fast.

3.5 Supplementary sequencing data

The following new data type can help to assemble, finish or correct an assembly of the reads, but at a supplementary cost.

10x linked read 10x linked read is a sequencing technology permitting to label the reads that come from the same fragment. It generates short reads that can be assembled by groups.

Hi-C protocol Hi-C is a technology highlighting physical contact between two regions in the genomes. It generates Illumina paired reads coming from two close genomic regions due to chromosome conformation. While two close regions during one conformation can be far away from each other along the genome, it is possible to prevent the compaction of the DNA in such a way that contacts only occur between two regions close to each other in the sequence. Such contacts are not very accurate and provide a square matrix (each side the length of the genome) with a strong diagonal and a lot of noise around.

Optical map Finally, optical mapping is a process of partitioning copies of the genome at specific known subsequence site thanks to nicking enzyme. For each copy a specific enzyme is associated. As a result, the positions of several subsequences are known, which helps for mapping and ordering fragments along the genome.

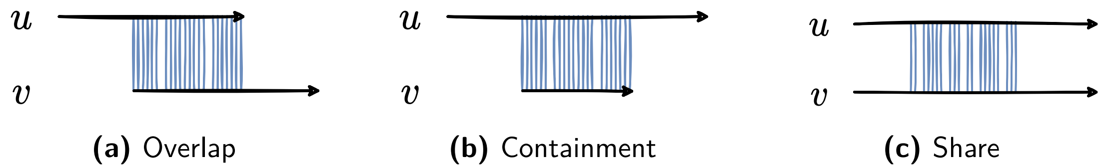
4 The challenges of fragment assembly

The fragment assembly problem began in the late 1960s with the assembly of digests. As the fragments are just a few base lengths (up to 8 bp) it raises the issue of the unicity of their assembly. Shapiro (1967) proposes an algorithm for RNA and protein assembly based on the digest overlap, and gives sufficient conditions for the unicity. Since 1977, more fragments are produced thanks to the Sanger sequencing technology, and has become necessary to use computers to assemble them. The first assemblers were proposed by Gingeras et al. (1979) and Staden (1980). They enable the first shotgun assembly project with the assembly of the 50 kbp virus λ using reads of length 200 bp (Sanger et al., 1982).

4.1 Read sequence alignment

Read sequences can be compared by sequence alignment algorithms. Figure 10 schematises three types of read alignment. (a) Overlapping sequences in different reads enable the algorithm to reconstruct the original DNA sequence. However, because of sequencing errors, two reads sequenced at the same genome region can differ in sequence, so it is necessary to find approximated overlaps. (b) Reads that are contained in others are useful to correct the sequencing errors by producing a consensus from the alignments. (c) Reads sharing a part of their sequence, and

whose alignment does not fall into the previous categories, can be considered as detecting repeats because the flanking regions are not aligned.



(a) Overlap **(b)** Containment **(c)** Share

Figure 10 – Sequence alignments.
Sequence alignments distributed into three categories. u and v are two sequences. The arrows give their orientation (tail: 3', head: 5'). The blue lines between them correspond to aligned nucleotides. **(a)** Overlap: the head of u is aligned with the tail of v. **(b)** Containment: u is containing v. **(c)** Share: a subpart of u is aligned with a subpart of v, and at least one of the subpart does not begin from or finish to an extremity (here each subpart respects the statement).

4.2 Unknown fragment orientations

As the DNA is double-stranded, and the reads are sequenced from DNA fragments, two reads can be sequenced from the two different strands. This introduces the notion of sequence orientations. Each read must be considered in two orientations: the one given by the technologies (defined as the *forward* orientation), and its reverse-complement (defined as the reverse orientation). As a consequence, (i) both the direct and its reverse must be considered when searching for read overlaps and (ii) at most one orientation must be chosen for the reads when assembling them. Figure 11 illustrates the fragment orientation issue.

4.3 Sequence similarity: single-copy or repeat?

The main challenge of genome assembly is the genome repeats resolution. The definition of a repeat depends on the length of the reads and the similarity between the repeats in the genome. Indeed, if reads are longer than the repeats, there are some reads that contains them with their non identical flanking sequences. Note that some repeats can exceed kilobases (e.g. long transposable elements in the maize genome or in bacteria genomes Kidwell 2002). Raising the threshold to accept overlaps is not sufficient as some repeats are very similar in sequence, and because of sequencing error the alignments must be flexible such as not losing true overlaps. Figure 12 illustrates the genome repeats impact on the reads' assembly in

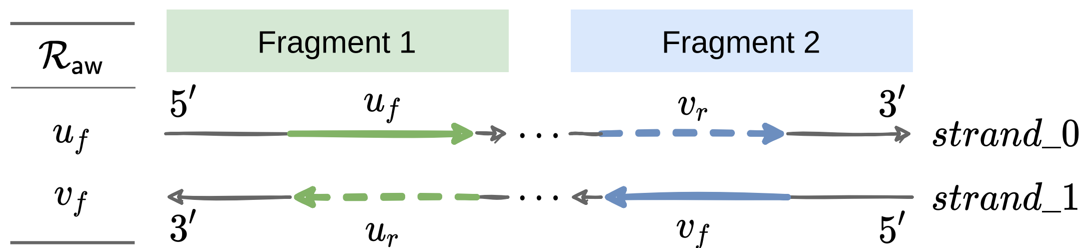


Figure 11 – Unknown fragment's orientation.
$\mathcal{R}_{aw}$ is the set of sequenced reads. Reads u_f and v_f have been sequenced from two different fragments of the same genome, but they do not come from the same strand. u_r and v_r are respectively the reverse of u_f and v_f. An assembly should either consider u_f followed by v_r, or v_f followed by u_r.

the case of two direct repeats. Furthermore, the overlapping of the reads must also take into account inverted repeats because both orientations must be considered.

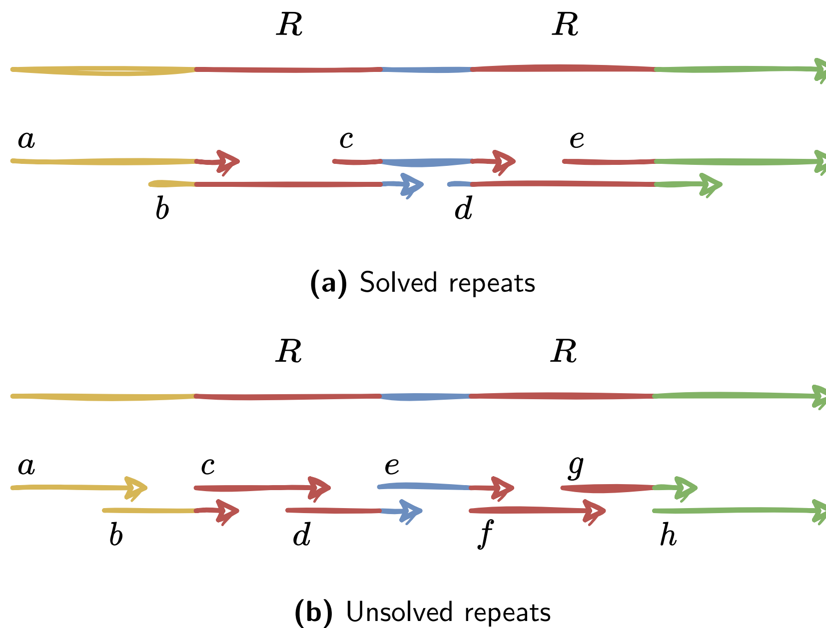
4.4 True sequence divergences or sequencing errors?

As previously mentioned, the repeats can slightly differ in sequence. Sequencing errors can hide the signal of a true difference. Although a substitution can be resolved by consensus, indel errors are more difficult to solve, and have the worse consequence as they can shift the protein reading phase (three nucleotides are translated into one amino acid). It is thus necessary to have a sufficient sequencing depth to counterbalance the errors. These errors are supposed to be randomly distributed along the genome (which is not really the case, e.g. for ONT). Identifying true biological signals is a prior step to further separating heterozygotes regions for polyploid organisms or splitting close ones shared by different species in metagenomic samples.

5 Fragment assembly approaches

5.1 The Shortest Common Superstring

Peltola et al. (1984) tackle the problem of fragment assembly with the Shortest Common Superstring (SCS) algorithm. Given an error ratio δ, each read must participate in the superstring F, in only one orientation, modified in sequence with an edit distance less than δ times the length of the read. F must be as short as possible, and this makes the problem mathematically non-trivial, although it is not



(a) Solved repeats

(b) Unsolved repeats

Figure 12 – Impact of the genome repeats on the fragment sequencing. The two subfigures show the same genome sequenced respectively with long and short reads. The genome possesses a repeated region R (in red) and three single-copies (resp. yellow, blue and green). **(a)** Reads b and d are covering the repeats entirely: the assembly gives $abcde$. **(b)** Reads c and f are subsequence of the repeats: the assembly can give $abcdefgh$, or $abfdecgh$ (if the repeats are not exactly the same, switching c and f can misassemble the genome), or even $abcgh$ that shortens the genome.

necessarily a biologically meaningful requirement (parsimony principle).

Assembling the fragments by choosing the longest overlaps first gives a heuristic to solve the SCS but does not guarantee the optimal solution. Räihä and Ukkonen (1981) have previously modelled SCS as the search for a Hamiltonian path in a *directed graph (digraph)* that represents overlaps. In that case, the fragment assembly problem is $\mathcal{NP}$-hard.

Li (1990) provided hypotheses that support the SCS modelling: the DNA molecule is assumed to be a random nucleotide sequence and the reads are uniformly sampled. However, these hypotheses do not reflect genomic complexity. Finally, Myers (1995) found that the SCS can wrongly make the genome shorter by merging the repeats.

5.2 Overlap-Layout-Consensus

To address the sequencing errors and the repeat issues, Kececioglu and Myers introduced the Overlap-Layout-Consensus (OLC) approach. It consists of three stages: (i) computing approximated overlaps between the reads; (ii) finding a layout on the overlaps to partition them into parts classified as single-copy and repeated regions; (iii) producing consensus sequences for each part.

The overlaps computed in stage (i) can be represented in a digraph denoted as the *overlap graph*. Kececioglu (1991) defined it as $G = (V, E)$, where V is the set of oriented reads and E is the set of arcs (u, v), u and v are two overlapping oriented reads (Figure 13b).

The fragment orientation and the maximum-weight dovetail-chain branching problems The premise formulations are given in Kececioglu (1991) and Kececioglu and Myers (1995). At this time, the orientation of the reads and the ordering of the oriented reads are separated in two global optimisation formulations: first, choose a subset of the overlaps that maximises the sum of their lengths and assign only one orientation for each read; then, find a non-branching *path* that maximises the overlap length. The authors observed that with known orientations and without errors, the latter corresponds to the SCS.

The string graph formulation To overcome the SCS shortened repeats, Myers (1995) proposed the *maximum likelihood* ϵ*-valid layout* formulation. It consists in preserving the read coverage along the assembly. Indeed, the coverage rate of compressed repeat regions should be abnormally higher than that of single-copy regions in the assembly. This leads to the *string graph* layout: the overlap graph is transformed to a string graph. (i) The contained reads are removed from V (criticised in Jain 2023); (ii) the graph is simplified by transitive *edge* removal (in pseudo-linear time complexity, Myers 2005); (iii) the single paths are collapsed. It results in a string graph for which, theoretically, edges correspond to genomic regions and the *vertices* are the junctions between them. Each edge is weighted by the number of times it may participate in a walk representing the genome. `Celera` implemented this approach and enabled to assemble the genome of *Drosophila melanogaster* (Myers et al., 2000).

Local and variant approaches The string graph formulation has inherited many variants. Some of them are focused on data cleaning strategies and successive local corrections, as introduced by `ARACHNE` (Batzoglou et al., 2002) for BAC data, and done in `Canu` (Koren et al. 2017, `Celera`'s successor) for TGS data. Some adopt aggressive strategies such as the Best Overlap Graph (BOG), which keeps only the

highest-quality extending overlap for each orientation of each read. Introduced in CABOG, a software dedicated to Roche 454 and Illumina paired-end reads (Miller et al., 2008), the BOG strategy is part of hifiasm (Cheng et al., 2021) which is dedicated to haplotype resolving thanks to HiFi long reads.

OLC complexity In Myers (2005), a valid assembly of the genome is equivalent to finding a cyclic walk respecting the number of times it passes through each edge in the string graph. According to the parsimony principle, this walk minimises the genome length the assembly produces. In this case, Medvedev et al. (2007) proved the problem to be $\mathcal{NP}$-hard. Nevertheless, due to the plethoras of heuristics in the current methods (e.g. in Canu and hifiasm), the actual complexity of the algorithms is highly correlated to the computation of the overlaps, a quadratic procedure.

5.3 De Bruijn Graph approach

New approaches have emerged to handle the millions of reads produced by NGS. Sequencing-by-Hybridisation (SBH) was the name of a left-behind sequencing technology that indicated the presence of sequences of size k among a set in the genome. It inspired the De Bruijn Graph (DBG) assembly approach. In 1989, Pevzner introduced the use of a partial *de Bruijn graph (dbgraph)* to assemble the genome. This approach first decomposes each read in words of length k (*k-mers*) overlapping by $k-1$ nucleotides. Then it builds a dbgraph $G = (V, E)$ of all the k-mers of all the reads is built: each vertex in V corresponds to a $k-1$ overlap between two k-mers, and each edge in E is a k-mer. Each read corresponds to a walk in G, and the genome corresponds to a superwalk (a walk of walks) (Figure 13c).

As for the OLC approach, it consists in reducing G: (i) Each path of vertices that do not have more than one in-edge and one out-edge are collapsed in one edge (*unitig*); (ii) branching vertices are resolved with read mapping on the paths. Idury and Waterman (1995) associated the genome as a double-Eulerian superwalk in G. If the Eulerian superwalk is unique, it should correspond to the genome. The first motivation was the analogy with the search for an Eulerian path, which is simpler than the search for a Hamiltonian path. Furthermore, it reduces the consensus phase to find paths through the k-mers.

Size of k and solid k-mers However, as the overlaps between the k-mers are exact, G is sensible to sequencing errors. In fact, each substitution in a k-mer forms a bubble (two paths starting from and ending to the same k-mers). For a small value of k, the graph is more connected but more branching, while for a larger value the graph branches out less but suffers from bubbles and can be partitioned into several

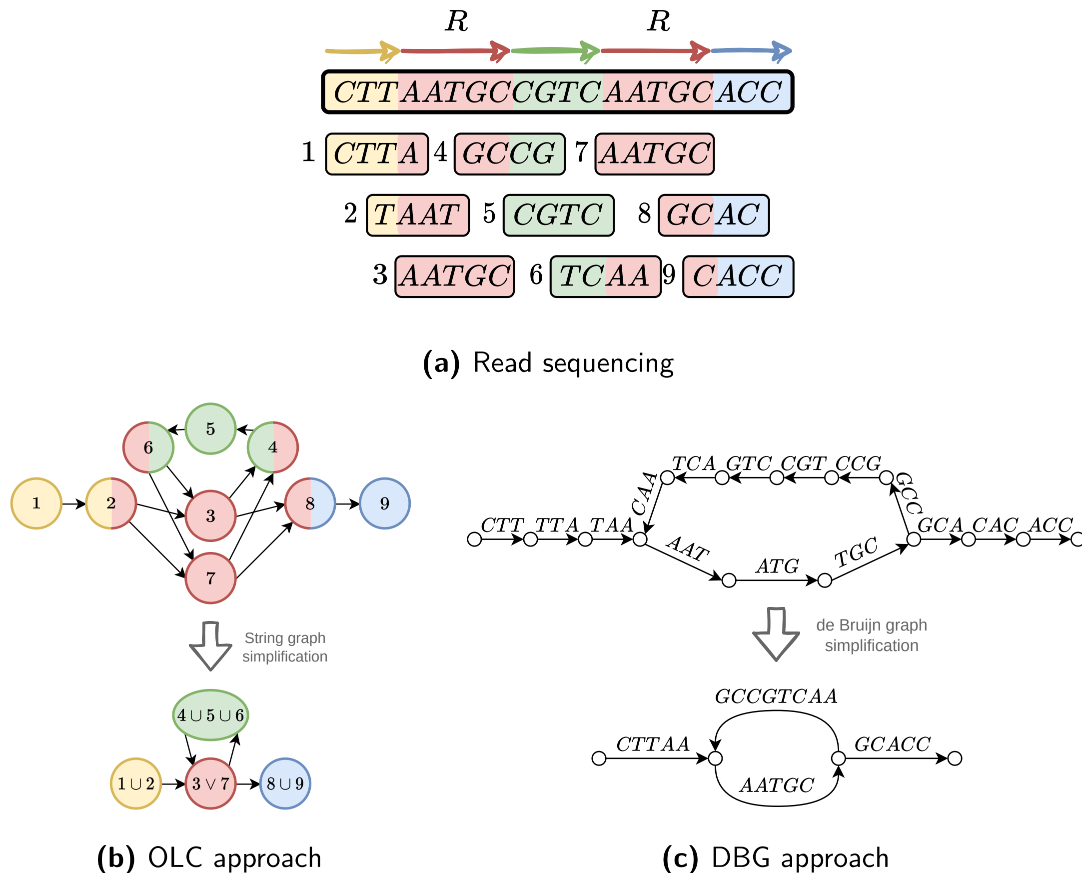


Figure 13 – OLC versus DBG fragment assembly approaches.
(a) The genome possesses a repeated region R. The sequencing produces nine reads. For the example, reads 3 and 7 are exactly the same and are contained in R. **(b)** From the overlap of the read (at least two bases), the string graph is built. Note that neither read 3 nor read 7 are removed because no one is longer than the other while they are mutually contained each other. **(c)** The reads are split into $k-1$ overlapping words of length k (k-mers). In the (partial) edge-centric dbgraph, vertices are the $k-1$ exact overlaps and edges are the k-mers. Here $k=3$. For **(b)** and **(c)**: unique paths are collapsed to form unitigs. The over-simplified illustrations for the OLC and respectively the DBG approaches suggest that the genome can be found by solving a Hamiltonian path through the simplified string graph resp. an Eulerian path through the simplified dbgraph.

connected components. One strategy is to compute the number of occurrences a k-mer appears in the set of reads, and only use those associated with a sufficiently large occurrence value to build the graph (solid k-mer, Pevzner et al. 2001).

Local and variant approaches Many DBG variant algorithms have implemented their own heuristic for specific targets. `Velvet` and `SOAPdenovo` merge bubbles thanks to a Dijkstra-like Breadth-First Search (BFS) algorithm (Zerbino and Birney, 2008; Li et al., 2010). `SPAdes` applies the DBG approach with different values of k and iteratively merges the results (Bankevich et al., 2012). While the latter software are suitable for short paired-end reads, the DBG principle inspires methods for long erroneous reads. `Flye` first generates a draft assembly without considering misassemblies, and detects repeats by aligning the draft against itself (Kolmogorov et al., 2019). While `wtdbg2` follows the OLC approach, it does not compute the overlap precisely: it partitions the reads in K groups (bin) of 256 bp (K-bin) so that two K-bin are the same if they share enough k-mer (Ruan and Li, 2020). Finally, `mdBG`, for Minimiser dbgraph, represents the reads as sequence of minimiser (in a minimiser alphabet), that are a subset of all possible k-mers (Ekim et al., 2021).

DBG complexity Pevzner (1989) has proposed two dbgraphs: the one described in this section is edge-centric (the k-mers are the edges), while the other one is vertex-centric (the k-mers are the vertices). Pevzner abandoned the latter because he associated it with the Hamiltonian path problem that is $\mathcal{NP}$-complete and for which no algorithm had been proposed. On the other hand, finding an Eulerian path can be done in linear time on the number of vertices and edges under the Euler hypothesis on even degrees (Hierholzer and Wiener, 1873). However, DBG corresponds to finding a super-walk. Medvedev and Brudno (2009) associate DBG with the Chinese postman problem (that echoes the SCS approach) on bidirected graphs, and propose an exact polynomial time algorithm. They next link DBG to a maximum-likelihood problem to avoid compressing the repeat at the opposite of the SCS, similar to the Myers' propositions concerning the near-constant coverage of the reads on the assembly. In the latter case, they prove DBG to be in $\mathcal{P}$.

5.4 Breaking down the fragment assembly problem

Because of sequencing errors and the difficulties of defining repeats in OLC or dbgraphs, formulating the fragment assembly problem as a global optimisation problem sounds unreachable. The literature suggests splitting the fragment assembly problem into several stages. The development of stage-dedicated methods and software has enabled researchers to create their pipelines depending on the available sequencing data. Here we present a list of hierarchical stages:

Filtering and correcting the sequencing data Reads associated with a low sequencing quality score can be partially or entirely removed from the read set. Multiple align-

ment techniques can also produce a self-correction of the reads before assembling them.

Assembling the reads into contigs *Contigs* are nucleotide sequences longer than the read. While unitigs correspond to unambiguous paths in the string and the dbgraphs, contigs result from resolving branches thanks to read alignment or paired-end data.

Scaffolding the contigs *Scaffolds* are sequences of oriented contigs with potential gaps between them. In Chapter II, Section 2, we describe several approaches for orienting and ordering the contigs.

Gap-filling the scaffolds The gaps between the oriented contig in each scaffold are filled thanks to overlapping reads joining the two contigs' extremities.

Haplotyping Generating consensus sequence on unitig in the OLC approach does not handle heterozygosity for polyploid organisms. Similarly, it does not address the difference between near-genome regions shared between two organisms in metagenomic samples. Read alignment can differentiate a sequencing error from a true signal. Finally, each group is assigned to homologous chromosomes or specific organisms (phasing). Concerning the DBG approach, the separation can use k-mer counting between the paths in bubbles.

Generating a consensus sequence Multiple alignment is the main approach to produce consensus sequences. The final nucleotide bases result from a weighted vote.

Evaluating the assembly The assembly evaluations focus on the number of contigs, their length distribution and their cumulative length. A reference genome enables mapping the contigs to measure mismatch and indel mapping error rates. It may be necessary to split the contigs in order to compare them to the reference: the number of breakpoints denotes the number of misassemblies. One can also compute the number of genes present in the assembly.

6 Addressed research topics

In this thesis we focus mainly on two aspects of genome assembly.

6.1 Graph structure for read assembly and scaffolding stages

Chapter II, Section 1, and Chapter III concern *graph* data structure for storing and iterating over *links* between oriented DNA fragments. In the *read assembly* stage, the links are the overlaps between the reads in the OLC approach (overlap graph). In the *scaffolding* stage, they often correspond to paired-end information between two reads, translated to link between oriented contigs (scaffolding graph). The overlap graph or the scaffolding graph are fundamental data structure in read assembly and scaffolding stages. Furthermore, their descriptions and their representations influence methods.

In the literature, the graph structures are described mathematically in the best cases, very often commented and illustrated, and sometimes suffer from confused description. To the best of our knowledge, no one has compared the impact of the different representations and especially their implementations.

In this thesis we analyse different representations and propose suitable implementations that we organise in a graph and implementation map. We then design associated algorithms to iterate over the vertices and the edges, and to dynamically maintain them.

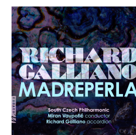
6.2 Scaffolding of chloroplast structural haplotypes

Chapter II, Sections 2 and 3, and Chapter IV focus on the assembly and the scaffolding of chloroplast genomes. We first summarise the scaffolding problem formulations and strategies in the general case, mixed with the read assembly stage or as independent stages. We then describe the whole fragment assembly process approaches dedicated to chloroplast genome from the literature.

As we consider that the dedicated approach does not appropriately handle the particularities of the chloroplast genome structure, we translate the biological knowledge into mathematical properties. We then formulate the scaffolding problem in the case of chloroplasts and model it as a global optimisation problem. We prove the decision version of the problem to be $\mathcal{NP}$-complete. Furthermore, our formulation enables us to tackle the presence of chloroplast structural haplotypes. Finally, we have implemented our approach and have tested it with synthetic data in order to measure the time complexity in practice and the robustness.

II STATE-OF-THE-ART

Richard Galliano & Miran Vaupotic & Czech Philharmonic Quartet. (2023). Petite Suite Française: III. Espiègle [Song]. On *Madreperla*. Navona Records



In this chapter

1 Graph structure for fragment assembly 24
1.1 Notations and fundamental definitions 24
1.1.1 Fragment set 24
1.1.2 Link set 26
1.2 Directed graph (DG): oriented fragments based 27
1.3 Bidirected graph (BG): oriented walk based 30
1.4 Undirected graph (UG): tail-head fragments based 33
2 Scaffolding the contigs 37
2.1 Scaffolding input data 38
2.2 Subsampling the input data 40
2.2.1 Bundling the links 42
2.2.2 Removing paired-end reads 42
2.2.3 Removing contigs 42
2.2.4 Removing links 43
2.2.5 Partitioning the instances 44
2.3 Orienting the contigs 44
2.3.1 Maximising the sum of used link weights 45
2.3.2 Remove the minimum number of contigs and links to avoid odd reversal cycles 46
2.3.3 Maximum log-likelihood function 46
2.4 Ordering the oriented contigs 46
2.4.1 Maximising the sum of used distances bunches' weights 46
2.4.2 Minimum spanning tree 47
2.4.3 Contig positioning only 47
2.5 Orienting and ordering the contigs simultaneously 47
2.5.1 Maximising the sum of distance bundles' weights to order linearly the contigs 47
2.5.2 The heaviest matching, paths and cycles 48

2.5.3 The maximum (weighted) matching and spanning tree 48
2.6 Solving approaches 49
2.6.1 Greedy approaches 49
2.6.2 Fix parameters dynamic programming 50
2.6.3 Mathematical programming 50
3 Chloroplast genome assembly 51
3.1 Chloroplast sequence extraction 51
3.1.1 Filtering the reads 52
3.1.2 Filtering the contigs 53
3.2 Chloroplast reads assembly 53
3.2.1 De Bruijn graph approach 53
3.2.2 Seed-and-extend 54
3.3 Chloroplast scaffolding 54
3.4 Chloroplast assembly validation 55

1 Graph structure for fragment assembly

Succession relationships between oriented reads are the keys to assembling them into contigs. Those between oriented contigs are necessary to assemble them into scaffolds. Those between oriented scaffolds enable a chromosome-scale assembly. All over the fragment assembly process, from the read assembly to the gap-filling through the scaffolding stages, it is fundamental to use suitable data structures to store and query these succession relationships between oriented fragments.

Most of the OLC approaches for the read assembly stage, and most of the scaffolding methods use graph structures to address the above issues. In this section, we present the underlying graph structures. We denote the graphs representing the fragments by abstraction of their sequence as fragment graphs, at the opposite of the dbgraph for which the vertices and the edges represent words. For example, two fragments with identical sequences may have several vertices in the fragment graph, while they will be compressed in the same path in the dbgraph.

1.1 Notations and fundamental definitions

1.1.1 Fragment set

A DNA fragment can either be a read resulting from the sequencing, a contig from the assembly of the reads, or a scaffold from the scaffolding of the contigs. The below definition serves to rigorously define an oriented fragment afterwards.

▶ Definition 1.1: Unoriented fragment set

Denote by $\mathcal{F}_{un}$ the set of unoriented fragments, in bijection with the set of integer label $\Sigma_{\mathcal{F}_{un}} = [\![0, |\mathcal{F}_{un}|[\![$, and denote by $unfid\colon \mathcal{F}_{un} \hookrightarrow\!\!\!\!\rightarrow \Sigma_{\mathcal{F}_{un}}$ the labelling function. Let $\Sigma_{nuc} = \{A, T, G, C\}$ be the nucleotide alphabet and $unseq\colon \mathcal{F}_{un} \to {\Sigma_{nuc}}^+$ the function that associates a fragment with its nucleotide sequence.

Because of the double-strand sequencing, illustrated in Figure 11, each fragment must be considered in two orientations: the original nucleotide sequence (forward), and the reverse-complemented nucleotide sequence (reverse):

▶ Definition 1.2: Fragment orientation set

Denote by $\{f, r\}$ the set of orientations, where f and r stand for the forward and the reverse orientation, respectively. Those orientations are complementary. The overline function $\overline{\cdot}$ gives the complementary orientation (its reverse), i.e. $\overline{f} = r$ and $\overline{r} = f$.

The set of oriented fragments results from the Cartesian product of $\mathcal{F}_{un}$ and $\{f, r\}$:

▶ Definition 1.3: Oriented fragment sets

Let $\mathcal{F} = \mathcal{F}_{un} \times \{f, r\} = \mathcal{F}_f \sqcup \mathcal{F}_r$ be the set of oriented fragments, associated with the projection functions $\pi_{\mathcal{F}_{un}}\colon \mathcal{F} \twoheadrightarrow \mathcal{F}_{un}$ and $for\colon \mathcal{F} \twoheadrightarrow \{f, r\}$, $\Sigma_{\mathcal{F}} = [\![0, 2|\mathcal{F}_{un}|[\![$ be the set of fragment integer identifiers and $fid\colon \mathcal{F} \hookrightarrow\!\!\!\!\rightarrow \Sigma_{\mathcal{F}}$ be the associated labelling function. Let denote by $seq\colon \mathcal{F} \to {\Sigma_{nuc}}^+$ the function that gives the nucleotide sequence of an oriented fragment. It holds that:

— $\mathcal{F}_f$ is the set of forward fragments, such that for each $a_f \in \mathcal{F}_f$:

- $for(a_f) = f$;
- $seq(a_f) = unseq(\pi_{\mathcal{F}_{un}}(a_f))$.

— $\mathcal{F}_r$ is the set of reverse fragments, such that for each $a_r \in \mathcal{F}_r$:

- $for(a_r) = r$;
- $seq(a_r) = \overline{unseq(\pi_{\mathcal{F}_{un}}(a_r))}$.

Where $\overline{\cdot}$ gives the reverse-complement of a given nucleotide sequence.

Unless stated otherwise, we will use the term fragment to denote an oriented fragment and precise when talking about an unoriented one.

▶ Definition 1.4: The fragment reverse operation

For a given fragment $a \in \mathcal{F}$, $\overline{a}$ denotes its reverse, where:

— $\left|fid(\overline{a}) - fid(a)\right| = 1$;

— $for(\overline{a}) = \overline{for(a)}$ (a and $\overline{a}$ have complementary orientations in $\{f, r\}$);

— $seq(\overline{a}) = \overline{seq(a)}$ (the nucleotide sequence of $\overline{a}$ is the reverse of this of a).

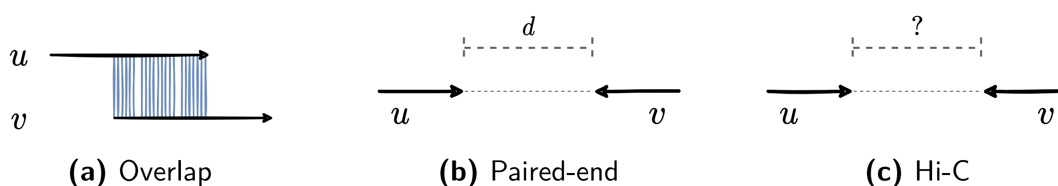
Figure 15 illustrates the fragment sets and their associated functions.

1.1.2 Link set

We denote by the links the succession relationships between the oriented fragments. Figure 14 gives three examples of links exploited in the fragment assembly stages.

▶ Definition 1.5: Link set

Denote by $\mathcal{L} = (\mathcal{L}', m_{\mathcal{L}})$ the multiset of ordered pairs of fragments, where $\mathcal{L}' \subset \mathcal{F}^2$ is a finite set, and $m_{\mathcal{L}} \colon \mathcal{L}' \to \mathbb{N}_{>0}$ gives the multiplicity of each element of $\mathcal{L}'$ in $\mathcal{L}$. Also let $linkp \colon \mathcal{L} \twoheadrightarrow \mathcal{L}'$ be the function that gives the fragment couple associated with a link. Denote by $\Sigma_{\mathcal{L}} = [\![0, |\mathcal{L}|[\![$ the set of link integer labels and its associated function $lid \colon \mathcal{L} \hookrightarrow\!\!\!\!\rightarrow \Sigma_{\mathcal{L}}$.



(a) Overlap **(b)** Paired-end **(c)** Hi-C

Figure 14 – Examples of links
Three examples of links. u and v are two fragments. The arrows give their orientation (tail: 5', head: 3'). **(a)** The head of u overlaps the tail of v. **(b)** Paired-end: u is linked with v, separated by a distance approximately of d nucleotides. **(c)** Hi-C: u is linked with v, separated by an unknown distance.

▶ Property 1.1: Link reverse symmetry

One link between two fragments implies one link between the reversed fragments:

$$\forall (u, v) \in \mathcal{F}^2,\ m_{\mathcal{L}}((u, v)) = m_{\mathcal{L}}((\overline{v}, \overline{u}))$$

i.e. $(u, v) \in \mathcal{L}' \iff (\overline{v}, \overline{u}) \in \mathcal{L}'$

Analogously to Definition 1.4, Definition 1.6 extends the reverse operation to the links:

▶ Definition 1.6: The reverse operation for links

For a given link $l \in \mathcal{L}$, where $linkp(l) = (u, v)$, $\bar{l}$ denotes its reverse such that $linkp(\bar{l}) = (\overline{v}, \overline{u}) = \overline{(u, v)} = \overline{linkp(l)}$.

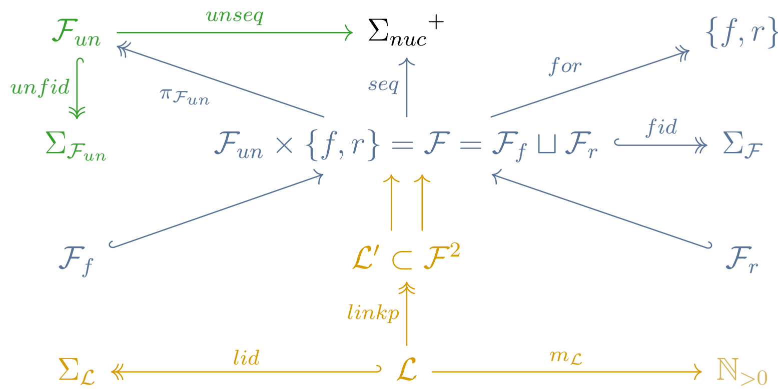


Figure 15 – Fragment and link sets with functions
Top left, in green: unoriented fragment sets. **Middle, in blue:** oriented fragment sets. **Bottom, in yellow:** Link (multi)sets.

Figure 15 illustrates the link sets and their associated functions. In Sections 1.2 to 1.4 we describe the three graph structures for the fragments and their links. They are loopless multigraphs.

1.2 Directed graph (DG): oriented fragments based

The directed graph (digraph) structure (DG) was the first idea to represent fragments and their links. As far as we know, this structure was first mentioned by Kececioglu (1991).

For the read assembly stage, DG is described in `FALCON` (Chin et al., 2016), `HINGE` (Kamath et al., 2017), `Shasta` (Shafin et al., 2020) and `hifiasm` (Cheng et al., 2021). Concerning the scaffolding stage, `GAT` refers to an unitig graph (Andonov et al., 2019).

▶ Definition 1.7: Oriented fragments based directed multigraph (DG)

Let $DG = (V, E, \Phi)$ be a *directed multigraph (multidigraph)* such that:

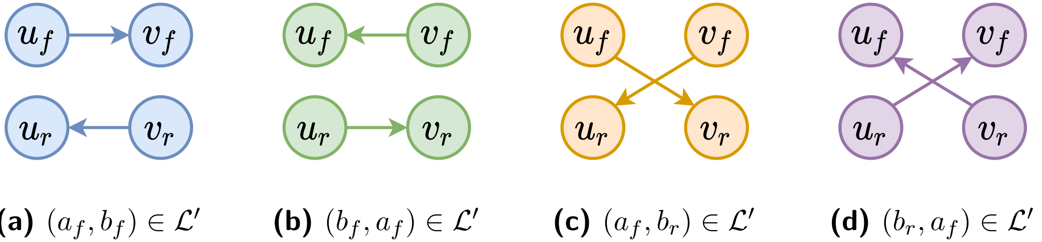
— V is the set of vertices, where each $v \in V$ represents one of the two orientations for a fragment in $\mathcal{F}$;

— E is the multiset of edges, where each $e \in E$ represents one link in $\mathcal{L}$;

— $\Phi\colon E \to \{(u, v) \mid u, v \in V, u \neq v\}$ is the incidence function that associates an edge with an ordered pair of vertices.

Denote by $frag\colon V \hookrightarrow\!\!\!\!\rightarrow \mathcal{F}$ the bijective function that associates each vertex with one and only one fragment, and by $link\colon E \hookrightarrow\!\!\!\!\rightarrow \mathcal{L}$ the bijective function that associates each edge with one and only one link.

Figure 16 illustrates the link case representations in DG.



(a) $(a_f, b_f) \in \mathcal{L}'$ **(b)** $(b_f, a_f) \in \mathcal{L}'$ **(c)** $(a_f, b_r) \in \mathcal{L}'$ **(d)** $(b_r, a_f) \in \mathcal{L}'$

Figure 16 – Link cases in DG.
Fragments a_f, b_f and their reverse a_r, b_r can link in four manners. Their associated vertices are u_f, v_f, u_r and v_r, respectively. Each subfigure illustrates one link in DG.

▶ Property 1.2: Sizes of DG

For $DG = (V, E, \Phi)$, $|V| = |\mathcal{F}|$ and $|E| = |\mathcal{L}|$.

Since one vertex is associated with only one fragment, Definition 1.8 extends the reverse operations given in Definitions 1.4 and 1.6 for DG.

▶ **Definition 1.8: The reverse operation in DG**

For each vertex $v \in V$ and edge $e \in E$, $\overline{v} \in V$ and $\overline{e} \in E$ denote their reverse. From the bijective properties of functions $frag$ and $link$, it holds that:

— $frag(\overline{v}) = \overline{frag(v)}$;

— $link(\overline{e}) = \overline{link(e)}$.

Definition 1.9 specifies the conditions for a valid path in DG.

▶ **Definition 1.9: Path in DG**

A path $p = (v_0, v_1, \ldots, v_{n-1}) \in V^n, n \in \mathbb{N}$ in $DG = (V, E, \Phi)$ is valid if the following two properties hold:

Contiguity

$$\forall\, i \in [\![0, n-1[\![, \exists\, e \in E \mid \Phi(e) = (v_i, v_{i+1})$$

Exclusive orientation

$$\forall\, v \in p, \overline{v} \notin p$$

Note that a path in DG satisfies the same properties as for a generic multidigraph, except that a vertex and its reverse cannot both participate in the path. Figure 17 gives an example of a valid path in DG.

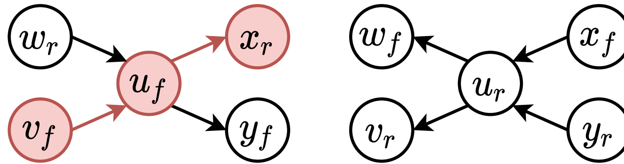


Figure 17 – A path in DG.
$p = v_f u_f x_r$ is a valid path coloured in red. $\overline{p} = x_f u_r v_r$ is its reverse and is also valid. However, p and $\overline{p}$ are exclusive.

Finally, Property 1.3 defines the reverse symmetry in DG, that inherits from Property 1.1 and illustrated in Figure 16.

▶ **Property 1.3: Reverse symmetry in DG**

— $\forall\, a \in \mathcal{F}, \exists!\, v \in V \mid frag(v) = a.$[a]

— $\forall\, l \in \mathcal{L}, \exists!\, e \in E \mid link(e) = l.$

[a] $\exists!\, x \in X$ means "there is a unique x in X".

▷ Proof

From Definition 1.7, $frag\colon V \hookrightarrow\!\!\!\!\rightarrow \mathcal{F}$ and $link\colon E \hookrightarrow\!\!\!\!\rightarrow \mathcal{L}$ are bijective.

◁

1.3 Bidirected graph (BG): oriented walk based

The *bidirected graph (bigraph)* was introduced by Edmonds and Johnson (1970). Myers (1995) is the first to suggest a bigraph structure (BG) to store the links exploiting the reverse symmetry of the fragment and the link sets $\mathcal{F}$ and $\mathcal{L}$. The key idea is to represent both a fragment and its reverse by only one vertex. Because the links are ordered pairs of oriented fragments, the edges are bidirected. Definition 1.10 provides a formal description, simpler than the one given in Gritsenko et al. (2012) and suggested by Myers (1995).

For the read assembly stage, BG is described in `Bambus` (Pop et al., 2004), `Minimus` (Sommer et al., 2007) and in `Edena` (Hernandez et al., 2008). Concerning the scaffolding stage, BG is found in `SOPRA` (Dayarian et al., 2010), `Bambus 2` (Koren et al., 2011), `MIP` (Salmela et al., 2011), `Opera` (Gao et al., 2011), `SSPACE` (Boetzer et al., 2011), `GRASS` (Gritsenko et al., 2012), `BOSS` (Luo et al., 2017), `LACHESIS` (Burton et al., 2013), `ScaffoldScaffolder` (Bodily et al., 2016) and `SLR` (Luo et al., 2019).

▶ **Definition 1.10: Oriented walk based bidirected multigraph (BG)**

Let $BG = (V, E, \Phi, attre)$ be a *bidirected multigraph (multibigraph)* such that:

— V is the set of vertices, where each $v \in V$ represents the two orientations for a fragment in $\mathcal{F}$;

— E is the multiset of edges, where each $e \in E$ represents one link and its reverse in $\mathcal{L}$;

— $\Phi\colon E \to \{\{u, v\} \mid u, v \in V, u \neq v\}$ is the incidence function that associates an edge with a pair of vertices;

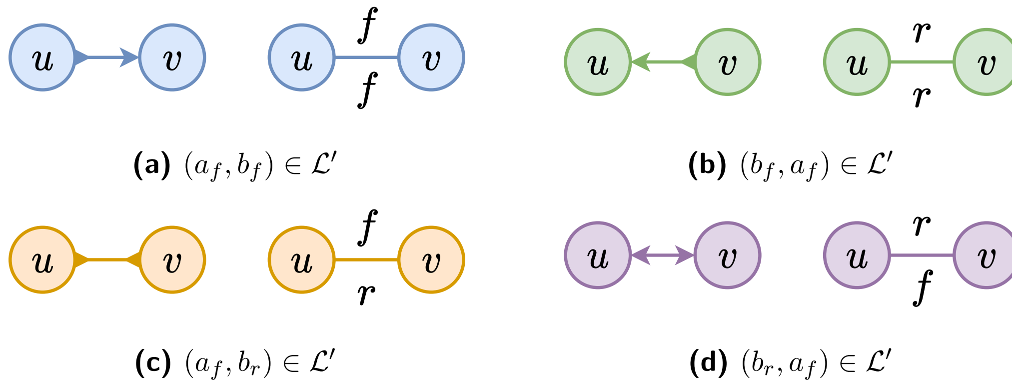
— $attre\colon E \to \{f, r\}^2$ gives the orientations of the vertices in the associated link, and in the lexicographic order of the unoriented fragment identifiers.

Denote by $unfrag\colon V \hookrightarrow\!\!\!\!\rightarrow \mathcal{F}_{un}$ the bijective function that associates each vertex with one and only one unoriented fragment, and by:

$$link\colon \{(u, v, e) \mid e \in E, \Phi(e) = \{u, v\}\} \hookrightarrow\!\!\!\!\rightarrow \mathcal{L}$$

the bijective function that associates each edge and its ordered vertices with one and only one link.

While visually each edge has two extremities in the multibigraph, Definition 1.10 describes the graph as an *undirected multigraph (multiungraph)*. Figure 18 provides the link representation in BG, where each subfigure gives the multibigraph and the multiungraph view.



(a) $(a_f, b_f) \in \mathcal{L}'$ **(b)** $(b_f, a_f) \in \mathcal{L}'$

(c) $(a_f, b_r) \in \mathcal{L}'$ **(d)** $(b_r, a_f) \in \mathcal{L}'$

Figure 18 – Link cases in BG.
Fragments a_f, b_f and their reverse a_r, b_r can link in four manners. Their associated vertices are u and v. For each subfigure, the first graph has bidirected edges and the second presents the same information but with undirected edges where the attributes are given by the function $attre$. Here we assume that $unfid(a) < unfid(b)$. In **(a)** and **(c)**, $attre$ gives the orientations in the order of the fragments in the described link. In **(b)** and **(d)**, $attre$ gives the orientations in the order of the fragment in the reverse of the described link ($\{r, r\}$ for (a_r, b_r) and $\{r, f\}$ for (a_r, b_f), respectively).

▶ **Property 1.4: Sizes of BG**

For $BG = (V, E, \Phi, attre)$, $|V| = \frac{1}{2}|\mathcal{F}|$ and $|E| = \frac{1}{2}|\mathcal{L}|$.

Less explicitly than in Definition 1.8, Definition 1.11 extends the reverse operations given in Definitions 1.4 and 1.6 for BG.

▶ Definition 1.11: The reverse operation in BG

Let $e \in E$ be an edge, $\Phi(e) = \{u, v\}$, $a = unfrag(u)$ and $b = unfrag(v)$. Assume that $unfid(a) < unfid(b)$. As function $link$ is bijective, it holds that:

— $link(v, u, e) = \overline{link(u, v, e)}$;

— $attre(e) = (for(a), for(b))$ and $attre(e)\begin{bmatrix} 0 & \bar{\cdot} \\ \bar{\cdot} & 0 \end{bmatrix} = \left(\overline{for(b)}, \overline{for(a)}\right)$.

Definition 1.12 specifies the conditions for a path in BG to be valid, illustrated in Figure 19.

▶ Definition 1.12: Path in BG

A path $p = (v_0, v_1, \ldots, v_{n-1}) \in V^n, n \in \mathbb{N}$ in $BG = (V, E, \Phi, attre)$ is valid if the following two properties hold:

Contiguity $\forall\, i \in [\![0, n-1[\![, \exists\, e_i, e_{i+1} \in E$ s.t.:

— $\Phi(e_i) = \{v_{i-1}, v_i\} \wedge \Phi(e_{i+1}) = \{v_i, v_{i+1}\}$

— $link(v_{i-1}, v_i, e_i)[1]$[a] $= link(v_i, v_{i+1}, e_{i+1})[0]$

Exclusive orientation $\forall\, v_i, v_j \in p, v_i = v_j \implies i = j$

[a]The notation $x[i]$ denotes the ith element in x.

Finally, Property 1.5 defines the reverse symmetry in BG, that inherits from Property 1.1.

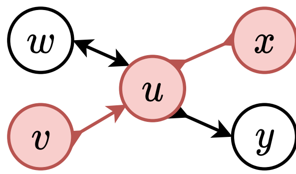
▶ Property 1.5: Reverse symmetry in BG

$\forall\, l \in \mathcal{L}, \exists!\, e \in E, \Phi(e) = \{u, v\} \mid link(u, v, e) = l \veebar link(v, u, e) = l.$[a]

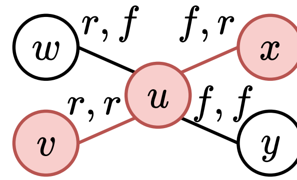
[a]$\veebar$ is the exclusive disjunction notation.

▷ Proof

From Definition 1.10, $link\colon \{(u, v, e) \mid e \in E, \Phi(e) = \{u, v\}\} \hookrightarrow\!\!\!\!\!\rightarrow \mathcal{L}$ *is bijective.*



(a) bidirected graph view



(b) undirected graph view

Figure 19 – A path in BG.
A valid path is coloured in red. It can be read in two possible ways: from vertex v or from vertex x. Let $p_v = vux$ be the one that begins from v and $p_x = xuv$ be the one that begins from x. Note that p_v is valid if and only if p_x is valid. In **(a)** edges are bidirected while in **(b)** they are undirected.

◁

1.4 Undirected graph (UG): tail-head fragments based

Huson et al. (2002) first describe the *undirect graph (ungraph)* structure (UG) for the fragment and their links, in the context of contig scaffolding. The idea is to keep the merge of the two strands as in BG, but with a simpler description of the orientations for the fragments. Each fragment is split by a tail and a head. Both the tail and the head are vertices, and they are connected by an edge. Passing through first the tail and then the head corresponds to choosing the fragment in its forward orientation, while passing through first the head and then the tail corresponds to choosing it in its reverse orientation. This new type of edges are called *fragment-edges*, at the opposite of *link-edges* that correspond to the links. A path in the graph must alternate between fragment-edges and link-edges.

The structure mainly appears in method for the scaffolding stage, as in `SCARPA` (Donmez and Brudno, 2013), `Scaftools` (Briot et al., 2014), `BESST` (Sahlin et al., 2014), `fragScaff` (Adey et al., 2014), in Chateau and Giroudeau (2015) and Weller et al. (2015), `ScaffMatch` (Mandric and Zelikovsky, 2015), `SALSA2` (Ghurye et al., 2019), `LRScaf` (Qin et al., 2019), in Davot et al. (2022) and Aganezov et al. (2022). Concerning the read assembly stage, UG can be found in Myers (2005) and Li (2016).

▶ Definition 1.13: Tail-head fragments based undirected multigraph (UG)

Let $UG = (V, E_{\mathcal{F}}, E_{\mathcal{L}}, \Phi_{\mathcal{L}})$ be an multiungraph such that:

— V is the set of vertices, where each $v \in V$ represents the tail or the head of a fragment;

— $E_{\mathcal{F}} = \{\{v_t, v_h\} \mid v_t, v_h \in V, v_t \neq v_h\}$ is the set of fragment-edges (v_t for the tail, v_h for the head), where each $e \in E_{\mathcal{F}}$ represents both the forward and the reverse orientation of one fragment;

— $E_{\mathcal{L}}$ is the multiset of link-edges, where each $e \in E_{\mathcal{L}}$ represents both one link and its reverse;

— $\Phi_{\mathcal{L}} \colon E_{\mathcal{L}} \to \{\{u, v\} \mid u, v \in V, u \neq v\}$ is the incidence function that associates a link-edge with two vertices that are not connected by a fragment-edge.

Denote by:

$$frag \colon \{(u, v) \mid \{u, v\} \in E_{\mathcal{F}}\} \hookrightarrow\!\!\!\!\rightarrow \mathcal{F}$$

the bijective function that associates each ordered fragment-edge with one and only one (oriented) fragment, and by:

$$link \colon \{(u, v, e) \mid e \in E_{\mathcal{L}}, \Phi_{\mathcal{L}}(e) = \{u, v\}\} \hookrightarrow\!\!\!\!\rightarrow \mathcal{L}$$

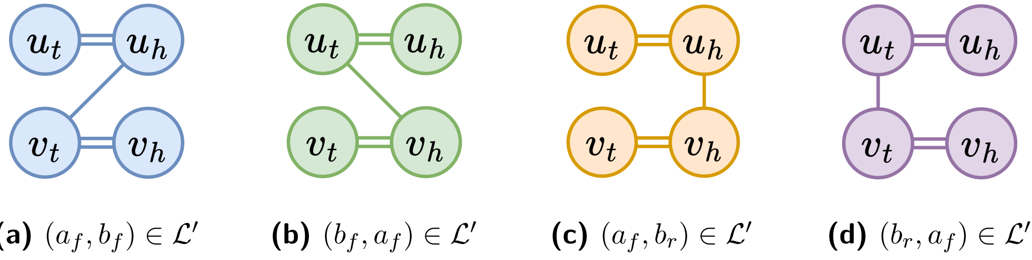
the bijective function that associates each edge and its ordered vertices with one and only one link.

To complete Definition 1.13, as in Chateau and Giroudeau (2015), we can define the set $E_{\mathcal{F}}$ of fragment-edges as a perfect matching of the vertices. Figure 20 illustrates each link case in UG.

▶ Property 1.6: Sizes of UG

For $UG = (V, E_{\mathcal{F}}, E_{\mathcal{L}}, \Phi_{\mathcal{L}})$, $|V| = |\mathcal{F}|$, $E_{\mathcal{F}} = \frac{1}{2}|\mathcal{F}|$ and $|E_{\mathcal{L}}| = \frac{1}{2}|\mathcal{L}|$.

Analogously to Definitions 1.8 and 1.11, Definition 1.14 formalises the reverse operation in UG.



(a) $(a_f, b_f) \in \mathcal{L}'$ **(b)** $(b_f, a_f) \in \mathcal{L}'$ **(c)** $(a_f, b_r) \in \mathcal{L}'$ **(d)** $(b_r, a_f) \in \mathcal{L}'$

Figure 20 – Link cases in UG.
u_t, u_h and v_t, v_h are the tail and the head of fragments a and b, respectively. The double edges correspond to fragment-edges while the plain edges correspond to link-edges. Each subfigure illustrates one link case in UG.

▶ Definition 1.14: The reverse operation in UG

— Let $\{u, v\} \in E_{\mathcal{F}}$. Hence $frag(v, u) = \overline{frag(u, v)}$.

— Let $e \in E_{\mathcal{L}}, \Phi_{\mathcal{L}}(e) = \{u, v\}$. Hence $link(v, u, e) = \overline{link(u, v, e)}$.

Definition 1.15 specifies the conditions for a path in UG to be valid, illustrated in Figure 21.

▶ Definition 1.15: Path in UG

A path $p = (v_0, v_1, \ldots, v_{2k-1}) \in V^{2k}, k \in \mathbb{N}$ in $UG = (V, E_{\mathcal{F}}, E_{\mathcal{L}}, \Phi_{\mathcal{L}})$ is valid if the following two properties hold:

Contiguity fragment-edges and link-edges alternate in the path:

— odd-numbered edges in the path are fragment-edges:

$$\forall\, i \in [\![0, k[\![, \{v_{2i}, v_{2i+1}\} \in E_{\mathcal{F}}$$

— even-numbered edges in the path are link-edges:

$$\forall\, i \in [\![1, k[\![, \exists\, e \in E_{\mathcal{L}} \mid \Phi_{\mathcal{L}} = \{v_{2i-1}, v_{2i}\}$$

Exclusive orientation $\forall\, v_i, v_j \in p, v_i = v_j \implies i = j$

Note that walking in UG is not similar that walking in a classic ungraph, because a valid *walk* must alternate between fragment-edges and link-edges. Figure 21 gives

an example of a valid path in UG.

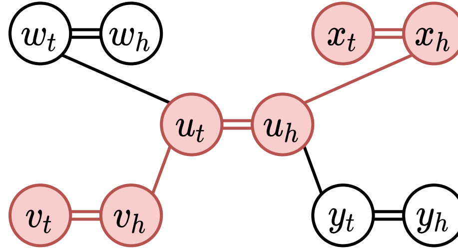


Figure 21 – A path in UG.
A valid path is coloured in red. As UG is undirected, the resulting path can be read in two possible ways: from vertex v_t or from vertex x_t. Let $p_v = v_t v_h u_t u_h x_h x_t$ the one that begins from v_t, and $p_x = x_t x_h u_h u_t v_h v_t$ the one that begins from x_t. Note that p_v is valid if and only if p_x is valid.

Finally, Property 1.7 defines the reverse symmetry in UG, that inherits from Property 1.1.

▶ **Property 1.7: Reverse symmetry in UG**

— $\forall a \in \mathcal{F}, \exists! \{u, v\} \in E_{\mathcal{F}}$ s.t.:

$$frag(u, v) = a \veebar frag(v, u) = a$$

— $\forall l \in \mathcal{L}, \exists! e \in E_{\mathcal{L}}, \Phi_{\mathcal{L}}(e) = \{u, v\}$ s.t.:

$$link(u, v, e) = l \veebar link(v, u, e) = l$$

▷ Proof

From Definition 1.13, $frag\colon \{(u, v) \mid \{u, v\} \in E_{\mathcal{F}}\} \hookrightarrow\!\!\!\!\rightarrow \mathcal{F}$ *and* $link\colon \{(u, v, e) \mid e \in E_{\mathcal{L}}, \Phi_{\mathcal{L}}(e) = \{u, v\}\} \hookrightarrow\!\!\!\!\rightarrow \mathcal{L}$ *are bijective.*

◁

Quick reference

For a quick overview of the different fragment graph structures, the readers can refer to Figure 22.

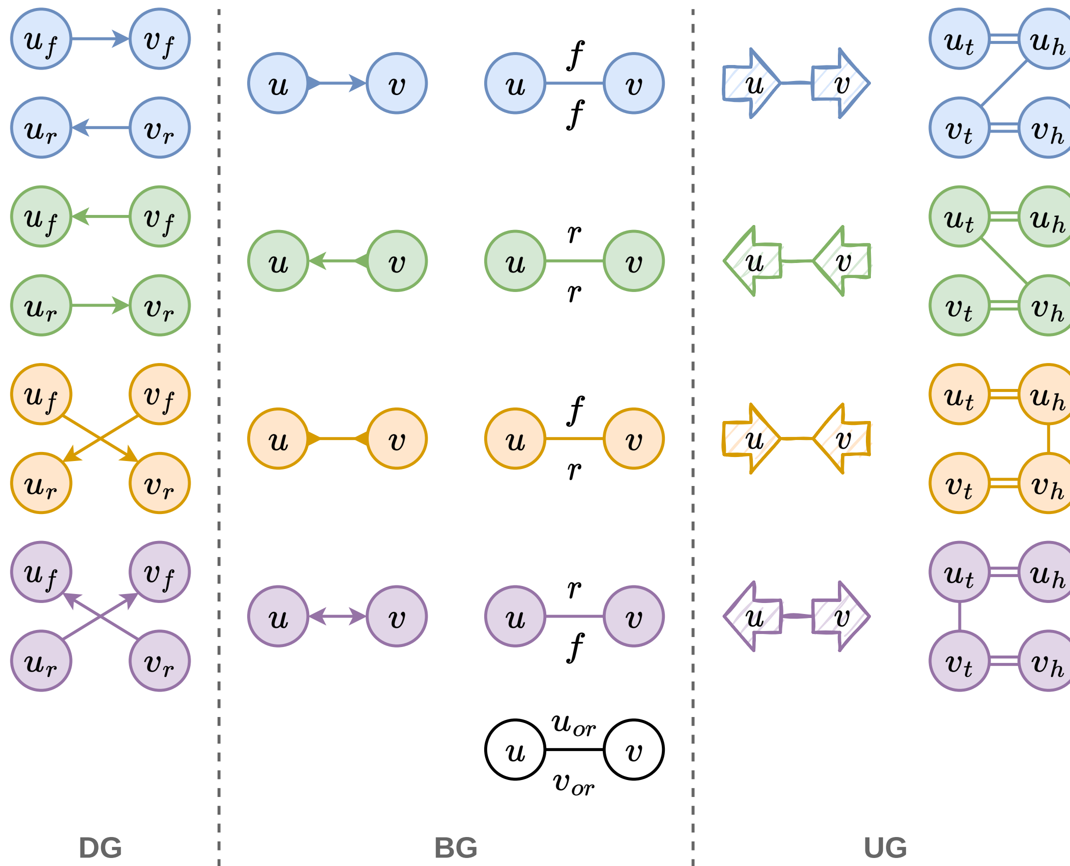


Figure 22 – Overview of all the fragment graph structures.
Let a and b two fragments. Each vertical dashed line separates two graph structures. Each colour is associated with an link case (and its reverse): $(a_f, b_f) \in \mathcal{L}'$ in blue, $(b_f, a_f) \in \mathcal{L}'$ in green, $(a_f, b_r) \in \mathcal{L}'$ in yellow and $(b_r, a_f) \in \mathcal{L}'$ in violet. **DG** Directed fragment graph. Each fragment is represented by two vertices for the two orientations. **BG** Bidirected fragment graph. Each fragment is represented by only one vertex. The first column gives the bidirected view while the second provides the undirected graph view with edge attributes (described at the bottom). **UG** Undirected fragment graph. Each fragment is represented by two vertices for its tail and its head, connected by a fragment-edge (double edges). Link-edges are simple edges.

2 Scaffolding the contigs

Recall from Chapter I, Section 5.4 that the first step of fragment assembly is to produce contigs from the reads. The second stage, the scaffolding, aims to orient and order the contigs. It results in a set of scaffolds, such that each of them is an

order of oriented contigs separated by gaps, with possible known nucleotide length.

We first categorise the type of scaffolding input data in Section 2.1 and their subsampling strategies in Section 2.2. In Sections 2.3 to 2.5 we describe the scaffolding formulations proposed in the literature and the approaches to solve them, exactly or heuristically.

As the scaffolding stage was originally designed to use the mate-pair and paired-end data (see Chapter I, Section 3 and Figure 9), the latter sections are especially focusing on methods that take them as input. Nevertheless, the way they address the scaffolding problem formulations enables to generalise the orienting and the ordering of the contigs, under the assumption that different sequencing data can provide the same link information, as discussed in the first section. Table 1 lists the scaffolding methods described in this section. The three graph representations (DG, BG, UG) can be compared in Figure 22.

2.1 Scaffolding input data

We propose to discriminate the scaffolding input data according to two properties. The first one is a link property: the data consist in a (multi)set of ordered pairs of oriented contigs (c.f. Section 1.1.2). The second one is a nucleotide distance property: in addition to their order, the oriented contigs are spaced appart by an estimated nucleotide distance.

As the contigs are the read assembly results, it is necessary to align the reads (or the sequencing data) against the contigs. In fact, especially with the DBG read assembly approach, the reads permitting to build a contig are unknown. Figure 23 illustrates the two properties with a sequencing data centric point of view, while Table 2 provides a property centric view.

Paired-end and Hi-C reads The reads come by pairs and are sequenced from the two different strands. The reads are mapped against the contigs. Consider r_i, r_j to be two paired-end reads such that r_i maps on contig c_i and r_j maps on contig c_j. Therefore, oriented contig c_i is before c_j. Two cases:

i. The two reads map two different contigs. The orientation of c_i is given by the one of r_i in the mapping, and the orientation of c_j is the reverse of this of r_j in the mapping. In the case of paired-end read and mate-pair reads (not for Hi-C), the distance between the oriented contig is determined according the mapping coordinates of the reads on the contigs and the distance between the paired reads.

ii. The two reads map the same contig. To be consistent, the reads must map in different orientations, otherwise it may indicate a misassembly of the two

Table 1 – Non-exhaustive categorised list of methods and approaches for the scaffolding stage.
Each row corresponds to a method dedicated to the scaffolding stage in the fragment assembly process. In the **Graph** column, DG, BG and UG respectively stand for directed, bidirected and undirected graphs.

Dedicated input data	Approach	Software	Graph
Paired-end/Mate-pair	Huson et al. (2002)	–	UG
	Pop et al. (2004)	Bambus	BG
	Dayarian et al. (2010)	SOPRA	BG
	Koren et al. (2011)	Bambus 2	BG
	Salmela et al. (2011)	MIP	BG
	Gao et al. (2011)	Opera	BG
	Boetzer et al. (2011)	SSPACE	BG**
	Roy et al. (2012)*	SLIQ	–
	Gritsenko et al. (2012)	GRASS	BG
	Donmez and Brudno (2013)	SCARPA	UG
	Sahlin et al. (2014)	BESST	UG
	Mandric and Zelikovsky (2015)	ScaffMatch	UG
	Luo et al. (2017)	BOSS	BG
	Andonov et al. (2019)	GAT	DG
Hi-C	Burton et al. (2013)	LACHESIS	BG
	Ghurye et al. (2019)	SALSA2	UG
Long reads	Qin et al. (2019)	LRScaf	UG
	Luo et al. (2019)	SLR	BG
Linked reads	Adey et al. (2014)	fragScaff	UG
	Coombe et al. (2018)*	ARKS	–
Generic link	Briot et al. (2014)	Scaftools	UG
	Chateau and Giroudeau (2015)	–	UG
	Weller et al. (2015)	–	UG
	Davot et al. (2022)	–	UG

*SLIQ and ARKS are pre-scaffolder methods.**Suggested in supplementary method.

strands. For paired-end data, if the distance between the reads and the absolute difference of the two mapping coordinates do not coincide, then it also suggest a misassembly (insertion or deletion).

Long reads and reference The order and the orientations of the contigs is determined according their mapping against a longer sequence (a long read or a reference).

Optical map The genome is cut multiple times by a specific restriction site. Each cut produces a map that consists in a list of distances between the same nucleotide site. The same process is artificially produced for the contigs. Then, the merged maps of a contig is aligned with the merged maps given by the optical map. This produce an order between the oriented contig.

Linked reads Each pair of reads is tagged with a barcode. Two pairs with the same barcode means they come from the same fragment. It enables to cluster the contigs, without orienting or ordering them. One way to find an order between two oriented contigs is to split each contig in two, and check which half parts of the contigs are covered by the reads with the same barcode.

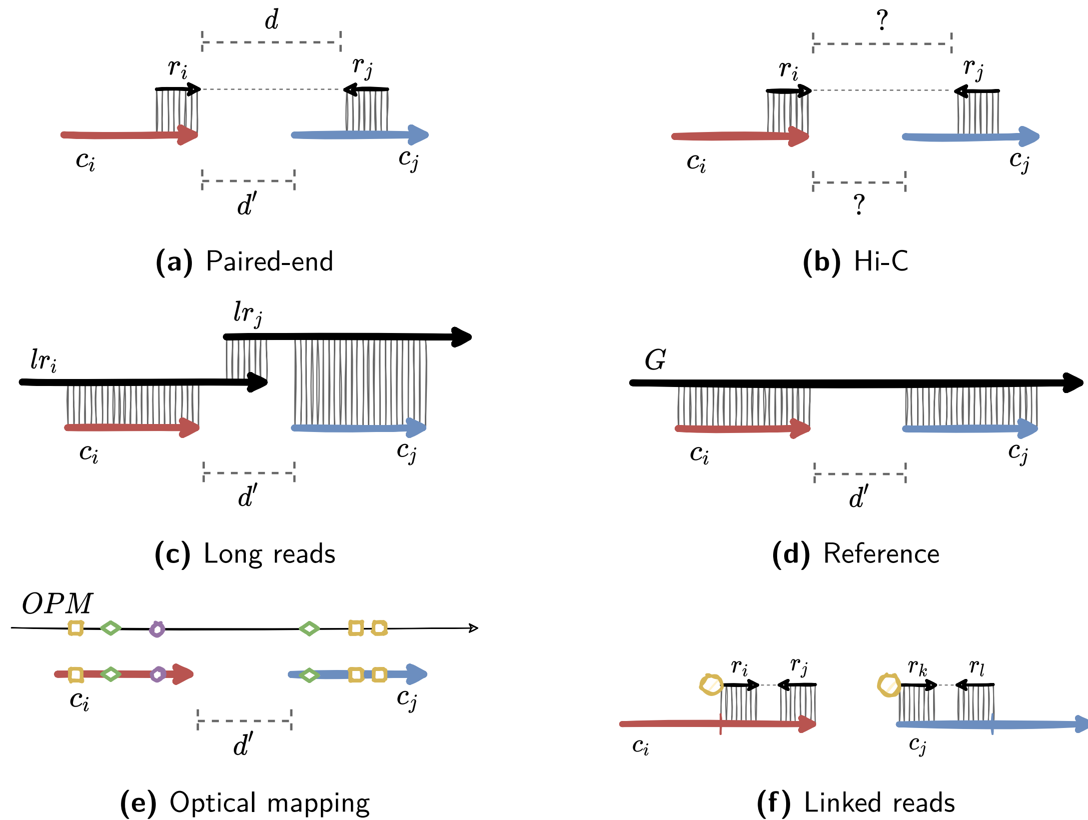
Table 2 – Scaffolding input data properties.
Each sequencing data is provided with at most the two properties, link and nucleotide distance.

Sequencing data	Link	Distance
Paired-end*	✓	✓
Mate pair*	✓	✓
Hi-C	✓	
Long reads	✓	✓
Reference	✓	✓
Optical mapping	✓	✓
Linked reads**	✓	

*Paired-end and mate pair data provide the same type of data except that the distances are larger for the second one but less confident. **The barcoding feature of linked read itself does not provide distance information. However, each paired-end read is barcoded.

2.2 Subsampling the input data

Several difficulties arise for the scaffolding problem. First, the number of links between the contigs is the main factor for the complexity of orienting and ordering them. Merging several links into one drops the computational time and memory use. Moreover, the presence of sequencing error suggests filtering the result of the alignments, or the sequencing data themselves. Even after a filtering step, artefact alignments may occur because of repeats in the genome, and consequently may produce links resulting in misassemblies if they are chosen.



(a) Paired-end **(b)** Hi-C

(c) Long reads **(d)** Reference

(e) Optical mapping **(f)** Linked reads

Figure 23 – Scaffolding input data.
For each subfigure, c_i and c_j are two contigs and the vertical grey lines represent nucleotide alignment. **(a)** The two paired-reads r_i and r_j map on two different contigs. It implies that c_i forward is followed by c_j forward, separated by a nucleotide distance d'. **(b)** This case is the same as (a) except that for Hi-C data, the distance d is unknown, implying an unknown distance d'. **(c)** c_i and c_j map on two overlapping long reads lr_i and lr_j. It implies that c_i forward is followed by c_j forward, separated by an approximated nucleotide distance d'. **(d)** G is a reference genome (e.g. of a species near to the organism of interest). Mapping c_i and c_j on G provides an orientation of the contigs and an order with a distance d'. **(e)** OPM is an optical map with three markers (yellow square, green diamond and purple circle). Finding the same markers orders in c_i and c_j enables to estimate their orientation and their order with a distance d'. **(f)** r_i, r_j, and r_k, r_l are two paired-end reads with the same barcode (yellow circle), so they are sequenced from the same fragment. Because r_i, r_j map on the head of c_i, and r_k, r_l map on the tail of c_j, it is possible to orient and order c_i and c_j.

2.2.1 Bundling the links

When several alignments produce the same ordered pair of oriented contigs, they can be bundled. Thus, only one link represents them. This is the strategy that Huson et al. (2002), Dayarian et al. (2010), Koren et al. (2011), Gao et al. (2011), Gritsenko et al. (2012), Salmela et al. (2011), Donmez and Brudno (2013), Briot et al. (2014), Mandric and Zelikovsky (2015) and Luo et al. (2017) adopt with paired-end data, Burton et al. (2013) and Ghurye et al. (2019) for Hi-C data, and Adey et al. (2014) and Coombe et al. (2018) for linked reads. The link representing the bundle is weighted proportionally to the number of links supporting the bundle. In the case of distances, the mean and the standard deviation is computed.

2.2.2 Removing paired-end reads

Detecting artefact data Low mapping score are ignored, as Luo et al. (2017) does. If only one read of the pair is aligned, the whole pair is ignored (Burton et al., 2013; Coombe et al., 2018). Mandric and Zelikovsky (2015) and Luo et al. (2017) remove paired-end alignments from the bundle if the distance they provide between the contigs is over a confident interval. Luo et al. (2017) compute the distance between the reads if the two contigs are concatenated, and remove the alignment of the pair if the distance is too long. Finally, because of the nature of Hi-C paired-end reads, Ghurye et al. (2019) trim the reads that are aligned after a ligation.

Detecting repeats When one of the two paired reads maps on distinct contigs, Mandric and Zelikovsky (2015), Burton et al. (2013) and Ghurye et al. (2019) do not consider the pair. Luo et al. (2017) ignore the paired-end reads when they map to a highly covered contig.

2.2.3 Removing contigs

Detecting artefact data Dayarian et al. (2010) remove a contig on which the two reads of a pair are mapped on it with the same orientations. They also compute the standard deviation of all the distances in all the contigs and remove those for which the reads' distance is out of the deviation. For Hi-C data, Burton et al. (2013) remove a contig if it does not contain enough restriction sites (only for their clustering step). In the long read scaffolding method of Luo et al. (2019), short contigs are ignored. Finally, Qin et al. (2019) only consider contigs that are contained in long reads.

Detecting repeats A contig is dismissed if its alignment coverage is over a threshold of beyond a distribution's confidence interval (Gao et al., 2011; Sahlin et al., 2014;

Mandric and Zelikovsky, 2015; Qin et al., 2019). For Hi-C data, Burton et al. (2013) identify a contig as a repeat if the number of restriction sites is high. Luo et al. (2019) remove a contig if it maps in the middle of different long reads, and if its neighbour contigs differ. Koren et al. (2011) computes all the shortest path between two contigs: a contig appearing in too many paths is dismissed.

2.2.4 Removing links

Detecting artefact data A bundle with a low confidence is not considered. The lower threshold can be a fixed value, or based on the distribution of the confidence over all the bundles (Gao et al., 2011; Gritsenko et al., 2012; Donmez and Brudno, 2013; Luo et al., 2017; Qin et al., 2019). Note that in Gao et al. (2011) this threshold is the result of a simulation. If the distribution of the distances on a link significantly differs from the distribution over all the bundles, then the link is dismissed (Sahlin et al., 2014; Qin et al., 2019) In Ghurye et al. (2019), the confidence of neighbours links is compared, and only the "best buddy weighted" link is kept (a $O(|\mathcal{L}|)$ procedure). In addition, as they take in input an assembly graph, the authors remove Hi-C links if there is no link in the assembly graph that match the contig orientations. Qin et al. (2019) dismiss a contig alignment if the long read is contained in the contig.

Reducing redundant links As proposed for read assembly OLC approach, Huson et al. (2002), Koren et al. (2011) and Qin et al. (2019) compute a transitive reduction of the links based on their distance distribution.

Aggressive removing Roy et al. (2012) predict the orientations based on the links for which the two contigs have the same orientation. A common aggressive link removing is to consider only one link between two contigs of the four cases. Dayarian et al. (2010), Luo et al. (2017) and Qin et al. (2019) keep the maximum-weighted link, in addition, Koren et al. (2011) remove the smallest number of unconfident alignments in the bundle according the distance distribution. Coombe et al. (2018) keep only one link if it passes a binomial test of being the real link among the others. Ghurye et al. (2019) compute all the shortest path between two contigs and keep only a link if its ratio of presence in the paths is significantly greater than the presence of the other links involving the two same contigs (a $O(|\mathcal{L}|(|\mathcal{C}| + |\mathcal{L}|))$ procedure). Finally, if two contigs are aligned against the same end coordinate of a long read, the best mapping is kept in Luo et al. (2019).

2.2.5 Partitioning the instances

Biconnected components can be solved independently For BG and UG fragment graphs (Sections 1.3 and 1.4), finding articulation edges and removing them enables to partition the graph into biconnected components. Dayarian et al. (2010) show that each component can be solved independently without a loss of optimality. Salmela et al. (2011) and Donmez and Brudno (2013) also apply this divide and conquer strategy. Finding all the articulation points can be done in $O(|\mathcal{C}|^2)$

Clustering according to the number of chromosomes In the Burton et al. (2013) Hi-C scaffolding method, the number of chromosomes N is an input. Based on the links' weight, the contig set is partitioned in N parts with a hierarchical agglomerative clustering approach. Note that the clustering does not take into consideration the relative orientations of the contigs.

2.3 Orienting the contigs

Here we present the methods that focus on the contig orientation problem independently of their ordering. Kececioglu and Myers (1995) have already defined it in the context of the read assembly. It consists in removing links or contigs in order to find only two orientation assignments, where one is the reverse of the other. A link set over a contig set is defined as an orientation-valid layout if it respects the latter definition.

▶ Definition 2.1: Orientation-valid layout

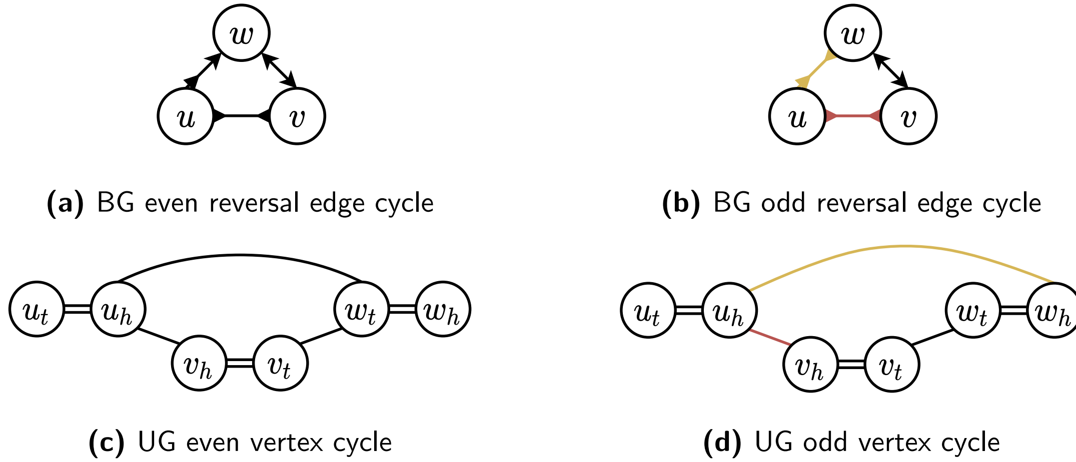
Denote by $\mathcal{C}^{\subset} = \{c_0, \ldots, c_{n-1}\}$ a subset of the contig set $\mathcal{C}$ and by $\mathcal{L}^{\subset}$ a subset of the link set $\mathcal{L}$. Without considering an ordering of the contigs in $\mathcal{C}^{\subset}$, $\mathcal{L}^{\subset}$ is an orientation-valid layout if and only if there are exactly two orientation assignments $o = (o_0, \ldots, o_{n-1}) \in \{f, r\}^{|\mathcal{C}^{\subset}|}$ and $\overline{o} = \left(\overline{o_0}, \ldots, \overline{o_{n-1}}\right)$ and there exists a balanced partition of $\mathcal{L}^{\subset} = \mathcal{L}^{\subset}_f \sqcup \mathcal{L}^{\subset}_r$ such that:

— $\left|\mathcal{L}^{\subset}_f\right| = |\mathcal{L}^{\subset}_r|$ and $\forall\, l \in \mathcal{L}^{\subset}_f, \overline{l} \in \mathcal{L}^{\subset}_r$;

— o and $\overline{o}$ are respectively consistent in $\mathcal{L}^{\subset}_f$ and $\mathcal{L}^{\subset}_r$, i.e. $\forall\, l \in \mathcal{L}^{\subset}_f$ where $linkp(l) = (a, b)$, and so $\overline{l} \in \mathcal{L}^{\subset}_r$ where $linkp(\overline{l}) = \left(\overline{b}, \overline{a}\right)$, $\exists!\; i, j \in [\![0, n[\![, i \neq j$ such that:

- $a = (c_i, o_i) \iff \overline{a} = (c_i, \overline{o_i})$;
- $b = (c_j, o_j) \iff \overline{b} = (c_j, \overline{o_j})$.

Figure 24 illustrates Definition 2.1 for the bigraph (BG) and for the ungraph (UG) as defined in Sections 1.3 and 1.4, and illustrated in Figure 22.



(a) BG even reversal edge cycle

(b) BG odd reversal edge cycle

(c) UG even vertex cycle

(d) UG odd vertex cycle

Figure 24 – Bidirected and undirected graph cycles.
(a) and **(c)**: there are exactly two vertex orientation assignments ($o = \{u_f, v_r, w_f\}$, and the reverse orientations $\overline{o}$) such all the valid paths assign vertex orientations respecting o, otherwise $\overline{o}$. **(b)** and **(d)**: there are at least two valid paths that imply two different orientation assignments with a non-null intersection, e.g. $o = \{u_f, v_r, w_f\}$ and $o' = \{v_r, w_f, u_r\}$. It is sufficient to remove one of the red or the yellow link to get a valid-orientation layout.

2.3.1 Maximising the sum of used link weights

As originally proposed by Kececioglu and Myers (1995), the orientation problem consists in removing links in order to get an orientation-valid layout that maximising the sum of the remaining links' weight. Here, no one of the contigs is removed, i.e. $\mathcal{C}' = \mathcal{C}$. Equivalently, it consists in minimising the sum of removed links' weight. Kececioglu (1991) reduced this formulation to the maximum-weight cut problem whose decision version is $\mathcal{NP}$-complete. Pop et al. (2004) consider the weight proportional to the number of supporting paired-end reads in the bundle, or inversely proportional to the nucleotide distance of the bundle. Dayarian et al. (2010) aim to minimise a cost function (simply, the sign of the weight is inverse comparing to the other methods). In `Bambus 2` (Koren et al., 2011) (as for its predecessor `Bambus`), the problem consists in removing the minimum number of reversal links, i.e. links where the two contigs in the pair are not identically oriented.

Luo et al. (2017) solve the problem iteratively by increasing cleaned subgraph. The first subgraph contains the heaviest links, and after resolving it for the

orientation, lighter links are added etc. Luo et al. (2019) maximise the sum of remaining links' weight, where the weights correspond to the number of aligned long reads.

2.3.2 Remove the minimum number of contigs and links to avoid odd reversal cycles

Donmez and Brudno (2013) remark that the contig can be misassembled and conclude that the orientation problem should consider removing the contig and the links. Consequently, they associate the orientation problem with finding a minimum odd cycle traversal in a transformed UG. Indeed, to generalise with the removing of contigs, they transform each link-edge (u, v) into a path $uu'v'v$ where u' and v' are two new artificial vertices. Although finding a minimum odd cycle traversal is a $\mathcal{NP}$-hard problem, Lokshtanov et al. (2009) propose a fixed-parameter tractable algorithm in $O(3^k k|E||V|)$, where k is the number of vertices to suppress, and V and E are respectively the vertex and the edge sets.

2.3.3 Maximum log-likelihood function

At the opposite of the previous methods, Burton et al. (2013) first order the contig. Then, they orient the contig according to a maximum log-likelihood function, where each edge weight equals to the inverse of the number of Hi-C links normalised by the number of fragment sites per contig.

2.4 Ordering the oriented contigs

Once an orientation assignment is determined for the contigs, the next step is ordering them and positioning them when nucleotide distances are provided.

2.4.1 Maximising the sum of used distances bunches' weights

Pop et al. (2004) reduce the ordering problem to the optimal linear arrangement problem. It consists in assigning a coordinate on an axe to each contig, a $\mathcal{NP}$-hard problem. As for the orientation problem, they consider to be the best weights either the number of supporting links in a bundle, or the smallest distances. However, they do not position the contigs by satisfying the distances, as done in Luo et al. (2019) for long read based scaffolding.

Donmez and Brudno (2013) refer to the feedback arc set problem ($\mathcal{NP}$-hard, Karp 1972) and define the contig ordering problem as finding a minimal set of

edges whose removal leads to a directed acyclic graph. Koren et al. (2011) position the contigs in confidence intervals of three times the standard deviation.

2.4.2 Minimum spanning tree

For Hi-C data, Burton et al. (2013) first order the contig. Recall that each link weight equals to the inverse of the number of Hi-C links normalised by the number of fragment sites per contig. They define the ordering problem as a sequence of process: (i) finding a minimum weight spanning tree; (ii) extracting the longest path (a trunk) to get a first order; (iii) ordering the branching contigs according to the trunk, from the longest to the smallest, with maximising the number of links in the bundle.

2.4.3 Contig positioning only

Donmez and Brudno (2013) define the positioning problem as a sub-step of the scaffolding problem following the ordering of the contigs. According to the distance distribution in each bundle, they aim to give a nucleotide position for each contig, such that it minimises the distance approximation in each link.

Gao et al. (2011) maximise a quadratic likelihood function on the distances to estimate the gaps in each scaffold. A unique solution can be found thanks to the Goldfarb-Idnani active-set dual method in polynomial time (Goldfarb and Idnani, 1983).

2.5 Orienting and ordering the contigs simultaneously

Already for the read assembly stage, Kececioglu (1991) has suggested combining both the orienting and the ordering/positioning problems in one step. The purpose is to seek a global optimisation approach. Here we enumerate the scaffolding methods that follow this approach.

2.5.1 Maximising the sum of distance bundles' weights to order linearly the contigs

Huson et al. (2002) aim to maximise the sum of weights of happy mate-edges in UG. A mate-edge (a bundle of paired-end links) is happy if the orientations of its contigs respect it, and if the interval of the contigs' coordinates is in a confidence interval of three times the standard deviation from the distances median provided by the edges. They prove the corresponding decision problem to be $\mathcal{NP}$-complete by reducing the bandwidth problem to it.

Gao et al. (2011) focus on a bounded version of the graph bandwidth problem, for which the complexity is in $O(|\mathcal{C}|^w |\mathcal{L}|^{p+1})$ where w is the fixed width, and there are at most p discordant edges in a scaffold.

In their Mixed-Integer Linear Programming (MILP) approach, Salmela et al. (2011) model if a link is kept by real variables between 0 and 1 instead of binaries. A value that approaches 1 indicates that two contigs' positions are close to the distance median provided by the link, while a value that approaches 0 indicates that the contigs' positions are far from the median. In the objective function, the links' weights are multiplied by these reals.

Gritsenko et al. (2012) combine three goals in a single objective function: minimising the cost of orientation, minimising the orientation and the distance cost, and minimising the orientation and the order cost.

2.5.2 The heaviest matching, paths and cycles

Briot et al. (2014) maximise the weight of the remaining links such that at most one link connects each contig's end in UG. This approach can result in several paths. Building on this, Weller et al. (2015) are looking for at most σ_p paths and σ_c cycles (where σ_p, σ_c are parameters) that cover all the contig-edges in UG. They follow the scaffolding definition given in Chateau and Giroudeau (2014), where the authors study the complexity of the problem on several classes of graphs and according to the number of contigs (n). In their paper, they are looking for exactly σ_p paths and σ_c cycles. Except in the case where $n = \sigma_p + 2\sigma_c$, for which the problem is in $\mathcal{P}$, the scaffolding problem (decision version) remains $\mathcal{NP}$-complete.

In the global Integer Linear Programming (ILP) formulation of Andonov et al. (2019), the objective function aims to find the longest nucleotide sequence that maximises the number of satisfied distances. It consists in searching for an elementary path in DG. Combining these two objectives in one linear function echoes the SCS criticisms formulated in the case of read assembly (see Chapter I, Section 5.1). Note that the paired-end links are not especially bundled, and each of them are two links in DG. At the opposite of the other approaches, here the contigs are unitigs associated with a multiplicity, which is an upper bound of its use. The model is enough general to integrate other weight definitions (both on the overlaps and on the unitigs).

2.5.3 The maximum (weighted) matching and spanning tree

Mandric and Zelikovsky (2015) reduce the scaffolding problem to the Maximum-Weight Acyclic 2-Matching (MWA2M) problem on the link-edges in UG. The latter one's decision version is $\mathcal{NP}$-complete.

In their Hi-C scaffolding method, Ghurye et al. (2019) iteratively find the maximum-weighted matching of the link-edges. Then, they remove the lightest links to eliminate the cycle in the matching to order and orient the covered contigs. Finally, the covered vertices are removed and they re-iterate on the resulting subgraph. The whole procedure is in $O(|\mathcal{L}||\mathcal{C}|^3)$.

Concerning the scaffolding with linked read, Adey et al. (2014) adopt a similar strategy as in Burton et al. (2013) for Hi-C data by finding a maximum-weighted minimum spanning tree, extract the longest path (trunk), and add the branching contigs to the linear ordering.

Finally, for long reads, Qin et al. (2019) only extract valid simple paths in UG and return them as scaffolds.

2.6 Solving approaches

2.6.1 Greedy approaches

As in the majority of the cases the scaffolding problem (orienting and ordering separately or in the same time) is $\mathcal{NP}$-hard, in practice some authors develop greedy algorithm to solve large instances.

Mate-pair in decreasing order of weight The main greedy strategy consists in prioritising the heaviest links. Huson et al. (2002) are the first to present the greedy path merging algorithm. Koren et al. (2011) adopt this strategy for the orienting and the ordering of the contigs separately. They use the same algorithm as in Kececioglu and Myers (1995): a $O(|\mathcal{C}| + |\mathcal{L}|)$ complexity algorithm on BG. Mandric and Zelikovsky (2015) propose a greedy heuristic for the MWA2M problem that chooses the heaviest feasible link-edge in UG. A link-edge is said feasible if it does not make a vertex degree higher than two and does not form cycles with the previously chosen edges. Their algorithm is in $O(|\mathcal{C}|log(|\mathcal{C}|))$ with a max heap implementation. In the continuity of Chateau and Giroudeau (2015), Davot et al. (2022) propose a $O(|\mathcal{C}||\mathcal{L}|\sigma_c^2)$ 3-approximation algorithm for connected cluster graphs (UG), where σ_c is the upper bound of cycles.

Seed-and-extend As for the read assembly stage, Seed-and-Extend (S&E) algorithms have been developed for the scaffolding stage. They ensure to form long scaffolds through the extension of the current paths. Boetzer et al. (2011) start with the largest contig and extend it with links supporting at least a given threshold. In the case of alternative links, they try to find an ordering of all the alternative adjacent contigs with the distance attributes. If it fails, they compute a ratio of the weight of the alternatives, and choose the best link if its ratio is above a fixed

threshold. As in Pop et al. (2004), Luo et al. (2017) first order the longest contigs and connect them with a linear BFS procedure.

Simulated annealing Dayarian et al. (2010) use a simulated annealing which is a Monte Carlo approach. By decreasing a parameter (the temperature) in the sample's weight, if the energy of the system reaches a value close to the minimum, most of the constraints are solved. This approach is applied for both the orientation and for the ordering, independently.

Spanning tree As described in the previous sections, Roy et al. (2012) implement a spanning tree algorithm for the orienting of the contigs while Adey et al. (2014) implement it for both the orienting and the ordering of the contigs.

Majority voting on predictions For the positioning of the contigs, Roy et al. (2012) employ a majority voting on the ordering predictions based on inequalities.

Randomised Greedy Algorithms for the maximum matching problem Finally, Ghurye et al. (2019) benefiting from the algorithm proposed in Poloczek and Szegedy (2012) running in $O(|\mathcal{L}|)$ with UG. The total complexity of their approach raises to $O(|\mathcal{C}||\mathcal{L}|)$.

2.6.2 Fix parameters dynamic programming

Gao et al. (2011) propose a $O(|\mathcal{C}|^w|\mathcal{L}|^{p+1})$ procedure, where w is the width and p is the upper bound of discordant edges in BG. They increase the parameters until a solution is found. Weller et al. (2015) give an exact algorithm in $O(|\mathcal{C}|\sigma_p^2)$ for trees (UG graph), where σ_p is the maximum number of paths to cover with the matching. By extension, they describe an exact tree decomposition algorithm in $O(tw^{tw}\sigma_p\sigma_c|\mathcal{C}|)$, where tw is the tree width for the decomposition, and σ_c is equivalent to σ_p for cycles.

2.6.3 Mathematical programming

Mathematical programming enables to model and to solve complex optimisation problems by using dedicated solvers.

Mixed Integer Linear Programming (MILP) Salmela et al. (2011) use a MILP approach on BG, where the link-choice variables are not binaries but are reals between 0 and 1. It enables to model the uncertainty of the link during the positioning of the contigs. Recall that they use MILP on each bi-connected

component found by removing articulation vertices. Briot et al. (2014) however model the link choice variables as binaries and connect a vertex with at most one link in UG. Luo et al. (2017) maximise the sum of weights of used links for orienting the contigs on iteratively increasing subgraphs. Andonov et al. (2019) model in one objective function the longest path and the maximum number of satisfied constraints on distances. Luo et al. (2019) apply the same strategy as in Luo et al. (2017) on both the orienting and the ordering problems.

Mixed Integer Quadratic Programming (MIQP) Gritsenko et al. (2012) are the only ones that use Mixed-Integer Quadratic Programming (MIQP) by merging orienting and ordering the contigs in one objective function.

Linear Programming (LP) Donmez and Brudno (2013) use linear programming to position the oriented and ordered contigs to minimise the positioning approximation according to the distance distributions on the links.

3 Chloroplast genome assembly

As the chloroplast is one of several living organelles in the same plant cell, the DNA sequencing process produces reads from chloroplasts, other organelles and nuclear DNA. Consequently, methods have been developed to produce a separate assembly of the chloroplasts. They mainly consist in an assembly pipeline and adopt various strategies in the filtering of the data during the whole assembly process.

In the following, all the described methods, listed in Table 3, are based on short paired-end read data. Recently, Freudenthal et al. (2020) have compared some of them on several criteria, from user-friendly aspects to assembly qualities. Here, we decompose the dedicated chloroplast genome methods that solve the whole or a part of the fragment assembly process.

Three major issues arise from the literature: (i) How to extract the chloroplast data from the mixture of genomic material in the sequencing output? (ii) What is the most suitable assembly formulation for the chloroplast genomes? (iii) How to deal with the structural haplotypes in the assembly results? Section 3.1 covers point (i), Sections 3.2 and 3.3 cover point (ii) and Section 3.4 covers point (iii).

3.1 Chloroplast sequence extraction

Although there are routine data filters, as in Bakker et al. (2016) and McKain and afinit (2017), here we discuss specific approaches to identifying chloroplast data (original and transformed) and extracting them.

Table 3 – Chloroplast genome assembly approaches.
Approach: the main contribution given in the paper. By Seed-and-Extend we mean a read assembly method. **Filter:** an arbitrary score (from 0, no filter, to 3 bullets) equivalent to the number of filter steps and the variability of their strategies. **Structural haplotypes:** if the method is chloroplast structural haplotypes aware, i.e. if it addresses the issue of several chloroplast forms in the sequencing data.

Paper and software	Approach	Filter	Structural haplotypes
Coissac et al. (2016)* `ORG.Asm`	Seed-and-Extend	?	?
Bakker et al. (2016) `IOGA`	Pipeline	••	
Dierckxsens et al. (2017) `NOVOPlasty`	Seed-and-Extend	•	
McKain and afinit (2017) `Fast-Plast`	Pipeline	••	
Ankenbrand et al. (2018) `chloroExtractor`	Pipeline	••	
Sancho et al. (2018) `Chloroplast assembly protocol`	Pipeline	•	
Andonov et al. (2019) `GAT`	Scaffolding		✓
Jin et al. (2020) `GetOrganelle`	Pipeline + Scaffolding	•••	✓

*Cannot find a description of the method, software manual not found.

3.1.1 Filtering the reads

De novo filtering Before any assembly of the raw data, Ankenbrand et al. (2018) assume that the chloroplast (and mitochondria) reads outnumber the nuclear reads. Therefore, they keep reads for which the k-mer distribution is higher comparing to the other reads (thanks to `Jellyfish`, Marçais and Kingsford 2011).

Genome reference-based filtering McKain and afinit (2017) and Sancho et al. (2018) align the reads against one reference chloroplast genome, Ankenbrand et al. (2018) against several closely related species, Bakker et al. (2016) against several but not

necessarily closely related. Jin et al. (2020) in addition align the reads against organelle subsequences. In the latter, they denote the reads they extract as seed reads. Note that in Sancho et al. (2018) the use of a reference genome is optional.

Baiting new reads from assembly Some methods continue to filter the data after assembling a subset of the reads into contigs. Bakker et al. (2016) iteratively align the previously unaligned reads to a running read assembly. Then they compute a new assembly on the larger read subset and repeat the process until no new read is extracted. Jin et al. (2020) extend the seed contigs (from the assembly of the seed reads) with the remaining reads.

3.1.2 Filtering the contigs

Similarly to the reads, the contig filters follow two approaches.

***De novo* filtering** McKain and afinit (2017) and Jin et al. (2020) keep the contigs with a high read mapping coverage. Especially, the latter authors partition the contig set in parts representing each organelle type according to a coverage distribution described by a Gaussian mixture.

Label database filtering In addition, Jin et al. (2020) align each contig to a label database, where each sequence (e.g. of a gene, a protein or a conserved region) is associated with a specific organelle (e.g. chloroplasts). They only keep the contigs that match the label database.

3.2 Chloroplast reads assembly

The assembly of the read is the first stage of the fragment assembly process. However, as we have seen in the previous subsection, assembling the reads often enables to bait more reads and thus raise the chances to have a complete chloroplast assembly.

3.2.1 De Bruijn graph approach

Recall that DBG is a dbgraph read assembly approach.

Baiting assemblies As mentioned before, Bakker et al. (2016) iteratively produce several read assemblies with several sizes of k-mer (recall Chapter I, Section 5.3) thanks to `SOAPdenovo2` (Luo et al., 2012). At each iteration they keep the assemblies

where the N50 criterion is high, and bait new reads on selected assemblies (i.e. contigs).

Assembling the reads Sancho et al. (2018) use `Velvet` (Zerbino and Birney, 2008), whereas Bakker et al. (2016), Ankenbrand et al. (2018) and Jin et al. (2020) prefer `SPAdes` that internally tests different size of k-mer (Bankevich et al., 2012). Unlike the latter that use paired-end for the read assembly, McKain and afinit (2017) use `SPAdes` without the paired-end option. Indeed, paired-end data are further exploited during the scaffolding stage.

3.2.2 Seed-and-extend

The S&E approach is a local assembling approach adapted for "simple" genomes, particularly with few repeats. Coissac et al. (2016) implement a S&E whose description could not be found at the time of writing this thesis. The S&E strategy in Dierckxsens et al. (2017) consists in: (i) extending the seed until a circularity is found; (ii) verifying the paired-end distance constraint at each extension; (iii) identifying repeats when extensions are not overlapping. In the case it cannot resolve the repeats based on more stringent extension parameters, the extension is cut, and it begins a new seed.

3.3 Chloroplast scaffolding

The scaffolding stage follows the assembly of the reads. Few chloroplast assembly tools tackle this stage.

Paired-end scaffolding McKain and afinit (2017) and Sancho et al. (2018) use the paired-end based scaffolder `SSPACE` (Boetzer et al., 2011). Finally, the latter fill the gap in the scaffolds with `GapFiller` (Boetzer and Pirovano, 2012). Andonov et al. (2019) test their scaffolder on chloroplast data. As described in Section 2.5.1, they aim to find the longest DNA sequence by finding a unitig path that maximises the number of satisfied distance constraints.

Estimating contig multiplicities to obtain a circular pattern Jin et al. (2020) aim to determine the multiplicity of each contig such that Eulerian circuits are found in the resulting graph. They use a MIQP aiming to minimise the squared difference between the observed and the variable multiplicities. Each equation expresses the multiplicity of a contig's end with those of its adjacent contigs' ends. They compute the observed multiplicities according to the contigs' coverages.

3.4 Chloroplast assembly validation

Validation of assemblies can be carried out either by conventional evaluation processes, or by processes more specific to chloroplasts. Furthermore, this last stage can enable to detect multiple solutions especially corresponding to structural haplotypes.

De novo evaluation Bakker et al. (2016) select the best assemblies according to the Assembly Likelihood Estimation (ALE) (Clark et al., 2013). The assembly candidates come from both the final assemblies during the iterative read baiting step and the final step.

Reference genome evaluation Ankenbrand et al. (2018) keep the contigs that align sufficiently against reference genomes.

Multiple solutions Andonov et al. (2019) detect in the DG path, representing the scaffold, pairs of subpaths $p = (v_0, \ldots, v_{n-1})$, $p' = \left(v'_0, \ldots, v'_{n-1}\right)$ such that: $\forall\, i \in [\![0, n[\![, frag(v_i) = \overline{frag(v_{n-1-i})}$, i.e. they detect subpaths that represent inverted repeats (IR). They cut the scaffold at the beginning and at the end of these subpaths in order to consider alternative paths representing structural haplotypes. They prove the alternative paths to be optimal. Jin et al. (2020) produce all the paths consuming all the multiplied contigs. They keep the paths that represent iso-IR, i.e. paths that have the same property as in Andonov et al. (2019).

III FRAGMENT GRAPH IMPLEMENTATIONS AND COMPARISON

David Krakauer & Kathleen Tagg. (2020). The Geyser [Song]. On *Breath & Hammer*. Table Pounding Records

In this chapter

1 Implementations 58
1.1 Directed graph (DG): oriented fragments based 59
1.1.1 All oriented fragments directed graph (DGA) 59
1.1.2 Oriented fragments' successors directed graph (DGS) 61
1.1.3 Forward fragments directed graph (DGF) 63
1.2 Bidirected graph (BG): oriented walk based 66
1.2.1 Unoriented fragments bidirected graph (BGU) 66
1.2.2 Transformation to DGF 68
1.3 Undirected graph (UG): tail-head fragments based 68
1.3.1 All oriented fragments undirected graph (UGA) 68
1.3.2 Transformation to DGS 70
1.4 Fragment graph map 70
2 Algorithms for DGS, DGF and BGU 72
2.1 Subfunctions 72
2.2 Iterating over the predecessors 73
2.3 Iterating over the successors 74
2.4 Adding a vertex 75
2.5 Adding an edge 76
2.6 Deleting a vertex 78
2.7 Deleting an edge 81
3 Time costs 82
4 Memory and time cost comparisons 84

5 Conclusions and perspectives 85

Section 1 describes several graph implementations for each structure from Chapter II, Section 1 (see the fragment graph structures overview in Figure 22). For the most interested ones, Section 2 provides algorithms to explore the graph, to add or to delete vertices and edges. Finally we conclude on the memory and the time costs of the proposed implementations and their associated algorithms. For a quick overview, Figure 30 is a map that groups the graph implementations according to their structures, and summarises their properties.

1 Implementations

The implementations should satisfy both querying and dynamic updating requirements:

Querying requirements

- given an (oriented) fragment, getting all the fragments linking it;
- given a link, answering true if it is represented in the graph.

Dynamic updating requirements

- adding a fragment/link;
- deleting a fragment/link.

A square matrix can represent the fragments and their links. The matrix is sparse in practice, e.g. in the read assembly stage, the reads are overlapping only if they are sequenced from closed genomic region or from repeats. A Compressed Sparse Row (CSR) or a Compressed Sparse Column (CSC) is suitable to address the sparsity. It also enables fast link querying. However, dynamic operations are costly. Thus, we implement the graphs with adjacency lists. They are suitable for both the sparsity and the efficiency of addition or deletion operations. However, as the number of *neighbours* is not *a priori* known, adjacency lists require the use of pointers to memory address, that are costly in memory.

The implementations require suitable graph labelling strategies: the vertices and the edges are labelled with indices, such that their ranges are minimised according to our purpose. For the sake of clarity, in the next sections we consider all the fragments and links are represented. Since for each graph structure there are bijective functions to the fragment set $\mathcal{F}$ and to the link set $\mathcal{L}$, there must be bijective functions from the implementations to the graph structure. In that case, we will ensure the implementations represent the data.

Common notations

Canonical link We described what a forward and a reverse fragments are ($\mathcal{F} = \mathcal{F}_f \sqcup \mathcal{F}_r$) but we have not yet split the link set into two. Analogously to the fragment set, we partition the link set in two, $\mathcal{L} = \mathcal{L}_f \sqcup \mathcal{L}_r$. $\mathcal{L}_f$ contains the links for which in the pair, the fragment with the smallest $\Sigma_{\mathcal{F}}$ label is in forward orientation, i.e. $\mathcal{L}_f = \left\{ l \in \mathcal{L} \mid for(a) = f, fid(a) = \min_{x \in linkp(l)} fid(x) \right\}$. Similarly to the unoriented fragment label set $\Sigma_{\mathcal{F}_{un}}$, let $\Sigma_{\mathcal{L}_{can}} = [\![0, \frac{1}{2}|\mathcal{L}| [\![$ be the label set of canonical links and $canlid\colon \mathcal{L} \twoheadrightarrow \Sigma_{\mathcal{L}_{can}}$ its associated function, such that $\forall\, l \in \mathcal{L}, canlid(l) = canlid(\bar{l})$. In this chapter, a link label represents both a link and its reverse, i.e. a canonical link. The order of the fragments enables distinguishing if the link label corresponds to a link in $\mathcal{L}_f$ or in $\mathcal{L}_r$.

Adjacency list notation The adjacency lists are represented by functions $f\colon \mathbb{N} \to X$, where X can be the set of neighbour lists (themselves represented by functions) or the set of neighbours. From a programming point view, $f[i]$ is the object at the ith position in the list f. Here we denote this object by $f(i)$ except in the pseudocodes. We denote by N_V and N_v the adjacency lists for all the vertices and for the particular vertex v, respectively. N^- and N^+ stand for the *predecessor* and the *successor* lists.

In/out degrees For a vertex $v \in V$, we denote by $d_v = d_v^- + d_v^+$ its total degree equals to the sum of the number of its predecessors and successors.

Octet-isation For $n \in \mathbb{N}$, $oct(n) = \left\lceil \frac{\log_2 n}{8} \right\rceil$ gives the number of octets necessary to memorise an integer between 0 and $n - 1$.

1.1 Directed graph (DG): oriented fragments based

Section 1.1.1 describes the first implementation for DG, while Sections 1.1.2 and 1.1.3 propose clever implementations thanks to the reverse symmetry. Recall from Definition 1.7 that $DG = (V, E, \Phi)$.

1.1.1 All oriented fragments directed graph (DGA)

DGA is a classical multidigraph implementation with both the predecessor and the successor lists. For each vertex (oriented fragment), we associate a unique label. We do the same for the edges (the links). Figure 25 illustrates each link case in DGA.

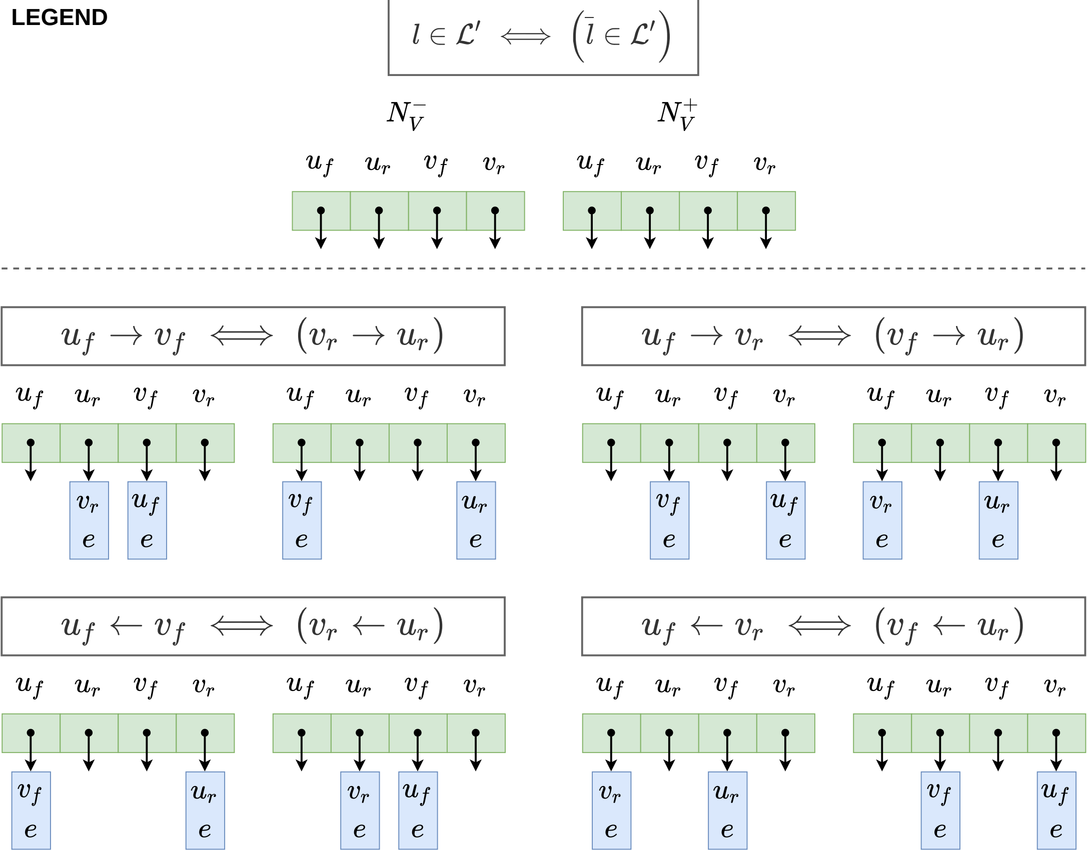


Figure 25 – DGA implementation.
Each squared mathematical formula provides the link case and its reverse (in parentheses). The left and the right lists are the predecessor and the successor lists for all the vertices. For each $v \in V$, $vind(v) = v_f$ if $frag(v) \in \mathcal{F}_f$, v_r otherwise. For each link and its reverse, e is their canonical label.

▶ **Definition 1.1: DGA labels and functions**

Let $vind\colon V \hookrightarrow\!\!\!\rightarrow \Sigma_V$ and $eind\colon E \twoheadrightarrow \Sigma_E$, where $\Sigma_V = \Sigma_{\mathcal{F}}$ and $\Sigma_E = \Sigma_{\mathcal{L}_{can}}$ are the label sets for the vertices and the edges, respectively. The predecessor and the successor lists are defined by two functions:

$$predl\colon \Sigma_V \to N_V^- = \left\{N_v^-, \forall v \in V \mid N_v^-\colon [\![0, d_v^-[\![\to \Sigma_V \times \Sigma_E\right\}$$

$$succl\colon \Sigma_V \to N_V^+ = \left\{N_v^+, \forall v \in V \mid N_v^+\colon [\![0, d_v^+[\![\to \Sigma_V \times \Sigma_E\right\}$$

For each vertex $v \in V$, $N_v^-\colon [\![0, d_v^-[\![\to \Sigma_V \times \Sigma_E$ and $N_v^+\colon [\![0, d_v^+[\![\to \Sigma_V \times \Sigma_E$ give its predecessors and its successors, respectively. The following property

holds:

$$\forall e \in E, \Phi(e) = (u, v), \exists!\, n \in [\![0, d_u^+[\![, \exists!\, m \in [\![0, d_v^-[\![\text{ s.t.}$$
$$succl(vind(u))(n) = (vind(v), eind(e))$$
$$predl(vind(v))(m) = (vind(u), eind(e))$$

Definition 1.2 provides the logic between the label functions and the reverse properties of the vertices and the edges.

▶ **Definition 1.2: DGA reverse labels**

Given two fragments $a_f \in \mathcal{F}_f$ and $a_r = \overline{a_f} \in \mathcal{F}_r$ and their associated vertices v_f and v_r ($frag(v_f) = a_f$, $frag(v_r) = a_r$), the labels of v_f and v_r respect $vind(v_r) - vind(v_f) = 1$. For each edge $e \in E$, its label respects $eind(e) = eind(\overline{e})$.

Proposition 1.1 gives the amount of octet DGA consumes.

▶ **Proposition 1.1: DGA memory consumption**

The memory size $Mem(DGA)$ of DGA (in octets) is equal to:

$$Mem(DGA) = 2\mathsf{P}(|\mathcal{F}| + 1) + 2|\mathcal{L}|\left(oct(|\mathcal{F}|) + oct\left(\frac{1}{2}|\mathcal{L}|\right)\right)$$

where P is the memory size of a memory address.

▷ Proof

There is one pointer for the predecessor and the successor lists, and two for each fragment, i.e. $2\mathsf{P}(|\mathcal{F}| + 1)$*. Then, for each link, for each predecessor and for each successor, the neighbour's label and the edge's label are provided, i.e.* $2|\mathcal{L}|\big(oct(|\mathcal{F}|) + oct(\frac{1}{2}|\mathcal{L}|)\big)$.

◁

1.1.2 Oriented fragments' successors directed graph (DGS)

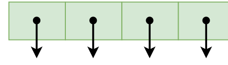
DGS corresponds to DGA without the predecessor lists. The property at the end of Definition 1.3 show how to get the predecessors of a vertex (note the $d^+_{\overline{v}}$). The reverse of the successors of the reverse are the predecessors of the vertex. Figure 26 illustrates each link case in DGS.

LEGEND

$l \in \mathcal{L}' \iff \left(\bar{l} \in \mathcal{L}'\right)$

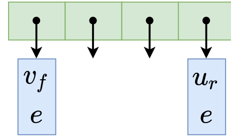
N_V^+

$u_f \quad u_r \quad v_f \quad v_r$



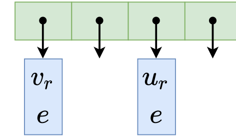
$u_f \to v_f \iff (v_r \to u_r)$

$u_f \quad u_r \quad v_f \quad v_r$



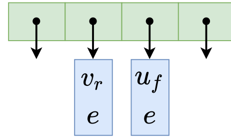
$u_f \to v_r \iff (v_f \to u_r)$

$u_f \quad u_r \quad v_f \quad v_r$



$u_f \leftarrow v_f \iff (v_r \leftarrow u_r)$

$u_f \quad u_r \quad v_f \quad v_r$



$u_f \leftarrow v_r \iff (v_f \leftarrow u_r)$

$u_f \quad u_r \quad v_f \quad v_r$

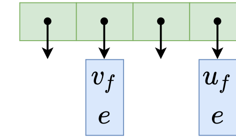


Figure 26 – DGS implementation.
Each squared mathematical formula provides the link case and its reverse (in parentheses). The list bellow is the successor list for all the vertices. The labels u_f, u_r, v_f and v_r are the ones of the vertices associated to two forward and reverse fragments. For each link and its reverse, e is their canonical label.

▶ Definition 1.3: DGS labels and functions

Let $vind\colon V \hookrightarrow\!\!\!\!\to \Sigma_V$ and $eind\colon E \twoheadrightarrow \Sigma_E$, where $\Sigma_V = \Sigma_{\mathcal{F}}$ and $\Sigma_E = \Sigma_{\mathcal{L}_{can}}$ are the label sets for the vertices and the edges, respectively. The successor list is defined by:

$$succl\colon \Sigma_V \to N_V^+ = \left\{N_v^+, \forall\, v \in V \mid N_v^+ \colon [\![0, d_v^+[\![\to \Sigma_V \times \Sigma_E\right\}$$

For each vertex $v \in V$, $N_v^- \colon [\![0, d_v^-[\![\to \Sigma_V \times \Sigma_E$ and $N_v^+ \colon [\![0, d_v^+[\![\to \Sigma_V \times \Sigma_E$ give its predecessors and its successors, respectively. The following property

holds:

$$\forall e \in E, \Phi(e) = (u, v), \exists!\, n \in [\![0, d_u^+[\![, \exists!\, m \in [\![0, d_{\overline{v}}^+[\![\text{ s.t.}$$
$$succl(vind(u))(n) = (vind(v), eind(e))$$
$$succl(vind(\overline{v}))(m) = \big(vind(\overline{u}), eind(\overline{e})\big)$$

Definition 1.4 provides the logic between the label functions and the reverse properties of the vertices and the edges.

▶ Definition 1.4: DGS reverse labels

Given two fragments $a_f \in \mathcal{F}_f$ and $a_r = \overline{a_f} \in \mathcal{F}_r$ and their associated vertices v_f and v_r ($frag(v_f) = a_f$, $frag(v_r) = a_r$), the labels of v_f and v_r respect $vind(v_r) - vind(v_f) = 1$. For each edge $e \in E$, its label respects $eind(e) = eind(\overline{e})$.

Proposition 1.2 gives the amount of octet DGS consumes.

▶ Proposition 1.2: DGS memory consumption

The memory size $Mem(DGS)$ of DGS (in octets) is equal to:

$$Mem(DGS) = \mathsf{P}(|\mathcal{F}| + 1) + |\mathcal{L}|\left(oct(|\mathcal{F}|) + oct\left(\frac{1}{2}|\mathcal{L}|\right)\right)$$

where P is the memory size of a memory address.

▷ Proof

There is one pointer for the successor lists, and one for each fragment, i.e. $\mathsf{P}(|\mathcal{F}| + 1)$*. Then, for each link, for each successor, the neighbour's label and the edge's label are provided, i.e.* $|\mathcal{L}|\big(oct(|\mathcal{F}|) + oct(\frac{1}{2}|\mathcal{L}|)\big)$*.*

◁

1.1.3 Forward fragments directed graph (DGF)

DGF is DGA only for the forward fragments. The neighbours of a reverse, are the reverse neighbours of the forward. The vertex label range is divided by two, so the adjacency lists must provide both the label of the neighbours and their orientation. Figure 27 illustrates each link case in DGF.

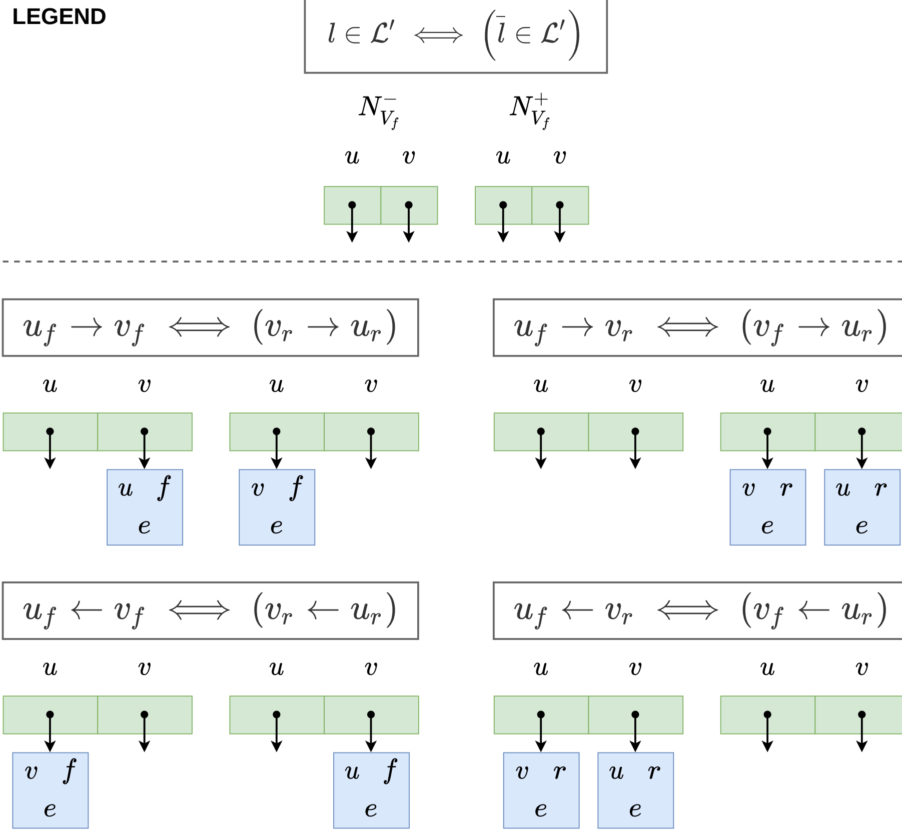


Figure 27 – DGF implementation.
Each squared mathematical formula provides the link case and its reverse (in parentheses). The left and the right lists are the predecessor and the successor lists for the forward vertices. A vertex and its reverse have the same label (v). For each link and its reverse, e is their canonical label. The orientation set $\{f, r\}$ is binarised ($f = 0$, $r = 1$).

▶ Definition 1.5: DGF labels and functions

Let $vind\colon V \twoheadrightarrow \Sigma_{V_f}$ and $eind\colon E \twoheadrightarrow \Sigma_E$, where $\Sigma_{V_f} = \Sigma_{\mathcal{F}_{un}}$ and $\Sigma_E = \Sigma_{\mathcal{L}_{can}}$ are the label sets for the forward vertices and the edges, respectively. The predecessor and the successor lists are defined by two functions:

$$predl\colon \Sigma_V \to N^-_{V_f} = \left\{N^-_v, \forall\, v \in V_f \mid N^-_v\colon [\![0, d^-_v[\![\to \Sigma_{V_f} \times \{f, r\} \times \Sigma_E\right\}$$

$$succl\colon \Sigma_V \to N^+_{V_f} = \left\{N^+_v, \forall\, v \in V_f \mid N^+_v\colon [\![0, d^+_v[\![\to \Sigma_{V_f} \times \{f, r\} \times \Sigma_E\right\}$$

where $V_f = \{v \in V \mid frag(v) \in \mathcal{F}_f\}$. For each forward vertex $v \in V_f$, $N^-_v\colon [\![0, d^-_v[\![\to \Sigma_{V_f} \times \{f, r\} \times \Sigma_E$ and $N^+_v\colon [\![0, d^+_v[\![\to \Sigma_{V_f} \times \{f, r\} \times \Sigma_E$ give its predecessors

and its successors, respectively. The following property holds:

$$\begin{aligned}
&\forall e \in E, \Phi(e) = (u, v),\\
&frag(u) \in \mathcal{F}_f \implies \exists!\, n \in [\![0, d_u^+[\![\text{ s.t.}\\
&\quad succl(vind(u))(n) = (vind(v), for(frag(v)), eind(e))\\
&frag(u) \in \mathcal{F}_r \implies \exists!\, n \in [\![0, d_{\overline{u}}^+[\![\text{ s.t.}\\
&\quad predl(vind(u))(n) = \big(vind(v), for(frag(\overline{v})), eind(\overline{e})\big)\\
&frag(v) \in \mathcal{F}_f \implies \exists!\, m \in [\![0, d_v^-[\![\text{ s.t.}\\
&\quad predl(vind(v))(m) = (vind(u), for(frag(u)), eind(e))\\
&frag(v) \in \mathcal{F}_r \implies \exists!\, m \in [\![0, d_{\overline{v}}^-[\![\text{ s.t.}\\
&\quad succl(vind(v))(m) = \big(vind(u), for(frag(\overline{u})), eind(\overline{e})\big)
\end{aligned}$$

Definition 1.6 provides the logic between the label functions and the reverse properties of the vertices and the edges.

▶ Definition 1.6: DGF reverse labels

For each vertex $v \in V$, its label respects $vind(v) = vind(\overline{v})$. For each edge $e \in E$, its label respects $eind(e) = eind(\overline{e})$.

Proposition 1.3 gives the amount of octet DGF consumes.

▶ Proposition 1.3: DGF memory consumption

The memory size $Mem(DGF)$ of DGF (in octets) is equal to:

$$Mem(DGF) = \mathsf{P}(|\mathcal{F}| + 2) + |\mathcal{L}|\left(oct(|\mathcal{F}|) + oct\left(\frac{1}{2}|\mathcal{L}|\right)\right)$$

where P is the memory size of a memory address.

▷ Proof

There is one pointer for the predecessor and for the successor lists, and two for each forward fragment, i.e. $\mathsf{P}(|\mathcal{F}| + 2)$. *Then, for each link, for each forward predecessors and successors, the neighbour's label, combined with a binary value (f or r), and the edge's label are provided, i.e.* $|\mathcal{L}|\big(oct(|\mathcal{F}|) + oct\big(\frac{1}{2}|\mathcal{L}|\big)\big)$.

◁

1.2 Bidirected graph (BG): oriented walk based

Section 1.2.1 describes the first implementation for BG and Section 1.2.2 describes how to transform BGU to DGF. Recall from Definition 1.10 that $BG = (V, E, \Phi, attre)$.

1.2.1 Unoriented fragments bidirected graph (BGU)

BGU is a classical multiungraph implementation with neighbour lists. The attributes on the edges are stored in a attribute list. For each vertex (unoriented fragment), we associate a unique index. We do the same for the edges (a links and its reverse). Figure 28 illustrates each link case in BGU.

▶ Definition 1.7: BGU labels and functions

Let $vind\colon V \hookrightarrow\!\!\!\!\!\rightarrow \Sigma_V$ and $eind\colon E \hookrightarrow\!\!\!\!\!\rightarrow \Sigma_E$, where $\Sigma_V = \Sigma_{\mathcal{F}_{un}}$ and $\Sigma_E = \Sigma_{\mathcal{L}_{can}}$ are respectively the label sets for the vertices and the edges. The neighbour and the edge attribute lists are defined by two functions:

$$nborl\colon \Sigma_V \to N_V = \{N_v, \forall\, v \in V \mid N_v\colon [\![0, d_v[\![\to \Sigma_V \times \Sigma_E\}$$

$$attrel\colon \Sigma_E \to \{f, r\}^2$$

For each vertex $v \in V$, $N_v\colon [\![0, d_v[\![\to \Sigma_V \times \Sigma_E$ gives its neighbours. The following property holds:

$$\begin{gathered}\forall\, e \in E, \Phi(e) = \{u, v\}, \exists!\; n \in [\![0, d_u[\![, \exists!\; m \in [\![0, d_v[\![\text{ s.t.}\\ nborl(vind(u))(n) = (vind(v), eind(e))\\ nborl(vind(v))(m) = (vind(u), eind(e))\\ attrel(eind(e)) = attre(e)\end{gathered}$$

Proposition 1.4 gives the amount of octet BGU consumes.

▶ Proposition 1.4: BGU memory consumption

The memory size $Mem(BGU)$ of BGU (in octets) is equal to:

$$Mem(BGU) = \mathsf{P}\left(\frac{1}{2}|\mathcal{F}| + 1\right) + |\mathcal{L}|\left(oct\left(\frac{1}{2}|\mathcal{F}|\right) + oct\left(\frac{1}{2}|\mathcal{L}|\right)\right) + oct(|\mathcal{L}|)$$

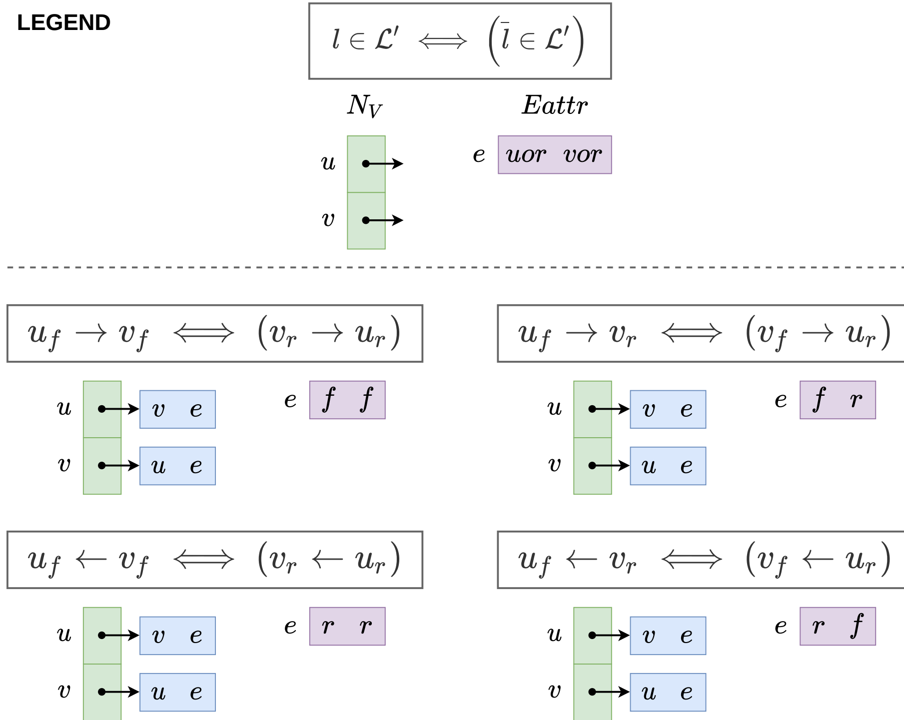


Figure 28 – BGU implementation.
Each squared mathematical formula provides the link case and its reverse (in parentheses). The left and the right lists are the neighbour for all the vertices and the edge attribute list, respectively. For each fragment and its reverse v is their canonical index. For each link and its reverse, e is their canonical index. $Eattr$ provides the orientations associated with the vertices, in the lexicographical order of their label (here $u < v$).

where P is the memory size of a memory address.

▷ Proof

There is one pointer for each vertex and one for the edge attribute list, i.e. $\mathsf{P}\left(\frac{1}{2}|\mathcal{F}| + 1\right)$. *Each vertex is associated with its neighbours and the edge's label. Each edge is stored twice* ($|\mathcal{L}|$). *So it gives* $|\mathcal{L}|\left(oct\left(\frac{1}{2}|\mathcal{F}|\right) + oct\left(\frac{1}{2}|\mathcal{L}|\right)\right)$. *Finally, Eattr provides for each edge two binary values (one bit per orientation), i.e.* $oct(|\mathcal{L}|)$.

◁

1.2.2 Transformation to DGF

It is possible to transform BGU to DGF. For each $u \in \Sigma_V$, for each $(v, e) \in \Sigma_V \times \Sigma_E$ in $cod\ nborl(u)$ (cod is the codomain of function $nborl$) such that $u < v$ implying $attrel(e) = (uor, vor)$:

— if $uor = f$, then append (v, vor, e) to the successor list N_u^+;

— else if $uor = r$, then append $(v, \overline{vor}, e)$ to the predecessor list N_u^-;

— if $vor = f$, then append (u, uor, e) to the predecessor list N_v^-;

— else if $vor = r$, then append $(u, \overline{uor}, e)$ to the successor list N_v^+.

1.3 Undirected graph (UG): tail-head fragments based

Section 1.3.1 describes the first implementation for UG and Section 1.3.2 describes how to transform UGA to DGS. Recall from Definition 1.13 that $UG = (V, E_{\mathcal{F}}, E_{\mathcal{L}}, \Phi_{\mathcal{L}})$.

1.3.1 All oriented fragments undirected graph (UGA)

UGA is a classical multiungraph implementation with neighbour lists. Each fragment is associated with two vertices corresponding to the two extremities (5' and 3'). We provide a unique label for each vertex. There is one neighbour list for the fragment-edges and another one for the link-edges. Figure 29 illustrates each link case in UGA.

▶ Definition 1.8: UGA labels and functions

Let $vind\colon V \hookrightarrow\!\!\!\!\rightarrow \Sigma_V$, and $eind\colon E_{\mathcal{L}} \hookrightarrow\!\!\!\!\rightarrow \Sigma_E$[a], where $\Sigma_V = \Sigma_{\mathcal{F}}$ and $\Sigma_E = \Sigma_{\mathcal{L}_{can}}$ are the label sets for the vertices and the link-edges, respectively. The *fragment-neighbour* function is denoted by $nborlf\colon \Sigma_V \hookrightarrow\!\!\!\!\rightarrow \Sigma_V$. For each vertex $v \in V$, $nborlf(v)$ gives the other extremity of the fragment. The *link-neighbour* list is defined by:

$$nborll\colon \Sigma_V \to N_V^{\mathcal{L}} = \left\{N_v^{\mathcal{L}}, \forall\, v \in V \mid N_v^{\mathcal{L}}\colon [\![0, d_v^{\mathcal{L}}[\![\to \Sigma_V \times \Sigma_E\right\}$$

For each vertex $v \in V$, $N_v^{\mathcal{L}}\colon [\![0, d_v^{\mathcal{L}}[\![\to \Sigma_V \times \Sigma_E$ provides its link-neighbours. The following property holds:

$$\begin{gathered}\forall\, e \in E_{\mathcal{L}}, \Phi_{\mathcal{L}}(e) = \{u, v\}, \exists!\, n \in [\![0, d_u^{\mathcal{L}}[\![, \exists!\, m \in [\![0, d_v^{\mathcal{L}}[\![\text{ s.t.}\\ nborll(vind(u))(n) = (vind(v), eind(e))\\ nborll(vind(v))(m) = (vind(u), eind(e))\end{gathered}$$

[a] We do not provide edge labels for fragment-edges.

Definition 1.9 provides the logic between the label functions and the reverse properties of the vertices.

▶ Definition 1.9: UGA reverse labels

Given a fragments $a_f \in \mathcal{F}_f$ and its reverse $a_r = \overline{a_f} \in \mathcal{F}_r$ and their associated vertices $\{v_t, v_h\} \in E_{\mathcal{F}}$ $(frag(v_t, v_h) = a_f,\ frag(v_h, v_t) = a_r)$, the label of v_t

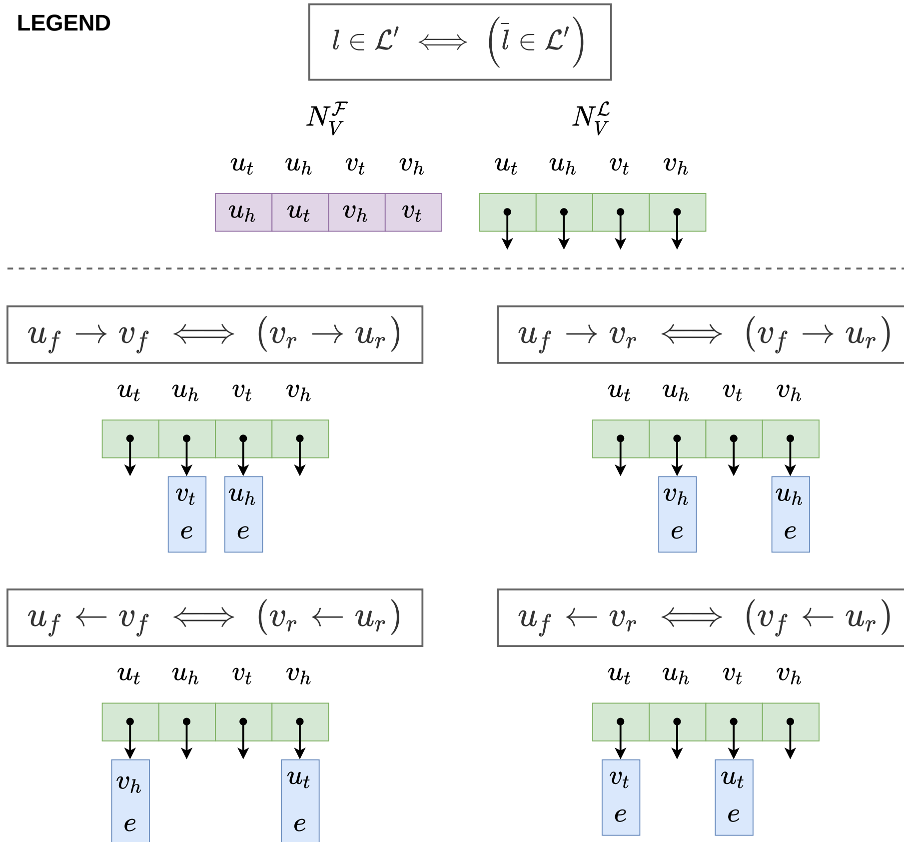


Figure 29 – UGA implementation.
Each squared mathematical formula provides the link case and its reverse (in parentheses). The violet list is not stored in memory because it does not depend of the link case. It corresponds to $nborlf$ function. The $N_V^{\mathcal{L}}$ green list provides the link-neighbours. Vertices v_t and v_h correspond to the tail and the head of a fragment. For each link and its reverse, e is their canonical index.

and v_h respect $vind(v_h) - vind(v_t) = 1$.

Proposition 1.5 gives the amount of octet UGA consumes.

▶ Proposition 1.5: UGA memory consumption

The memory size $Mem(UGA)$ of UGA (in octets) is equal to:

$$Mem(UGA) = \mathsf{P}(|\mathcal{F}| + 1) + |\mathcal{L}|\left(oct(|\mathcal{F}|) + oct\left(\frac{1}{2}|\mathcal{L}|\right)\right)$$

where P is the memory size of a memory address.

▷ Proof

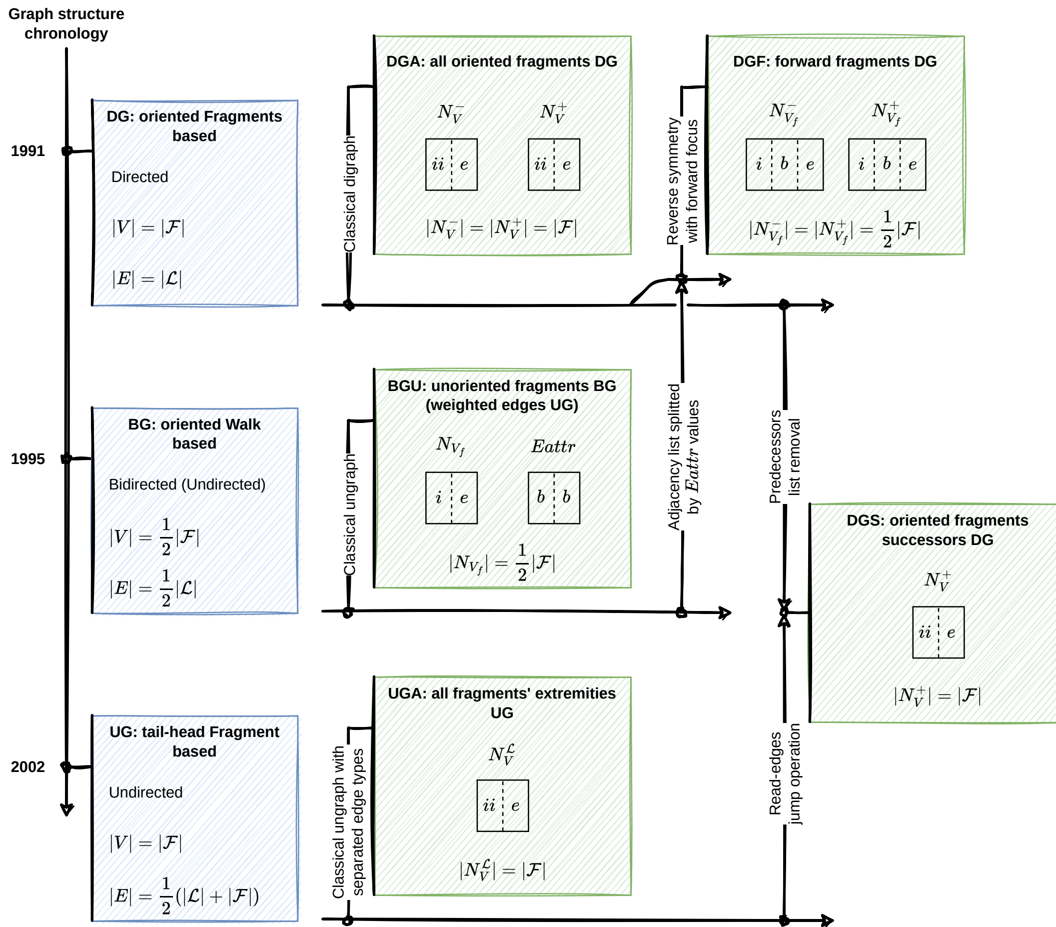
There is one pointer for the link-neighbour lists and one for each fragment extremity, i.e. $\mathsf{P}(|\mathcal{F}| + 1)$. *Then, for each link, for each extremity, the neighbour's label and the edge's label are provided, i.e.* $|\mathcal{L}|\big(oct(|\mathcal{F}|) + oct\big(\frac{1}{2}|\mathcal{L}|\big)\big)$.
◁

1.3.2 Transformation to DGS

UGA can easily be adapted to obtain DGS. For each $v \in \Sigma_V$, for each $n \in [\![0, d_v^{\mathcal{L}}[\![$ where $nborll(v)(n) = (w, e)$, replace w by $nborlf(w)$. In a valid path, a link-edge is followed by a fragment-edge, which is unique for each vertex (see Definition 1.15). Finally, to exactly match DGS, the order of the index for the extremities is reversed, i.e. $vind(v_t) - vind(v_h) = 1$.

1.4 Fragment graph map

Figure 30 – Graph structures and their implementations.
Blue boxes: they correspond to graph structures. They are described by the type of graph they represent, the theoritical number vertices and edges. **Green boxes:** they correspond to discussed graph implementations. After their name, the second line gives the adjacency list(s) definition and the last line gives the length of the adjacency list(s) (i.e. the number of vertices for which the neighbours are given).



LEGEND

1991 (arrow)	Graph structures ordered chronologically
(arrow)	Graph implementations linked by transformations
$\mathcal{F}$	Oriented fragment set
$\mathcal{L}$	Link set
V	Vertex set
E	Edge set

Graph structure

Theoritical graph type

Theoritical number of vertices and edges

Graph implementation

V, V_f	Sets of vertices for all the oriented fragments and for only the forward
N^-, N^+	Lists of predecessors and successors
$N_V^{\mathcal{L}}$, $N_V^{\mathcal{F}}$	Lists of link-edges and fragment-edges
(box)	Adjacency lists
i	integer of range $0 \leq i < \frac{1}{2}|\mathcal{F}|$
ii	integer of range $0 \leq ii < |\mathcal{F}|$
e	integer of range $0 \leq e < \frac{1}{2}|\mathcal{L}|$
b	boolean value $b \in \{0, 1\}$

2 Algorithms for DGS, DGF and BGU

In this section, we give elementary operation algorithms for DGS, DGF and BGU graph implementations. To respect the computing notations, N_V, N_V^- and N_V^+ are the neighbour, predecessor and successor lists, respectively, codomains of the functions *nborl*, *predl* and *succl*. Similarly, for any vertex v we denote by N_v, N_v^- and N_v^+ the neighbour, predecessor and successor lists of v, respectively. Iterating over one of these lists is equivalent to increasing an integer k from 0 to the length of the list minus one and peek the kth element.

2.1 Subfunctions

The following subfunction algorithms are often subparts of the next algorithms. Algorithm 1 is usefull for DGS. Algorithms 2 and 3 are usefull when deleting vertices or edges.

▶ Algorithm 1: Reverse operation on index

Require: Integer ind.
Ensure: Returns $2\lfloor \frac{ind}{2} \rfloor + 1 - (ind \bmod 2)$.

```
1: function REV(ind)
2:     if ind | 2 then
3:         return ind + 1
4:     return ind − 1
```

For the sake of clarity, $\text{REV}(ind) = \overline{ind}$.

▶ Algorithm 2: Find place of edge index in the neighbour list

Require: List of tuple $list$, edge index e, index t of the place of the edge indices in the tuples contained in $list$. The edge index must be in a tuple.
Ensure: Returns the index ind of the tuple containing e in $list$.

```
1: function GET_INDEX(list, e, t)
2:     ind ← 0
3:     while list[ind][t] ≠ e and ind < |list| − 1 do
4:         ind ← ind + 1
5:     return ind
```

▶ Algorithm 3: Delete an entry in a list and arrange it

Require: List $list$, an entry index $ind \in [\![0, |list|[\![$.
Ensure: Only the element in $list[ind]$ is removed, and $|list|$ decreases by one.

```
1: function DELETE_LIST_ENTRY(list, ind)
2:     if ind = |list| − 1 then                    ▷ Just delete the last element
3:         list.POP( )
4:     else                          ▷ Replace the element of interest by the last one
5:         list[ind] ← list.POP( )
```

2.2 Iterating over the predecessors

Given a fragment, we aim to return the fragments linking it upstream associated with the canonical link labels.

▶ Algorithm 4: Iterate over the predecessors for DGS

Require: A vertex label $v \in \Sigma_V$.
Ensure: Returns a list containing tuples in $\Sigma_V \times \Sigma_E$ representing the predecessors of the vertex labelled by v.

```
1: function DGS_PREDS(v)
2:     preds ← EMPTY_LIST( )
3:     for (u, e) ∈ N⁺_v̄ do
4:         preds.APPEND(ū, e)
5:     return preds
```

▶ Algorithm 5: Iterate over the predecessors for DGF

Require: A vertex label $v \in \Sigma_V$ associated with an orientation $vor \in \{f, r\}$.
Ensure: Returns a list containing tuples in $\Sigma_V \times \{f, r\} \times \Sigma_E$ representing the predecessors of the vertex labelled by v with fragment orientation vor.

```
1: function DGF_PREDS(v, vor)
2:     preds ← EMPTY_LIST( )
3:     if vor = f then                                    ▷ Forward fragment
4:         for (u, uor, e) ∈ N⁻_v do
5:             preds.APPEND(u, uor, e)
6:     else                                               ▷ Reverse fragment
7:         for (u, uor, e) ∈ N⁺_v do
```

8: $preds$.APPEND$(u, 1 - uor, e)$
9: **return** $preds$

▶ Algorithm 6: Iterate over the predecessors for BGU

Require: A vertex label $v \in \Sigma_V$ associated with an orientation $vor \in \{f, r\}$ representing the predecessors of the vertex labelled by v and with fragment orientation vor.
Ensure: Returns a list containing tuples in $\Sigma_V \times \{f, r\} \times \Sigma_E$ representing the predecessors of the vertex labelled by v with the fragment orientation vor.

```
1: function BGU__PREDS(v, vor)
2:     preds ← EMPTY__LIST( )
3:     for (u, e) ∈ N_v do
4:         if u < v ∧ Eattr[e][1] = vor then
5:             preds.APPEND(u, Eattr[e][0], e)
6:         else if 1 − Eattr[e][0] = vor then
7:             preds.APPEND(u, 1 − Eattr[e][1], e)
8:     return preds
```

2.3 Iterating over the successors

Given a fragment, we aim to return the fragments linking it downstream associated with the canonical link labels.

▶ Algorithm 7: Iterate over the successors for DGS

Require: A vertex label $v \in \Sigma_V$.
Ensure: Returns a list containing tuples in $\Sigma_V \times \Sigma_E$ representing the successors of the vertex labelled by v.

```
1: function DGS__SUCCS(v)
2:     succs ← EMPTY__LIST( )
3:     for (w, e) ∈ N_v^+ do
4:         succs.APPEND(w, e)
5:     return succs
```

▶ Algorithm 8: Iterate over the successors for DGF

Require: A vertex label $v \in \Sigma_V$ associated with an orientation $vor \in \{f, r\}$.
Ensure: Returns a list containing tuples in $\Sigma_V \times \{f, r\} \times \Sigma_E$ representing the predecessors of the vertex labelled by v with fragment orientation vor.

1: **function** DGF_SUCCS(v, vor)
2: $succs \leftarrow$ EMPTY_LIST( )
3: **if** $vor = f$ **then** ▷ *Forward fragment*
4: **for** $(w, wor, e) \in N_v^+$ **do**
5: $succs$.APPEND(w, wor, e)
6: **else** ▷ *Reverse fragment*
7: **for** $(w, wor, e) \in N_v^-$ **do**
8: $succs$.APPEND$(w, 1 - wor, e)$
9: **return** $succs$

▶ Algorithm 9: Iterate over the successors for BGU

Require: A vertex label $v \in \Sigma_V$ associated with an orientation $vor \in \{f, r\}$ representing the successors of the vertex labelled by v and with fragment orientation vor.
Ensure: Returns a list containing tuples in $\Sigma_V \times \{f, r\} \times \Sigma_E$ representing the predecessors of the vertex labelled by v with the fragment orientation vor.

1: **function** BGU_SUCCS(v, vor)
2: $succs \leftarrow$ EMPTY_LIST( )
3: **for** $(w, e) \in N_v$ **do**
4: **if** $v < w \wedge Eattr[e][0] = vor$ **then**
5: $succs$.APPEND$(w, Eattr[e][1], e)$
6: **else if** $1 - Eattr[e][1] = vor$ **then**
7: $succs$.APPEND$(w, 1 - Eattr[e][0], e)$
8: **return** $succs$

2.4 Adding a vertex

The following algorithms allocate a place for new vertices by providing their label.

▶ Algorithm 10: Add a new vertex in the graph for DGS

Ensure: The length of N_V^+ is increased by two and returns the last vertex label in fragment orientation forward.

1: **function** DGS_ADD_VERTEX( )
2: N_V^+.APPEND(EMPTY_LIST( ))
3: N_V^+.APPEND(EMPTY_LIST( ))
4: **return** $\left|N_V^+\right| - 2$

▶ Algorithm 11: Add a new vertex in the graph for DGF

Ensure: The lengths of N_V^- and N_V^+ are increased by one and returns the last vertex label.

1: **function** DGF_ADD_VERTEX( )
2: N_V^-.APPEND(EMPTY_LIST( ))
3: N_V^+.APPEND(EMPTY_LIST( ))
4: **return** $\left|N_V^+\right| - 1$

▶ Algorithm 12: Add a new vertex in the graph for BGU

Ensure: The length of N_V is increased by one and returns the last vertex label.

1: **function** BGU_ADD_VERTEX( )
2: N_V.APPEND(EMPTY_LIST( ))
3: **return** $|N_V| - 1$

2.5 Adding an edge

The following algorithms append new edges by providing their label.

▶ Algorithm 13: Add a new edge and its reverse in the graph for DGS

Require: Two vertex labels u and v.

Ensure: Returns the new edge index e such that $(u, e) \in$ PREDS(v) and $(v, e) \in$ SUCCS(u).

1: **function** DGS_ADD_EDGE(u, v)
2: ▷ *ind_edges is the number of edges. It is always even.* ◁
3: N_u^+.APPEND(v, ind_edges)
4: $N_{\overline{v}}^+$.APPEND$(\overline{u}, ind_edges)$

5: $ind_edges \leftarrow ind_edges + 1$
6: $card_edges \leftarrow card_edges + 2$
7: **return** $ind_edges - 1$

▶ Algorithm 14: Add a new edge and its reverse in the graph for DGF

Require: Two vertex labels u and v, associated with their respective fragment orientation uor and vor.
Ensure: Returns the new edge index e such that $(u, uor, e) \in \text{PREDS}(v, vor)$ and $(v, vor, e) \in \text{SUCCS}(u, uor)$.

1: **function** $\text{DGF__ADD__EDGE}(u, uor, v, vor)$
2: ▷ *ind_edges is the number of edges. It is always even.* ◁
3: **if** $uor = f$ **then** ▷ $u_f \to v_f$ *or* $u_f \to v_r$
4: $N_u^+.\text{APPEND}(v, vor, ind_edges)$
5: **else** ▷ $u_f \leftarrow v_f$ *or* $u_f \leftarrow v_r$
6: $N_u^-.\text{APPEND}(v, 1 - vor, ind_edges)$
7: **if** $vor = f$ **then** ▷ $u_f \to v_f$ *or* $u_r \to v_f$
8: $N_v^-.\text{APPEND}(u, uor, ind_edges)$
9: **else** ▷ $u_f \leftarrow v_f$ *or* $u_r \leftarrow v_f$
10: $N_v^+.\text{APPEND}(u, 1 - uor, ind_edges)$
11: $ind_edges \leftarrow ind_edges + 1$
12: $card_edges \leftarrow card_edges + 2$
13: **return** $ind_edges - 1$

▶ Algorithm 15: Add a new edge and its reverse in the graph for BGU

Require: Two vertex labels u and v, associated with their respective fragment orientation uor and vor.
Ensure: Returns the new edge index e such that $(u, uor, e) \in \text{PREDS}(v, vor)$ and $(v, vor, e) \in \text{SUCCS}(u, uor)$.

1: **function** $\text{BGU__ADD__EDGE}(u, uor, v, vor)$
2: $N_u.\text{APPEND}(v, ind_edges)$
3: $N_v.\text{APPEND}(u, ind_edges)$
4: **if** $u < v$ **then**
5: $Eattr.\text{APPEND}(uor, vor)$
6: **else**
7: $Eattr.\text{APPEND}(1 - vor, 1 - uor)$
8: $ind_edges \leftarrow ind_edges + 1$
9: $card_edges \leftarrow card_edges + 1$

10: **return** $ind_edges - 1$

2.6 Deleting a vertex

The following algorithms delete a vertex (and its reverse where appropriate) and all their edges.

▶ Algorithm 16: Delete a vertex for DGS

Require: A vertex label v such that $v \mid 2$.
Ensure: The edges containing the vertex or its reverse are deleted from the neighbour lists. The length of N_V^+ is reduced by two.

```
1: function DGS_DELETE_VERTEX(v)
2:     ▷ Delete it from its predecessors ◁
3:     v_rev ← v + 1
4:     for (u_rev, e) ∈ N+_{v_rev} do
5:         u ← overline(u_rev)
6:         adj_ind ← GET_INDEX(N+_u, e, 1)
7:         DELETE_LIST_ENTRY(N+_u, adj_ind)
8:         card_edges ← card_edges − 2
9:     DELETE(N+_{v_rev})
10:    ▷ Delete it from its successors ◁
11:    for (w, e) ∈ N+_v do
12:        w_rev ← overline(w)
13:        adj_ind ← GET_INDEX(N+_{w_rev}, e, 1)
14:        DELETE_LIST_ENTRY(N+_{w_rev}, adj_ind)
15:        card_edges ← card_edges − 2
16:    DELETE(N+_v)
17:    ▷ Delete the whole vertex ◁
18:    if v = |N+_V| − 2 then
19:        ▷ It is the last, just pop it and its reverse ◁
20:        N+_V.POP( )
21:        N+_V.POP( )
22:    else
23:        ▷ Replace it and its reverse by the last and its reverse ◁
24:        N+_{v_rev} ← N+_V.POP( )
25:        N+_v ← N+_V.POP( )
26:        ▷ Update the vertex index ◁
```

27: **for** $(u_rev, e) \in N^+_{v_rev}$ **do**
28: $u \leftarrow \overline{u_rev}$
29: $adj_ind \leftarrow \text{GET_INDEX}(N^+_u, e, 1)$
30: $N^+_u[adj_ind] \leftarrow (v, e)$
31: **for** $(w_rev, e) \in N^+_v$ **do**
32: $w \leftarrow \overline{w_rev}$
33: $adj_ind \leftarrow \text{GET_INDEX}(N^+_w, e, 1)$
34: $N^+_w[adj_ind] \leftarrow (v_rev, e)$

▶ Algorithm 17: Delete a vertex for DGF

Require: A vertex label v.
Ensure: The edges containing the vertex or its reverse are deleted from the neighbour lists. The lengths of N^-_V and N^+_V are reduced by one.
1: **function** DGF_DELETE_VERTEX(v)
2: ▷ *Delete it from its predecessors* ◁
3: **for** $(u, uor, e) \in N^-_v$ **do**
4: **if** $uor = f$ **then**
5: $adj_ind \leftarrow \text{GET_INDEX}(N^+_u, e, 1)$
6: $\text{DELETE_LIST_ENTRY}(N^+_u, adj_ind)$
7: **else**
8: $adj_ind \leftarrow \text{GET_INDEX}(N^-_u, e, 1)$
9: $\text{DELETE_LIST_ENTRY}(N^-_u, adj_ind)$
10: $card_edges \leftarrow card_edges - 2$
11: $\text{DELETE}(N^-_v)$
12: ▷ *Delete it from its successors* ◁
13: **for** $(w, wor, e) \in N^+_v$ **do**
14: **if** $wor = f$ **then**
15: $adj_ind \leftarrow \text{GET_INDEX}(N^-_w, e, 1)$
16: $\text{DELETE_LIST_ENTRY}(N^-_w, adj_ind)$
17: **else**
18: $adj_ind \leftarrow \text{GET_INDEX}(N^+_w, e, 1)$
19: $\text{DELETE_LIST_ENTRY}(N^+_w, adj_ind)$
20: $card_edges \leftarrow card_edges - 2$
21: $\text{DELETE}(N^+_v)$
22: ▷ *Delete the whole vertex* ◁
23: **if** $v = \left|N^+_V\right| - 1$ **then**
24: ▷ *It is the last, just pop it and its reverse* ◁
25: $N^-_V.\text{POP}(\,)$

26: N_V^+.POP( )
27: **else**
28: ▷ *Replace it and its reverse by the last and its reverse* ◁
29: $N_v^- \leftarrow N_V^-$.POP( )
30: $N_v^+ \leftarrow N_V^+$.POP( )
31: ▷ *Update the vertex index* ◁
32: **for** $(u, uor, e) \in N_v^-$ **do**
33: **if** $uor = f$ **then**
34: $adj_ind \leftarrow$ GET_INDEX$(N_u^+, e, 1)$
35: $N_u^+[adj_ind] \leftarrow (v, f, e)$
36: **else**
37: $adj_ind \leftarrow$ GET_INDEX$(N_u^-, e, 1)$
38: $N_u^-[adj_ind] \leftarrow (v, r, e)$
39: **for** $(w, wor, e) \in N_v^+$ **do**
40: **if** $wor = f$ **then**
41: $adj_ind \leftarrow$ GET_INDEX$(N_w^-, e, 1)$
42: $N_w^-[adj_ind] \leftarrow (v, f, e)$
43: **else**
44: $adj_ind \leftarrow$ GET_INDEX$(N_w^+, e, 1)$
45: $N_w^+[adj_ind] \leftarrow (v, r, e)$

▶ Algorithm 18: Delete a vertex for BGU

Require: A vertex label v.

Ensure: The edges containing the vertex or its reverse are deleted from the neighbour lists. The length of N_V is reduced by one.

1: **function** BGU_DELETE_VERTEX(v)
2: **for** $(w, e) \in N_v$ **do**
3: $adj_ind \leftarrow$ GET_INDEX$(N_w, e, 1)$
4: DELETE_LIST_ENTRY(N_w, adj_ind)
5: $card_edges \leftarrow card_edges - 1$
6: DELETE(N_v)
7: ▷ *Delete the whole vertex* ◁
8: **if** $v = |N_V| - 1$ **then**
9: ▷ *It is the last, just pop it* ◁
10: N_V.POP( )
11: **else**
12: ▷ *Replace it by the last* ◁
13: $N_v \leftarrow N_V$.POP( )

14: ▷ *Update the vertex index* ◁
15: **for** $(w, e) \in N_v$ **do**
16: $adj_ind \leftarrow$ GET_INDEX$(N_w, e, 1)$
17: $N_w[adj_ind] \leftarrow (v, e)$
18: **if** $w > v$ **then**
19: ▷ *The last (the greatest) identifier becomes a fewer and breaks the identifier order* ◁
20: $Eattr[e] \leftarrow (1 - Eattr[e][1], 1 - Eattr[e][0])$

2.7 Deleting an edge

The following algorithms delete an edge (and its reverse where appropriate).

▶ Algorithm 19: Delete an edge for DGS

Require: Two vertex labels u and v and a edge label.
Ensure: The edges are deleted for the two vertices.
1: **function** DGS_DELETE_EDGE(u, v, e)
2: ▷ *Remove v from u's successors* ◁
3: $adj_ind \leftarrow$ GET_INDEX$(N_u^+, e, 1)$
4: DELETE_LIST_ENTRY(N_u^+, adj_ind)
5: ▷ *Remove u from v's predecessors.* ◁
6: $v_rev \leftarrow \overline{v}$
7: $adj_ind \leftarrow$ GET_INDEX$(N_{v_rev}^+, e, 1)$
8: DELETE_LIST_ENTRY$(N_{v_rev}^+, adj_ind)$
9: $card_edges \leftarrow card_edges - 2$

▶ Algorithm 20: Delete an edge for DGF

Require: Two vertex labels u and v associated with their fragment orientation uor and vor, respectively, and a edge label.
Ensure: The edges are deleted for the two vertices.
1: **function** DGF_DELETE_EDGE(u, uor, v, vor, e)
2: ▷ *Remove v from u's successors* ◁
3: **if** $uor = f$ **then**
4: $list_succs \leftarrow N_u^+$
5: **else**
6: $list_succs \leftarrow N_u^-$

```
7:      adj_ind ← GET_INDEX(list_succs, e, 2)
8:      DELETE_LIST_ENTRY(list_succs, adj_ind)
9:      ▷ Remove u from v's predecessors.                    ◁
10:     if vor = f then
11:         list_preds ← N_v^-
12:     else
13:         list_preds ← N_v^+
14:     adj_ind ← GET_INDEX(list_preds, e, 2)
15:     DELETE_LIST_ENTRY(list_preds, adj_ind)
16:     card_edges ← card_edges − 2
```

▶ Algorithm 21: Delete an edge for BGU

Require: Two vertex labels u and v and a edge label.
Ensure: The edges are deleted for the two vertices.

```
1: function BGU_DELETE_EDGE(u, v, e)
2:      ▷ Remove v from u's neighbours.                      ◁
3:      adj_ind ← GET_INDEX(N_u, e, 1)
4:      DELETE_LIST_ENTRY(N_u, adj_ind)
5:      ▷ Remove u from v's neighbours.                      ◁
6:      adj_ind ← GET_INDEX(N_v, e, 1)
7:      DELETE_LIST_ENTRY(N_v, adj_ind)
8:      card_edges ← card_edges − 1
```

3 Time costs

In this section we provide the time costs for each of the algorithms in Section 2, in the best, worst and average cases. In the average cases, we consider all the events to be equiprobable. Table 4 details the cost for basic operations. In order to easily compare the algorithms' theoritical speed, we assume there is an even integer k such that each fragment has k upstream and k downstream fragments linking it. It holds if $|\mathcal{F}| >> k$ and $|\mathcal{L}| >> k$.

Table 5 gives the algorithmic costs of subfunctions in Algorithms 1 and 2. Table 6 provides the algorithmic costs for Algorithms 4 to 9. Table 7 gives the algorithmic costs of Algorithms 10 to 15. Table 8 gives the algorithmic costs of Algorithms 16 to 18. Table 9 gives the algorithmic costs of Algorithms 19 to 21.

Table 4 – Calculus detail of basic operations costs.
$c(x)$ is the cost of operation x.

Operation	Cost	Description
x	0	Memory access
$x \pm y$	$1 + c(x) + c(y)$	Basic operation
$x \leftarrow y$	$1 + c(y)$	Affectation
$\lvert x \rvert$	$1 + c(x)$	Absolute value or element's length
if $x \lesseqqgtr y$	$1 + c(x) + c(y)$	Conditional
$\text{EMPTY_LIST}()$	1	Empty list constructor
$list.\text{APPEND}(x)$	$1 + c(x)$	Add x to the end of the list $list$
$list.\text{POP}()$	1	Delete the last element of the list $list$
$\text{DELETE}(list)$	$\lvert list \rvert$	Delete all the list
for $i \in [\![a, b]\!]$ **do** x	$(b - a + 1) \times c(x)$	For loop

Table 5 – Algorithmic costs for subfunctions.
$\lvert list \rvert$ is the length of the input list.

	Best	Worst	Average
REV		2	
GET_INDEX	4	$1 + 5\lvert list \rvert$	$1 + 5\left\lceil \frac{\lvert list \rvert}{2} \right\rceil$
DELETE_LIST_ENTRY	3	4	$4 - \frac{1}{\lvert list \rvert}$

Table 6 – Algorithmic costs of iterating over the neighbours.

(a) Iterating over the predecessors

	Best	Worst	Average
DGS		$3k + 4$	
DGF	$k + 3$	$2k + 3$	$\frac{3}{2}k + 3$
BGU	$6k + 2$	$12k + 2$	$\frac{17}{2}k + 2$

(b) Iterating over the successors

	Best	Worst	Average
DGS		$k + 2$	
DGF	$k + 3$	$2k + 3$	$\frac{3}{2}k + 3$
BGU	$6k + 2$	$12k + 2$	$\frac{17}{2}k + 2$

Table 7 – Algorithmic costs of adding a vertex or an edge.

(a) Adding a vertex

	Best	Worst	Average
DGS		5	
DGF		5	
BGU		3	

(b) Adding an edge

	Best	Worst	Average
DGS		11	
DGF	9	11	10
BGU	9	11	10

Table 8 – Algorithmic costs of deleting a vertex.

	Best	Worst	Average
DGS	$28k+6$	$20k^2+36k+8$	$10\frac{\lvert\mathcal{F}\rvert-1}{\lvert\mathcal{F}\rvert}k^2+36\frac{\lvert\mathcal{F}\rvert-\frac{2}{3}}{\lvert\mathcal{F}\rvert}k+\frac{1-\lvert\mathcal{F}\rvert}{\lvert\mathcal{F}\rvert}$
DGF	$24k+4$	$20k^2+28k+6$	$10\frac{\lvert\mathcal{F}\rvert-1}{\lvert\mathcal{F}\rvert}k^2+28\frac{\lvert\mathcal{F}\rvert-\frac{4}{7}}{\lvert\mathcal{F}\rvert}k+4\frac{\lvert\mathcal{F}\rvert-1}{\lvert\mathcal{F}\rvert}$
BGU	$22k+3$	$40k^2+38k+7$	$20\frac{\lvert\mathcal{F}\rvert-1}{\lvert\mathcal{F}\rvert}k^2+24\frac{\lvert\mathcal{F}\rvert-\frac{1}{2}}{\lvert\mathcal{F}\rvert}k+\frac{11}{2}\frac{\lvert\mathcal{F}\rvert-\frac{14}{11}}{\lvert\mathcal{F}\rvert}$

Table 9 – Algorithmic costs of deleting an edge.

	Best	Worst	Average
DGS	21	$10k+17$	$5k+17-\frac{2}{k}$
DGF	22	$10k+18$	$5k+18-\frac{2}{k}$
BGU	18	$20k+14$	$10k+14-\frac{1}{k}$

4 Memory and time cost comparisons

Table 10 provides memory consumption and elementary operation time cost comparisons between *DGS*, *DGF* and *BGU*. Each row presents a score ranging from one to three stars for the three implementations, simplifying the comparison under specific conditions. In the same row, the cost functions of *DGS*, *DGF* and *BGU* are all polynomial functions with the same degree. The polynomial function with the smallest coefficient on the highest degree is used as the basis for comparison. Let f_{min} be this function, and g and h be the two others. Let $c = \lim_{x\to\infty} \frac{g(x)}{f_{min}(x)}$ and $d = \lim_{x\to\infty} \frac{h(x)}{f_{min}(x)}$ be the results of the asymptotic comparisons. The scores are determined whether c and d belong to the following intervals: ⋆⋆⋆ $[1, 1.2]$; ⋆⋆ $[1.2, 2[$; ⋆ $[2, +\infty[$.

Table 10 – Comparison between DGS, DGF and BGU.
Each implementation is associated with starred scores equal to 1 (bad, in red), 2 (medium, in orange) or 3 (good, in green). Two implementations can have the same score if their memory or algorithmic costs are near-equal, and if the third one's cost is far away from them. For each elementary graph operation, for each of the three cases (best, worst, average), the stars correspond to a score.

	DGS	DGF	BGU
Memory consumption			
	⋆	⋆	⋆⋆⋆
Iterating over the predecessors			
Best	⋆	⋆⋆⋆	⋆
Worst	⋆⋆	⋆⋆⋆	⋆
Average	⋆	⋆⋆⋆	⋆
Iterating over the successors			
Best	⋆⋆⋆	⋆⋆⋆	⋆
Worst	⋆⋆⋆	⋆	⋆
Average	⋆⋆⋆	⋆⋆	⋆
Adding a vertex			
Best	⋆⋆	⋆⋆	⋆⋆⋆
Worst	⋆⋆	⋆⋆	⋆⋆⋆
Average	⋆⋆	⋆⋆	⋆⋆⋆
Adding an edge			
Best	⋆⋆	⋆⋆⋆	⋆⋆⋆
Worst	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆
Average	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆
Deleting a vertex			
Best	⋆⋆	⋆⋆⋆	⋆⋆⋆
Worst	⋆⋆⋆	⋆⋆⋆	⋆
Average	⋆⋆⋆	⋆⋆⋆	⋆
Deleting an edge			
Best	⋆⋆⋆	⋆⋆	⋆⋆⋆
Worst	⋆⋆⋆	⋆⋆⋆	⋆
Average	⋆⋆⋆	⋆⋆⋆	⋆

5 Conclusions and perspectives

Graphs are mathematical structures that are useful to store links between two fragments. Two main stages of the fragment assembly employ graphs to handle

the input data or to output the result. In the read assembly stage, especially in the OLC approach, the fragments correspond to the reads and the links are the overlaps between two reads. In the scaffolding stage, the fragments represent the contigs and links can come from paired-end read or long read alignments against the contigs.

The literature describes three graph structures: a digraph, a bigraph and an ungraph. While the first one highlights visually the double-strand sequencing, its weakness is that it requires two vertices for each read (Kececioglu, 1991). Myers (1995) is the first to employ a bigraph to store overlaps. The key idea is to aggregate the two orientations of a read into only one vertex, a link and its reverse into only one edge. Finally, the ungraph structure associates one vertex for each of the two extremities of a fragment to simplify the graph traversal (Huson et al., 2002). The ungraph handles two sets of edges. A fragment-edge connects the two vertices corresponding to the two extremities of a fragment (the tail and the head). Two vertices representing the extremities of a fragment are connected by a fragment-edge. The second one is the multiset of link-edges. A link-edge represents both a link and its reverse.

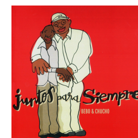
Although some authors have provided conceptual descriptions of these graph structures, the latter have never been compared before from an implementation stat point of view. In this thesis, we have proposed an implementation design based on adjacency lists for each graph structure. At the opposite of a compressed sparse row or column, the adjacency lists are more suitable for adding or deleting vertices or edges in the graphs. We have also described transformation processes to pass from one implementation to another. We then have visualised the graph structures and their implementations in a map.

We have retained three implementations: *DGS* and *DGF* for the multidigraph and *BGU* for the multibigraph. After we have compared their memory consumption, we theoretically measure the cost function for each of their algorithm on elementary operations, such as iterating over the neighbours, adding or deleting a vertex or an edge. We come to the conclusion that if memory is the critical issue, then the *BGU* implementation should be preferred. Otherwise, we recommend *DGS* and *DGF* for iterating over the neighbours and for deleting a vertex or an edge. *BGU* is preferable for adding vertices or edges. To conclude, *DGF* proves to be well-balanced and ideally tailored for dynamic graph operations. *DGF* is available in a `Python3` package named `revsymg`[1] and is easily installable via PyPI.

[1] https://pypi.org/project/revsymg/

IV GLOBAL EXACT OPTIMISATIONS FOR CHLOROPLAST STRUCTURAL HAPLOTYPE SCAFFOLDING

Bebo & Chucho. (2008). Tres Palabras [Song]. On *Juntos Para Siempre*. Sony Music | Latin



In this chapter

1 Introduction 88

1.1 Chloroplast genome specificities . . . 88

1.2 State-of-the-art . . . 89

1.3 Our approach . . . 89

2 Input data and notation 90

2.1 Set of contigs $\mathcal{C}$ . . . 90

2.2 Set of links $\mathcal{L}$ . . . 91

2.3 Mathematically defining genomic regions . . . 91

3 Chloroplast scaffolding problem formulations 92

4 Graph and repeated fragment sets 96

4.1 Graph structure . . . 97

4.2 Repeated fragment sets . . . 98

5 Integer Linear Programming (ILP) formulation 102

5.1 Circuit constraints . . . 103

5.2 Repeated regions constraints . . . 104

5.3 Fixing regions constraints . . . 107

5.4 Speed-up constraints . . . 108

5.5 Scaffolding problems ILP . . . 108

6 Hierarchical problem succession 109
7 From an ILP solution to a genome structure 110
8 Multiple genome forms 114
9 $\mathcal{NP}$-completeness 117
10 Numerical results 121
10.1 Complexity validation on artificial data 122
10.1.1 Perfect artificial data. 122
10.1.2 Noisy artificial data 122
10.2 Synthetic chloroplast input data. 123
10.2.1 Input data generation 123
10.2.2 The evaluation's metrics 125
10.2.3 Initial version 126
10.2.4 Modified version 129
11 Conclusion 130
12 Discussion and perspectives 131

In this chapter, we focus on the scaffolding problem specific to chloroplast genomes. We first recall the specificities of these genomes as described in the introduction (Chapter I, Section 2), highlight the state-of-the-art in chloroplast genome assembly (Chapter II, Section 3) and present our approach.

1 Introduction

1.1 Chloroplast genome specificities

Chloroplasts are plants' organelles derived from the integration of a cyanobacterium in an eukaryotic host. They conduct photosynthesis, a process to convert light energy into chemical energy. Over the evolution time, the chloroplast genome has reduced in length and loosed in terms of complexity (Xiao-Ming et al., 2017). As a result, chloroplast genomes possess few repeats that are usually identical in nucleotide sequences. One the most studied forms of chloroplast genome is a circular quadripartite DNA molecule. It consists of four regions: two identical (or highly similar) nucleotide subsequences, separated by two single-copies (Long Single-Copy, LSC, and Short Single-Copy, SSC) (Palmer, 1985; Bock and Knoop, 2012). There are two types of repeats: (i) the Direct Repeat (DR), where the sequences are highly similar; (ii) the Inverted Repeat (IR), where one sequence is

the reverse-complement of the other. Figure 5 illustrates the common chloroplast genome architectures.

Furthermore, each chloroplast has multiple copies of its genome, and the molecular forms of the copies differ (*structural haplotypes* leading to heteroplasmy, and multigenomic structures – not discussed here, Palmer 1983; Bendich 2004). This phenomenon can be induced by flip-flop inversion: one subsequence is reverse-complemented (*reversed*) during the DNA replication. This inversion is provoked by the existence of facing inverted repeats on either side of the reversed subsequence.

1.2 State-of-the-art

Although there are chloroplast genome assemblers and scaffolders, they do not fully exploit the chloroplast genome's specificities. Some of them are pipelines of generic methods applied on cleaned input dataset (Ankenbrand et al., 2018), or based on locally approaches as seed-and-extend algorithms (Coissac et al., 2016; Dierckxsens et al., 2017). Jin et al. (2020), in `GetOrganelle`, statistically compute the contigs' multiplicities by minimising the squared distance between them and the mapping coverage by the reads.

Concerning the handling of the distinct genome forms, `GetOrganelle` returns several solutions and explores in post-process the corresponding architectures. In Andonov et al. (2019) the flip-flop inversion breakpoints are detected in a post-scaffolding-process, and new optimal solutions can be constructed in polynomial time.

1.3 Our approach

We raise the following two questions: (i) how to mathematically model chloroplast genomic biological knowledge? (ii) How to reveal the structural haplotypes through the scaffolding problem formulation?

To formalise the scaffolding by integrating the structural haplotypes in its core, we postulate on the particularities of the chloroplast genome: (i) repeats are pairs of regions; (ii) the two regions of a repeat have identical (or reversed) nucleotide sequence; (iii) structural haplotypes can be seen as permutations in a sequence of oriented contigs.

We first describe the input data for our approach and provide mathematical definitions (Section 2). Based on the above three assumptions, we propose a new formulation for scaffolding chloroplast genomes without requiring any distances (Section 3). Our approach is region-driven and focuses on retrieving the repeats first. We model the optimisation problem on a digraph where we apply several ILP (Sections 4 and 5). We then detail how we combine the ILP solutions (Section 6).

The ILP's solutions and the digraph correspond to an oriented contig sequence representing one genome's form (Section 7). We partition this sequence into genomic regions we express in a region graph. This graph enables us to return multiple genome forms (Section 8).

We prove the decision version of the chloroplast genome scaffolding problem to be $\mathcal{NP}$-complete (Section 9). However, we exactly solve the problem by profiting from the small size of the chloroplast instances and providing some numerical results (Section 10). We finally conclude (Sections 11 and 12).

2 Input data and notation

Table 11 – Toy example of input data.
(a) Set of contigs $\mathcal{C}$. **(b)** Set of links $\mathcal{L}$. Each row represents a link. For the sake of space, for no one of the links in the table, its reverse is given, although it belongs to $\mathcal{L}$.

(a) Contig set $\mathcal{C}$

contig	mult	wex
a	1	0.70
b	2	0.83
c	2	0.17
d	1	0.43

(b) Link set $\mathcal{L}$

contig	**orient**	**contig**	**orient**
a	f	c	r
a	r	c	r
b	r	c	f
b	f	d	f
b	f	d	r

2.1 Set of contigs $\mathcal{C}$

Contigs are words in the nucleotides DNA alphabet ${\Sigma_{nuc}}^+$. A contig can occur in the genome up to an integer called *multiplicity*. Function $mult\colon \mathcal{C} \to \mathbb{N}_{>0}$ provides its value. Any of contig's *occurrence* can appear in one of two possible reverse-complementary and mutually exclusive *orientations*: forward ($f = 0$) and reverse ($r = 1$). Each contig is provided with an *existence-weight* in $\mathbb{R}_{\geq 0}$ given by the function wex. The weight is proportional to the number of times a contig aligns with chloroplast sequences from a given set (from related or unrelated species). Finally, one contig in this set is defined as the *starter* (s) that must uniquely participate in the genome ($mult(s) = 1$). The starter is a contig whose sequence matches a sequence shared by most chloroplast genomes in a single-copy. Table 11a gives an example of contig set.

2.2 Set of links $\mathcal{L}$

Each link is an ordered pair of oriented contigs. We denote by $\mathcal{L} \subset (\mathcal{C} \times \{f, r\})^2$ the link set[1]. The nature of the double-strands DNA requires that $\forall (c, d) \in \mathcal{L}, (\overline{d}, \overline{c}) \in \mathcal{L}$, where $\overline{c}$ and $\overline{d}$ denote the oriented contigs c and d in their reverse orientation, respectively (note that $\overline{\overline{c}} = c$). The links between two oriented contigs c and d are valid for all occurrences of c and d respecting the same orientations. Table 11b gives an example of link set.

2.3 Mathematically defining genomic regions

We aim to order oriented occurrences of the contigs based on their links. Each genome form corresponds to a sequence of oriented contigs. Not all contigs or their occurrences are included. Indeed, the contig set may contain contigs belonging to the plant genome or other organelles. The link set may also contain artefact links. Definition 2.1 provides the properties the sequence of oriented contigs must respect.

▶ Definition 2.1: Sequence of oriented contigs

> Let $SOC = (c_0, c_1, \ldots, c_{n-1})$ be a sequence of oriented contigs:
>
> — $\forall i \in [\![0, n-1[\![, (c_i, c_{i+1}) \in \mathcal{L}$;
>
> — $\forall c \in \mathcal{C}, \sum_{c_i \in SOC | c_i = c} 1 \leq mult(c)$.

Based on the biological knowledge, we address the dedicated chloroplast scaffolding problem as a region-driven scaffolding, such that specific regions fit into a circular structure. We identify three types of regions to scaffold: the Inverted Repeat (IR), the Direct Repeat (DR) and the Single-Copy (SC).

▶ Definition 2.2: Region

> A region $r = (c_0, c_1, \ldots, c_{n-1})$ is a sequence of oriented contigs. Each region is oriented. Let $\overline{r} = \left(\overline{c_{n-1}}, \ldots, \overline{c_1}, \overline{c_0}\right)$ be the reverse region of r. It is composed of the oriented contigs of r, considered in their reverse orientation, and given in the reversed order. According to the reverse symmetry in the links, $\overline{r}$ also respects Definition 2.1.

[1] In Chapter II, Section 1, $\mathcal{L}$ is a multiset while in this chapter, $\mathcal{L}$ is a set.

▶ Definition 2.3: Direct repeat (DR)

A DR is a couple of regions (dr_i, dr_j) where $dr_i = dr_j$.

▶ Definition 2.4: Inverted repeat (IR)

An IR is a couple of regions (ir_k, ir_l) where $ir_l = \overline{ir_k}$.

▶ Definition 2.5: Repeat

A repeat is the generic term to denote either DR or IR. The length $replen(R)$ of a repeat $R = (r_i, r_j)$ equals the lengths of r_i and r_j ($replen(R) = |r_i| + |r_j|$).

▶ Definition 2.6: Single-copy (SC)

A SC is a region that is not part of a repeat.

▶ Definition 2.7: Region weight

The weight $rwex(r)$ of a region r is defined as $rwex(r) = \sum_{c \in r} wex(c)$.

A chloroplast genome consists of a sequence of oriented regions. A genome form is a result of iterative transformations of an initial one. Section 8 introduces the region graph to model multiple genome forms (sequences of oriented contigs).

▶ Definition 2.8: Sequence of oriented regions

Consider a sequence $SOR = (r_0, r_1, \ldots, r_{n-1})$:

— $\forall\, i \in [\![0, n-1[\![, (r_i[|r_i| - 1]^a, r_{i+1}[0]) \in \mathcal{L}$;

— $\forall\, c \in \mathcal{C}, \sum_{r \in SOR} \sum_{c_i \in r | c_i = c} 1 \leq mult(c)$.

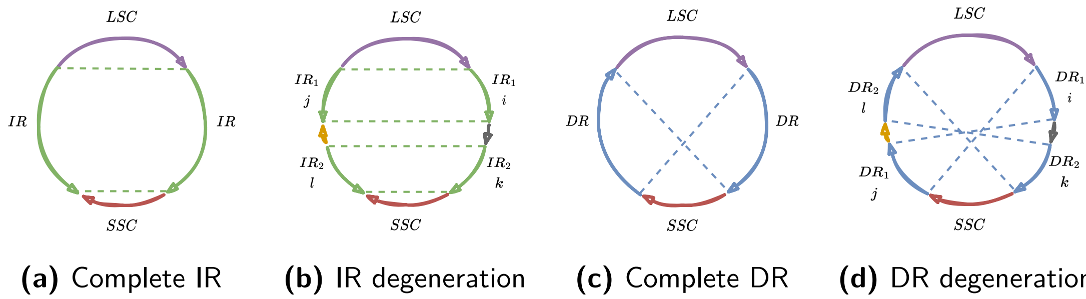
[a] The notation $x[i]$ denotes the ith element in x.

3 Chloroplast scaffolding problem formulations

Solving the repeats is the most challenging task. A formulation that does not restrict the occurrences can lead to misassemblies where the results are longer than

the solution genomes. Therefore, the use of an occurrence is limited to conformity with the biological knowledge of genome forms. A contig should participate in the sequence only if it enables the formation of pairs of repeated regions. In this case, we would be inclined to assemble the minimum number of repeats.

However, in the case of repeat degeneration (e.g. two subsequences inside the two regions of an identified repeat differ, note that some IR losses have been reported in the chloroplast genomes of green algae – Turmel et al. 2017) finding the minimum number of repeats is not an appropriate model. Figure 31 illustrates the impact of degenerations on quadripartite structures. Indeed, in Figure 31b, we cannot guaranty to find both IR_1 and IR_2, but perhaps only one of them. For each repeat type, we address this issue by maximising the cumulative repeat lengths only if their regions respect a specific order.



(a) Complete IR **(b)** IR degeneration **(c)** Complete DR **(d)** DR degeneration

Figure 31 – Repeat degeneration and region orders.
With time, a repeat may degrade so that its occurrences differ. **(a)** and **(c)** show two common quadripartite structures with an IR and a DR, respectively. **(b)** and **(d)** highlight the impact of a degeneration on their structures. We give the bellow region orders according to the LSC arrow's direction. In (b), the degeneration results in two IRs: $IR_1 = (i, j)$ and $IR_2 = (k, l)$, such that i is before k and l precedes j. In (d) it results in two DRs: $DR_1 = (i, j)$ and $IR_2 = (k, l)$, such that i is before k and j precedes l.

▸ **Definition 3.1: Chloroplast scaffolding problem $\mathcal{CHSP}$**

> Given a set of contigs with their multiplicities and their weights, a starting contig and a link set. The aim is to obtain a circular sequence of oriented regions maximising the cumulative repeat lengths and minimising the number of repeats, with single-copies of maximum-weight.

For instance, let Cases (A) and (B) be two distinct and feasible sequences of oriented contigs:

(A) $(\dots, a, b, c, d, \dots, a, b, c, d, \dots)$ has one DR (i, j), where $i = j = (a, b, c, d)$;

(B) $(\dots, a, b, \dots, c, d, \dots, a, b, \dots, c, d, \dots)$ has two DRs (k, l) and (m, n), where $k = l = (a, b)$ and $m = n = (c, d)$.

For Cases (A) and (B) the cumulative lengths are the same ($replen((i, j)) = replen((k, l)) + replen((m, n)) = 8$). However, Case (A) has one less repeat, which we prefer.

$\mathcal{CHSP}$ involves three subproblems, each one associated with a particular type of region: (i) $\mathcal{DRP}$ for the direct repeats (Definition 3.2); (ii) $\mathcal{IRP}$ for the inverted repeats (Definition 3.3) and (iii) $\mathcal{SCP}$ for the single-copies (Definition 3.4). We tackle $\mathcal{CHSP}$ in a hierarchical succession of $\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ (Definition 3.6). Any intermediate problem must preserve the regions found by its predecessors (Definition 3.5).

$\mathcal{DRP}$ and $\mathcal{IRP}$ constrain the number of occurrences to the structure of pairs of repetitions. Indeed, each repeat type defines a valid repeat structure. The problems consist in maximising the cumulative length of the minimum number of repeats.

▶ Definition 3.2: Chloroplast direct repeat scaffolding problem $\mathcal{DRP}$

Consider a set of contigs, their multiplicities, a starting contig and a link set. Find a circular sequence of oriented regions SOR, such that:

— it maximises the cumulative length of the minimum number of DRs, joined by regions of any kind;

— for any couple of DRs (i, j) and (k, l) found in SOR such that their respective positions in SOR given by function σ respect $\sigma(i) < \sigma(j)$, $\sigma(k) < \sigma(l)$, and $\sigma(i) < \sigma(k)$, then:

- $[\![\sigma(i), \sigma(j)]\!] \cap [\![\sigma(k), \sigma(l)]\!] = \varnothing$;
- or $[\![\sigma(k), \sigma(j)]\!] \subset [\![\sigma(i), \sigma(l)]\!]$.

▶ Definition 3.3: Chloroplast inverted repeat scaffolding problem $\mathcal{IRP}$

Consider a set of contigs, their multiplicities, a starting contig and a link set. Find a circular sequence of oriented regions SOR, such that:

— it maximises the cumulative length of the minimum number of IRs, joined by regions of any kind;

— for any couple of IRs (i, j) and (k, l) found in SOR such that that their respective positions in SOR given by function σ respect $\sigma(i) < \sigma(j)$, $\sigma(k) < \sigma(l)$ and $\sigma(i) < \sigma(k)$, then:

- $[\![\sigma(i), \sigma(j)]\!] \cap [\![\sigma(k), \sigma(l)]\!] = \varnothing$;
- or $[\![\sigma(k), \sigma(l)]\!] \subset [\![\sigma(i), \sigma(j)]\!]$.

Figure 32 provides examples of valid oriented contig positioning for each common genome structure (Figure 5). Although Figure 36 illustrates the authorised and forbidden positions for the latter defined repeated fragments, it is also applicable for the $\mathcal{DRP}$ and $\mathcal{IRP}$ regions' position cases.

▶ Definition 3.4: Chloroplast single-copy scaffolding problem $\mathcal{SCP}$

Consider a set of contigs, their multiplicities, their weights, a starting contig and a link set. Find a circular sequence of oriented regions such that all the single-copies maximise their weights.

Note that in Definition 3.4, if they are no repeats, the problem is reduced to find the maximum-weighted circuit of oriented contigs.

▶ Definition 3.5: Chloroplast scaffolding problem succession

$\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ (Definitions 3.2 to 3.4) can also take as input a set of fixed regions that must be preserved in the resulting sequence of oriented regions.

Our hierarchical approach prioritises the scaffolding of the repeats previously to the SC regions. Indeed, scaffolding the repeats is the most difficult task as it can lead to misassemblies (wrongly chosen links). Hence, a contig should only be used as many times as possible if its occurrences enable the scaffolding of repeats that most closely represent the architecture of the chloroplast genome, as illustrated in Figures 5 and 32. We assume that the weights concern events that are less relevant comparing to the genomes architecture. Also, the weights on the contigs are less relevant than if they were weights on the links.

▶ Definition 3.6: Hierarchical problem succession

The form of each solution of $\mathcal{CHSP}$ satisfies one of the two problem successions: $\mathcal{DRP}$–$\mathcal{IRP}$–$\mathcal{SCP}$ (h_1) and $\mathcal{IRP}$–$\mathcal{DRP}$–$\mathcal{SCP}$ (h_2).

The next question is how to prioritise $\mathcal{DRP}$ and $\mathcal{IRP}$? We propose resolving the order by comparing the scores defined in Section 5.5: if $\mathcal{DRP}$ score is better than this of $\mathcal{IRP}$, then the retained succession will be $\mathcal{DRP}$–$\mathcal{IRP}$–$\mathcal{SCP}$, otherwise it will be $\mathcal{IRP}$–$\mathcal{DRP}$–$\mathcal{SCP}$. In the equality case, we discriminate at a further step of the hierarchical successions. The process is detailed in Section 6.

Finally, each hierarchical problem succession produces a circular sequence of oriented regions. From the obtained sequence it is possible to extract a set of ordered pairs of oriented regions. This procedure allows the building of several circular sequences of oriented regions of the same length. Each of them represents one possible chloroplast genome form. This all-equivalent-form process is described in Section 8.

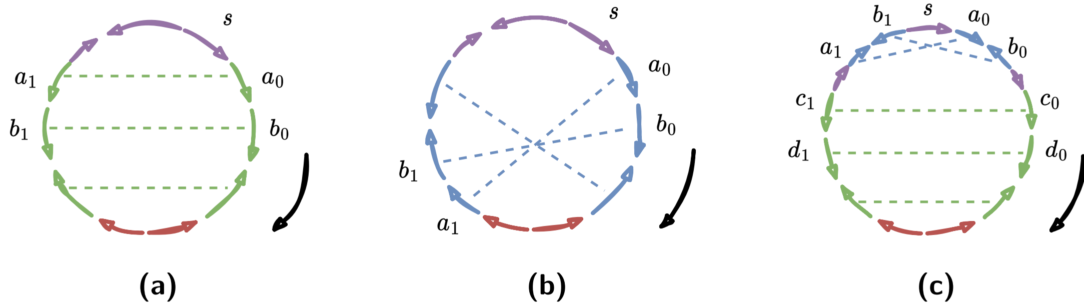


Figure 32 – Chloroplast repeat scaffolding.
Each subfigure is a common chloroplast genome structure with its associated order of oriented contigs (coloured arrows). The green and the blue sequences of arrows are IR and DR, respectively. The purple and the red ones are single-copy regions. Contig s is the starter, and the right side black arrow determines the contigs' order. Contigs a_0, a_1, b_0, b_1, c_0, c_1 and d_0, d_1 are two occurrences of four contigs a, b, c and d, respectively. Each coloured dashed line links two occurrences of the same contig. **(a)** The order of the occurrences is reversed, and their arrows are oppositely oriented. Visually, an IR produces parallel dashed lines. **(b)** The order and the orientation of the occurrences is preserved, revealing a DR. **(c)** A chloroplast genome can contain the two types of repeats. Here, we will retain the hierarchical problem succession $\mathcal{IRP}$–$\mathcal{DRP}$–$\mathcal{SCP}$ (h_2) since the IR contains more contigs than the DR.

4 Graph and repeated fragment sets

In order to efficiently handle the multiplicities of the contigs, hence of the links, we need to build adapted data structures. On the one hand, finding a sequence of oriented contigs, when the links correspond to ordered pairs of oriented contigs, justifies the use of a directed graph to represent the oriented contigs and their links. Section 4.1 defines such a directed graph structure. On the other hand, scaffolding the repeats requires choosing pairs of contigs occurring several times in the oriented contig sequence. Section 4.2 defines the sets of such repeat contig candidates.

4.1 Graph structure

Here we describe a directed graph suitable for further algorithms and the mathematical formulation of the scaffolding problems from Definitions 3.2 to 3.4.

▶ **Definition 4.1: Multiplied Doubled Contig Graph –** $MDCG$

Given a set of contigs $\mathcal{C}$, their multiplicities and the link set $\mathcal{L}$, the multiplied doubled contig graph $MDCG = (V, E, vwex)$ is defined such that[a]:

— $V = \{v_{f,0}, \ldots, v_{f,n-1}, v_{r,0}, \ldots, v_{r,n-1} \mid c \in \mathcal{C}, n = mult(c)\}$ is the set of all the forward an reverse occurrences of all the contigs ($|V| = 2\sum_{c\in\mathcal{C}} mult(c)$). The vertices are associated with four functions:

- $contig\colon V \twoheadrightarrow \mathcal{C}$ provides the contig associated with a vertex;
- $vor\colon V \twoheadrightarrow \{f, r\}$ provides the orientation of the contig;
- $vocc\colon V \rightarrow \mathbb{N}_{>0}$ provides the occurrence number of the contig;
- $vwex\colon V \rightarrow \mathbb{R}_{\geq 0}$ provides the weight of each vertex such that $\forall v \in V, vwex(v) = wex(contig(v))$.

— $E = \{(u, v) \in V^2 \mid ((contig(u), vor(u)), (contig(v), vor(v))) \in \mathcal{L}\}$ is the set of multiplied links ($|E| = \sum_{(c,d)\in\mathcal{L}} mult(c)mult(d)$).

[a] $MDCG$ is a digraph, not a multidigraph as DG.

Figure 33 illustrates the $MDCG$ representing the example data given in Table 11.

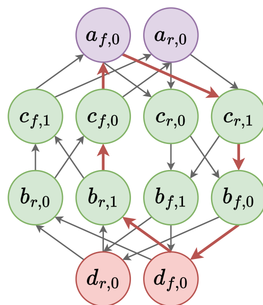


Figure 33 – $MDCG$ **example.**
Each vertex is associated with an occurrence of an oriented contig, and each contig is represented by an even number of vertices. For example, vertex labelled $c_{r,1}$ means that it comes from contig $c = contig(c_{r,1})$, in its reverse orientation ($vor(c_{r,1}) = r$), and in its second occurrence ($vocc(c_{r,1}) = 1$). The colours are the same as the ones in Figure 5a to relate the input data with the IR architecture. The bold red edges draw a circuit corresponding to an IR scaffolding where $a_{f,0}$ is the starter.

4.2 Repeated fragment sets

A repeat is a couple containing two identical (or reverse for IR) sequences of oriented contigs (Definitions 2.3 and 2.4). Therefore, a repeat consists of couples containing two identical (or reverse) contigs. In the context of $MDCG$, this leads to the concept of *repeated fragments*.

▶ Definition 4.2: Repeated fragment

A repeated fragment is an unordered pair of vertices such that one of the corresponding oriented contig belongs to the first region of a repeat, while the other belongs to the second region. The vertices are associated with the same contig but their occurrences differ, i.e. for each repeat (r_i, r_j), $\exists\, u, v \in V, c \in \mathcal{C}$ where $contig(u) = contig(v) = c$ and $vocc(u) \neq vocc(v)$ such that $(c, vor(u)) \in r_i$ and $(c, vor(v)) \in r_j$.

For example, $(c_{f,0}, c_{r,1})$ is a repeated fragment for the IR in Figure 33. We then precise the set of repeated fragments for each repeat type. Denote by $\mathcal{R} = \{c \in \mathcal{C} \mid mult(c) > 1\}$ the set of contigs candidate to be part of repeats. For the sake of clarity, for each vertex $v \in V$, we note $ctg_v = contig(v)$, $or_v = vor(v)$, $occ_v = vocc(v)$, $wex_v = vwex(v)$ and $mult_v = mult(contig(v))$. We also assume there is an arbitrary strict total order on $\mathcal{C}$, i.e. $\forall\, c, d \in \mathcal{C}, c \neq d \iff c < d \veebar c > d$.

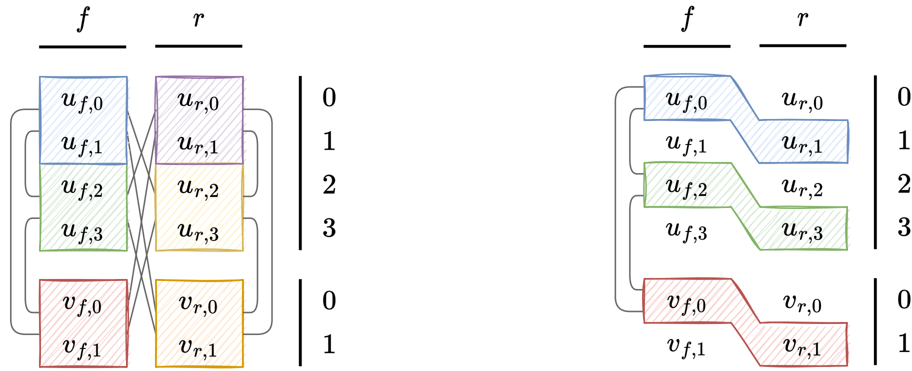
▶ Definition 4.3: Set of direct fragments $DirF$

A direct fragment $(u, v) \in V^2$ is a repeated fragment, such that u and v have the same orientation.

$$DirF = \bigcup_{c \in \mathcal{R}} \left\{ \begin{array}{l} (i, j) \in V^2 \text{ s.t.} \\ \quad ctg_i = ctg_j = c \\ \quad \wedge\, or_i = or_j \in \{f, r\} \\ \quad \wedge\, occ_i = occ_j - 1 = 2k \\ \qquad 0 \leq k < \left\lfloor \dfrac{mult(c)}{2} \right\rfloor \end{array} \right\}$$

▶ Definition 4.4: Set of inverted fragments $InvF$

An inverted fragment $(u, v) \in V^2$ is a repeated fragment, such that the orienta-



(a) Direct fragments and their pairs **(b)** Inverted fragments and their pairs

Figure 34 – Repeated fragment sets illustration for two contigs c and d. In the two subfigures, $mult(c) = 4$ and $mult(d) = 2$, so that $contig(u_{or,occ}) = c$ and $contig(v_{or,occ}) = d$. Two vertices coming from the same contig are respectively direct/inverted fragments if they are in the same coloured box, and so they belong to $DirF/InvF$. A tight grey line connects two direct/inverted fragments if their pair belong to $PDirF/PInvF$. **(a)** $|DirF| = 6$, and, e.g. $(u_{f,0}, u_{f,1}) \in DirF$ so $dirfrag(u_{f,0}) = dirfrag(u_{f,1}) = (u_{f,0}, u_{f,1})$. $|PDirF| = 12$, and, e.g. $((u_{f,0}, u_{f,1}), (u_{r,2}, u_{r,3})) \in PDirF$. **(b)** $|InvF| = 3$, and, e.g. $(u_{f,2}, u_{r,3}) \in InvF$ so $invfrag(u_{f,2}) = invfrag(u_{r,3}) = (u_{f,2}, u_{r,3})$. $|PInvF| = 3$, and, e.g. $((u_{f,2}, u_{r,3}), (v_{f,0}, v_{r,1})) \in PInvF$.

tions of u and v differ.

$$InvF = \bigcup_{c \in \mathcal{R}} \left\{ \begin{array}{l} (i,j) \in V^2 \text{ s.t.} \\ \quad ctg_i = ctg_j = c \\ \quad \wedge\, or_i = f \wedge or_j = r \\ \quad \wedge\, occ_i = occ_j - 1 = 2k \\ \qquad 0 \leq k < \left\lfloor \dfrac{mult(c)}{2} \right\rfloor \end{array} \right\}$$

Figure 34 illustrates $DirF$ and $InvF$ sets. In addition, we add two functions to retrieve the repeated fragments from the vertices in $\Theta(1)$: $dirfrag\colon V' \subset V \to DirF$ and $invfrag\colon V' \subset V \to InvF$ (abstracted with the $repfrag$ function, c.f. Appendix 1 for their definitions).

Furthermore, Definitions 3.2 and 3.3 constrain the order between the repeated fragments. Hence, they respectively require comparing pairs of direct/inverted

fragments, that must be defined:

▶ Definition 4.5: Set of pairs of direct fragments

$$PDirF = \left\{ \begin{array}{l} \big((i,j),(k,l)\big) \in DirF^2 \text{ s.t.} \\ \quad ctg_j < ctg_k \\ \quad \vee \\ \quad ctg_j = ctg_k \wedge occ_j < occ_k \end{array} \right\}$$

▶ Definition 4.6: Set of pairs of inverted fragments

$$PInvF = \left\{ \begin{array}{l} \big((i,j),(k,l)\big) \in InvF^2 \text{ s.t.} \\ \quad ctg_j < ctg_k \\ \quad \vee \\ \quad ctg_j = ctg_k \wedge occ_j < occ_k \end{array} \right\}$$

Figure 34 illustrates $PDirF$ and $PInvF$ sets. The constraints defining $DirF$, $InvF$, $ADirF$, $AInvF$, $PDirF$ and $PInvF$ are explained in details in Appendix 2 where we proof they are the smallest sets enabling to find all the distinct solutions.

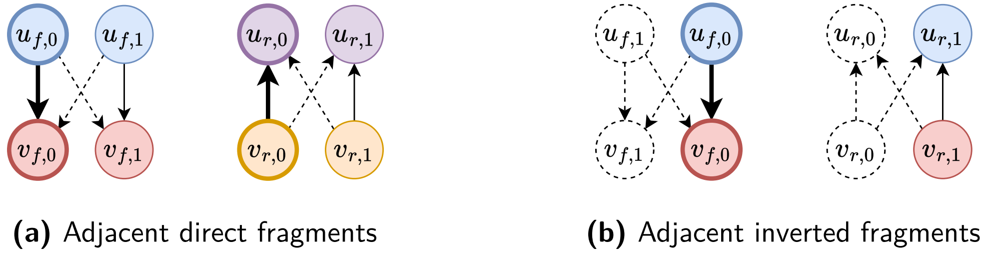
Furthermore, a repeat is a couple of regions (Definition 2.5), themselves defined as oriented contig sequences (Definition 2.2). We need to define the edges connecting two repeated fragments.

▶ Definition 4.7: Set of adjacent repeated fragments

An adjacent repeated fragment is an edge $(u, v) \in E$ such that u and v participate in two distinct repeated fragments.

▶ Definition 4.8: Set of adjacent direct fragments

An adjacent direct fragment is an edge between two direct fragments. Let



(a) Adjacent direct fragments **(b)** Adjacent inverted fragments

Figure 35 – Adjacent repeated fragment sets examples.
The two subfigures represent the multiplied link (and its reverse) $((c, f), (d, f)) \in \mathcal{L}$, where $mult(c) = 2$ and $mult(d) = 2$, so that $contig(u_{or,occ}) = c$ and $contig(v_{or,occ}) = d$. Two vertices of the same colour visualise a repeated fragment. Bold edges (canonical) are the ones that belong to the adjacent repeated fragments sets. The functions $diradj/invadj$ enable to retrieve the normal edges with the bold ones, and vice-versa. Dashed edges do not participate in $ADirF/AInvF$. Remember that $\forall (u, v) \in E, (\overline{v}, \overline{u}) \in E$. **(a)** $diradj\,(u_{f,0}, v_{f,0}) = (u_{f,1}, v_{f,1})$; **(b)** $invadj\,(u_{f,0}, v_{f,0}) = (v_{r,1}, u_{r,1})$.

$ADirF$ be the set of adjacent direct fragments:

$$ADirF = \left\{ \begin{aligned} &(u, v) \in E \text{ s.t.} \\ &\quad ctg_u \neq ctg_v \\ &\quad \wedge occ_u = 2k \\ &\qquad 0 \leq k < \left\lfloor \frac{mult_u}{2} \right\rfloor \\ &\quad \wedge occ_v = 2k' \\ &\qquad 0 \leq k' < \left\lfloor \frac{mult_v}{2} \right\rfloor \end{aligned} \right\} \bigcup \left\{ \begin{aligned} &(u, v) \in E \text{ s.t.} \\ &\quad ctg_u = ctg_v \\ &\quad \wedge (or_u = f \vee or_v = f) \\ &\quad \wedge occ_u = 2k \\ &\quad \wedge occ_v = 2k' \\ &\qquad 0 \leq k < k' < \left\lfloor \frac{mult_u}{2} \right\rfloor \end{aligned} \right\}$$

▶ **Definition 4.9: Set of adjacent inverted fragments**

An adjacent inverted fragment is an edge between two inverted fragments. Let

$AInvF$ be the set of adjacent inverted fragments:

$$AInvF = \left\{ \begin{aligned} &(u,v) \in E \text{ s.t.} \\ &\quad ctg_u < ctg_v \\ &\quad \wedge\, occ_u = 2k + or_u \\ &\qquad 0 \leq k < \left\lfloor \frac{mult_u}{2} \right\rfloor \\ &\quad \wedge\, occ_v = 2k' + or_v \\ &\qquad 0 \leq k' < \left\lfloor \frac{mult_v}{2} \right\rfloor \end{aligned} \right\} \bigcup \left\{ \begin{aligned} &(u,v) \in E \text{ s.t.} \\ &\quad ctg_u = ctg_v \\ &\quad \wedge\, (or_u = f \vee or_v = f) \\ &\quad \wedge\, occ_u - or_u = 2k \\ &\quad \wedge\, occ_v - or_v = 2k' \\ &\qquad 0 \leq k < k' < \left\lfloor \frac{mult_u}{2} \right\rfloor \end{aligned} \right\}$$

Edges in $ADirF$ and $AInvF$ play the role of *canonical edges* between two adjacent repeated fragments, see Figure 35. In addition, we add two functions to retrieve the adjacent repeated fragments from the edges in $\Theta(1)$: $diradj\colon E \to E$ and $invadj\colon E \to E$ (abstracted with the *repadj* function, c.f. Appendix 1 for their definitions).

5 Integer Linear Programming (ILP) formulation

Modelling $\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ from Definitions 3.2 to 3.4 requires finding a valid circuit in $MDCG$.

▶ Definition 5.1: Valid circuit in $MDCG$

Given a graph $MDCG = (V, E)$ and a starting vertex s, where ctg_s is the starting contig, $or_s = f$ and $occ_s = 0$. A circuit cp in $MDCG$ is valid if:

— it starts and ends with s;

— $\forall\ v \in cp, \overline{v} \notin cp$, where $ctg_v = ctg_{\overline{v}}$, $or_v = 1 - or_{\overline{v}}$ and $occ_v = occ_{\overline{v}}$;

— consecutive vertices u and v in cp are connected by an edge $(u, v) \in E$.

First we describe common constraint blocks in Sections 5.1 to 5.3 for ILPs formulations, and then we give the $\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ scaffolding problems ILP in Section 5.5.

Let $\mathsf{M} = \sum_{c \in \mathcal{C}} mult(c)$ be a constant, N_v^- and N_v^+ be the sets of predecessors and successors of vertex $v \in V$, respectively.

5.1 Circuit constraints

The following set of constraints defines a valid circuit of oriented contig in $MDCG$, and is defined with a flow formulation as in Andonov et al. (2019) instead of using Miller-Tucker-Zemlin constraints to avoid cycles (Miller et al., 1960).

Binary variables

— x_e encodes if the edge $e \in E$ participates in the circuit.

Continuous variables

— $i_v \in [0,1]$ encodes if the vertex $v \in V \setminus \{s, \overline{s}\}$ participates in the circuit. Although it is a continuous variable, it acts as a binary one as proven in (François et al., 2018).

— $f_e \in \mathbb{R}_{\geq 0}$ is the positive flow on the participating edge $e \in E$ in the circuit (zero otherwise).

Constraint C1 defines the flow. The circuit must start and end with the starter in its forward orientation (Constraints C2 to C5). If a vertex participates, its reverse cannot (Constraint C6). Defining a circuit is equivalent to requiring an edge to exit a vertex if it has an incoming one (Constraint C7). Constraint C8 forces the flow to be monotonically increasing. This property avoids cycles.

$CCircuit$ constraints

$$x_e \leq f_e \leq \mathsf{M} x_e \qquad \forall\, e \in E \tag{C1}$$

$$\sum_{v \in N_s^+} x_{sv} = \sum_{v \in N_s^-} x_{vs} = 1 \tag{C2}$$

$$\sum_{v \in N_s^+} f_{sv} = 1 \tag{C3}$$

$$x_{v\overline{s}} = 0 \qquad \forall\, v \in N_{\overline{s}}^- \tag{C4}$$

$$x_{\overline{s}v} = 0 \qquad \forall\, v \in N_{\overline{s}}^+ \tag{C5}$$

$$\forall v \in V \setminus \{s, \overline{s}\} :$$

$$i_v + i_{\overline{v}} \leq 1 \tag{C6}$$

$$\sum_{u \in N_v^-} x_{uv} \leq i_v \leq \sum_{w \in N_v^+} x_{vw} \tag{C7}$$

$$\sum_{w \in N_v^+} f_{vw} - \sum_{u \in N_v^-} f_{uv} = i_v \tag{C8}$$

$$x_e \in \{0,1\} \quad \forall e \in E$$

$$i_v \in [0,1] \quad \forall v \in V \setminus \{s, \overline{s}\}$$

$$f_e \in \mathbb{R}_{\geq 0} \quad \forall e \in E$$

5.2 Repeated regions constraints

The following constraints are general to define ILPs for $\mathcal{DRP}$ and $\mathcal{IRP}$. Definitions 3.2 and 3.3 define the repeated regions according to the positions of the oriented contig in them. It follows that some order of the vertices in the pairs of repeated fragments are allowed, and some others are forbidden. We decide to write the constraints for the forbidden cases because they are fewer than the allowed ones. To model the forbidden orders between 4 vertices, we compare the positions between two.

Specifically for $\mathcal{IRP}$, modelling the forbidden orders echoes the approach for the RNA folding problem (Gusfield, 2019), except that the positions of the RNA's nucleotides are known.

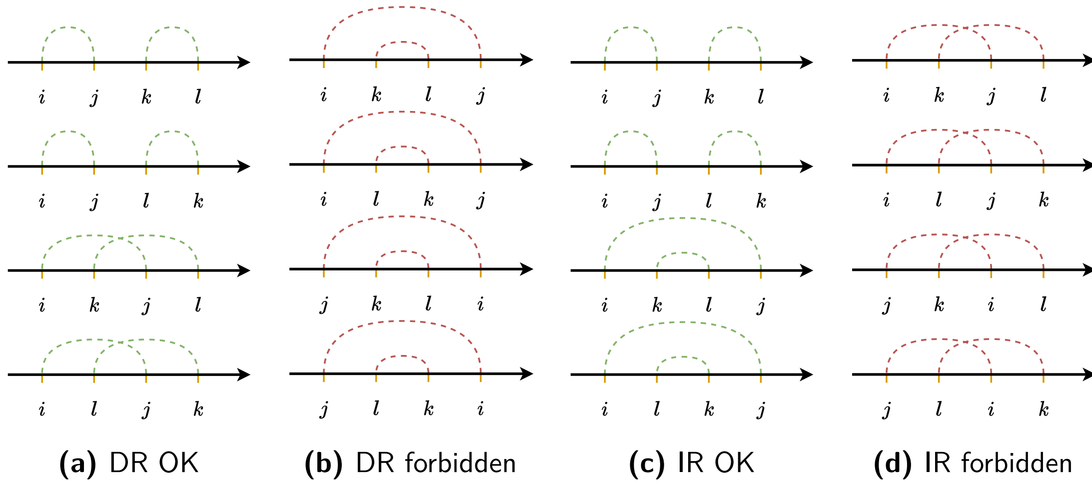
According to Definitions 3.2 and 3.3, and given $PDirF$ and $PInvF$ (Definitions 4.5 and 4.6), denote by $ForbidDR$ and $ForbidIR$ the sets of forbidden quartet vertices for the DRs and IRs, respectively:

$$ForbidDR = \left\{ \begin{matrix} (i,k,l,j) \\ (k,i,j,l) \end{matrix} \middle| \begin{matrix} i,j \in p, i \neq j \wedge k,l \in q, k \neq l \\ \forall (p,q) \in PDirF \end{matrix} \right\}$$

$$ForbidIR = \left\{ \begin{matrix} (i,k,j,l) \\ (k,i,l,j) \end{matrix} \middle| \begin{matrix} i,j \in p, i \neq j \wedge k,l \in q, k \neq l \\ \forall (p,q) \in PInvF \end{matrix} \right\}$$

Figure 36 illustrates the authorised and forbidden positions for $\mathcal{DRP}$ and $\mathcal{IRP}$.

To know if we are in the forbidden cases described in these two sets, we propose to compare the vertices two-by-two. Denote by $AlphaDR$ and $AlphaIR$ the sets containing the couples of vertices to be compared to determine the forbidden cases



(a) DR OK **(b)** DR forbidden **(c)** IR OK **(d)** IR forbidden

Figure 36 – Non-exhaustive illustrations for authorised and forbidden order cases for two repeated fragments $((i,j),(k,l)) \in PRepF$**.**
(a) and **(b)** Authorised and forbidden orders for $PDirF$; **(c)** and **(d)** Authorised and forbidden orders for $PInvF$.

respectively associated with $ForbidDR$ and $ForbidIR$ sets, such that:

$$AlphaDR = \left\{ \begin{array}{l} (i,j),(k,l),(i,k), \\ (j,l),(i,l),(j,k) \\ \text{s.t. } ((i,j),(k,l)) \in PDirF \end{array} \right\}$$

$$AlphaIR = \left\{ \begin{array}{l} (i,k),(i,l),(j,k),(j,l) \\ \text{s.t. } ((i,j),(k,l)) \in PInvF \end{array} \right\}$$

In the following, the sets of repeated fragments and these for the forbidden orders are abstracted to generalise $\mathcal{DRP}$ and $\mathcal{IRP}$. Table 12 gives the correspondence of the sets depending on the problem to solve.

Table 12 – ILP sets and functions corresponding table.

ILP	RepF	PRepF	ARepF	Forbid	Alpha	repfrag	repadj
$\mathcal{DRP}$	$DirF$	$PDirF$	$ADirF$	$ForbidDR$	$AlphaDR$	$dirfrag$	$diradj$
$\mathcal{IRP}$	$InvF$	$PInvF$	$AInvF$	$ForbidIR$	$AlphaIR$	$invfrag$	$invadj$

Binary variables

— m_{ij} encodes if the repeated fragment $(i,j) \in RepF$ is a part of a repeat.

— $isadj_e$ encodes if two repeated fragments connected by the edge $e \in ARepF$ (and $repadj(e) \in E$) are adjacent in the circuit.

— $forbid_{ijkl}$ encodes whether we are in the forbidden vertices order $(i, j, k, l) \in Forbid$.

— α_{uv} encodes whether the vertex u is before the vertex v in the circuit. Since $\alpha_{uv} = 1 - \alpha_{vu}$, even if $(v, u) \notin Alpha$, for clarity we write α_{vu} instead of $1 - \alpha_{uv}$.

Continuous variables

— $i_v \in [0, 1]$ encodes if the vertex $v \in V \setminus \{s, \overline{s}\}$ participates in the circuit, and acts as a binary variable.

— $f_e \in \mathbb{R}_{\geq 0}$ is the positive flow on the participating edge $e \in E$ in the circuit (zero otherwise). We use the exiting flow to define the position $pos(v)$ of a vertex $v \in V$, i.e. $pos(v) = \sum_{w \in N_v^+} f_w$.

The vertices of participating repeated fragments must be in the circuit (Constraints C9 and C10). Constraints C11 to C13 implement with linear constraints the α_{uv} definition. Constraints C14 to C16 implement the $forbid_{ijkl}$ definition. Constraints C17 to C20 implement the $isadj_e$ definition.

$CRepeat$ constraints

Add the set of constraints $CCircuit$

$\forall (i, j) \in RepF$:

$$m_{ij} \leq i_i \tag{C9}$$

$$m_{ij} \leq i_j \tag{C10}$$

$\forall (u, v) \in Alpha$:

$$pos(v) - pos(u) \leq \mathsf{M}\alpha_{uv} \tag{C11}$$

$$pos(u) - pos(v) \leq \mathsf{M}(1 - \alpha_{uv}) \tag{C12}$$

$$pos(u) + pos(v) \geq \alpha_{uv} \tag{C13}$$

$$\forall (i,j,k,l) \in Forbid:$$
$$3forbid_{ijkl} \leq \alpha_{ij} + \alpha_{jk} + \alpha_{kl} \tag{C14}$$
$$2 + forbid_{ijkl} \geq \alpha_{ij} + \alpha_{jk} + \alpha_{kl} \tag{C15}$$

$$\forall (p,q) \in PRepF:$$
$$m_p + m_q \leq 2 - \sum_{\substack{(i,j,k,l) \\ \in Forbid \\ \text{s.t. } (p,q)}} forbid_{ijkl} \tag{C16}$$

$$\forall (u,v) \in ARepF:$$
$$isadj_{uv} \leq x_{uv} \tag{C17}$$
$$isadj_{uv} \leq x_{repadj(u,v)} \tag{C18}$$
$$isadj_{uv} \leq m_{repfrag(u)} \tag{C19}$$
$$isadj_{uv} \leq m_{repfrag(v)} \tag{C20}$$

$$m_p \in \{0,1\} \quad \forall p \in RepF$$
$$isadj_e \in \{0,1\} \quad \forall e \in ARepF$$
$$forbid_{ijkl} \in \{0,1\} \quad \forall (i,j,k,l) \in Forbid$$
$$\alpha_{uv} \in \{0,1\} \quad \forall (u,v) \in Alpha$$

5.3 Fixing regions constraints

When repeats are previously scaffolded, the involved regions are fixed as input for the next problems. Let $ADirF^*$, $DirF^*$, $AInvF^*$ and $InvF^*$ respectively be the sets of (adjacent) direct and (adjacent) inverted fragments composing the direct and inverted repeats that have been scaffolded.

$CFixRegions$ constraints

$$\forall (u,v) \in DirF^* \cup InvF^*:$$
$$i_u = 1 \tag{C21}$$
$$i_v = 1 \tag{C22}$$

$$\forall (u,v) \in ADirF^* \cup AInvF^*:$$
$$x_{uv} = 1 \tag{C23}$$

$$x_{diradj(u,v)} = 1 \quad \forall (u,v) \in ADirF^* \tag{C24}$$
$$x_{invadj(u,v)} = 1 \quad \forall (u,v) \in AInvF^* \tag{C25}$$

5.4 Speed-up constraints

Constraints C26 and C27 prevent the solver to loop on strictly equivalent solutions, e.g. solutions that differ according to a permutation of the occurrences. Denote by $ConsOcc$ the set of occurrence-consecutive vertices, such that:

$$ConsOcc = \left\{ (u,v) \in V^2 \;\middle|\; \begin{array}{l} ctg_u = ctg_v \\ \wedge\, or_u = or_v = f \\ \wedge\, occ_u = occ_v - 1 \end{array} \right\}$$

Also, denote by $ConsRepF$ the set of consecutive repeated fragments, such that:

$$ConsRepF = \left\{ \begin{array}{l} ((i,j),(k,l)) \in RepF^2 \text{ s.t.} \\ \quad ctg_i = ctg_j = ctg_k = ctg_l \\ \quad \wedge\, or_i = or_k \wedge or_j = or_l \\ \quad \wedge\, occ_i = occ_k - 2 \\ \quad \wedge\, occ_j = occ_l - 2 \end{array} \right\}$$

$$i_v + i_{\overline{v}} \leq i_u + i_{\overline{u}} \qquad \forall (u,v) \in ConsOcc \tag{C26}$$
$$m_q \leq m_p \qquad \forall (p,q) \in ConsRepF \tag{C27}$$

5.5 Scaffolding problems ILP

Finally, it is possible to define the ILP formulations for the $\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ scaffolding problems as a union of the constraints described before.

For $\mathcal{DRP}$ and $\mathcal{IRP}$ the ILP formulations are the same, and it is sufficient to choose the sets $RepF$, $PRepF$, $ARepF$, $Alpha$, $Forbid$ and $ConsRepF$ according to the repeats the problems scaffold. We aim to maximise the cumulative length of the minimum number of repeats. The objective value corresponds to:

$$\begin{aligned} &\sum_{r \in Repeats} replen(r) - |Repeats| \\ &= \sum_{p \in RepF} 2m_p - \left(\sum_{p \in RepF} m_p - \sum_{e \in ARepF} isadj_e \right) \\ &= \sum_{p \in RepF} m_p + \sum_{e \in ARepF} isadj_e \end{aligned}$$

where $Repeats$ is the set of repeats.

$\mathcal{DRP}$ and $\mathcal{IRP}$ models

$$
\begin{aligned}
\text{max} \quad & \sum_{p \in RepF} m_p + \sum_{e \in ARepF} isadj_e \\
\text{s.t.} \quad & CCircuit \\
& CRepeat \\
& (\text{C26}) \\
& (\text{C27}) && \text{if no repeats to fix} \\
& CFixRegions && \text{otherwise}
\end{aligned}
$$

Traditionally, $\mathcal{SCP}$ finds the maximum weighted circuit.

$\mathcal{SCP}$ model

$$
\begin{aligned}
\text{max} \quad & \sum_{v \in V \setminus \{s, \bar{s}\}} \text{wex}_\text{v} i_v \\
\text{s.t.} \quad & CCircuit \\
& (\text{C26}) \\
& CFixRegions && \text{if repeats to fix}
\end{aligned}
$$

Both for $\mathcal{DRP}$ and $\mathcal{IRP}$, the number of variables and constraints are in $O(|V|^2 + |E|)$. The number of variables and constraints for $\mathcal{SCP}$ are in $O(|V| + |E|)$.

6 Hierarchical problem succession

Finally, here we describe how we combine the $\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ scaffolding problems. As described in Definition 3.6, two problem combinations are opposed. The combinations are kept depending on the value of the problems' objective functions.

▶ Definition 6.1: Hierarchical problem succession solutions

Denote by h_1, h_2 the two hierarchical problem successions $\mathcal{DRP}$–$\mathcal{IRP}$–$\mathcal{SCP}$ and $\mathcal{IRP}$–$\mathcal{DRP}$–$\mathcal{SCP}$. For each $h \in \{h_1, h_2\}$, denote by $Z^*_h \in \mathbb{R}_{\geq 0}{}^3$ the vector containing the values of the objective functions for each problem in the order of the problem succession corresponding to h.

By S we denote the set of optimal problem successions, such that:

$$S = \left\{ s \,\middle|\, \forall\, k \in [\![0, 2]\!], Z^*_s[k] = \max_{h \in \{h_1, h_2\}} Z^*_h[k] \right\}$$

Note that $0 \leq |S| \leq 2$, and Definition 6.1 is stable for any problem with an objective value equal to zero. For example, $Z^*_{h_1}[1] = 0$ means that there is no inverted repeat. For an easier interpretation of the architecture, we adopt a *problem code combination*, summarised in Table 13.

Table 13 – Problem code combinations

Z^*_h			h_1	h_2
[0]	[1]	[2]	$\mathcal{DRP}$–$\mathcal{IRP}$–$\mathcal{SCP}$	$\mathcal{IRP}$–$\mathcal{DRP}$–$\mathcal{SCP}$
0	0	–	sc	sc
> 0	0	–	dr–sc	ir–sc
> 0	> 0	–	dr–ir–sc	ir–dr–sc

7 From an ILP solution to a genome structure

From a solution found by the best hierarchical problem succession, we extract the corresponding genome architecture. Let $m, n \in \mathbb{N}$ be the number of repeats (pair of regions) and the number of single-copies, respectively. A genome contains $2m + n$ regions. Two items are sufficient to describe a genome with its regions:

— $COR = (r_0, r_1, \ldots, r_{m+n-1})$ contains the forward regions, i.e. it is a $(m+n)$-uplet of $m + n$ oriented contig sequences.

— $SOR = ((rid_0, ror_0), \ldots, (rid_{2m+n-1}, ror_{2m+n-1})) \in (\mathbb{N} \times \{f, r\})^{2m+n}$ is a linearised circular sequence of oriented regions. For each $i \in [\![0, 2m + n[\![$, if $ror_i = f$ then $SOR[i]$ represents the forward region $COR[rid_i]$, else if $ror_i = r$ then $SOR[i]$ represents the reverse $\overline{COR[rid_i]}$.

Figure 37 illustrates the regions extracted by Algorithm 25 in the toy example.

The key idea of such an extraction algorithm is to start from the starting vertex s and walk over the chosen edges. During the walk, for each vertex we need to identify the type of its region, and then add the vertex to the corresponding region (to the corresponding region identifier rid_r). Algorithm 22 aims to determine the

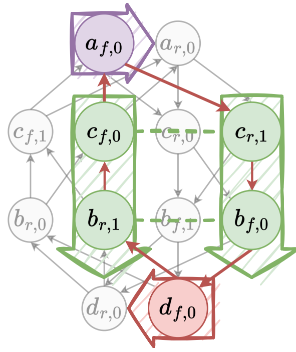


Figure 37 – Extracting the genome architecture in $\mathcal{MDCG}$ from a $\mathcal{CHSP}$ solution.
The illustrated $\mathcal{CHSP}$ solution consists of the red circuit $(a_{f,0}, c_{r,1}, b_{f,0}, d_{f,0}, b_{r,1}, c_{f,0})$ and of the two choosen inverted fragments $InvF^* = \{(c_{f,0}, c_{r,1}), (b_{f,0}, b_{r,1})\}$ (vertex pairs linked by the green dashed lines). Algorithm 25 returns four regions: $m = 1$ inverted repeat (the two green arrows pointing downwards) and $n = 2$ single-copies (the purple and the red arrows). $COR = ((a_f), (c_r, b_f), (d_f))$ and $SOR = (0_f, 1_f, 2_f, 1_r)$, where $x_y = (x, y)$ for clarity.

region type for a given vertex. The sequence of oriented regions SOR begins by the starter's region that is necessarily a single-copy (because $mult(contig(s)) = 1$). The first oriented contig may not be the starter. Algorithm 23 gives the *initial vertex* associated to the first oriented contig of the starter's region. During the walk in the solution circuit from the initial vertex, a new region begins each time the region type changes. When the current vertex participates in a repeat, we must check if the next one participates in the same repeat, although the region type may not change (Algorithm 24). At the end, Algorithm 25 builds COR and SOR from an ILP solution.

To build the repeated regions, we use a First In First Out (FIFO) for the DRs, and a Last In First Out (LIFO) for the IRs. Each queue *rep_queue*, is associated with three methods:

rep_queue.PUT(x) append x to the FIFO/LIFO;

rep_queue.IS_EMPTY returns true if the queue is empty;

rep_queue.PEEK returns the first/last value in the FIFO/LIFO;

rep_queue.POP deletes the first/last value in the FIFO/LIFO and returns it.

In the following, the given time complexities are under the assumption that the belonging test "*is* $x \in X$*?*" for an object x in a set X is in $\Theta(1)$. Algorithm 22 is in $\Theta(1)$. Algorithm 23 is in $O(|SC_s|)$, where $|SC_s|$ is the number of contigs composing the single-copy region that contains the starting vertex. Algorithm 24 is in $\Theta(1)$, and so Algorithm 25 is in $O(|V| + |E|)$.

▶ Algorithm 22: Get the region type for a given vertex

Require: A vertex v, the chosen direct fragments $DirF^*$ and the chosen inverted fragments $InvF^*$.
Ensure: Returns the region type of the vertex.

1: **function** GET_REGION_CODE(v)
2:     **if** $dirfrag\,(v) \in DirF^*$ **then**
3:         **return** DR
4:     **if** $invfrag\,(v) \in InvF^*$ **then**
5:         **return** IR
6:     **return** SC

▶ Algorithm 23: Get the initial vertex of the single-copy region containing the starting vertex

Require: The starting vertex s, the set of fixed variable values $\mathsf{x}^*_\mathsf{e}, e \in E$, the chosen direct fragments $DirF^*$ and the chosen inverted fragments $InvF^*$.
Ensure: Returns the first vertex of the single-copy region containing the starting vertex.

1: **function** INITIAL_VERTEX( )
2:     $v \leftarrow$ s
3:     $u \leftarrow u \mid \mathsf{x}^*_{\mathsf{us}} = 1$
4:     $reg_code \leftarrow$ GET_REGION_CODE(u)
5:     **while** $u \neq s \wedge reg_code =$ SC **do**
6:         $v \leftarrow u$
7:         $u \leftarrow u \mid \mathsf{x}^*_{\mathsf{uv}} = 1$
8:         $reg_code \leftarrow$ GET_REGION_CODE(u)
9:     **if** $region_code =$ SC **then** ▷ *Special case of a unique single-copy region*
10:         **return** s
11:     **return** v

▶ Algorithm 24: Is the repeat contiguous?

Require: Two vertices $u, v \in V$, the set of fixed variable values $\mathsf{x}^*_\mathsf{e}, e \in E$, the previous and current region codes $prev_region_code$ and $region_code$.
Ensure: Returns true if the repeat is contiguous, else false.

1: **function** REPEAT_IS_CONTIGUOUS(u, v, $prev_region_code$, $region_code$)
2:     **if** $prev_region_code \neq region_code$ **then**
3:         **return** False

4: **if** $region_code$ = DR **then**
5: **return** $x^*_{diradj(u,v)} = 1$
6: **if** $region_code$ = IR **then**
7: **return** $x^*_{invadj(u,v)} = 1$
8: **return** False

▶ **Algorithm 25: ILP solution to regions**

Require: Graph $MDCG$, the starting vertex s, the set of fixed variable values $x^*_e, e \in E$, the chosen direct fragments $DirF^*$ and the chosen inverted fragments $InvF^*$.
Ensure: Returns the oriented region sequence SOR, and the oriented contig sequence for each forward oriented region COR.
1: **function** ILP__SOLUTION__TO__REGIONS( )
2: $prev_region_code \leftarrow$ SC
3: $SOR \leftarrow [0_f]$ ▷ *Oriented region sequence, initialised by the single-copy in forward orientation that contains the starting vertex*
4: $COR \leftarrow [\,[\,]\,]$ ▷ *Oriented contig sequence for each forward region, initialised for the first region*
5: $region_index \leftarrow 0$ ▷ *First region index*
6: init_v ← INITIAL__VERTEX()
7: $u \leftarrow$ init_v; $v \leftarrow$ init_v
8: $rep_queue \leftarrow$ HASHTABLE() ▷ *For each repeat index is associated a queue of vertices (FIFO for DR, LIFO for IR)*
9: $repf_rep \leftarrow$ HASHTABLE() ▷ *Each repeated fragment is associated with its repeat index*
10: **repeat**
11: $region_code \leftarrow$ GET__REGION__CODE(v)
12: **if** $region_code$ = SC **then**
13: **if** $prev_region_code \neq region_code$ **then** ▷ *New single-copy*
14: $region_index \leftarrow |COR|$
15: SOR.APPEND($region_index_f$)
16: COR.APPEND([ ])
17: $COR[region_index]$.APPEND(ctg_v, or_v)
18: **else**
19: $repf \leftarrow dirfrag(v)$ if $region_code$ = DR else $invfrag(v)$
20: **if** $repf \notin repf_rep$ **then** ▷ *Fist region of the repeat*
21: **if** ¬REPEAT__IS__CONTIGUOUS($u, v, prev_region_code$, $region_code$) **then**

```
22:                 region_index ← |COR|
23:                 SOR.APPEND(region_index_f)
24:                 COR.APPEND([])
25:                 rep_queue[region_index] ← FIFO() if region_code = DR
                    else LIFO()
26:             COR[region_index].APPEND(ctg_v, or_v)
27:             rep_queue.PUT(dirfrag(v) if region_code =
                DR else invfrag(v))
28:             repf_rep[repf] ← region_index
29:         else                                ▷ Second region of the repeat
30:             if ¬REPEAT_IS_CONTIGUOUS (u, v, prev_region_code,
                region_code) then
31:                 region_index ← repf_rep[repf]
32:                 SOR.APPEND(region_index_f if region_code =
                    DR else region_index_r)
33:             assert v = rep_queue[region_index].POP( )
34:     prev_region_code ← region_code
35:     u ← v
36:     v ← v | x*_uv = 1
37: until v ≠ init_v
38: return SOR, COR
```

8 Multiple genome forms

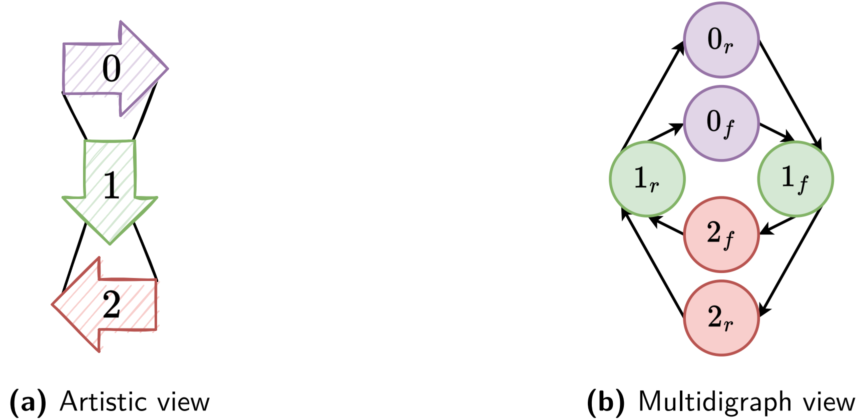
The sequence of oriented regions SOR represents one chloroplast genome form. Recall that especially with the LSC-IR-SSC-IR architecture (Figure 5a), the SSC can be reversed during the DNA replication phase. In the following, our goal is to retrieve other forms from the one obtained by the hierarchical problem succession.

Towards this goal, we introduce a specific assembly graph: the *region graph*. The discovery of multiple genome forms is associated with the search of Eulerian circuits in this graph. Figure 38 illustrates the procedure for the toy example.

▶ Definition 8.1: Region graph $RegGraph$

Given the $(m+n)$-uplet COR of forward regions and the sequence SOR of oriented regions, $RegGraph = (Vreg, Ereg, \Phi)$ denotes a multidigraph named the region graph, such that:

— $Vreg = \{v_{0,f}, \ldots, v_{m+n-1,f}, v_{0,r}, \ldots, v_{m+n-1,r} \mid |COR| = m+n\}$ is the



(a) Artistic view **(b)** Multidigraph view

Figure 38 – Region graph for the toy example.
The toy example's solution given in Figure 37 results in the graph visualised in Figure 33 will give a region graph that shows an LSC-IR-SSC-IR architecture as the one given in Figure 5a. Two oriented region sequences (genome forms) are obtained by finding the Eulerian circuits: $0_f \to 1_f \to 2_f \to 1_r$ and $0_f \to 1_f \to 2_r \to 1_r$. They correspond to two structural haplotypes commonly found in the chloroplast cells, and described in Figure 6. **(a)** Each arrow represents a region. Each link connects two arrows' extremity. Entering the tail/head and exiting the head/tail of an arrow corresponds to choosing the region in its forward/reverse orientation. **(b)** The same information is visualised. Each vertex is a region with a fixed orientation, and each edge connects two oriented regions.

set of oriented regions ($|Vreg| = 2|COR|$). For each vertex, bijective function $vreg\colon Vreg \hookrightarrow\!\!\!\rightarrow \{r \in COR\} \times \{f, r\}$ provides the oriented region it represents.

— $Ereg$ is the multiset of links between two oriented regions in SOR (including between the last and the first regions). Note that $\forall\, e \in Ereg$, $\overline{e} \in Ereg$ denotes its reverse where $\Phi(e) = (u, v)$, $\Phi(\overline{e}) = (\overline{v}, \overline{u})$ ($|Ereg| = 2|SOR|$).

— $\Phi\colon Ereg \to \{(u, v) \mid u, v \in Vreg\}$ is the incident function, such that for two consecutive oriented regions r_i and r_j in SOR (including the last and the first ones), $\exists!\, e \in Ereg \mid \Phi(e) = (vreg^{-1}(r_i), vreg^{-1}(r_j))$.

Figure 38 illustrates the region graph for the toy example's solution given in Figure 37. Based on the above graph, we can find different sequences of oriented regions. Each region, independently of its orientation, in each sequence, must participate the same number of times.

▶ **Definition 8.2: Eulerian circuit in** $RegGraph$

A circuit in $RegGraph = (Vreg, Ereg, \Phi)$ is defined as Eulerian when:

— it begins from and ends with vertex 0_f representing the region containing the starter in forward orientation, i.e. $vreg(0_f) = COR[0]$;

— it passes through exactly one of the two versions of each edge ($e \in Ereg$ otherwise $\overline{e} \in Ereg$).

▶ **Proposition 8.1: An eulerian circuit is a valid oriented contig sequence for** $RegGraph$

An Eulerian circuit in $RegGraph = (Vreg, Ereg, \Phi)$ provides a valid sequence of oriented contigs (Definition 2.1).

▷ Proof

Let $RegGraph = (Vreg, Ereg, \Phi)$ *be a region graph. Denote by* $euc = (v_0, v_1, \ldots, v_{2m+n-1})$ *an Eulerian circuit in RegGraph. For each two consecutive oriented regions* v_i, v_j *in euc, there exists an edge* $e \in Ereg$ *such that* $\Phi(e) = (v_i, v_j)$*. According to Definition 8.1,* $vreg(v_i) = r_i$ *and* $vreg(v_j) = r_j$ *are also consecutive in the oriented region sequence that has originally built RegGraph, otherwise in its reverse. Thus, according to Algorithm 25, there is an edge* $(u, v) \in E$ *in* $MDCG$ *such that* (ctg_u, or_u) *and* (ctg_v, or_v) *equal* $r_i[|r_i| - 1]$ *and* $r_j[0]$*, respectively.*

◁

We can easily verify that the number of Eulerian circuits is bounded by $O(2^{m'})$, where $m' \leq m$ is the number of inverted repeats.

Now, given a region graph *RegGraph*, finding all the Eulerian circuits is equivalent to retrieve all the possible chloroplast genome forms. Each Eulerian circuit traverses exactly the same regions, but not necessary in the same orientations. Figure 38 gives the resulting region graph obtained from the input data given in Table 11.

We accept all the Eulerian circuits, although they may contradict the repeated region interval constraints given in Definitions 3.2 and 3.3. For example, the oriented region sequence $(0_f, 1_f, 2_f, 3_f, 1_f, 3_r)$ respects the definitions, where $(1_f, 1_f)$ is a DR and $(3_f, 3_r)$ is an IR. One of the Eulerian circuit produces the oriented region sequence $(0_f, 1_f, 2_f, 3_f, 1_r, 3_r)$. Region 1_f now evolves in a new IR $(1_f, 1_r)$. The order between the regions of the two IRs contradicts Definition 3.3.

9 $\mathcal{NP}$-completeness

To prove the $\mathcal{NP}$-completeness of one of the two hierarchical problem successions (decision version), it is sufficient to focus on only one scaffolding problem for the repeat. Here, we will focus on the decision version of $\mathcal{IRP}$ ($\mathcal{DIRP}$).

▶ Definition 9.1: $\mathcal{IRP}$ decision problem ($\mathcal{DIRP}$)

> Given a set of contigs $\mathcal{C}$, their multiplicities, a set of links $\mathcal{L}$, a starting contig s, two integers $k, m' \in \mathbb{N}$, is there a valid sequence of oriented regions for $\mathcal{IRP}$ with $\sum_{ir \in IR} replen(ir) \geq k$ and $|IR| \leq m'$, where IR is the solution set of inverted repeats?

▶ Proposition 9.1: $\mathcal{DIRP}$ is in $\mathcal{NP}$

> Given the input of $\mathcal{DIRP}$, a sequence of oriented regions SOR, the sequence of oriented contigs for each region COR, two integers $k, m' \in \mathbb{N}$. There is a polynomial time algorithm that checks if the given solution is valid and if its cumulative repeat length equals at least k and the number of repeats equals at most m'.

▷ Proof

Algorithm 26 verifies if the sequence of oriented regions is valid. It requires two traversals of SOR: (i) to identify which regions in the sequence form IRs; (ii) to verify if the order between the IRs is valid (thanks to the use of a LIFO). It also checks if the associated cumulative repeat length equals at least k and the number of repeats equals at most m'.

A LIFO ir_lifo is associated with four methods:

ir_lifo.PUT(x) *append x to the LIFO;*

ir_lifo.IS_EMPTY *returns true if the LIFO is empty;*

ir_lifo.PEEK *returns the last value in the LIFO;*

ir_lifo.POP *deletes the last value in the LIFO and returns it.*

Algorithm 27 verifies if the corresponding sequence of oriented contigs is valid. It first retrieves the total sequence of oriented contig SOC from the sequence of oriented regions SOR and the sequence of each forward regions COR. It traverses the oriented contigs in SOC, verify if each two consecutive

oriented contigs are linked in the link set, and count the number of times a contig occur. At the end, it checks if each contig does not appear more than its multiplicity number of times.

It is straightforward to see that Algorithm 26 and Algorithm 27 are linear according the size of SOR and COR. Remember we assume that the belonging test "is $x \in X$?" for an object x in a set X is in $\Theta(1)$.

◁

▶ Algorithm 26: Verify the validity of the sequence of oriented regions

Require: Input of $\mathcal{DIRP}$, a sequence of oriented regions SOR, the sequence of oriented contigs for each region COR, an integer $k \in \mathbb{N}$.
Ensure: Returns True if the sequence of oriented regions is valid.

```
1:  function IS_SOR_VALID( )
2:      IR ← { }                                   ▷ Set of inverted repeats
3:      count_reg ← HASHTABLE( ) ▷ Keep the orientations of a region in a hash
        table
4:      ▷ Identify the inverted repeats                                    ◁
5:      for (rid, ror) ∈ SOR do
6:          if rid ∉ count_reg then
7:              count_reg[rid] ← [ror]
8:          else
9:              if |count_reg[rid] = 2| then
10:                 return False
11:             if ror ≠ count_reg[rid][0] then
12:                 IR.ADD(rid)                ▷ Repeats are pairs of regions
13:             count_reg.APPEND(ror)
14:     ▷ Verify parameters k and m′                                        ◁
15:     if ∑_{ir∈IR} |COR[ir]| < k or |IR| > m′ then
16:         return False
17:     ▷ Verify the order between IRs                                      ◁
18:     ir_lifo ← LIFO( )
19:     for (rid, ror) ∈ SOR do
20:         if rid ∈ IR then
21:             if ir_lifo.IS_EMPTY( ) ∨ rid ≠ ir_lifo.PEEK( ) then
22:                 ir_lifo.PUT(rid)
23:             else
24:                 ir_lifo.POP( )
25:     return ir_lifo.IS_EMPTY( )
```

▶ Algorithm 27: Verify the validity of the sequence of oriented contigs

Require: Input of $\mathcal{DIRP}$, a sequence of oriented regions SOR, the sequence of oriented contigs for each region COR.
Ensure: Returns True if the sequence of oriented contigs is valid.

1: **function** IS_SOC_VALID( )
2: Transform SOR in the sequence of oriented contigs SOC using COR.
3: $contig_count \leftarrow$ HASHTABLE( )
4: **if** $SOC[0] \neq (s, f)$ **then** ▷ *The circuit must start with the starting contig in forward orientation*
5: **return** False
6: ▷ *Verify the links and count the occurrences* ◁
7: $(prev_c, prev_or) \leftarrow SOC[|SOC| - 1]$
8: ▷ *Count the contigs and verify the links* ◁
9: **for** $(c, or) \in SOC$ **do**
10: **if** $((prev_c, prev_or), (c, or)) \notin \mathcal{L}$ **then**
11: **return** False
12: **if** $c \notin contig_count$ **then**
13: $contig_count[c] \leftarrow 1$
14: **else**
15: $contig_count[c] \leftarrow contig_count[c] + 1$
16: $(prev_c, prev_or) \leftarrow (c, or)$
17: ▷ *Verify the multiplicity* ◁
18: **for** $c \in contig_count$ **do**
19: **if** $contig_count[c] > mult(c)$ **then**
20: **return** False

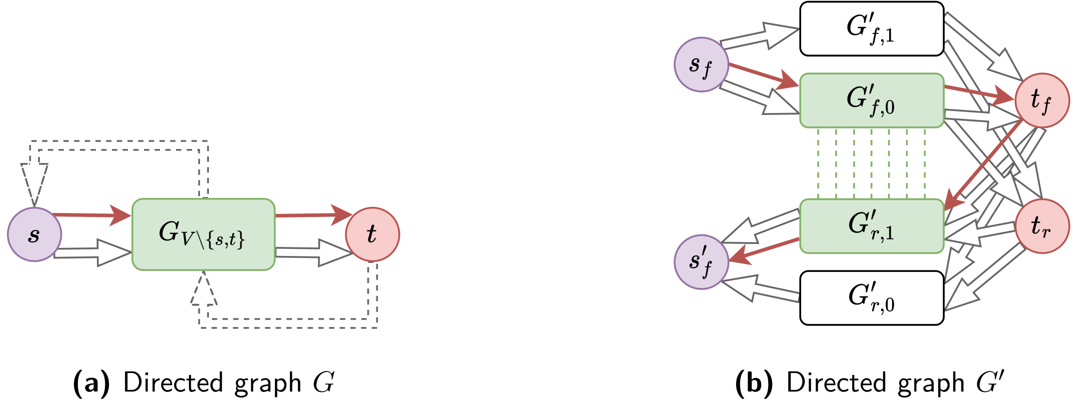
▶ Proposition 9.2: $\mathcal{NP}$-hardness of $\mathcal{IRP}$ decision problem

> $\mathcal{DIRP}$ is $\mathcal{NP}$-hard.

▷ Proof

By reduction from the longest path decision problem from vertex s to vertex t ($\mathcal{LPSTP}$), known to be $\mathcal{NP}$-complete (Schrijver, 2003).

Consider an instance $\mathbb{I} \in \mathcal{LPSTP}$ composed of a directed graph $G = (V, E)$, two vertices $s, t \in V$ and an integer $k \in \mathbb{N}$ (the hypothetical number of vertices between s and t in the longest path). We shall build an instance transform function tf such that $\mathbb{I} \in \mathcal{LPSTP} \iff tf(\mathbb{I}) \in \mathcal{DIRP}$. Function



(a) Directed graph G **(b)** Directed graph G'

Figure 39 – From a digraph G for $LPSTP$ to a digraph G' for $\mathcal{IRP}$. Bold red edges in both sub-figures correspond to the solution path for $LPSTP$ and $\mathcal{IRP}$ problems, respectively. **(a)** $G_{V\setminus\{s,t\}}$ is the subgraph induced by the vertex set $V\setminus\{s,t\}$. As the longest path exits s and enters t, dashed edges do not participate in the solution. **(b)** Green dashed line between vertices in G'_{f0} and G'_{r1} visualise the inverted fragments.

tf transforms graph G to graph $G' = (V', E')$, vertex $s \in V$ to vertices $s_f, s'_f \in V'$, $t \in V$ to $t_f, t_r \in V'$, k to $k' = 2k$, and fix parameter $m' = 1$. Figure 39 illustrates the transformation.

All four subgraphs $G'_{or,i}$ in Figure 39b can be seen as copies of $G_{V\setminus\{s,t\}} = \left(V\setminus\{s,t\}, E_{V\setminus\{s,t\}}\right)$ (the subgraph induced by the vertex set $V\setminus\{s,t\}$) in Figure 39a. For each $or \in \{f,r\}$, for each $i \in \{0,1\}$, $G'_{or,i} = \left(V'_{or,i}, E'_{or,i}\right)$ such that:

- *there is a bijective function $vtrans_{or,i}\colon V\setminus\{s,t\} \hookrightarrow\!\!\!\!\rightarrow V_{or,i}$, where $\forall\, v \in V'_{or,i}$, $vor(v) = or$ and $vocc(v) = i$;*
- *there is a bijective function $etrans_{or,i}\colon E_{V\setminus\{s,t\}} \hookrightarrow\!\!\!\!\rightarrow E'_{or,i}$ where $\forall\,(u,v) \in E_{V\setminus\{s,t\}}, \exists!\,(u',v') \in E'_{or,i}$ such that:*

$$vtrans_{or,i}(u), vtrans_{or,i}(v) = \begin{cases} u', v' & \text{if } or = f \\ v', u' & \text{if } or = r \end{cases}$$

There is a bijective function $vsttrans_{st}\colon \{s,t\} \hookrightarrow\!\!\!\!\rightarrow \{(s_f, s'_f), (t_f, t_r)\}$. There is a function $esttrans\colon \{(s,w) \in E\} \cup \{(u,t) \in E\} \to E'$ such that:

— $\forall (s, w) \in E$:

$$esttrans(s,w) = \{(s_f, vtrans_{f,i}(w)) \in V'^2 \mid i \in \{0,1\}\} \\ \cup \{(vtrans_{r,i}(w), s'_f) \in V'^2 \mid i \in \{0,1\}\}$$

— $\forall (u, t) \in E$:

$$esttrans(u,t) = \{(vtrans_{f,i}(u), t_f), (t_f, vtrans_{r,i}(u)) \in V'^2 \mid i \in \{0,1\}\} \\ \cup \{(vtrans_{f,i}(u), t_r), (t_r, vtrans_{r,i}(u)) \in V'^2 \mid i \in \{0,1\}\}$$

It is straightforward to see that there exists an algorithm in $O(|V| + |E|)$ that computes this transform function.

The sets $InvF$, $PInvF$ and $AInvF$ are built based on $G'_{or,i}$ graphs. Inverted fragments are visualised by green dashed vertical lines in Figure 39b, where $InvF = \{(i,j) \in V_{f,0} \times V_{r,1} \mid vtrans_{f,0}(i) = vtrans_{r,1}(j)\}$.

As the adjacent inverted fragments associate only vertices in $V_{f,0}$ with those in $V_{r,1}$, the path that maximises the number of contiguous inverted fragments exits s_f, goes through $G'_{f,0}$ to t_f (or t_r, it does not matter), and passes through $G'_{r,1}$ to s'_f.

Since $G'_{f,0}$ is a copy of $G_{V\setminus\{s,t\}}$, while $G'_{r,1}$ is its reverse graph, there is a bijection between $V'_{f,0}$ and $V'_{r,1}$ vertices sets.

The way G' is built implies only one IR is assembled (so parameter $m' = 1$ is respected). The length of this IR ($k' = 2k$) gives the hypothetical length of the longest path in $\mathcal{LPSTP}$.

To conclude, as there is a linear time complexity transform function tf such that $\mathbb{I} \in \mathcal{LPSTP} \iff tf(\mathbb{I}) \in \mathcal{DIRP}$, $\mathcal{DIRP}$ is at least $\mathcal{NP}$-hard.

$\triangleleft$

From Propositions 9.1 and 9.2 we conclude that $\mathcal{DIRP}$ is $\mathcal{NP}$-complete.

10 Numerical results

We develop `khloraascaf`[2], a python package that computes the scaffolding of chloroplast contigs. It can either use `Gurobi` solver or `CBC`. All the following runs have been executed on a Linux laptop computer (32GB RAM, Intel® Core™ i7–10610U CPU @ 1.80GHz ×8). Each time, we select `Gurobi` solver.

[2] https://khloraa-scaffolding.readthedocs.io/en/latest

10.1 Complexity validation on artificial data

`khloraascaf` is also accessible as an API, that permits in this section to study the combinatorial behaviour of $\mathcal{IRP}$.

We demonstrate in Section 9 that $\mathcal{DIRP}$ is in the general case $\mathcal{NP}$-complete. Furthermore, the heteroplasmy for the chloroplast genome is very often caused by the presence of an inverted repeat, that reverses the region(s) between it.

Thus, we artificially build a contig set with the associated attributes, and a link set, such that the genome architecture behind corresponds to the following circular sequence of oriented regions: $SC1 - IR - SC2 - \overline{IR}$. In the following, we run what corresponds to $\mathcal{IRP}$ computation in `khloraascaf` on two types of growing generated data: perfect and noisy artificial ones. To emphasise the effect of the inverted repeats in the complexity of $\mathcal{IRP}$, we fix the length of the single-copies, i.e. $|SC1| = |SC2| = |SC| = 20$, and we incrementally raise the length of the inverted repeat $|IR| = 20k$ for $k \in [\![1, 10]\!]$.

10.1.1 Perfect artificial data

The data generated for this section correspond to the smallest set of contigs, links, and the smallest multiplicities to make sure that $\mathcal{IRP}$ is feasible. The multiplied doubled contig graph associated with these perfect data has exactly the same topology as the one illustrated in Figure 33. For instance, testing perfect artificial data acts as a control for further tests. Table 14 gives some `Gurobi` metrics[3].

Observe that the gap is equal to 0 % and the problem is solved either during the presolving or the linear relaxation. Indeed, for the class of graph that contains only the perfect artificial data, there exist a polynomial algorithm. However, the relaxation time seems to fit an exponential distribution as well as for the B&B time, as shown in Figure 40, even though the distributions should be treated with caution because of the limited number of points.

10.1.2 Noisy artificial data

Here we test the behaviour of the solver when we add noise to the perfect artificial data. In that case, for each generated contig, the multiplicity has 25 % chance to be overestimated by one (that increases $InvF$, $PInvF$ and possibly $AInvF$ sets). Similarly, for each contig, there is 25 % of chance to create a new link to another randomly chosen contig. This can generate more loops, and can increase

[3]To reproduce the results, please refer to https://khloraascaf-results.readthedocs.io/en/latest/benchmark_4/.

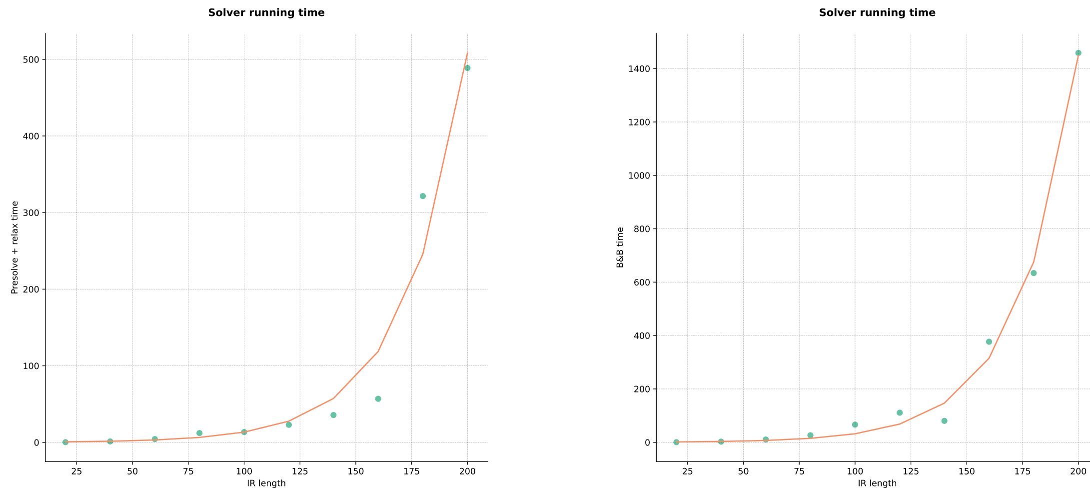


Figure 40 – Solver running time distributions for perfect artificial data. Points are measured times, the red curves correspond to the best ae^{bx} function applied on IR length axis.

the $AInvF$ set[4].

As expected, now the gap is not ensured to be null, and some instances are not solved at the presolving or at the linear relaxation steps.

These numerical results corroborate the $\mathcal{NP}$-complete demonstration for a more general class of graphs.

10.2 Synthetic chloroplast input data

In this section, we aim to validate experimentally the relevance of our scaffolding problem definition by running `khloraascaf` on synthetic data.

10.2.1 Input data generation

Here we briefly describe our protocol for input data generation[5]. 200 chloroplast genomes (selected in CpGDB[6]) were downloaded from the NCBI[7]. For each of them,

[4]To reproduce the results, please refer to https://khloraascaf-results.readthedocs.io/en/latest/benchmark_5/.

[5]For more details, please refer to https://khloraascaf-results.readthedocs.io/en/latest/benchmark_3/

[6]CpGDB: A Comprehensive Database of Chloroplast Genomes http://www.gndu.ac.in/CpGDB/index.aspx

[7]NCBI: The National Center for Biotechnology Information advances science and health by providing access to biomedical and genomic information https://www.ncbi.nlm.nih.gov/

Table 14 – **`Gurobi` solver metrics on perfect artificial growing data.** |V|, |E|, |SC|, |IR| and $|\mathcal{L}|$ respectively stand for the number of vertices, edges, contigs in each single-copy region, contigs in each region of the inverted repeat, and links; **Time**: the presolve time plus the relaxation time (above) and the B&B time (below); **Opt.**: the linear relaxation bound UB (above) and the integer optimal value Opt (below); **% Gap**: the MIP gap equals $100 \times \frac{(UB-Opt)}{UB}$; **Nodes**: number of explored B&B nodes; **Iter.**: number of iterations for the LP relaxation (above) and for the B&B phase (below).

|V|	|E|	|SC|	|IR|	$\|\mathcal{L}\|$	Time	Opt.	% Gap	Nodes	Iter.
160	244	20	20	122	0.25	39.00	0.00	1	1242
					0.48	39.00			2944
240	404	20	40	162	1.23	79.00	0.00	1	4690
					2.54	79.00			10 946
320	564	20	60	202	4.27	119.00	0.00	1	8945
					10.45	119.00			23 864
400	724	20	80	242	12.15	159.00	0.00	1	15 196
					26.29	159.00			39 562
480	884	20	100	282	13.46	199.00	0.00	1	23 109
					66.28	199.00			68 740
560	1044	20	120	322	22.91	239.00	0.00	1	44 048
					110.98	239.00			97 646
640	1204	20	140	362	35.68	279.00	0.00	1	46 071
					80.26	279.00			125 084
720	1364	20	160	402	56.89	319.00	0.00	1	75 371
					376.74	319.00			256 831
800	1524	20	180	442	321.52	359.00	0.00	1	71 971
					634.01	359.00			196 905
880	1684	20	200	482	488.74	399.00	0.00	1	88 157
					1458.66	399.00			236 406

a set of reads was generated. The contigs were generated with `Minia` (Chikhi and Rizk, 2012), a de Bruijn graph assembly approach. The links correspond to k-mer paths in the resulting compacted de Bruijn graph (cDBG) that connect two contigs. Finally, 31 instances were selected for which various difficulties have been detected, e.g. extra-links in the link set or combination of repeats.

The starter is the contig for which the matK gene, usually found in a single-copy region, maps on.

To obtain the multiplicity of a given contig c, we sum the length of the alignments of the reads mapping on c (we denote by $MapR_c$ the set containing the reads that map on c). This sum is defined as the coverage cov_c of the contig c by the reads:

$$cov_c = \sum_{read \in MapR_c} \left|align_c^{read}\right|$$

Table 15 – `Gurobi` solver metrics on noisy artificial growing data.
The column descriptions are the same as the one in Table 14. Except that because of the noise on multiplicities, the sum of contig multiplicity for each region can change: this is the value below the number of contig in columns |SC1|, |SC2| and |IR|. Similarly, because of the noise on the number of links, the below value for $\mathcal{L}$ corresponds to the number of noisy links.

|V|	|E|	|SC1|	|SC2|	|IR|	$\|\mathcal{L}\|$	Time	Opt.	% Gap	Nodes	Iter.
186	372	20	20	20	152	1.01	39.00	0.00	1	2862
		26	24	43	30	5.16	39.00			6888
280	688	20	20	40	212	6.89	79.00	0.00	1	6727
		28	23	89	50	12.26	79.00			19 930
366	946	20	20	60	262	42.52	123.50	3.64	1	16 821
		25	26	132	60	79.28	119.00			50 246
452	1208	20	20	80	320	90.06	161.50	1.55	1	27 822
		22	23	181	78	295.22	159.00			129 174
556	1366	20	20	100	322	196.04	199.00	0.00	1	42 292
		23	24	231	40	244.41	199.00			92 009
662	1804	20	20	120	412	1007.59	242.50	1.44	1	84 639
		29	26	276	90	1434.16	239.00			210 798
736	1946	20	20	140	454	1108.09	283.00	1.41	1	228 198
		24	26	318	92	3540.79	279.00			691 619
822	2212	20	20	160	514	1118.76	323.00	1.24	1	86 592
		26	27	358	112	2449.55	319.00			287 146
902	2362	20	20	180	542	1591.18	363.00	1.10	1	91 936
		26	27	398	100	2576.60	359.00			269 958
996	2656	20	20	200	602	2294.85	404.00	1.24	1	116 315
		26	26	446	120	3747.02	399.00			351 501

Where $align_c^{read}$ is the sequence representing the alignment of the read $read$ on the contig c. Its length equals the number of nucleotides of $read$ that match on c (identity or substitution). Then the multiplicity $mult(c)$ of c is obtained by normalising its coverage cov_c by this of the starter s, cov_s: $mult(c) = \max(1, \lceil cov_c/cov_s - 0.1 \rceil)$. As the multiplicity is an upper-bound of the usage of contig, we prefer to round up the normalisation only if the decimal part is greater than 0.1.

The existence-weight $wex(c)$ for a contig c is computed by counting the number of nucleotides of c that are covered by at least a gene of protein from a chloroplast near-species, normalised by the length of c.

10.2.2 The evaluation's metrics

For each synthetic instance, we know the sequence of the oriented contigs and the sequence of the oriented regions we search for. In the sequel we test our scaffolding

approach and evaluate the obtained region graph. For each instance, for each optimisation problem combination, we provide the following metrics:

— the total number of eulerian circuits in the region graph (genome forms);

— how many of them coincide with the sequence of oriented contigs we search for;

— how many of them coincide with the sequence of oriented regions we search for.

Evaluating the sequence of the oriented contigs is stringent. On the other hand, although a result can be evaluated as a false one, we can still retrieve the sequence of the chloroplast genome by applying an alternative sequence. As a consequence, for each instance we use `Quast` (Gurevich et al., 2013) to evaluate the sequences associated to each genome form. As the genome reference is known, `Quast` tries to find the minimum number of differences (relocation, inversion, indels) between the reference and the sequence we provide. Three metrics are chosen to evaluate the best genome form for each problem succession:

— the genome fraction of the reference;

— the number of misassemblies;

— the number of local misassemblies.

For more detailed descriptions of `Quast` metrics, you can refer to Appendix 3.1.

It would be expected that `Quast` metrics illustrate wrong assembly for the instances for which the sequence of the contigs, and a fortiori this of the regions, are not retrieved. Analogously, the instance that truthfully retrieves the sequences would have good `Quast` metric. However, these assertions may be contradicted because of the contig and the link sequences generation.

In Section 10.2.3, we provide the two metric sets when `khloraascaf` is applied to the original synthetic data, while in Section 10.2.4 the metrics are reported for a subset of modified synthetic data.

10.2.3 Initial version

Table 16 provides all the metrics defined above for the 31 instances. The instances are solved very quickly (solver times < 4.5 sec., Table 19). `khloraascaf` successfully founds the sequence of the oriented contigs in 20 of them and retrieves the sequence of the oriented regions in 28 of them[8]. Three categories of failures are identified.

[8] For more details on the result for the initial version of the synthetic data, please refer to https://khloraascaf-results.readthedocs.io/en/latest/benchmark_3/v1_order_analysis/ and https://khloraascaf-results.readthedocs.io/en/latest/benchmark_3/v1_quast_analysis/.

Wrong starting contig and multiplicities estimations This is the category for which our approach is dependent and sensible. In the presented version, we have used a given starting contig, and a wrong one can lead to reduce all the multiplicities. This is the case for Begonia_pulchrifolia and Lamprocapnos_spectabilis for which the starters are contigs that normally participate into an IR, that contradicts our assumption that the starter participates in an SC.

Independently of a right starting contig, the multiplicity computation is sensible from the noise on the contig coverage by the reads. Agathis_dammara and Pelargonium_nanum both suffer from only one contig under-estimated multiplicity.

$\mathcal{IRP}$'s objective: maximising the cumulative length of the minimum number of repeats The sequence of oriented regions for the Cucumix_hystrix's reference contains the following sub-sequences: $\left(\dots, IR, IR', \dots, \overline{IR'}, SC, \overline{IR}\right)$. As $\mathcal{IRP}$ also aims to minimise the number of repeats, it results in the merge of $IR{-}IR'$ and consequently does not retrieve SC, normally inserted between $\overline{IR'}$ and $\overline{IR}$.

Model's robustness While the sequences of oriented contigs for the repeats are retrieved, the ones of the single-copies suffer from extra-links combined with low, sometimes null, weights. On the one hand, in case of null weights, the circuits in Lathyrus_pubescens and Triosteum_pinnatifidum do not pass through contigs that must participate in the SCs. On the other hand, Podocarpus_totara possesses two objective-equivalent subpaths in an SC.

Surprisingly, our tool `khloraascaf` reversed some subparts of single-copies. This is due to extra-links, but they are specifically caused by the existence of very short IRs hidden in them (remember that the links correspond to paths between pairs of contigs in the cDBG). This behaviour is observed in Carpodetus_serratus, Jasminum_tortuosum and Lophocereus_schottii.

Nucleotide sequences misassemblies To avoid confusion, we always use nucl. seq. to denote "nucleotide sequence" to contrast with sequences of contigs/regions. In the following we analyse the instances presenting an unexpected behaviour for the `Quast` metrics, regarding if the sequences are found. Supplementary Table 18 permits verifying if the contigs that participate in the sequence have been correctly assembled. Except for Lathyrus_pubescens where one local misassembly is due to the missing contig in the sequence, all the (local) missassemblies in Commiphora_foliacea, Eucommia_ulmoides, Juniperus_scopulorum, Musa_ornata, Sciadopitys_verticillata, Taxus_baccata, Welwitschia_mirabilis provided by `Quast` are found in the nucl. seq. of links.

Table 16 – Sequence and Quast metrics for the initial synthetic data version. For each instance in the column **Instance**: **ILPs** provides the optimal (at most two) hierarchical problem successions; **Total** reports the number of eulerian circuits in the region graph (genome forms); SOC is the number of oriented contig sequences that equal to the reference oriented contig sequence; SOR is the number of oriented region sequences in bijection with the reference oriented region sequence; **%gnm** is the genome fraction of the best sequence produced by one of the genome forms; **#mis** are the number of misassemblies (left) and of local misassemblies (right).

			Successful		Quast		
Instance	**ILPs**	**Total**	SOC	SOR	**%gnm**	**#mis**	
Abies_alba	dr-sc	1	0	0	99.88	3	0
	ir-sc	2	1	2	100.00	0	0
Acorus_americanus	ir-sc	2	1	2	99.99	0	0
Agathis_dammara	dr-ir-sc	10	0	2	80.51	1	2
	ir-dr-sc	6	0	0	80.94	2	2
Azima_tetracantha	ir-sc	2	1	2	99.99	0	0
Begonia_pulchrifolia	—	—	—	—	—	—	—
Carpodetus_serratus	ir-sc	2	0	2	100.00	2	0
Circaeaster_agrestis	ir-sc	2	1	2	99.99	0	0
Clematis_repens	ir-sc	2	1	2	100.00	0	0
Commiphora_foliacea	ir-sc	4	1	4	98.82	0	5
Cucumis_hystrix	ir-sc	2	0	0	100.00	0	0
Eucommia_ulmoides	ir-sc	2	1	2	99.99	0	2
Jasminum_tortuosum	ir-sc	2	0	2	99.99	2	0
Juniperus_scopulorum	sc	1	1	1	99.72	0	1
Lamprocapnos_spectabilis	dr-sc	1	0	0	35.86	3	0
Lathyrus_pubescens	dr-sc	1	0	0	98.47	2	2
	ir-sc	2	0	2	98.50	0	2
Lophocereus_schottii	sc	1	0	1	99.88	1	0
Musa_ornata	ir-sc	2	1	2	99.97	0	2
Oenothera_glazioviana	ir-sc	2	1	2	100.00	0	0
Pelargonium_nanum	ir-sc	2	0	2	96.44	4	0
Podocarpus_totara	sc	1	0	1	99.71	4	0
Porphyra_purpura	dr-sc	1	1	1	100.00	0	0
Sagittaria_trifolia	ir-sc	2	1	2	100.00	0	0
Sciadopitys_verticillata	dr-sc	1	0	0	98.99	6	3
	ir-sc	2	1	2	99.00	1	4
Sciaphila_densiflora	sc	1	1	1	100.00	0	0
Selaginella_kraussiana	dr-sc	1	1	1	100.00	0	0
Selaginella_vardei	dr-sc	1	1	1	100.00	0	0
Taxus_baccata	sc	1	1	1	99.77	0	2
Triosteum_pinnatifidum	ir-dr-sc	2	0	2	99.28	0	2
Uvaria_macrophylla	ir-sc	2	1	2	99.90	0	0
Welwitschia_mirabilis	ir-sc	2	1	2	100.00	0	1

(the table continues on the next page)

Table 16, continued

Instance	ILPs	Total	Successful SOC	SOR	Quast %gnm	#mis	
Wolffia_australiana	`ir-sc`	4	1	4	99.99	0	0

10.2.4 Modified version

In this section, we present a manually changed synthetic data version to succeed the scaffolding. The goal is to precisely evaluate the robustness of `khloraascaf`. We detail the modifications bellow[9]:

Agathis_dammara The multiplicity of contig 1 is raised to be equal to 3.

Begonia_pulchrifolia Contig 4 becomes the starter. So according to the multiplicity computation described in Section 10.2.1, the multiplicities of contigs 1 to 5 become 1, while this one of contig 0 raises to 2.

Carpodetus_serratus Link $(10_r, 11_f)$ is deleted.

Jasminum_tortuosum Link $(6_f, 4_f)$ is added.

Lamprocapnos_spectabilis Contig 8 becomes the starter. So the multiplicities of contigs 2, 4, 6, 10 and 11 increase by one, while the ones of contigs 0, 1 and 3 increase by two.

Lathyrus_pubescens The weight of contig 12 raises to 0.01.

Lophocereus_schottii Link $(10_f, 3_f)$ is deleted.

Pelargonium_nanum The multiplicity of contig 2 raises from 3 to 4, without respecting the computation of the multiplicity described in Section 10.2.1.

Podocarpus_totara Link $(6_f, 11_r)$ is deleted.

Triosteum_pinnatifidum The weight of contig 7 raises to 0.01.

Table 17 provides all the metrics obtained by running `khloraascaf`. The instances are also solved very quickly (solver times < 3 sec., Table 20). It truthfully finds all the sequences for the modified synthetic data except for Agathis_dammara instance. Although all the repeats (both the direct and the inverted ones) have been

[9] For more details, please refer to https://khloraascaf-results.readthedocs.io/en/latest/benchmark_3/synthetic_data_v2/.

retrieved, one of the single-copy region has not been found. It can be explained by extra-links that create alternative paths with the same optimal value for $\mathcal{SCP}$. Note that the oriented region sequence has been still retrieved.

All the (local) misassemblies provided by Quast for Carpodetus_serratus, Lathyrus_pubescens, Pelargonium_nanum and Triosteum_pinnatifidum just concern the nucl. seq. of links that is not a khloraascaf issue[10].

Table 17 – Sequence and Quast metrics for the modified synthetic data version.
The caption is the same as in Table 16.

			Successful		Quast		
Instance	**ILPs**	**Total**	SOC	SOR	**%gnm**	**#mis**	
Agathis_dammara	dr-ir-sc	10	0	2	99.97	3	2
	ir-dr-sc	6	0	0	99.99	2	2
Begonia_pulchrifolia	ir-sc	2	1	2	99.99	0	0
Carpodetus_serratus	ir-sc	2	1	2	100.00	0	1
Jasminum_tortuosum	ir-sc	2	1	2	99.99	0	0
Lamprocapnos_spectabilis	ir-sc	2	1	2	99.63	0	3
Lathyrus_pubescens	dr-sc	1	0	0	99.01	2	4
	ir-sc	2	1	2	99.73	0	1
Lophocereus_schottii	sc	1	1	1	100.00	0	0
Pelargonium_nanum	ir-sc	2	1	2	97.66	1	2
Podocarpus_totara	sc	1	1	1	99.75	0	0
Triosteum_pinnatifidum	ir-dr-sc	2	1	2	99.53	0	2

11 Conclusion

While the scaffolding problem is traditionally defined with distances data between the contigs, we show it is possible to avoid them in the case of the well-studied circular chloroplast genomes. Based on their specificities, we provide a new scaffolding formulation focused on revealing structural haplotypes.

Under the assumption that chloroplast genomes possess few repeats, we formalise their architectures as combinations of direct and inverted repeats, joined by single-copies, where the repeats are couples of identical (or reversed) nucleotide sequences. We tackle the chloroplast genome scaffolding as a discrete optimisation problem

[10]For more details on the results for the cleared version of the synthetic data, please refer to https://khloraascaf-results.readthedocs.io/en/latest/benchmark_3/v2_order_analysis/ and https://khloraascaf-results.readthedocs.io/en/latest/benchmark_3/v2_quast_analysis/.

that yields three suboptimisation ones. We split the inherent multi-objective problem into one optimisation problem per region type. As a consequence, it is necessary to choose the order of subproblem resolutions as a function of the results of previously solved problems. This is what has been addressed through the hierarchical combination strategy. We model each subproblem with an ILP.

As our dedicated chloroplast scaffolding definition is a region-scaffolding-driven, the region graph is a natural data structure to reveal distinct genome forms that can coexist in a same cell. Indeed, particularly due to an IR flip-flop mechanism, regions between the IRs can be reversed during the genome replication process.

Moreover, we prove the decision version of the Chloroplast Scaffolding Problem ($\mathcal{CHSP}$) to be $\mathcal{NP}$-complete in the general case, even though numerical results on perfect artificial data suggest there is a class of $MDCG$ graphs where the problem is in $\mathcal{P}$. Without surprise, the noisy artificial data benchmark confirms the theoretical complexity.

We have implemented our approach and the ILP formulations in a `Python3` package, `khloraascaf`[11], that we test on synthetic chloroplast contigs and links data. When the input data permit finding the solution, `khloraascaf` successfully retrieves the genome forms. Even if the decision problem is $\mathcal{NP}$-complete, the small size of the input data enables to quickly solve optimally $\mathcal{CHSP}$.

Our results show that the scaffolding-repeat problem formulations $\mathcal{DRP}$ and $\mathcal{IRP}$ seem to be sufficient to scaffold the repeats. This tends to validate our assumptions on the small number of repeats, and especially on the sufficiency of defining the repeats as pairs of equal (reversed) nucleotide sequences.

12 Discussion and perspectives

While our scaffolding problem formulation seems to be sufficient to retrieve the repeats, it seems it is not fully suitable for single-copies. If we have applied the maximum-weighted circuit problem to scaffold the single-copies after having scaffolded the repeats, it was only with the intention of staying in the context of global optimisation. On the one hand, having weights on links may have been more appropriate than just considering weights on the contigs: in some sense, that is the purpose of distances. On the other hand, `khloraascaf` could initially scaffold the repeats and then propose several solutions to link them, e.g. scored by the weights.

Concerning the tests on synthetic data, we should use a more traditional assembly graph input: in fact, as revealed by comparing the `khloraascaf` results with the reference genomes, the used link generation suffers from local, or worse, global, misassemblies. As next step, we plan to inject `khloraascaf` into a state-of-

[11] https://pypi.org/project/khloraascaf/

the-art chloroplast genome pipeline, like `GetOrganelle`, and substitute what can be identified as the scaffolding part by our method. Hence, we should be able to relevantly compare `khloraascaf` approach with the state-of-the-art.

`khloraascaf` is sensible to the contig multiplicity computation. For now, a contig multiplicity is obtained by normalising its coverage by this of the starter. A better strategy may be to choose the smallest coverage for the normalisation as we expect the multiplicities to be upper-bounds of the contigs use.

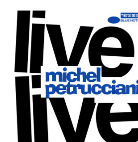
Another issue concerns the choice of the starter: while it depends on the result of the mapping of matK gene map on the contigs, $\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ problems may be adapted for a set of candidate for the starter.

To generalise $\mathcal{CHSP}$ on non-equally (reversed) pair of regions for the repeats, two combining ideas are proposed: on the one hand, we can add pairs of contigs to the repeated fragment sets from the user-input. On the other hand, $\mathcal{CHSP}$ should be able to handle the case when a single-copy region is in only one of the regions of a repeat: for now, the contiguity constraint and the objective exclude this case. The contiguity constraint can be adapted to accept only one contiguous region out of the two.

From a user-case perspective, the region graph data structure can be used to determine what genome forms are present in the read set, and in which proportion. Indeed, as the region graph explicitly describes the junctions between the regions, especially between the inverted repeats, one may extract the nucleotide sequences of these junctions to answer the existence of the forms in the read set.

V CONCLUSIONS AND PERSPECTIVES

Michel Petrucciani. (1994). Estate [Song]. On *Live*. Blue Note



In this chapter

Fragment graph 134
Thesis contribution 134
Short-term future work 135
Long-term future work 135
Chloroplast genome scaffolding 136
Thesis contribution 136
Short-term future work 137
Long-term future work 138

Since the 1980s, the fragment assembly's aims and their formulations have evolved following the development of sequencing technologies and the data characteristics they produce. Two main stages compose the fragment assembly process. The read assembly stage aims to assemble the reads into contigs based on sequence overlaps. Then, the scaffolding stage evolves orienting and ordering the contigs to form scaffolds. Finally, the scaffolds would represent the chromosomes with the minimum number of gaps and the minimum number of nucleotide differences.

The literature describes numerous formulations of combinatorial optimisation problems for the scaffolding problem, and as many approaches for dealing with large, noisy datasets. In this thesis, we outline the subsampling strategies, the combinatorial optimisation problem formulations and the solving approaches described in the state-of-the-art methods. We analyse and compare the fragment graph structures of the literature based on an implementation design. We then propose a formulation for the specific scaffolding of chloroplast genome, which we model with an Integer Linear Programming (ILP) and solve in `Python3`. In subsequent sections, we combine the conclusions of Chapters III and IV to discuss

their contribution to this thesis.

Fragment graph

Graphs are mathematical structures that are useful to store links between two fragments. Two main stages of the fragment assembly employ graphs to handle the input data or to output the result. In the read assembly stage, especially in the OLC approach, the fragments correspond to the reads and the links are the overlaps between two reads. In the scaffolding stage, the fragments represent the contigs and links can come from paired-end read or long read alignments against the contigs.

The literature describes three graph structures: a digraph, a bigraph and an ungraph. While the first one highlights visually the double-strand sequencing, its weakness is that it requires two vertices for each read (Kececioglu, 1991). Myers (1995) is the first to employ a bigraph to store overlaps. The key idea is to aggregate the two orientations of a read into only one vertex, a link and its reverse into only one edge. Finally, the ungraph structure associates one vertex for each of the two extremities of a fragment to simplify the graph traversal (Huson et al., 2002). The ungraph handles two sets of edges. A fragment-edge connects the two vertices corresponding to the two extremities of a fragment (the tail and the head). Two vertices representing the extremities of a fragment are connected by a fragment-edge. The second one is the multiset of link-edges. A link-edge represents both a link and its reverse.

Thesis contribution

Although some authors have provided conceptual descriptions of these graph structures, the latter have never been compared before from an implementation stat point of view. In this thesis, we have proposed an implementation design based on adjacency lists for each graph structure. At the opposite of a compressed sparse row or column, the adjacency lists are more suitable for adding or deleting vertices or edges in the graphs. We have also described transformation processes to pass from one implementation to another. We then have visualised the graph structures and their implementations in a map.

We have retained three implementations: *DGS* and *DGF* for the multidigraph and *BGU* for the multibigraph. After we have compared their memory consumption, we theoretically measure the cost function for each of their algorithm on elementary operations, such as iterating over the neighbours, adding or deleting a vertex or an edge. We come to the conclusion that if memory is the critical issue, then

the *BGU* implementation should be preferred. Otherwise, we recommend *DGS* and *DGF* for iterating over the neighbours and for deleting a vertex or an edge. *BGU* is preferable for adding vertices or edges. To conclude, *DGF* proves to be well-balanced and ideally tailored for dynamic graph operations. *DGF* is available in a `Python3` package named `revsymg`[1] and is easily installable via PyPI.

Short-term future work

We plan to implement *DGS*, *DGF* and *BGU* in `Rust` or an equivalent compiled programming language to compare them experimentally and confirm the theoretically obtained costs. Similarly, as the selection of a graph structure impacts further fragment assembly modelling, we aim to suggest integer linear models for path and vertex or edge coverage searches in each graph, analyse them, and compare them in theory and practice.

Long-term future work

This thesis is the first to propose fragment graph implementations for comparison. The graphs are expressed as adjacency lists. It would be useful to propose implementations based on compressed matrices and compare them mutually and with adjacency lists.

However, these structures are suitable for RAM storage. If the size of the instances exceeds the RAM storage capacity, the entire or partial graph should remain on disk. An efficient disk storage strategy may benefit from using the disk or, even better, CPU caches.

The `GFA`[2] and `PAF`[3] file formats, which store assembly graphs and matches between pairs of oriented fragments, could serve as a source of inspiration for disk storage methods. They are reminiscent of bidirected or undirected graphs. One strategy might be to keep the whole graph in this form on disk and convert a part of interest into a *DGS* or *DGF* in RAM.

This strategy may rely on neighbourhood analyses or the specific topology of fragment graphs. It should therefore be related to the research on graph algorithms, partitioning and matrix computation.

Although this work illustrates the use of fragment graphs for the assembly process, other areas involving nucleotide sequences may benefit from this work. For instance, genome graphs only represent variation between genomes of individuals of the same species, but the presence of complementary reverse regions implies

[1] https://pypi.org/project/revsymg/
[2] https://github.com/GFA-spec/GFA-spec
[3] https://github.com/lh3/miniasm/blob/master/PAF.md

double-stranded behaviour. In this context, one could examine the adaptation of fragment graphs.

Chloroplast genome scaffolding

While the scaffolding problem is traditionally defined with distances data between the contigs, we show it is possible to avoid them in the case of the well-studied circular chloroplast genomes. Based on their specificities, we provide a new scaffolding formulation focused on revealing structural haplotypes.

Thesis contribution

Under the assumption that chloroplast genomes possess few repeats, we formalise their architectures as combinations of direct and inverted repeats, joined by single-copies, where the repeats are couples of identical (or reversed) nucleotide sequences. We tackle the chloroplast genome scaffolding as a discrete optimisation problem that yields three suboptimisation ones. We split the inherent multi-objective problem into one optimisation problem per region type. As a consequence, it is necessary to choose the order of subproblem resolutions as a function of the results of previously solved problems. This is what has been addressed through the hierarchical combination strategy. We model each subproblem with an ILP.

As our dedicated chloroplast scaffolding definition is a region-scaffolding-driven, the region graph is a natural data structure to reveal the distinct genome forms that can coexist in a same cell. Indeed, particularly due to an IR flip-flop mechanism, regions between the IRs can be reversed during the genome replication process.

Moreover, we prove the Chloroplast Scaffolding Problem ($\mathcal{CHSP}$) to be $\mathcal{NP}$-complete in the general case, even though numerical results on perfect artificial data suggest there is a class of $MDCG$ graphs where the problem is in $\mathcal{P}$. Without surprise, the noisy artificial data benchmark confirms the theoretical complexity.

We have implemented our approach and the ILP formulations in a `Python3` package, `khloraascaf`[4], that we test on synthetic chloroplast contigs and links data. When the input data permit finding the solution, `khloraascaf` successfully retrieves the genome forms. Even if the decision problem is $\mathcal{NP}$-complete, the small size of the input data enables to quickly solve optimally $\mathcal{CHSP}$.

Our results show that the scaffolding-repeat problem formulations $\mathcal{DRP}$ and $\mathcal{IRP}$ seem to be sufficient to scaffold the repeats. This tends to validate our assumptions on the small number of repeats, and especially on the sufficiency of defining the repeats as pairs of equal (reversed) nucleotide sequences.

[4] https://pypi.org/project/khloraascaf/

Short-term future work

While our scaffolding problem formulation seems to be sufficient to retrieve the repeats, it seems it is not fully suitable for single-copies. If we have applied the maximum-weighted circuit problem to scaffold the single-copies after having scaffolded the repeats, it was only with the intention of staying in the context of global optimisation. On the one hand, having weights on links may have been more appropriate than just considering weights on the contigs: in some sense, that is the purpose of distances. On the other hand, `khloraascaf` could initially scaffold the repeats and then propose several solutions to link them, e.g. scored by the existence-weights.

Concerning the tests on synthetic data, we should use a more traditional assembly graph input: in fact, as revealed by comparing the `khloraascaf` results with the reference genomes, the used link generation suffers from local, or worse, global, misassemblies. As next step, we plan to inject `khloraascaf` into a state-of-the-art chloroplast genome pipeline, like `GetOrganelle`, and substitute what can be identified as the scaffolding part by our method. Hence, we should be able to relevantly compare `khloraascaf` approach with the state-of-the-art.

`khloraascaf` is sensible to the contig multiplicity computation. For now, a contig multiplicity is obtained by normalising its coverage by this of the starter. A better strategy may be to choose the smallest coverage for the normalisation as we expect the multiplicities to be upper-bounds of the contigs use.

Another issue concerns the choice of the starter: while it depends on the result of the mapping of matK gene map on the contigs, $\mathcal{DRP}$, $\mathcal{IRP}$ and $\mathcal{SCP}$ problems may be adapted for a set of candidate for the starter.

To generalise $\mathcal{CHSP}$ on non-equally (reversed) pair of regions for the repeats, two combining ideas are proposed: on the one hand, we can add pairs of contigs to the repeated fragment sets from the user-input. On the other hand, $\mathcal{CHSP}$ should be able to handle the case when a single-copy region is in only one of the regions of a repeat: for now, the contiguity constraint and the objective exclude this case. The contiguity constraint can be adapted to accept only one contiguous region out of the two.

From a user-case perspective, the region graph data structure can be used to determine what genome forms are present in the read set, and in which proportion. Indeed, as the region graph explicitly describes the junctions between the regions, especially between the inverted repeats, one may extract the nucleotide sequences of these junctions to answer the existence of the forms in the read set.

Long-term future work

There are numerous generic methods for genome assembly and scaffolding in the literature. When applying generic methods to the genomes of specific organisms, the application often results in data subsampling strategies and hyper-parameterisation. We aimed to develop a scaffolding method that is aware of the specific characteristics of the data at the core of the problem formulation.

The challenge lies in the mathematical formalisation of biological knowledge. It is a matter of clearly postulating the knowledge behind the specific method. Postulating provides a solid basis for developing an approach to solve the problem, and facilitate discussion.

As we succeeded in formulating the scaffolding for chloroplast genomes, we would be able to propose a specific formulation for the other organelles in a plant cell. Mitochondria, for instance, are likely to have the same genomic structures as the chloroplasts. The next step could be to combine the dedicated formulations. If they each represent one objective, future work will involve multi-objective studies.

We can also adapt our scaffolding formulation for the repeats to identify the regions resulting from an event of repeat degeneration. In this case, we could apply our methods to genomes of different organisms, such as bacterial, plant or animal genomes, without the need to assemble them.

BIBLIOGRAPHY

Adey, A., Kitzman, J. O., Burton, J. N., Daza, R., Kumar, A., Christiansen, L., Ronaghi, M., Amini, S., Gunderson, K. L., Steemers, F. J., and Shendure, J. (2014). In vitro, long-range sequence information for de novo genome assembly via transposase contiguity. *Genome Research*, 24(12):2041–2049.

Aganezov, S., Avdeyev, P., Alexeev, N., Rong, Y., and Alekseyev, M. A. (2022). Orienting Ordered Scaffolds: Complexity and Algorithms. *SN Computer Science*, 3(4):308.

Andonov, R., Djidjev, H., François, S., and Lavenier, D. (2019). Complete assembly of circular and chloroplast genomes based on global optimization. *Journal of Bioinformatics and Computational Biology*, 17(3):1950014.

Ankenbrand, M. J., Pfaff, S., Terhoeven, N., Qureischi, M., Gündel, M., Weiß, C. L., Hackl, T., and Förster, F. (2018). chloroExtractor: Extraction and assembly of the chloroplast genome from whole genome shotgun data. *Journal of Open Source Software*, 3(21):464.

Bakker, F. T., Lei, D., Yu, J., Mohammadin, S., Wei, Z., van de Kerke, S., Gravendeel, B., Nieuwenhuis, M., Staats, M., Alquezar-Planas, D. E., and Holmer, R. (2016). Herbarium genomics: Plastome sequence assembly from a range of herbarium specimens using an Iterative Organelle Genome Assembly pipeline. *Biological Journal of the Linnean Society*, 117(1):33–43.

Bankevich, A., Nurk, S., Antipov, D., Gurevich, A. A., Dvorkin, M., Kulikov, A. S., Lesin, V. M., Nikolenko, S. I., Pham, S., Prjibelski, A. D., Pyshkin, A. V., Sirotkin, A. V., Vyahhi, N., Tesler, G., Alekseyev, M. A., and Pevzner, P. A. (2012). SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology*, 19(5):455–477.

Batzoglou, S., Jaffe, D. B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J. P., and Lander, E. S. (2002). ARACHNE: A whole-genome shotgun assembler. *Genome Research*, 12(1):177–189.

Bendich, A. J. (1987). Why do chloroplasts and mitochondria contain so many copies of their genome? *BioEssays*, 6(6):279–282.

Bendich, A. J. (2004). Circular chloroplast chromosomes: The grand illusion. *The Plant Cell*, 16(7):1661–1666.

Bock, R. and Knoop, V., editors (2012). *Genomics of Chloroplasts and Mitochondria*, volume 35 of *Advances in Photosynthesis and Respiration*. Springer Netherlands, Dordrecht.

Bodily, P. M., Fujimoto, M. S., Snell, Q., Ventura, D., and Clement, M. J. (2016). ScaffoldScaffolder: Solving contig orientation via bidirected to directed graph reduction. *Bioinformatics*, 32(1):17–24.

Boetzer, M., Henkel, C. V., Jansen, H. J., Butler, D., and Pirovano, W. (2011). Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics*, 27(4):578–579.

Boetzer, M. and Pirovano, W. (2012). Toward almost closed genomes with GapFiller. *Genome Biology*, 13(6):R56.

Briot, N., Chateau, A., Coletta, R., de Givry, S., Leleux, P., and Schiex, T. (2014). An integer linear programming approach for genome scaffolding. In *WCB: Workshop on Constraint-Based Methods for Bioinformatics*, 10th Workshop on Constraint-Based Methods for Bioinformatics (WCB), 2014, page 16 p., Lyon, France.

Burton, J. N., Adey, A., Patwardhan, R. P., Qiu, R., Kitzman, J. O., and Shendure, J. (2013). Chromosome-scale scaffolding of de novo genome assemblies based on chromatin interactions. *Nature Biotechnology*, 31(12):1119–1125.

Chateau, A. and Giroudeau, R. (2014). Complexity and Polynomial-Time Approximation Algorithms around the Scaffolding Problem. In Dediu, A.-H., Martín-Vide, C., and Truthe, B., editors, *Algorithms for Computational Biology*, Lecture Notes in Computer Science, pages 47–58, Cham. Springer International Publishing.

Chateau, A. and Giroudeau, R. (2015). A complexity and approximation framework for the maximization scaffolding problem. *Theoretical Computer Science*, 595:92–106.

Cheng, H., Concepcion, G. T., Feng, X., Zhang, H., and Li, H. (2021). Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods*, 18(2):170–175.

Chikhi, R. and Rizk, G. (2012). Space-Efficient and Exact de Bruijn Graph Representation Based on a Bloom Filter. In Raphael, B. and Tang, J., editors, *Algorithms in Bioinformatics*, Lecture Notes in Computer Science, pages 236–248, Berlin, Heidelberg. Springer.

Chin, C.-S., Peluso, P., Sedlazeck, F. J., Nattestad, M., Concepcion, G. T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R., Morales-Cruz, A., Cramer, G. R., Delledonne, M., Luo, C., Ecker, J. R., Cantu, D., Rank, D. R., and Schatz, M. C. (2016). Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods*, 13(12):1050–1054.

Clark, S. C., Egan, R., Frazier, P. I., and Wang, Z. (2013). ALE: A generic assembly likelihood evaluation framework for assessing the accuracy of genome and metagenome assemblies. *Bioinformatics*, 29(4):435–443.

Coissac, E., Hollingsworth, P. M., Lavergne, S., and Taberlet, P. (2016). From barcodes to genomes: Extending the concept of DNA barcoding. *Molecular Ecology*, 25(7):1423–1428.

Coombe, L., Zhang, J., Vandervalk, B. P., Chu, J., Jackman, S. D., Birol, I., and Warren, R. L. (2018). ARKS: Chromosome-scale scaffolding of human genome drafts with linked read kmers. *BMC Bioinformatics*, 19(1):234.

Daniell, H., Lin, C.-S., Yu, M., and Chang, W.-J. (2016). Chloroplast genomes: Diversity, evolution, and applications in genetic engineering. *Genome Biology*, 17(1):134.

Davot, T., Chateau, A., Fossé, R., Giroudeau, R., and Weller, M. (2022). On a greedy approach for genome scaffolding. *Algorithms for Molecular Biology*, 17(1):16.

Dayarian, A., Michael, T. P., and Sengupta, A. M. (2010). SOPRA: Scaffolding algorithm for paired reads via statistical optimization. *BMC Bioinformatics*, 11(1):345.

Delahaye, C. and Nicolas, J. (2021). Sequencing DNA with nanopores: Troubles and biases. *PLOS ONE*, 16(10):e0257521.

Deng, X.-W., Wing, R. A., and Gruissem, W. (1989). The chloroplast genome exists in multimeric forms. *Proceedings of the National Academy of Sciences*, 86(11):4156–4160.

Dierckxsens, N., Mardulyn, P., and Smits, G. (2017). NOVOPlasty: De novo assembly of organelle genomes from whole genome data. *Nucleic Acids Research*, 45(4):e18.

Donmez, N. and Brudno, M. (2013). SCARPA: Scaffolding reads with practical algorithms. *Bioinformatics*, 29(4):428–434.

Edmonds, J. and Johnson, E. L. (1970). Matching: A well-solved class of integer linear programs. In *In: Combinatorial Structures and Their Applications (Gordon and Breach*, pages 89–92.

Ekim, B., Berger, B., and Chikhi, R. (2021). Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12(10):958–968.e6.

François, S., Andonov, R., Lavenier, D., and Djidjev, H. (2018). Global Optimization for Scaffolding and Completing Genome Assemblies. *Electronic Notes in Discrete Mathematics*, 64:185–194.

Franklin, R. E. and Gosling, R. G. (1953). The structure of sodium thymonucleate fibres. I. The influence of water content. *Acta Crystallographica*, 6(8):673–677.

Freudenthal, J. A., Pfaff, S., Terhoeven, N., Korte, A., Ankenbrand, M. J., and Förster, F. (2020). A systematic comparison of chloroplast genome assembly tools. *Genome Biology*, 21:254.

Gao, S., Nagarajan, N., and Sung, W.-K. (2011). Opera: Reconstructing Optimal Genomic Scaffolds with High-Throughput Paired-End Sequences. In Bafna, V. and Sahinalp, S. C., editors, *Research in Computational Molecular Biology*, Lecture Notes in Computer Science, pages 437–451, Berlin, Heidelberg. Springer.

Ghurye, J., Rhie, A., Walenz, B. P., Schmitt, A., Selvaraj, S., Pop, M., Phillippy, A. M., and Koren, S. (2019). Integrating Hi-C links with assembly graphs for chromosome-scale assembly. *PLOS Computational Biology*, 15(8):e1007273.

Gingeras, T., Milazzo, J., Sciaky, D., and Roberts, R. (1979). Computer programs for the assembly of DNA sequences. *Nucleic Acids Research*, 7(2):529–543.

Goldfarb, D. and Idnani, A. (1983). A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming*, 27(1):1–33.

Gould, S. B. (2012). Algae's complex origins. *Nature*, 492(7427):46–48.

Gritsenko, A. A., Nijkamp, J. F., Reinders, M. J., and de Ridder, D. (2012). GRASS: A generic algorithm for scaffolding next-generation sequencing assemblies. *Bioinformatics*, 28(11):1429–1437.

Gurevich, A., Saveliev, V., Vyahhi, N., and Tesler, G. (2013). QUAST: Quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075.

Gusfield, D. (2019). The RNA-Folding Problem. In Gusfield, D., editor, *Integer Linear Programming in Computational and Systems Biology: An Entry-Level Text and Course*, pages 105–121. Cambridge University Press, Cambridge.

Hernandez, D., François, P., Farinelli, L., Østerås, M., and Schrenzel, J. (2008). De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809.

Hierholzer, C. and Wiener, C. (1873). Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32.

Hon, T., Mars, K., Young, G., Tsai, Y.-C., Karalius, J. W., Landolin, J. M., Maurer, N., Kudrna, D., Hardigan, M. A., Steiner, C. C., Knapp, S. J., Ware, D., Shapiro, B., Peluso, P., and Rank, D. R. (2020). Highly accurate long-read HiFi sequencing data for five complex genomes. *Scientific Data*, 7(1):399.

Huson, D. H., Reinert, K., and Myers, E. W. (2002). The greedy path-merging algorithm for contig scaffolding. *Journal of the ACM*, 49(5):603–615.

Idury, R. M. and Waterman, M. S. (1995). A New Algorithm for DNA Sequence Assembly. *Journal of Computational Biology*, 2(2):291–306.

Jain, C. (2023). Coverage-preserving sparsification of overlap graphs for long-read assembly. *Bioinformatics*, 39(3):btad124.

Jin, J.-J., Yu, W.-B., Yang, J.-B., Song, Y., dePamphilis, C. W., Yi, T.-S., and Li, D.-Z. (2020). GetOrganelle: A fast and versatile toolkit for accurate de novo assembly of organelle genomes. *Genome Biology*, 21(1):241.

Kamath, G. M., Shomorony, I., Xia, F., Courtade, T. A., and Tse, D. N. (2017). HINGE: Long-read assembly achieves optimal repeat resolution. *Genome Research*, 27(5):747–756.

Karp, R. M. (1972). Reducibility among Combinatorial Problems. In Miller, R. E., Thatcher, J. W., and Bohlinger, J. D., editors, *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and Sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, The IBM Research Symposia Series, pages 85–103. Springer US, Boston, MA.

Kececioglu, J. D. (1991). *Exact and Approximation Algorithms for DNA Sequence Reconstruction.* PhD thesis, The University of Arizona.

Kececioglu, J. D. and Myers, E. W. (1995). Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1):7.

Kidwell, M. G. (2002). Transposable elements and the evolution of genome size in eukaryotes. *Genetica*, 115(1):49–63.

Kim, K.-J. and Lee, H.-L. (2005). Widespread Occurrence of Small Inversions in the Chloroplast Genomes of Land Plants. *Molecules and Cells*, 19(1):104–113.

Kobayashi, T., Takahara, M., Miyagishima, S.-y., Kuroiwa, H., Sasaki, N., Ohta, N., Matsuzaki, M., and Kuroiwa, T. (2002). Detection and Localization of a Chloroplast-Encoded HU-Like Protein That Organizes Chloroplast Nucleoids. *The Plant Cell*, 14(7):1579–1589.

Kolmogorov, M., Yuan, J., Lin, Y., and Pevzner, P. A. (2019). Assembly of long, error-prone reads using repeat graphs. *Nature Biotechnology*, 37(5):540–546.

Korbel, J. O., Urban, A. E., Affourtit, J. P., Godwin, B., Grubert, F., Simons, J. F., Kim, P. M., Palejev, D., Carriero, N. J., Du, L., Taillon, B. E., Chen, Z., Tanzer, A., Saunders, A. C. E., Chi, J., Yang, F., Carter, N. P., Hurles, M. E., Weissman, S. M., Harkins, T. T., Gerstein, M. B., Egholm, M., and Snyder, M. (2007). Paired-End Mapping Reveals Extensive Structural Variation in the Human Genome. *Science*, 318(5849):420–426.

Koren, S., Treangen, T. J., and Pop, M. (2011). Bambus 2: Scaffolding metagenomes. *Bioinformatics*, 27(21):2964–2971.

Koren, S., Walenz, B. P., Berlin, K., Miller, J. R., Bergman, N. H., and Phillippy, A. M. (2017). Canu: Scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, 27(5):722–736.

Kumar, R. A., Oldenburg, D. J., and Bendich, A. J. (2014). Changes in DNA damage, molecular integrity, and copy number for plastid DNA and mitochondrial DNA during maize development. *Journal of Experimental Botany*, 65(22):6425–6439.

Li, H. (2016). Minimap and miniasm: Fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110.

Li, M. (1990). Towards a DNA sequencing theory (learning a string). In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 125–134 vol.1.

Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., and Wang, J. (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272.

Lokshtanov, D., Saurabh, S., and Sikdar, S. (2009). Simpler Parameterized Algorithm for OCT. In Fiala, J., Kratochvíl, J., and Miller, M., editors, *Combinatorial Algorithms*, Lecture Notes in Computer Science, pages 380–384, Berlin, Heidelberg. Springer.

Long, L., Li, Y., Wang, S., Liu, Z., Wang, J., and Yang, M. (2023). Complete chloroplast genomes and comparative analysis of Ligustrum species. *Scientific Reports*, 13(1):212.

Luo, J., Lyu, M., Chen, R., Zhang, X., Luo, H., and Yan, C. (2019). SLR: A scaffolding algorithm based on long reads and contig classification. *BMC Bioinformatics*, 20(1):1–11.

Luo, J., Wang, J., Zhang, Z., Li, M., and Wu, F.-X. (2017). BOSS: A novel scaffolding algorithm based on an optimized scaffold graph. *Bioinformatics*, 33(2):169–176.

Luo, R., Liu, B., Xie, Y., Li, Z., Huang, W., Yuan, J., He, G., Chen, Y., Pan, Q., Liu, Y., Tang, J., Wu, G., Zhang, H., Shi, Y., Liu, Y., Yu, C., Wang, B., Lu, Y., Han, C., Cheung, D. W., Yiu, S.-M., Peng, S., Xiaoqian, Z., Liu, G., Liao, X., Li, Y., Yang, H., Wang, J., Lam, T.-W., and Wang, J. (2012). SOAPdenovo2: An empirically improved memory-efficient short-read de novo assembler. *GigaScience*, 1(1):18.

Mandric, I. and Zelikovsky, A. (2015). ScaffMatch: Scaffolding algorithm based on maximum weight matching. *Bioinformatics*, 31(16):2632–2638.

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770.

McKain, M. and afinit (2017). Mrmckain/Fast-Plast: Fast-Plast v.1.2.6. Zenodo.

Medvedev, P. and Brudno, M. (2009). Maximum Likelihood Genome Assembly. *Journal of Computational Biology*, 16(8):1101–1116.

Medvedev, P., Georgiou, K., Myers, G., and Brudno, M. (2007). Computability of Models for Sequence Assembly. In Giancarlo, R. and Hannenhalli, S., editors, *Algorithms in Bioinformatics*, Lecture Notes in Computer Science, pages 289–301, Berlin, Heidelberg. Springer.

Mehdi, K., Gibrat, J.-F., and Elloumi, M. (2017). Generations of Sequencing Technologies: From First to Next Generation. *Electromagnetic Biology and Medicine*, 9(3):8 p.

Miller, C. E., Tucker, A. W., and Zemlin, R. A. (1960). Integer Programming Formulation of Traveling Salesman Problems. *Journal of the ACM*, 7(4):326–329.

Miller, J. R., Delcher, A. L., Koren, S., Venter, E., Walenz, B. P., Brownley, A., Johnson, J., Li, K., Mobarry, C., and Sutton, G. (2008). Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics (Oxford, England)*, 24(24):2818–2824.

Myers, E. W. (1995). Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology*, 2(2):275–290.

Myers, E. W. (2005). The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85.

Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K. H., Remington, K. A., Anson, E. L., Bolanos, R. A., Chou, H. H., Jordan, C. M., Halpern, A. L., Lonardi, S., Beasley, E. M., Brandon, R. C., Chen, L., Dunn, P. J., Lai, Z., Liang, Y., Nusskern, D. R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G. M., Adams, M. D., and Venter, J. C. (2000). A whole-genome assembly of Drosophila. *Science (New York, N.Y.)*, 287(5461):2196–2204.

O'Connor, M., Peifer, M., and Bender, W. (1989). Construction of Large DNA Segments in Escherichia coli. *Science*, 244(4910):1307–1312.

Ohyama, K., Fukuzawa, H., Kohchi, T., Shirai, H., Sano, T., Sano, S., Umesono, K., Shiki, Y., Takeuchi, M., Chang, Z., Aota, S.-i., Inokuchi, H., and Ozeki, H. (1986). Chloroplast gene organization deduced from complete sequence of liverwort Marchantia polymorpha chloroplast DNA. *Nature*, 322(6079):572–574.

Palmer, J. D. (1983). Chloroplast DNA exists in two orientations. *Nature*, 301(5895):92–93.

Palmer, J. D. (1985). Comparative Organization of Chloroplast Genomes. *Annual Review of Genetics*, 19(1):325–354.

Peltola, H., Söderlund, H., and Ukkonen, E. (1984). SEQAID: A DNA sequence assembling program based on a mathematical model. *Nucleic Acids Research*, 12(1 Pt 1):307–321.

Pervez, M. T., ul Hasnain, M. J., Abbas, S. H., Moustafa, M. F., Aslam, N., and Shah, S. S. M. (2022). A Comprehensive Review of Performance of Next-Generation Sequencing Platforms. *BioMed Research International*, 2022:e3457806.

Pevzner, P. A. (1989). L-Tuple DNA Sequencing: Computer Analysis. *Journal of Biomolecular Structure and Dynamics*, 7(1):63–73.

Pevzner, P. A., Tang, H., and Waterman, M. S. (2001). An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 98(17):9748–9753.

Poloczek, M. and Szegedy, M. (2012). Randomized Greedy Algorithms for the Maximum Matching Problem with New Analysis. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 708–717.

Pop, M., Kosack, D. S., and Salzberg, S. L. (2004). Hierarchical Scaffolding With Bambus. *Genome Research*, 14(1):149–159.

Qin, M., Wu, S., Li, A., Zhao, F., Feng, H., Ding, L., and Ruan, J. (2019). LRScaf: Improving draft genomes using long noisy reads. *BMC Genomics*, 20(1):955.

Räihä, K.-J. and Ukkonen, E. (1981). The Shortest Common Supersequence Problem over Binary Alphabet is NP-Complete. *Theor. Comput. Sci.*

Rice, E. S. and Green, R. E. (2019). New Approaches for Genome Assembly and Scaffolding. *Annual Review of Animal Biosciences*, 7:17–40.

Roy, R. S., Chen, K. C., Sengupta, A. M., and Schliep, A. (2012). SLIQ: Simple Linear Inequalities for Efficient Contig Scaffolding. *Journal of Computational Biology*, 19(10):1162–1175.

Ruan, J. and Li, H. (2020). Fast and accurate long-read assembly with wtdbg2. *Nature Methods*, 17(2):155–158.

Sahlin, K., Vezzi, F., Nystedt, B., Lundeberg, J., and Arvestad, L. (2014). BESST - Efficient scaffolding of large fragmented assemblies. *BMC Bioinformatics*, 15(1):281.

Salmela, L., Mäkinen, V., Välimäki, N., Ylinen, J., and Ukkonen, E. (2011). Fast scaffolding with small independent mixed integer programs. *Bioinformatics*, 27(23):3259–3265.

Sancho, R., Cantalapiedra, C. P., López-Alvarez, D., Gordon, S. P., Vogel, J. P., Catalán, P., and Contreras-Moreira, B. (2018). Comparative plastome genomics and phylogenomics of Brachypodium: Flowering time signatures, introgression and recombination in recently diverged ecotypes. *New Phytologist*, 218(4):1631–1644.

Sanger, F., Coulson, A. R., Hong, G. F., Hill, D. F., and Petersen, G. B. (1982). Nucleotide sequence of bacteriophage λ DNA. *Journal of Molecular Biology*, 162(4):729–773.

Sanger, F., Nicklen, S., and Coulson, A. R. (1977). DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467.

Schrijver, A. (2003). *Combinatorial Optimization: Polyhedra and Efficiency*. Springer Science & Business Media.

Seyer, P., Kowallik, K. V., and Herrmann, R. G. (1981). A physical map of Nicotiana tabacum plastid DNA including the location of structural genes for ribosomal RNAs and the large subunit of ribulose bisphosphate carboxylase/oxygenase. *Current Genetics*, 3(3):189–204.

Shafin, K., Pesout, T., Lorig-Roach, R., Haukness, M., Olsen, H. E., Bosworth, C., Armstrong, J., Tigyi, K., Maurer, N., Koren, S., Sedlazeck, F. J., Marschall, T., Mayes, S., Costa, V., Zook, J. M., Liu, K. J., Kilburn, D., Sorensen, M., Munson, K. M., Vollger, M. R., Monlong, J., Garrison, E., Eichler, E. E., Salama, S., Haussler, D., Green, R. E., Akeson, M., Phillippy, A., Miga, K. H., Carnevali, P., Jain, M., and Paten, B. (2020). Nanopore sequencing and the Shasta toolkit enable efficient de novo assembly of eleven human genomes. *Nature Biotechnology*, 38(9):1044–1053.

Shapiro, M. B. (1967). An Algorithm for Reconstructing Protein and RNA Sequences. *Journal of the ACM*, 14(4):720–731.

Shinozaki, K., Ohme, M., Tanaka, M., Wakasugi, T., Hayashida, N., Matsubayashi, T., Zaita, N., Chunwongse, J., Obokata, J., Yamaguchi-Shinozaki, K., Ohto, C., Torazawa, K., Meng, B., Sugita, M., Deno, H., Kamogashira, T., Yamada, K., Kusuda, J., Takaiwa, F., Kato, A., Tohdoh, N., Shimada, H., and Sugiura, M. (1986). The complete nucleotide sequence of the tobacco chloroplast genome: Its gene organization and expression. *The EMBO Journal*, 5(9):2043–2049.

Sommer, D. D., Delcher, A. L., Salzberg, S. L., and Pop, M. (2007). Minimus: A fast, lightweight genome assembler. *BMC bioinformatics*, 8:64.

Staden, R. (1979). A strategy of DNA sequencing employing computer programs. *Nucleic Acids Research*, 6(7):2601–2610.

Staden, R. (1980). A new computer method for the storage and manipulation of DNA gel reading data. *Nucleic Acids Research*, 8(16):3673–3694.

Sun, J., Wang, Y., Liu, Y., Xu, C., Yuan, Q., Guo, L., and Huang, L. (2020). Evolutionary and phylogenetic aspects of the chloroplast genome of Chaenomeles species. *Scientific Reports*, 10(1):11466.

Thode, V. A., Oliveira, C. T., Loeuille, B., Siniscalchi, C. M., and Pirani, J. R. (2021). Comparative analyses of Mikania (Asteraceae: Eupatorieae) plastomes and impact of data partitioning and inference methods on phylogenetic relationships. *Scientific Reports*, 11(1):13267.

Trevors, J. T. (1996). Genome size in bacteria. *Antonie van Leeuwenhoek*, 69(4):293–303.

Tsai, C.-H. and Strauss, S. H. (1989). Dispersed repetitive sequences in the chloroplast genome of Douglas-fir. *Current Genetics*, 16(3):211–218.

Turmel, M., Otis, C., and Lemieux, C. (2017). Divergent copies of the large inverted repeat in the chloroplast genomes of ulvophycean green algae. *Scientific Reports*, 7(1):994.

Venter, J. C., Smith, H. O., and Hood, L. (1996). A new strategy for genome sequencing. *Nature*, 381(6581):364–366.

Wang, W. and Lanfear, R. (2019). Long-Reads Reveal That the Chloroplast Genome Exists in Two Distinct Versions in Most Plants. *Genome Biology and Evolution*, 11(12):3372–3381.

Watson, J. D. and Crick, F. H. C. (1953). The Structure of Dna. *Cold Spring Harbor Symposia on Quantitative Biology*, 18:123–131.

Weller, M., Chateau, A., and Giroudeau, R. (2015). Exact approaches for scaffolding. *BMC Bioinformatics*, 16(14):S2.

Xiao-Ming, Z., Junrui, W., Li, F., Sha, L., Hongbo, P., Lan, Q., Jing, L., Yan, S., Weihua, Q., Lifang, Z., Yunlian, C., and Qingwen, Y. (2017). Inferring the evolutionary mechanism of the chloroplast genome size by comparing whole-chloroplast genome sequences in seed plants. *Scientific Reports*, 7(1):1555.

Zerbino, D. R. and Birney, E. (2008). Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829.

APPENDIX

In this chapter

1 Repeated fragment set functions A1
2 Reduction of the repeated fragment sets A2
2.1 Repeated fragment set reductions . . . A2
2.2 Pairs of repeated fragment set reductions . . . A5
2.3 Adjacent repeated fragment set reductions . . . A8
3 Metrics A17
3.1 Quast metrics . . . A17
4 Supplementary results A18
4.1 v1 scaffolding benchmark . . . A18
4.2 v2 scaffolding benchmark . . . A20

1 Repeated fragment set functions

$$
\begin{aligned}
dirfrag\colon V &\to DirF \\
v &\mapsto \begin{cases}
(v, v') \quad \text{if } occ_v \mid 2 \\
\text{where } occ_{v'} = occ_v + 1 \\
(v', v) \quad \text{else } occ_v \nmid 2 \\
\text{where } occ_{v'} = occ_v - 1 \\
or_v = or_{v'}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
invfrag\colon V &\to InvF \\
v &\mapsto \begin{cases}
(v, v') \quad \text{if } or_v = 0 \\
\text{where } or_{v'} = 1 \wedge occ_{v'} = occ_v + 1 \\
(v', v) \quad \text{else } or_v = 1 \\
\text{where } or_{v'} = 0 \wedge occ_{v'} = occ_v - 1
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
diradj\colon E &\to E \\
(u,v) &\mapsto (u',v') \\
&\quad ctg_{u'} = ctg_u \\
&\quad or_{u'} = or_u \\
&\quad occ_{u'} = \begin{cases} occ_u + 1 \text{ if } occ_u \mid 2 \\ occ_u - 1 \text{ otherwise} \end{cases} \\
&\quad ctg_{v'} = ctg_v \\
&\quad or_{v'} = or_v \\
&\quad occ_{v'} = \begin{cases} occ_v + 1 \text{ if } occ_v \mid 2 \\ occ_v - 1 \text{ otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
invadj\colon E &\to E \\
(u,v) &\mapsto (v',u') \\
&\quad ctg_{v'} = ctg_v \\
&\quad or_{v'} = 1 - or_v \\
&\quad occ_{v'} = 2\left\lfloor \frac{occ_v}{2} \right\rfloor + (1 - or_v) \\
&\quad ctg_{u'} = ctg_u \\
&\quad or_{u'} = 1 - or_u \\
&\quad occ_{u'} = 2\left\lfloor \frac{occ_u}{2} \right\rfloor + (1 - or_u)
\end{aligned}
$$

2 Reduction of the repeated fragment sets

In this section, we give prove the minimality and the sufficiency of the repeated fragment sets $(RepF, PRepF, ARepF)$ definitions.

2.1 Repeated fragment set reductions

Here we consider the reduction operations for $DirF$ and $InvF$, and conclude on their minimality. Only two vertices $i,j \in V$ with the same identifier, i.e. $ctg_i = ctg_j$ can form a repeated fragment. The relative orientations between i and j is determined according to the type of the repeat the fragment is associated to. The occurrences of the two vertices in a repeated fragment must differ.

Two combinatorial reductions are necessary — *commutative* and *pairing* reductions:

i. commutative reduction means that $\forall (i,j) \in RepF, (j,i) \notin RepF$;

ii. pairing reduction means that it is not necessary to pair all the occurrences cross all the occurrences, and we can group by consecutive occurrences without any intersection.

Reminder of the $DirF$ definition:

$$DirF = \bigcup_{c \in \mathcal{R}} \left\{ \begin{array}{l} (i,j) \in V^2 \text{ s.t.} \\ \quad ctg_i = ctg_j = c \\ \quad \wedge\, or_i = or_j \in \{f, r\} \\ \quad \wedge\, occ_i = occ_j - 1 = 2k \\ \qquad 0 \leq k < \left\lfloor \dfrac{mult(c)}{2} \right\rfloor \end{array} \right\}$$

▶ Proposition 2.1: Commutative reduction for $DirF$

$$\forall (i,j) \in DirF, (j,i) \notin DirF$$

▷ Proof

Let $(i,j) \in DirF$. $occ_j = occ_i + 1$ so $(j,i) \notin DirF$

◁

▶ Proposition 2.2: Pairing reduction for $DirF$

The direct fragment set contains a sufficient number of direct fragments to form all the solutions.

▷ Proof

The idea is to find the minimum occurrence difference between two direct fragments such that they can both be chosen to form DRs. Let $(i,j) \in DirF$ be a direct fragment and suppose that it is chosen to participate into a DR. Let $k,l \in V \mid ctg_k = ctg_l = ctg_i = ctg_j \wedge or_k = or_l = or_i = or_j$ be a candidate for the next direct fragment:

— *the occurrences of k and l must differ from the ones of i and j, as a vertex can occupy at most one position, so $occ_j < occ_k$;*

— *according to Proposition 2.1*, $occ_k < occ_l$ *and since occurrences are integers*, $occ_k = occ_l - 1$.

So $occ_i = occ_j - 1 \leq occ_k - 2 = occ_l - 3 \implies occ_i + 3 \leq occ_l$. *The smallest possible values for* occ_k *and* occ_l *equal* $occ_i + 2$ *and* $occ_j + 2$.

◁

▶ Proposition 2.3

Direct fragment set $DirF$ is a minimal set for $\mathcal{DRP}$.

▷ Proof

Firstly, we must keep the direct fragment for both the forward ($or_i = or_j = 0$) *and the reverse orientations* ($or_i = or_j = 1$) *as the scaffolding problem aims to order and orientate the contigs, so by definition we cannot determine which orientation will participate in the order. Secondly, making step bigger than* $2k$ *prevents to use all the occurrences of a contig, so it contradicts the contig multiplicity definition.*

◁

Reminder of the $InvF$ definition:

$$InvF = \bigcup_{c \in \mathcal{R}} \left\{ \begin{array}{l} (i,j) \in V^2 \text{ s.t.} \\ \quad ctg_i = ctg_j = c \\ \quad \wedge\, or_i = f \wedge or_j = r \\ \quad \wedge\, occ_i = occ_j - 1 = 2k \\ \qquad 0 \leq k < \left\lfloor \dfrac{mult(c)}{2} \right\rfloor \end{array} \right\}$$

▶ Proposition 2.4: Commutative reduction for $InvF$

$$\forall\, (i,j) \in InvF, (j,i) \notin InvF$$

▷ Proof

Let $(i,j) \in InvF$. $occ_j = occ_i + 1$ *so* $(j,i) \notin InvF$

◁

▶ Proposition 2.5: Pairing reduction for $InvF$

> The inverted fragment set contains a sufficient number of inverted fragments to form all the solutions.

▷ Proof

The idea is to find the minimum occurrence difference between two inverted fragments such that they can both be chosen to form IRs. Let $(i, j) \in InvF$ be an inverted fragment and suppose that it is chosen to participate into an IR. Let $k, l \in V \mid ctg_k = ctg_l = ctg_i = ctg_j \wedge or_k \neq or_l$ be a candidate for the next direct fragment:

- *to respect the property given by Proposition 2.4, $or_k = 1 - or_l = 0$*
- *the occurrences of k and l must differ from the ones of i and j when their orientations are matching, as a vertex can occupy at most one position. Furthermore, for a given occurrence and a given identifier, the two orientations are mutually exclusive, so $occ_j < occ_k$;*
- *according to Proposition 2.4, $occ_k < occ_l$ and since occurrences are integers, $occ_k = occ_l - 1$.*

So $occ_i = occ_j - 1 \leq occ_k - 2 = occ_l - 3 \implies occ_i + 3 \leq occ_l$. The smallest possible values for occ_k and occ_l equal $occ_i + 2$ and $occ_j + 2$.

◁

▶ Proposition 2.6

> Inverted fragment set $InvF$ is a minimal set for $\mathcal{IRP}$.

▷ Proof

Step bigger than $2k$ prevents to use all the occurrences of a contig, so it contradicts the contig multiplicity definition.

◁

2.2 Pairs of repeated fragment set reductions

Here we consider the reduction operations for $PDirF$ and $PInvF$, and conclude on their minimality. Just the *commutative reduction* is necessary.

Reminder of pairs of direct fragments set definition:

$$PDirF = \left\{ \begin{array}{l} \big((i,j),(k,l)\big) \in DirF^2 \text{ s.t.} \\ \quad ctg_j < ctg_k \\ \quad \vee \\ \quad ctg_j = ctg_k \wedge occ_j < occ_k \end{array} \right\}$$

▶ Proposition 2.7: Commutative reduction for $PDirF$

$$\forall \, ((i,j),(k,l)) \in PDirF, ((k,l),(i,j)) \notin PDirF$$

▷ Proof

Let $((i,j),(k,l)) \in PDirF$:

— *if $ctg_j < ctg_k$ thus $ctg_l > ctg_i$ so $((k,l),(i,j)) \notin PDirF$*

— *else $ctg_j = ctg_k \wedge occ_j < occ_k$ thus $occ_l > occ_i$ so $((k,l),(i,j)) \notin PDirF$*

◁

▶ Proposition 2.8

Pair of direct fragment set $PDirF$ is a minimal set for $\mathcal{DRP}$.

▷ Proof

By reductio ad absurdum*: if $PDirF$ can be minimised to a $PDirF'$ set, then there exists a pair of direct fragments $(p,q) \in PDirF \mid (p,q) \notin PDirF'$.*

Let $p = (i,j)$ and $q = (k,l)$, and let $q_2 = (m,n) \mid (p,q_2) \in PDirF' \wedge (q,q_2) \in PDirF'$. i,j,k,l,m,n vertices can simultaneously participate in the solution.

Let build a counter-example: suppose that we have the order $i\,k\,l\,j\,m\,n$. This implies the direct fragments p and q fall into a forbidden case for $\mathcal{DRP}$, but this is not detected by Constraints C14 and C15 because $(p,q) \notin PDirF'$. According to Constraint C16, the direct fragments p and q_2 can both match, and direct fragments q and q_2 too. So $m_p = m_q = m_{q_2} = 1$: absurd because p and q are nested so m_p and m_q should be equal to 0.

◁

Reminder of pairs of inverted fragments set definition:

$$PInvF = \left\{ \begin{array}{l} \big((i,j),(k,l)\big) \in InvF^2 \text{ s.t.} \\ \quad ctg_j < ctg_k \\ \quad \vee \\ \quad ctg_j = ctg_k \wedge occ_j < occ_k \end{array} \right\}$$

▶ **Proposition 2.9: Commutative reduction for $PInvF$**

$$\forall\, ((i,j),(k,l)) \in PInvF, ((k,l),(i,j)) \notin PInvF$$

▷ Proof

Let $((i,j),(k,l)) \in PInvF$*:*

— *if* $ctg_j < ctg_k$ *thus* $ctg_l > ctg_i$ *so* $((k,l),(i,j)) \notin PInvF$

— *else* $ctg_j = ctg_k \wedge occ_j < occ_k$ *thus* $occ_l > occ_i$ *so* $((k,l),(i,j)) \notin PInvF$

◁

▶ **Proposition 2.10**

Pair of inverted fragment set $PInvF$ is a minimal set for $\mathcal{IRP}$.

▷ Proof

By reductio ad absurdum*: if* $PInvF$ *can be minimised to a* $PInvF'$ *set, then there exists a pair of inverted fragments* $(p,q) \in PInvF \mid (p,q) \notin PInvF'$*.*

Let be $p = (i,j)$ *and* $q = (k,l)$*, and let* $q_2 = (m,n) \mid (p,q_2) \in PInvF' \wedge (q,q_2) \in PInvF'$*.* i,j,k,l,m,n *vertices can simultaneously participate in the solution.*

Let build a counter-example: suppose that we have the order $i\,k\,j\,l\,m\,n$*. This implies the inverted fragments* p *and* q *fall into a forbidden case for* $\mathcal{DRP}$*, but this is not detected by Constraints C14 and C15 because* $(p,q) \notin PInvF'$*. According to Constraint C16, the inverted fragments* p *and* q_2 *can both match, and inverted fragments* q *and* q_2 *too. So* $m_p = m_q = m_{q_2} = 1$*: absurd because* p *and* q *intersect so* m_p *and* m_q *should be equal to* 0*.*

◁

2.3 Adjacent repeated fragment set reductions

Here we consider the reduction operations for $ADirF$ and $AInvF$, and conclude on their minimality.

Reminder of adjacent direct fragments set definition:

$$ADirF = \left\{ \begin{array}{l} (u,v) \in E \text{ s.t.} \\ \quad ctg_u \neq ctg_v \\ \quad \wedge\, occ_u = 2k \\ \qquad 0 \leq k < \left\lfloor \frac{mult_u}{2} \right\rfloor \\ \quad \wedge\, occ_v = 2k' \\ \qquad 0 \leq k' < \left\lfloor \frac{mult_v}{2} \right\rfloor \end{array} \right\} \bigcup \left\{ \begin{array}{l} (u,v) \in E \text{ s.t.} \\ \quad ctg_u = ctg_v \\ \quad \wedge\, (or_u = 0 \vee or_v = 0) \\ \quad \wedge\, occ_u = 2k \\ \quad \wedge\, occ_v = 2k' \\ \qquad 0 \leq k < k' < \left\lfloor \frac{mult_u}{2} \right\rfloor \end{array} \right\}$$

This set is the minimum exhaustive set of canonical edges which represent adjacent direct fragments. Two combinatorial reductions are necessary — *canonical reduction for different identifiers* (Propositions 2.11 and 2.12) and *same identifiers occurrences* reductions (Propositions 2.13 and 2.14).

▶ Proposition 2.11: Canonical reductions for $ADirF$ for different identifiers

> $ADirF$ set contains only two of the eight edges that exist between two adjacent direct fragments when the vertex' identifiers are different, i.e.

$\forall (i,j) \in DirF, \forall (k,l) \in DirF \mid ctg_i \neq ctg_k$:

$$\begin{array}{l} (i,k) \in E \\ \iff (i,l) \in E \\ \iff (j,k) \in E \\ \iff (j,l) \in E \end{array} \iff \begin{array}{l} (i,k) \in ADirF \\ \wedge(\overline{k},\overline{i}) \in ADirF \\ \wedge(i,l) \notin ADirF \\ \wedge(\overline{l},\overline{i}) \notin ADirF \\ \wedge(j,k) \notin ADirF \\ \wedge(\overline{k},\overline{j}) \notin ADirF \\ \wedge(j,l) \notin ADirF \\ \wedge(\overline{l},\overline{j}) \notin ADirF \end{array}$$

$$\begin{array}{l} (k,i) \in E \\ \iff (k,j) \in E \\ \iff (l,i) \in E \\ \iff (l,j) \in E \end{array} \iff \begin{array}{l} (k,i) \in ADirF \\ \wedge(\overline{i},\overline{k}) \in ADirF \\ \wedge(k,j) \notin ADirF \\ \wedge(\overline{j},\overline{k}) \notin ADirF \\ \wedge(l,i) \notin ADirF \\ \wedge(\overline{i},\overline{l}) \notin ADirF \\ \wedge(l,j) \notin ADirF \\ \wedge(\overline{j},\overline{l}) \notin ADirF \end{array}$$

▷ Proof

The reciprocal are trivial as $ADirF \subset E$.

All the edges using only i, $\overline{i}$, k or $\overline{k}$ belong to $ADirF$ as all the vertex' occurrences are even. While all the edges using j, $\overline{j}$, l or $\overline{l}$ cannot belong to $ADirF$ as one of the vertex' occurrence is odd.

◁

▶ Proposition 2.12: Minimum $ADirF$ set for different identifiers

When the vertex' identifiers are different, it is not possible to reduce the number of canonical edges based on occurrences comparisons.

▷ Proof

In order to reduce $ADirF$ when the vertex' identifiers are different, we have to be more restrictive on the occurrences constraints. To reduce the loops on k and k', we have to find an order between the occurrences of u and v.

However, for each order relation between the occurrences ($occ_u \leq occ_v$ and $occ_u \geq occ_v$), we can find two associated counter-examples. For the sake of clarity, we will find a counter-example for the case $occ_u \leq occ_v$. The reasoning is analogous for the second case.

Let a, b and c be three contigs in $\mathcal{C}$ such that all their identifiers are different, $mult_a = 2$, $mult_b = mult(c) = 1$, and $(a_f, b_f) \in \mathcal{L}$, $(b_f, a_f) \in \mathcal{L}$ and $(a_f, c_f) \in \mathcal{L}$.

Let $fa_1 = (i, j)$, $fa_2 = (i', j')$ be the two direct fragments associated to a and $fb = (k, l)$, $fc = (m, n)$ the two respectively associated to b and c. The idea behind the proof is to know if the adjacent direct fragments $fa_1\, fb\, fa_2\, fc$ can participate into a solution path $Path$. In such a case, $Path$ is sub-defined by edges $i \to k \to i' \to m$ (and $j \to l \to j' \to n$). Remark that k can permute with l, and m with n.

According to $ADirF$ set definition, $(i, k) \in ADirF$, $(k, i') \in ADirF$ and $(i', m) \in ADirF$. If we constrain the occurrences by the relation $occ_u \leq occ_v$, it implies that:

$$\begin{aligned} &occ_i \leq occ_k (= occ_l - 1) \\ &occ_l - 1 \leq occ_{i'} - 1 \\ &occ_{i'} \leq occ_m \end{aligned}$$

As $fa_1 \neq fa_2$, $occ_i \neq occ_{i'} - 1$:

if $occ_i < occ_{i'} - 1$ *then when we focus on the first edges path (in the "best" case n permutes with m) $occ_k (= 0) \leq occ_{i'} (= 2) \leq occ_n (= 1)$, so it is absurd;*

else $occ_i > occ_{i'} - 1$ *then when we focus on the first edges path, you can permute i' and i and you fall into the same absurd implication as above.*

◁

▶ Proposition 2.13: Occurences reduction for equal identifiers in $ADirF$

There exists an order between the occurrences of the vertices of an edge between two direct fragments that does not reduce the number of feasible and distinct solutions when the vertex' identifiers are equal.

▷ Proof

Let $u_1, u_2, \ldots, u_n$, $n = 2\lfloor \frac{mult_u}{2} \rfloor$ *be same identifiers vertices participating in a solution path for* $\mathcal{IRP}$, *in this order (i.e.* $\forall\, k \in [\![1, n[\![$, u_k *is before* u_{k+1} *in the solution path).*

It can be proven[a] *that: regardless the equation set of* $k-1$ *order relation* $R_{k,k+1} \in \{<, >\}$ *between two vertices* ctg_{uk} *and* ctg_{uk+1} *(i.e.* $ctg_{uk} R_{k,k+1} ctg_{uk+1}, \forall\, k \in [\![1, n[\![$*), there exists a permutation between all distinct occurrences of* u *such that all the relations* $R_{k,k+1}$ *are satisfied. Thus, let defined the following equation set (used in* $ADirF$*):*

$$R_{k,k+1} := \begin{cases} < & \textit{if } or_{uk} = 0 \vee or_{uk+1} = 0 \\ > & \textit{else } or_{uk} = or_{uk+1} = 1 \end{cases}$$

Have we got enough edges to build a path with $\frac{n}{2}$ *adjacent direct fragments? On the one hand, the maximum number of edges we would need is:* $2\big(\lfloor \frac{mult_u}{2} \rfloor - 1\big)$ *(the* 2 *is caused by direct fragments). On the other hand,* $ADirF$ *provides:*

$$2 \sum_{k'=1}^{\lfloor \frac{mult_u}{2} \rfloor - 1} 1 = 2\left(\left\lfloor \frac{mult_u}{2} \right\rfloor - 1\right)$$

edges, (the 2 *is caused by diradj function).*

◁

[a] With Lucas Robidou we are currently writing a paper and an associated algorithm

▶ **Proposition 2.14: Canonical reductions for** $ADirF$ **for equal identifiers**

$ADirF$ set contains only one of the eight edges that exist between two adjacent direct fragments when the vertex' identifiers are equal, i.e.

$\forall (i,j) \in DirF, \forall (k,l) \in DirF \mid ctg_i = ctg_k \wedge occ_i < occ_k$:

$$\begin{array}{l} (i,k) \in E \\ \iff (i,l) \in E \\ \iff (j,k) \in E \\ \iff (j,l) \in E \end{array} \iff \begin{array}{l} (i,k) \in ADirF \\ \wedge(\overline{k},\overline{i}) \notin ADirF \\ \wedge(i,l) \notin ADirF \\ \wedge(\overline{l},\overline{i}) \notin ADirF \\ \wedge(j,k) \notin ADirF \\ \wedge(\overline{k},\overline{j}) \notin ADirF \\ \wedge(j,l) \notin ADirF \\ \wedge(\overline{l},\overline{j}) \notin ADirF \end{array}$$

$$\begin{array}{l} (k,i) \in E \\ \iff (k,j) \in E \\ \iff (l,i) \in E \\ \iff (l,j) \in E \end{array} \iff \begin{array}{l} (k,i) \notin ADirF \\ \wedge(\overline{i},\overline{k}) \in ADirF \\ \wedge(k,j) \notin ADirF \\ \wedge(\overline{j},\overline{k}) \notin ADirF \\ \wedge(l,i) \notin ADirF \\ \wedge(\overline{i},\overline{l}) \notin ADirF \\ \wedge(l,j) \notin ADirF \\ \wedge(\overline{j},\overline{l}) \notin ADirF \end{array}$$

▷ Proof

The reciprocals are trivial as $ADirF \subset E$.

Only $(i,k) \in ADirF$ and $(\overline{i},\overline{k}) \in ADirF$ as the occurrences of i, $\overline{i}$, k or $\overline{k}$ are even, and $occ_i = occ_{\overline{i}} < occ_k = occ_{\overline{k}}$.

For the other cases, either they contradict the even-occurrences constraint or the constraint on the occurrences order.

◁

Reminder of adjacent inverted fragments set definition:

$$AInvF = \left\{ \begin{array}{l} (u,v) \in E \text{ s.t.} \\ \quad ctg_u < ctg_v \\ \quad \wedge\, occ_u = 2k + or_u \\ \qquad 0 \leq k < \left\lfloor \dfrac{mult_u}{2} \right\rfloor \\ \quad \wedge\, occ_v = 2k' + or_v \\ \qquad 0 \leq k' < \left\lfloor \dfrac{mult_v}{2} \right\rfloor \end{array} \right\} \bigcup \left\{ \begin{array}{l} (u,v) \in E \text{ s.t.} \\ \quad ctg_u = ctg_v \\ \quad \wedge\, (or_u = 0 \vee or_v = 0) \\ \quad \wedge\, occ_u - or_u = 2k \\ \quad \wedge\, occ_v - or_v = 2k' \\ \qquad 0 \leq k < k' < \left\lfloor \dfrac{mult_u}{2} \right\rfloor \end{array} \right\}$$

This set is the minimum exhaustive set of canonical edges which represent adjacent inverted fragments. Two combinatorial reductions are necessary — *canonical reduction for different identifiers* (Propositions 2.15 and 2.16) and *same identifiers occurrences* reductions (Propositions 2.17 and 2.18).

▶ Proposition 2.15: Canonical reductions for $AInvF$ for different identifiers

$AInvF$ set contains only one of the four edges that exist between two adjacent inverted fragments when the vertex' identifiers are different, i.e.
$\forall (i,j) \in InvF, \forall (k,l) \in InvF \mid ctg_i < ctg_k$:

$$(i,k) \in E \iff \begin{array}{l} (i,k) \in AInvF \\ \wedge(\bar{k},\bar{i}) \notin AInvF \\ \wedge(l,j) \notin AInvF \\ \wedge(\bar{j},\bar{l}) \notin AInvF \end{array}$$

$$(k,i) \in E \iff \begin{array}{l} (k,i) \notin AInvF \\ \wedge(\bar{i},\bar{k}) \notin AInvF \\ \wedge(j,l) \in AInvF \\ \wedge(\bar{l},\bar{j}) \in AInvF \end{array}$$

$$(i,l) \in E \iff \begin{array}{l} (i,l) \in AInvF \\ \wedge(k,j) \notin AInvF \\ \wedge(\bar{l},\bar{i}) \notin AInvF \\ \wedge(\bar{j},\bar{k}) \notin AInvF \end{array}$$

$$(j,k) \in E \iff \begin{array}{l}(j,k) \in AInvF \\ \wedge(\overline{k},\overline{j}) \notin AInvF \\ \wedge(l,i) \notin AInvF \\ \wedge(\overline{i},\overline{l}) \notin AInvF\end{array}$$

▷ Proof

The reciprocals are trivial as $AInvF \subset E$.

Let $(i,j) \in InvF$, $(k,l) \in InvF$ be two adjacent inverted fragments such that $ctg_i < ctg_k$. We will focus on demonstrating the case $(i,k) \in E$. The others follow the same logics.

— $(i,k) \in AInvF$ *as* $\exists\, n \in [\![0, \lfloor \frac{mult_i}{2} \rfloor[\![$ *such that* $occ_i = 2n + or_i = 2n$, *and* $\exists\, n' \in [\![0, \lfloor \frac{mult_k}{2} \rfloor[\![$ *such that* $occ_k = 2n' + or_k = 2n' + 1$;

— $(\overline{k},\overline{i}) \notin AInvF$ *as* $ctg_{\overline{k}} > ctg_{\overline{i}}$;

— $(l,j) \notin AInvF$ *as* $ctg_l > ctg_j$;

— $(\overline{j},\overline{l}) \notin AInvF$ *as* $or_{\overline{j}} = 0$ *and* $occ_{\overline{j}} \nmid 2$ *so* $\nexists n \in [\![0, \left\lfloor \frac{mult_j}{2} \right\rfloor[\![$ *such that* $occ_{\overline{j}} = 2n$.

◁

▶ Proposition 2.16: Minimum $AInvF$ set for different identifiers

When the vertex' identifiers are different, it is not possible to reduce the number of canonical edges based on occurrences comparisons.

▷ Proof

In order to reduce $AInvF$ when the vertex' identifiers are different, we have to be more restrictive on the occurrences constraints. To reduce the loops on k and k', we have to find an order between the occurrences of u and v.

However, for each order relation between the occurrences ($occ_u \leq occ_v$ and $occ_u \geq occ_v$), we can find two associated counter-examples. For the sake of clarity, we will find a counter-example for the case $occ_u \leq occ_v$. The reasoning is analogous for the second case.

Let c and d be two contigs in $\mathcal{C}$ such that $ctg_c < ctg_d$, $mult(c) = 4$ and

$mult_d = 2$, and $(c_f, d_f) \in \mathcal{L}$ and $(d_f, c_f) \in \mathcal{L}$. Let $p_1 = (i, j)$, $p_2 = (i', j')$ be the two inverted fragments associated to c and $q = (k, l)$ the one associated to d. The idea behind the proof is to know if the adjacent inverted fragments $p_1\, q\, p_2$ can participate into a solution path $Path$. In such a case, $Path$ is sub-defined by edges $i \to k \to i'$ and $j' \to l \to j$.

According to $AInvF$ set definition, $(i, k) \in AInvF$ and $(j', l) \in AInvF$. If we constrain the occurrences by the relation $occ_u \leq occ_v$, thus on the one hand, $occ_i \leq occ_k (= occ_l - 1)$, and on the other hand $occ_l - 1 \geq occ_{j'} - 1$. As $p_1 \neq p_2$, $occ_i \neq occ_{j'} - 1$:

if $occ_i < occ_{j'} - 1$ *then* $occ_k(= 0) > occ_i(= 0)$ *so it is absurd;*

else $occ_i > occ_{j'} - 1$ *then* $occ_i(= 2) \leq occ_k(= 0)$ *so it is absurd.*

◁

▶ Proposition 2.17: Occurences reduction for equal identifiers in $AInvF$

There exists an order between the occurrences of the vertices of an edge between two inverted fragments that does not reduce the number of feasible and distinct solutions when the vertex' identifiers are equal.

▷ Proof

Let $u_1, u_2, \ldots, u_n$, $n = 2\lfloor \frac{mult_u}{2} \rfloor$ be same identifiers vertices participating in a solution path for $\mathcal{IRP}$, in this order (i.e. $\forall\, k \in [\![1, n[\![$, u_k is before u_{k+1} in the solution path).

It can be proven[a] that: regardless the equation set of $k - 1$ order relation $R_{k,k+1} \in \{<, >\}$ between two vertices ctg_{uk} and ctg_{uk+1} (i.e. $ctg_{uk} R_{k,k+1} ctg_{uk+1}, \forall\, k \in [\![1, n[\![$), there exists a permutation between all distinct occurrences of u such that all the relations $R_{k,k+1}$ are satisfied. Thus, let define the following equation set (used in $AInvF$):

$$R_{k,k+1} := \begin{cases} < & \text{if } or_{uk} = 0 \vee or_{uk+1} = 0 \\ > & \text{else } or_{uk} = or_{uk+1} = 1 \end{cases}$$

Have we got enough edges to build a path with $\frac{n}{2}$ adjacent inverted fragments? On the one hand, the maximum number of edges we would need is: $2\left(\lfloor \frac{mult_u}{2} \rfloor - 1\right)$ (the 2 is caused by inverted fragments). On the other

hand, $AInvF$ provides:

$$2\sum_{k'=1}^{\left\lfloor\frac{mult_u}{2}\right\rfloor-1} 1 = 2\left(\left\lfloor\frac{mult_u}{2}\right\rfloor - 1\right)$$

edges, (the 2 is caused by invadj function).

◁

[a]With Lucas Robidou we are currently writing a paper and an associated algorithm

▶ Proposition 2.18: Canonical reductions for $AInvF$ for equal identifiers

$AInvF$ set contains only one of the four edges that exist between two adjacent inverted fragments when the vertex' identifiers are different, i.e.
$\forall\,(i,j) \in InvF, \forall\,(k,l) \in InvF \mid ctg_i = ctg_k \wedge occ_i < occ_k$:

$$(i,k) \in E \iff \begin{array}{l}(i,k) \in AInvF\\ \wedge(\overline{k},\overline{i}) \notin AInvF\\ \wedge(l,j) \notin AInvF\\ \wedge(\overline{j},\overline{l}) \notin AInvF\end{array}$$

$$(i,l) \in E \iff \begin{array}{l}(i,l) \in AInvF\\ \wedge(k,j) \notin AInvF\\ \wedge(\overline{l},\overline{i}) \notin AInvF\\ \wedge(\overline{j},\overline{k}) \notin AInvF\end{array}$$

$$(j,k) \in E \iff \begin{array}{l}(j,k) \in AInvF\\ \wedge(\overline{k},\overline{j}) \notin AInvF\\ \wedge(l,i) \notin AInvF\\ \wedge(\overline{i},\overline{l}) \notin AInvF\end{array}$$

Nota bene $(k,i) \in E$ case is not here because (i,k) does not change the solution as their sequences are equal. This non-redundancy advantage is allowed by occurrence reduction in Proposition 2.17.

▷ Proof

The reciprocals are trivial as $AInvF \subset E$.
Let $(i,j) \in InvF$, $(k,l) \in InvF$ be two adjacent inverted fragments such

that $ctg_i = ctg_k$ *and* $occ_i < occ_k$. *We will focus on demonstrating the case* $(i,k) \in E$. *The others follow the same logics.*

— $(i,k) \in AInvF$ *as* $\exists\, n \in [\![0, \lfloor \frac{mult_i}{2} \rfloor[\![$ *such that* $occ_i = 2n + or_i = 2n$, *and* $\exists\, n' \in [\![0, \lfloor \frac{mult_k}{2} \rfloor[\![$ *such that* $occ_k = 2n' + or_k = 2n' + 1$;

— $(\overline{k}, \overline{i}) \notin AInvF$ *as* $occ_{\overline{k}} > occ_{\overline{i}}$;

— $(l, j) \notin AInvF$ *as* $occ_l > occ_j$;

— $(\overline{j}, \overline{l}) \notin AInvF$ *as* $or_{\overline{j}} = 0$ *and* $occ_{\overline{j}} \nmid 2$ *so* $\nexists n \in [\![0, \left\lfloor \frac{mult_j}{2} \right\rfloor[\![$ *such that* $occ_{\overline{j}} = 2n$.

◁

3 Metrics

3.1 Quast metrics

`Quast` metric descriptions can be found at https://quast.sourceforge.net/docs/manual.html#sec3.1. Here we adapt the description of the ones we use in this paper.

 # misassemblies is the number of positions in the contigs (breakpoints) that satisfy one of the following criteria:

— the left flanking sequence aligns over 1 kbp away from the right flanking sequence on the reference;

— flanking sequences overlap on more than 1 kbp;

— flanking sequences align to different strands.

 # local misassemblies is the number of positions in the contigs (breakpoints) that satisfy the following conditions:

— the gap or overlap between left and right flanking sequences is less than 1 kbp, and larger than 200 bp (the maximum indel length);

— the left and right flanking sequences both are on the same strand.

Genome fraction (%) is the percentage of aligned bases in the reference genome. A base in the reference genome is aligned if there is at least one contig with at

least one alignment to this base. Contigs from repetitive regions may map to multiple places, and thus may be counted multiple times.

4 Supplementary results

4.1 v1 scaffolding benchmark

Table 18 – Benchmark 3 v1 contig `Quast`

Instance	ILPs	%gnm	#mis	
Abies_alba	dr-sc	99.32	0	0
	ir-sc	99.32	0	0
Acorus_americanus	ir-sc	99.52	0	0
Agathis_dammara	dr-ir-sc	80.19	0	0
	ir-dr-sc	80.19	0	0
Azima_tetracantha	ir-sc	99.93	0	0
Begonia_pulchrifolia	—	—	—	—
Carpodetus_serratus	ir-sc	98.61	0	0
Circaeaster_agrestis	ir-sc	99.56	0	0
Clematis_repens	ir-sc	99.93	0	0
Commiphora_foliacea	ir-sc	98.34	0	0
Cucumis_hystrix	ir-sc	99.49	0	0
Eucommia_ulmoides	ir-sc	99.01	0	0
Jasminum_tortuosum	ir-sc	99.77	0	0
Juniperus_scopulorum	sc	98.82	0	0
Lamprocapnos_spectabilis	dr-sc	35.11	0	0
Lathyrus_pubescens	dr-sc	98.20	0	0
	ir-sc	98.20	0	0
Lophocereus_schottii	sc	98.63	0	0
Musa_ornata	ir-sc	99.08	0	0
Oenothera_glazioviana	ir-sc	98.50	0	0
Pelargonium_nanum	ir-sc	96.04	0	0
Podocarpus_totara	sc	98.76	0	0
Porphyra_purpura	dr-sc	99.95	0	0
Sagittaria_trifolia	ir-sc	98.55	0	0
Sciadopitys_verticillata	dr-sc	96.24	0	0
	ir-sc	96.24	0	0
Sciaphila_densiflora	sc	99.87	0	0
Selaginella_kraussiana	dr-sc	99.84	0	0
Selaginella_vardei	dr-sc	99.98	0	0
Taxus_baccata	sc	98.07	0	0
Triosteum_pinnatifidum	ir-dr-sc	98.30	0	0
Uvaria_macrophylla	ir-sc	98.35	0	0

(the table continues on the next page)

Table 18, continued

Instance	ILPs	%gnm	#mis	
Welwitschia_mirabilis	`ir-sc`	99.17	0	0
Wolffia_australiana	`ir-sc`	99.83	0	0

Table 19 – Benchmark 3 v1 ILP stats

Instance	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{L}\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	Time
Abies_alba	9	28	20	36	0.02
					0.02
Acorus_americanus	5	16	16	40	0.04
					0.05
Agathis_dammara	20	54	50	96	0.20
					0.64
Azima_tetracantha	7	20	16	28	0.00
					0.00
Begonia_pulchrifolia	6	14	12	14	0.00
					0.00
Carpodetus_serratus	13	44	36	76	0.14
					0.21
Circaeaster_agrestis	13	36	44	112	0.37
					4.06
Clematis_repens	6	16	18	38	0.04
					0.04
Commiphora_foliacea	16	38	38	58	0.04
					0.04
Cucumis_hystrix	9	22	22	40	0.02
					0.03
Eucommia_ulmoides	13	32	32	52	0.03
					0.05
Jasminum_tortuosum	10	26	30	68	0.07
					0.13
Juniperus_scopulorum	10	28	20	28	0.00
					0.00
Lamprocapnos_spectabilis	12	36	26	46	0.00
					0.00
Lathyrus_pubescens	17	44	36	52	0.02
					0.02
Lophocereus_schottii	12	36	24	36	0.00
					0.00
Musa_ornata	15	32	46	82	0.24
					0.87
Oenothera_glazioviana	11	30	30	56	0.07
					0.08
Pelargonium_nanum	11	32	34	108	0.17

(the table continues on the next page)

Table 19, continued

Instance	$\|\mathcal{C}\|$	$\|\mathcal{L}\|$	$\|V\|$	$\|E\|$	Time
					0.43
Podocarpus_totara	13	42	26	42	0.00
					0.03
Porphyra_purpura	3	8	8	16	0.00
					0.00
Sagittaria_trifolia	15	32	34	46	0.02
					0.02
Sciadopitys_verticillata	25	70	52	78	0.02
					0.04
Sciaphila_densiflora	4	8	8	8	0.00
					0.00
Selaginella_kraussiana	5	12	14	26	0.01
					0.01
Selaginella_vardei	3	8	8	16	0.00
					0.00
Taxus_baccata	17	42	34	42	0.00
					0.00
Triosteum_pinnatifidum	13	40	32	66	0.07
					0.15
Uvaria_macrophylla	13	30	42	86	0.24
					0.84
Welwitschia_mirabilis	13	34	30	48	0.02
					0.03
Wolffia_australiana	10	28	24	44	0.02
					0.02

4.2 v2 scaffolding benchmark

Table 20 – Benchmark 3 v2 ILP stats

Instance	$\|\mathcal{C}\|$	$\|\mathcal{L}\|$	$\|V\|$	$\|E\|$	Time
Agathis_dammara	20	54	52	108	0.22
					0.83
Begonia_pulchrifolia	6	14	14	22	0.00
					0.00
Carpodetus_serratus	13	42	36	74	0.13
					0.20
Jasminum_tortuosum	10	28	30	70	0.08
					0.13
Lamprocapnos_spectabilis	12	36	48	180	0.68
					2.67

(the table continues on the next page)

Table 20, continued

Instance	$\|\mathcal{C}\|$	$\|\mathcal{L}\|$	$\|V\|$	$\|E\|$	Time
Lathyrus_pubescens	17	44	36	52	0.02
					0.02
Lophocereus_schottii	12	34	24	34	0.00
					0.00
Pelargonium_nanum	11	32	36	128	0.33
					0.84
Podocarpus_totara	13	40	26	40	0.00
					0.00
Triosteum_pinnatifidum	13	40	32	66	0.07
					0.18

ACRONYMS

A | B | C | D | F | I | L | M | N | O | P | S | T

A

ALE Assembly Likelihood Estimation 55

B

BAC Bacterial Artificial Chromosome 9, 11, 17

BFS Breadth-First Search 20, 50

BOG Best Overlap Graph 17, 18

C

CSC Compressed Sparse Column 58

CSR Compressed Sparse Row 58

D

DBG De Bruijn Graph 18–21, 38, 53

DNA Deoxyribonucleic acid 2–4, 6–9, 11–14, 22, 24, 51, 54

DR Direct Repeat 6, 7, 88, 91–93, 111

F

FIFO First In First Out 111

I

ILP Integer Linear Programming 48, 89, 90, 131, 133, 136

IR Inverted Repeat 6–8, 88, 91–93, 111, 116, 117, 136

L

LIFO Last In First Out 111, 117

LSC Long Single-Copy 6, 7, 88, 93

M

MILP Mixed-Integer Linear Programming 48, 50

MIQP Mixed-Integer Quadratic Programming 51, 54

MWA2M Maximum-Weight Acyclic 2-Matching 48, 49

N

NGS Next Generation Sequencing technologies 10, 18

O

OLC Overlap-Layout-Consensus 17–22, 24, 43, 86, 134

ONT Oxford Nanopore Technology 12, 15

P

PCR Polymerase Chain Reaction 10, 11

S

S&E Seed-and-Extend 49, 52, 54

SBH Sequencing-by-Hybridisation 18

SBS Sequencing By Synthesis 9–11

SC Single-Copy 91, 92

SCS Shortest Common Superstring 15–17, 20, 48

SMRT Single Molecule Run Time technologies 11

SSC Short Single-Copy 6, 7, 88

T

TGS Third Generation Sequencing technologies 11, 17

SYMBOLS

Fragment | Graph | Common sets

Fragment

$\mathcal{C}$ Set of contigs 43–45, 48–50, 90–92, 97, 98, 102, 117, A21

f Forward orientation 25, 28, 29, 31, 35, 37, 59, 64, 65, 68, 73, 75, 77, 79–82, *see also* forward & $\{f, r\}$

$\mathcal{F}_f$ Set of forward fragments 25, 59–61, 63–65, 69

$\mathcal{F}_r$ Set of reverse fragments 25, 59, 61, 63, 65, 69

$\mathcal{F}$ Set of fragments 25–28, 30, 31, 34, 36, 58, 59, 61, 63, 65–67, 70, 82, 84

$\mathcal{L}_f$ (Multi)set of forward links 59, *see also* $\mathcal{L}$ & $\mathcal{L}_r$

$\mathcal{L}'$ Underlying set of the multiset $\mathcal{L}$ 26–28, 31, 35, 37, *see also* $\mathcal{L}$ & $\mathcal{F}$

$\mathcal{L}_r$ (Multi)set of reverse links 59, *see also* $\mathcal{L}$ & $\mathcal{L}_f$

$\mathcal{L}$ (Multi)set of links 26–28, 30–32, 34, 36, 43, 44, 48–50, 58, 59, 61, 63, 65–68, 70, 82, 90–92, 97, 101, 117, 119, 124, 125, A19, G3, *see also* $\mathcal{L}'$ & $\mathcal{F}$

$\{f, r\}$ Set of orientations 25, 26, 31, 44, 64, 66, 73–75, 91, 97, 98, 110, 115, 120, A3

r Reverse orientation 25, 28, 29, 31, 35, 37, 64, 65, 68, 77, 80, *see also* reverse & $\{f, r\}$

$\mathcal{R}_{aw}$ Set of raw reads 15

$\overline{\cdot}$ Reverse operation 25–27, 29, 32, 35, 44, 45, 55, 59, 61, 63, 65, 68, 69, 72, 73, 76, 78, 79, 81, 91, 92, 101–104, 106, 108–110, 115, 116, 122, 127, A17, *see also* reverse

$\Sigma_{\mathcal{F}}$ Fragment label set 25, 59, 60, 62, 68

$\Sigma_{\mathcal{F}_{un}}$ Unoriented fragment label set 25, 59, 64, 66

$\Sigma_{\mathcal{L}}$ Link label set 26

$\Sigma_{\mathcal{L}_{can}}$ Canonical link label set 59, 60, 62, 64, 66, 68

$\mathcal{F}_{un}$ Set of unoriented fragments 25, 31

Graph

G Generic graph *see also* undirected graph, directed graph, bidirected graph, vertex & edge

DG Directed fragment multigraph, $DG = (V, E, \Phi)$ 27–29, 38, 39, 48, 55, 59, 97, *see also* directed graph

BG Bidirected fragment multigraph, $BG = (V, E, \Phi, attre)$ 30–33, 38, 39, 44, 49, 50, 66, *see also* bidirected graph

UG Undirected fragment multigraph, $UG = (V, E_{\mathcal{F}}, E_{\mathcal{L}}, \Phi_{\mathcal{L}})$ 33–36, 38, 39, 44, 46–51, 68, *see also* undirected graph

V Vertex set 28–32, 34, 35, 46, 59, 60, 62, 64–66, 68, 69, 72, 76, 78–80, 97, G4, *see also* vertex

V_f Forward vertex set 64, *see also* vertex & forward

E Edge set 28–32, 46, 59–66, 97, G4, *see also* edge

$E_{\mathcal{F}}$ Fragment-edge set 34–36, 68, 69, G4, *see also* edge, fragment & UG

$E_{\mathcal{L}}$ Link-edge set 34–36, 68, G4, *see also* edge, link & UG

Φ Incidence function 28–32, 59, 61, 63, 65, 66, 114–116, G5, *see also* edge

$\Phi_{\mathcal{L}}$ Incidence function for link-edges 34–36, 68, G4, *see also* edge, link & UG

N Neighbour set 59, 66, 72, 74–77, 80–82, *see also* edge

N^- Predecesor set 60, 62, 64, 68, 72, 73, 75–77, 79–82, *see also* edge, N & N^+

N^+ Successor set 60, 62, 64, 68, 72–82, *see also* edge, N & N^-

$N^{\mathcal{F}}$ Fragment extremity neighbour set *see also* edge, N & UGA

$N^{\mathcal{L}}$ Link neighbour set 68, 69, *see also* edge, N & UGA

Σ_V Vertex label set 60, 62, 64, 66, 68, 70, 73–75, *see also* vertex

Σ_{V_f} Forward vertex label set 64, *see also* vertex

Σ_E Edge label set 60, 62, 64, 66, 68, 73–75, *see also* edge

DGA All oriented fragments multidigraph 59, 61, 63, *see also* DG & directed multigraph

DGS Oriented fragments' successors multidigraph 61, 63, 68, 70, 72, 84, 86, 134, 135, *see also* DG & directed multigraph

DGF Forward fragments multidigraph 63, 65, 66, 68, 72, 84, 86, 134, 135, *see also* DG & directed multigraph

BGU Unoriented fragments multibigraph 66, 68, 72, 84, 86, 134, 135, *see also* BG & bidirected multigraph

UGA All fragments' extremities undirected graph multiungraph 68, 70, *see also* UG & undirected multigraph

$MDCG$ Multiplied doubled contig graph, $MDCG = (V, E, vwex)$ 97, 98, 102, 103, 113, 116, 131, 136, *see also* DG, directed graph, contig & link

$RegGraph$ Region graph, $RegGraph = (Vreg, Ereg, \Phi)$ 114, 116, *see also* DG & directed multigraph

$Vreg$ Set of oriented region 114–116, G5, *see also* $RegGraph$ & V

$Ereg$ Multiset of links between two oriented regions 114–116, G5, *see also* $RegGraph$ & E

Common sets

$\{0, 1\}$ Set of binaries 104, 107

$\mathbb{N}$ The set of natural (positive) integer 26, 29, 32, 35, 59, 90, 97, 110, 117–119

$\mathbb{R}$ The set of real numbers 90, 97, 103, 104, 106, 109

Σ_{nuc} Nucleotide alphabet set $\Sigma_{nuc} = \{A, C, G, T\}$ 2, 25, 90

COMPUTATIONAL TERMS

Complexity | **Database** | **File formats** | **Programs and packages**

Complexity

$\mathcal{NP}$ Nondeterministic polynomial time 117

$\mathcal{NP}$-complete Nondeterministic polynomial-time complete 20, 22, 45, 47, 48, 90, 119, 121–123, 131, 136

$\mathcal{NP}$-hard Nondeterministic polynomial-time hardness 16, 18, 46, 49, 119, 121

$\mathcal{P}$ Contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time 20, 48, 131, 136

Database

CpGDB Chloroplast Genome Database 123

NCBI National Center for Biotechnology Information 123

PyPI Python Package Index vii, viii, 86, 135

File formats

`GFA` Graphical Fragment Assembly (GFA) Format 135

`PAF` Pairwise mApping Format 135

Programs and packages

`ARACHNE` A whole-genome shotgun assembler 17

`ARKS` Alignment-free linked read genome scaffolding methodology 39

`Bambus` General-purpose scaffolder based on paired-reads 30, 39, 45

`Bambus 2` Metagenome scaffolder 30, 39, 45

`BESST` Scaffolder based on paired-end data 33, 39

`BOSS` Scaffolder based on paired-end or mate-pair read set 30, 39

`CABOG` Celera Assembler with the Best Overlap Graph 18

`Canu` Canu is a fork of the Celera Assembler, designed for high-noise single-molecule sequencing 17, 18

`CBC` Open-source mixed integer linear solver 121

`Celera` A whole genome assembler originally developed at Celera Genomics for the assembly of the human genome 17

`chloroExtractor` Pipeline for DNA extraction of chloroplast DNA from whole genome plant data 52

`Chloroplast assembly protocol` A set of scripts for the assembly of chloroplast genomes out of whole-genome sequencing reads 52

`Edena` *De novo* assembler for Illumina reads 30

`FALCON` Diploid aware genome assembler designed for Pacific Biosciences long read data 28

`Fast-Plast` Chloroplast short-read assembly pipeline 52

`Flye` *De novo* assembler for single-molecule sequencing reads, such as those produced by PacBio and Oxford Nanopore Technologies 20

`fragScaff` Scaffolder based on linked-read data 33, 39

`GapFiller` Paired-read based gap filler 54

`GAT` Scaffolder based on paired-reads and global optimisation method 28, 39, 52

`GetOrganelle` Organelle genome assembly toolkit x, 52, 89, 132, 137

`GRASS` Generic short-read scaffolder based on paired-reads or on a reference genome 30, 39

`Gurobi` Linear, integer and non-linear solver 121, 122

`hifiasm` Haplotype-resolved *de novo* assembler initially designed for PacBio HiFi reads 18, 28

`HINGE` OLC long read assembler based on an idea called *hinging* 28

`IOGA` Iterative organellar genome assembly 52

`Jellyfish` k-mer counting tool 52

`khloraascaf` Chloroplast contig scaffolder aware of structural haplotypes based on several integer linear programs viii, 121–123, 126, 127, 129–132, 136, 137

`LACHESIS` Chromosome-scale scaffolder based on chromatin interactions 30, 39

`LRScaf` Hybrid scaffolder based on long reads 33, 39

`mdBG` Minimizer-space de Bruijn graphs for whole-genome assembly 20

`Minia` Short-read assembler based on a de Bruijn graph 124

`Minimus` Assembly pipelines designed specifically for small data-sets, such as the set of reads covering a specific gene 30

`MIP` Paired-read scaffolder based on mixed integer programming 30, 39

`NOVOPlasty` *De novo* assembler and heteroplasmy/variance caller for short circular genomes 52

`Opera` Paired-end read scaffolder based on the graph bandwidth problem 30, 39

`ORG.Asm` Organellar read assembler 52

`Quast` Quality assessment tool for genome assemblies 126, 127, 130, A17

`revsymg` Python implementation of DNA fragment graph vii, 86, 135

`SALSA2` Hi-C scaffolder 33, 39

`ScaffMatch` Short read scaffolding tool based on Maximum-Weight Matching 33, 39

`ScaffoldScaffolder` Scaffolder based on a bidirected to directed graph reduction 30

`Scaftools` Integer linear programming approach for genome scaffolding 33, 39

`SCARPA` Scaffolder which combines fixed-parameter tractable and bounded algorithms with linear programming 33, 39

`Shasta` *De novo* assembler for long reads, optimized for Oxford Nanopore (ONT) reads 28

`SLIQ` Simple linear inequalities based mate-pair reads filtering and scaffolding 39

`SLR` Hybrid scaffolder based on long reads 30, 39

`SOAPdenovo` Short-read assembly method that can build a *de novo* draft assembly for the human-sized genomes 20, G10

`SOAPdenovo2` Successor of `SOAPdenovo` 53

`SOPRA` Scaffolding algorithm for paired reads via statistical optimisation 30, 39

`SPAdes` St. Petersburg genome assembler — is an assembly toolkit containing various assembly pipelines 20, 54

`SSPACE` Paired-read scaffolder 30, 39, 54

`Velvet` Short read *de novo* assembler using de Bruijn graphs 20, 54

`wtdbg2` *De novo* sequence assembler for long noisy reads produced by PacBio or Oxford Nanopore Technologies 20

GLOSSARY

A | B | C | D | E | F | G | K | L | M | N | O | P | R | S | U | V | W

A

alignment Nucleotide comparison between at least two sequences

assembly graph One of the output of genome assembly method *see also* read assembly

B

bidirected graph Graph that contains three edge types 30, 45, 86, 134

bidirected multigraph A graph where the same bioriented edge between the same two vertices can occur several times 30, 31, 86, 134, *see also* multigraph & bidirected graph

C

chloroplast Organelle in the plants'cells specialised in the photosynthesis process 2, 4–8, 22, 51, 90, 93, 133

chromosome A chromosome is a DNA molecule that contains the genetic information for an organism 4, 13, 21, 24, 133, *see also* genome & DNA

contig Result of the assembly of the reads, nucleotide sequence longer than the reads 21, 22, 37, 38, 46–48, 50, 54, 86, 90, 130, 133, 134, 136, *see also* read assembly & scaffolding

D

de Bruijn graph Graph structure where the vertices are word of length k, and edges are overlaps of length $k-1$. Can be vertex-centric or edge-centric. 18–21, 24, 53, *see also* k-mer

directed graph Graph where the edges are directed 16, 17, 27, 86, 89, 90, 97, 134

directed multigraph A graph where the same oriented edge between the same two vertices can occur several times 28, 29, 59, 86, 97, 114, 134, *see also* multigraph & directed graph

E

edge Component of a graph that connects two vertices 17, 22, 28–31, 33–35, 46, 48, 49, 58–67, 69, 70, 72, 76–82, 86, 134, 135, *see also* graph & vertex

F

forward Original nucleotide sequence orientation 14, 25, 33, 34, 59, 62–65, 73, 75, 76, 90, *see also* reverse

fragment Generic nucleotide sequence 2, 8–10, 12–14, 22, 24–28, 30, 31, 33–35, 58, 59, 61–63, 65–70, 73–77, 81, 82, 85, 86, 133, 134

fragment assembly Generic term for the DNA fragment assembly, can also denote the whole process of assembling the reads to obtain the whole DNA sequences 8, 13, 15, 16, 22, 37, 51, 133, *see also* read assembly & scaffolding

fragment-edge Component of a graph that connects two vertices 33–35, 68–70, 86, 134, *see also* graph & vertex

fragment-neighbour Vertex connected to a given one forming a fragment-edge 68, *see also* neighbour, fragment-edge & UG

G

genome Entire set of DNA instructions found in a cell 2–10, 13–17, 22, 90, 133, *see also* DNA

graph Object composed of vertices connected by edges 22, 28, 58, 70, 72, 85, 86, 133–135, *see also* vertex & edge

K

k-mer Word of lenght k 18–21, 52–54, 124, *see also* read assembly, SBH, DBG & de Bruijn graph

L

link Ordered pair of oriented fragment 22, 28, 30, 33, 34, 38, 58–70, 73, 74, 86, 91, 134, *see also* fragment

link-edge Component of a graph that connects two vertices 33–35, 46, 48, 49, 68, 70, 86, 134, *see also* graph & vertex

link-neighbour Vertex connected to a given one forming a link-edge 68–70, *see also* neighbour, link-edge & UG

M

mitochondrion Organelle found in the cells of most eukaryotes specialised in the aerobic respiration 4, 5, 8

multigraph A graph where there can be several edges between the same two vertices, *see also* graph

N

neighbour Vertex connected to a given one 58, 61, 63, 65–68, 70, 72, 78–80, 86, 134

nucleotide Basic building block of nucleic acids (RNA and DNA) 2, 3, 6, 8–12, 16, 38, *see also* DNA

O

organelle Specialised subunit, usually within a cell, that has a specific function 4, 5, 8

overlap Suffix-prefix alignment type between two sequences 9, 13, 14, 16–18, 48, 86, 133, 134, *see also* alignment & link

overlap graph Graph structure that stores overlaps between reads 17, 22, *see also* overlap

P

path A sequence of vertices such that two consecutive vertices are connected by an edge in the graph 17, 29, 32, 33, 35, 36, *see also* walk, graph, vertex & edge

predecessor In-vertex of one edge of one given vertex 59–62, 64, 65, 68, 72–75, *see also* successor

R

read DNA fragment from one DNA's strand, output by a sequencer 9–18, 20, 21, 24, 37, 38, 51–53, 58, 86, 133, 134, *see also* sequencer

read assembly Fragment assembly stage to obtain longer nucleotide sequence (contigs) from the reads 22, 28, 33, 38, 47, 48, 53, 133, *see also* fragment assembly & contig

reverse Reverse-complement nucleotide sequence orientation 2, 3, 14, 25–31, 33, 34, 59–64, 66, 67, 69, 73, 75, 78–81, 86, 90, 134, *see also* forward

S

scaffold Sequence of oriented contigs separated by gap 21, 37, 133, *see also* scaffolding

scaffolding Orienting and ordering the contigs 22, 28, 33, 37, 38, 86, 133, 134, *see also* fragment assembly, read assembly, contig & scaffold

sequencer Sequencing technology machine 8

sequencing Method to generate nucleotide fragments 2, 8–11, 13, 15, 24, 51

strand The DNA molecule is made up of two strands, each of which has a complementary sequence to the other 2, 3, 8–10, 12

string graph Graph structure where vertices are genomic regions, and edges overlaps between them 17, *see also* overlap graph & overlap

structural haplotype Copies of the genome with different structures 7, 22, 51, 55, 136

successor Out-vertex of one edge of one given vertex 59–65, 68, 72, 74, 75, *see also* predecessor

U

undirected graph Graph where the edges are undirected 33, 35, 45, 86, 134

undirected multigraph A unoriented graph where there can be several edges between the same two vertices 31, 34, 66, 68, *see also* multigraph & directed graph

unitig Special case of a contig, it represents a non-ambiguous assembled sequence 18, 19, 21, 28, 48, 54, *see also* contig, read assembly & scaffolding

V

vertex Component of a graph 17, 22, 28–31, 33, 34, 36, 46, 49, 51, 58–70, 72–82, 86, 134, 135, *see also* graph & edge

W

walk A sequence of vertices such that two consecutive vertices are connected by an edge in the graph 35, *see also* path, graph, vertex & edge

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Assemblage de fragments ADN : structures de graphes et échafaudage de génomes de chloroplastes

Mot clés : Assemblage de génomes, Programmation linéaire en nombres entiers, Répétitions génomiques, Haplotypes structuraux

Résumé : L'obtention de la séquence nucléotidique d'une molécule ADN nécessite sa fragmentation par des technologies de séquençage et l'assemblage des fragments. Ces fragments sont appelés lectures. Elles souffrent d'erreurs de séquençage et sont considérées sous deux orientations : celle de leur brin ADN d'origine ou l'inverse-complémentaire pour l'autre brin. L'assemblage se base sur des chevauchements deux à deux entre des lectures orientées, et est composé de trois phases : l'assemblage des lectures pour obtenir des contigs (des séquences plus longues que les lectures), l'échafaudage des contigs, pour obtenir des échafaudages (des ordres de contigs orientés), et la complétion des échafaudages (trouver les séquences de nucléotides séparant les contigs orientés dans les échafaudages).

Dans ce manuscrit, nous comparons des structures de graphes représentant des relations de successions entre des séquences ADN orientées, utiles à différentes phases de l'assemblage. Puis, nous nous penchons sur le problème de l'échafaudage dédié aux génomes de chloroplastes en proposant une nouvelle formulation, une résolution exacte et une implémentation.

Title: DNA fragment assembly: graph structures and chloroplast genome scaffolding

Keywords: Genome assembly, Integer linear programming, Genome repeats, Structural haplotypes

Abstract: To obtain the nucleotide sequence of a DNA molecule, the molecule is fragmented using a sequencing technology and the fragments are assembled. These fragments are called reads. They are subject to sequencing errors and must be considered in two orientations: that of their original DNA strand, or the reverse-complementary for the other strand. Assembly is based on pairwise overlaps between oriented reads and consists in three phases: assembling the reads to obtain contigs (sequences longer than the reads), scaffolding the contigs to obtain scaffolds (orders of oriented contigs), and completing the scaffolds (finding the nucleotide sequences separating the oriented contigs in the scaffolds).

In this manuscript, we compare graph structures representing succession relations between oriented DNA sequences, useful at different phases of assembly. Then, we address the scaffolding problem dedicated to chloroplast genomes by proposing a new formulation, an exact resolution and an implementation.