Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Identifying Clusters in Graph Representations of Genomes

Master's Thesis

2023
Bc. Eva Herencsárová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Identifying Clusters in Graph Representations of Genomes

Master's Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. Mgr. Bronislava Brejová, PhD.

Bratislava, 2023
Bc. Eva Herencsárová

90519009

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Eva Herencsárová
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Identifying Clusters in Graph Representations of Genomes
Identifikácia zhlukov v grafovej reprezentácii genómov

Anotácia: V mnohých bioinformatických aplikáciách identifikujeme v genóme významné miesta a následne chceme nájsť zhluky s vysokou hustotou takýchto významných miest. Miesta môžu zodpovedať napríklad mutáciám alebo oblastiam s určitou biologickou funkciou. Cieľom tejto práce je preštudovať existujúce metódy na hľadanie hustých zhlukov v lineárnych sekvenciách a rozšíriť ich na grafy, ktoré sa používajú na reprezentáciu skupín príbuzných genómov. Vrcholy týchto grafov zodpovedajú častiam sekvencií a hrany susednostiam týchto častí.

Vedúci:	doc. Mgr. Bronislava Brejová, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	13.12.2021
Dátum schválenia:	03.01.2022

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

....

študent vedúci práce

90519009

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Eva Herencsárová
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Identifying Clusters in Graph Representations of Genomes

Annotation: In many bioinformatics applications, we identify significant locations in an individual genome, and we are interested in finding clusters with high density of such significant locations. The locations may represent for example mutations or positions with a particular biological role. The goal of this thesis is to study existing methods for finding such dense clusters in linear sequences and to extend them to graphs, which are used to represent collections of related genomes. Vertices in these graphs correspond to parts of the sequences and edges adjacencies between them.

Supervisor:	doc. Mgr. Bronislava Brejová, PhD.	
Department:	FMFI.KI - Department of Computer Science	
Head of department:	prof. RNDr. Martin Škoviera, PhD.	
Assigned:	13.12.2021	
Approved:	03.01.2022	prof. RNDr. Rastislav Kráľovič, PhD. Guarantor of Study Programme

....

Student Supervisor

Acknowledgments: I would like to express my sincere appreciation to my supervisor, doc. Mgr. Broňa Brejová, PhD., for her invaluable guidance and feedback, as well as to my family for their unwavering support during my master's thesis journey. Without their support, this accomplishment would not have been possible.

Abstrakt

V mnohých bioinformatických aplikáciách identifikujeme biologicky významné miesta v individuálnom genóme. Tieto miesta môžu predstavovať napríklad mutácie, gény alebo pozície s určitou biologickou funkciou. V našej práci sa zaoberáme hľadaním zhlukov s vysokou hustotou takýchto biologicky významných miest v grafickom pangenóme. Grafický pangenóm je súbor príbuzných genómov reprezentovaný grafom. Vrcholy v týchto grafoch zodpovedajú častiam sekvencií a hrany zodpovedajú susednostiam medzi nimi. V tejto práci preštudujeme existujúce metódy na nájdenie takýchto hustých zhlukov v lineárnych sekvenciách a rozšírime ich na grafy. Implementovali sme jeden z našich algoritmov a použili sme ho na vyhľadávanie oblastí bohatých na GC v elasticko-degenerovanom reťazci, ktorý je špecifickou reprezentáciou pangenómu.

Kľúčové slová: pangenóm, elasticko-degenerovaný reťazec, problém maximálneho súčtu segmentov, rozklad grafu na cestu, šírka rozkladu na cestu

Abstract

In many bioinformatics applications, we identify biologically significant locations in an individual genome. These locations may represent for example mutations, genes or positions with a particular biological role. In our work, we are interested in finding clusters with high density of such biologically meaningful locations in a graphical pan-genome, which is a collection of related genomes represented by a graph. Vertices in these graphs correspond to parts of the sequences and edges to adjacencies between them. In this work we study existing methods for finding such dense clusters in linear sequences, and extend them to graphs. We implemented one of our algorithms and used it to search for GC-rich regions in an elastic-degenerate string, which is a specific representation of a pan-genome.

Keywords: pan-genome, elastic-degenerate string, maximum-sum segments problem, path-decomposition, path-width

Contents

Introduction

The DNA molecules encode genetic information of organisms. Thanks to sequencing techniques, it is possible to obtain, reconstruct and store this information in a digital form. Overall, genomic analysis is a versatile tool and has numerous applications in various fields like genetics, evolutionary biology, medicine and so on. The genetic information stored in the DNA can help us understand the genetic basis of various diseases, analyse the evolutionary relationships and genetic diversity, or it can be even used for customizing treatments specific to an individual's unique needs [37].

A *reference genome* is a DNA sequence that is considered representative of the population or species of interest. It is often used as a basis for various genomic analyses. Although there is a significant overlap in the DNA sequence among individuals of a given species, the presence of genetic differences and variations contribute to the uniqueness of each individual's DNA. Given the increasing amount of available genomic data, there has been a proposal to expand the reference genome by incorporating additional genetic sequences. Rather than relying on a single reference genome, the term *pan-genome* was introduced [1]. The pan-genome is a joint representation of multiple genomic sequences used as a reference. Such expansion enhances the diversity of the reference genome, allowing for more comprehensive genomic analyses.

A possible representation of the pan-genome is a graph. In our work, we aim to introduce algorithms that identify clusters of biologically meaningful areas in this graphical pan-genome. These biologically significant regions may represent for example mutations, genes, or positions with a particular biological role.

In the first chapter, we explain the basic concepts from bioinformatics and graph theory that are used in the thesis. In chapter 2, we define the mathematical problem of *Maximum-Score Disjoint Paths* which corresponds to the problem of identifying clusters of biologically meaningful locations in graphical pan-genomes. We propose algorithms for solving the problem on various classes of graphs. We describe a linear-time algorithm on bubble-like graphs which correspond to *elastic-degenerate strings* [21]. For general directed acyclic graphs, we describe an algorithm that processes a graph based on a special decomposition, and whose complexity is exponential in a parameter of the decomposition, but linear in the overall size of the graph. In the third chapter, we create pangenomes from genomic sequences of *E. coli*, and search for GC-rich regions using our linear-time algorithm for the bubble-like graphs.

Chapter 1

Background

In this chapter, we introduce basic terms and concepts in the field of bioinformatics and graph theory related to our work. We summarize the main idea behind an organized collection of the genomic information, the pan-genome, and describe the most popular, graphical representation.

1.1 Basic concepts in bioinformatics

Deoxyribonucleic acid (*DNA*) is a molecule that contains genetic instructions in humans and almost all other organisms. These instructions include information necessary for development, reproduction and functioning.

Nucleotides (or DNA bases) are the basic building blocks of the DNA. These are the following: adenine (A), thymine (T), guanine (G) and cytosine (C). From a less biological perspective, we can imagine the DNA as a linear sequence of characters of a *base alphabet A, T, G* and *C*.

The *genome* is the entire genetic information, i.e. the complete set of DNA of an organism.

Genome sequencing is the process of getting the DNA sequence of an organism's genome.

A *reference genome* of an organism is a digital DNA sequence assembled as a representative example of that organism's genome. This is often used as a standard when comparing and analysing genomic sequences.

1.2 Graph theory definitions

In this section, we define basic terms from graph theory [11] that are used in this work.

In this thesis, we will be working mostly with directed graphs. A directed graph $G = (V, E)$ is a structure that consists of a set of vertices V and a set of arcs E. An *arc* (u, v) is an ordered pair of vertices, where $u, v \in V$, u and v are called the endpoints of the arc. A directed graph with a weight function $w : V \to \mathbb{R}$ is called a *vertex-weighted directed graph*, or *weighted directed graph* for short. Note that in this thesis we consider only vertex-weighted directed graphs.

Definition 1.2.1 (*Induced subgraph*)**.** Let $S \subseteq V$ be a subset of vertices of G, then the induced subgraph $G[S]$ is a graph $G' = (S, E')$ where E' consists of all the arcs from E that have both endpoints in S.

Definition 1.2.2 (*Path*)**.** A path is a sequence of distinct vertices $(v_1, ..., v_n)$ where arc $(v_i, v_{i+1}) \in E$.

Definition 1.2.3 (*Cycle*)**.** A cycle is a path $(v_1, ..., v_n)$ where $(v_n, v_1) \in E$.

Definition 1.2.4 (*DAG*)**.** A *directed acyclic graph* (DAG), is a directed graph without cycles.

Definition 1.2.5 (*Topological order*)**.** A topological ordering of a DAG is a linear ordering of its vertices such that for every arc (u, v), vertex u comes before v in the ordering.

A DAG $G = (V, E)$ can be topologically ordered.

Definition 1.2.6 (*Path graph*)**.** A path graph is a graph which consists of a single path.

Definition 1.2.7 (*Predecessor*)**.** Vertex p is a predecessor of vertex v if there is an arc (p, v).

1.3 Computational pan-genomics

At the end of the 20th century, the first complete genome sequence was published which was for the bacterium *Haemophilus influenzae* [16]. Later on, new, cheaper sequencing technologies were introduced [28]. This meant significant growth of available genome sequences. Redefining the term *reference genome* to comprise more information of several genome sequences could probably improve the results of some experiments and analyses reducing reference bias. To take advantage of more available genome sequences instead of one reference genome, the term *pan-genome* was introduced [1]. The *pan-genome* refers to the representation of all genomic content in certain species or related individuals, i.e. it is a collection of genomic sequences used jointly for reference.

The analysis of the SARS-CoV-2 genome sequences during the COVID-19 pandemic is a great example to show the significance of the use of a pan-genome [29]. For comparative purposes, the genome of the virus is often re-sequenced and compared to a reference. However, if there is a subsequence of a newly sequenced virus genome that significantly differs from the reference genome, it is often ignored. Thanks to the added diversity in a pan-genome, this reference bias is reduced.

Graphical pan-genomics

There are several approaches how the pan-genomes can be represented. As the pan-genome tries to replace the traditional, linear reference genome with a collection of genomic sequences, a natural representation might be a graph. In this graph, the vertices correspond to parts of the genome sequences, edges denote the adjacencies among them.

We list two specific representations and show how they look like on an example of three genomic sequences seen in Figure 1.1.

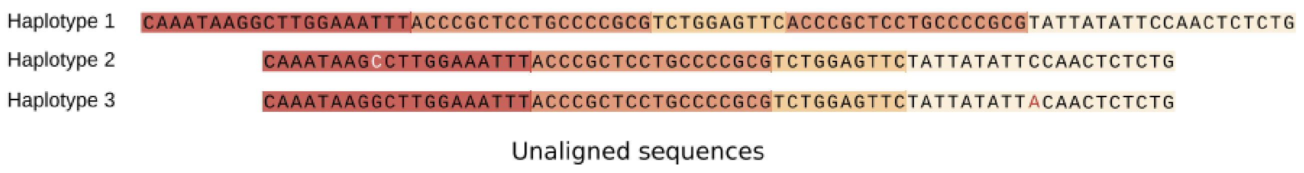
Haplotype 1 CAAATAAGGCTTGGAAATTTACCCGCTCCTGCCCCGCGTCTGGAGTTCACCCGCTCCTGCCCCGCGTATTATATTCCAACTCTCTG
Haplotype 2 CAAATAAGCCTTGGAAATTTACCCGCTCCTGCCCCGCGTCTGGAGTTCTATTATATTCCAACTCTCTG
Haplotype 3 CAAATAAGGCTTGGAAATTTACCCGCTCCTGCCCCGCGTCTGGAGTTCTATTATATTACAACTCTCTG
Unaligned sequences

Figure 1.1: *Three unaligned genomic sequences* [1]

De Bruijn Graph

The de Bruijn Graph is a special type of graph with a given parameter k that represents similarities in sequences of symbols. The graph nodes represent a contiguous subsequence of k matching symbols (i.e. *k-mers*). The edges of this graph represent an overlap of k - *1* matching symbols between a pair of nodes [29].

Obviously, picking the best parameter k is an important task. For larger k we

are able to capture similarities better, however, for divergent parts it can result in huge "bubbles".
This representation has many advantages as it is simple, fast and robust, and it is not necessary to create an alignment. However, this representation does not explicitly represent structural information for distances greater than k.
Below in Figure 1.2 we can see an illustration of a de Bruijn graph.

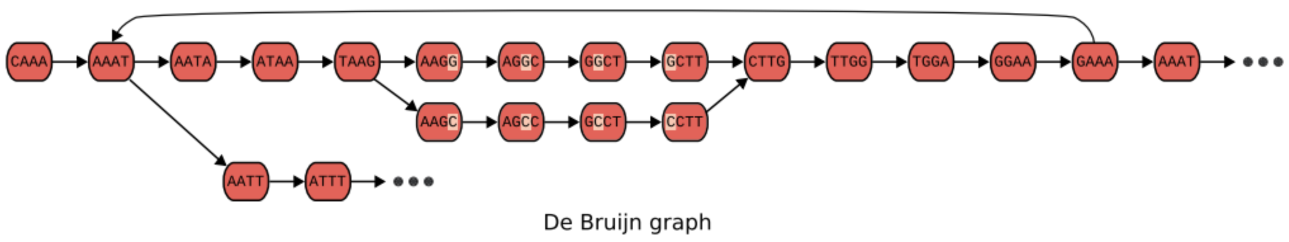

Figure 1.2: *De Bruijn graph representation for parameter $k = 4$ created from genomic sequence Haplotype 1 from Figure 1.1* [1]

Sequence graph

Many modern formulations of the graphical models use a sequence graph. In this representation, nodes correspond to segments of the genome and edges connect adjacent segments. The sequence graph represents matched subsequences from multiple strings as nodes [1, 29]. Individual genome sequences can be represented as paths in this graph where node identifiers may serve as a "coordinate system". In Figure 1.3 we can see an acyclic sequence graph pan-genome.

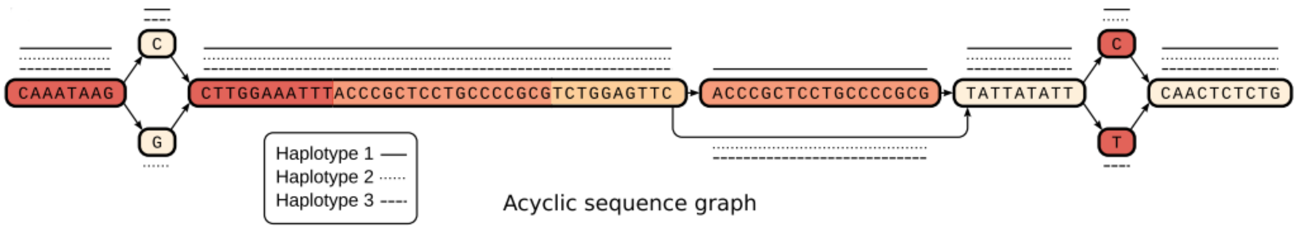

Figure 1.3: *Acyclic sequence graph that represents 3 genomic sequences from Figure 1.1. Each genomic sequence corresponds to a path in the graph.* [1]

In our work, we will use directed acyclic sequence graphs.

Chapter 2

Identifying clusters in pan-genome graphs

In this chapter, we first describe the motivation of finding clusters of biologically significant areas in pan-genomes, and define the mathematical problem of *Maximum-Score Disjoint Paths* which corresponds to the biological problem. We then propose algorithms for solving the problem on various classes of graphs. First we look at graphs consisting of a single path (i.e. a linear sequence) for which an $O(|V|)$-time algorithm already exists [9]. Then we look at bubble graphs and n-layered bubble graphs, which we define later, and propose an $O(|V|)$-time algorithm for solving the problem on them. Finally, we look at directed acyclic graphs and propose an algorithm which runs in $O(|V| \cdot 2^{width} \cdot width)$-time where $width$ is the size of the largest subset of vertices in our graph decomposition (defined later).

2.1 Motivation and related work

There are various types of biologically significant areas in the genome. One example are conserved regions which are slowly changing regions throughout evolution. These might correspond to gene regulatory regions or protein coding regions [36]. Another example are GC-rich regions, i.e. regions where the relative frequency of bases guanine (G) and cytosine (C) is high. Locating GC-rich regions can be used to identify non-coding RNA genes, since the GC-content is significantly higher in non-coding RNA genes compared to the GC-content of the whole genome in some organisms [9]. Another genomic feature that can be investigated are e.g. CpG islands [38], i.e. stretches of DNA from 500 to 1500 basepair length with CG:GC ratio of more than 0.6. Identifying CpG islands can helpful, as they are often associated with gene regulatory elements [10, 26]. Regions with high density of mutations can be considered biologically meaningful. Dense mutations, i.e. areas with dense base substitutions, may for example suggest horizontal sequence transfer [8], which is transmission of portions of genomic DNA between organisms.

All of these examples involve identifying individual bases with some biological property and then looking for groups of such bases located close together. Some algorithms use statistical methods [23, 8, 20, 15, 7, 35], others use variations of the maximum sum subsequence problem [9, 3, 18] or maximum density segment problem [6]. All these algorithms are working on linear genomic sequences. There are some algorithms for finding maximum weight paths on weighted trees [27, 22].

In our work, we would like to find these biologically significant regions in pan-genomes which are represented as graphs. Our approach was based on the maximum sum subsequence problem algorithm used in [9] which we extended for various classes of graphs.

2.2 Problem definition

In the previous chapter, we explained the idea behind having a collection of genomic sequences that are used jointly for reference, a pan-genome. Pan-genomes can be represented as graphs, where the vertices of the graph represent bases or shorter sequences of the genome, and an arc between two vertices denotes adjacency, i.e. how the genome sequence continues. The biological problem of finding biologically meaningful segments of the pan-genome can be translated into the following problem on graphs:

Definition 2.2.1 (*Maximum-Score Disjoint Paths Problem*)**.** The problem is given by a tuple (G, w, x), where $G = (V, E)$ is a directed graph with a weight function $w : V \rightarrow \mathbb{R}$, and $x \in \mathbb{R}^+$ is a penalty value. The problem is finding disjoint paths with the maximal sum of scores. The score of a single path $P = (v_1, v_2, ..., v_n)$ where $v_1, ..., v_n \in V$ is defined as $\sum_{i=1}^{n} w(v_i) - x$.

See an example solution of the problem in Figure 2.1.

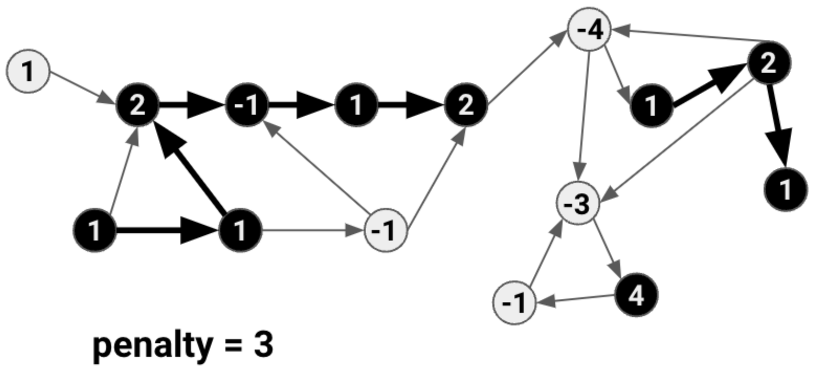

Figure 2.1: *An example weighted directed graph where the selected paths maximise the score for the Maximum-Score Disjoint Paths Problem. The score of this selection is* $(1+1+2-1+1+2-\textit{penalty})+(1+2+1-\textit{penalty})+(4-\textit{penalty}) = 5$.

Complexity of *Maximum-Score Disjoint Paths Problem*

The problem of *Maximum-Score Disjoint Paths* for an arbitrary weighted directed graph is NP-hard. The NP-hardness can be easily proved by using the well-known NP-hard problem, *Hamiltonian Path Problem.* The *Hamiltonian Path Problem* is a problem of determining whether a *Hamiltonian path* exists in a given graph. A *Hamiltonian path* is a path that contains each vertex of the graph exactly once. The reduction of the *Hamiltonian Path Problem* to ours is simple. Take graph $G = (V, E)$, weight function $w(v) = 1$ for $v \in V$, and penalty $x = 1$. The graph contains a Hamiltonian path, if the *Maximum-Score Disjoint Paths Problem* has a solution with score $|V| - 1$.

2.3 Solving the problem on path graphs

In this section, we describe an existing algorithm for solving the *Maximum-Score Disjoint Paths Problem* on path graphs.

This problem can be solved with a dynamic programming algorithm that runs in $O(n)$ time [9]. Note that the algorithm in the original article [9] was designed for a sequence of weights which corresponds to a path graph. Let the path graph G consist of a single path $P = (v_1, ..., v_n)$. The algorithm iterates through the sequence of vertices in path P while populating matrix $W(i, s)$ for all $1 \leq i \leq n$ and $s \in \{0, 1\}$. Value $W(i, 0)$ is defined as the maximum score of selected paths on the subgraph induced by vertices $v_1, ..., v_i$, where v_i is not part of a selected path. Value $W(i, 1)$ is the maximum score of selected paths on the subgraph induced by vertices $v_1, ..., v_i$ where v_i is part of a selected path. To populate matrix W, the algorithm uses the following recursive rules:

$$W(1,0) = 0$$
$$W(1,1) = w(v_1) - x$$

For $2 \leq i \leq n$:
If the vertex v_i is not selected, the best score is the same as the best score for subgraph induced by vertices $v_1, ..., v_{i-1}$.

$$W(i,0) = \max \begin{cases} W(i-1,0) \\ W(i-1,1) \end{cases}$$

If vertex v_i is selected, it may start a new selected path or extend a selected path.

$$W(i,1) = w(v_i) + \max \begin{cases} W(i-1,0) - x \\ W(i-1,1) \end{cases}$$

Notice that the penalty x is applied right away when a new path is selected.

2.4 Solving the problem on simple bubble graphs

In this section, we describe our new algorithm for solving the *Maximum-Score Disjoint Paths Problem* on a bubble-like structure, simple bubble graphs, which can be interpreted as pan-genomes consisting of two genomes. This approach is an extension of the algorithm for path graphs from the previous section.

2.4.1 Preliminaries

Definition 2.4.1 (*Bubble*)**.** A *bubble* [5] is a directed acyclic graph with a start vertex s, an end vertex t and two non-empty vertex-disjoint directed paths, referred to as layers, connecting vertex s and t.

Definition 2.4.2 (*Simple bubble graph*)**.** A *simple bubble graph* can be constructed by taking a sequence of vertices $u_1, ..., u_k$ and connecting each pair of u_i and u_{i+1} by a directed path or by a bubble with the start vertex u_i and the end vertex u_{i+1}.

See Figure 2.2 with an example of a simple bubble graph. Simple bubble graphs can be interpreted as a pan-genome consisting of two genomes. The vertices represent single bases or a sequence of bases. Identical parts of the two sequences can be represented with a common vertex or a path of vertices in the pan-genome. Layers in bubbles represent areas where the bases in the genomes vary.

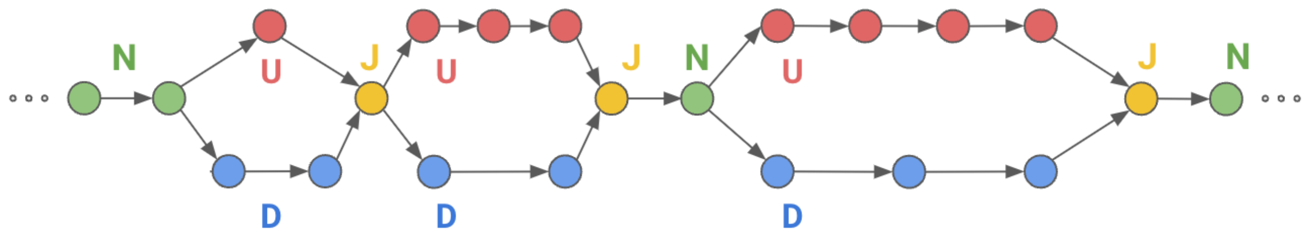

Figure 2.2: *Example of a simple bubble graph from Definition 2.4.2 with 3 bubbles*

Notation 1. Let $G = (V, E)$ be a simple bubble graph. Graph G's vertices are split into mutually disjoint subsets N, J, U and D, i.e. $V = N \cup J \cup U \cup D$, the following way:

- Subset D and U: as mentioned in the definition of a bubble, a bubble consists of a start and end vertex and two disjoint paths. Set D contains vertices from one of the paths, and set U contains vertices from the other path.
  In addition, we define set $D_{first} \subseteq D$ which contains every vertex of D that does not have a predecessor in D. Set $D_{later} = D - D_{first}$.
  Similarly, $U_{first} \subseteq U$ contains every vertex of U that does not have a predecessor in U. Set $U_{later} = U - U_{first}$.

- Subset J: contains all the end vertices of the bubbles.

- Subset N: all remaining vertices.

The algorithm on simple bubble graphs processes the vertices in a special order defined below:

Definition 2.4.3 (*Custom topological sort on simple bubble graphs*)**.** A *custom topological sort on simple bubble graphs* is a topological ordering with the following additional property: within each bubble, vertices in D are before vertices in U.

2.4.2 $O(|V|)$-time algorithm on simple bubble graphs

The algorithm can be summarised as follows: The algorithm is a natural extension of the dynamic programming algorithm for path graphs from Section 2.3. The vertices are being processed in the order of the custom topological sort for simple bubble graphs from Definition 2.4.3. The score matrix W contains the best scores for subgraphs of G. For vertices $N \cup D \cup J$, the score matrix W contains the best score for the subgraph induced by vertices in the custom topological order up to the current vertex. For vertices in U, the scores are limited only to the vertices in U of the current bubble. This is needed so it is possible to calculate the score for vertices in J, i.e. the bubble's end vertex. See Figure 2.3 for an example of the induced subgraphs. The algorithm also needs to consider that a selected path can continue into either of the layers of a bubble. This is stored as an additional dimension in the score matrix W.

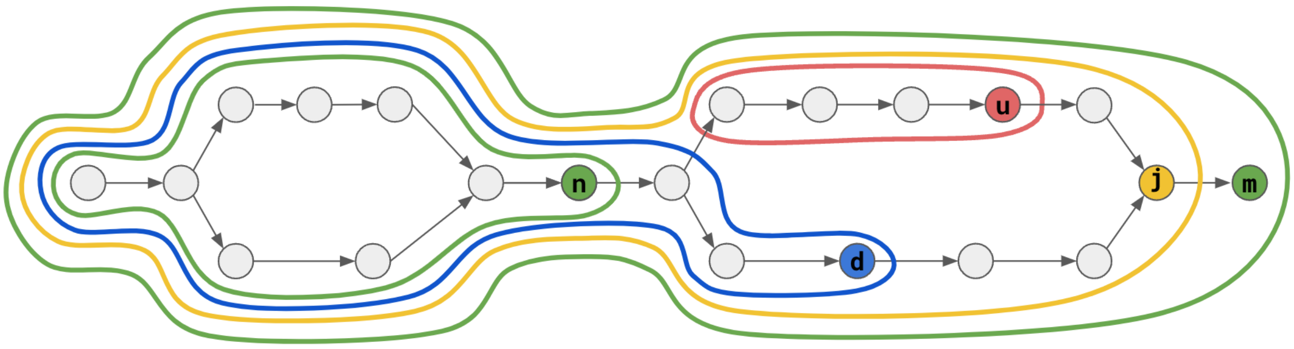

Figure 2.3: *This example highlights vertices $n, m \in N$, $d \in D$, $u \in U$ and $j \in J$, and the related induced subgraphs for which the score matrix W stores the best scores.*

Detailed description of the algorithm

Notation 2. Additional notation used in the algorithm:

- Vertex a: the current vertex that is being processed by the algorithm.
- Vertex p: predecessor vertex of the current vertex a. Note that each vertex has only one predecessor p, except for vertices in J which have two predecessors:
  - predecessor vertex p_D: the predecessor vertex from D,

  - predecessor vertex p_U: the predecessor vertex from U.

- Custom topological ordering $O = v_1, ..., v_n$ on graph G from Definition 2.4.3.
- Score matrix $W(v_i, s, l)$, where $v_i \in V$, selection value $s \in \{0, 1\}$, and path continuation value $l \in \{D, U\}$.
  Value $W(v_i, s, l)$ denotes:
  - for $v_i \in N \cup J \cup D$:
    the best score for the subgraph induced by vertices in the custom topological order up to vertex v_i, i.e. induced by the vertices $\{v_1, ..., v_i\}$;
  - for $v_i \in U$:
    the best score for the subgraph induced by vertices $\{v_1, ..., v_i\} \cap U'$ where U' are the vertices from U in the current bubble;

  where:

  - $s = 1$ means v_i is selected in a path;
  - $s = 0$ means v_i is not selected in any of the paths;

  and where:

  - $l = U$ means that $v_i \in D \cup U$ and the start vertex and the U_{first} vertex of the current bubble is selected on the same path (see Figure 2.4 and Figure 2.5);
  - $l = D$ means that the condition for $l = U$ is not true, i.e. either:
    * $v_i \notin D \cup U$;
    * $v_i \in D \cup U$, but not both the start vertex and the U_{first} vertex of the current bubble is selected (see Figure 2.6 and Figure 2.7);
    * $v_i \in D \cup U$, both the start vertex and the U_{first} vertex of the current bubble is selected, but they are part of two different selected paths (see Figure 2.8).

  Note that for $v_i \notin D \cup U$ the $W(v_i, s, U)$ is not defined and can be considered as being $-\infty$.

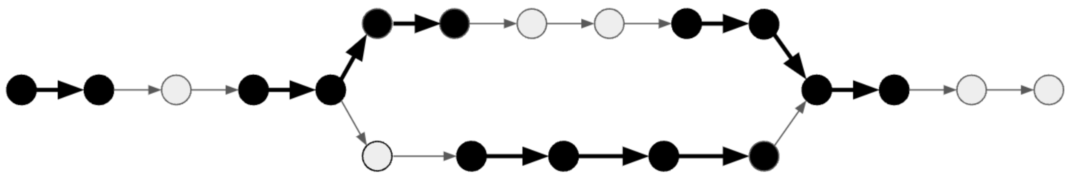
Figure 2.4: *Example of selected paths (marked with black)*

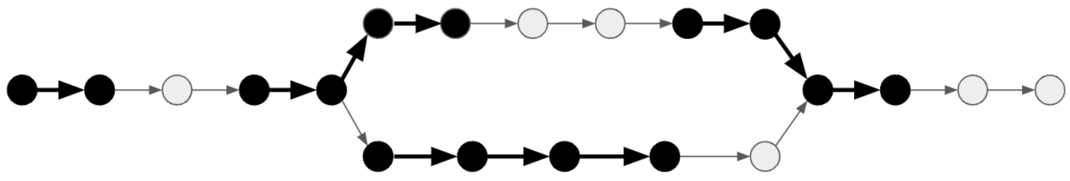
Figure 2.5: *Example of selected paths (marked with black)*

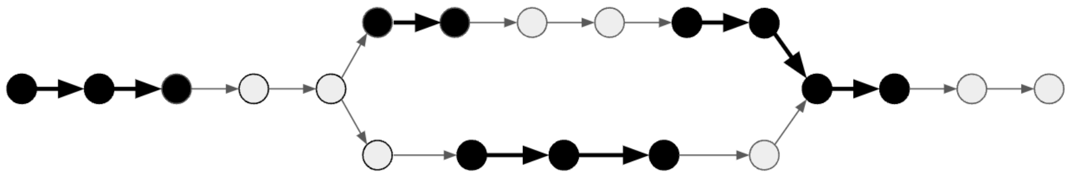
Figure 2.6: *Example of selected paths (marked with black)*

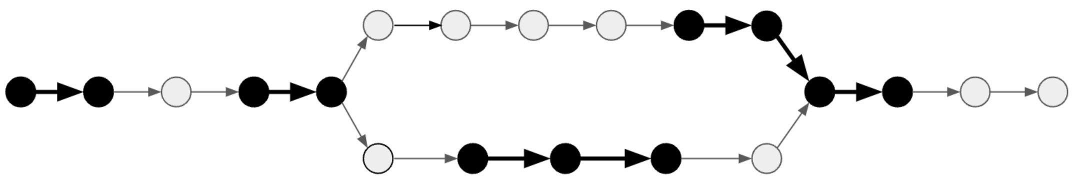
Figure 2.7: *Example of selected paths (marked with black)*

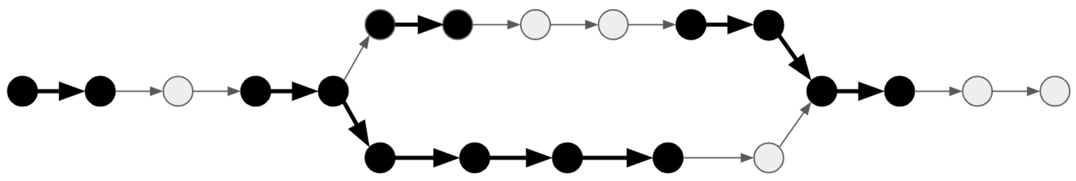
Figure 2.8: *Example of selected paths (marked with black)*

The algorithm processes the vertices in the order of the custom topological sort O, and calculates the values in the score matrix W based on the rules below:

- For v_1, i.e. the first vertex of the custom topological ordering O:

  $$W(v_1, 0, D) = 0$$
  $$W(v_1, 1, D) = w(v_1) - x$$

  Since $v_1 \in N$, value $W(v_1, 0, U)$ and $W(v_1, 1, U)$ are not defined and can be considered as being $-\infty$.

- For $a = v_i \in N \cup D_{later}$ where $i > 1$:
  For these vertices, W stores the best score for the subgraph induced by vertices $v_1, ..., v_i$, therefore, the rules are very similar to the ones described in Section 2.3 for the path graph.
  Notice that the predecessor vertex p of a is v_{i-1}.
  If vertex a is not selected, the best score is the same as for the graph induced by $v_1, ..., v_{i-1}$:

  $$W(a, 0, l) = \max \begin{cases} W(p, 0, l) \\ W(p, 1, l) \end{cases}$$

  where $l \in \{D, U\}$.

  If vertex a is selected, we may start a new selected path or extend an existing selected path. Selecting the vertex increases the score by $w(a)$ and starting a new path incurs a penalty x:

  $$W(a, 1, l) = w(a) + \max \begin{cases} W(p, 0, l) - x \\ W(p, 1, l) \end{cases}$$

  where $l \in \{D, U\}$.

- For $a = v_i \in D_{first}$:
  Also for these vertices, W stores the best score for the subgraph induced by vertices $v_1, ..., v_i$. In this case, however, different rules are needed depending on the value l. The predecessor p of vertex a is v_{i-1} and it is also the start vertex of the current bubble.
  If $l = D$, it means that p may or may not be selected, and if p and a are selected they can be part of the same selected path. This means that the same rules apply as above:

  $$W(a, 0, D) = \max \begin{cases} W(p, 0, D) \\ W(p, 1, D) \end{cases}$$

$$W(a,1,D) = w(a) + \max\begin{cases} W(p,0,D) - x \\ W(p,1,D) \end{cases}$$

If $l = U$, it means that p is selected, and even if a is selected, vertex a and p are part of two distinct selected paths, i.e. if a is selected the score is penalised.

$W(a,0,U) = W(p,1,D)$

$W(a,1,U) = w(a) + W(p,1,D) - x$

- For $a = v_i \in U$:

  For these vertices, W stores the best score for the subgraph induced by vertices $\{v_1, ..., v_i\} \cap U'$ where U' are the vertices from U in the current bubble. This means that the scores are calculated similarly as if the vertices U' formed a completely separate path graph. The only difference is that for $a \in U_{first}$ the penalty is not applied if $l = U$. This is because $l = U$ means that there is a selected path containing both the current bubble's start vertex and the U_{first} vertex of the current bubble.

  For $a \in U_{later}$:

$$W(a,0,l) = \max\begin{cases} W(p,0,l) \\ W(p,1,l) \end{cases}$$

$$W(a,1,l) = w(a) + \max\begin{cases} W(p,0,l) - x \\ W(p,1,l) \end{cases}$$

  where $l \in \{D, U\}$.

  For $a \in U_{first}$:

  $W(a,0,D) = 0$

  $W(a,1,D) = w(a) - x$

  $W(a,0,U) = -\infty$

  $W(a,1,U) = w(a)$

  Note that $W(a,0,U)$ is $-\infty$, because this case must not happen, as $l = U$ means that $a \in U_{first}$ is selected, but $s = 0$ means that a is not selected.

- For $a = v_i \in J$:
  For these vertices, W stores the best score for the subgraph induced by vertices $v_1, ..., v_i$, but in this case vertex a has two predecessors $p_D \in D$ and $p_U \in U$.
  The values stored in score matrix W for p_D and p_U were calculated for two disjoint subgraphs (see Figure 2.9).

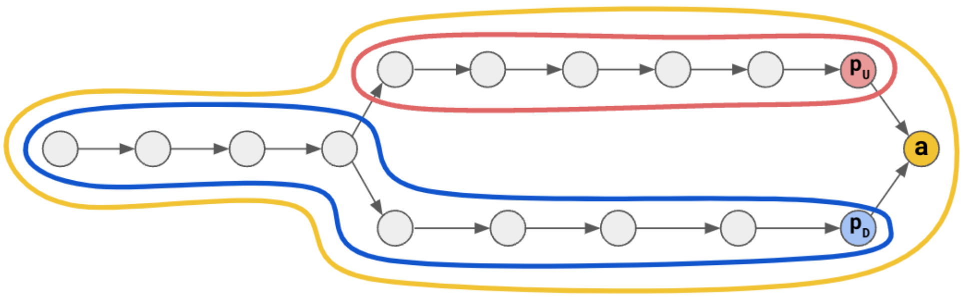

Figure 2.9: *The disjoint subgraphs for which the values* $W(p_U, \_, \_)$ *and* $W(p_D, \_, \_)$ *were calculated.*

Summing up these scores and adding $w(a)$ if a was selected gives the best score for the subgraph induced by vertices $v_1, ..., v_i$. When summing up the scores, all the possibilities whether the previous vertices were selected or not need to be considered. The W values for the two predecessors have to be considered with the same l value, since it does not make sense to use the $l = D$ value for one of the predecessors and $l = U$ value for the other predecessor.
Penalty is only applied when a is selected and none of the predecessors are selected. If at least one of the predecessors is selected, the path can be extended with a without a penalty.

$$W(a, 0, D) = \max_{l \in \{D,U\}} \left(\max_{s \in \{0,1\}} (W(p_D, s, l)) + \max_{s \in \{0,1\}} (W(p_U, s, l)) \right)$$

$$W(a, 1, D) = w(a) + \max_{l \in \{D,U\}} \left(\max \begin{cases} W(p_D, 0, l) + W(p_U, 0, l) - x \\ W(p_D, 0, l) + W(p_U, 1, l) \\ W(p_D, 1, l) + W(p_U, 0, l) \\ W(p_D, 1, l) + W(p_U, 1, l) \end{cases} \right)$$

Since $a \in J$, value $W(a, 0, U)$ and $W(a, 1, U)$ are not defined and can be considered as being $-\infty$.

Notice that the penalty x is applied right away when a new path is selected.

Let v be the last vertex of the graph. Since $v \in N \cup J$, the best score of the whole graph is $\max\{W(v, 0, D), W(v, 1, D)\}$.

Path reconstruction

To return the selected paths, besides matrix *W*, the algorithm populates matrix *C* which has the exact same dimensions as *W*. On position *C(v, s, l)* the choice of how the rule for $W(v, s, l)$ was derived is encoded. This serves as a traceback pointer. That is, after the algorithm finishes with the score calculation, it starts reconstructing the paths by iterating through the graph in the reversed custom topological ordering. By following the pointers in matrix *C*, it can trace back which vertices were selected and whether a new path was started.

Complexity

Calculating the scores stored in W and values stored in C can be done in constant time for a single vertex. Each vertex is visited twice by the algorithm (once during the score calculation and once during the path reconstruction). Therefore, the overall time complexity of the algorithm is $O(|V|)$. Regarding space complexity, the algorithm stores a constant number of values for each vertex and a few runtime variables. The whole graph can be stored in $O(|V|)$ space. Therefore, the space complexity of this algorithm is $O(|V|)$.

2.5 Solving the problem on n-layered bubble graphs

The simple bubble graph can be interpreted as a pan-genome consisting of two genomes. A natural step to extend this graph is to add more layers inside the bubbles, and with this make it possible to represent pan-genomes with multiple genomes.

In this section, we describe an algorithm for solving the *Maximum-Score Disjoint Paths Problem* on n-layered bubble graphs.

2.5.1 Preliminaries

Definition 2.5.1 (*b-layered bubble*)**.** A *b-layered bubble* is a directed acyclic graph with a start vertex s, an end vertex t and b non-empty vertex-disjoint directed paths, referred to as layers, connecting vertex s and t.

Definition 2.5.2 (*n-layered bubble graph*)**.** An *n-layered bubble graph* can be constructed by taking a sequence of vertices $u_1, ..., u_k$ and connecting each pair of u_i and u_{i+1} by a directed path or by a b-layered bubble ($2 \leq b \leq n$) with the start vertex u_i and the end vertex u_{i+1}.

See Figure 2.10 with an example of a 5-layered bubble graph with a 5-layered and 3-layered bubble. The 5-layered bubble graph can be interpreted as a pan-genome consisting of 5 genomes.

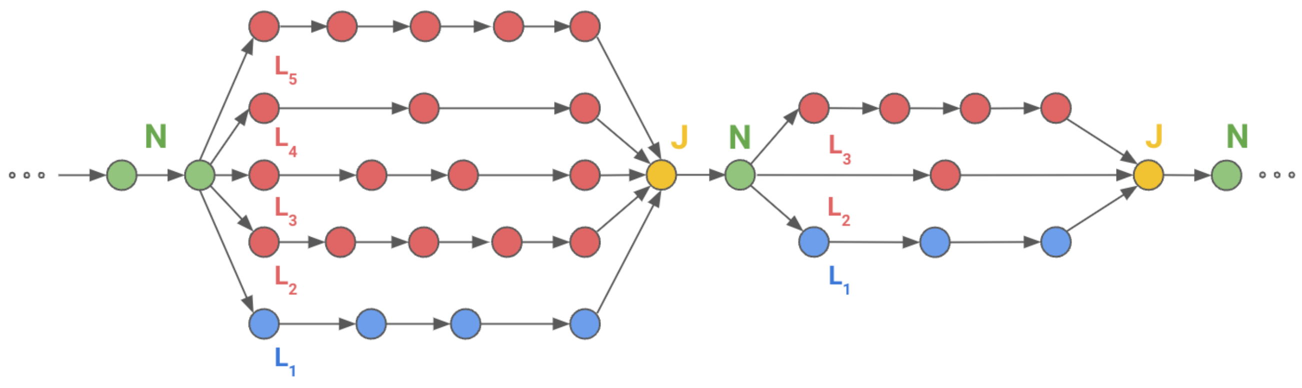

Figure 2.10: *Example of a 5-layered bubble graph from Definition 2.5.2 with a 5-layered bubble and a 3-layered bubble*

Notation 3. For n-layered bubble graphs, a similar notation to Notation 1 is used.

Let $G = (V, E)$ be an n-layered bubble graph. Graph G's vertices are split into mutually disjoint subsets $N, J, L_1, ..., L_n$, i.e. $V = N \cup J \cup L_1 \cup ... \cup L_n$, the following way:

- Subset $L_i \in \{L_1, L_2, ..., L_n\}$: as mentioned in the Definition 2.5.1, an n-layered bubble consists of a start and an end vertex and n disjoint paths $q_1, ..., q_n$. Subset L_i contains all the vertices of disjoint path q_i.
  Furthermore, each L_i is split into two disjoint subsets $L_{i,first}$ and $L_{i,later}$.

Subset $L_{i,first}$ contains all vertices of L_i that do not have a predecessor in L_i, while $L_{i,later} = L_i - L_{i,first}$.

- Subset J: contains all the end vertices of all the b-layered bubbles, $1 \leq b \leq n$.
- Subset N: all remaining vertices.

Definition 2.5.3 (*Custom topological sort on n-layered bubble graphs*)**.** A *custom topological sort on n-layered bubble graphs* is a topological ordering with the following additional property: if vertices u and v belong to the same bubble and $u \in L_i$ and $v \in L_j$ and $i < j$, then vertex u is before vertex v in the topological order.

2.5.2 Elastic-degenerate strings

One possible way to represent graphical pan-genomes is using *elastic-degenerate strings* [21] which correspond to our n-layered bubble graphs. An elastic-degenerate string is a string where at one or more positions an *elastic-degenerate symbol* can occur. The elastic degenerate symbol is defined as a set of substrings, potentially of different lengths. See an example of an elastic-degenerate string and the corresponding n-layered bubble graph in Figure 2.11. Notice that in the n-layered bubble graph an additional vertex was added between the two neighbouring elastic-degenerate symbols and after the last elastic-degenerate symbol (marked with yellow color). These additional vertices act as the start or the end vertices of the bubbles. Their weight is set to 0, and therefore, they do not have an impact on the score calculation.

If the elastic-degenerate symbol contains an ε symbol, a vertex is added for it with 0 weight. This is needed so that selecting this vertex has no effect on the score.

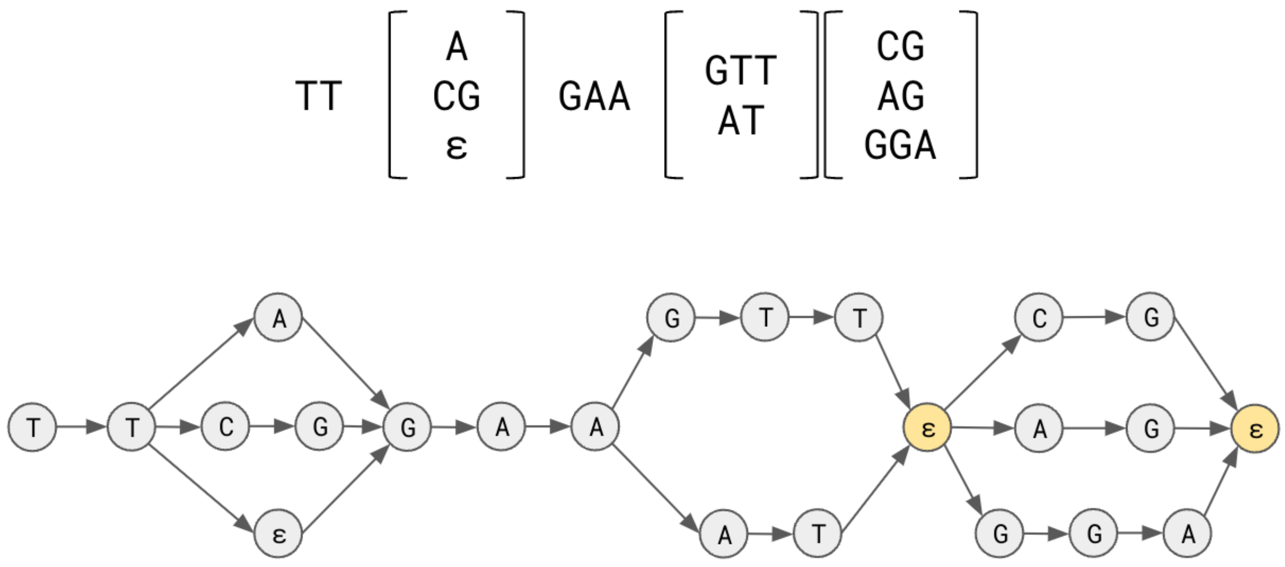

Figure 2.11: *Example of an elastic-degenerate string and the corresponding 3-layered bubble graph. The bases correspond to vertices, adjacencies to edges.*

2.5.3 $O(|V|)$-time algorithm on n-layered bubble graphs

The algorithm on n-layered bubble graphs follows the *O(|V|)*-time algorithm for simple bubble graphs from Subsection 2.4.2. The only difference is in handling vertices in $L_{i,first}$ and vertices in J. A natural step to extend the algorithm for the n-layered bubble graph would be to use $l \in \{L_1, ..., L_n\}$ instead of $L \in \{D, U\}$ when calculating the score $W(v, s, l)$. However, it is enough to have just two possible values for l.
The algorithm considers for each vertex a whether it is included in a path. In contrast to the algorithm for simple bubble graphs, the $s = 0$ has a slightly different meaning, it means that the vertex may or may not be selected. In other words, for selection value $s = 0$ the score in W holds the maximum value of both the option that the vertex is selected or is not selected. This change is needed so that the we can calculate the score for vertices in J more efficiently.

Detailed description of the algorithm

Notation 4. In addition to Notation 3, the following notation is used in the algorithm:

- Vertex a: current vertex that is being processed by the algorithm.

- Vertex p: predecessor vertex of the current vertex a. Note that each vertex has only one predecessor p, except vertices in J which have predecessors $p_1, ..., p_b$ if vertex a is the end vertex of a b-layered bubble.
- Custom topological ordering $O = v_1, ..., v_{|V|}$ on graph G from Definition 2.5.3.
- Score matrix $W(v_i, s, l)$, where $v_i \in V$, selection value $s \in \{0, 1\}$, and path continuation value $l \in \{I, E\}$.
  Value $W(v_i, s, l)$ denotes:
  - for $v_i \in N \cup J \cup L_1$:
    the best score for the subgraph induced by vertices in the custom topological order up to vertex v_i, i.e. induced by the vertices $v_1, ..., v_i$
  - for $v_i \in L_i, 2 \leq i \leq b$ in a bubble B with b layers:
    the best score for the subgraph induced by vertices $\{v_1, ..., v_i\} \cap L'_i$ where L'_i are the vertices from L_i in bubble B;

  where:
  - $s = 1$ means v_i is selected in a path;
  - $s = 0$ means v_i may or may not be selected in a path;

  and where the value l means:
  if $v_i \in L_1$:
  - $l = I$: there is no selected path which contains the bubble's start vertex and the subsequent vertex from $L_{k,first}$ where $k > 1$. Note that the selected path may continue on $L_{1,first}$.
  - $l = E$: the bubble's start vertex is selected in a path which continues with a vertex from $L_{k,first}$ where $k > 1$.

  if $v_i \in L_k, k > 1$:
  - $l = I$: the path from the bubble's start vertex continues on layer L_k, i.e. both the bubble's start vertex and the $L_{k,first}$ vertex of the current bubble are selected;
  - $l = E$: there is no path containing both the current bubble's start vertex and the $L_{k,first}$ vertex of the current bubble.

Note that for $v_i \in N \cup J$ the $W(v_i, s, E)$ is not defined and can be considered as being $-\infty$.

Similarly as the algorithm for simple bubble graphs, the algorithm processes the vertices in the order of the custom topological sort O, and calculates the values in the score matrix W based on the rules below:

- For v_1, i.e. the first vertex of the custom topological ordering:

  $$W(v_1, 1, I) = w(v_1) - x$$

  $$W(v_1, 0, I) = \max \begin{cases} 0 \\ W(v_1, 1, I) \end{cases}$$

  Since $v_1 \in N$, value $W(v_1, 0, E)$ and $W(v_1, 1, E)$ are not defined and can be considered as being $-\infty$.

- For $a = v_i \in N \cup L_{1,later}$ where $i > 1$:
  For these vertices, W stores the best score for the subgraph induced by vertices $v_1, ..., v_i$.
  The rules are very similar as in simple bubble graphs. The difference arises from the fact that $s = 0$ has a different meaning for n-layered bubble graphs: it means that matrix W contains the best score of both cases that a is selected or not selected.

  $$W(a, 1, l) = w(a) + \max \begin{cases} W(p, 0, l) - x \\ W(p, 1, l) \end{cases}$$

  where $l \in \{I, E\}$.

  If a is not selected, then the best score is the same as for the predecessor which is stored in $W(p, 0, l)$. If a is selected, then the best score is $W(a, 1, l)$:

  $$W(a, 0, l) = \max \begin{cases} W(p, 0, l) \\ W(a, 1, l) \end{cases}$$

  where $l \in I, E$.

- For $a = v_i \in L_{1,first}$:
  Also for these vertices, W stores the best score for the subgraph induced by

vertices $v_1, ..., v_i$. The rules are again similar as for simple bubble graphs and the only difference is again because of the different meaning of $s = 0$.

$$W(a,1,I) = w(a) + \max \begin{cases} W(p,0,I) - x \\ W(p,1,I) \end{cases}$$

$$W(a,0,I) \ = \max \begin{cases} W(p,0,I) \\ W(a,1,I) \end{cases}$$

$$W(a,1,E) = w(a) + W(p,1,I) - x$$

$$W(a,0,E) = \max \begin{cases} W(p,1,I) \\ W(a,1,E) \end{cases}$$

- For $a = v_i \in L_k$ where $k > 1$:
  This is again similar as for the simple bubble graphs. For these vertices, W stores the best score for the subgraph induced by vertices $\{v_1, ..., v_i\} \cap L'_k$ where L'_k are the vertices from L_k in the current bubble. This means that the scores are calculated similarly as if the vertices L'_k formed a completely separate path graph. The only difference is that for $a \in L_{k,first}$ the penalty is not applied if $l = I$. This is because $l = I$ means that there is a selected path containing both the current bubble's start vertex and the $L_{k,first}$ vertex of the current bubble.

  For $a \in L_{k,later}$ where $k > 1$:

$$W(a,1,l) = w(a) + \max \begin{cases} W(p,0,l) - x \\ W(p,1,l) \end{cases}$$

$$W(a,0,l) = \max \begin{cases} W(p,0,l) \\ W(a,1,l) \end{cases}$$

  where $l \in \{I, E\}$.

  For $a \in L_{k,first}$ where $k > 1$:

  $W(a,1,I) = w(a)$

  $W(a,0,I) = W(a,1,I)$

$$W(a,1,E) = w(a) - x$$

$$W(a,0,E) = \max\begin{cases} 0 \\ W(a,1,E) \end{cases}$$

- For $a = v_i \in J$, which is the end vertex of the currently processed bubble with $b \leq n$ layers:
  For these vertices, W stores the best score for the subgraph induced by vertices $v_1, ..., v_i$, but in this case vertex a has b predecessors: $p_k \in L_k$, where $1 \leq k \leq b$.
  The values stored in score matrix W for $p_1, ..., p_b$ were calculated for b disjoint subgraphs. To get the score for a, the algorithm has to sum up these scores for $p_1, ..., p_b$. The exact rules are a bit more complex compared to the rules in the algorithm for the simple bubble graph.
  The way the path continuation value l was defined, when deriving the score for $a \in J$, it makes only sense to sum up the scores of predecessors where $l = I$ for exactly one predecessor and $l = E$ for all the others. This is because the selected path from the bubble's start vertex may continue with at most one vertex $v \in L_{k,first}, 1 \leq k \leq b$. Recall that score for p_1 when $l = I$ also includes the possibility that the selected path does not continue to any of the layers from the bubble's start vertex.

  First, let's take a look at rule $W(a,1,I)$, i.e. when a is surely selected. If none of a's predecessors were selected, a starts a new selected path. To allow a to continue an existing path, at least one of its predecessors has to be surely selected. The algorithm also has to consider all the possible l values of predecessors as described above. To calculate the score for $W(a,1,I)$ efficiently, 3 groups of sums are created. The maximum of these sums is used to derive the maximal score for $W(a,1,I)$:

  1. No predecessor vertex is surely selected, therefore, a starts a new path which incurs a penalty.

$$group_1 = \max\begin{cases} W(p_1,0,I) + W(p_2,0,E) + ... + W(p_b,0,E) - x \\ W(p_1,0,E) + W(p_2,0,I) + ... + W(p_b,0,E) - x \\ \cdots \\ W(p_1,0,E) + W(p_2,0,E) + ... + W(p_b,0,I) - x \end{cases}$$

The maximum in $group_1$ goes over b sums, each having path continuation value I at a different position. The value $group_1$ can be calculated in $O(b)$ time. First, the sum $W(p_1, 0, E) + W(p_2, 0, E) + ... + W(p_b, 0, E) - x$ is calculated. Then the algorithm changes exactly one addend at a time from $W(p_k, 0, E)$ to $W(p_k, 0, I)$ for all $1 \leq k \leq b$, and chooses the maximum sum.

2. Predecessor p_k is surely selected (i.e. a does not have to start a new selected path which means no penalty), and $l = I$ for predecessor p_k:

$$group_2 = \max \begin{cases} W(p_1, 1, I) + W(p_2, 0, E) + ... + W(p_b, 0, E) \\ W(p_1, 0, E) + W(p_2, 1, I) + ... + W(p_b, 0, E) \\ \quad ... \\ W(p_1, 0, E) + W(p_2, 0, E) + ... + W(p_b, 1, I) \end{cases}$$

Similarly as $group_1$, value $group_2$ can be calculated in $O(b)$ time by first calculating the sum $W(p_1, 0, E) + W(p_2, 0, E) + ... + W(p_b, 0, E)$, and then always changing exactly one addend at a time from $W(p_k, 0, E)$ to $W(p_k, 1, I)$ for all $1 \leq k \leq b$, and choosing the maximum sum at the end.

3. Predecessor p_k is surely selected (i.e. a does not have to start a new selected path which means no penalty), and $l = I$ for predecessor p_j where $k \neq j$.

   In total $O(b^2)$ sums of length b need to be calculated and compared. This could be done in $O(b^2)$ time similarly as above.

   However, with a better approach, the maximum sum can be calculated in $O(b)$ time. The algorithm first calculates the sum $group_3 = W(p_1, 0, E) + W(p_2, 0, E) + ... + W(p_b, 0, E)$. Then it finds addends $W(p_y, 0, E)$ and $W(p_z, 0, E)$ which when replaced with $W(p_y, 0, I)$ and $W(p_z, 1, E)$ maximises the sum.

   To do this, the algorithm first finds p_c and p_d ($c \neq d$) for which the difference $W(p_y, 0, I) - W(p_y, 0, E)$ is the largest and second largest, respectively ($1 \leq y \leq b$). This is done in $O(b)$ time.

   Next, it finds the predecessors p_e and p_f ($e \neq f$) for which the difference $W(p_z, 1, E) - W(p_z, 0, E)$ is the largest and second largest, respectively, again in $O(b)$ time ($1 \leq z \leq b$).

   There are 4 candidates for the replacement. If $p_c \neq p_e$, then the

addend $W(p_c, 0, E)$ is replaced with $W(p_c, 0, I)$, and $W(p_e, 0, E)$ with $W(p_e, 1, E)$ in the $group_3$ sum. If $p_c = p_e$, then one of the addends is replaced by $W(p_d, 0, I)$ or $W(p_f, 1, E)$ instead, whichever results in a larger sum.
All together, this is done in $O(b)$ time.

That is, $W(a, 1, I)$ is derived in the following way:

$$W(a,1,I) = w(a) + \max \begin{cases} group_1 \\ group_2 \\ group_3 \end{cases}$$

Now let's look at $W(a, 0, I)$. The best score if a is surely not selected is calculated similarly as $group_1$, the only difference is that the penalty is not applied:

$$group\_4 = \max \begin{cases} W(p_1,0,I) + W(p_2,0,E) + ... + W(p_b,0,E) \\ W(p_1,0,E) + W(p_2,0,I) + ... + W(p_b,0,E) \\ \quad ... \\ W(p_1,0,E) + W(p_2,0,E) + ... + W(p_b,0,I) \end{cases}$$

The score of $W(a, 0, I)$ is the maximum of the score when a is surely not selected and of the score when a is surely selected:

$$W(a,0,I) \; = \max \begin{cases} group\_4 \\ W(a,1,I) \end{cases}$$

Since $a \in J$, value $W(a, 0, E)$ and $W(a, 1, E)$ are not defined and can be considered as being $-\infty$.

Let v be the last vertex of the graph. Since $v \in N \cup J)$, the maximal score of the graph is $W(v, 0, I)$.

Path reconstruction

The path reconstruction is done as described for the algorithm on simple bubble graphs, see 2.4.2.

Complexity

Regarding time complexity, apart from vertices in J, calculating the scores stored in W for a vertex is done in $O(1)$. Calculating the scores for vertices in J is done in $O(\sum_{b \in B} size(b)) \leq O(|V|)$ where B is the set of all bubbles in G and *size(b)* is the number of layers in a bubble $b \in B$. To reconstruct the path, the algorithm iterates through the graph once again based on values stored in C in time $O(|V|)$. Therefore, the algorithm's runtime is $O(|V|)$. Similarly, the space complexity is also $O(|V|)$, as the algorithm stores the graph, matrices W and C, and a few runtime variables.

2.5.4 Possible extension

One possible way to extend our algorithm on n-layered bubble graphs to a bigger class of graphs would be to have recursive b_r-layered bubbles inside the b-layered bubbles, $b, b_r \in \{2, ..., n\}$. The acyclic *two-terminal series-parallel graphs* [13] corresponds to this class, see the definition below:

Definition 2.5.4 (*Directed two-terminal series-parallel graphs*)**.** [13] A directed *two-terminal series-parallel graph* G, with terminals (terminal vertices) s and t can be created by a sequence of operations $(i) - (iii)$:

(i) Create a new graph consisting of only one arc (s, t).

(ii) *Parallel composition*: let both X with terminals s_X, t_X and Y with terminals s_Y, t_Y be two-terminal series-parallel graphs. Form a new graph $G = P(X, Y)$ by identifying $s = s_X = s_Y$ and $t = t_X = t_Y$.

(iii) *Series composition*: let both X with terminals s_X, t_X and Y with terminals s_Y, t_Y be two-terminal series-parallel graphs. Form a new graph $G = S(X, Y)$ by identifying $s = s_X$, $t = t_Y$ and $t_X = s_Y$.

See in Figure 2.12 how operations (ii) and (iii) can form a new two-terminal series-parallel graph.

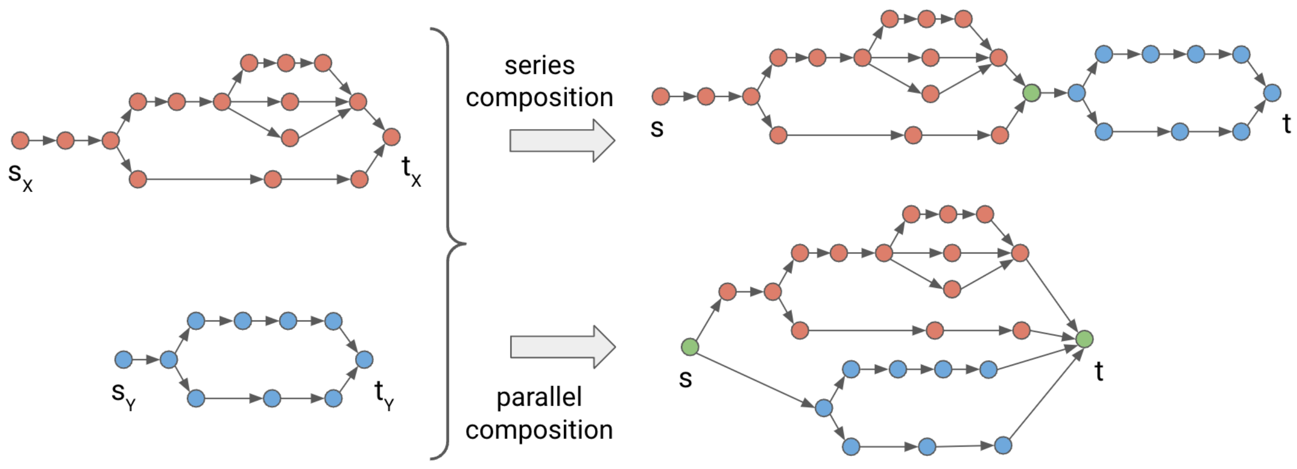

Figure 2.12: *Parallel composition or series composition of two two-terminal series-parallel graphs form a new two-terminal series-parallel graph.*

We discovered this class of graphs shortly before the thesis submission deadline based on a proposal from prof. RNDr. Rastislav Královič, PhD. We leave it as an open problem, however, most likely our algorithm on n-layered bubble graphs can be extended to work on this class of graphs in $O(|V|)$ time.

2.6 Solving the problem on a DAG

In the previous sections, we described algorithms for the *Maximum-Score Disjoint Paths Problem* on simple bubble graphs and n-layered bubble graphs (see Definition 2.4.2 and Definition 2.5.2). In Figure 2.13, we can see a 2-layered bubble. By definition, between vertices denoted by B and J there are only two disjoint paths, i.e. layers.

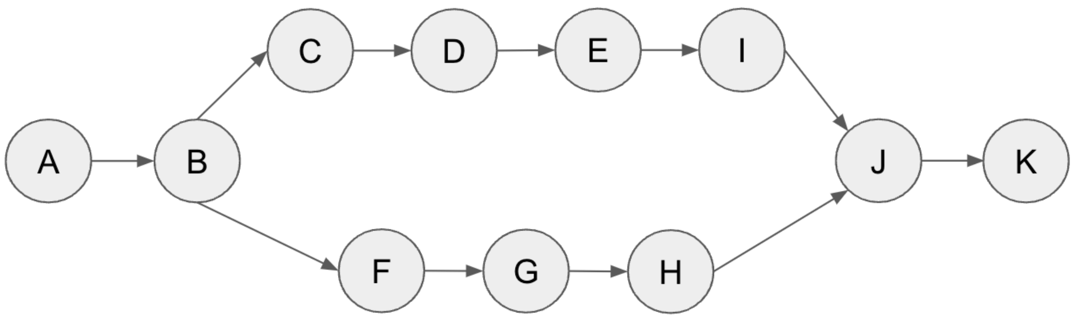

Figure 2.13: *A 2-layered bubble from the previous section, see Definition 2.5.1*

A natural way to extend the graph in Figure 2.13 is to add arcs in between vertices on different layers as seen in Figure 2.14. This newly constructed graph does not match our definition of b-layered bubble graphs, so the previously defined algorithms cannot be used.

Having this in our mind as a motivation, in this section, we describe an algorithm that solves the *Maximum-Score Disjoint Paths Problem* on DAGs.

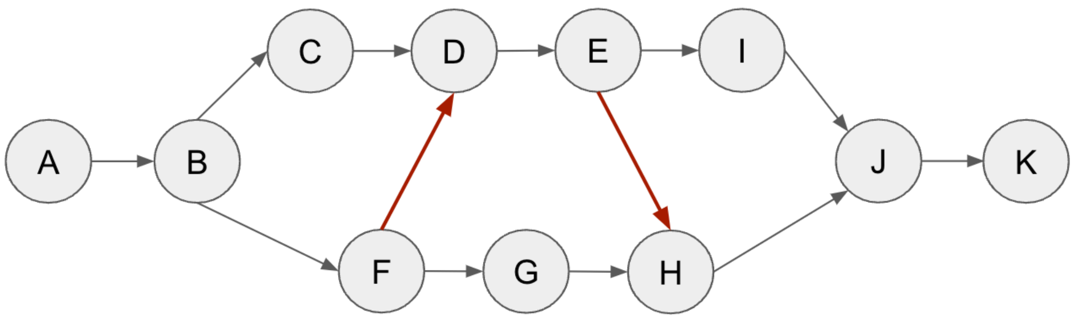

Figure 2.14: *An extended simple bubble graph from Figure 2.13 by adding arcs FD and EH*

2.6.1 Preliminaries

The dynamic programming algorithm can be summarised as follows: The dynamic programming algorithm gets a connected weighted DAG G as an input and also a special decomposition of G. This decomposition splits the vertices into a sequence of non-disjoint subsets called *bags*. The bags are processed one by one based on the order in the sequence. For each vertex in each bag, the algorithm considers whether the vertices are part of a selected path, and if so, whether the selected path ends in that vertex. For each of these possibilities, the algorithm calculates the score for the bag. This is done based on the scores calculated for the previous bag.
As one can probably assume, the time complexity of this problem heavily depends on the size of the bags. Let us define below the decomposition that is used by this algorithm.

Our decomposition can be viewed as a variant of the *path decomposition* defined usually for undirected graphs. The path-decomposition of graph G can be interpreted as a *thickened* path graph. The term *path-width* is a value describing that how much this path is thickened to get G. Formally:

Definition 2.6.1 (*Path-decomposition*)**.** [33] Let $G = (V, E)$ be an undirected graph. The *path-decomposition* of G is a sequence of subsets $X_1, ..., X_n$ of V, with two properties:

(i) For each edge of G, there exists an $i \in \{1, ..., n\}$ such that both endpoints of the edge belong to bag X_i.

(ii) For every three indices $1 \leq i \leq j \leq k \leq n,\ X_i \cap X_k \subseteq X_j$.

Definition 2.6.2 (*Path-width*)**.** [33] Let $G = (V, E)$ be an undirected graph. The *path-width* of G is the minimum value of $k \geq 0$ such that G has a path-decomposition $P = X_1, ..., X_n$ where $k = \max\limits_{i \in \{1,...,n\}} |X_i| - 1$.

Note that it is NP-hard to find the path-width of an arbitrary graph [19].

In our case, G is a DAG. An existing definition of a *directed path-decomposition* is the following:

Definition 2.6.3 (*Directed path-decomposition*)**.** [2, 14] Let $G = (V, E)$ be a directed graph. A *directed path-decomposition* is a pair (P, X) consisting of a path P, say with $V(P) = \{t_1, t_2, ..., t_n\}$, together with a collection of subsets of vertices $X = \{V_i \subseteq V(G) : i \in [n]\}$ such that:

(i) $V(G) = \bigcup_{t \in V(P)} V_t$;

(ii) if $i < j < k$, then $V_i \cap V_k \subseteq V_j$;

(iii) for every edge $e = (x, y) \in E(P)$ there exists $i \leq j$ such $x \in V_i$ and $y \in V_j$.

The algorithm for n-layered bubbles processed the graph based on a special topological order. This ensured that each vertex's predecessor was already processed before the vertex itself. This property should be ensured when processing the path-decomposition, i.e. all predecessors of a vertex should be processed in a previous or the current bag. However, Definition 2.6.3 does not ensure this. In Figure 2.15, we can see an example directed path-decomposition based on Definition 2.6.3. In this example, we can see that vertex B, vertex D's predecessor, appears in a bag for the first time only after vertex D's first appearance.

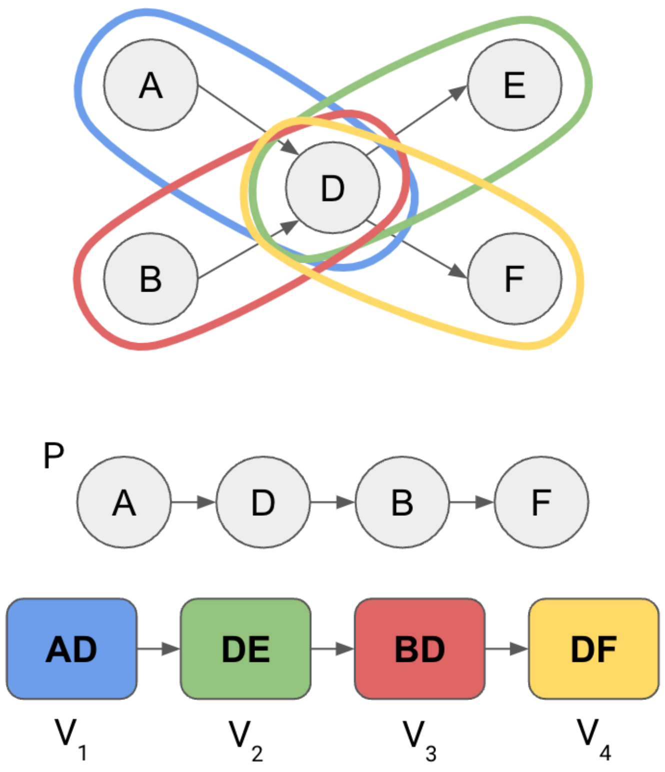

Figure 2.15: *A possible directed path-decomposition of a graph based on Definition 2.6.3. Notice that not all predecessors of D appear before D's first appearance in a bag in the path-decomposition.*

To ensure the property that when processing vertex v all predecessors of v were already processed or are processed together with v, we define our own *modified directed path-decomposition* by extending Definition 2.6.1:

Definition 2.6.4 (*Modified directed path-decomposition*)**.** Let $G = (V, E)$ be a directed graph. The *modified directed path-decomposition* of G is a sequence of subsets $X_1, ..., X_n$ of V (we refer to them as *bags* of vertices), with three properties:

(i) For each arc of G, there exists an $i \in \{1, ..., n\}$ such that both endpoints of the arc belong to bag X_i.

(ii) For every three indices $1 \leq i \leq j \leq k \leq n$, $X_i \cap X_k \subseteq X_j$.

(iii) If vertex $v \in X_j$ then for each of v's predecessors p exists a bag X_i containing p where $i \leq j$.

The definition of directed path-width conceptually remains the same, that is:

Definition 2.6.5 (*Width of the path-decomposition*)**.** Let $G = (V, E)$ be a directed graph and $P = X_1, ..., X_n$ its path-decomposition. The *width of the path-decomposition* P is $k = \max_{i \in \{1,...,n\}} |X_i| - 1$.

Definition 2.6.6 (*Modified directed path-width of a graph*)**.** Let $G = (V, E)$ be a directed graph. The *modified directed path-width of* G is the minimal possible width of a modified directed path-decomposition on G. A path-decomposition with this width is called *minimal path-decomposition*.

Note that in this work, whenever we refer to a path-decomposition, or a path-width of a graph, we are referring to Definition 2.6.4 and Definition 2.6.6.

Below in Figure 2.16, we can see a path-decomposition of width 2 based on Definition 2.6.4 of the graph from Figure 2.14. Based on Corollary 2.6.1 shown below, the path-width of the graph is at least 2. Therefore, this is a minimal path-decomposition of the graph.

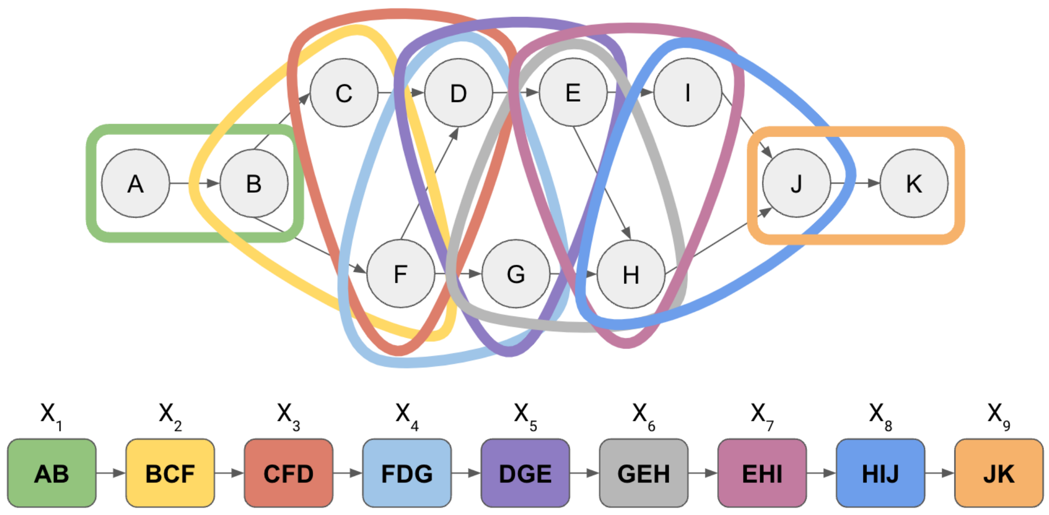

Figure 2.16: *Minimal path-decomposition based on Definition 2.6.4 of the graph from Figure 2.14 with path-width 2*

The following lemma shows an interesting property of our modified directed path-decomposition. This can be useful for example for showing a lower bound for the path-width of a graph.

Lemma 2.6.1. *Let $G = (V, E)$ be a directed graph and $P = X_1, ..., X_n$ its path-decomposition from Definition 2.6.4. Assume X_i is the bag where vertex v appears for the first time in P, i.e. $v \in X_i$ and $v \notin X_j$ where $j < i$. Then bag X_i contains all v's predecessors.*

Proof. From (*iii*) in Definition 2.6.4 we know that all predecessors of v have to be in bag X_i or bag X_h where $h < i$:

- If the predecessor vertex $p \in X_i$, then the statement holds for p.
- If $p \in X_h$, where $h < i$: Since X_i is the bag where v appears for the first time in P, based on (*i*) in Definition 2.6.4 there exists a bag X_k, where $k \geq i$, containing vertices p and v. Based on (*ii*) in Definition 2.6.4 X_i contains $p \in X_h \cap X_k$.

□

Corollary 2.6.1 (from *Lemma 2.6.1*)**.** *The path-width of a directed graph G is at least the maximum indegree of G, where the indegree of a vertex v is the number of v's predecessors.*

The following path-decomposition is used by our algorithm, because it allows to process a single new vertex in each bag which simplifies the algorithm.

Definition 2.6.7 (*Incremental path-decomposition*)**.** Let $G = (V, E)$ be a DAG and $P = X_1, ..., X_n$ its path-decomposition from Definition 2.6.4. We consider $X_0 = \emptyset$. We call P an *incremental path-decomposition* if $|X_i \setminus X_{i-1}| = 1$ for $1 \leq i \leq n$.
The vertex in $X_i \setminus X_{i-1}$ is called the *incremental vertex.*

The incremental path-decomposition always can be constructed from an existing path-decomposition. This is proven in the lemma below:

Lemma 2.6.2. *Let $G = (V, E)$ be a DAG and P a path-decomposition from Definition 2.6.4. There exists an incremental path-decomposition for G with a width that is not higher than the width of P.*

Proof. Let us have a path-decomposition $P = X_1, ..., X_n$ from Definition 2.6.4. We show that it is always possible to expand P so that it becomes an incremental path-decomposition.

Let's assume that $|X_i \setminus X_{i-1}| = k$. If $k = 0$, it means that $X_i \subseteq X_{i-1}$ and X_i can be left out of the path-decomposition without breaking properties (*i*), (*ii*) and (*iii*) from Definition 2.16. If $k > 1$, we can create a path-decomposition $P' = X_1, ...X_{i-1}, N, X_i, ...X_n$ where $|N \setminus X_{i-1}| = 1$ and $|X_i \setminus N| = k - 1$. By repeating these steps, we get to an incremental path-decomposition.

Bag N can be always constructed, as there always exists a vertex $v \in X_i \setminus X_{i-1}$ such that $N = X_{i-1} \cap X_i \ \cup \ \{v\}$ and P' is still a valid path-decomposition.

Properties (i) and (ii) of Definition 2.6.4 stay valid for P' regardless of which vertex from $X_i \setminus X_{i-1}$ is chosen as v. Regarding property (iii): Since G is a DAG, the graph induced by vertices $X_i \setminus X_{i-1}$ is a DAG, therefore, there is a vertex in $X_i \setminus X_{i-1}$ that has no predecessors in $X_i \setminus X_{i-1}$. Choosing this as vertex v fulfills property (iii).
Also notice that $|N| \leq |X_i|$, i.e. the width of the path-decomposition was not increased. □

Corollary 2.6.2 (from *Lemma 2.6.2*). *The incremental path-decomposition of a DAG $G = (V, E)$ consists of $|V|$ bags.*

2.6.2 $O(|V| \cdot 2^{width} \cdot width)$-time algorithm

In this subsection, we describe an algorithm for solving the *Maximum-Score Disjoint Paths Problem* on a DAG G. The input to the algorithm is an incremental path-decomposition P of G. The algorithm runs in $O(|V| \cdot 2^{width} \cdot width)$ time where $width$ is the width of P.

Detailed description of the algorithm

Notation 5. Let $G = (V, E)$ be a weighted DAG with a weight function $w : V \to \mathbb{R}$. Let $P = X_1, ..., X_n$ be graph G's incremental path-decomposition with width $width$. Let *penalty* be a penalty value for selecting a path from G. Let $subG_i$ be the subgraph induced by vertices in $X_1 \cup ... \cup X_i$.

The algorithm walks through the sequence of bags X_i $(1 \leq i \leq n)$ in an incremental order and processes them, one at a time. For each bag X_i the algorithm calculates the valid assignments of values $\{end, not\_end\}$ to vertices of X_i. The meaning of these values assigned to a vertex v is the following:

- value *end*: v is on a selected path and is the last vertex of the selected path, where selected paths are considered only in subgraph $subG_i$;
- value *not_end*: otherwise. Note that vertex v can be part of a selected path or can be not selected at all.

A valid assignment means that it is possible to select paths such that the *end* and *not_end* values assigned to the vertices follow the definition of values $\{end, not\_end\}$ from above. The valid assignments for bag X_i are calculated based on the valid assignments for bag X_{i-1}. The valid assignments are stored in set T_i. The algorithm calculates the best possible score for an assignment based on the scores calculated for X_{i-1}. The best scores for each assignment are stored in the *scores* map. These scores represent the best scores achievable for subgraph $subG_i$.

The valid assignments and scores for bag X_i are calculated from X_{i-1} the following way:
Let v be the incremental vertex of X_i. For each valid assignment in T_{i-1}, the algorithm considers the following options:

1. option: Vertex v is not selected, i.e. assigns value *not_end* to it. The score remains the same as for the old assignment.
2. option: Vertex v is selected, i.e. assigns value *end* to it, and vertex v starts a new path. The score is increased by $w(v)$ and decreased by *penalty*.
3. option: Vertex v is selected, i.e. assigns value *end* to it, and vertex v continues an existing selected path. Based on Lemma 2.6.1 we know that all predecessors of v are in bag X_i, and since it is an incremental path-decomposition, all these predecessors are also present in X_{i-1}, i.e. in the assignments from T_{i-1}. The algorithm checks which predecessors have value *end* assigned to them, i.e. they end a selected path, and extends the path with vertex v by assigning value *not_end* to the predecessor and value *end* to v. The score is increased by $w(v)$.

The final score of the new assignment is going to be the maximum of all these options.
The maximum score for G is the maximum score calculated for the final bag.

See the algorithm pseudo-code in Algorithm 1.

Algorithm 1

```
function GetScores(path_decomp $P$, graph $G$, int $penalty$, weights $w$)
    $T_0 = \{empty\_assignment\}$
    $score[0][empty\_assignment] = 0$

    for $bag$ in 1 ... $length(P)$ do
        // $v$ is the incremental vertex in bag $X_i$
        $v = P.X_{bag} - P.X_{bag-1}$
        for $assignment$ in $T_{bag-1}$ do
            // Option 1: $v$ is not selected
            $assignment\_1 = assignment \cap P.X_{bag} + [v : not\_end]$
            $T_{bag} \mathrel{+}= assignment\_1$
```

```
                $score[bag][assignment\_1] = max($
                    $score[bag-1][assignment],$
                    ValueOrZero($score[bag][assignment\_1]$))

                // Option 2: $v$ is selected and starts a new path
                $assignment\_2 = assignment \cap P.X_{bag} + [v : end]$
                $T_{bag}$ += $assignment\_2$
                $score[bag][assignment\_2] = max($
                    $score[bag-1][assignment] + w(v) - penalty,$
                    ValueOrZero($score[bag][assignment\_2]$))

                // Option 3: $v$ is selected and continues an existing path
                for $p$ in $predecessors(G, v)$ do
                    if $assignment.p \neq end$ then continue
                    $assignment\_3 =$
                        $assignment \cap P.X_{bag} + [v : end] - p + [p : not\_end]$
                    $T_{bag}$ += $assignment\_3$
                    $score[bag][assignment\_3] = max($
                        $score[bag-1][assignment] + w(v),$
                        ValueOrZero($score[bag][assignment\_3]$))
                end for
            end for
        end for
        return $max(score[length(P)])$
    end function

    function ValueOrZero(value $value$)
        if $value$ is defined then return $value$
        return 0
    end function
```

Path reconstruction

To reconstruct the paths, the algorithm also needs to save how the maximum scores were derived. Whenever the algorithm saves a maximal score for an assignment, it stores the assignment from the previous bag that was used to derive the current assignment and the option used to get the current assignment. For option 3 it also saves which vertex is the predecessor on the path of the incremental vertex. This can be used to trace back from the highest scored assignment of the last bag until the assignments of the first bag and the vertex selections that were made.

Complexity

Function GetScores in Algorithm 1 contains 3 nested loops. The first iterates over each bag. Based on Corollary 2.6.2, we know that the incremental path-decomposition has $|V|$ bags. The second for loop iterates over possible assignments in a bag, there are 2^{width} different assignments. The 3rd for loop iterates through the predecessors of the vertex, there are at most $width$ predecessors. The $assignment \cap P.X_{bag}$ can be calculated in the second for loop, and can be done in $O(width)$ time. Thus the complexity of the entire algorithm is $O(|V| \cdot 2^{width} \cdot width)$.

2.6.3 Creating an incremental path-decomposition

As you might have noticed, our algorithm gets the incremental path-decomposition as an input. In this section, we describe how to create an incremental path-decomposition for a DAG G, although, not necessarily the one with the smallest width.
Description of the algorithm:

1. Let $v_1, ..., v_n$ be the topological sort of DAG G. Put these vertices into subsequent bags, i.e. $X_i = \{v_i\}$. These bags already fulfill property (iii) from Definition 2.6.4.

2. From Lemma 2.6.1 we know that the bag where a vertex v appears for the first time also contains all v's predecessors. In our case, in each bag X_i a new vertex v_i appears, therefore, add all predecessors of v_i into bag X_i. This does not break property (iii) from Definition 2.6.4 and it fulfills property (i).

3. To fulfill property (ii) in Definition 2.6.4, find the first and last occurrence of each vertex v in the bags, and add vertex v into the bags in-between. This does not break property (i) and (iii) from Definition 2.6.4 and it fulfills property (ii).

This path-decomposition is an incremental path-decomposition. Bag X_i is the first bag where vertex v_i appears. Therefore, $|X_i \setminus X_{i-1}| \geq 1$. The difference cannot be 2 or more, as then the other additional vertex has to be v_j where $j < i$ which means it appeared already in bag X_j, and due to condition (ii) from Definition 2.6.4 it means $v_j \in X_{i-1}$.

In Figure 2.17 and 2.18, we can see two different incremental path-decompositions of the same DAG based on our algorithm. Next to each vertex is its order in a topological ordering which differs in the two figures. Below the graph, we can see the incremental path decomposition, i.e. bags $X_1, ..., X_8$. Each bag is divided into 3 parts with a horizontal line, so it is easy to see which vertices were added in which step of the algorithm (the top part corresponds to the first step, the middle part to the second step, and the bottom part to the third step of the algorithm). Notice that the width of the path-decomposition depends on the topological order of the vertices from the first step. The width of the path-decomposition in Figure 2.17 is 2 which is minimal based on Corollary 2.6.1. The width of the second path-decomposition in Figure 2.18 is 4.

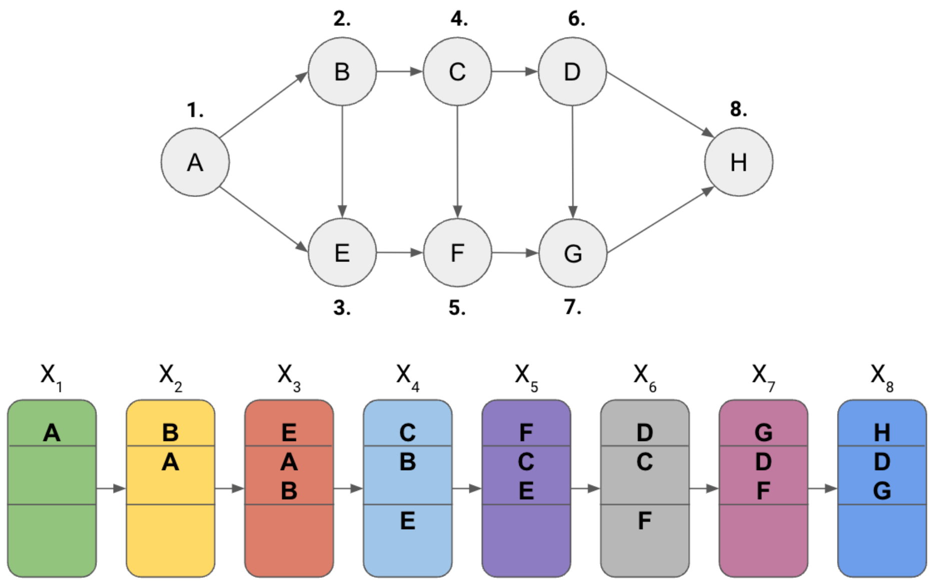

Figure 2.17: *Incremental path-decomposition of a DAG based on our algorithm.*

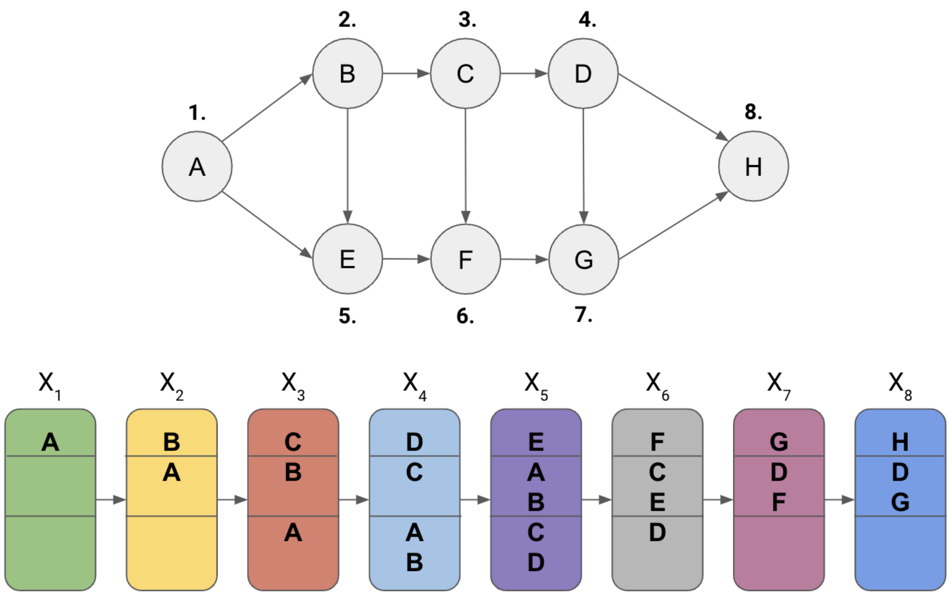

Figure 2.18: *Same DAG as in Figure 2.17, but the incremental path-decomposition is different.*

Complexity

Regarding the complexity of the algorithm for the incremental path-decomposition: step 1 can be done in $O(|V| + |E|)$ time, step 2 can be done in $O(|V|)$ time, and step 3 can be done in $O(|V| \cdot width)$ where *width* is the width of this path-decomposition. The time complexity of this algorithm is $O(|V| \cdot width + |E|)$.

Chapter 3

Experiments

The GC-content of DNA sequences, i.e. the percentage of guanine (G) and cytosine (C) bases, is a frequently used statistic when analysing genomes. It has been well studied across organisms, revealing connections between GC-content and various genomic characteristics [30]. We decided to do an experiment where we search for GC-rich regions in pan-genomes of bacteria *Escherichia coli* (or *E. coli*) represented as elastic-degenerate strings [21]. As mentioned before, elastic-degenerate strings correspond to n-layered bubble graphs, see more in Figure 2.11 and the corresponding explanation. Therefore, we ran the experiments using the the n-layered bubble graph algorithm described in Section 2.5.

3.1 Gathering data

We used the complete genome of *E. coli K12-MG1655* as the reference genome [4], and some reads (i.e. genomic sequences, small sections of DNA) from a study [32] stored in the European Nucleotide Archive (ENA) database under the project PRJNA563564 [12]. Out of these genomic sequences we created Variant Call Format (VCF) files containing up to 8 E. coli genomes. The VCF files store genomic sequence variations. To align the sequences and to create the VCF files we used BWA [24], SAMtools [25] and Freebayes [17] tools. To construct the elastic-degenerate strings from the VCF files, we used the EDSO [31] tool. As a result we got an elastic-degenerate string pan-genome. Our tool transforms this to an n-layered bubble graph where vertices are single bases (see Figure 2.11) before executing the n-layered bubble graph algorithm.

Note that the VCF files, besides variations, also contain fields with statistics (e.g. describing quality or sequencing depth) which can be useful to distinguish true from false variants and filter them out [34]. However, in our experiment we were more interested in how the number of bubbles influences the scores and selected paths. Therefore, we did not perform any filtering.

In Table 3.1 we can see which genomic sequences are included in the n-layered bubble graph besides the reference genome, and the number of vertices in the n-layered bubble graph.

Graph	Included genomic sequences of E. coli [12]	Number of vertices
ID 0	only the reference genome [4]	4 641 654
ID 1	ID 0 and SRR10058833	4 686 570
ID 2	ID 1 and SRR10058834	4 743 566
ID 3	ID 2 and SRR10058835	4 768 722
ID 4	ID 3 and SRR10058836	4 769 070
ID 5	ID 4 and SRR10058837	4 769 264
ID 6	ID 5 and SRR10058838	4 883 827
ID 7	ID 6 and SRR10058839	4 883 922
ID 8	ID 7 and SRR10058840	4 884 102

Table 3.1: *Genomic sequences included in the n-layered bubble graph besides the reference genome*

In Figure 3.1 we can see how many 2, 3, 4 and 5-layered bubbles there are in each of the n-layered bubble graphs from Table 3.1. We can observe that the number of bubbles increased rapidly when the first two genomic sequences were added, i.e. *SRR10058833* and *SRR10058834*. Afterwards the growth slowed down. However, notice the significant jump in the number of bubbles in graph *ID 6*. Genomic sequence *SRR10058838* seems to significantly differ from the previous genomic sequences, therefore, introducing new genomic variations which resulted in new bubbles. This added diversity can be also seen in Table 3.1 in the significant growth of vertices for graph *ID 6* compared to graph *ID 5*.
Note: we can see that e.g. graph *ID 1* has some 3-layered bubbles which does not make sense, since it was constructed out of 2 genomic sequences. One possible explanation could be that when creating the VCF file, the tool Freebayes might find more variants for a position from the reads.

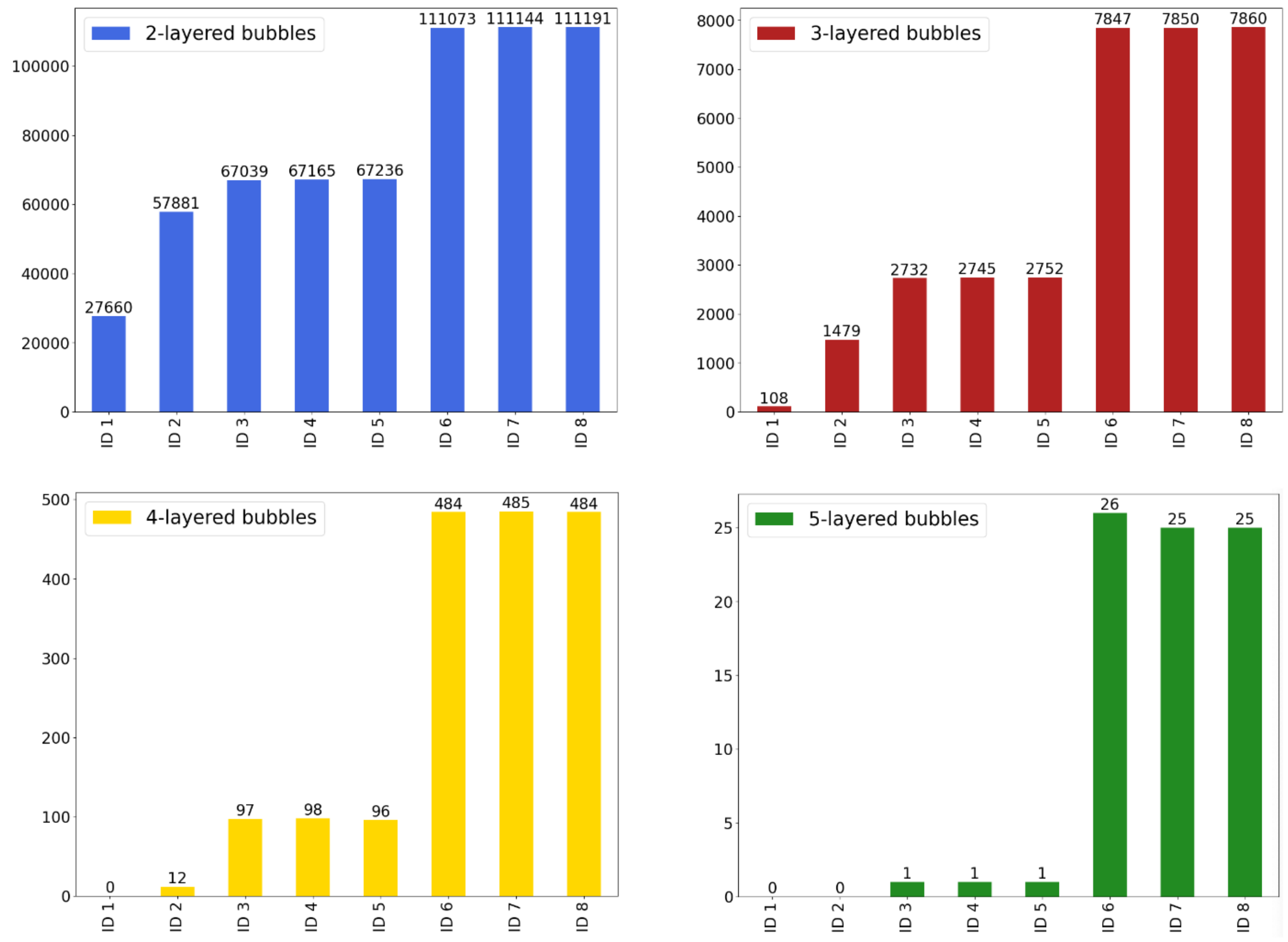

Figure 3.1: *Number of 2,3,4 and 5-layered bubbles in each n-layered bubble graph from table 3.1. On x-axis we can see the identification of the n-layered bubble graph, while on y-axis the number of b-layered bubbles is displayed.*

3.2 Results

We ran the algorithm for n-layered bubble graphs. To find paths with high GC-content we used the following scoring:

- weight of bases G and C is 1,
- weight of bases A and T is -2,
- $penalty \in \{5, 6, 7, 8, 9, 10\}$.

The GC-content of E. coli's genome is 50.8% on average. These weights mean that the GC-content of a selected path is at least 66%. The penalty ensures that the length of the path is at least *penalty*.

It is possible to define more sophisticated scoring schemes based on probabilistic models [9]: In their study, they were searching for segments that maximise the log-likelihood ratio. The likelihood of the G, C and A, T bases was calculated for the whole genome and for known biologically significant regions. The score of each base was the difference of logarithms of the two likelihoods.

In Figure 3.2, we can see the coverage, i.e. the percentage of the graph that is covered by the selected paths. We can see that if the penalty is increased, then the coverage is reduced. This is an expected behaviour, as some selected paths might get a negative score for a larger penalty.

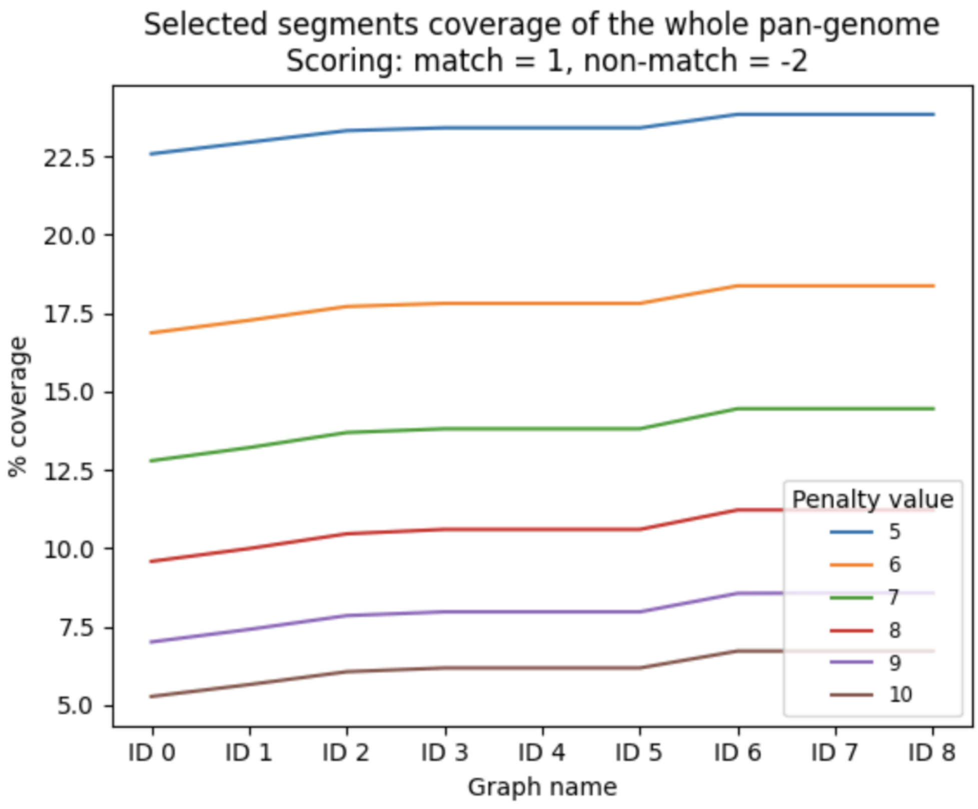

Figure 3.2: *Selected paths coverage percentage for pan-genomes from Table 3.1*

In Appendix A, we listed more detailed results. Notice that by adding genomic sequences to the pan-genome, the coverage and the score is increasing. This is because by adding new genomic sequences, we introduce new variants with richer GC content, therefore, some parts of the selected paths might continue with vertices on a newly added layer in a bubble, or new selected paths may appear.

Conclusion

In this work, we looked at the problem of finding biologically meaningful regions in a graphical pan-genome. We translated this problem into a mathematical problem of finding *Maximum-Score Disjoint Paths*. We proposed an $O(|V|)$-time algorithm on n-layered bubble graphs which correspond to elastic-degenerate strings, and an $O(|V| \cdot 2^{width} \cdot width)$-time algorithm on directed acyclic graphs where $width$ is the width of the incremental path-decomposition of the graph. We implemented the algorithm on n-layered bubble graphs and did some simple experiments for locating GC-rich regions in pan-genomes of bacteria E. coli.

Future work

There are multiple ways our work can be improved or extended.

Regarding the $O(|V|)$-time dynamic programming algorithm on n-layered bubble graphs, one could make the algorithm more efficient. As the goal is to maximise the score of the selected paths, areas with many vertices with negative weights will not be be selected into paths. Therefore, the algorithm could identify these regions and skip them without calculating the scores for these vertices.

Close to the deadline of the thesis submission, we learned about the class of directed two-terminal series-parallel graphs (see Subsection 2.5.4). This is a broader class of graphs to which our algorithm on n-layered bubbles could be extended to.

The $O(|V| \cdot 2^{width} \cdot width)$-time algorithm on directed acyclic graphs gets an incremental path-decomposition of the graph as an input. We described an algorithm that creates an incremental path-decomposition, however, we also showed that this path-decomposition does not have to be optimal for some cases, i.e. the width of the path-decomposition might not be minimal. By using an algorithm that creates a minimal or close to minimal path-decomposition for directed graphs, our algorithm would be more efficient. (Note that we showed that we can create an incremental-path decomposition from a path-decomposition.)

Regarding experiments, we only performed a simple experiment to find GC-rich regions and compared properties related to the score and selected paths. These experiments could be extended by analysing whether the selected regions of the pan-genome represent some functional elements of the organism. Furthermore, more sophisticated scoring could be used based on probabilistic methods. Another experiment we had in our mind was aligning longer reads by finding maximum scoring paths. This could be done by mapping k-mers of a long read onto the pan-genome. Areas of the pan-genome that have k-mers matched to them would have positive score. Therefore, finding maximum scoring paths could correspond to aligning the long read to the pan-genome.

Appendix A

Results of the n-layered bubble graph algorithm on E. coli pan-genome graph with genomic sequences from Table 3.1 with scoring:

- $w(G) = w(C) = 1$
- $w(A) = w(T) = -2$
- $penalty \in \{5, 6, 7, 8, 9, 10\}$.

The columns encode the graph's name, the score calculated by the algorithm, the number of selected paths, the ratio of the graph that is covered by the selected paths and the average length of a selected path.

penalty = 5				
Graph ID	score	#paths	path cover	avg path len
0	156871	92235	22.58%	11.36
1	163033	93450	22.95%	11.51
2	170302	94800	23.32%	11.67
3	172576	95234	23.41%	11.72
4	172598	95240	23.41%	11.72
5	172615	95244	23.41%	11.72
6	183691	97457	23.84%	11.94
7	183709	97460	23.84%	11.94
8	183722	97458	23.84%	11.95

penalty = 6

Graph ID	score	#paths	path cover	avg path len
0	99844	55109	16.87%	14.21
1	104738	56222	17.27%	14.39
2	110574	57503	17.71%	14.61
3	112448	57853	17.81%	14.68
4	112464	57858	17.81%	14.68
5	112476	57864	17.81%	14.68
6	121365	59828	18.37%	14.99
7	121384	59828	18.37%	14.99
8	121396	59828	18.37%	15.00

penalty = 7

Graph ID	score	#paths	path cover	avg path len
0	64391	34777	12.79%	17.07
1	68247	35701	13.21%	17.35
2	72827	36816	13.69%	17.64
3	74333	37152	13.81%	17.73
4	74344	37157	13.81%	17.73
5	74352	37162	13.81%	17.73
6	81328	38936	14.45%	18.12
7	81344	38938	14.45%	18.12
8	81350	38943	14.45%	18.12

penalty = 8

Graph ID	score	#paths	path cover	avg path len
0	41925	22083	9.58%	20.14
1	44969	22834	9.99%	20.51
2	48568	23723	10.46%	20.92
3	49766	24010	10.60%	21.05
4	49774	24012	10.60%	21.05
5	49781	24011	10.60%	21.05
6	55235	25447	11.22%	21.53
7	55250	25448	11.22%	21.53
8	55255	25447	11.22%	21.53

penalty = 9

Graph ID	score	#paths	path cover	avg path len
0	27806	13980	7.01%	23.28
1	30216	14596	7.41%	23.78
2	33042	15328	7.85%	24.28
3	34004	15562	7.97%	24.44
4	34010	15563	7.97%	24.44
5	34016	15563	7.97%	24.44
6	38290	16716	8.56%	25.02
7	38301	16720	8.57%	25.02
8	38309	16717	8.57%	25.03

penalty = 10

Graph ID	score	#paths	path cover	avg path len
0	18468	9266	5.27%	26.39
1	20327	9783	5.65%	27.05
2	22525	10386	6.06%	27.69
3	23285	10578	6.18%	27.85
4	23290	10578	6.18%	27.85
5	23296	10576	6.18%	27.86
6	26613	11497	6.72%	28.54
7	26621	11500	6.72%	28.54
8	26631	11499	6.72%	28.55

Appendix B

The source code of the algorithm on n-layered bubble graphs and the input data is available at `https://github.com/evicy/thesis`.

An alternative way for viewing the source code is to view it from the attached disc.

Bibliography

[1] Computational pan-genomics: status, promises and challenges. *Briefings in bioinformatics*, 19(1):118–135, 2018.

[2] János Barát. Directed path-width and monotonicity in digraph searching. *Graphs and Combinatorics*, 22(2):161–172, 2006.

[3] Fredrik Bengtsson and Jingsen Chen. *Computing maximum-scoring segments optimally.* Luleå tekniska universitet, 2007.

[4] Bethesda (MD): National Library of Medicine (US), National Center for Biotechnology Information. Assembly ASM584v2, Escherichia coli str. K-12 substr. MG1655 (E. coli). `https://www.ncbi.nlm.nih.gov/assembly/GCF_000005845.2/`, 2013. Accessed: 2023-04-10.

[5] Etienne Birmelé, Pierluigi Crescenzi, Rui Ferreira, Roberto Grossi, Vincent Lacroix, Andrea Marino, Nadia Pisanti, Gustavo Sacomoto, and Marie-France Sagot. Efficient bubble enumeration in directed graphs. In *String Processing and Information Retrieval: 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings 19*, pages 118–129. Springer, 2012.

[6] Kai-Min Chung and Hsueh-I Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34(2):373–387, 2005.

[7] Alessandro Coppe, Gian Antonio Danieli, and Stefania Bortoluzzi. REEF: searching REgionally Enriched Features in genomes. *BMC bioinformatics*, 7(1):1–7, 2006.

[8] Nicholas J Croucher, Andrew J Page, Thomas R Connor, Aidan J Delaney, Jacqueline A Keane, Stephen D Bentley, Julian Parkhill, and Simon R Harris. Rapid phylogenetic analysis of large samples of recombinant bacterial whole genome sequences using Gubbins. *Nucleic acids research*, 43(3):e15–e15, 2015.

[9] M. Csuros. Maximum-scoring segment sets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(4):139–150, 2004.

[10] Aimée M Deaton and Adrian Bird. CpG islands and the regulation of transcription. *Genes & development*, 25(10):1010–1022, 2011.

[11] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[12] ENA. Project: PRJNA563564. `https://www.ebi.ac.uk/ena/browser/view/PRJNA563564?show=reads`. Accessed: 2023-04-10.

[13] David Eppstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98(1):41–55, 1992.

[14] Joshua Erde. Directed path-decompositions. *SIAM Journal on Discrete Mathematics*, 34(1):415–430, 2020.

[15] Francesco Ferrari, Aldo Solari, Cristina Battaglia, and Silvio Bicciato. Preda: an R-package to identify regional variations in genomic data. *Bioinformatics*, 27(17):2446–2447, 2011.

[16] Robert D Fleischmann, Mark D Adams, Owen White, Rebecca A Clayton, Ewen F Kirkness, Anthony R Kerlavage, Carol J Bult, Jean-Francois Tomb, Brian A Dougherty, Joseph M Merrick, et al. Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. *science*, 269(5223):496–512, 1995.

[17] Erik Garrison and Gabor Marth. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907*, 2012.

[18] Paweł Gawrychowski and Patrick K Nicholson. Encodings of range maximum-sum segment queries and applications. In *Combinatorial Pattern Matching: 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29–July 1, 2015, Proceedings 26*, pages 196–206. Springer, 2015.

[19] Jens Gustedt. On the Pathwidth of Chordal Graphs. *Discret. Appl. Math.*, 45(3):233–248, 1993.

[20] Zihuai He, Bin Xu, Joseph Buxbaum, and Iuliana Ionita-Laza. A genome-wide scan statistic framework for whole-genome sequence data analysis. *Nature communications*, 10(1):3018, 2019.

[21] Costas S Iliopoulos, Ritu Kundu, and Solon P Pissis. Efficient pattern matching in elastic-degenerate strings. *Information and Computation*, 279:104616, 2021.

[22] Sung Kwon Kim, Jung-Sik Cho, and Soo-Cheol Kim. Path Maximum Query and Path Maximum Sum Query in a Tree. *IEICE TRANSACTIONS on Information and Systems*, 92(2):166–171, 2009.

[23] Martin Kulldorff. Spatial scan statistics: models, calculations, and applications. In *Scan statistics and applications*, pages 303–322. Springer, 1999.

[24] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[25] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[26] Wentian Li, Pedro Bernaola-Galván, Fatameh Haghighi, and Ivo Grosse. Applications of recursive segmentation to the analysis of DNA sequences. *Computers & chemistry*, 26(5):491–510, 2002.

[27] Hsiao-Fei Liu and Kun-Mao Chao. Algorithms for finding the weight-constrained k longest paths in a tree and the length-constrained k maximum-sum segments of a sequence. *Theoretical computer science*, 407(1-3):349–358, 2008.

[28] Kchouk Mehdi, Jean-Francois Gibrat, and Mourad Elloumi. Generations of sequencing technologies: from first to next generation. *Electromagnetic Biology and Medicine*, 9(3):8–p, 2017.

[29] Joseph Outten and Andrew Warren. Methods and Developments in Graphical Pangenomics. *Journal of the Indian Institute of Science*, 101(3):485–498, 2021.

[30] Allison Piovesan, Maria Chiara Pelleri, Francesca Antonaros, Pierluigi Strippoli, Maria Caracausi, and Lorenza Vitale. On the length, weight and GC content of the human genome. *BMC research notes*, 12(1):1–7, 2019.

[31] Solon P. Pissis and Ahmad Retha. Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line. In

Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Source code is available at `https://github.com/webmasterar/edso`.

[32] Cameron J Reid, Khald Blau, Sven Jechalke, Kornelia Smalla, and Steven P Djordjevic. Whole genome sequencing of Escherichia coli from store-bought produce. *Frontiers in microbiology*, 10:3050, 2020.

[33] Neil Robertson and Paul D Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[34] Samtools. Filtering and handling VCFs — speciationgenomics.github.io. `https://speciationgenomics.github.io/filtering_vcfs/`. [Accessed 01-May-2023].

[35] Elena D Stavrovskaya, Tejasvi Niranjan, Elana J Fertig, Sarah J Wheelan, Alexander V Favorov, and Andrey A Mironov. StereoGene: rapid estimation of genome-wide correlation of continuous or interval feature data. *Bioinformatics*, 33(20):3158–3165, 07 2017.

[36] Nikola Stojanovic, Liliana Florea, Cathy Riemer, Deborah Gumucio, Jerry Slightom, Morris Goodman, Webb Miller, and Ross Hardison. Comparison of five methods for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, 27(19):3899–3910, 10 1999.

[37] Yunhao Wang, Yue Zhao, Audrey Bollas, Yuru Wang, and Kin Fai Au. Nanopore sequencing technology, bioinformatics and applications. *Nature biotechnology*, 39(11):1348–1365, 2021.

[38] C-Ting Wu and Jay C Dunlap. *Homology Effects: Volume 46 - Advances in Genetics*. Elsevier Science Publishing Co Inc, 2002.