ON THE RELATIONSHIP BETWEEN DE BRUIJN AND VARIATIONS GRAPHS

Advanced Algorithms

Santi Enrico
University of Udine
Academic year 2023/24

Contents

1 Introduction

Throughout the years several data structures have been described to effectively represent collections of strings allowing specific operations on such strings to be performed efficiently[1]. The objective of this work is to provide an introduction to de Brujin and variation graphs, give an overview of the work presented in [1] and try to extend such work by giving more insights on the desired properties of such graphs and possibly how we can derive graphs with those properties. The context on which we are focusing is the one of *graph pangenome*[2] [2].

2 Problem presentation

Consider the problem of transforming a data structure into a different one while preserving different characteristics of the data described. Such problem can be important for several reasons, for example because a data structure can be more efficiently used to perform certain operations, while another one may allow for an easier and more clear representation of the information. For this reason we may be interested in switching between the two representations. The contribution of [1] was to provide a procedure for transforming a de Bruijn graph into a variation one (while also axiomatizing the description of particular class of variation graphs).

2.1 String graphs

As in [1] we start by introducing the general idea used to represent a set of strings via a (string) multi-graph, such graphs will encode the commonalities and differences among a collection of genomes [2].

Fixed an alphabet Σ, a string multi-graph is a triple $G = \langle V, E, l \rangle$ where V and E are respectively a set of nodes[3] and directed edges, while $l : V \longrightarrow \Sigma^+$ is a labelling function which given a node returns a non empty string. Such function is used to associate strings to nodes, though strings represented by a single node are usually just a small substring of a string $s_i \in S$, (where $S = \{s_1..s_n\}$ is the set of strings we want to depict). For this reason the function l must be *extendable* in order to capture strings represented by paths[4] in the graph.

2.1.1 Subpath-comaptible extension of l

Before jumping into the extension of l, namely $\hat{l}$, some basic notions on paths must be recalled.

Given a path $p = \langle v_1, ..., v_n \rangle$ its length describes the number of edges traversed by the path (i.e. $n - 1$ for a path with n nodes) [3]. The set of intervals of a given path p, are all the couples of indexes denoting two nodes in the path $Int(p) = \{\langle i, j \rangle | 1 \leq i \leq j \leq n\}$. The last notion needed is the one of subpath of p, a subpath defined over $\langle i_1, i_2 \rangle$ is a path contained in p which starts at node v_{i_1} and ends at v_{i_2}, it's denoted by $p[i_1...i_2] = \langle v_{i_1}, ..., v_{i_2} \rangle$.

We can now define $\hat{l}$ as the function that given a subpath of p returns the string denoted by that subpath, such fucntion satisfies the property (*subpath-compatibility*):

[1]An example is the suffix tree which allows to represent in linear space all the suffixes of a string, allowing to perform an efficient search for the LCS (longest common substring).

[2]A graph-based representation of multiple genomes.

[3]Or Vertices, though we will always refer to them as nodes from now on.

[4]An ordered sequence of adjacent nodes.

$$\hat{l}(p[i_1..i_2]) = \hat{l}(p)[i'_1..i'_2]$$

2.1.2 How to represent string sets

Given a set of strings $S = \{s_1, ..., s_n\}$ what characteristics a string graph $G = \langle V, E, l\rangle$ has to have in order to represent S?
There must exists a function $\pi : S \longrightarrow \mathcal{P}(G)$ (where $\mathcal{P}(G)$ is the set of all paths in G) such that:

- $\hat{l}(\pi(s_i)) = s_i \quad \forall i \in \{1, ..., n\}$, which means that the string we read by traversing the path describing s_i is s_i itself. Note that this property is central in describing S. If this property wasn't required then a string graph would need to indicate also a set of paths denoting the genomes $\{s_1..s_n\}$ we want to describe in the graph, since the graph could present paths describing genomes not in $\{s_1..s_n\}$.
- $\forall v \in V,\ \exists i : v \in \pi(s_i)$, meaning that we don't want G to have nodes which aren't present in any useful path describing a string.
- A characteristic similar to the previous one but for the edges. For each edge in G it must join two consecutive nodes in some $\pi(s_i)$. Again notice that this requirement alone doesn't imply that G depicts only the strings in S.

The three properties mentioned above are necessary for representing a collection of strings, indeed $\langle G, \pi\rangle$ represents $S \iff$ the mentioned properties hold.
On the other hand there are also additional properties that deal with k-mers (strings of length k) we'd like our graph representation to satisfy:

k-completeness: *$\langle G, \pi\rangle$ is k-complete if every k-mer in S is represented by the same path in G.*
More formally, given $s_i, s_{i'} \in S$ which share a substring of length k (k-mer), i.e. $s_i[j..(j+k-1)] = s_{i'}[j'..(j'+k-1)]$, $\pi(s_i)$ and $\pi(s_{i'})$ must share a part of the path of the same length which depicts the k-mer. Thus we must have that $\pi(s_i)[p..(p+q)] = \pi(s_{i'})[p'..(p'+q)]$ and $\hat{l}(\pi(s_i)[p..(p+q)]) = s_i[j..(j+k-1)]$. This has to be verified for all k-mers.

k-faithfulness: A formal description of k-faithfulness requires first to introduce the notion of k-extendability. Consider two strings s_i,$s_{i'}$ and two nodes indexes (which refer to the same node v) in their paths, namely j in $\pi(s_i)$ and j' in $\pi(s_{i'})$, such that $\pi(s_{i'})[j'] = \pi(s_i)[j] = v$. We say that the pair $\langle i, j\rangle$ $\langle i', j'\rangle$ is *directly k-extendable* iff $\pi(s_i)[(j-m)..(j+m')] = \pi(s'_i)[(j'-m)..(j'+m')]$ for $m, m' \geq 0$ with the length of the string associated to those subpaths greater or equal than k.
The meaning of direct k-extendability is that two strings share a common node in their path iff they share a substring longer than k, otherwise, the node is duplicated.
There is then the notion of *k-extendability* which generalizes the previous one, namely $\langle i, j\rangle$ $\langle i', j'\rangle$ are k-extendable if it exists a sequence of occurrences of v, from $\langle i, j\rangle$ $\langle i', j'\rangle$ such that each consecutive occurrence in the sequence is directly k-extendable.

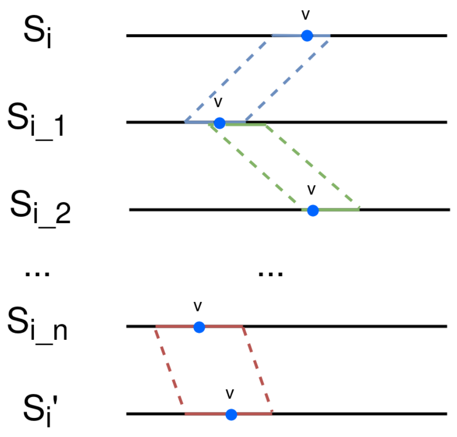

Figure 1: k-extendability for $\langle i, j\rangle$ $\langle i', j'\rangle$, occurrences of v.

The idea behind k-extendability is that given more nodes depicting the same substring in different paths describing strings, such nodes must be the same node only if all strings share a common substring around that node of length longer than k (see Appendix A).

$\langle G, \pi\rangle$ is k-faithful for S if all pairs $\langle i, j\rangle$ $\langle i', j'\rangle$ of the same nodes are k-extendable. The meaning of k-faithfulness is then to limit when nodes depicting the same substring can be merged into a single one.

We will now focus on two concrete types of string graphs, namely de Bruijn and variation graphs.

2.2 De Bruijn graphs

Fixed an integer k, we define a de Bruijn graph of length k as a string graph with the following conditions:

- $|l(v)| = k \ \forall v \in V$, the length of the string described by each node is a k.
- $l(v) = l(w) \implies v = w \ \ \forall v, w \in V$, in a de Bruijn graph there are no repeated nodes. This condition, implies that if all strings in S are longer than k the representation is k-complete (and k-faithful).
- $l(v)[2..k] = l(w)[1..(k-1)] \ \ \forall (v, w) \in E$, which indicates that v has an outgoing edge to w only if the last $k-1$ characters describing v are equal to the prefix of length $k-1$ of the string describing w.

We now need to describe $\hat{l}$ which, given a path $p = \langle v_1, .., v_n\rangle$ is defined as the concatenation of the k-mer described by v_1 and the last character of the k-mer describing each node v_i, namely $\hat{l}(p) = l(v_0) \cdot l(v_1)[k] \cdot ... \cdot l(v_n)[k]$.

One important choice when dealing with dBGs is the choice of the parameter k, since this can effectively impact the size of G. This choice can also influence the efficiency of the operations performed on G, for example, if k-mer lengths fit in a machine word, bit-wise operations and integer arithmetic can be performed on strings [4].

We conclude this section by recalling a Theorem from [1] which states the *uniqueness (up to isomorphism) of a k-de Bruijn graph for a set S* if each string is longer than k.

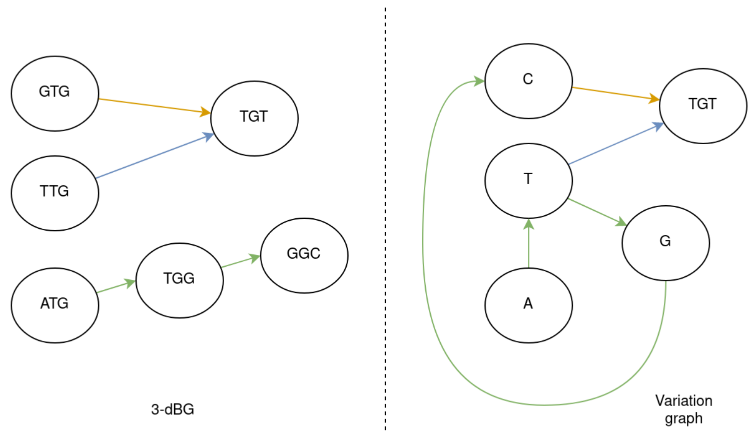

Figure 2: A comparison between *the* 3-dBG graph (3-complete and 3-faithfull) and *a* 3-complete variation graph for $S = \{GTGT, TTGT, ATGGC\}$

2.3 Variation graphs

A different way to concertize the notion of a string graph is presented by the variation graphs. A variation graph is a triple $G = \langle V, E, l \rangle$, but in contrast to dBGs $|l(v)|$ doesn't need to be constant and the generalization of l, $\hat{l}$ is the concatenation of the labels of each node in the path.
The subpath-compatibility of $\hat{l}$ still holds, consider a path $p = \langle v_1, ..., v_n \rangle$ and suppose we are interested in the label spelled out by the nodes in between node v_{i_1} and v_{i_2}, namely $\hat{l}(p[i_1..i_2])$. Since we have defined $\hat{l}$ to be the concatenation of the labels, we observe that the string read by the subpath $p[i_1..i_2]$ is the same as the one read by considering the string on the whole path and then skipping the first characters of the corresponding to the $0..i_1 - 1$ nodes, and early stopping after having read the characters up to the ones of v_{i_2}:

$$\hat{l}(p[i_1..i_2]) = \hat{l}(p)[\sum_{i=1}^{i_1-1} |l(v_i)| + 1 .. \sum_{i=1}^{i_2} |l(v_i)|]$$

We conclude this brief subparagraph by recalling that two variation graphs representations on a set S are equivalent if they present the same k-mers, $\forall k \geq 1$. In addition, given S, if two variation graphs representations $\langle G, \pi \rangle$ $\langle G', \pi' \rangle$ are both k-complete and k-faithful, then they are equivalent.

3 The transformation procedure

Having introduced dBGs and variation graphs, we now describe a procedure that given a dBGs representation $\langle G, \pi \rangle$ for S returns a corresponding variation graph representation.
Such algorithm adopts three subprocedures and uses an *intermediate graph representation*, called transition graphs, to store the partial results of the subprocedures.

In a nutshell, fixed an alphabet Σ (which is related to the set S of strings we want to describe) a transition graph is a triple $G = \langle V, E, l \rangle$ where the nodes now are used to denote single characters in Σ (i.e. $l : V \longrightarrow \Sigma$) and $E := E_V \cup E_B$. E_V and E_B are disjoint sets, the latter is the set of *de*

Brujin edges (B-edges) while the former is the set of *variation edges* (V-edges).

Before diving into the details of the three procedures we present a simple overview of the algorithm itself in Figure 3.

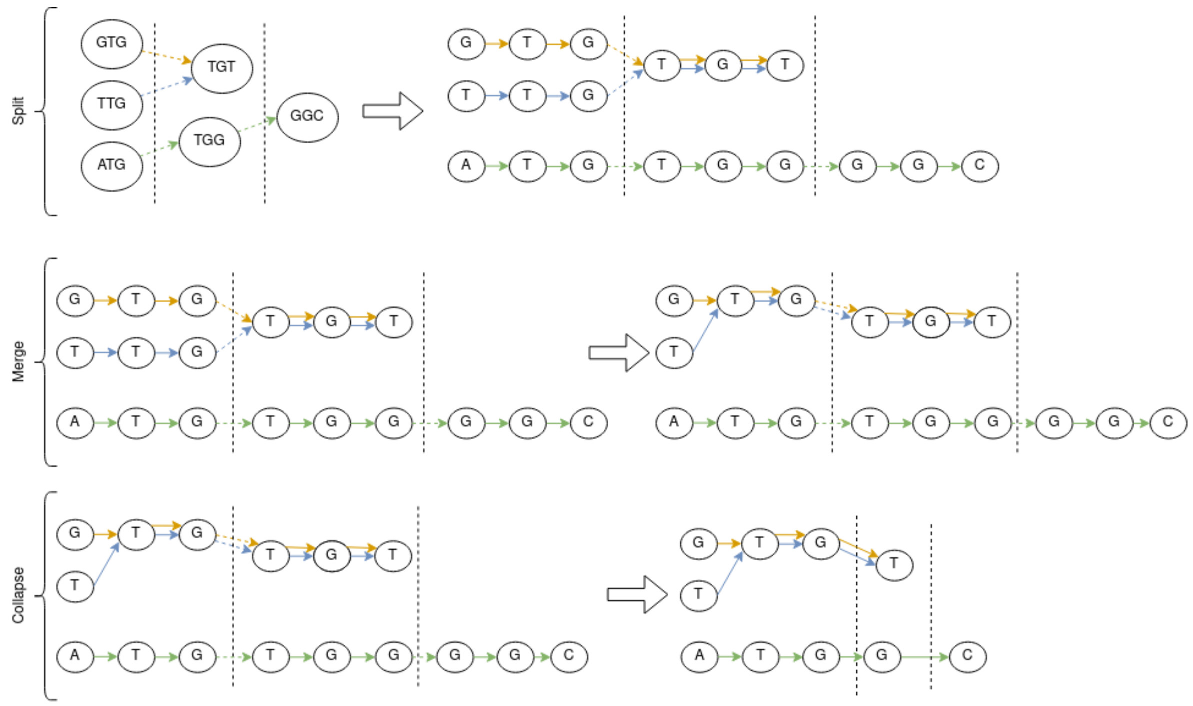

Figure 3: The results of the application of each procedure (in sequence) to the 3-dBG of Figure 2. On the left of each arrow the input structure for the corresponding procedure is found, while on the right the output is shown.

3.1 The algorithm

Having in mind the overview of the algorithm depicted in Figure 3 let's now dive into the procedures. Given in input the dBG representation the following procedures are invoked:

- **Split:** $\langle G, \pi \rangle \longrightarrow \langle G' = \langle V', E'_B, E'_V, l' \rangle, \pi' \rangle$, is the procedure that transforms the dBG in input into a transition graph. The idea is to split all the nodes representing k-mers (which we will call *k-nodes*) into linear subgraphs in which each node represents a single character (*single nodes*)[5]. These subgraphs are connected by B-edges, while the nodes composing them are connected by V-edges. As reported in [1] we have that $\langle G', \pi' \rangle$ represents S k-completely, k-faithfully and the consistency is maintained (i.e. the same strings are represented), the proof is immediate. More formally G' is defined as:
  - $V' = \bigcup_{v \in V} \{v_1, ..., v_k\}$, namely the new (single) nodes are all the nodes we obtain by splitting each k-node v into it's k components.
  - $E_V = \bigcup_{v \in V} \{\langle v_1, v_2 \rangle, ..., \langle v_{k-1}, v_k \rangle\}$, the set of the V-edges (which previously was the empty set since we had a dBG) consists in the union of all the edges connecting the k

[5]In [1] graphs in which all nodes are labelled with only a single label are referred as singular graphs.

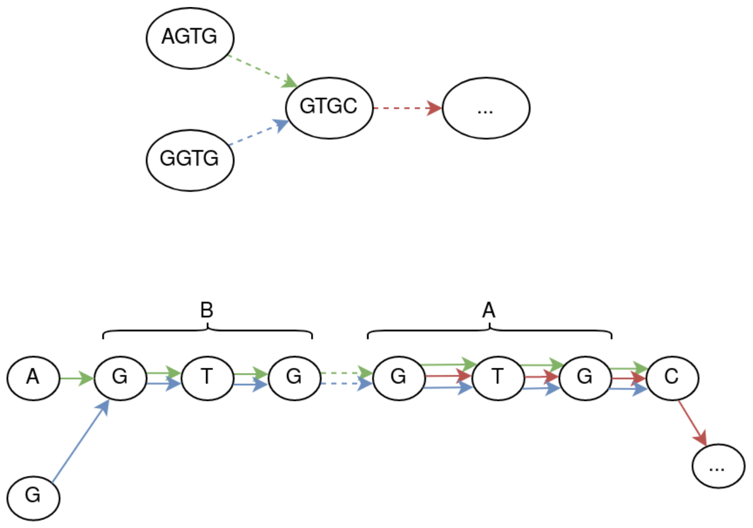

Figure 4: How the 4-dBG on top is transformed (after the Merge operation, before the Collapse) into a transition graph, highlighting the linear graphs A and B before and after the B-edge.

single nodes we obtained by splitting the original k-nodes.

  – $E_B = \{\langle v_k, w_1\rangle | \langle v, w\rangle \in E\}$, the set of the B-edges is restricted to those edges which connect single nodes that were obtained as the first single node and last single node as the result of the split procedure on two connected k-nodes.

  – $l'(v_i) = l(v)[i] \ \ \forall v \in V, j = 1..k$, namely the string (character) the new label function associates to a single node obtained as the $i - th$ node by the splitting of a k-node v, is the character in position i of the string obtained by the labelling function l on v.

- **Merge**: $\langle G' = \langle V', E'_B, E'_V, l'\rangle, \pi'\rangle \longrightarrow \langle G'' = \langle V'', E''_B, E''_V, l'\rangle, \pi''\rangle$[6]. The idea of this procedure is to merge (thus eliminating) redundant nodes introduced by the previous procedure. Redundant in this context means two linear subgraphs $\langle V = \{v_1, .., v_n\}, E = \{\langle v_1, v_2\rangle, .., \langle v_{n-1}, v_n\rangle\}\rangle$ and $\langle V = \{v'_1, .., v'_n\}, E = \{\langle v'_1, v'_2\rangle, .., \langle v'_{n-1}, v'_n\rangle\}\rangle$, sharing a parent[7] node and presenting single nodes with the same label (character) for each node. The function π is modified such that whenever a string s_i was mapped into a path p_i containing one merged node v_j, s_i will be mapped into p'_i in which the occurrences of v_j are replaced by those of the node resulting from the merging.

  Also this transformation preserves the consistency, k-completeness and k-faithfulness.

- **Collapse**: $\langle G'' = \langle V'', E'_B, E'_V, l'\rangle, \pi'\rangle \longrightarrow \langle G_{Var}, \pi_{Var}\rangle$, represents the last step of the algorithm and for this reason the output of this procedure must be a variation graph, not a transition graph. Having understood this, it's clear that during this phase all the B-edges will be removed. Consider a B-edge (let's call it E since it marks the *End* of a former k-node), by the consistency we have that the considered B-edge is followed and preceded by a linear subgraph describing a path of length k-1, see Figure 4.

  The key is to notice that these two linear subgraphs, which we will call B (for the subgraph coming *Before* the edge considered) and A (for the one coming *After* the B-edge) contain a

[6] Note that l' isn't modified.

[7] A node which has an in-going B-edges in both subgraphs.

sequence of nodes in which each pair $\langle A[i], B[i]\rangle$[8] for $i = 1..k-1$ satisfies $l'(A[i]) = l'(B[i])$. This means that we have two linear graphs whose nodes (ordered) are mapped to the same letter by l. The reason for this is that A and B are linear graphs resulting from the split of two adjacent k-dDB nodes. By definition of a k-dDB, we have that two adjacent nodes share a prefix and suffix of length $k-1$. By realizing this, it follows that the resulting variation graph shouldn't present the first $k-1$ nodes of A since the string represented by them has already been kept into account by reading the last $k-1$ nodes of B. The collapse procedure works by cases:

- **If G'' doesn't contains a cycle including a B-edge in which the subgraphs A and B overlap** then the actions to perform are the following: merge each pair $\langle A[i], B[i]\rangle$ into a node $C[i]$ and redirect all the incident edges of the pair on $C[i]$ after removing the B-edge (this avoids the creation of a cycle of length two involving E).

  The last step is to modify π' and $\hat{l}$ accordingly, i.e. consider each string $s \in S$ such that s was described by a path containing B and A, replace that part (i.e. B and A) by C [9].
- **Otherwise** it means that there exists a value $t \in 1..k-1$ for which $B[i] = A[t+i]\ \forall i \in 1..k-1$. In such case in addition to the operations performed in the previous case we merge the nodes $C[n]...C[2t+n]$ for $n = 1..t-1$. Also in this case π' should be slightly modified accordingly.

For what concerns the termination of the algorithm, it's based on the fact that each subprocedure terminates (since Split and Merge only iterate over all nodes once, while Collapse either removes a B-edge at each iteration, or stops). The correctness is also guaranteed [1].

4 Conclusions

The algorithm presented provides a relationship between k-complete and k-faithful variation graphs and k-dBGs, allowing among other things to transfer data between pangenome models based on different representations. In Appendix A and B the original part of the work done is presented, insights and equivalent formulations of the notions found in Section 2 are given.

[8] We depict linear subgraphs as arrays, so $A[i]$ indicates the $i-th$ node of A.

[9] The fact that a path that traversing B also traverses A comes from the idea that these used to represent the common $k-1$ characters of two adjacent k-nodes

Appendices

A Examples and wrong alternatives

Consider the string graph in Figure 7, by defining $\hat{l}$ as the concatenation of $l(v)$ on a given path, we have that the graph depicts the set $S = \{s_1, s_2, s_3, s_4\} = \{AAGC, AAGG, GAGT, GAGA\}$. Such string graph is 3-complete but not 3-faithful.

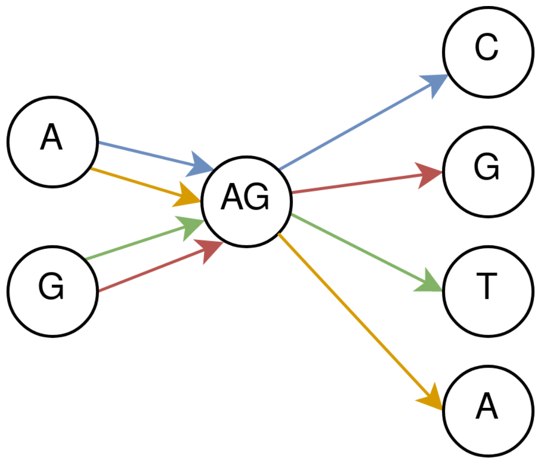

Figure 5: A 3-complete but not 3-faithful representation of S

The reason for which it's not 3-faithful is that the node "AG" shared by all four paths has four occurrencies: $\langle 1,2\rangle$, $\langle 2,2\rangle$, $\langle 3,2\rangle$,$\langle 4,2\rangle$, but not all pair of occurrencies share a 3-mer containing "AG" (e.g. $\langle 1,2\rangle$, $\langle 3,2\rangle$). In order to achieve the 3-faithfulness we need thus to duplicate the node "AG" as shown in Figure 6.

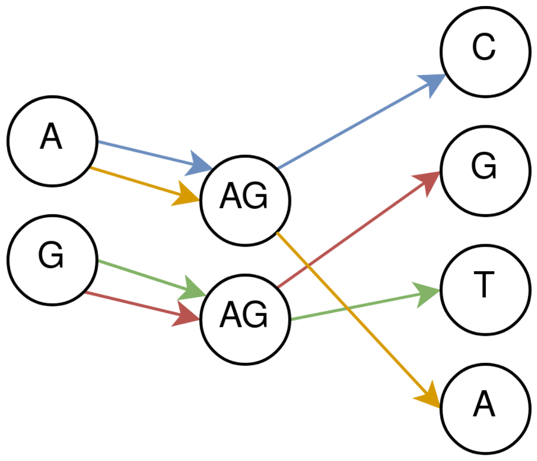

Figure 6: A 3-complete and 3-faithful representation of S

The idea of a graph representation being k-faithful and k-complete is thus a balance between trying to repeat less nodes[10] possible and not merging all the repeated substrings in S if the strings

[10]Nodes with the same label.

considered don't share at least a k-mer.

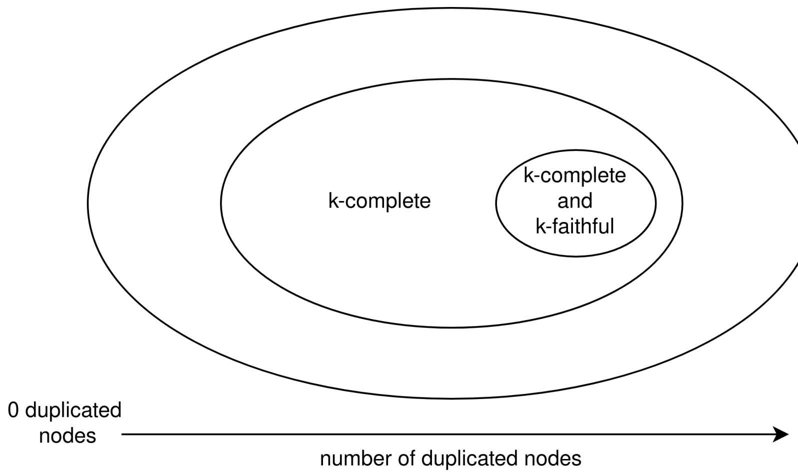

Figure 7: An overview of only the k-complete graphs (fixed k) for S. There's a boundary reachable by adding the correct duplicated nodes above which the representations become k-faithful. Note that if too much or the wrong nodes are added the graph representation could fall outside of the k-complete graphs.

A.1 An alternative

Playing around with the notion of k-extendability it could seem to be equivalent to the following definition.

Definition 1. *$\langle i, j\rangle$ $\langle i', j'\rangle$ are weakly k-extendable iff in the set of occurrences of v for each string s_l there's another string s'_l such that they are directly k-extendable.*

Though, while being introduced to try to simplify the notion of *k-extendability* it turned out to be weaker, thus non equivalent. Indeed the notion of *weakly k-extendability* indicates that a node can be shared by multiple string paths (or by the same string in different positions) if at least two share a k-mer around that node.

Definition 2. *$\langle G, \pi\rangle$ represents S weakly k-faithfully if every pair of a vertex is weakly k-extendable.*

Lemma A.1. *Let $\langle G, \pi\rangle$ be k-faithful then:*

- *$\langle G, \pi\rangle$ is k-complete.*
- *$\langle G, \pi\rangle$ is weakly k-faithful.*
- *$\langle G^-, \pi'\rangle$ is not k-faithful, where $G^- = G \setminus \{u : \exists u_1..u_r \in V \wedge l(u) = l(u_1) = .. = l(u_r)\}$ and π' redirects each in going edge on u to $u_1..u_r$ (not necessary all such edges are mapped on the same u_i), adding also the respective outgoing edges in order to describe the same set of strings.*

Proof. Given $\langle G, \pi\rangle$ k-faithful we first need to prove its also k-complete. Given two strings s_i and s'_i sharing a k-mer (since $\hat{l}(\pi(s_i)[j..j + m_1]) = \hat{l}(\pi(s'_i)[j'..j' + m_2])$) by k-extendability we have that $\pi(s_i)[j..j + m] = \pi(s'_i)[j'..j' + m]$, thus we have that the k-mer is described by the same path ($m_1 = m_2$ and we substitute it by m). The same reasoning can be extended to more than two

strings.
The proof of the second implication is immediate since given two or more vertices occurrences if they are k-extendable there they are also weakly k-extendable.
We now need to prove the third point. We know that in $\langle G, \pi\rangle$ G has at least a repeated node $(u, u_1, ..u_r)$, suppose wlog that only the path related to the string s_i passes through u, namely $\pi(s_i) = v_1, v_2..u..v_h$ (wlog assume it's not the first or last node). This means that the other paths of strings $s_{i_1}, .., s_{i_n}$ passing through $u_1..u_r$ don't share a k-mer containing $l(u)$ with s_i. From this last statement and the completeness of G we can derive that $l(u) < k$ otherwise s_i and $s_{i_1}, .., s_{i_n}$ would share a k-mer. Removing u and replacing its occurrence in $\pi(s_i)$ by a generic u_{r_j} i.e. creating $\pi'(s_i) = v_1, v_2..u_{r_j}..u_h$ makes the graph G^- not k-faithful since the strings in $s_{i_1}..s_{i_n}$ whose path passes through u_{r_j} don't share a k-mer around $l(u_{r_j})$. □

Note that $\langle G, \pi\rangle$ being *k-faithful* doesn't imply that $\langle G^-, \pi'\rangle$ is not complete, (again see Figure 7 and 6). Let's now see a lemma stating the opposite implication (vice versa).

Lemma A.2. *Let $\langle G, \pi\rangle$ be k-complete. If $\langle G^-, \pi'\rangle$ is not k-faithful, where $G^- = G \setminus \{u : \exists u_1..u_m \in V \wedge l(u) = l(u_1) = .. = l(u_m)\}$ and π' redirects each in going edge on u to $u_1..u_r$ (not necessary all such edges are mapped on the same u_i), adding also the respective outgoing edges in order to describe the same set of strings, then it's NOT guaranteed for $\langle G, \pi\rangle$ to be k-faithful.*

Proof. Consider the following counterexample:

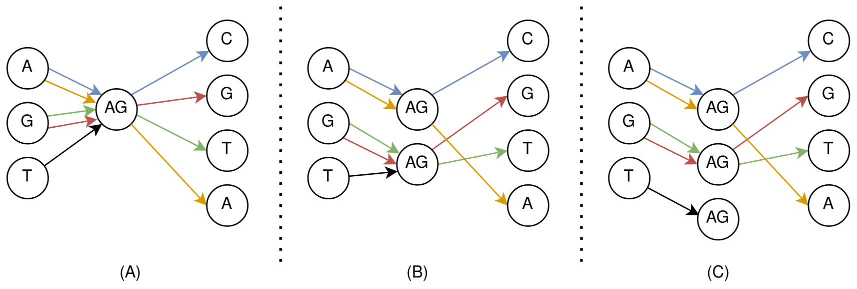

Figure 8: Duplicating the node "AG" twice to get a 3-faithful representation.

Consider the (B) string graph in Figure 8, it's 3-complete, and (A) is not 3-faithful, but (B) is not 3-faithful (the string "TAG" doesn't share a 3-mer with "GAGT" or "GAGA", though it shouldn't share a node with them). □

Lemma A.3. *$\langle G, \pi\rangle$ is weakly k-faithful $\implies$ $\langle G, \pi\rangle$ is k-complete*

Proof. Given two strings s_i and s'_i sharing a k-mer (since $\hat{l}(\pi(s_i)[j..j+m_1]) = \hat{l}(\pi(s'_i)[j'..j'+m_2])$) by directly k-extendability we have that $\pi(s_i)[j..j+m] = \pi(s'_i)[j'..j'+m]$, thus we have that the k-mer is described by the same path ($m_1 = m_2$ and call it m). The same reasoning can be extended to more than two strings. □

Lemma A.4. *$\langle G = \langle V = \{v_1..\}, E\rangle, \pi\rangle$ representing S is weakly k-faithful iff:*

- *Each path $\pi(s_i)$ that shares a subpath $v_1..v_m$ with another path $\pi(s_j)$ implies s_i shares with s_j a string containing $\hat{l}(v_{m_1}..v_{m_l})$ longer or equal than k and vice versa.*

Proof. ($\implies$) Given $\langle G, \pi \rangle$ weakly k-faithful we know by lemma A.3 that $\langle G, \pi \rangle$ is k-complete. From k-completeness we have that each k-mer is reflected by a common subpath, now we need to prove the double implication (iff):

- ($\implies$) Consider a path $\pi(s_i) = v_{i_1}, v_{i_2}..v_{i_p}, v_{m_1}..v_{m_l}..$ sharing $v_{m_1}..v_{m_l}$ in position $i_p + 1$ to $i_{p'}$ with $\pi(s_j) = v_{j_1}, v_{j_2}..v_{j_q}, v_m..v_{m'}..$, in position $j_q + 1$ to $j_{q'}$. Since $\langle G, \pi \rangle$ is k-faithful, by k-extendability we have that $\pi(s_i)[i_p + 1..i_{p'}] = \pi(s_j)[j_q + 1..j_{q'}] \implies |\hat{l}(pi(s_i)[i_p + 1..i_{p'}]) \geq k|$. The fact that $\hat{l}(pi(s_i)[i_p + 1..i_{p'}])$ contains $\hat{l}(v_{m_1}..v_{m_l})$ comes from the fact that the positions $i_p + 1$ to $i_{p'}$ are related to the nodes $v_{m_1}..v_{m_l}$.

- ($\impliedby$) Consider two strings, s_i and s_j sharing a string $sk = \hat{l}(v_{m_1}..v_{m_l})$ of length longer or equal than k. Since $\langle G, \pi \rangle$ is complete and $|sk| \geq k$ we have that sk is reflected by a common subpath, namely $\pi(s_i)[q..q + p] = \pi(s_j)[q'..q' + p]$. We now need to show $v_{m_1}..v_{m_l} = \pi(s_i)[q..q + p]$, so, for sake of contraddiction assume $v_{m_1}..v_{m_l} \neq \pi(s_i)[q..q + p]$ thus it must exists a different path $v_{i_1}..v_{i_t}$ shared by s_i and s_j describing $\hat{l}(v_{m_1}..v_{m_l})$. By k-completeness though we get the contradiction (since $\hat{l}(v_{m_1}..v_{m_l}) \geq k$) we have that a $v_{m_1}..v_{m_l} = v_{i_1}..v_{i_t}$.

($\impliedby$) Now given a string graph representation $\langle G, \pi \rangle$ for S in which each path $\pi(s_i)$ sharing a subpath $v_{m_1}..v_{m_l}$ with another path $\pi(s_j)$ implies s_i shares with s_j a string longer or equal than k containing $\hat{l}(v_{m_1}..v_{m_l})$ and vice versa, we want to show that G is weakly k-faithful. Consider a generic node v'_m in $v_{m_1}..v_{m_l}$ there will be at least two $\pi - occurrencies$ of such node, by the assumptions made, such occurrences are directly k-extendible. The reasoning can be generalized to three or more strings s_i, s_j sharing a subpath $v_{m_1}..v_{m'}..v_{m_l}$ and s_j, s_h sharing a subpath $v_{l_1}, v_{l_2}..v_{m'}..v_{l_t}$. $\square$

B From singular variation graphs to Pdor's algorithm

We now propose an alternative way[11] to derive from $S = \{s_1..s_n\}$ a k-complete and k-faithful (singular) variation graph representation without passing for a k-dBG. Such singular graph could subsequently be compacted into a non singular variation graph[12]. The algorithm we are going to present now is based on a suffix tree (slightly modified) works in linear time with respect to $n = \sum_{s_i \in S} |s_i|$ and no assumption is made on the alphabet size is $|\Sigma|$.

B.1 Single string idea

Let's first consider the case of $S = \{s_1\}$ and see the idea of the algorithm. Given S We can very easily construct the singular (linear) variation graph by creating for each character in s_1 a singular node and concatenate them.

Construct then the suffix tree of $s_1\$$ (\$ is added to have as many leaves as suffixes).
To see which nodes merge we need to consider the graph nodes whose label depicts paths from a root to an internal node (with depth $\geq k$ considering the length of the strings on the edges, which we will call *string depth*) from which two or more leaves are reachable. This because such paths indicate a common sub-string longer than k occurring two or more times in s_1. Lastly merge the corresponding nodes in the singular graph (with a bit of care, see the next subsection) and the variation graph obtained will be singular but $k - complete$ and $k - faithful$.

We will now consider some simple examples, then in B.2 a more formal description of the algorithm (as well as the data structures needed) will be presented.

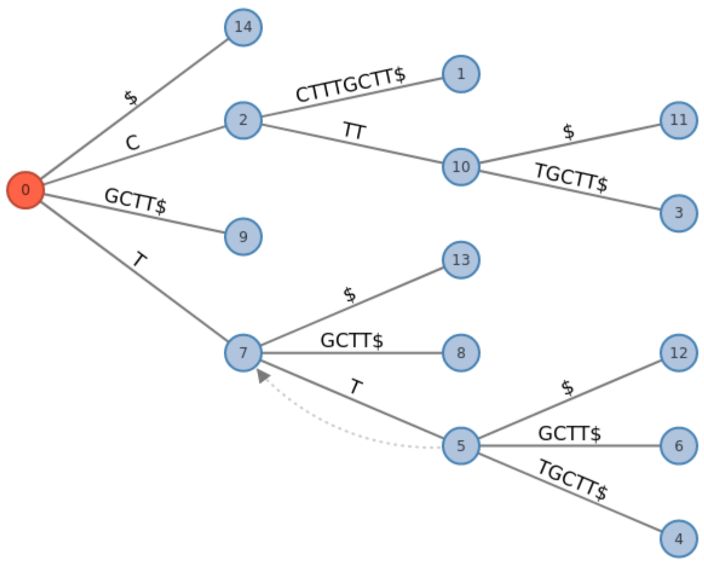

Figure 9: The suffix tree of $s_1\$$

[11]Respect with the one presented in [1].

[12]Which won't be done here

Consider as an example $s_1 = CCTTTGCTT$, in figure 9 its suffix tree is shown [13]. Suppose we want to derive the $3-complete$ and $3-faithful$ version of the singular variation graph. From the tree we see that the only labels describing a path (in the tree) longer or equal to 3 which also leads to more leaves are C - TT. Such labels correspond to three nodes (labelled C,T,T) repeated twice in the original variation graph (since the leaves are two). Having actually stored the indexes (depicting strings) and not the strings themselves on the edges on the tree[14] (the image depicts the strings for simplicity) we now know that we need to merge the two pair of nodes TT in position $[2,3]$ and $[7,8]$. These intervals, are obtained by traversing the tree backwards (starting from the leaves[15]) and propagating the children nodes starting indices. Namely consider again the leaves 11 and 3, their edges \$ and $TGCTT\$$ are labelled respectively $[9,9]$ and $[4,9]$. For $TGCTT\$$ the nodes T,T will be located at $[4-2=2, 4-1=3]$, the reason is that the depicted label TT is of length 2, so the starting position will be 2 indexes before the starting position of the leaf. The ending position is always 1 index before the beginning of the leaf. The same reasoning can be applied for \$, which will lead to mark the nodes T,T in position $[9-2=7, 9-1=8]$. So the interval of nodes $[2,3]$ and $[7,8]$ must be merged (note that obviously they depict singular nodes with the same labels).

The same reasoning **can't** done for the edge C. Indeed node 2 in the tree is not at a string depth greater or equal than k. This in turn means that its *direct* children don't share a k-mer. Though, their grandchildren do, for such reason in this situation we consider the two intervals coming from the TT edge, namely $[2,3]$ and $[7,8]$, and we expand them one more character to the left (since the edge c has only one character), to get $[1,1]$ $[6,6]$. These are the nodes that will be merged. The process then ends since we reached the root and there are not other paths with two or more leaves sharing an ancestor with a string depth greater or equal than k.

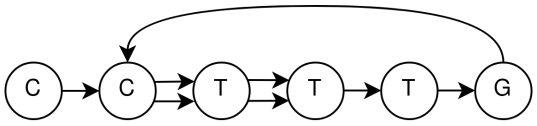

Figure 10: The resulting variation graph

It's immediate to see that the variation graph obtained by this procedure is $k-complete$: paths in the tree depicting multiple paths in the graph which in turn depict strings whose prefix longer or equal than k are merged together. It's also $k-faithful$ since no single nodes not sharing a $k-mer$ are merged. To this regard Figure 10 may not be very immediate, but consider the equivalent graph obtained by merging the three nodes CTT into a single non singular node, Figure 11, it's now immediate to see the $3-faithful$.

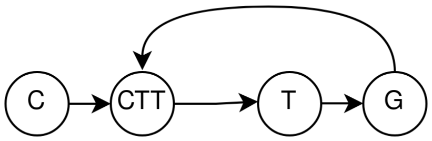

Figure 11: The resulting variation graph

Before formalising the algorithm we present one last example, consider $s_1 = AGGGGGT$. The suffix tree is shown in Figure 12.

[13]Tool used: `https://brenden.github.io/ukkonen-animation/`

[14]e.g. instead of the label $TGCTT\$$ we will have $[4,9]$,assuming the indexes start from 0.

[15]Only the leaves generated from the aforementioned internal nodes, not every leaf.

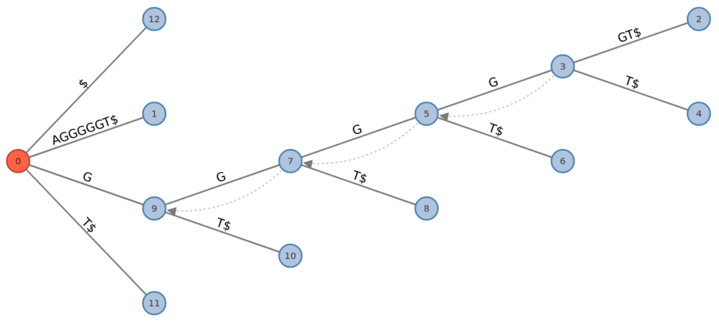

Figure 12: The suffix tree of $AGGGGGT\$$

Suppose again we want a $3 - complete$ and $3 - faithful$ representation. In this case the leaves to consider are the ones in Figure 12 numbered: $2, 4, 6$ since they present root-leaves paths whose labels (of the shared part) are longer than 3. The respective edges are labelled with the intervals: $[5, 7]$, $[6, 7]$,$[6, 7]$, we start the merging process from the deepest leaves, namely 2 and 4. The two intervals we obtain for the edge labelled G[16] leading to node 3 are: $[5 - 1 = 4, 5 - 1 = 4]$ and $[6 - 1 = 5, 6 - 1 = 5]$. This means nodes depicting in the singular variation graph the characters 4 and 5 will need to be merged. The result of this first (and the following) merging operations are shown in Figure 13.

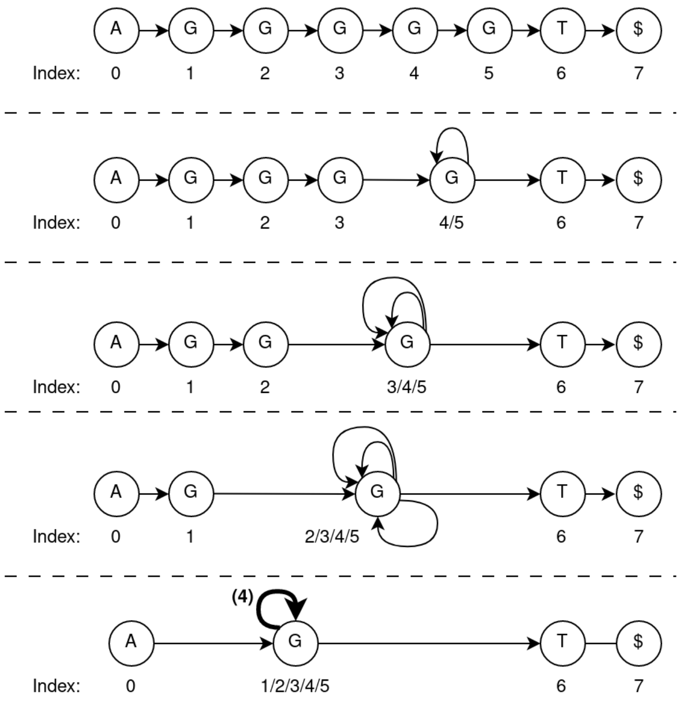

Figure 13: The merging process of $AGGGGGT\$$, in the last step the bold self loop arrow denotes that there are 4 self loops.

[16]The node $(5, 3)$ in the tree.

Now, carrying on, the edge preceding node 5 in the tree (labelled by G) will have associated the following intervals: $[5, 5]$ (coming from it's direct leaf, 6), $[5 - 1 = 4, 5 - 1 = 4]$ and $[3, 3]$, these last two coming from the other branch. Note that now that the nodes originally in the graph indexed 4 and 5 are merged, so, what does it mean to have an interval containing 4 or 5? It means considering the new $4/5$ node. So, we know node $4/5$ and 3 need to be merged. After the merging, as before, the same reasoning **can't** repeated for the remaining two G edges on the path, since once again the string depth of the node 7 in the tree is less than k. So given the two intervals coming from the edge G[17] $[3, 3]$ and $[4/5, 4/5]$ we consider the single character on the left, namely we merge the intervals $[2, 2]$ and $[3/4/5, 3/4/5]$. Nodes (in the graph) indexed 2 and $3/4/5$ are thus merged into node $2/3/4/5$. The last merging is analogous, and the result is the graph at the bottom in Figure 13.

The key point is to notice that the nodes to merge are *only* those around which a common string longer than k is shared, and the longer ($\geq k$) paths in the keyword trees depict exactly such graph nodes. In addition, when on such paths nodes at a lower ($\geq k$) string depth are reached, the merging process is just the extension of extending the merge already done to all k-extendable nodes.

B.2 The algorithm for a single string

Having understood the idea of the algorithm we now formalize the merge operation and the remaining procedures. In doing so we will introduce some modifications to the "classic" data structures in order to implement the algorithm in linear time.

B.2.1 The suffix tree

As seen in the examples above, once the tree is constructed we need to traverse some of its subtrees bottom up, starting from the leaves, thus we will do a DFS. Notice that no pair of indexes is needed for the internal edges, but we still need to know the length of the string depicted by all edges, since while traversing a subtree bottom up we need to extend to the left the intervals to continue the merging. To this regard, we could directly store during the creation of the suffix tree, not the pair of indexes for each internal edge, but just their difference, to save some space[18].

B.2.2 The Variation graph

While constructing the singular variation graph depicting $s := s_1\$$ we need to introduce (and construct) and additional array of pointers to the variation graph nodes. The size of the array will be the number of nodes in the graph (thus $|s_1\$|$) and at the beginning of the algorithm each cell i will reference the $i-th$ character of $s_1\$$, see Figure 15 (A). This linear (both in time and space) structure is necessary in order for the merge operation to be performed in linear time with respect to the size of intervals to merge.

B.2.3 The Merge operation

Consider to have fixed a variation graph (e.g. like in a global variable), the merge procedure takes as inputs two intervals (of nodes with the same labels), and returns an equivalent variation graph (in terms of string represented) in which each pair of respective[19] nodes in the interval is merged into a

[17]In terms of tree nodes is the edge $(7, 5)$.

[18]Note, the algorithm will still work also with the standard Ukkonnen's algorithm.

[19]I.e. the first node of the first interval is merged with the first node of the second interval and so on.

single node. This operation can be viewed as the repetition of a simple operation *join* performed on each respective pair of the intervals.

Algorithm 1 Join(i,j)

```
Result: Equivalent singular variation graph, in which nodes $i$,$j$ are merged (wlog $i < j$)
if A[i]==A[j] then
    return
end
Let O be the outgoing edges from j
Let I be the ingoing edges from i
for node in I do
    A[node].addOutgoingEdge(A[i])
end
for node in O do
    A[i].addOutgoingEdge(node)
end
A[j]=A[i]
```

Algorithm 2 Merge([i$_1$...i_n], [i'_1...i'_n])

```
Result: Equivalent singular variation graph, in which nodes of the two intervals are merged
for j in 1..n do
    merge(i$_j$, $i'_j$)
end
```

B.2.4 The algorithm

Recalling we what said about the structures:

- Suffix tree: For each edge we only need its string length.
- Suffix tree: The edges are not oriented (bi-directional).
- Variation graph: Needs an auxiliary array.

All the additional data on the suffix tree can be obtained slightly modifying Ukkonnen's algorithm (still making it work in linear time). The main procedure is the following[20]:

Algorithm 3 Pdor(k,s)

```
Result: Singular variation graph k-complete and k-faithful depicting s, we assume s to have already
        the termination character $.
suffTree = GenerateSuffixTree(s)                    ▷ generate suffix tree (Ukkonnen's alg)
varGraph,A=GenerateSingularVG(s)                    ▷ A is the auxiliary array
DFSandMerge(suffTree.root(),0,k,false)
return varGraph
```

In the $DFSandMerge$ procedure the $append$ function discards empty intervals, for example consider $list = [[1,2][2,3]]$, $list.append([])$ doesn't modify the object list. In addition the function $extendLeftSide$ applied on an interval, given an integer, returns the interval shifted by the value specified. As an example $[3,4].extendLeftSide(2)$ modifies the interval to $[1,2]$.

[20]Pdor stands for "Pangenome data operative representation"

Algorithm 4 DFSandMerge(node,stringDepth,doMerge)

```
if node.isLeaf() && stringDepth $\geq k$ then
    if doMerge==true then
        endIndex=STRLen-node.getEdgeLength()    ▷ STRLen is a global constant denoting $|s|$
        return [endIndex-node.getParent().getEdgeLength(),endIndex-1]
    else
        retrun []
    end
end
if node.isLeaf() then
    return []
end
if doMerge==false && stringDepth $\geq k$ then
    doMerge=true
end
listsToMerge=[]
for child in node.children do
    listsToMerge.append(DFSandMerge(child,stringDepth+node.getEdgeLength(child)),doMerge)
end
if node==root then
    return []    ▷ we don't want to merge negative intervals, so cut short
end
for i in 0..|listsToMerge|-2 do
    Merge(listsToMerge[i],listsToMerge[i+1])
    listsToMerge[i].extendLeftSide(node.node.getParent().getEdgeLength())
end
listsToMerge[|listsToMerge|-1].extendLeftSide(node.node.getParent().getEdgeLength())
return listsToMerge
```

B.2.5 A full example

Let's now see a full detailed more complex example:

Consider the string $s1 = AAACCAAAGAAACCAAAT$, once again we'd like to obtain the $3-complete$ and $3-faithful$ representation. Figure 15 depicts the algorithm described on $s1\$$.

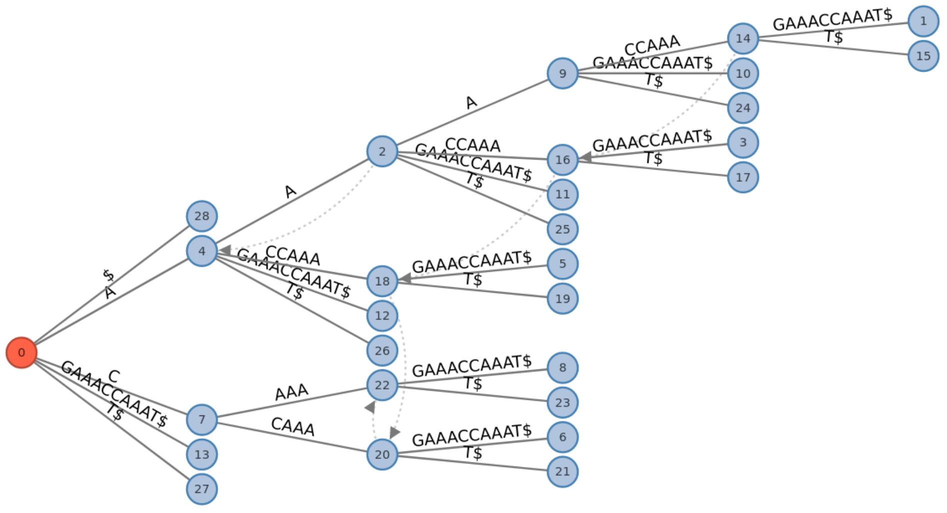

Figure 14: The suffix tree of $s_1\$$

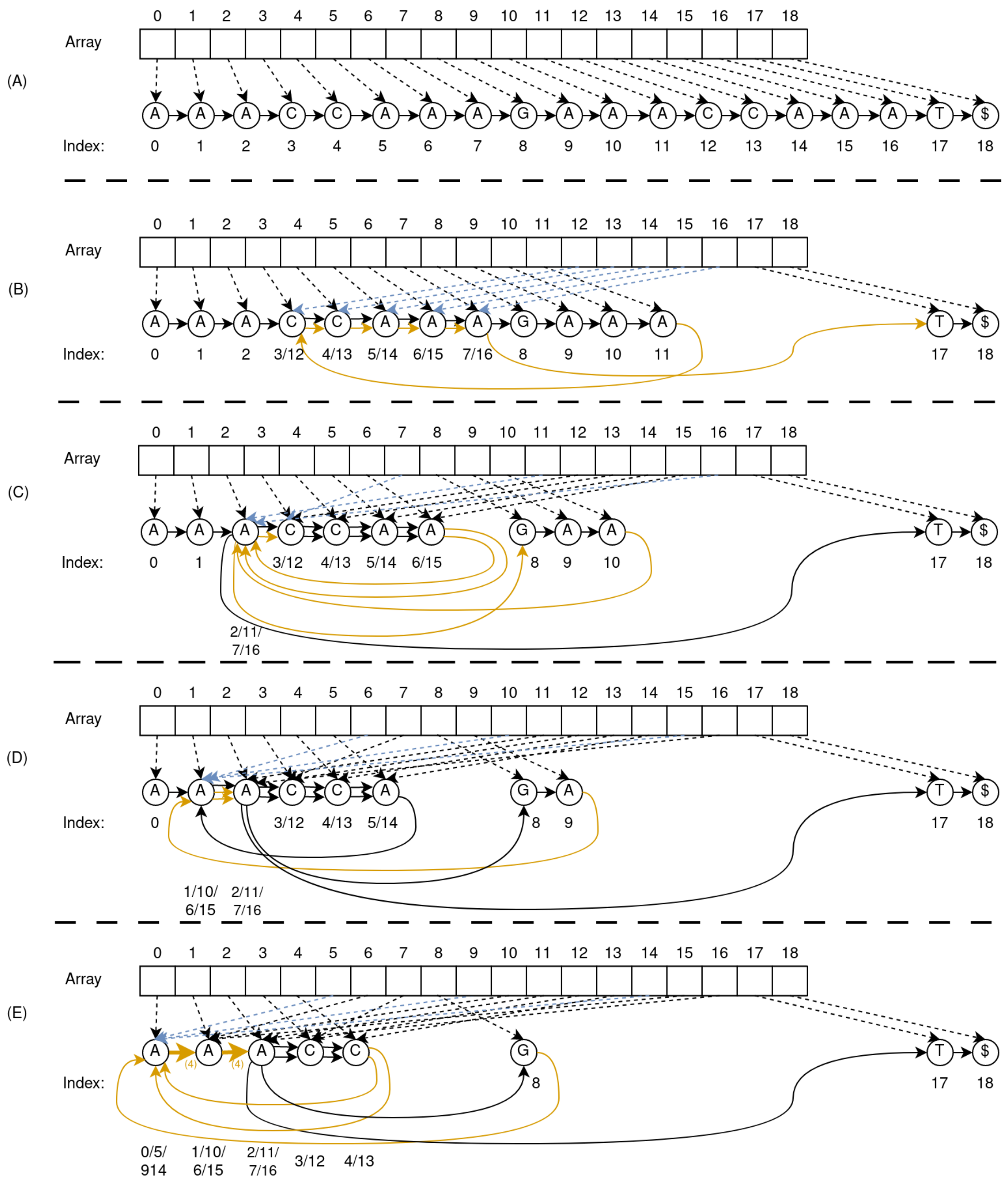

Figure 15: How the singular variation graph of $s_1\$$ evolves during the algorithm execution. Each step depicts the merge operations done at each level of the tree on the branch $0-4-2-9$. For example the first step, describes $merge([3,7],[12,16])$. The algorithm in this case after exploring such branch won't do any useful merging on the other sufficiently deep branches (namely $0-4-18$, $0-7-22$ and $0-7-20$). Yellow Lines represent the changes made at each level, while blue dotted lines show how the pointers in the array change.

B.3 To multiple strings and beyond

Let's now see the general case in which $S = \{s_1, .., s_n\}$. The first step is to construct the (non connected) singular linear variation graph encoding each string in S. We will thus have n non connected linear components, respectively of size $|s_1| + 1..|s_n| + 1$. The nodes in the non connected components are still enumerated with increasing numbers. Namely the $i - th$ node of the linear graph s_j will be enumerated:

$$(\sum_{l=1}^{j-1} |s_l| + 1) + i$$

We can then concatenate the strings $s_1..s_n$ separating them with different characters non present in any string and construct $s := s_1\$_1 s_2\$_2\$_3 \; .. \; \$_{n-1} S_n \$_n$.

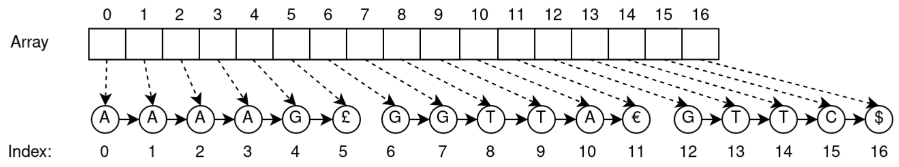

Figure 16: Enter Caption

Now we can construct via Ukkonnen's algorithm the suffix tree for s and invoke $Pdor(k, s)$. By doing so now, each and only path depicting a substring longer than k in one or more strings will be shared in the variation graph. The resulting singular variation graph will be k-complete and k-faithful.

We will now see a full example (executing the code). Consider the following set of strings $S = \{AAAAG, GGTTA, GTTC\}$, $s = AAAAG£GGTTA€GTTC\$$, the relative suffix tree is presented in Figure 17, while the starting singular graph can be viewed in Figure 16.

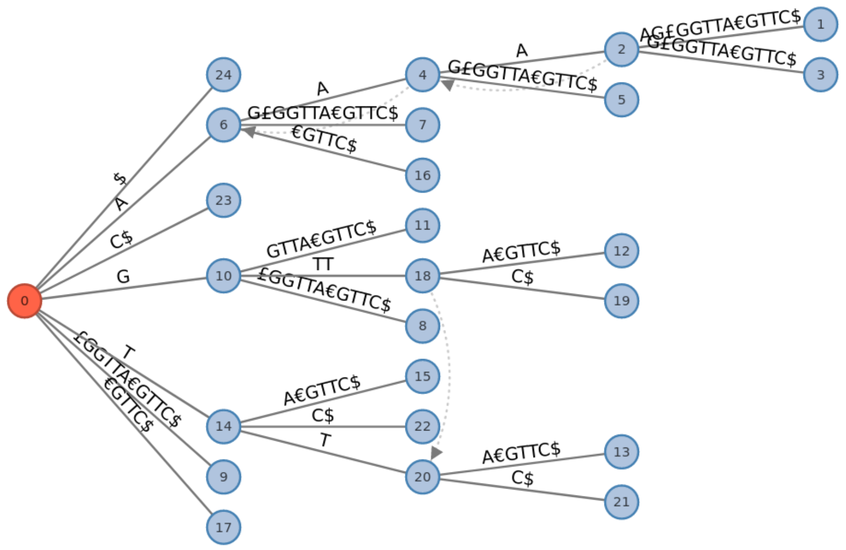

Figure 17: The suffix tree of s

Suppose we want to get its 3-complete and 3-faithful representation. As it can be seen in Figure 17 there are only two paths from the root containing a subtree with two or more leaves at a string depth greater or equal than k. These are labelled by $A - A - A$ and $G - TT$ and correspond to the paths depicted by nodes $0 - 6 - 4 - 2$ and $0 - 10 - 18$ in the tree. The code execution is the following:

- Once generated the graph and the suffix tree `DFSandMerge(root,0,3,false)` is called.
- The root is not a leaf, so the first two conditions are skipped over, so is the third since the string depth is 0. We then create a list with five possible lists and call the procedure recursively.
- WLOG, we assume to the ordering of the children of a node to be the one depicted in Figure 17. The first call, on node 24, returns [] since it's a leaf but the string depth is not sufficient.
- Then `DFSandMerge(6,1,3,false)` is called. Again the first three conditions fail, so a list of three possible lists is created.
- `DFSandMerge(4,2,3,false)` is called. The first three conditions fail, a list of two possible lists is created.
- `DFSandMerge(2,3,3,false)` is called. Now first two conditions fail, but the third is satisfied (meaning from this depth eventual merges can happen). A list of two possible lists is created.
- `DFSandMerge(1,17,3,true)`. Now, we are in a leaf, the first if conditions are satisfied as well as the nested if. So $endIndex = 17 - 14 = 3$, the parent (node 2) is connected to its parent (4) with an edge of length 1, so the pair $[3 - 1 = 2, 3 - 1 = 2]$ is returned.
- `DFSandMerge(3,16,3,true)`. Similar reasoning as above, $[4 - 1 = 3, 4 - 1 = 3]$ is returned.
- Now, back in node 2 $liststoMerge$ is $[[2, 2], [3, 3]]$. Merge is called one time and since the pointers to nodes (in the graph) 2 and 3 are different, the nodes are merged and the pointers now coincide. Lastly $liststoMerge[0]$ is modified in the for to $[1, 1]$ and $liststoMerge[1]$ is modified outside to $[2, 2]$ and returned.
- Now, back in node 4 we call `DFSandMerge(5,2+13=15,3,false)`. The first if condition is matched, though, the nested if fails (since the depth of the father wasn't greater or equal than 3), so [] is returned. Now, back in 4 again we have $liststoMerge = [[1, 1], [2, 2]]$. Merge is called one time and since the pointers to nodes (in the graph) 1 and 2 are different the nodes are merged and now the pointers coincide. Lastly $liststoMerge[0]$ modified in the for to $[0, 0]$ and $liststoMerge[1]$ is modified outside to $[1, 1]$ and returned.
- Back in node 6 we call `DFSandMerge(7,14,3,false)` and `DFSandMerge(16,7,3,false)`, which will both return []. So in 6 we have $liststoMerge = [[0, 0], [1, 1]]$. Merge is called one time and since the pointers to nodes (in the graph) 0 and 1 are different the nodes are merged and the pointers now coincide. Lastly $liststoMerge$ is modified but such modifications will be ignored since the parent of 6 is the root.
- We skip over the call to 23 and we proceed with `DFSandMerge(10,1,3,false)`. From this node, three calls are made, `DFSandMerge(11,11,3,false)` and `DFSandMerge(8,11,3,false)` which will both return [] and `DFSandMerge(18,3,3,false)` which we will now analyze.

- First, in node 18 $doMerge$ is set to true by the if statement, then `DFSandMerge(12,10,3,true)` and `DFSandMerge(19,5,3,true)` are called.
- Now in node 12 the condition of the first if is matched, as well as the nested one. For this reason endIndex is set to $16 - 6 = 10$ and $[10 - 2 = 8, 9]$ is returned.
- Similarly in node 19 $[13, 14]$ is returned.
- Back in node 18 we have $listToMerge = [[8, 9][13, 14]]$, so merge is called on such intervals, node 8 in the graph is merged with node 13 and 9 is merged with 14 (since the respective pointers were different). At the end the intervals in $listToMerge$ are modified to $[12, 13]$,$[7, 8]$ and they are returned.
- Back in node 10 $listToMerge$ is $[[12, 13], [7, 8]]$ since the two [] returned, are discarded in the append operation. Again merge is called once and the respective four nodes pointed by the array are merged into two.
- Back in the root, the remaining `DFSandMerge` calls don't arise any merge operation.

The final result is shown in Figure 18.

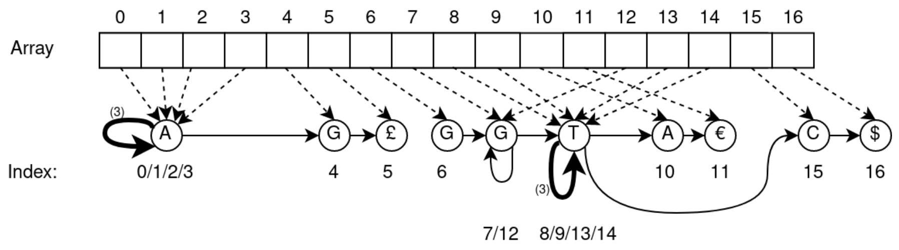

Figure 18: A 3-complete and faithful representation for the three strings

As a last node we recall that $Pdor$ returns only the variation graph and not the array, though, in order to represent the original set of strings the $\$_1..\$_n$ characters must be removed.

References

[1] A. Cicherski and N. Dojer, “From de bruijn graphs to variation graphs – relationships between pangenome models,” in *String Processing and Information Retrieval*, F. M. Nardini, N. Pisanti, and R. Venturini, Eds. Cham: Springer Nature Switzerland, 2023, pp. 114–128.

[2] E. Garrison, “Computational graph pangenomics: a tutorial on data structures and their applications,” 2022. [Online]. Available: https://link.springer.com/article/10.1007/s11047-022-09882-6

[3] M. Axenovich, “Lecture notes graph theory,” 2014.

[4] E. Garrison, “Graphical pangenomics,” 2019. [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/294516