Parameterized Algorithms for Function Merging and Branch Fusion

Informal problem statement In the FunctionMerging problem, we are given $k \geq 2$ functions, and our goal is to merge them into a single function of minimum size. An adjacent problem which we will not focus on for now is MergePartition where we want to partition the set of functions into parts s.t. the functions of each part are merged together (through FunctionMerging) and we want the total size to be minimized. Closely related to FunctionMerging is the problem BranchFusion, where we are given a branching statement with two (or more) branches, and we want to match these branches together to minimize the size of the resulting code.

Summary We propose a new and expressive formulation for FunctionMerging/BranchFusion as a problem that we call OptimalFlattening, which takes into account the program's CFG structure and the different ways this CFG can be flattened into linear code (previous approaches look at only one flattening). Suppose that we are given two functions (branches) of sizes n_1, n_2 that we want to merge (fuse), and that they have branching factor b_1, b_2* and nesting depth d_1, d_2 respectively.

- We give a simple dynamic programming algorithm that is FPT w.r.t. b_1, d_1, b_2, d_2.
- We give an FPT algorithm w.r.t. b_1, b_2, d_i, i.e., the problem remains FPT even when one of the functions/branches has unbounded depth. The algorithm relies on constructing a weighted pushdown system of FPT size and then solving, using [1], a reachability problem on it in time polynomial in the size of the system.
- We give two lower bounds showing that the problem is NP-hard when $d \leq 2$ for both functions/branches, or when b is bounded for one function/branch and d is bounded for the other function/branch.

1 Preliminaries and problem formulation

Representation of functions Each function/branch has two representations: a *flat* representation as a sequence of instructions (including jumps/ifs), and a (structured) *control-flow graph* representation $G = (V, E)$ where each node $u \in V$ represents a basic blocks containing straight-line code that doesn't affect control flow. Each CFG can be flattened by choosing, for every $u \in V$, an order over its children branches – which dictates which branch appears first– and then adding if/jump instructions accordingly. Each flattening is in one-to-one correspondence with such ordering, and it is clear that we can have an exponential number of flattenings. More formally, for a CFG $G = (V, E)$, denote the *nesting depth* of a node u by $nd(u)$, and its *children* by $ch(u)$. Note that for $v \in ch(u)$, we have $nd(v) \in \{nd(u) - 1, nd(u), nd(u) + 1\}$. We define the *branches* of u as $B(u) := \{v \in ch(u) \mid nd(v) = nd(u) + 1\}$, and the *branching factor* as $\max_{u \in V} |B(u)|$. A *branching order over* u as an ordering o_u over $B(u)$. Given a collection of orderings $\{o_u\}_{u \in V}$, they correspond to a flattening of the CFG G into a sequence of instructions in the obvious way, where for each $u \in V$, the code of its branches appears in the order given by o_u. Our ultimate goal is, given CFGs for multiple functions (branches), flatten each of them in a way that makes their merge (fusion) as small as possible.

Functions/branches as tree strings These definitions inspire the following representation of each function/branch as a *tree string*, which is a special type of string where we have characters that recursively branch into a finite number of tree strings. Formally, suppose that the code of our programs is over alphabet

*The branching factor is maximum number of branches, e.g., an if-elif-else gives a factor of 3

Σ. A tree string T is defined recursively over alphabet $\Sigma \cup \{\oplus, [,]\}$ as follows:

$$T \to S \mid S\ [T \oplus \cdots \oplus T]\ T$$
$$S \in \Sigma^+.$$

To transform a function/branch into a tree string, suppose its code is of the form

$$P := P_0; \langle \text{if/while} \rangle \{P_1; \ldots; P_k\}; P_{k+1},$$

where P_0 is a straight-line program whose code, as a string, is c_0. Then, its tree string will be:

$$T(P) := c_0\ [T(P_1) \oplus \cdots \oplus T(P_k)]\ T(P_{k+1}).$$

If P has no branching at all, then it simply maps to its straight-line code, which corresponds to the rule $T \to S$.

The nesting depth of a tree string is the maximum number of nested []'s and its branching factor is the maximum number of T's appearing in a $[T \oplus \cdots \oplus T]$ sub-expression. Clearly, the nesting depth and branching factor of a tree string coincide with those of the CFG it was derived from. The size of a tree string is the number of characters in it (including $\oplus$/[]'s). Note how $T(P)$ has size $O(|P|)$. A *flattening* of a tree string is a string over Σ^+ that is obtained by arbitrarily reordering the T's in the $[T \oplus \cdots \oplus T]$ sub-expressions and then removing the $\oplus$/[]'s. Note how different flattenings yield strings of the same size.

Formal problem statement We now define the problem OptimalFlattening: Given two functions/branches represented as tree strings T_1 and T_2 of sizes n_1, n_2, nesting depths d_1, d_2, and branching factors b_1, b_2, find two flattenings F_1 and F_2 of T_1 and T_2 respectively, such that $\text{LCS}(F_1, F_2)$ is maximized. (All of this is easily generalized to ≥ 3 functions and with different profit/penalty for different matches/mismatches/insertions/deletions/substitutions over Σ). We shall take this problem as our model for FunctionMerging/BranchFusion by first transforming the input functions/branches into tree strings, solving OptimalFlattening, then converting back the optimal flattenings into code, and finally performing the merging/fusion according to the LCS over the flattenings. The differences between the original program, flattenings of its tree string, and the reordered programs derived from those flattenings, is reflected in the labels and jump instructions (which can differ between different orderings of the program, and are not present in flattenings). Those differences are taken to be negligible, and the way to resolve them is left as an implementation detail.

2 Related works

Function merging Previous works look at the more general MergePartition and not only FunctionMerging, and almost all of them consider merging only a pair of functions (i.e., $k = 2$). Existing compilers were limited to only merging identical functions, and the first work to go beyond that is [2] which merges two functions if their CFGs satisfy a strong notion of isomorphism[†] and the total difference of basic-block code in corresponding nodes stays within a certain threshold. Another approach that showed up later is [3] which looks at (only one) flat representation of the two functions and computes a sequence alignment between them to do the merging. They use a simple scoring scheme that makes the problem boil down to edit distance, which itself is essentially LCS with weights. An improvement over this using the same idea, but tackling SSA-related problems, is in [4]. A more practical version of it that runs faster while giving only slightly worse reductions is [5]. [6] proposes a better method to do find a good partitioning of the functions before merging them. [7] extend [6] to merge more than 2 functions at a time. From a view that cares more about size reduction than the algorithm being fast, the state-of-the-art can be taken to be [4]'s approach to FunctionMerging (which is sequence alignment + SSA tricks), combined with [6]'s partitioning algorithm. [8] is another paper that used sequence alignment/LCS in the context of compiler optimization to merge redundant code.

[†]Namely, their algorithm assumes that for every node, it has a pre-determined order over its children. Then, the algorithm tries to find an isomorphism that also perserves this ordering, i.e., if $u \in F_1$ is mapped to $v \in F_2$, then the i'th child of u must be mapped to the i'th child of v. This is a much simpler problem than the general graph isomorphism, and can be solved in polynomial time.

Branch fusion Branch fusion is first introduced in [8] to reduce thread divergence in GPUs, and not for the aim of reducing code size. [9] then adapted it for size reduction, and is the SOTA paper for branch fusion.

Relation to edit distance/similarity in graphs From a more abstract perspective, OptimalFlattening can be seen as determining similarilty between two trees using a notion of *edit distance* between (labeled) trees [10]. Those trees are simply the parse trees of tree strings, where the only relevant labels are those holding Σ-strings. The hardness of such task depends on two features: whether the trees are *ordered* or *unordered* (or something in the middle), and the precise definition of edit distance. For unordered unlabeled trees where our goal is to make the trees isomorphic through edge insertion/removal, the problem is NP-hard [11]. NP-hardness also holds for unordered trees where our goal is label-respecting isomorphism through node insertion/removal, or changeing label of nodes. This result holds even when the trees are binary and labels have alphabet of size two [12]. On the other hand, this problem becomes tractable on *ordered* trees [13]. Our problem differs from all of these: it can be seen as edit distance on labeled trees that is a hybrid between being ordered and unordered. Namely, consider the parse tree of the tree string $T_0 := S[T_1 \oplus \cdots \oplus T_n] \dots$. The children corresponding to T_i's should be unordered amongst themselves, but are all ordered before the child corresponding to S. Moreover, the edit distance operation are not defined on the trees' structure directly, but on a string of labels that is spelt out by a flattening of them.

3 Algorithms

We first show a dynamic programming algorithm to solve the problem in FPT time w.r.t. b_1, d_1, b_2, d_2. Then, we show an FPT algorithm when one of the tree strings, say T_1, has unbounded depth, i.e., an FPT algorithm w.r.t. b_1, b_2, d_2.

Dynamic programming We solve the problem by extending the dynamic programming solution for LCS to tree strings. The character at x'th position of a tree string T is denoted by $T(x)$. For a position x containing a Σ-character, suppose that it is enclosed within l []'s:

$$\dots[_1 \dots [_2 \dots [_l \dots T(x) \dots]_l \dots]_2 \dots]_1 \dots \tag{1}$$

where each parenthesis is of the form $[_i T_{i,1} \oplus \cdots \oplus T_{i,m_i}]_i$. Further, suppose that $T(x)$ lies within the subexpression $T_{i,par_i(x)}$, for all i. We will associate with x a collection of l subsets $\overline{S} = (S_1, \dots, S_l)$ where S_i satisfies

$$\{par_i(x)\} \subseteq S_i \subseteq \{1, \dots, m_i\}.$$

$T_{i,j}, j \in S_i \backslash \{par_i(x)\}$ indicates that $T_{i,j}$ have been already fully scanned. Scanning is done from right to left. At position x, we have partially scanned $T_{i,par_i(x)}$, which we always remember in S_i. $\overline{S}$ is called a *signature* for x.

Now suppose we want to move from x to its predecessor (in a flattened string). The signature $(S_1, \dots, S_l)$ gives us complete information of where such predecessor can be. By exploring all of those predecessors, we properly explore all possible flattenings.

Let us parameterize all the notation in the previous paragraphs with $I \in \{1, 2\}$ which refers to the input tree strings T_1 and T_2. Our state will be of the form:

$$dp[x_1, S_{1,1}, \dots, S_{1,l_1}, x_2, S_{2,1}, \dots, S_{2,l_2}],$$
$$\text{where } \forall I \in \{1,2\}.\ (x_I \in \{1, \dots, n_I\} \wedge T_I(x_I) \in \Sigma$$
$$\wedge \forall i \in \{1, \dots, l_I\}.\ \{T_{I,i,par_i(x)}\} \subseteq S_{I,i} \subseteq \{1, \dots, m_{I,i}\}).$$

Intuitively, this state corresponds to solving the problem on a partially processed T_1, T_2 after we made two types of choices: move choices over Σ-characters to explore different values for the LCS, and choices over the ordering of some of the branches. The positions x_1/x_2 record where in the tree string we are, and the signatures $\overline{S_1}/\overline{S_2}$ record the relevant choices w.r.t. ordering branches. The value of the dp state is the maximum LCS we can achieve over all flattenings of the tree strings from that situation.

More formally, the value of that state is the maximum LCS over all flattenings of tree strings T_1' and T_2'. T_I' is obtained from T_I as follows:

- Remove the prefix from the right of $T(x_I)$ (excluding $T(x_I)$) until we either reach $]_{l_I}$ or an $\oplus$ symbol that is enclosed by l_I parenthesis.[‡]
- For $i \in \{2, \ldots, l_I\}$, remove the prefix from the right of $]_{I,i}$ (excluding $]_{I,i}$) until we either reach $]_{I,i-1}$ or an $\oplus$ symbol that is enclosed by $i-1$ parenthesis.
- Remove the prefix from the right of $]_{I,1}$ (excluding $]_{I,1}$) until the end of T_I.
- For $i \in \{1, \ldots, l_I\}$, remove all $T_{I,i,j}$'s from the expression $[_{I,i} T_{I,i,1} \oplus \cdots \oplus T_{I,i,m_{I,i}}]_{I,i}$ for $j \in S_{I,i} \setminus \{par_i(x)\}$.
- For $i \in \{1, \ldots, l_I\}$ in decreasing order *move* $T_{I,i,par_i(x_I)}$ to the right of $]_{I,i}$, i.e., apply the following transformation:

$$\ldots [_{I,i} \cdots \oplus T_{I,i,par_i(x_I)} \oplus \ldots]_{I,i} \ldots$$
$$\downarrow$$
$$\ldots [_{I,i} \ldots]_{I,i} T_{I,i,par_i(x_I)} \ldots$$

  Note that since we go in decreasing order, $par_i(x_I)$ remains the same for the i under consideration. This transformation corresponds to the fact that since $T_{I,i,par_i(x_I)}$ are already partially scanned, we should continue our (right-to-left) scan from there before moving to other branches.

We define the transitions as follows:

$$dp[x_1, \overline{S_1}, x_2, \overline{S_2}] = \max \begin{cases} [T_1(x_1) = T_2(x_2)] + \max_{\substack{(x_1', \overline{S_1'}) \in pred(x_1, \overline{S_1}) \\ (x_2', \overline{S_2'}) \in pred(x_2, \overline{S_2})}} dp[x_1', \overline{S_1'}, x_2', \overline{S_2'}], \\ \max_{(x_2', \overline{S_2'}) \in pred(x_2, \overline{S_2})} dp[x_1, \overline{S_1}, x_2', \overline{S_2'}], \\ \max_{(x_1', \overline{S_1'}) \in pred(x_1, \overline{S_1})} dp[x_1', \overline{S_1'}, x_2, \overline{S_2}]. \end{cases}$$

$pred(x_I, \overline{S_I})$ simply returns the set of pairs of positions and signatures that can precede x_I in a flattening consistent with S_I. Unlike the LCS standard dp, where the predecessor is always the previous position, here we can have multiple ones due to different choices of ordering branches. We now define $pred(x, \overline{S} = S_1, \ldots, S_l)$ for a position x in T, using the same notation as above.

- If the letter to the left of x is a letter from Σ, then we have one predecessor $(x-1, \overline{S})$.
- If the letter to the left of x is a closing bracket, i.e., we have the situation

$$\ldots [_{l+1} T_{l+1,1} \oplus \cdots \oplus T_{l+1,m_{l+1}}]_{l+1} T(x) \ldots$$

  then we go through $j \in \{1, \ldots, m_{l+1}\}$, and for each such j, we add to $pred(x, \overline{S})$ the pair

$$(x', \overline{S}[S_{l+1} \leftarrow \{j\}])$$

  where x' is the last position in $T_{l+1,j}$ (which is always well-defined and contains a Σ-character, by definition of our grammar).
- If the letter to the left of x is an opening bracket $[_l$ or a $\oplus$ symbol, then we go through $j \in \{1, \ldots, m_l\} \setminus S_l$, and for each j, we add to *pred* the pair

$$(x', \overline{S}[S_l \leftarrow S_l \cup \{j\}])$$

  where x' is the last position in $T_{l,j}$. If no such j exist, then we have a single predecessor where we remove S_l from the signature, and go to position immediately to the left of $[_l$.

[‡]By definition, those would be the same l_I parenthesis enclosing $T(x_I)$.

Running time For each of T_i, we remember a position in $[1, n_i]$, as well as up to d_i masks over sets of size b_i, which gives $2^{b_i \cdot d_i} \cdot d_i \cdot n_i$. Each signature takes $O(b_i d_i)$ time/space to create, and $|pred| \leq b_i$, meaning that the computation of *pred* can be done in $O(b_i^2 d_i)$. Going through all predecessor pairs for the two tree strings and looking up the dp table can be done in $O(b_1 b_2 (b_1 d_1 + b_2 d_2))$. This gives a running time of $O\left(2^{b_1 \cdot d_1 + b_2 \cdot d_2} d_1 d_2 b_1 b_2 (b_1 d_1 + b_2 d_2) \cdot n_1 \cdot n_2\right)$. If we idetify parameters for both T_i's, we would get a running time of $O(4^{bd} \cdot (bd)^3 \cdot n^2)$ Memory usage can be optimized and running time can be improved, at least in practice, using bit tricks. Note that the running time is independent of the alphabet size.

Automata-theoretic view of the dp solution The rest of this section assumes that the maximal LCS/edit distance values is bounded by a value v^* that is polynomial in n_1, n_2. This holds immediately for LCS, and it is reasonable to assume that edit distance weights are small integers, which subsequently gives a polynomial v^*. The previous dp algorithm can be viewed as a weighted finite automata over a *finite* max-plus tropical semiring

$$(\{0, \ldots, 2v^*\} \cup \{-\infty\}, \oplus := \max, \otimes := +, -\infty, 0).$$

The states of the automata are the states of the dp, i.e., pairs $\langle x_1, \overline{S_1}, x_2, \overline{S_2} \rangle$, as well as a special state q^*. The transitions outgoing from $\langle x_1, \overline{S_1}, x_2, \overline{S_2} \rangle$ are defined through the following formula, where the weight of a transition is written above the arrow:

$$\begin{aligned}
\forall (x_1', \overline{S_1'}) \in pred(x_1, \overline{S_1}), (x_2', \overline{S_2'}) \in pred(x_2, \overline{S_2}).\ \langle x_1, \overline{S_1}, x_2, \overline{S_2} \rangle &\xrightarrow{[T_1(x_1) = T_2(x_2)]} \langle x_1', \overline{S_1'}, x_2', \overline{S_2'} \rangle \\
\forall (x_2', \overline{S_2'}) \in pred(x_2, \overline{S_2}).\ \langle x_1, \overline{S_1}, x_2, \overline{S_2} \rangle &\xrightarrow{0} \langle x_1, \overline{S_1}, x_2', \overline{S_2'} \rangle \\
\forall (x_1', \overline{S_1'}) \in pred(x_1, \overline{S_1}).\ \langle x_1, \overline{S_1}, x_2, \overline{S_2} \rangle &\xrightarrow{0} \langle x_1', \overline{S_1'}, x_2, \overline{S_2} \rangle.
\end{aligned}$$

Moreover, if $x_1 = x_2 = 0$, we add a transition $\langle x_1, \overline{S_1}, x_2, \overline{S_2} \rangle \xrightarrow{v^*} q^*$. The weights can also be adjusted to the case where we have different weights for different edit distance operations. The initial state is $\langle n_1, \emptyset, n_2, \emptyset \rangle$. What we want to compute is the maximum weight of a maximal path from the initial state, which is equivalent to solving the the *generalized pushdown predecessor (GPP) problem* on the automata, which itself can be solved in time polynomial in the size of the automata as well as v^* [1]. This gives an FPT algorithm w.r.t. b_1, b_2, d_1, d_2. Note that the transitions to q^* ensures that the GPP solution only considers maximal paths that fully scans both tree strings. Further, by definition of v^*, the GPP solution will be at most $2v^*$, which fits in our finite semiring.

Algorithm for unbounded d_1 Now suppose that T_1's depth d_1 unbounded. We will show an FPT algorithm w.r.t. b_1, b_2, d_2. The idea is to turn the above weighted finite automata into a pushdown system (PDS) of FPT size, and then solve the GPP problem on it. A *pushdown system* is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is the state-set, Γ is the stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of transitions that may push/pop elements to the stack. A weighted PDS is a PDS that is augmented with a semiring and a function assigning weights from the semiring to its transitions Δ. We now define a weighted PDS that captures our problem. It differs from the above construction is that we do not remember signatures for x_1, and instead encode its content on the stack.

We will stick to the same semiring as above. Define

$$P := \{\langle x_1, x_2, \overline{S_2} \rangle \mid \overline{S_2} \text{ is a signature for } x_2\} \cup \{q^*\}.$$

Γ is defined as $2^{\{1, \ldots, b_1\}} \cup \{\$\}$, i.e., the set of subsets of $\{1, \ldots, b_1\}$, as well as a special stack symbol \$. The initial configuration is $(\langle n_1, n_2, \emptyset \rangle, \$)$. For transitions, define for every $(x_1, \gamma) \in \{1, \ldots, n_1\} \times \Gamma$ the function $\overline{pred}(x_1, \gamma) \subseteq \{1, \ldots, n_1\} \times \Gamma^*$, which will tell us how to move from x_1 by specifying a new position as well as an appropriate stack operation. $\overline{pred}(x_1, \gamma)$ is defined as follows, where we again use the same notation as in Equation 1 (the definition is very similar to *pred*).

- If it is not the case that $par_l(x_1) \subseteq \gamma \subseteq \{1, \ldots, m_l\}$, then $\overline{pred}(x_1, \gamma) = \emptyset$. The following cases assume that this condition holds.
- If the letter to the left of x is a letter from Σ, then we have one predecessor $(x - 1, \gamma)$.

- If the letter to the left of x is a closing bracket, i.e., we have the situation

$$\dots[_{l+1}T_{l+1,1} \oplus \dots \oplus T_{l+1,m_{l+1}}]_{l+1}T(x)\dots$$

then we go through $j \in \{1, \dots, m_{l+1}\}$, and for each such j, we add to $\overline{pred}(x, \gamma)$ the pair

$$(x', \gamma\{j\})^{\S}$$

where x' is the last position in $T_{l+1,j}$.

- If the letter to the left of x is an opening bracket $[_l$ or a $\oplus$ symbol, then we go through $j \in \{1, \dots, m_l\} \setminus \gamma$, and for each j, we add to $\overline{pred}$ the pair

$$(x', (\gamma \cup \{j\}))$$

where x' is the last position in $T_{l,j}$. If no such j exist, then we have a single predecessor (x', ϵ) where x' is the position immediately to the left of $[_l$.

Now the transitions of the PDS Δ are defined as follows for every $\langle x_1, x_2, \overline{S_2}\rangle, \gamma \in \Gamma$:

$$\begin{aligned}
\forall (x_1', \gamma') \in \overline{pred}(x_1, \gamma), (x_2', \overline{S_2'}) \in pred(x_2, \overline{S_2}). \quad & \left(\langle x_1, x_2, \overline{S_2}\rangle, \gamma\right) \xrightarrow{[T_1(x_1)=T_2(x_2)]} \left(\langle x_1', x_2', \overline{S_2'}\rangle, \gamma'\right) \\
\forall (x_2', \overline{S_2'}) \in pred(x_2, \overline{S_2}). \quad & \left(\langle x_1, x_2, \overline{S_2}\rangle, \gamma\right) \xrightarrow{0} \left(\langle x_1, x_2', \overline{S_2'}\rangle, \gamma\right) \\
\forall (x_1', \gamma') \in \overline{pred}(x_1, \gamma). \quad & \left(\langle x_1, x_2, \overline{S_2}\rangle, \gamma\right) \xrightarrow{0} \left(\langle x_1', x_2, \overline{S_2}\rangle, \gamma'\right),
\end{aligned}$$

and finally, we add the following transition for $x_1 = x_2 = 0$:

$$\left(\langle x_1, x_2, \overline{S_2}\rangle, \$\right) \xrightarrow{v^*} (q^*, \$)\,.$$

4 Complexity

Hardness of unbounded branching and bounded depth We show that the problem becomes NP-hard when the branching factor is unbounded but the nesting depth is restricted to 2. This proof goes by a reduction from Hamiltonian path, where we are given a graph $G = (V, E)$ of n nodes and we wish to find a path that visits all nodes exactly once. Suppose that $V = \{1, \dots, n\}$. Without loss of generality, we will consider a variant of the problem where the path we want to find must start from node 1. The general problem reduces to this special case by adding a new node connected to all other nodes, and requiring the path to start from this new node.

We will construct two tree strings T_1 and T_2 on an alphabet $\Sigma := \{1, \dots, n, X, \$, \#, *\}$ of $n + 4$ symbols. Let $a_{i,j} \in \Sigma$ be defined as j if $(i, j) \in E$ and be $*$ otherwise. T_1 and T_2 are defined as follows:

$$\begin{aligned}
T_1 &:= 1[U_2 \oplus \dots \oplus U_n] \underbrace{X \dots X}_{n \text{ times}} \\
U_i &:= i \underbrace{X \dots X}_{n \text{ times}} i \\
T_2 &:= V_1[V_2 \oplus \dots \oplus V_n]\# \\
V_i &:= i[a_{i,1}X \oplus a_{i,2}X \oplus \dots \oplus a_{i,n}X]\#
\end{aligned}$$

Note how both T_1 and T_2 have size polynomial in n. Correctness is established through the following claim.

Claim 1. *T_1 and T_2 have flattenings that achieve an LCS of size $1 + (n-1) \cdot (n+2) + n$ if and only if G has a Hamiltonian path starting from node 1.*

[§]The top of the stack appears on the right, so this just means to push $\{j\}$.

Proof. First observe that the symbols # exists only in T_2, and thus they will never be matched, so we ignore them. They are only added to make our construction fits the grammar of tree strings.

Suppose G has a hamiltonian path $P := p_1 \dots, p_n$ where $p_1 = 1$ and $\{p_2, \dots p_n\} = \{2, \dots, n\}$. We will flatten T_1 as $1U_{p_2} \dots U_{p_n} X^n$, and T_2 as $V_1 V_{p_2} \dots V_{p_n}$, where within each $V_{p_i}, i \in [1, n-1]$, we flatten it in any way that matches the form

$$p_i \; a_{p_i,p_{i+1}} X \underbrace{a_{p_i,q_1} X \dots a_{p_i,q_{n-1}} X}_{\substack{q_1 \dots q_{n-1} \text{ is a perm. of} \\ \{1 \dots n\} \setminus \{p_{i+1}\}}}.$$

Note that since $(p_i, p_{i+1}) \in E$, we have $a_{p_i,p_{i+1}} = p_{i+1} \neq *$. We flatten V_{p_n} arbitrarily. By expanding those flattenings, we get the following two strings:

$$F_1 := 1 \overbrace{p_2 \quad X^n \quad p_2}^{U_{p_2}} \overbrace{p_3 \quad X^n \quad p_3}^{U_{p_3}} \dots \overbrace{\dots p_{n-1}}^{U_{p_{n-1}}} \overbrace{p_n \quad X^n \quad p_n}^{U_{p_n}} \quad X^n$$

$$F_2 := \underbrace{1 \; (p_2 \; X(a_{1,\_}X)^{n-1})}_{V_{p_1}=V_1} \underbrace{p_2 \; (p_3 \; X(a_{p_2,\_}X)^{n-1})}_{V_{p_2}} \underbrace{p_3 \; (p_4 \; X(a_{p_3,\_}X)^{n-1})}_{V_{p_3}} \dots \underbrace{p_{n-1}(p_n \; X(a_{p_{n-1},\_}X)^{n-1})}_{V_{p_{n-1}}} \underbrace{p_n(a_{p_n,\_}X)^n}_{V_{p_n}}$$

Note that we can match *all* $1 + (n-1) \cdot (n+2) + n$ *characters* in F_1 : we match the first 1's in F_1 and F_2, then the first p_2 in U_{p_2} with p_2 in V_{p_1} and then match X^n with the n (separated) X's in V_{p_1}. Then we match the second p_2 in U_{p_2} to the first p_2 in V_{p_2}. This continues until we match all the X's in U_{p_n} to the X's in $V_{p_{n-1}}$, after which we match the last p_n in U_{p_n} to p_n V_{p_n}. Finally, we match the X^n suffix in F_1 to the Xs in V_{p_n}.

For the other direction, suppose that G does not have a Hamiltonian path starting from 1, and that for the sake of a contradiction, there are flattenings F_1 and F_2 and achieving an LCS of size $1+(n-1)\cdot(n+2)+n$, i.e., all of F_1 is matched with F_2. Suppose that T_1 is flattened as $1U_{q_2}, \dots, U_{q_n} X^n$, and T_2 as $V_{r_1} V_{r_2} \dots V_{r_n}$ where $r_1 = 1$, and $q_2 \dots q_n$ and $r_2 \dots r_n$ are permutations of $\{2, \dots, n\}$. Then we can write F_1 and F_2 as follows:

$$F_1 := 1 \overbrace{q_2 \quad X^n \quad q_2}^{U_{q_2}} \overbrace{q_3 \quad X^n \quad q_3}^{U_{q_3}} \dots \overbrace{\dots q_{n-1}}^{U_{q_{n-1}}} \overbrace{q_n \quad X^n \quad q_n}^{U_{q_n}} \quad X^n$$

$$F_2 := \underbrace{1 \; (f_2 \; X(a_{1,\_}X)^{n-1})}_{V_{r_1}=V_1} \underbrace{r_2 \; (f_3 \; X(a_{r_2,\_}X)^{n-1})}_{V_{r_2}} \underbrace{r_3 \; (f_4 \; X(a_{r_3,\_}X)^{n-1})}_{V_{r_3}} \dots \underbrace{r_{n-1}(f_n \; X(a_{r_{n-1},\_}X)^{n-1})}_{V_{r_{n-1}}} \underbrace{r_n(a_{r_n,\_}X)^n}_{V_{r_n}}$$

where for $i \in \{2, \dots, n\}$, $f_i \in \{a_{r_{i-1},1}, \dots a_{r_{i-1},n}\}$.

The crucial observation is that the number of Xs in both F_1 and F_2 is the same (n^2), and since all characters of F_1 (and thus all of its Xs) are matched, the matching between the Xs is determined. This implies that the first (last) X in U_{q_i}, for $i \in \{2, \dots n\}$ is matched to the first (last) X in $V_{r_{i-1}}$, and that he X_n suffix in F_1 is mapped to the Xs in V_{r_n}. This entails the following:

$$\text{for } i \in \{2, \dots, n-1\}, \; q_i = f_i = r_i \neq * \; \wedge \; q_n = f_n \neq *. \tag{2}$$

By the fact that $f_i \neq *$, we have:

$$(1, f_2) \in E \wedge (r_2, f_3) = (f_2, f_3) \in E \wedge \dots \wedge (f_{n-1}, f_n) \in E,$$

which implies that G has a Hamiltonian path starting from 1, a contraditcion. □

Hardness of parameterization depth of a tree string and branching of the other We can modify the above reduction so that T_2 has constant branching factor (of 2) and logarithmic depth, and T_1 maintains its constant depth (of 1) and unbounded branching factor. First observe that unbounded branching is *inherently more powerful than unbounded depth* in the sense that it allows us to obtain more flattenings than depth does. In particular, for any constant c, we can construct two tree strings of size $O(N)$, the first has constant branching factor c and unbounded depth, whereas the other has constant depth and unbounded

branching. In the first case, we can obtain $O(c^N)$ flattenings, whereas in the second we can obtain $\Omega(N!)$ flattenings using a single expressions with N branches. Since we have $c^N \ll N!$ for sufficiently large N, branching gives us more flattenings. This entails that we cannot mimic a tree string $[T_1 \oplus \dots T_N]$ using large depth and constant branching. However, we can exploit this to *select* one of the N branches, while ignoring the remaining $N-1$ symbols. This is the main observation of the reduction.

We extend the alphabet with an extra symbol Y, and re-define the tree strings as follows:

$$
\begin{aligned}
T_1 &:= 1[U_2 \oplus \dots \oplus U_n] X^n Y^{n-1} \\
U_i &:= i X^n Y^{n-1} i \\
T_2 &:= V_1 Y^{n-1} \underbrace{G_{[2,n]} \dots G_{[2,n]}}_{n-1 \text{ times}} \# \\
V_i &:= i W_{i,[1,n]} \# \\
\forall l, r. \quad G_{[l,r]} &:= \begin{cases} V_l Y & \text{if } l = r \\ \# \left[G_{[l,\lfloor (l+r)/2 \rfloor]} \oplus G_{[\lfloor (l+r)/2 \rfloor + 1, r]} \right] \# & \text{otherwise} \end{cases} \\
\forall l, r. \quad W_{i,[l,r]} &:= \begin{cases} a_{i,l} X & \text{if } l = r \\ \# \left[W_{i,[l,\lfloor (l+r)/2 \rfloor]} \oplus W_{i,[\lfloor (l+r)/2 \rfloor + 1, r]} \right] \# & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that T_2 indeed has branching factor of 2 and $O(\log n)$ depth. The reduction can clearly be applied in polynomial time. We now prove Claim 1 for this new construction, changing the LCS value in the claim to $1 + (n-1) \cdot (n+2) + n + n \cdot (n-1)$, which is the total number of symbols in T_1.

Proof. As before, we ignore # symbols since they only occur in T_2. For the tree string $G_{[l,r]}$, note that it has $r - l + 1$ leaves and is always flattened into a string of the form $V_{q_1} Y \dots V_{q_{r-l+1}} Y$ where q_j's is a permutation of $[l, r]$. Moreover, for every $k \in [l, r]$, there is a possible flattening that makes $V_k Y$ appears as a prefix. Similarly for $W_{i,[l,r]}$, it gets flattened into $a_{i,q_1} X \dots a_{i,q_{r-l+1}} X$, and for every $k \in [l, r]$, there is a flattening making $a_{i,k} X$ appear first. Applying this to the $W_{i,[1,n]}$ in V_i, we get that V_i can only be flattened into $i(a_{i,q_1} X \dots a_{i,q_n} X)$ and for every $k \in [1, n]$, there is at least one such flattening with $q_1 = k$.

Suppose we have a hamiltonian path $P := p_1, \dots, p_n \wedge p_1 = 1$. For T_1, we use the same flattening as in the previous proof. For T_2, we flatten the k'th $G_{[2,n]}$ $(1 \le k \le n-1)$ so that $V_{p_{k+1}} Y$ appears first, and for all V_i's (either the single occurrence of V_i when $i = 1$ or the $n-1$ copies appearing in the $n-1$ $G_{[2,n]}$'s when $i > 1$), we flatten them so that in V_{p_i}, $p_i a_{p_i, p_{i+1}} X$ appears first. Those flattenings look as follows:

$$
\begin{aligned}
F_1 := 1 \; & \overbrace{p_2 \quad X^n Y^{n-1}}^{U_{p_2}} p_2 \; \overbrace{p_3 \quad X^n Y^{n-1}}^{U_{p_3}} p_3 \; \dots \; \overbrace{\dots p_{n-1}}^{U_{p_{n-1}}} \; \overbrace{p_n \quad X^n Y^{n-1}}^{U_{p_n}} p_n \quad X^n Y^{n-1} \\
F_2 := & \underbrace{1 \; (p_2 \; X(a_{1,\_} X)^{n-1}) Y^{n-1}}_{V_{p_1} Y^{n-1} = V_1 Y^{n-1}} \underbrace{p_2 \; (p_3 \; X(a_{p_2,\_} X)^{n-1}) Y}_{V_{p_2} Y} (V\_Y)^{n-2} \underbrace{p_3 \; (p_4 \; X(a_{p_3,\_} X)^{n-1}) Y}_{V_{p_3} Y} \dots \underbrace{p_{n-1} (p_n \; X(a_{p_{n-1},\_} X)^{n-1}) Y}_{V_{p_{n-1}} Y} (V\_Y)^{n-2} \underbrace{p_n (a_{p_n,\_} X)^n Y}_{V_{p_n} Y} (V\_Y)^{n-2} \\
& \underbrace{\qquad\qquad\qquad\qquad}_{G_{[2,n]}} \qquad \underbrace{\qquad\qquad\qquad\qquad}_{G_{[2,n]}} \qquad\qquad\qquad \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{G_{[2,n]}} \qquad \underbrace{\qquad\qquad\qquad\qquad}_{G_{[2,n]}}
\end{aligned}
$$

We match all letters in F_1 with F_2 similar to before, with three simple differences: we match the Y^{n-1} in U_{p_2} with the Y^{n-1} in $V_1 Y^{n-1}$. For $p_k, k \in \{3, \dots, n\}$, we match Y^{n-1} with the $n-1$ Y's in the $k-2$'th $G_{[2,n]}$, and since such matching occurs strictly after the X^n matching and strictly before the p_k following the Y's, there is no interference and the matching is valid. Finally, we match the $X^n Y^{n-1}$ suffix in F_1 with the $X^n Y^{n-1}$ subsequence in the last $G_{[2,n]}$.

For the other direction, suppose that G does not have a Hamiltonian path starting from 1, and that for the sake of a contradiction, there are flattenings F_1 and F_2 where all of F_1 is matched with F_2. Note that the number of Y's $((n-1) \cdot n)$ is the same in F_1 and F_2, and therefore the matching between them is determined. However, F_2 now has more X's than F_1 due to the duplication of V_i's caused by the $G_{[2,n]}$'s. We show that this is not a problem since the positioning of Y's forces the X's to be matched in the same way as before.

Suppose that T_1 is flattened as $1U_{q_2}, \ldots, U_{q_n} X^n Y^{n-1}$, where $q_2, \ldots, q_n$ is a permutation of $\{2, \ldots, n\}$. For T_2, let $r_1 = 1$, and $r_2, \ldots, r_n$ be (not necessarily distinct) elements from $[2, n]$ s.t. the k'th $G_{[2,n]}$ $(1 \leq k \leq n-1)$ has the prefix $V_{r_{k+1}} Y$. Then, F_1 and F_2 have the following form:

$$F_1 := 1 \overbrace{q_2 \quad X^n Y^{n-1}}^{U_{q_2}} q_2 \overbrace{q_3 \quad X^n Y^{n-1}}^{U_{q_3}} q_3 \ \ldots \overbrace{\ldots q_{n-1}}^{U_{q_{n-1}}} \overbrace{q_n \quad X^n Y^{n-1}}^{U_{q_n}} q_n \quad X^n Y^{n-1}$$

$$F_2 := \underbrace{1\ (f_2\ X(a_{1,\_}X)^{n-1})Y^{n-1}}_{V_{r_1} Y^{n-1} = V_1 Y^{n-1}} \underbrace{\underbrace{r_2\ (f_3\ X(a_{r_2,\_}X)^{n-1})Y}_{V_{r_2} Y} (V\_Y)^{n-2}}_{G_{[2,n]}} \underbrace{\underbrace{r_3\ (f_4\ X(a_{r_3,\_}X)^{n-1})Y}_{V_{r_3} Y} \ldots \underbrace{r_{n-1}(f_n\ X(a_{r_{n-1},\_}X)^{n-1})Y}_{V_{r_{n-1}} Y} (V\_Y)^{n-2}}_{G_{[2,n]}} \underbrace{\underbrace{r_n(a_{r_n,\_}X)^n Y}_{V_{r_n} Y} (V\_Y)^{n-2}}_{G_{[2,n]}}$$

where for $i \in \{2, \ldots, n\}$, $f_i \in \{a_{r_{i-1},1}, \ldots a_{r_{i-1},n}\}$. Since the first Y in U_{q_2} is matched with the first Y in $V_1 Y^{n-1}$, implying $q_2 = f_2$. For $q_k, k \in \{3, n\}$, the first Y in U_{q_k} is matched with the first Y in the $(k-2)$'th $G_{[2,n]}$, which also implies $q_k = f_k$ and $q_{k-1} = r_{k-1}$. This implies Equation 2 and thus the same conclusion as in the previous reduction. □

Note that the first reduction implies that OptimalFlattening parameterized by d_1, d_2 is *not* in XP and thus not FPT (unless P = NP): if it were solvable in XP in time $O(n^{f(d_1,d_2)})$, then we can apply the reduction and solve the reduced instance in total time $O(poly(n)^{f(1,2)}) = O(n^{c \cdot f(1,2)})$ where n is the number of nodes in the graph and the $poly(n) = O(n^c)$ blowup is due to running the reduction. This running time is polynomial since $c \cdot f(1,2)$ is a constant, which is a contradiction to the NP-hardness of Hamiltonian path. Similarly, the second reduction implies that OptimalFlattening parameterized by d_1, b_2 or d_2, b_1 is also not in XP.

5 Remaining questions

Here are the remaining parameterizations of the problem whose algorithms and complexity are still unanswered, and some intuition of why they seem difficult:

- Parameterization by b_1, b_2 :
  - **Algorithms** This admits a naive XP algorithm where we brute force all n^b flattenings of each string and find the flattening-pair with the largest LCS. This makes it easier than the parameterizations of the previous section (which are not in XP). At the same time, the automata-based solution for the parameterization by b_1, b_2, d_i would not work here since we would need to keep track of two stacks, one for each tree string. Simple problems on pushdown automata with two stacks are already undecidable, and only severely restricted variants are decidable.
  - **Complexity** Might be possible to show that it is not FPT by, say, showing W[1]-hardness. Reducing from Hamiltonian path will not work since we know the problem is in XP.
- Parameterization by b_i, d_1, d_2 :
  - **Algorithms** We can first solve this very simple problem: given a string S, and a set of strings $T_1, \ldots, T_m$, find an ordering over T_i's s.t. their concatenation (according to the ordering) yields the largest LCS with S. We can model it as a maximum matching problem: we first guess a partitionining of S into m segments, and have a complete matching from the segments to the T_i's, with weights carrying LCS values of the corresponding string-pairs. The hard part is doing the guessing efficiently. One property we can use is monotonicity: the more we expand the segment, the larger the LCS value can be with another fixed string.
  - **Complexity** The problem is easier than next one, so we should adress the complexity there first.
- Parameterization by b_i, d_i :
  - **Algorithms** The problem is harder than previous one, so we should address algorithms there first.
  - **Complexity** We already know that parameterization by d_1, d_2 is hard. To modify this hardness to show that b_1, d_1 is hard, we need take the first tree string's ability to branch (i.e., bound b_1) and in exchange, give to the other tree string the extra ability of having unbounded d_2. However, it seems that having both unbounded d_2 and b_2 is an overkill, and that having unbounded b_2 is sufficient, and subsequently, adding unbounded d_2 does not add any extra power to the second tree string. So, the first tree string lost the ability to do anything interesting, and the second one

did not gain much.

References

[1] T. W. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," *Sci. Comput. Program.*, vol. 58, no. 1-2, pp. 206–263, 2005. [Online]. Available: https://doi.org/10.1016/j.scico.2005.02.009

[2] T. J. K. E. von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *LCTES*. ACM, 2014, pp. 85–94.

[3] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *CGO*. IEEE, 2019, pp. 149–163.

[4] ——, "Effective function merging in the SSA form," in *PLDI*. ACM, 2020, pp. 854–868.

[5] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. M. Hazelwood, and H. Leather, "Hyfm: function merging for free," in *LCTES*. ACM, 2021, pp. 110–121.

[6] S. Stirling, R. C. O. Rocha, K. M. Hazelwood, H. Leather, M. F. P. O'Boyle, and P. Petoumenos, "F3M: fast focused function merging," in *CGO*. IEEE, 2022, pp. 242–253.

[7] Y. Saito, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "Multiple function merging for code size reduction," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 1, pp. 7:1–7:26, 2025.

[8] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr., "Divergence analysis and optimizations," in *PACT*. IEEE Computer Society, 2011, pp. 320–329.

[9] R. C. O. Rocha, C. Saumya, K. Sundararajah, P. Petoumenos, M. Kulkarni, and M. F. P. O'Boyle, "Hybf: A hybrid branch fusion strategy for code size reduction," in *CC*. ACM, 2023, pp. 156–167.

[10] M. Grohe, "Some thoughts on graph similarity," in *Principles of Verification (1)*, ser. Lecture Notes in Computer Science, vol. 15260. Springer, 2024, pp. 369–392.

[11] M. Grohe, G. Rattan, and G. J. Woeginger, "Graph similarity and approximate isomorphism," in *MFCS*, ser. LIPIcs, vol. 117. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 20:1–20:16.

[12] K. Zhang, R. Statman, and D. E. Shasha, "On the editing distance between unordered labeled trees," *Inf. Process. Lett.*, vol. 42, no. 3, pp. 133–139, 1992.

[13] K. Zhang and D. E. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, 1989.