Optimal Program Refactoring for Size is NP-Complete

Alexandre Blanché and Théo Matricon[1][0000−0002−5043−3221]

INRIA, France theo.matricon@inria.fr

Abstract. Programs are becoming increasingly complex, especially with automatic code generations systems such as large language models (LLMs). A new challenge arises in refactoring the ever growing produced code. We transform the code as a forest of trees, and translate the refactoring problem into a problem of minimizing the description size of the forest by replacing a code part by call to the function. In other words, we replace all the occurrences of a subtree by call to the function and add the function to the forest. We then prove that this formulation of the problem as a decision problem is NP-complete. This suggests that future works should focus on approximate methods.

1 Introduction

Programs are becoming increasingly larger. There is inherent complexity with software. The rate at which we produce code has been ever increasing. With the advent of code generation tools such as large language models (LLMs), the quantity of code produced is skyrocketing. A challenge with this complexity is code complexity and duplication. The idea behind refactoring is to transform the code in order to make it easier to understand, and reduce code duplication. It also aims at increasing code velocity and maintainability.

In this paper we target refactoring as the problem of finding code duplicates and merging them into a function. We see programs as trees and aim at minimizing the total size of all the trees in the forest, which represents the code. The optimal refactor is thus the one which reduces most the description size of the forest. There are multiple close problems of tree isomorphism [2] or largest common subtree [3,4] but they all have one particular property in the sense that they can only find one subtree per tree in the forest whereas this is not the case here we can find many or none. This problem is also of particular interest to the compiler community which is interested in automated techniques for code size reduction [6]. This interest is also shared for compression [1].

In this paper, we show that:

1. the optimal refactoring decision problem is NP-complete;
2. the weighted optimal refactoring decision problem is also NP-complete.

First, we introduce the model in section 2, then we prove the main claim of this paper in section 3 and finally discuss what this implies for refactoring in section 4.

2 Background

A program can be represented as a tree.

Programs emerge from a grammar which is based on a ranked alphabet. THis ranked alphabet defines the primitives or letters that can be used and their arity, a primitive of arity 0 is a constant or a variable.

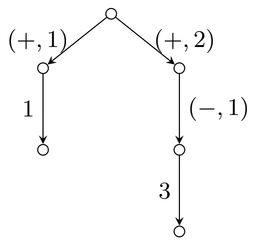

Fig. 1. Example of a tree with edges labelled representing the program (+ 1 (− 3)). Edges from nodes to nodes are labelled by the primitive used and the argument number whereas edges from nodes to leaves are only labelled by primitives.

In order to correctly represent the arguments and their order, instead of tagging the nodes, we tag the edges. See Figure 1 for an example.

One could wonder whether after replacing part of a program with the new learned primitive, this could in turn create a new embedding of the new primitive into the tree. This cannot happen, since a new tag is created for the new learned primitive, thus there is no recursive phenomenon happening.

For a tree t, $|t|$ denotes the number of edges in the tree.

Definition 1. *A tree t embeds into another tree t', noted $t \hookrightarrow t'$, if t is isomorphic to a subgraph of t' and the isomorphism respects the labels of the edges. See Figure 2 for an example.*

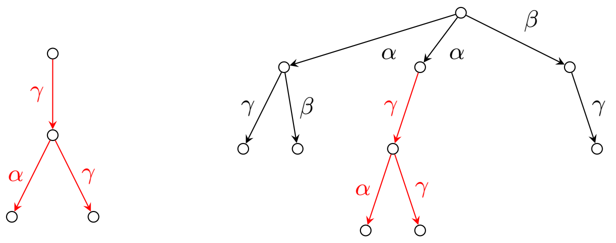

Fig. 2. On the left the tree t. On the right, the embedding of t in t', when t' is the tree from Figure 1.

Note that according to the definition it is possible that a tree can have multiple edge-disjoint embeddings into another tree. We denote by #Emb the function that, given a pair of trees t and t', returns the maximum number of edge-disjoint embeddings of t in t'. We say that t embeds once in t' if $\#\text{Emb} = 1$. Even such a t can embed in multiple ways in t', any two of these embeddings share at least one edge.

3 Proof

Definition 2. *The optimal refactoring decision problem is the following decision problem:*

1. Input*: a set of trees $\mathcal{F} = (t_i)_{i \leq n}$ and $s \in \mathbb{N}$;*

2. Output*: Yes if and only if there exists a tree t such that $score(t, \mathcal{F}) \geq s$ where score is defined as:*

$$score(t, \mathcal{F}) = |t| \sum_{t_i \in \mathcal{F}} \#\mathrm{Emb}(t, t_i)$$

Proposition 1. *The optimal refactoring decision problem is in NP.*

Proof. The certificate we consider is the tree t, the forest $\mathcal{F}$ and the embedding of t in $\mathcal{F}$. The certificate is polynomial in size, and it is easy to compute $score(\mathcal{F}, t)$ and check if it is greater or equal to s.

Proposition 2. *The optimal refactoring decision problem is NP-hard.*

Proof. We will show a reduction from the set cover decision problem to the optimal refactoring decision problem.

The set cover decision problem is the following:

1. *Input*: A set $U = [\![1; n]\!] \subseteq \mathbb{N}$, a family $(S_i)_{1 \leq i \leq m}$ of subsets of U and $0 < k < m$;
2. *Output*: Yes if and only if there exists a set $A \subseteq [\![1; m]\!]$ such that $\bigcup_{a \in A} S_a = U$ and $|A| \leq k$.

The set cover decision problem was proven to be NP-complete by Karp [5] in 1972. Since $k < m$, k can be encoded into unary, since we need more than one bit per S_i. Thus, changing the encoding of k between unary and binary does not change the complexity of the decision problem.

Now, let us introduce the gadgets that we will be using: the $(C_i)_{1 \leq i \leq n}$-gadgets (see Figure 3) and the T-gadget (see Figure 4). For $i \in [\![1; n]\!]$, C_i is a tree, with a root vertex u_i to which are attached:

– a path of length $l \in \mathbb{N}$, whose edges are labeled β;
– $k - 1$ edges labeled α, each of them having m incident edges labeled $\gamma_1, \ldots, \gamma_m$ respectively;
– an edge labeled α, that we will call the "choice branch", with incident edges labeled γ_a if S_a contains i, for each a in $[\![1; m]\!]$.

The integer l is the same among all the C_i gadgets. The edges labeled α, β and γ are called *α-edges*, *β-edges* and *γ-edges* respectively.

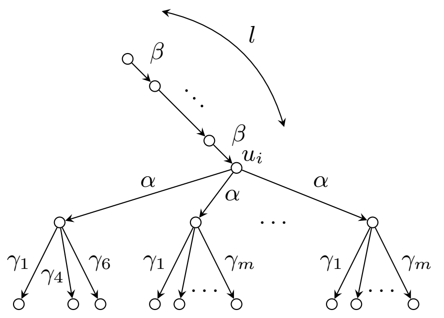

Fig. 3. The C_i gadget, if i belongs to only S_1, S_4, S_6. The choice branch on the left has three γ-edges, and the other $k - 1$ branches on the right have m γ-edges each.

The T-gadget is a tree, with a root vertex v, to which is attached:

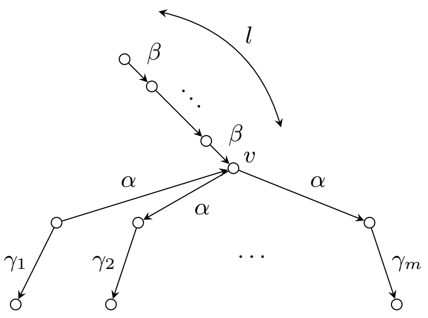

Fig. 4. The T-gadget, with m branches having one γ-edge each.

- a path of β-edges of length l;
- m α-edges, each of them having one incident edge labeled $\gamma_1, \ldots, \gamma_m$ respectively.

We will set the target score to $(l + 2k)(n + 1)$ and prove that there exists a tree t that reaches this score if and only if there exists a solution to this instance of the set cover problem. To this end, the value of l must satisfy some conditions, in addition to being polynomial in the size of the instance.

Condition 1: $(l + 2k)n > 2(m - k)$

Claim. If l satisfies Condition 1, then any tree t that reaches the target score embeds in at least one C_i gadget.

Proof. The T-gadget has $l + 2m$ edges, so if a tree t could cover all the edges of the T-gadget and none of the C_i gagdets, then its score would be at most $(l + 2m)$. Let us show that this score is strictly lower that the target score. Condition 1 states that $2(m - k) < (l + 2k)n$. By adding l to each side, we deduce that $l + 2m < (l + 2k)(n + 1)$.

Condition 2: $l + 2k > kn(m - 1)$

Claim. If l satisfies Condition 2, then any tree t that reaches the target score embeds in the T-gadget.

Proof. Let us calculate an upper bound on the number of edges of the C_i. There are at most $l+k(m+1)$ edges per C_i and there are n of them, so a tree t could cover all the edges of the n C_i and none from the T-gadget, then its score would be at most $(l + mk + k)n$. Let us show that this score is strictly lower that the target score.

Condition 2 states that $kn(m - 1) < l + 2k$

$$\begin{aligned}
\Rightarrow nmk &< l + 2k + kn \\
\Rightarrow nk(m + 1) &< l + 2k + 2kn \\
\Rightarrow n(l + k(m + 1)) &< (l + 2k)(n + 1)
\end{aligned}$$

Thus the maximum score is strictly lower than the target score.

Condition 3: $l + 2k > \frac{2m+nk(m+1)}{n+1}$

Claim. If l satisfies Condition 3, then any tree t that reaches the target score contains l β-edges.

Proof. If a tree t does not contain any β-edge, then by taking all other edges from all the gadgets it can reach a score of at most $2m+nk(m+1)$. Condition 3 states that $2m+nk(m+1) < (l+2k)(n+1)$. So if a tree t does not contain any β-edge, it cannot reach the target score.

In addition, a tree t that contains one β-edge may as well contain all l of them, since it can only increase its score by adding them.

Thus, a tree t that reaches the target score can embed at most once in each gadget. Since there are $n+1$ gadgets and the size of t is at most $l+2k$ (the size of the T-gadget), then t must have a size of $l+2k$ and embed exactly once in each C_i and in the T-gadget.

Let us now choose a value for l.

Claim. There exists a value for l that is polynomial in the size of the instance and satisfies Conditions 1, 2, 3.

Proof. Let us restate the three conditions:
Condition 1: $(l+2k)n > 2(m-k)$
Condition 2: $l+2k > kn(m-1)$
Condition 3: $l+2k > \frac{2m+nk(m+1)}{n+1}$

Choosing $l_0 = mnk$ satisfies conditions 1 and 2, and taking an $l > l_0$ does not break either condition, so let us consider $l = \max(l_0, \frac{2m+nk(m+1)}{n+1} - 2k)$. This value is polynomial in the size of the instance and satisfies all three conditions.

Given an instance $(U,(S_i)_{1\leq k\leq m},k)$ of the set cover problem, let us denote $L(U,(S_i)_{1\leq k\leq m},k)$ the instance of the optimal refactoring decision problem made up of the C_i-gadgets and the T-gadget, with a value for l as described above and a target score of $(l+2k)(n+1)$.

Let us finally prove that our gadgets actually constitute a reduction.

Lemma 1. *$(U,(S_i)_{1\leq k\leq m},k)$ is a positive instance of the set cover problem if and only if $L(U,(S_i)_{1\leq k\leq m},k)$ is a positive instance of the optimal refactoring decision problem, i.e. if and only if there exists a tree t that reaches the target score of $(l+2k)(n+1)$.*

Proof. – $\rightarrow$ Let us assume $(U,(S_i)_{1\leq k\leq m},k)$ is a positive instance of the set cover problem: there exists a set $A \subseteq [\![1;m]\!]$ such that $\bigcup_{a\in A} S_a = U$ and $|A| \leq k$. Let t be a tree made up of a root vertex u to which are attached:
- a path of β-edges of length l;
- k α-edges, each of them having one incident edge labelled respectively γ_a for each $a \in A$.

It is easy to see that this tree embeds once in the T-gadget. Let us show that it also embeds in all the C_i gadgets. Let $i \in [\![1;n]\!]$. Since A is a solution to the set cover instance, then there exists an $a \in A$ such that $i \in S_a$, hence the choice branch of C_i contains an edge labelled γ_a. So the β-edges of t embed in the β-edges of C_i, the α-edge attached to the γ_a edge embeds in the choice branch of C_i, and remaining edges of t embed in the other branches of C_i.
Thus, the score of t is $(l+2k)(n+1)$, since its size is $l+2k$, and it embeds once in the n C_i gadgets and in the T-gadget.

- $\leftarrow$ Let us assume that $L(U, (S_i)_{1 \leq k \leq m}, k)$ is a positive instance of the optimal refactoring decision problem. Let t be a tree with a score of at least $(l + 2k)(n + 1)$. By the previous claims, t embeds once in all the C_i-gadgets and in the T-gadget, and it has l β-edges. To embed in the T-gadget, it must have at most k α-edges, and thus at most k γ-edges. Thus, since its score of at least $(l + 2k)(n + 1)$ and it embeds once in $n + 1$ gadgets, it has exactly k α-edges and k γ-edges.
  Because the tree t has k γ-edges, it embeds in the choice branch of each C_i gadget with a γ_{a_i} edge. Thus, let $A = \{a_i, i \in [\![1; n]\!]\}$. Since t has k γ-edges, $|A| = k$. It is now easy to see that $\bigcup_{a \in A} S_a = [\![1, n]\!]$, since for each $i \in [\![1; n]\!]$, $i \in S_{a_i} \subseteq \bigcup_{a \in A} S_a$.

Theorem 1. *The optimal refactoring decision problem is NP-Complete.*

Definition 3. *The weighted optimal refactoring decision problem is the following decision problem:*

1. Input*: a set of trees $\mathcal{F} = (t_i)_{i \leq n}$, $w : \rightarrow \mathbb{Q}$ and $s \in \mathbb{N}$*
2. Output*: Yes if and only if there exists a tree t such that $score_w(t, \mathcal{F}) \geq s$ where $score_w$ is defined as:*
$$score_w(t, \mathcal{F}) = |t|_w \sum_{t_i \in \mathcal{F}} \#\mathrm{Emb}(t, t_i)$$
and $|t|_w$ is defined as:
$$|t|_w = \sum_{e \in Edges(t)} w(e)$$

Theorem 2. *The weighted optimal refactoring decision problem is NP-Complete.*

Proof. First, we show that the problem is in NP just like for the previous problem. The certificate we consider is the tree t, the forest $\mathcal{F}$ and the embedding of t in $\mathcal{F}$. The certificate is polynomial in size and it is easy to compute $score(\mathcal{F}, t)$ and check if it is greater or equal to k. To prove the NP-hardness, we can reduce the problem from the optimal refactoring decision problem, with $\forall t \in , w(t) = 1$. So the weighted optimal refactoring decision problem is NP-hard, thus NP-complete.

4 Discussion

The fact that the optimal refactoring problem is NP-complete has several implications. First, this implies that in practice we should not aim at finding the optimal refactoring and instead find a suitable approximation. Second, future work can reframe the refactoring problem in other fashions that can be relevant taken into account more human factors.

References

1. Beszédes, A., Ferenc, R., Gyimóthy, T., Dolenc, A., Karsisto, K.: Survey of code-size reduction methods. ACM Comput. Surv. **35**(3), 223–267 (Sep 2003). `https://doi.org/10.1145/937503.937504`, `https://doi.org/10.1145/937503.937504`
2. Droschinsky, A., Kriege, N.M., Mutzel, P.: Faster algorithms for the maximum common subtree isomorphism problem. arXiv preprint arXiv:1602.07210 (2016)
3. Grossi, R.: On finding common subtrees. Theoretical Computer Science **108**(2), 345–356 (1993)
4. Gupta, A., Nishimura, N.: Finding largest subtrees and smallest supertrees. Algorithmica **21**(2), 183–210 (1998)

5. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations. pp. 85–103. The IBM Research Symposia Series, Plenum Press, New York (1972), `http://dblp.uni-trier.de/db/conf/coco/cocc1972.html#Karp72`
6. Edler von Koch, T.J., Franke, B., Bhandarkar, P., Dasgupta, A.: Exploiting function similarity for code size reduction. SIGPLAN Not. **49**(5), 85–94 (Jun 2014). `https://doi.org/10.1145/2666357.2597811`, `https://doi.org/10.1145/2666357.2597811`