
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# HEURISTICS USING MACHINE LEARNING FOR GRAALVM NATIVE IMAGE

**HEURISTIKY VYUŽÍVAJÍCÍ STROJOVÉHO UČENÍ PRO GRAALVM NATIVE IMAGE**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **TOMÁŠ KENDER**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. DAVID KOZÁK**
**VEDOUCÍ PRÁCE**



**BRNO 2025**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Master's Thesis Assignment

164392

| | |
|---|---|
| Institut: | Department of Intelligent Systems (DITS) |
| Student: | **Kender Tomáš, Bc.** |
| Programme: | Information Technology and Artificial Intelligence |
| Specialization: | Cybersecurity |
| Title: | **Heuristics using Machine Learning for GraalVM Native Image** |
| Category: | Artificial Intelligence |
| Academic year: | 2024/25 |

Assignment:

1. Get familiar with GraalVM, especially its Native Image component. Native Image is a modern compiler for JVM bytecode using static analysis. Learn about the different phases of compilation and the internal representation of programs that Native Image uses.
2. Thoroughly familiarize yourself with the compiler configuration and static analysis options used in Native Image. Identify appropriate places to apply heuristics using machine learning. Suitable adepts might include inlining before analysis and partial selective context-sensitive analysis.
3. Suggest at least one heuristic for some area of the compiler or static analysis. Then implement the proposed heuristic.
4. Evaluate the implemented heuristic using a suitable set of benchmarks.
5. Describe and discuss the attained results, along with potential avenues for further improvement.

Literature:


- Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. Proc. ACM Program. Lang. 3, OOPSLA, Article 184 (October 2019), 29 pages. https://doi.org/10.1145/3360610
- Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013). Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581
- Dongjie He, Jingbo Lu, and Jingling Xue. Qilin: A new framework for supporting fine-grained1006 context-sensitivity in Java pointer analysis. In Proceedings of the European Conference on1007 Object-Oriented Programming, volume 222 of Leibniz International Proceedings in Informatics1008 (LIPIcs), pages 30:1–30:29. Leibniz-Zentrum für Informatik, 2022.


Requirements for the semestral defence:
The first two points of the assignment and at least some initial work on the third point.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Supervisor: | **Kozák David, Ing.** |
| Consultant: | Vojnar Tomáš, prof. Ing., Ph.D. |
| Head of Department: | Kočí Radek, Ing., Ph.D. |
| Beginning of work: | 1.11.2024 |
| Submission deadline: | 21.5.2025 |
| Approval date: | 31.10.2024 |

# Abstract

An important part of code optimization is method calls. Each call of a method has an extra computing overhead, which can be avoided by inlining, i.e., replacing the method call with the method body. This thesis is focused on improving the heuristic used to inline methods by the use of machine learning in Native Image, which is a part of the GraalVM toolkit. To achieve this, it optimizes the intermediate representation of the code with graph-based neural networks. To train these networks, we designed a pipeline inspired by genetic algorithms. The pipeline deploys the models it has generated, evaluates them by benchmarking them, and uses the best models as reference for future generations of models. Two variants of model architectures are trained and tested, one is a traditional feedforward neural network and one a convolutional graph network. For each type, we validate the best performing network configurations on a different set of scenarios than the one used for training.



# Abstrakt

Důležitou součástí optimalizace kódu jsou volání metod. Každé volání metody má navíc výpočetní režii, které se lze vyhnout inlinováním, tj. nahrazením volání metody tělem metody. Tato práce se zaměřuje na vylepšení heuristiky používané k inlinování metod pomocí strojového učení v nástroji Native Image, který je součástí sady nástrojů GraalVM. Za tímto účelem optimalizuje vnitřnou reprezentaci kódu pomocí neuronových sítí založených na grafu. K trénování těchto sítí jsme navrhli kódové řešení inspirované genetickými algoritmy. Toto řešení nasazuje vygenerované modely, vyhodnocuje je pomocí srovnávacího testu a nejlepší modely z aktuální generace používá jako referenční pro budoucí generace modelů. Trénujeme a testujeme dvě varianty architektur modelů, jedna je tradiční neuronová síť s dopředným posuvem a druhá je konvoluční grafová síť. Pro každý typ ověřujeme nejlepší sítové konfigurace na nové sadě scénářů, než která byla použita pro trénování.



# Keywords

Java, GraalVM, compiler, function inlining, Torch, graph, neural network, optimization

# Klíčová slova

Java, GraalVM, kompilátor, function inlining, Torch, graf, neuronová síť, optimalizace


# Reference


KENDER, Tomáš. *Heuristics using Machine Learning for GraalVM Native Image.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. David Kozák

# Rozšířený abstrakt

GraalVM je sada nástrojů, která umožňuje kompilovat, optimalizovat a spouštět programy napsané v jazycích, jako jsou Java, Scala, JavaScript, Python nebo Ruby. Mezi jeho silné stránky patří mimo jiné možnost generovat nativní spustitelné soubory pomocí nástroje Native Image. Programy jsou přeloženy do strojového kódu určeného pro konkrétní architekturu spolu s odlehčeným virtuálním prostředím, ve kterém se program spouští. Typická virtuální prostředí interpretují bytecode a používají JIT (just-in-time) překladače, které překládají pouze určité často volané funkce za běhu programu. Native Image umožňuje předkompilovat celý program pomocí AOT (ahead-of-time) překladače, čímž ušetří čas při spuštění programu a jeho následném běhu v systému.

Jedním z optimalizačních úkolů překladačů je hledání míst v programu, kde se vyplatí nahradit volání metod jejich tělem (inlining). Tato optimalizace může size výrazně zrychlit provádění programu, ale v jiných případech jej může i zpomalit. V případě překladačů JIT probíhá překlad za běhu, takže výpočetní a časové možnosti překladače jsou omezené. Překladače proto používají jednoduché a rychlé rozhodovací heuristiky s omezeným kontextem programu.

V rámci této práce se zaměřujeme na problém optimálního inlinování a využíváme k tomu techniky strojového učení. V hlavní části práce se zabýváme tím, jak se ve strojovém učení přistupuje k problematice vkládání kódu do překladače a jaké problémy komplikují nalezení optimálního řešení. Z několika různých možností trénování modelu se nakonec rozhodneme implementovat infrastrukturu pro generování inlinovacích modelů pomocí genetických algoritmů, jejich nasazení na lokální REST API, vyhodnocování na základě počtu dosažitelných metod po přeložení programu a náhodné vyhledávání budoucích možných konfigurací modelů, konkrétně s využitím knihovny implementující algoritmus NeuroEvolution of Augmenting Topologies (NEAT).

Funkčnost tohoto prostředí zpočátku ověřujeme na jednoduché dopředné neuronové síti, která pracuje jenom s několika základními atributy popisujícími volající a volané metody a vrací překladači pravděpodobnost inlinovaní pro každého kandidáta na inlinování.

V další fázi přichází na řadu návrh konvoluční grafové neuronové sítě založené na knihovnách PyTorch a PyTorch Geometric. Převádíme lokální okolí kódu volání metody a tělo volané metody na grafovou reprezentaci, kterou posíláme do modelu. Parametry grafových modelů opět trénujeme pomocí genetických algoritmů, ale dosud používaná implementace NEAT nemá podporu pro modely PyTorch. Vyvinuli jsme proto mezivrstvu, která obě prostředí knihoven propojila a umožnila nám trénovat, nasazovat a ověřovat modely grafového typu založené na tomto frameworku.

Jakmile jsou tyto modely natrénovány, jsou následně validovány na nových scénářích, kde nejlepší modely, grafové i negrafové, fungují téměř identicky. Všechny konfigurace modelu snížili počet dosažitelných metod o zhruba 21% pro každý testovaný scénář, ale zvýšili velikost binárního souboru na dvojnásobek až trojnásobek původní velikosti.

Při vyhodnocení jsme zjistili, že ochranná omezení v překladači, která jsme doprogramovali, aby modely usměrňovali během trénování, ve skutečnosti omezují jejich schopnost učit se rozlišovat škodlivá rozhodnutí. To vede k tomu, že trénované modely spoléhají na tato omezení v kódu a provádějí agresivní inlinování, dokud je kompilátor nezastaví.

# Heuristics using Machine Learning for GraalVM Native Image

## Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. David Kozák. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....

Tomáš Kender

May 21, 2025

# Contents

# Chapter 1

# Introduction

The programming language Java has been around since 1995 [16]. In its long lifetime, it has earned the trust of numerous software companies that have decided to build their services on top of it. The concept of a virtual machine, in which the program code is executed, has been providing a robust multiplatform environment and has been a factor in the success of the programming language. With the evolution of personal computers and their scale, programs written in Java have started struggling with performance issues related to the startup time of service instances, the memory they consume, or the unnecessary computing each of these instances need to do during their run-time. These issues were eventually addressed by GraalVM, which was developed with support for a range of programming languages, including Java. The support for multiple programming languages is achieved through the use of a standalone polyglot language run-time called Truffle [24].

GraalVM [13] has made it possible to save computing power, lower memory requirements, increase security, and speed up the start of services for Java applications. This is achieved with the use of Native Image [23], which precompiles Java bytecode into a native binary for the given architecture and operating system.

The bytecode of the compiled application is analyzed during the Native Image build to find ways to improve performance at run-time. One of these steps is dedicated to the identification of reachable code. As part of this process, some methods are inlined to simplify the resulting graphs of internal program representations for the purpose of further analyzing, optimizing, and performing even more inlining in the later stages of compilation by different inlining models. The simple inliner responsible for the initial quick pass over the graph is based on hardcoded metric thresholds. When fine-tuning these thresholds, we have to always take into account the benchmark that is being used. Each benchmark provides a unique program that uses specific algorithms, and the optimal thresholds for one benchmark set do not provide optimal thresholds for a different benchmark set. In reality, these thresholds will in some scenarios even worsen the performance [10]. The only way to solve this without profile-guided optimizations (PGO) is to inline only when it is very certain that the inlined version of the code will perform better in a broad range of scenarios. We search not for the best static thresholds for each benchmark, but for a model that can develop a transferable understanding of interactions between different attributes and still provide an improvement in execution speed [10].

Inlining is a technique used to reduce the code execution overhead introduced by a method call. It is achieved by copying the function code directly into the calling method. The excessive use of it, however, increases the memory program code takes up, which leads to longer execution time [10]. That is why it is desirable to find a good balance for when

to use this technique. To do that, GraalVM uses a set of metrics calculated based on the structure of the surrounding code for every method call that is being analyzed.

This thesis is dedicated to finding a better way to handle inlining, powered by machine learning without manual thresholds set in the code. Instead of finding a universally applicable threshold setting, we use a neural network to analyze our code and select the relevant features on which to base the final decision. We used a training pipeline that utilizes genetic algorithms to generate the parameters of neural networks, and we have tested two variants of these networks: one less complex, traditional feedforward neural network and one graph-based convolutional neural network.

The rest of the thesis is structured as follows. In Chapter 2, we describe the theory around GraalVM, its alternatives, components, internal code representation, and techniques used for optimization. In Chapter 3, we discuss options to improve the optimization heuristics used by GraalVM. In Chapter 4, we introduce machine learning and the way in which we collect training data, and in Chapter 5, we suggest a training pipeline and a simple feedforward neural network that serves as a baseline for the more complicated graph-based neural network proposed in Chapter 6. Both types of neural networks are evaluated in their own respective Chapter 5 or 6. Finally, in Chapter 7 we draw the final conclusion for both network architectures and results.

# Chapter 2

# GraalVM

Programs written in Java need a special environment based on the principles defined by the Java Virtual Machine (JVM) specification [15]. Such an environment typically also contains standard libraries, and together they form the Java Runtime Environment (JRE). JVM encompasses its own instruction set defined in the JVM standard.

Hotspot [18] is a JVM implementation originally developed by Sun Microsystems and currently maintained by Oracle. In addition to interpreting bytecode, it also contains two compilers, client-side C1 and server-side C2 [18]. They are used for JIT compilation of code fragments that are „hot“[1] based on run-time statistics, with C1 being the faster, but less effective optimizer and C2 being the slower, but more effective optimizer [4] [12].

GraalVM, which is the implementation upon which this thesis is built, is a complex toolkit based on Hotspot. It uses its own toolkit to manage the downloading of dependencies, building and benchmarking, called *mx*[2]. GraalVM supports languages such as Java, JavaScript, Ruby, Python, and LLVM. It allows their mutual interoperability with minimal overhead and compared with Hotspot uses its own just-in-time (JIT) and ahead-of-time (AOT) compiler.

During code execution, the interpreter is going to encounter functions that are called frequently. Such functions are candidates for compiling into native machine code to speed up their execution. The process of compilation takes away performance from run-time code execution and also takes up more space to save the compiled machine code in the virtual machine's cache. That is why the decision about compilation of these functions is a practical trade-off that, in simplified terms, is mainly rewarded for functions frequently invoked [18]. The compiled functions are optimized, among other things, with standard compiler optimization techniques such as loop unrolling, vectorization, constant folding, (partial) escape analysis, inlining, peephole optimizations, dead code elimination, and more [19]. The AOT compiler shares many of these ideas, except that it avoids the performance penalty of having to compile individual functions during run-time interpretation and precompiles the entire code into machine code at build time instead. This shifts the computation away from run-time execution to build time, which reduces the startup time.

[1] Code that is deemed to be executed often and could benefit from being optimized for faster execution.
[2] You can download mx from `github.com/graalvm/mx`.

## 2.1 Native Image

Native Image is an ahead-of-time compiler that can run a static analysis of the program and perform optimization tasks based on this analysis, including preexecution. The resulting artifact is compiled into a standalone executable, which means that subsequent launching of program instances is less expensive and faster. The executable binary contains an embedded SubstrateVM, which acts as a minimal runtime environment for the native program and elimites the need to run any of the bytecode inside a standard JVM [1].

In a regular Java bytecode execution environment, there would be at least 4 different types of components interacting with each other: Java Development Kit (JDK), virtual machine (VM), application, and third-party libraries.

Every component from above exposes a standard interface for other components to connect with. Native Image takes all these components, integrates them together into a single bundle, runs optimizations to allow for the more effective use of this bundle as a whole on the specified machine, and exports the resulting platform-specific code into an executable file.

There are 3 main steps in this process that are repeated in cycles (as seen in Figure 2.1):

- **Points-to analysis (Section 2.3)**: Builds the internal representation of program and detects reachable code.
- **Running initialization code (Section 2.4)**: Executes initializers in classes and prepares heap memory into a state ready for snapshotting.
- **Heap snapshotting (Section 2.5)**: Captures the state of heap memory after initialization and persists it in the generated native file.

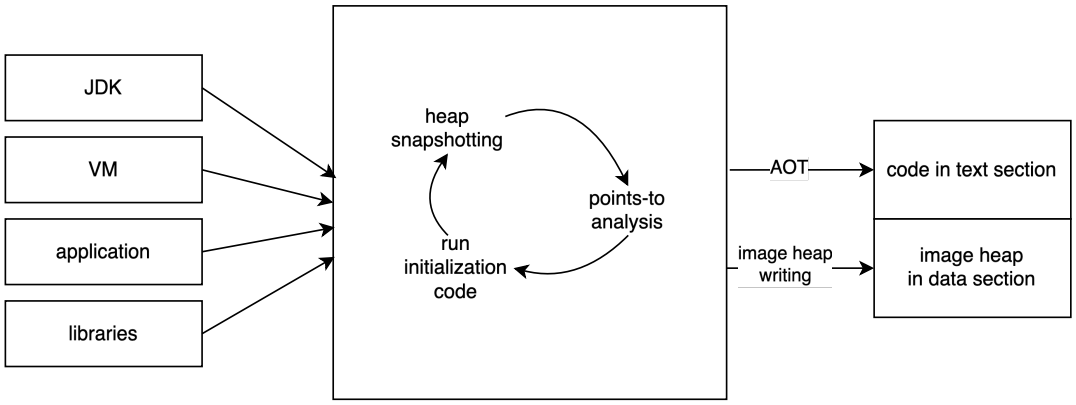


Figure 2.1: Components of Native Image.

Such an approach allows us to assemble an ELF/MachO executable compiled for a specific machine. It contains the native machine code that runs directly on the operating system level (hence the name Native Image). It also contains the heap image, which is an artifact of the code that has been preexecuted during compilation and no longer needs to run at run-time again. The resulting executable is smaller in memory, faster to start up, and also faster to execute compared to bytecode running in a virtual environment.

In addition to all the advantages, there are also disadvantages to consider. By default, there is no JIT compiler in SubstrateVM, so it is not possible to perform additional optimizations once the AOT compilation is finished. Recompiling code into a native image is

also not always possible without having to modify the code first. Even then, the behavior of the program might still be different from what was originally intended.

## 2.2 Internal Representation

When compilers start parsing code, they store the program in an internal representation in their memory. GraalVM uses a directed graph-based structure for this purpose, which is designed to make it easy to apply different transformations to the program during optimizations. As Duboscq et al. [4] describe in their paper, the node types are defined as classes, with the edges between the nodes represented as fields of these classes. They are annotated as `@Input` in the node on the incoming side of the edge. These edges are called *data flow* edges.

In addition to *data flow* edges, there are also *control flow* edges. They are annotated as `@Successor` on the outgoing side of the edge. Control flow graphs start with *BeginNode*, *StartNode* (one per method), *MergeNode* or *LoopBeginNode*, and end with *EndNode* or *LoopEndNode*. *PhiNode* is used to resolve the value of a variable in *data flow* in the event that it can be assigned different values due to, for example, branching.

The nodes can be classified as *floating* or *fixed* based on their role in the control flow. Fixed nodes provide constraints on the flow of the program execution, whereas floating nodes are restricted only by data flow.

The nodes feature *reverse edges* in nodes on the unannotated side of the edge, which are used for backlinking. Thanks to them, the compiler can also access *predecessors* and *outputs*, which are the opposite of *successors* and *inputs*.

In Figure 2.2, we observe a visualization of the main function of a short *HelloWorld* program. The graph starts with a *StartNode* and ends with an *EndNode*. Between them, a public static field *System.out* (identifier 2 in the graph) is loaded, which is of type *PrintStream*. The method *println*, which is defined in *PrintStream*, is invoked (identifier 7) for the loaded field with an attribute *„Hello World“* (identifier 9). Then, *println* performs the process of printing out a string to the standard output. An exception might be thrown in case *System.out* is null and we attempt to call *println* anyway. The invoke node represents a candidate for inlining.

## 2.3 Points-to Analysis

Using static analysis, the compiler optimizes the bytecode before it is run. During the analysis, Graal builds an intermediate representation (IR) of the reachable code to determine the transitive reachability of methods, fields, and classes from a given set of root methods (e.g. `{main}`) [23].

Using the IR graphs of so-far reachable methods, a second graph is incrementally built, called the type-flow graph [9]. The graph uses directed edges to capture the relation between a definition and its usages. Each node contains a list of possible types that is propagated to the nodes that use the current node. This process happens iteratively, and every change in the type list triggers a new propagation. The list of types eventually stops growing, leading to the convergence of the final graph. This state in terms of the analysis is called a fixed point.

Optimizations during AOT compilation are applied using the information gained in this step, e.g., dead code elimination, loop unrolling, escape analysis for detecting the use of an

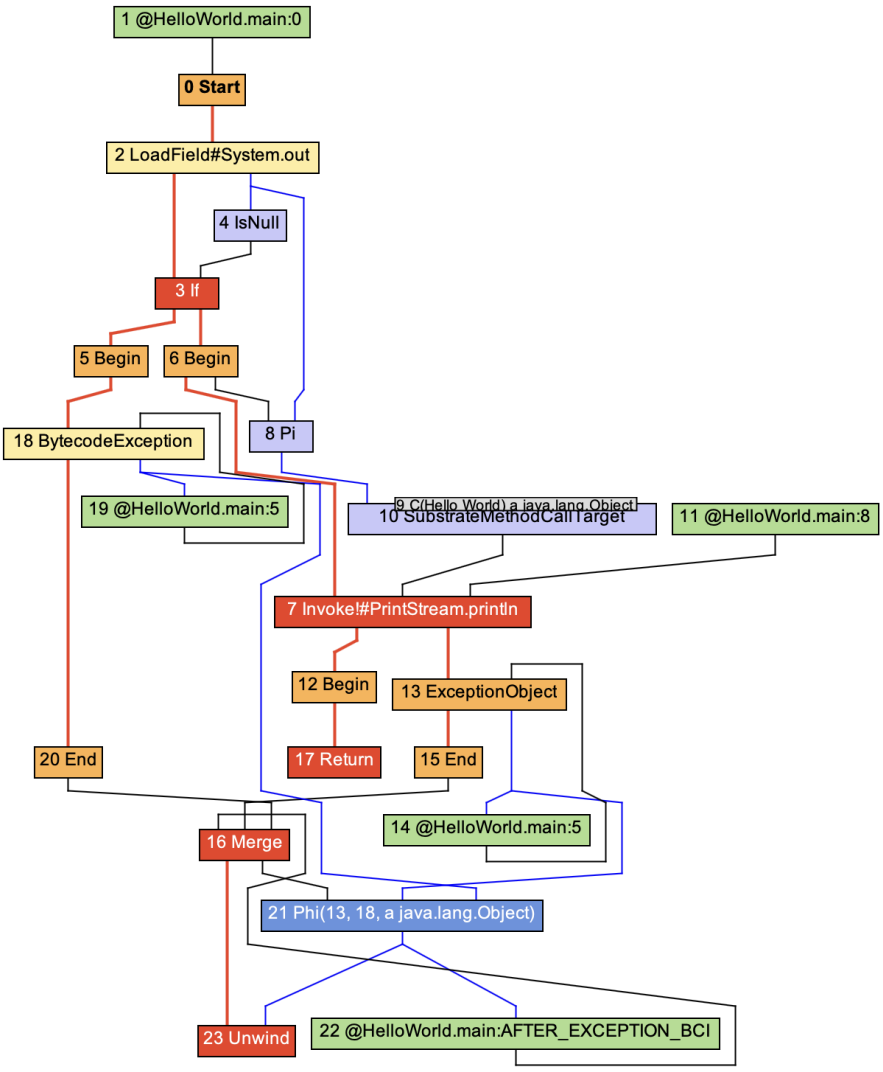


Figure 2.2: Graal IR example for a Hello World native program.

object outside of the allocating method [19], or constant folding of fields that are only read by the nodes using it and never written in by them.

## 2.4 Running Initialization Code

There are two kinds of class initializers in OOP: class and instance initializers. In programming, we usually explicitly define instance initializers in the form of constructors, which execute the code independently for every instance of the given class.

Class initializers, although less frequently mentioned, are important for initializing static variables shared by all instances. The explicit definition of a static initializer is in the form of a block of code (as seen in Listing 1). However, in practical scenarios, we often only specify the values of static variables with their declaration. Some of these class initializers can be executed upfront and do not require run-time state to be present for them to work properly. Such cases are ideal subjects to be included in the image heap.

```
public class Foo {
    private static final UUID identifier = UUID.randomUUID();
    private static final List<StorageBlock> sentences = new ArrayList<>();

    private final Consumer<String> callback;

    static {
        String paragraph = """
            This is an example paragraph.
            It could have been an essay about Graal and Native Image,
            but we choose to keep it short and simple instead.
            Hope you enjoyed it!
            """;

        for (String s : paragraph.split("(?<=[.!?])\\s+")) {
            sentences.add(new StorageBlock(s));
        }
    }

    public Foo(Consumer<String> callback) {
        this.callback = callback;
    }
}
```

Listing 1: **Different types of initializers in action.** Static variables `identifier` and `sentences` can be pre-computed and saved into the binary machine code. On the other hand, `callback` is resolved for individual instances at runtime.

The decision about running the initialization of a class during image build is made by an automatic analysis of dependencies. This works only in some cases; for everything else, the developer has to manually flag the individual classes for build-time or run-time initialization. This is done not directly through the code, but via console arguments of the utility that builds the native image.

## 2.5 Heap Snapshotting

In this step, a heap object graph is built using the points-to analysis described in Section 2.3. The image heap is a graph structure with nodes picked from those that have been marked as read. The root nodes are static fields or methods that contain an embedded constant. All other read nodes are processed as part of a depth-first search and connected to related roots and their subnodes. Possible types of fields are collected and the object graph is pushed back to points-to analysis to start a new cycle of all 3 steps again: points-to analysis, initialization, and heap snapshotting.

Every heap snapshot iteration always starts from an empty graph; it is not possible to reuse graphs from previous iterations. Tracking changes between iterations and adding missing nodes would be much more expensive than recreating the entire graph from the start [23].

The three steps mentioned above are repeated until a final graph converges and points-to analysis no longer produces a different result compared with the previous run. The heap image is then saved in the native image and loaded into heap memory on every startup of the program.

# Chapter 3

# Inlining

The compiler during the compilation process decides, among other things, whether it is beneficial to keep a method call as is or to copy the content of a method at the place where it is called from. In case of a less complex function, that could lead to saving computing capacity by relieving the processor of having to process the overhead of calling an independent function. However, in exchange for that, the program may take up more space in memory. That is because it is necessary to copy the method body and replace the original method call with the entire code of the method. This may happen simultaneously in multiple parts of the program, leading to duplicate code. The job of the compiler is to find a good trade-off between these two properties of a program: the execution speed and space that the code takes up in memory.

## 3.1 Reference Algorithm

The existing solution to the inlining problem is based on multiple different inliners across the compilation pipeline. Our proposed inliners operate on top of the IR graph before static analysis. The original inliner that is already present at this level works based on making a conservative estimate. Its objective is to make a quick decision, which puts a restriction on how precisely it can analyze the entirety of the program. Instead of analyzing the program, it makes a decision based on a quick analysis of the body of the subject method. To avoid accidental inlining caused by the lack of a better analysis, the algorithm uses a very conservative setting that would rather not inline at all than to inline a method that is not supposed to be inlined and cause performance problems.
There are four basic rules which all lead to the decision not to inline:

- **Recursion in function**: A function that calls itself.
- **Function depth exceeds a specified threshold**: Let us use an example to demonstrate the depth attribute of a function: Function A contains an invocation of function B that contains an invocation of function C, which does not invoke any further function. Function A has the depth of zero, function B has one, and function C has two.
- **Function size exceeds a specified threshold**: The number of memory blocks that the definition of the function occupies.
- **Inlining is set as forbidden**: Inlining of a function can be prevented by the programmer via annotations like `@NeverInline`.

## 3.2 Link-Time Optimization

A counter-argument against the use of local conservative inliners is a complex analysis of the IR that processes the entirety of code at once, predicting the places where inlining is desirable. The inliner has access to the entire code structure and provides the best informed decision based on that. That provides an opportunity to inline in a smarter way by taking into account a more global perspective at the IR of the program. A more complex analysis is used as part of Link-Time Optimization (LTO) in GCC[1].

During the initial compilation of a module, the compiler does not see all of the modules at once, and therefore does not know which functions of a module are used externally by other modules. It can see the external functions, but it does not know anything about them. In this phase, the code is parsed into an internal representation (IR) in the form of a GIMPLE bytecode (three-address code) that is gradually optimized down to a set of machine code, a symbol table, relocations, and optionally debug symbols. They are all stored as an *.o* object file. If the compiler is run with a special flag *-flto*, it dumps GIMPLE into the file.

Once all modules are linked to each other by the linker, a global call graph is constructed that encompasses all modules with their function invocations and function definitions. If the previously generated object files contain the GIMPLE IR, the LTO is performed [5]. As part of that, inlinings are performed to further optimize the graph. Information such as function code size, estimated time speed-up, and number of calls is taken into account to calculate the *badness* of a function. Functions are picked for inlining based on their badness until certain limits are hit.

LLVM also has a link-time optimizer in a shared library called *libLTO*, which is called by the linker [11]. The linker first reads both native object files (regular object files) and LLVM bitcode files (object files with serialized LLVM IR) and builds a global symbol table. The information from LLVM files is extracted using libLTO. In the next phase, link-time optimization is performed on top of LLVM bitcode files, which are merged and optimized into a single native object file. During optimization, inlining occurs as part of one of the steps. The inliner uses metrics like function code size, call frequency, instruction costs, function arguments, return value of the function, or profiling data. Different inlining thresholds are applied depending on the frequency of calls to the function (sourced from profiling data) or the presence of an inlining hint for a function [21]. Once optimization is done, only regular object files remain and they are all linked together.

Although GCC's and LLVM's LTO architecture is different from GraalVM Native Image, we will draw inspiration from it when designing our approach tailored to the Graal compilation process.

## 3.3 Analysis of Surrounding

To achieve better scaling capabilities, only analysis of a certain surrounding of the inlined invoke might be preferable. On one hand, we lose information about past inlinings or the global vision of the entirety of the IR graph. On the other hand, we collect context information faster for the method call that is being inlined, and it can generalize even for previously unseen inlining scenarios, since the surroundings of the method call capture less detail than the complex analysis presented in Section 3.2. A smaller scope means

[1] GCC is a compiler for a set of GNU languages, see gcc.gnu.org/

that the limited local information we provide to the inliner will be generic enough to map the unseen scenario to a scenario the inliner knows how to solve correctly, assuming that the captured scope itself contains enough information to decide about the meaningfulness of the inlining in the first place. If not, the resulting model will not generalize well for previously unseen programs. The definition of the scope is arbitrary and that is why it is necessary to run experiments in multiple configurations to get an intuition for how the behavior of inlining models changes with different scopes of available information.

# Chapter 4

# Machine Learning and Datasets

Thanks to large language models and deep learning, the discipline of machine learning has gained widely recognized popularity among the general public. In the last couple of decades, we have been accustomed to building complex conditional rules in our algorithms that work in deterministic, explainable ways to provide a reliable answer to our input. With the inception of neural networks, this programming paradigm has slowly started to change. In the second half of the 20th century, there were multiple periods of excitement and disappoitment that followed each other in a cyclical way [20], advancing the brand new field mostly among the scientific community. Today, it is adopted by software companies across all domains of everyday life.

Previously, software engineers spent a lot of time perfecting thresholds in their fine-tuned algorithms. These algorithms would become more complex as the number of inputs grew and presented a maintenance challenge to preserve their relevance and accuracy. With neural networks, it is now possible to capture complex non-linear relations between parameters and predict the result more accurately with less manual thresholds set by the programmers [10]. Instead, the thresholds are calculated during the training phase of building the model, where the developer provides the training pipeline with sample inputs and desired outputs, with the training algorithm adjusting the internal values used for calculating the output prediction. The technical challenge has shifted from manual parameter fine-tuning in conditions to designing a good neural network architecture for a specific task and collecting high-quality data for training purposes in high quantity. Inlining in Graal presents an opportunity to test this new approach in practice.

## 4.1 Traditional Machine Learning

Most of the problems solved with machine learning have a similar sequence of steps that one needs to take to design a model. One of the first questions you need to answer is what kind of model you are designing. Are you predicting a numerical value? How many values are being predicted? Are you performing classification? How many possible classes are there? What kind of information do you have available that you can feed into your future model? Inputs and outputs put constraints on the designs and architectures with which we can experiment.

Our use case assumes models making inlining decisions for each invoke found in the internal graph representation of the code. The output is thus a boolean decision, we want to either inline or not to inline. In terms of machine learning, this task is a subcategory of

classification. Our network outputs a single number in the range of 0.0 (0%) to 1.0 (100%) based on how confident the model is that we need to inline. The certainty returned to our inlining API maps to `True` if the confidence exceeds 0.5, otherwise it is `False`.

For inputs, we need to collect all the relevant numerical or categorical data about the invocation context that might help the network to make a decision. These data are then multiplied and summed with different biases, passed down the neural network node-to-node with further multiplications and sums taking place at each layer of neurons. After each layer, an activation function such as *softmax* or *sigmoid* is applied to the values before they are passed further. In the final layer, the activation function normalizes the value to fit in the range <0.0,1.0>.

Once the final activation is performed during training, we can compute the error between the predicted output of the network and the annotated training data and use backpropagation to adjust the parameters of the network, so that the error decreases for future network predictions. If we have a diverse training set of inputs and outputs, the model will learn to generalize and recognize patterns in input data.

This has been a general approach to solving similar machine learning problems in other domains. As we are about to learn, in our case, it is going to be more complicated.

## 4.2 Programs for Analysis

For any neural network by design, it is not possible to prepare a model without a sufficient amount of training data. That is why, before we train our network, we collect data and prepare a dataset.

Based on experiments with example programs in the GraalVM demo repository[1], we have learned that even with very small example projects in Java, thousands of inlining decisions are being made at compile time. That means that we do not necessarily need complex projects for our training dataset to teach the model the logic behind optimal inlining.

In Table 4.1 we can see the approximate time it takes for an inlining decision to be made using the original reference algorithm described in Section 3.1.

| | Inlined | Not inlined | Avg time taken [ms/decision] |
|---|---|---|---|
| hello-world | 72724 | 1084 | 5.0 |
| add-jfr | 77336 | 1100 | 8.5 |
| add-logger | 72755 | 1084 | 6.2 |

Table 4.1: **Inlining statistics for the reference algorithm.** Examples from the GraalVM demo repository were compiled to test its capabilities.

Our custom inliner is not going to replace the reference inliner, instead it is going to complement it and decide about the inlinings of previously uninlined invocations. That is why, although Table 4.1 gives us a perspective into the performance of the existing inliner, its speed and inlining scenarios it performs are not comparable to the inlinings our proposed inliner performs.

The models we train are deployed outside of Graal's codebase, which means that on top of the delay introduced by the original inliner, we get an extra delay caused by REST

[1]GraalVM demo repository can be found at github.com/graalvm/graalvm-demos

overhead and the calculations inside of the model itself. The resulting speed of inlining decision-making is therefore expected to drop significantly at compile time. Once a specific model configuration is found, the model inference could be moved to Graal to eliminate the overhead. This aspect is beyond the scope of this thesis, and our goal remains to explore the applicability of machine learning models for inlining.

Based on the observations above, we can choose to use small programs to gather data for training. Knowing this, we can turn to open-source datasets used to benchmark different JVMs. One of such data sets is Renaissance[2]. It contains programs used to solve a wide range of complicated problems, providing different algorithmic ways to approach the same problem and write the code that solves it. The proof of this is that it contains much more hot code (frequently executed segments of code) and hot methods than its benchmarking peers [17]. Hot method invocations are ideal for learning to inline, because they are executed frequently and create a lot of overhead while jumping in program code, hence making the perfect dilemma for whether to inline or not. Mastering such scenarios should give us a good chance that whatever kind of codebase our model is deployed into to do inlining, the model should already be familiar with similar code patterns and decide accordingly.

## 4.3 Annotating Inlining Decision

Sample program source codes are not enough, a mechanism is needed that will decide whether the inlining in a specific place is desirable. Remember, the reference algorithm is conservative and does not inline unless its strict, yet information-wise shallow conditions are met.

Unlike the graph analysis in the reference code, where there is no place for lengthy operations and decision making cannot be thorough by design, during the training we have an unlimited time per each decision to make the right choice. That is why we are allowed to perform more static analysis on top of the method, including even dynamic checks. An example of that would be the comparison of the runs of an inlined variant of a function and an uninlined variant. Thanks to this, we are able to explore the effect of riskier inlinings that would be refused by the original algorithm. During inference, we can put the learned knowledge to good use and make a better informed estimate that will lead to faster code execution on the virtual machine.

We have our program dataset that we will use as input for our training. What we do not have are the desired inlining decisions we want to teach our network. As it turns out, if we want to propose a deterministic way to make yes/no inlining decisions for our dataset, then we would end up writing some kind of hardcoded heuristic similar to what the baseline is, except that ours would inline more aggressively. We would then test the performance and expect it to perform better than the baseline.

The natural way to approach this issue is to make a version of the graph where the node is inlined and another graph where the node is not inlined, test the performance of both versions, and choose the better performing decision. Repeat this for every decision in a program, and we have our training annotation for a single graph. Repeat for every program, and we have our training annotation for the entire dataset. The problem that Kulkarni et al. [10] have found out about is that an individual inlining decision has too insignificant an impact individually, but might affect any subsequent inlinings elsewhere. We have learned from Table 4.1 that even in small programs consisting of a few lines, there are

[2]Renaissance dataset is available at `renaissance.dev`

thousands of inlinings to be made during compilation thanks to the programs using parts of the standard library and including the runtime environment. Testing individual inlinings thus does not provide enough information. The solution to this problem proposed in the study is to use an algorithm called *Neuroevolution of Augmenting Topologies* (NEAT) [6]. It is a form of genetic algorithm that is used to construct a neural network for a specific task by evolution.

Cooper et al. [2] managed to prove that it is possible to find an optimal inlining heuristic for every piece of code, although it is computationally expensive. What we are trying to do here is something slightly different. All mentioned papers describe an exploratory approach to heuristic parameters. In our case, we are trying to explore different inlining combinations in a random way. We can gather a collection of optimal inlinings for our dataset and use them to train our network. Since our network is designed to understand the code dependencies, it might be able to learn a general explanation of why certain inlinings work and why some do not, which cannot be captured with hardcoded local method heuristics that are being used now.

The other option is to generate model configurations that capture random logic and try to identify the configuration that captures an existing real pattern.

There are 3 general approaches to building a model to solve inlining:

1. Try every inlining individually, measure its effect on performance, label the training dataset with optimal decisions, and learn to imitate such decisions.

2. Use genetic algorithms to find a near-optimal solution.

   (a) Generate models, create solutions, measure performance, keep the best performing one.

   (b) Generate solutions, measure performance, train a model to imitate the best one.

The approach 1 is the only one that would deterministically find the optimal solution in an isolated environment, unfortunately, for reasons described in Section 4.3, inlining is not performed in such an environment.

An alternative to a deterministic approach is to use genetic algorithms (discussed in detail in Section 4.5). Genetic algorithms are a family of solutions that are designed to generate different configurations by random mutations, crossovers of existing configurations; and finding the one configuration that makes the best decisions in a real-life scenario. This is beneficial in scenarios where it would be difficult to directly arrive at the objectively best decisions in an analytical way, which is the case here. However, even if we make the decision to use genetic algorithms (GA), there is still a decision to be made about which problem we are solving with GA.

All GA papers mentioned so far unanimously implement approach 2a. The fact that they managed to find success and achieve a speed-up of 11% [10] means that this path has already proven to be promising.

There is also the approach 2b. It has not been mentioned in any of these studies and is probably due to the fact that it could suffer from issues similar to approach 1. We attempt to design the approach 2b to find out more about why this could be the case.

## 4.4 Naive Approach to Optimal Inlinings

First, we export our current inlining decisions for each method call. That is done by logging every inlining made during the building of the native image for a selected program. Every

analyzed method call can be identified by the parent method, where the method call resides, and the index of the currently parsed bytecode instruction in the parent method. After that, we can set up a genetic algorithm to generate mutations of these combinations of inlining decisions.

Once that is done, we can benchmark these solutions. For that, we export our mutated solutions, which Graal is going to read during the next run of the same benchmark. This time, it will not be inlined based on reference heuristics, but it is going to decode the mutated solution and find the correct method call identifier and the inlining decision generated by the genetic algorithm. This step is repeated for every inlining decision in the code.

After that, the benchmark is run, the performance of every solution is measured using the same test suite, and we pick the best solution for each code.

This approach assumes the reproducibility of each sequence of inlinings. However, this is not the case. Each compilation introduces a randomness factor that leads to a slightly different number of inlining decisions being made in a different order on a different subset of invokes. The randomness aspect is further explored in Section 5.4. It discouraged me from continuing with this approach and instead made me implement the safer variant of approach 2a.

Unlike replicating pre-generated inlinings, using a random model to predict multiple inlinings is less prone to break the compilation of a program in case some previously unexpected inlining decisions appear in the current instance. In case of replicating inlinings, you need to make an uninformed guess about each individual inlining, whereas in a model-based solution, with each call to the same model you make a similar inlining decision as the one you have probably already made elsewhere in the code for a different invoke with similar characteristics.

## 4.5 Pipeline with Genetic Algorithms

Our approach consists of building up a training architecture based on the NEAT algorithm [6]. We will adapt a traditional flow of tasks used by genetic algorithms to our use case, which repeats a set of steps:

1. **Initialization**: The initial random population of a specified size is created.
2. **Fitness Function**: Genomes in the population are individually evaluated using a single numerical metric that we want to minimize or maximize.
3. **Selection**: Genomes with the best fitness form the parent pool for future new genomes.
4. **Crossover**: Random parents are chosen to create their offspring genome, which is a combination of their parents.
5. **Mutation**: The offspring is further modified using a random uniform distribution to create a unique genome for the next generation.
6. **New Generation**: Population for the next iteration is made up of the best performing genomes from the previous generation, plus the newly generated offsprings.

In the context of genetic algorithms, we use the term population to describe a set of genomes, where each genome is a potential solution. As mentioned previously, we are using

the NEAT algorithm to find a neural network configuration, so each genome in our case stands for a standalone combination of biases and weights of a neural network. In each iteration, we generate a population of a specified fixed size and group genomes into species based on the similarity of the genomes. The similarity is evaluated using a distance metric, which needs to be lower than a configurable threshold. We evaluate each genome using a fitness function, preserve the best performing genome from each species (the behavior is called elitism), and complement them with newly created genomes until we reach the desired population size for the next generation.

The new genome is created by sorting the previous generation by their descending fitness, picking $n$ best genomes (survival threshold), selecting two random parents from the best genomes, and creating a new genome by combining the parents into one. Further random mutations of the parameters take place to ensure that we do not run out of unique parent genomes in later generations. These mutations include adding and removing nodes in the neural network, adding and removing edges between nodes, and also modifications to the weights of edges and the biases of nodes. All these mutations are controlled by a uniform distributional random generator, and the chances of each kind of mutation are modifiable by setting the thresholds in our configuration file.

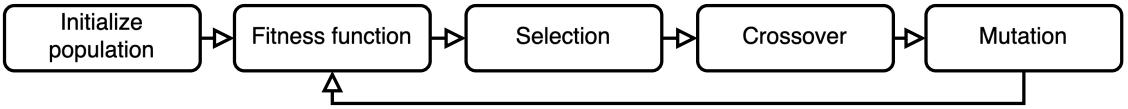


Figure 4.1: Steps during the training with Genetic Algorithms.

Although each step presents a generic set of challenges, there is one that is unique to the inlining domain. Genetic algorithms always need a fitness function to tell which solution is better and which is worse. We have already established that we want to use benchmarks to evaluate solutions, as it is not possible to tell if an individual inlining is right or wrong in the context of all other inlining decisions that have been made, so it is desirable to incorporate their results into the fitness function. How do we do that?

We draw a line between two distinct services. The first one is responsible for running the NEAT algorithm to find the right neural network. The second one runs the benchmark. We connect them by replacing the fitness function in the training service with a function that first opens up an API endpoint with the currently examined network, then runs the benchmark in a subprocess. The benchmark will first run a modified version of Native Image compiler that, instead of running just the original heuristics for inlining, makes an additional pass over the invokes in the IR and make an API call to the endpoint we have previously opened for each of them. After the compilation, the benchmark tool tests the performance of the binary produced by the compiler. Once the benchmark is finished, its output is captured and the performance metrics are parsed back into the training service, where they are used as the fitness function.

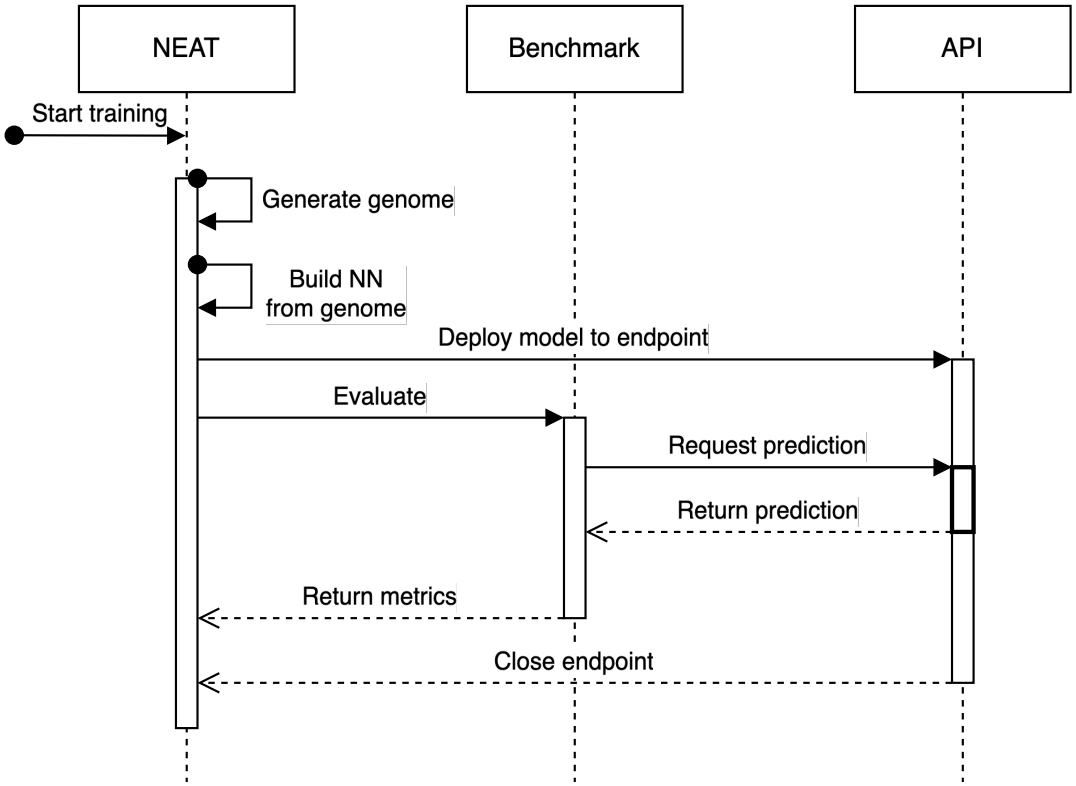


Figure 4.2: Sequence diagram of the evaluation pipeline.

Genomes with the best fitness are selected to succeed peers from their generation. Choosing what numerical metric to use as fitness is not straightforward. We have the option to focus on the time it takes to execute the binary. However, a quickly executed binary will not be helpful if the resulting binary takes up too much space in memory. Another metric tracks the number of reachable methods left in our IR after optimizations. Intuitively, fewer methods should mean faster execution and more space taken up. However, in the same way as the bigger binary does not necessarily have the faster execution time on a machine, lesser reachable methods in an IR do not necessarily imply either of the other two metrics. Using either of the metrics as the fitness function has direct consequences that lead to different results. We will revisit this topic in depth in Section 5.4.

# Chapter 5

# Implementation

In this chapter, we dive into the implementation details of the architecture described in the previous chapter. We describe the implementation of the individual nodes needed for our pipeline based on the proposed architecture in Section 5.1. For the NEAT client, we have implemented a Python service that utilizes the NEAT library to run model training. NEAT takes care of the control loop that generates individual populations for each generation, picks the best individual genomes from each, and performs mutations that make it into the next generation.

Each evaluation of a genome is handled by a custom callback that deploys a neural network with weights and biases loaded up from the generated genome to a REST endpoint on the local host. It launches a benchmark subprocess that compiles programs using our newly deployed endpoint, measures metrics of the compiled binaries and parses the benchmark statistics back into the evaluation callback, inside of the NEAT client service. These statistics are used as the fitness function, which is returned to NEAT's control schema. The endpoint is built on top of *FastAPI*[1], *Asyncio*[2] and *Hypercorn*[3] libraries. Asyncio is used to schedule the API-deployment thread (running in parallel to the master thread), where Hypercorn starts a FastAPI endpoint that exposes models to other applications on the local host.

The reference environment used in this thesis is powered by Graal JDK Community Edition 21, Python 3.11, and Conda. For the exact list of libraries and their versions used to carry out experiments, refer to file `requirements.txt` from Appendix D.

## 5.1 Pipeline Prototype

The proposed approach encapsulates two separate issues:

- Modifying Graal to call a model deployed to an endpoint when inlining. Encoding the internal representation of code inside Graal into a format that can be used as input for the neural network.
- Writing a pipeline that controls generation of neural network configurations using genetic algorithms, their deployment, and the launch of a benchmarking tool that measures the inlining performance of the modified Graal inliner.

[1]The FastAPI repository can be found at github.com/fastapi/fastapi.
[2]The documentation of *Asyncio* can be found at docs.python.org/3/library/asyncio.html.
[3]The documentation of *Hypercorn* can be found at hypercorn.readthedocs.io/en/latest/.

To build our solution in an iterative way, we first build a pipeline for training simple feedforward neural networks and evaluating their performance. The models produced by this prototype use the numerical metrics for their input that Graal already uses in its original inliner, described in Section 3.1. Since we are using the same input data for making decisions as before, intuitively, we do not expect to make much better decisions than the original inliner that is based on individual static thresholds for each of the metrics.

By replacing thresholds with a model and thus obfuscating the decision-making process, we might be able to learn more complex patterns than a human-readable set of rules can achieve.

The input contains 4 values, with two different origins of information:

- invocation node metadata: **estNodeCycles**, **estNodeSize**
- invoked method: **codeSize** (size of bytecode in bytes), **maxStackSize** (stack slots used in bytecode)

The invocation node metadata come from the annotation that is manually assigned to each node type and is determined by invocation type. Its value is determined on the basis of domain knowledge and previous optimizations performed on Graal.

Data about the invoked method are calculated based on the internal representation of the method. It clarifies the nature of the inlined content, while the type of invocation node determines the way it is used in the parent context.

To retrieve information about the invoked method, we generate the method's Graal IR before deciding whether to inline the method in the place of its invocation. During IR generation, inlining is performed on the underlying method. This underlying method may also contain invocations of different methods that need to be processed before inlining. This implies that the deepest, lowest-level invocations are processed first, with the analysis making its way up the tree structure until it finally emerges in the highest-level graph. As we will learn in Section 5.2, we need to establish certain hand-crafted limits on the inlinings to spread their localities throughout all node trees and avoid their concentration on certain types of low-level methods and their Graal IR.

The model used for our prototype is a feedforward model bundled within the NEAT framework. In the framework example configuration, it was initially used to solve a XOR circuit simulation problem. For the original configuration, refer to the file `config-xor` from Appendix D. It offers customizations via its configuration file, which we adjust to support our use case. For our metrics, we need the input vector to be of length 4. The single output is the probability of inlining in the range <0.0,1.0>. The feedforward model offers an option to use one level of a hidden layer with a configurable number of nodes. This increases the complexity of relations between the inputs and outputs that the model architecture can encapsulate. However, it also increases the number of optimal parameters we generate for these nodes; with 1 extra bias per each hidden node and 1 edge weight for every combination of input node and hidden node, and hidden node and output node. For the purposes of a prototype solution, we set the hidden layer to use 3 nodes. The modified network architecture looks as shown in Figure 5.1. The population size for each generation is 10, with two genomes with the best fitness of their generation being transferred into the next generation. The other 8 genomes are a result of the two elite genomes mating with each other. An additional random mutation is applied to their offspring, which involves changes to the weights of their edges and the biases of their nodes. The probabilities of these mutations are the same as in the original XOR feedforward configuration, with the

exception of the removal of any direct edges between the input and output nodes by setting the initial state to `full_nodirect`. The input nodes are only connected to the hidden nodes, and the hidden nodes are only connected to the output nodes. This places more emphasis on the hidden layers and the patterns they can learn with the limited training opportunities we have - we only run our benchmarks for 5 generations due to restricted computing resources. For a complete configuration, refer to the file `config-feedforward` from Appendix D located in the root folder of the provided NEAT pipeline codebase.

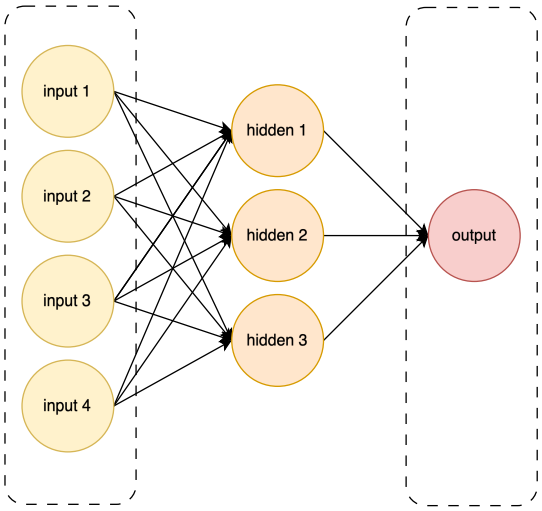


Figure 5.1: **Architecture of the prototype model.** Inputs are a collection of invocation metadata and data about the target method's IR graph.

As proposed in Figure 4.2, our pipeline will be controlled by a Python instance that will implement training and evaluation data. For training, we are using the NEAT framework. It contains many sample configuration files to set up the desired specifications of the models that should be produced by the framework.

In our case with the feedforward model, we have four input values and a single output that contains the inlining decision. A hidden layer has been added to add complexity to the way information from 4 inputs is combined into a single output. Since inlining is a boolean decision, we use a sigmoid activation function in the final layer. The resulting value is in the range of 0 to 1, which is transformed into a boolean with 0.0-0.49 being mapped to *false* (no inlining) and 0.5-1.0 to *true* (inlining).

The chosen fitness function is the number of reachable methods as determined by the static analysis, which we aim to minimize. We switch the fitness criterion to minimization and remove any fitness threshold. Manual safeguards have been put in place for the kind of functions and invokes that can be inlined, since their inlining may lead to the program crashing during execution. These safeguards involve checks for the current size of IR graphs or to check that the inlined function is not tagged as native.

Despite that, it is possible that the model might learn to inline different invokes that have not been expected and masked out before, which is why there is also an automatic error feedback mechanism implemented. In case benchmarking fails, the reported fitness is a disproportionately high number, to discourage the model from learning to inline such methods.

## 5.2 Inlining Safeguards

As mentioned in Section 5.1, several safeguards have been implemented to minimize the likelihood that inlining causes a compilation or run-time crash. The checks are as follows:

- Current IR graph consists of less than 4500 nodes.
- The invoke of the method is direct.
- The invoke of the method is flagged to use for inlining.
- The target method does not have a `@NeverInline` annotation.
- The target method is not native.
- The target method is not excluded from inlining and has associated bytecode.
- Our inlining model returns true for its inlining.

If any of these checks fail, the inlining is denied without calling the model for a prediction.

## 5.3 Model Deployment and Benchmarking

To test the performance of every model, we use the benchmarking functionality of the `mx` tool built for the purpose of developing Graal. From the master pipeline process, a subprocess is launched that changes its working directory to `graal/vm` and runs the following command from there:

```
mx --env ni-ce benchmark "renaissance-native-image:akka-uct"
```

Listing 2: Running a benchmark for *akka-uct* in the environment of Native Image.

The benchmarking is run for every selected training scenario separately. Only a subset of scenarios is used for training, as the training runs on a personal computer with limited compute: *akka-uct*, *db-shootout* and *dotty*. For every scenario, Native Image builds an optimized native code for the program that needs to be compiled. During one of the first phases of optimization, inlining decisions are made.

The class responsible for this decision is `InlineBeforeAnalysis` from Appendix D. With every call to the method `shouldInlineInvoke`, an HTTP POST request is made to a port on localhost, where our model is deployed. After modifying Graal like this, we rebuild it by calling `mx build` as in Listing 3. Note how we are building the source code into environment `ni-ce`, which is the same environment as the one used to run the benchmark in Listing 2. It stands for the Community Edition of Native Image.

This introduces the issue of artificial delays caused by making a REST round trip mentioned in Section 4.2 during the compilation of a program. Based on the measurements made using an example request in Listing 4, REST alone can account for around $\tilde{4}2$ ms of a delay for each inlining decision made. As we have learned, an entire inlining decision in Graal can be made in 5 ms, presenting a significant performance setback.

```
mx --env ni-ce build
```

Listing 3: Building Graal Native Image from source.

```
{
    "estNodeSize": 2
    "codeSize" : 2,
    "maxStackSize" : 5,
    "estNodeCycles": 2
}
```

Listing 4: An example of a payload sent to the model from Graal during the compilation.

However, there is a reason the model needs to be kept outside of Graal. We generate individual models through our pipeline client and once it is deployed, we launch the benchmark without recompiling the Graal compiler's internal source code. Once the optimal model is identified, there is no practical reason to keep it outside the compiler's original codebase. At that point, the REST overhead disappears.

Once inlinings are performed and all other optimization passes are finished, the benchmarking tool runs a performance test of the generated native code and stores the measured metrics in `bench-results.json`.

## 5.4 Fitness Function

The pipeline waits for the benchmarking subprocess to finish and loads the metrics collected from `bench-results.json`. We have handpicked four important metrics: **binary size**, **max-rss**[4], number of **reachable methods**, and **execution time**.

Each of them represents a different way of measuring the performance of the inlinings made, and we are looking for a specific subset of them with a formula, how to combine them into a single number to score the quality of a model. In our prototype, we will only use the count of reachable methods.

The reason for choosing this metric instead of the execution speed of the native image or binary size in memory is that there are two inlining stages in the pipeline during optimization. The first one, which we are modifying, is supposed to reduce the size of the analyzed internal graph before the actual analysis and optimization takes place. In the second inlining stage, a more aggressive inliner takes over. This inliner has a larger impact on the execution speed of the final binary program.

We have observed that even when no inlinings were performed by our model, the final execution speed, size of maximum resident set, size of binary and to a lesser degree count of reachable methods vary between different runs of the same program. There is a sense of nondeterminism that causes inlinings to happen in different order, which originates before compilation makes a pass through our inlining layer. Because of that, even if our model denies every inlining opportunity, the actual number of inlining decisions the model makes

[4] en.wikipedia.org/wiki/Resident_set_size

is different with each run. Assuming that we are building a single executable file out of *akka-uct* and the model refuses every inlining opportunity, our model is asked about inlining approximately *556300* times, as demonstrated by our tests in Figure 5.1. Each run differs by a couple of tens of model invocations. The following introduction of minor changes between IRs explain the differences between the metrics, despite our model behaving the same in each instance. The same can be observed with a model that always inlines for as long as is allowed in Figure 5.2.

| Run | # of refused inlinings |
|---|---|
| #1 | 556296 |
| #2 | 556305 |
| #3 | 556306 |
| #4 | 556322 |
| #5 | 556284 |

Table 5.1: Decisions made for benchmark *akka-uct* by a model that refuses each inlining.

| Run | # of accepted inlinings |
|---|---|
| #1 | 606864 |
| #2 | 606831 |
| #3 | 609464 |
| #4 | 606811 |
| #5 | 606950 |

Table 5.2: Decisions made for benchmark *akka-uct* by a model that allows each inlining.

After running the training for 5 generations, 10 different genomed models each, for the benchmark *akka-uct* from the *Renaissance Benchmark Suite*, we can observe a continuous improvement in the fitness score of the best performing neural network configuration from each generation.

| Solution | Binary size [MB]↓ | Max-RSS [MB]↓ | Reach. methods↓ | Execution [s]↓ |
|---|---|---|---|---|
| Reference | 28.897 | 2,737 | 35,386 | 30,619.179 |
| Model | 91.563 | 3,010 | 27,779 | 29,499.999 |

Table 5.3: Metrics comparison for reference (the original approach) and the best feedforward model after 5 generations of training on benchmark *akka-uct*.

Based on Figure 5.3, we demonstrate that the pipeline produces better neural network configurations as training progresses, we must solve another problem. Every performance test of a neural network is done over a benchmark suite containing a set of problems, and while one network might perform better for one specific benchmark, it can underperform for others. A method to combine multiple benchmark scores into one is needed. For simplicity, the prototype will use a relative improvement of a single selected metric over the baseline solution. We also implement fitness caching for network configurations that make it into the next generation without a change in their parameters. When the parameters do not change, it does not make sense to run benchmarks for training scenarios again, despite the possibility of getting slightly different metrics due to the randomness issue discussed in this section. After running the benchmarks for a genome, its fitness value is persisted

in the object instance of the genome. When the same instance of a genome appears in the next population, the pipeline notes that the genome already has a fitness set and skips the deployment of the network configuration, launch of the benchmarks, and calculation of the total fitness. Since two out of ten genomes that move into the next generation, from the second generation onward, two of the ten genomes that form the current population are instantly evaluated by the use of caching.

## 5.5 Evaluation of Non-Graph Model

In this section, we evaluate the metrics collected during training between different network configurations. In Section 5.5.1, we compare the number of reachable methods (the fitness function) for each network and benchmarking scenario. We do the same for the binary size in Section 5.5.2. In Section 5.5.3, we validate the best performing models from training using these two metrics again, but this time for previously unseen benchmark scenarios.

### 5.5.1 Fitness Progression

Our simple fitness function has been the number of reachable methods. Based on how genetic algorithms evolve the network configuration in the direction that allows for lower fitness, it should not come as a surprise that our final generation of networks have, in fact, lower total fitness than the networks from the first generation. As for `akka-uct` and `db-shootout`, we found networks that inline aggressively in the first generation already. The problem is that the third benchmarking scenario, `dotty` crashes for almost all of these aggressive models.

In Section 5.1, we have mentioned that we have limited the number of nodes in a single IR graph. When a graph exceeds a certain number of nodes, further inlinings are no longer made by our model for the current graph. This limit has been fine-tuned by testing the `akka-uct` scenario with different numerical limits until we found a value that allows to inline as much as possible without crashing the program. This limit safeguarded the stability for `db-shootout` without us explicitly tuning the value for this benchmark. That is not the case for `dotty`, which is a Scala program with much more complexity than the other two scenarios.

As we can see, the green column for `dotty` fitness is completely missing from most of the genomes in Figures 5.2 and 5.4. This happens when the program crashes either during compilation or execution of the benchmark. In case of crashing during compilation, the system runs out of available RAM. In case of crashing during benchmarking, the program usually runs out of heap memory allocation. The safetyguard thus failed to keep the inliner in check for `dotty`, the program crashes without producing metrics (including the number of reachable methods), and that results in the pipeline attributing infinite fitness to the genome. A real number needs to be assigned for fitness, so for practical reasons, we arbitrarily set it to `999,999`. We want to minimize the number of reachable methods, so if any of the network configurations crash the benchmark for any of the scenarios, the resulting fitness is so high that there is a very low chance such configurations can survive to the next generation. This benefits configurations that might not inline too much, but they also do not crash any of the programs during benchmarking.

As observed in Figures 5.3 and 5.5, by the final generation, all of the model configurations manage to find a way to inline *akka-uct* and *db-shootout* just as much as in the first generation, but this time most of them do not cause the compilation and execution of *dotty*

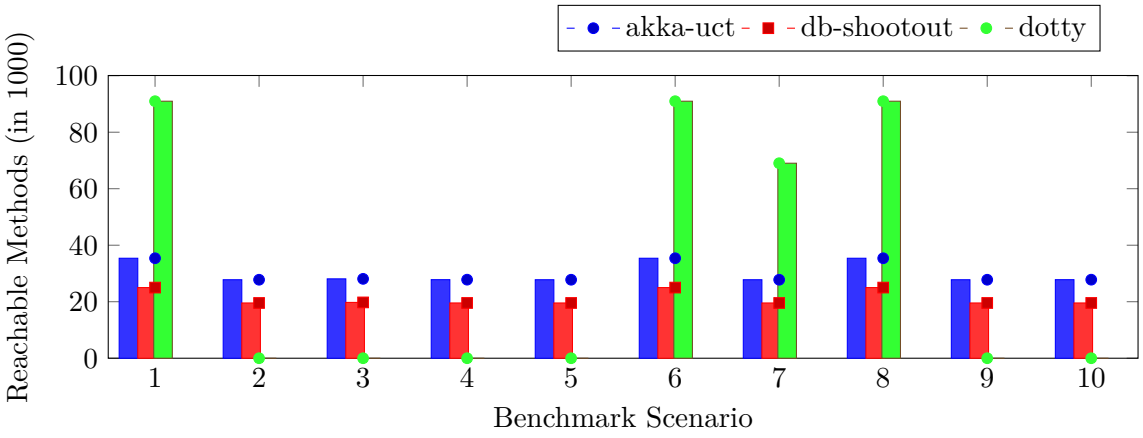


Figure 5.2: Reachable methods of the first generation non-graph models.

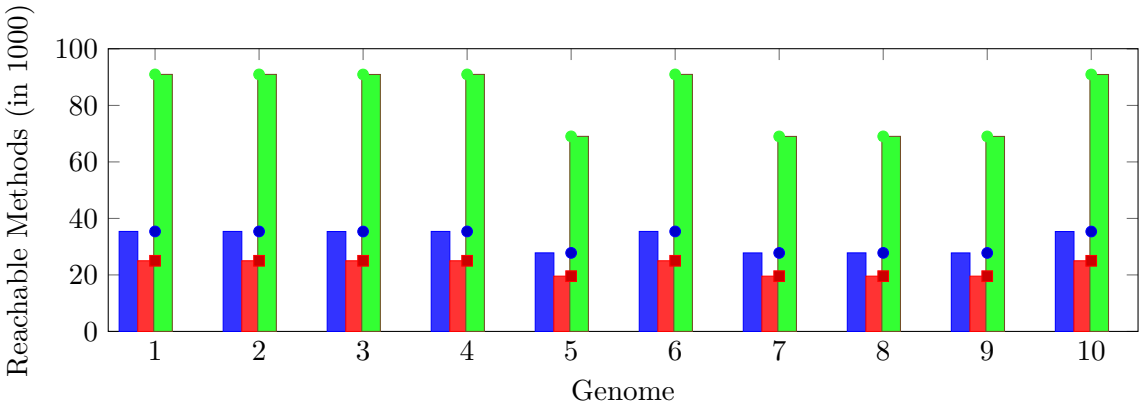


Figure 5.3: Reachable methods of the fifth generation non-graph models.

to crash. This happens without us explicitly fine-tuning the graph node limit to work for *dotty*. Not only does *dotty* not crash, but the final generation of models manages to decrease the number of dotty's reachable methods by a similar margin (in terms of percent of the original reachable methods) as it does for *akka-uct* and *db-shootout*.

| Solution | # of reachable methods | % of reference reachable methods |
|---|---|---|
| Reference | 35,386 | 100% |
| Best model | 27,780 | 78.5% |

Table 5.4: Comparison of reachable methods for baseline and the best non-graph model for *akka-uct*.

The resulting generation offers four different network configurations that produce a roughly similar number of reachable methods across the three benchmark scenarios. We could be tempted to pick a single model that keeps the lowest absolute number of total methods. However, there are more metrics on which to base our evaluation than just the fitness function used for training.

### 5.5.2 Binary Size

For the binary size metric, we can observe the same phenomena with missing metrics in crashed dotty benchmarks and the subsequent recovery of success rate in the final population as in Section 5.5.1 for reachable methods. As for the relationship between the decreasing number of reachable methods and the increasing size of the resulting binary file, we observe a relationship that in most cases seems negatively correlated. However, it turns out that the extent of increase in binary size is not comparable between different models that achieve similar counts of reachable methods, and in some cases, less reachable methods do not translate to higher binary size compared with instances with more reachable methods.

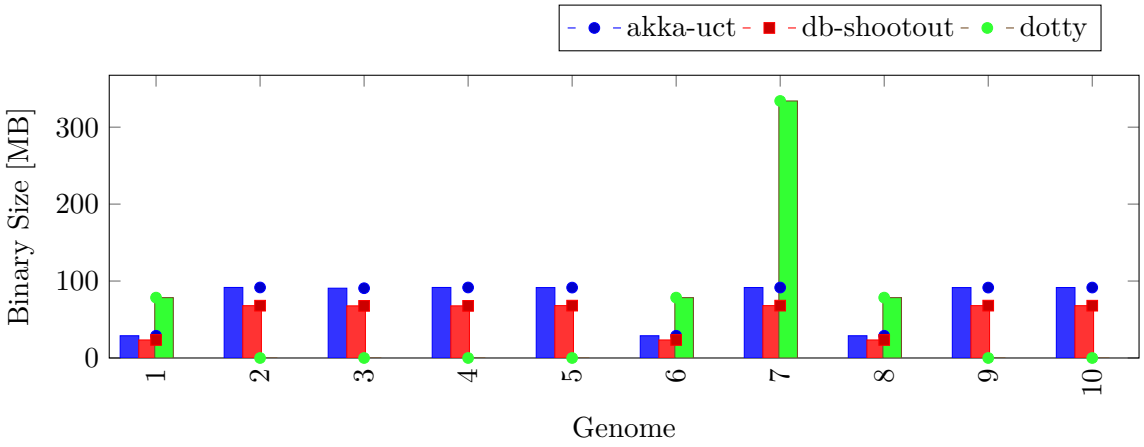


Figure 5.4: Binary size of the first generation non-graph models.

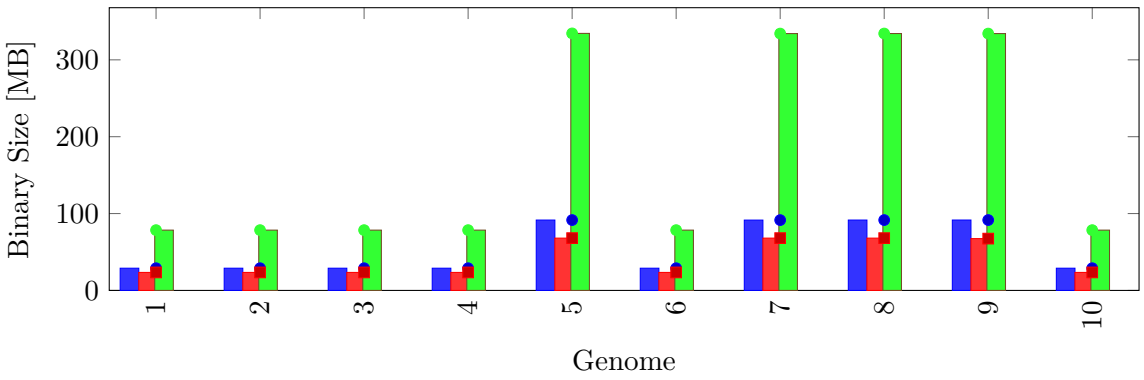


Figure 5.5: Binary size of the fifth generation non-graph models.

This discrepancy is due to our inliners not choosing to inline all invocations of the same function. As a result, functions tend to be partially inlined for some invokes, while staying not inlined in other cases, thus generating larger binaries (due to code duplication) without reducing the count of reachable methods in an IR. In terms of numbers, we have observed an increase in the binary size of the resulting executable in the range of **2 to 3 times** the original for models with the best fitness.

### 5.5.3 Validation

Once the training has been completed, we attempt to confirm the performance of the best models compared to the baseline in scenarios that were not used for training. For this purpose, we have handpicked three other programs found in the Rennaissance benchmark suite: `philosophers`, `reactors` and `scrabble`.

To perform validation on these benchmarks, we pick the four best-performing network configurations from the training phase and individually deploy them to a port on the localhost using our `infere.py` script, which parses the network parameters from the configuration file, creates an instance of a network with provided parameters and deploys it to port 8001. The neural network configurations are located in the `configs` folder in a JSON format: `nograph1.json`, `nograph2.json`, `nograph3.json` and `nograph4.json`.

Once one of the networks is deployed (Listing 5), we open a new terminal session and run `mx benchmark` with the appropriate modifiers (Listing 6) to generate a benchmark report in `bench-results.json` under the current working directory. If you run each benchmark in the same working directory, mx will overwrite the file with each iteration, so a copy of the file needs to be saved after each benchmark.

```
python infere.py configs/nograph1.json
```

Listing 5: **Deploy a pretrained neural network to a local port.** The script accepts a mandatory argument with the path to the file with biases and weights of the network.

```
mx --env ni-ce benchmark "renaissance-native-image:philosophers"
mx --env ni-ce benchmark "renaissance-native-image:reactors"
mx --env ni-ce benchmark "renaissance-native-image:scrabble"
```

Listing 6: Run each validation benchmark separately.

| Solution | Reach. methods ↓ | Binary size [MB] ↓ | Max-RSS [MB] ↓ | Execution [s] ↓ |
|---|---|---|---|---|
| Reference | 28,329 | 23.880 | 264 | 3,797.749 |
| Model #1 | 22,287 | 74.897 | 249 | 3,833.973 |
| Model #2 | 22,267 | 74.992 | 260 | 3,941.562 |
| Model #3 | 22,267 | 74.986 | 257 | 4,012.233 |
| Model #4 | 22,266 | 74.608 | 267 | 3,969.633 |

Table 5.5: Metrics for reference and non-graph models for benchmark *philosophers*.

| Solution | Reach. methods ↓ | Binary size [MB] ↓ | Max-RSS [MB] ↓ | Execution [s] ↓ |
|---|---|---|---|---|
| Reference | 28,675 | 24.108 | 795 | 22,740.325 |
| Model #1 | 22,616 | 73.337 | 1,218 | 19,240.166 |
| Model #2 | 22,613 | 73.338 | 1,242 | 18,302.744 |
| Model #3 | 22,614 | 73.480 | 805 | 18,135.389 |
| Model #4 | 22,613 | 73.485 | 815 | 17,106.370 |

Table 5.6: Metrics for reference and non-graph models for benchmark *reactors*.

| Solution | Reach. methods ↓ | Binary size [MB] ↓ | Max-RSS [MB] ↓ | Execution [s] ↓ |
|---|---|---|---|---|
| Reference | 25,494 | 31.058 | 755 | 551.990 |
| Model #1 | 20,057 | 76.747 | 709 | 516.158 |
| Model #2 | 20,054 | 77.068 | 729 | 459.514 |
| Model #3 | 20,054 | 77.302 | 741 | 494.129 |
| Model #4 | 20,053 | 77.522 | 719 | 516.095 |

Table 5.7: Metrics for reference and non-graph models for benchmark *scrabble*.

As we can see in Tables 5.5, 5.6, and 5.7, the decrease in reachable methods is in a similar percentual range as in the case of benchmarks used for training. The number of reachable methods has been reduced by 21.4% (philosphers), 21.2% (reactors) and 21.4% (scrabble) for each of the four variants of the non-graph model. The binary size also confirms the trends from training, with the generated executable increasing in size by 214.0% (philosophers), 204.7% (reactors) and 148.8% (scrabble).

All four models performed in a very similar way in both reachable methods and binary size, which is probably related to the fact they all rely on the same safety guardrails implemented by us in the AOT compiler that make sure inlining is cut off when the IR graph exceeds a threshold in node count or we are attempting to inline an uninlinable function. The safety guardrails were explained more in depth in Section 5.2.

Based on the very similar reachable methods and binary size metrics for all four networks, we assume that there is no model among them, which is distinctively better in general than the other three models. To validate this idea, we can look at the maximum resident set size metric (max-rss), which gives better results for a different model configuration in each benchmark scenario. We will move on to a neural network design that can capture more detail about the internal representation of the program.

# Chapter 6

# Inlining Based on Graph Neural Networks

For a long time, neural networks have been used with traditional architectures to solve issues that can be represented as Euclidean data, i.e., vectors of numbers. Such use cases can be found in text, video and image domains. Neither of them requires an accurate representation of complex relations between different pieces of data.

However, with the spread of machine learning use for almost any kind of problem where datasets can be collected, we have arrived at a cross-road where old architectures are no longer well suited for the problem. An example of such an issue is inline predictions. Although our feedforward solution offers a way of iterative improvement over legacy solutions, it does not answer the fundamental issue of reflecting code interactions surrounding the invocation of a method. A simple set of conditions that checks for individual attributes with hardcoded thresholds is replaced by a neural network that combines these local attributes using an evolutionary algorithm. The next step is to parse such attributes for all surrounding code blocks around our invoke method and create a sense of a wider context on which to base our decision. These blocks are all related to each other in some way based on a parent-child relationship and what order they are executed in. We could use a text-based Large Language Model (LLM) to tokenize the entire code and optimize it; however, LLMs by design are not yet suitable for this, despite showing promising signs in research. As Cummins et al. [3] noted, the main obstacles to solve are the sequence length and the lack of arithmetic reasoning.[1]

Instead, we want to use the code already parsed in the Graal compiler in the form of a graph-structured intermediate representation and transform it using neural networks that can load graph input. In this way, we will be able to preserve the relations between the methods and their calls without having to encode the state of the entire graph into a complicated vector of numbers.

We get to load data as a set of nodes (plus their features) and directional edges. This set of information gets passed into a graph convolution layer that outputs one-dimensional numerical values that can be further processed by traditional layers, like in any other simple feedforward neural network.

[1] During the writing of this thesis, new models such as OpenAI o3 came out. They have demonstrated promising mathematical reasoning abilities that may unblock this area of experiments.

## 6.1 Graph Neural Networks

Neural networks used to struggle with data in the graph domain. It was challenging to represent certain types of data, such as genome structures or citation relationships between different research documents. Graph neural networks were designed to mitigate these issues [22]. The core concept is based on the principle of message passing between interconnected neighboring nodes until information spreads everywhere through neighbours and their neighbours, as visualized in Figure 6.1. Message passing is, for example, used in belief propagation for Bayesian networks [14] to pass probability distributions across the directed tree structure. Graph networks take this idea one step further, and instead of passing conditional probabilities (beliefs), they pass the features of individual nodes. That is how neighbouring nodes learn about the features of their own neighbours.

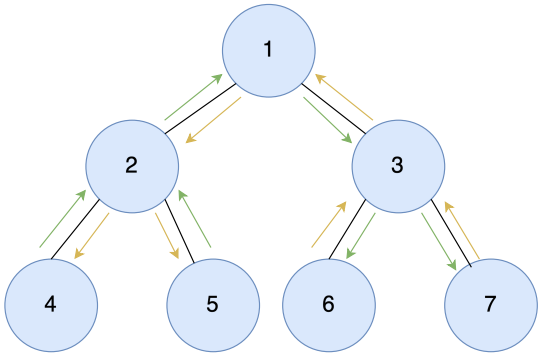


Figure 6.1: **Message passing in action.** In this example information propagates between nodes 4 & 5 to 6 & 7 via 1, 2 and 3.

There are multiple types of graph neural networks, each type being predominantly used for a different category of inference tasks. Graph sample aggregation (GraphSAGE) [7] is often used to predict links between nodes in an existing graph; graph attention networks (GATConv) [22] are used to make predictions on top of the entire graph level. For us, the most relevant type is a **graph convolutional network** (GCN), which is used in networks to predict at the node level. The principle of a multi-layered GCN network is built upon the following equation:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

There is a lot going on in this formula, so let us break it down. The activation function $\sigma$ - usually ReLU ($f(x) = max(0, x)$) - processes the result of multiplying matrices $W^{(l)}$ (trainable weight matrix in the $l$-th layer), $\tilde{A}$ (adjacency matrix of the graph), $H^{(l)}$ (matrix of node features in the $l$-th layer) and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ (diagonal degree matrix of $\tilde{A}$)) [8]. The degree matrix $\tilde{D}$ is used to normalize each row in matrix $A$, so that when $A$ multiplies $H$, it does not change the scale of the feature vectors.

Fortunately, we do not have to implement the formula directly in code, as it is already available in **PyG** (PyTorch Geometric) library. To get started, it is important to understand that the output of a GCN layer is the result of the interaction between nodes, their attributes and the directed edges that connect the nodes with each other to form a relationship between them, as shown in Figure 6.2.

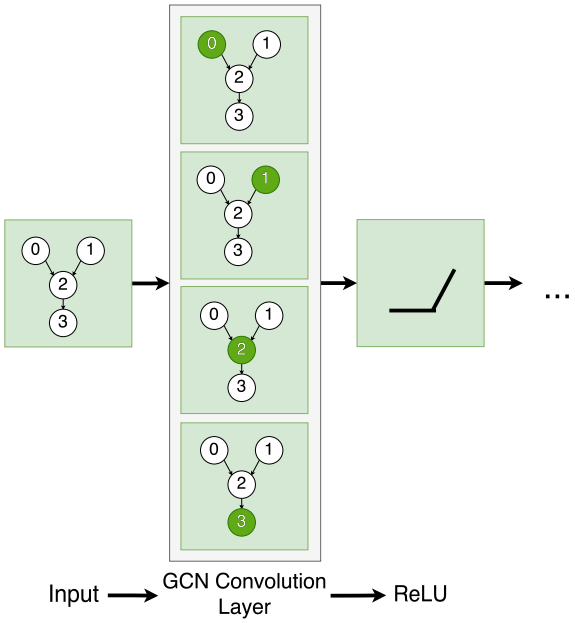


Figure 6.2: **Input graph is processed by a hidden GCN layer.** Output of the GCN convolution layer is a matrix $N \times F$, where $N$ is the number of nodes (4) in the graph and $F$ the number of output features for each node (1). The output features are fed into a ReLU activation function.

## 6.2 Graph as Input

Before building inlining pipelines with graph-based models, we look at how we can transform the data we have in Graal into a format that can be fed as input into models with GCN layers. PyG defines a class *torch_geometric.data.Data* that stores $x$ (a matrix with attributes for each node) and *edge_index* (a transposed matrix with directional node-to-node relationships).

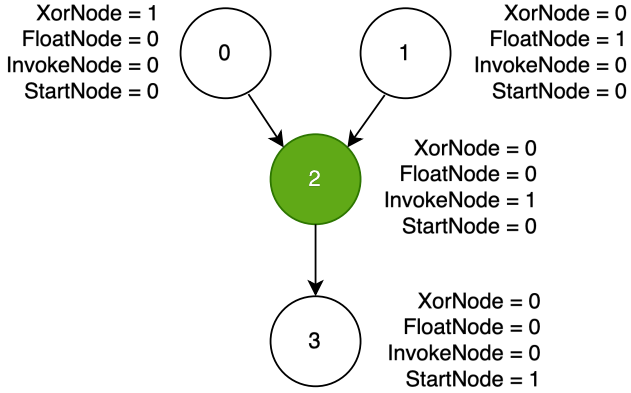


Figure 6.3: **An example graph.** The attributes of each node are a list of possible node types, with 1 meaning belonging of the node to the type and 0 not belonging.

Inside Graal, we make a POST request for the current inlining decision. The body of this request consists of a list of nodes with their node type and a list of directional edges,

each edge being represented by a tuple of 2 zero-based indexes referring to the position of the node in the *nodes* array. An example of such a request can be seen in Figure 6.3 and Listing 7.

```
{
    "nodes": [
        {"nodeType": 0},
        {"nodeType": 1},
        {"nodeType": 2},
        {"nodeType": 3}
    ],
    "edges": [
        [0, 2],
        [1, 2],
        [2, 3]
    ]
}
```

Listing 7: An example REST request made during compilation from Graal.

The node type is an enumerated datatype identified by a number, with each number representing an index in the list of nodes we recognize. For a complete list of recognized node types, refer to the Appendix A.

The root node of each graph is the invocation of the method we are trying to inline. Invocation has an IR graph (IR of the method where invocation resides), and the invoked method has an IR of its own. By collecting the surrounding of the invocation, we understand collecting the immediate limited surrounding of both the invocation and the invoked method, and merging the two IRs into one.

The challenge is to identify duplicate nodes introduced by merging of the graphs; in our case, these are the parameters of the invoked method and the returned value. We identify a connection between nodes in the parent and child graph for both the parameters and the returned value. After removal of duplicate nodes, we adjust the indexing of the nodes in *edges*.

Once the POST request is received, we transform the content into a Torch tensor compatible format, shown in Listing 8. As `nodeType` represents a categorical value, we transform it using one-hot encoding into the final input for the neural network. One-hot encoding transforms a non-negative number into a vector of zeros with a single number 1 among them. For the number $n$, the number 1 would be placed at index $n$ in the vector. The length of the vector is as long as the highest value of the categories we plan on encoding. For example, if we have four categories in total, i.e., $[0, 1, 2, 3]$, and we want to encode the second category, i.e., 1, the one-hot encoded equivalent of that is $[0, 1, 0, 0]$.

This is only done on the model's backend, since there is no reason to inflate the REST request's JSON body by including an array of numbers instead of a single *nodeType* value for each node found in the request. Each such request contains tens of nodes each, so the size of requests does add up with no optimizations.

```
import torch
from torch_geometric.data import Data

# each row represents a directional node-to-node edge
edge_index = torch.tensor([[0, 2],
                           [1, 2],
                           [2, 3]], dtype=torch.long)

# each row represents all attributes of a single cell
x = torch.tensor([[1,0,0,0],
                  [0,1,0,0],
                  [0,0,1,0],
                  [0,0,0,1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index.t().contiguous())
```

Listing 8: Request is transformed into its equivalent in the form of a Torch tensor.

## 6.3 Network Architecture

The NEAT library used up to now did not support graph algorithms. Therefore, we have built our own network that fits its existing internals by replacing the original feedforward model used in our prototype and reverse engineering its interface to fit our custom model. A part of this backporting effort focuses on supporting the generation of parameters done by NEAT, and consequent loading of these parameters as weights and biases into our model. Another important aspect is the ability to have a custom number of inputs, all configurable via the generic configuration file used for other NEAT models. This allows us to experiment with different mappings, each variant supporting a different number of node types. As it turns out, there are hundreds of different nodes, and processing input arrays of such length has a negative impact on performance.

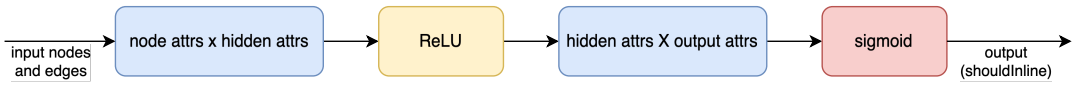


Figure 6.4: **The architecture of a classification graph network.** The number of node attributes, hidden attributes and output channels come from the NEAT configuration file.

The final proposed graph model consists of three types of elements: **GCNConv**, **ReLU** and **Sigmoid**. GCNConv stands for a graph convolution layer. We are using two of them, each with its own softmax layer, bringing the total number of layers in our model to four. As you can see in Figure 6.4 and Listing 9, we use the NEAT configuration file to provide us with the number of node attributes, hidden attributes and output channels for each layer.

The node attributes serve as input to our neural network. In fact, there is only one attribute for each node, its type. It is, however, one-hot encoded into an array of ones and zeros. The array's length equals the number of recognized node types in our model, plus one that serves as a bin for all unmapped and unrecognized node types.

Hidden attributes are an arbitrary number. The higher the number of hidden attributes, the more complex information can be propagated between the input layer and the output layer. A higher number of hidden nodes also means the need to generate more random, yet optimal parameters with NEAT. Due to the one-hot encoding done on the input layer, the number of parameters utilized for that layer is already greater than what we used in our prototype, so it is desirable to keep the number of hidden channels as small as possible. For hidden layer activations, we use ReLU as it is the go-to default function in hidden layers for most neural networks. When it comes to the output layer, we are again classifying into one of two categories, so we are keeping the same sigmoid final activation that we have used previously.

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv


class GCN(torch.nn.Module):
    def __init__(self, inputs, hidden, outputs):
        super().__init__()
        self.conv1 = GCNConv(inputs, hidden)
        self.conv2 = GCNConv(hidden, outputs)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        y = F.softmax(x, dim=1)
        return y
```

Listing 9: Architecture of the graph neural network in PyTorch.

## 6.4 Training

The base NEAT library does not use a deep learning framework such as Tensorflow or PyTorch, instead it implements its linear regression operations directly in its source code. We build a custom wrapper that bridges one library with another. The workflow goes as follows:

1. NEAT generates a genome.
2. A custom wrapper creates a neural network in PyTorch and sets it up using:
   (a) Weights and biases located in the genome generated in step 1.
   (b) Parameters found in NEAT's configuration file (number of input, hidden, and output features).

3. The wrapper containing the neural network is deployed to an API endpoint.

4. During benchmarking, each inference call is passed to the wrapper first, which transforms the input received inside the REST payload into Torch compatible data structures before calling the actual model (Listing 10). The result of the call is then transformed back into the format NEAT expects.

Transformation of the input JSON into arrays of node attributes and edges is performed in `api.py`, which passes these two arrays into the Torch wrapper. There, the Python arrays are transformed into Tensor arrays, which can be loaded into the GPU memory and inference can be computed on a graphics card that offers more memory and parallelism capabilities than an ordinary computer processor, and thus might calculate the results for each inlining faster. After some testing, it has been observed that running our network on the CPU is actually faster. This is down to more factors. The GPU we are training on is an NVIDIA GTX 1650, which does not have enough performance to be considered a serious machine learning grade hardware. The second factor is that traditional machine learning workflows take advantage of GPU parallelism by increasing the batch size. In simple terms, they execute the same kind of operation on more data in parallel, which is much faster than performing operations on a single data entry at a time, one by one. The problem with batch size parallelism is that Native-Image's AOT compiler is already running in multiple threads, so we would collect individual inference requests into a queue on the REST API side and make the requesting threads on Graal's side wait for a response. Once enough REST requests are in a queue, we can execute the requests in a single batch in Torch and then redistribute the results for each compiler thread. For the scope of this thesis, we are not going to explore this architecture, since it does not constitute a minimal viable product that this thesis aims for.

There are various workarounds needed to make the custom solution work due to inconsistencies in the version of the NEAT library used. For example, I had to turn the fitness minimization task into the maximization of negative fitness because the NEAT library did not adjust its behavior when the appropriate parameter in its configuration was changed. It always maximized fitness, regardless of the parameter in its configuration.

Another example was that the library did not generate a consistent number of connections between its genomes at all times. That introduced an error when setting up our PyTorch model, as it requires a fixed-length list of parameters for all the weights and biases our model possesses. Generating fewer connections is generally not a bad idea, as it allows us to dilute a neural network. However, in our case, sometimes there would not only be fewer connections than we predefined, but there would also sometimes be more of them. Due to our model architecture, we already have as many connections between nodes as it is possible for a feedforward network to have, so for the sake of consistency, I have decided to use a fixed number of connections no matter how many connections are in the generated genome. There is a parameter in the configuration file that modifies the probability of adding and dropping connections, which can effectively forbid such mutations to our genomes, should we set the probability to zero. A similar parameter exists for adding and removing nodes in our network, it was also set to disallow any changes to the topology. The number of inputs in the configuration file corresponds to the number of attributes of a node, which in our case is the length of the one-hot encoded type of the node, i.e., 178. The number of hidden features was set to 16, and the output features stay the same as in the non-graph model configuration, i.e., 1. With the increase of nodes and edges, the total number of parameters also increases, and with that, the changes caused by mutations to

```python
class GraphNetwork(object):
  def __init__(self, genome_config):
    self.device = torch.device('cpu') # my GPU was slower than my CPU
    self.model = GCN(genome_config).to(self.device)

  def activate(self, nodes: np.ndarray, edges: np.ndarray):
    data = Data(
      x=torch.tensor(nodes, dtype=torch.float),
      edge_index=torch.tensor(edges, dtype=torch.int16)
    ).to(self.device)

    with torch.no_grad():
      out = self.model(data)
    return out.cpu().numpy()[0].tolist() # node at idx 0 = invocation

  def set_parameters(self, genome_config, genome):
    self.model.conv1.lin.weight = torch.nn.Parameter(<weights>)
    self.model.conv1.bias = torch.nn.Parameter(<biases>)
    self.model.conv2.lin.weight = torch.nn.Parameter(<weights>)
    self.model.conv2.bias = torch.nn.Parameter(<biases>)

  @staticmethod
  def create(genome, config):
    network = GraphNetwork(config.genome_config)
    network.set_parameters(config.genome_config, genome)
    return network
```

Listing 10: **Compatibility layer between PyTorch and NEAT.** The genome represents weights and biases that are used to set up the Torch network.

the individual weights and biases add up to a bigger total difference between genomes in the population. Although the original compatibility threshold for the non-graph configuration was high enough for all genomes in the same population to belong to the same species, that is no longer the case for graph genomes. Each genome has its own pieces assigned, making it 10 species per generation. Since elitism (the preservation of best fitted genomes of the previous generation) is applied per species, either zero genomes can be preserved into the next generation, or all ten of them are preserved. We opt to increase `compatibility_threshold` to a very high number, which brings all genomes back into the same single species. For the rest of the configuration, refer to the file `config-gnn` from Appendix D.

## 6.5 Evaluation of the Graph Model

### 6.5.1 Reachable Methods and Binary Size

With our training pipeline already proven from training the non-graph models, our main area of focus is not whether the training progresses in the direction of models that inline

more. Our focus lies on finding trends in the training - both similarities compared to the training of non-graph models, but also the differences. In the case of graph models, we have a much higher number of parameters compared to non-graph models - it is 2881 parameters for graph models, versus only 19 for non-graph models.

The difference is explained in the formula used to calculate the number of parameters of a GCN network: $Params = F_i \times F_h + F_h + F_h \times F_o + F_o$, where $F_i$ stands for input features (178), $F_h$ for hidden features (16) and $F_o$ for output features (1). The first GCN layer has $F_i \times F_h$ number of weight parameters, plus $F_h$ bias parameters. The second GCN layer has $F_h \times F_o$ number of weight parameters, plus $F_o$ bias parameters.

Although the feedforward non-graph models adhere to the same formula, in their case $F_i$ stands for 4 input numbers, $F_h$ for 3 hidden features, and $F_o$ for 1 output feature.

As it becomes apparent, the significant increase in parameters count for graph models originates in our input layer, which consumes 171 numbers instead of 4. While for the feedforward network we are consuming four continuous numerical attributes, for graphs we are using a categorical attribute (type of a node), which can not be encoded in a single number the same way as continuous data. Doing that would assume that categorical data are ordered and that `IfNode` can be, e.g., more than `StartNode` and less than `InvokeNode`. It also assumes a sense of transitive relationship between categories, where `StartNode` < `IfNode` and `IfNode` < `InvokeNode` implies `InvokeNode` being more than `StartNode` on a scale greater than `IfNode` is greater than `StartNode`. None of these assumptions are correct; node types are not ordered, and they are only labels - there is no sense of scale when comparing them. Continuous variables satisfy both of these requirements, e.g., a 180 cm tall person is taller than a 170 cm tall person, but not as much taller as a 190 cm person would be. Categorical data are not like that.

The higher number of parameters paradoxically leads to less variability in fitness between genomes in the same generation, with the initial population starting out on a mostly conservative spectrum of inlining decisions. The genomes that do inline, always run out of memory on the heap during compilation of *dotty*. As generations start adding up, the genomes eventually start inlining and they arrive at metrics similar to those observed when training non-graph models.

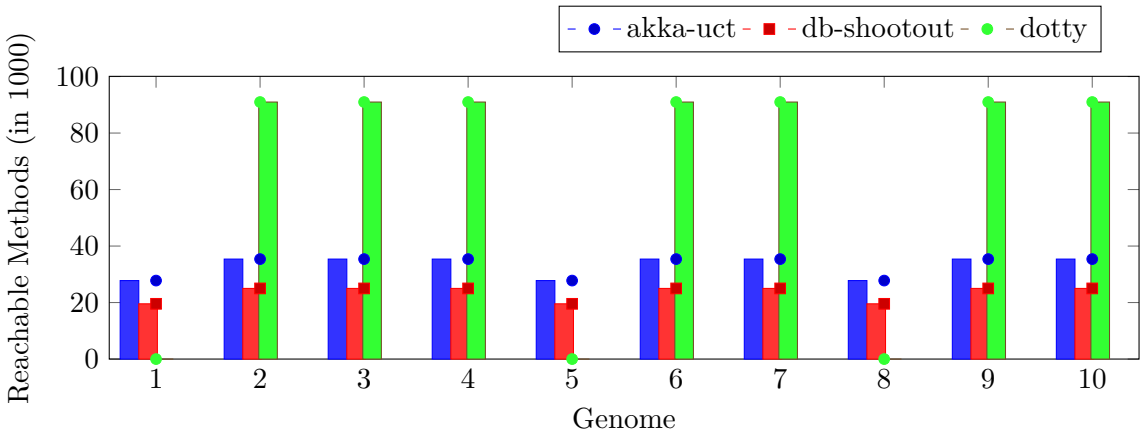


Figure 6.5: Reachable methods of the first generation graph models.

As presented in Figures 6.5, 6.6, 6.7 and 6.8, similarly to non-graph models, we observe a slight decrease in reachable methods and an increase in binary size for the best models. The apparent indifference between non-graph and graph models makes it obvious that despite

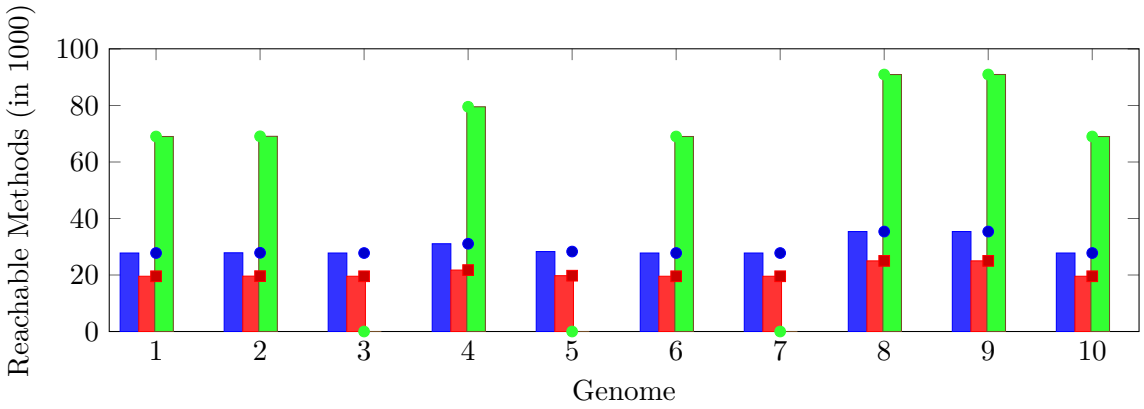


Figure 6.6: Reachable methods of the fifth generation graph models.

graph models having more structural information at their disposal, they are trained to perform similarly to the feedforward network with the existing setup and fitness function.

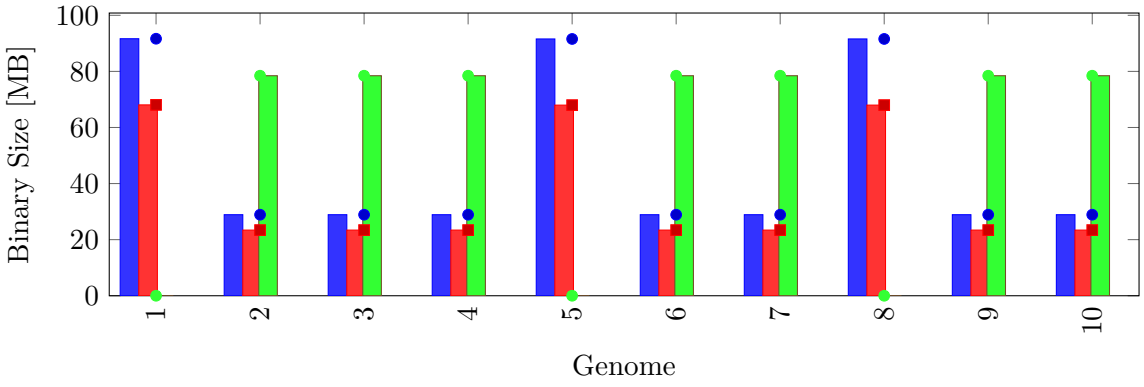


Figure 6.7: Binary size of the first generation graph models.

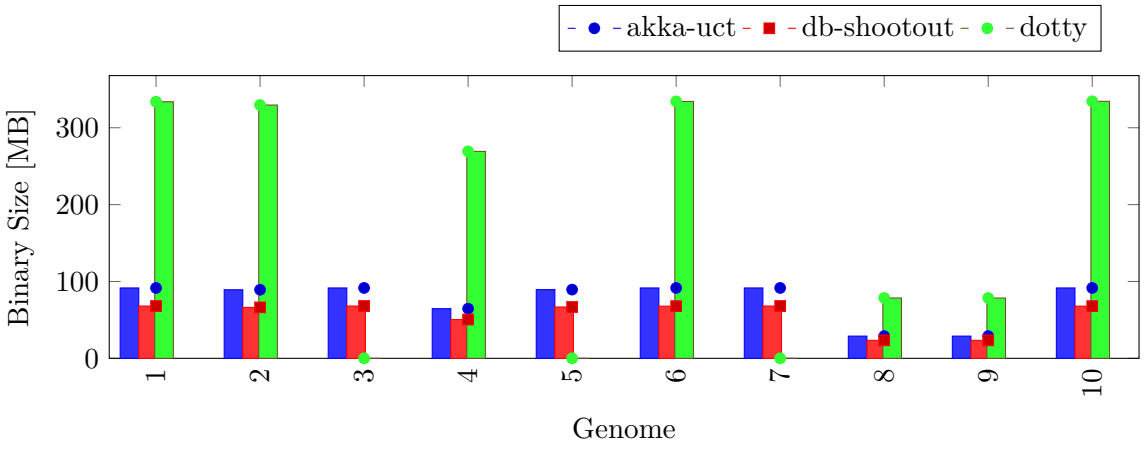


Figure 6.8: Binary size of the fifth generation graph models.

### 6.5.2 Validation

In this section, we compare the four best network configurations against the reference metrics. We also compare the selected configurations against one of the non-graph models in Section 5.5.3. As all non-graph models performed similarly to each other, we arbitrarily selected Model #1 to represent the entire non-graph group.

| Solution | Reach. methods↓ | Binary size [MB]↓ | Max-RSS [MB]↓ | Execution [s]↓ |
|---|---|---|---|---|
| Reference | 28,329 | 23.880 | 264 | 3,797.749 |
| Non-graph | 22,287 | 74.897 | 249 | 3,833.973 |
| Graph #1 | 22,266 | 74.836 | 247 | 3,989.168 |
| Graph #2 | 22,266 | 74.912 | 273 | 4,015.206 |
| Graph #3 | 22,266 | 74.913 | 257 | 4,003.657 |
| Graph #4 | 22,266 | 74.914 | 265 | 4,023.690 |

Table 6.1: Metrics for reference, non-graph and graph models for benchmark *philosophers*.

| Solution | Reach. methods↓ | Binary size [MB]↓ | Max-RSS [MB]↓ | Execution [s]↓ |
|---|---|---|---|---|
| Reference | 28,675 | 24.108 | 795 | 22,740.325 |
| Non-graph | 22,616 | 73.337 | 1,218 | 19,240.166 |
| Graph #1 | 22,613 | 73.409 | 890 | 16,968.259 |
| Graph #2 | 22,613 | 73.333 | 816 | 18,373.149 |
| Graph #3 | 22,613 | 73.332 | 857 | 17,224.159 |
| Graph #4 | 22,614 | 73.111 | 858 | 18,341.642 |

Table 6.2: Metrics for reference, non-graph and graph models for benchmark *reactors*.

| Solution | Reach. methods↓ | Binary size [MB]↓ | Max-RSS [MB]↓ | Execution [s]↓ |
|---|---|---|---|---|
| Reference | 25,494 | 31.058 | 755 | 551.990 |
| Non-graph | 20,057 | 76.747 | 709 | 516.158 |
| Graph #1 | 20,053 | 77.370 | 809 | 507.561 |
| Graph #2 | 20,053 | 77.530 | 770 | 563.633 |
| Graph #3 | 20,053 | 77.453 | 754 | 502.499 |
| Graph #4 | 20,054 | 76.907 | 726 | 501.725 |

Table 6.3: Metrics for reference, non-graph and graph models for benchmark *scrabble*.

Again, in Figures 6.1, 6.2 and 6.3 we observe results that are very similar to those in Section 5.5.3. It is unavoidable to talk about how models so different in parameters, and in the case of non-graph and graph models, also different in architecture, yield so similar results. For all validation cases, every model tested has made only a positive decision about inlining, which means that no inlining was refused by any of the models tested. This leads to the conclusion that although all of the best models from both non-graph and graph categories possess attributes that make them refuse to inline in specific cases where other models would inline too aggressively and cause benchmarks to fail (e.g., *dotty* benchmark), in other cases they might inline everything they can and they are only stopped by the guardrails we have implemented in Section 5.2. During training, the guardrails had the important task of keeping the training on track and avoiding models from losing their

way while exploring too aggressive inlinings that would make the execution of the compiled machine code fail. Unfortunately, this also makes aggressive inliners safe, as they can try to inline everything - they will eventually be stopped by the automatic guardrails before they can cause any harm. Such models do not need to develop a complex understanding of harmful inlinings, they only need to avoid inlining in cases where the hardcoded guardrails are not activated. Those cases are only a subset of the problems that the inliner should recognize and solve in a real world, otherwise we are back to inlining mostly based on hardcoded thresholds in code.

The issue of inliners' reliance on guardrails originates in the inliner's objective to reduce the number of reachable methods. A more complex fitness function that incorporates the size of the binary and the time it takes to execute the compiled binary file could penalize the models that inflate the executable size irresponsibly. Keeping the size low means the inliner will encounter guardrail limits less often (the guardrails cut off inlining when the IR graph exceeds a fixed number of nodes). By doing this, we shift the responsibility for handling excessive inlining to the fitness function and therefore to the process of training the models, rather than silently guarding the actions of models that are greedy with inlining. However, incorporating all metrics requires a formula to compute a single number that serves as the fitness function. This task requires non-trivial research on top of the existing infrastructure we already built and used to train our models, and it is out of the scope of this work.

# Chapter 7

# Conclusion

The goal of the thesis was to investigate the capabilities of GraalVM, its Native Image component, and to design and implement a machine-learning-based heuristic in the environment of GraalVM Native Image. The chosen area of focus became inlining during the construction of the internal representation (IR) graph of the AOT compiler, even before the static analysis is executed. In the original Native Image implementation, the heuristics are set up using constant thresholds fine-tuned according to predefined benchmarks. The aim was to research an improved heuristic using machine learning and neural networks.

The first step was to explore and collect training data. We decided not to train a classic model using annotated training data, as it is not possible to determine which individual inlining is good or not. We decided to implement a pipeline with genetic algorithms, which generates neural network parameters that are benchmarked against each other on a set of training programs. To verify the functionality of the pipeline, a simple feedforward neural network was designed and evaluated. This network is later used as a comparison for the more complex graph neural network. However, the graph neural network was incompatible with the implementation of the base NEAT library, which is why we implemented a wrapper middleman to tie together NEAT library and Torch.

During training, the first generations of models would inline either too aggressively and cause the program to crash during benchmarks, or inline too little and not produce much different executable images compared with the conservative baseline inliner. As training progressed, the models improved inlining, and the generated native images stopped crashing at runtime, but it came at a cost. Although we could reduce the number of reachable methods by around 20-25%, the size of the binary would increase two to three times. The best networks from the training stage of both non-graph and graph models were then deployed locally and validated on previously unseen programs. There, the models confirmed the decrease in reachable methods, but also the significant increase in binary size. In most of the benchmarks, both for training and validation scenarios, we observed a modest increase in execution time of the generated native binary. There were also scenarios where these models increased the execution time by bigger margins and scenarios where they decreased the time compared with the existing baseline.

As a consequence, it is not possible to determine a statistically positive impact of the neural networks presented in this thesis. During the writing of this thesis, we have encountered many options, where we could have taken a different approach that could have produced a different result. Some of these options include modifications to the API endpoint to execute model inference in batches, which may increase the speed of compilation, which in turn allows to train more generations of networks. This would have helped con-

sidering our training sessions were limited by the population and species size, which was a trade-off made due to hardware constraints. A related performance improvement is to embed the model directly into the compiler's source code and remove the REST overhead that way. Another option is to experiment with models that have multi-layered graph convolutions instead of the two-layered convolution used in our approach. Finally, we found that the combination of our simple fitness function, the restricted number of training iterations, and the hardcoded security guardrails between the model and the compiler, constrain the ability of our models to learn more generic patterns about harmful decisions, because they learn to inline aggressively and rely on the hardcoded guardrails to stop them.

# Bibliography


[1] Christian Wimmer, C. S. *Safe and Efficient Hybrid Memory Management for Java*. 2015 [cit. 2025-05-18]. Available at: oracle.com/technetwork/java/jvmls2015-wimmer-2637907.pdf.

[2] Cooper, K., Harvey, T. and Waterman, T. An Adaptive Strategy for Inline Substitution. In:. April 2008, vol. 4959, p. 69–84. DOI: 10.1007/978-3-540-78791-4_5. ISBN 978-3-540-78790-7.

[3] Cummins, C., Seeker, V., Grubisic, D., Elhoushi, M., Liang, Y. et al. Large Language Models for Compiler Optimization. *ArXiv*. 2023, abs/2309.07062. Available at: https://api.semanticscholar.org/CorpusID:261705851.

[4] Gilles Duboscq, T. W. D. S. C. W. H. M. Graal IR : An Extensible Declarative Intermediate Representation. In:. 2013. Available at: https://api.semanticscholar.org/CorpusID:52231504.

[5] Glek, T. Optimizing real world applications with GCC Link Time Optimization. *ArXiv*. 2010, abs/1010.2196. Available at: https://api.semanticscholar.org/CorpusID:225421.

[6] Gomede, E. *NeuroEvolution of Augmenting Topologies (NEAT): Innovating Neural Network Architecture Evolution*. 2023 [cit. 2025-05-18]. Available at: medium.com/@evertongomede/neuroevolution-of-augmenting-topologies-neat-innovating-neural-network-architecture-evolution-bc5508527252.

[7] Hamilton, W. L., Ying, R. and Leskovec, J. Inductive representation learning on large graphs. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2017, p. 1025–1035. NIPS'17. ISBN 9781510860964.

[8] Kipf, T. and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. september 2016. DOI: 10.48550/arXiv.1609.02907.

[9] KOZÁK, D., STANCU, C., WIMMER, C. and WÜRTHINGER, T. Scaling Type-Based Points-to Analysis with Saturation. *Proceedings of the ACM on Programming Languages*. 2024, vol. 8, PLDI, p. 990–1013. DOI: 10.1145/3656417. ISSN 2475-1421. Available at: https://dl.acm.org/doi/pdf/10.1145/3656417.

[10] Kulkarni, S., Cavazos, J., Wimmer, C. and Simon, D. Automatic construction of inlining heuristics using machine learning. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2013, p. 1–12. DOI: 10.1109/CGO.2013.6495004.

[11] LLVM Foundation. *Link Time Optimization: Design and Implementation*. [cit. 2025-05-18]. Available at: llvm.org/docs/LinkTimeOptimization.html.

[12] Majó, Z. *Compilation in the HotSpot VM*. 2015 [cit. 2025-05-18]. Available at: ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2015/210_Compiler_Design/Slides/hotspot.pdf.

[13] Micronaut foundation. *Micronaut*. 2023. Available at: https://micronaut.io.

[14] Nuel, G. Tutorial on Exact Belief Propagation in Bayesian Networks: from Messages to Algorithms. *ArXiv: Probability*. 2012. Available at: https://api.semanticscholar.org/CorpusID:88519010.

[15] Oracle. *Java Language and Virtual Machine Specifications*. [cit. 2025-05-18]. Available at: docs.oracle.com/javase/specs/.

[16] Oracle. *A brief history of Java*. 2019 [cit. 2025-05-18]. Available at: oracle.com/a/ocom/docs/dc/ww-brief-history-java-infographic.pdf.

[17] Oracle. *Oracle GraalVM Enterprise Edition*. 2023 [cit. 2025-05-18]. Available at: oracle.com/a/ocom/docs/graalvm-enterprise-white-paper.pdf.

[18] Sipek, M., Mihaljevic, B. and Radovan, A. Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. *CoRR*. 2021, abs/2112.14716. Available at: https://arxiv.org/abs/2112.14716.

[19] Stadler, L., Würthinger, T. and Mössenböck, H. Partial Escape Analysis and Scalar Replacement for Java. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY, USA: Association for Computing Machinery, 2018, p. 165–174. CGO '14. DOI: 10.1145/2544137.2544157. ISBN 9781450326704. Available at: https://doi.org/10.1145/2544137.2544157.

[20] Toosi, A., Bottino, A., Saboury, B., Siegel, E. and Rahmim, A. A Brief History of AI: How to Prevent Another Winter (A Critical Review). *PET Clinics*. september 2021, vol. 16. DOI: 10.1016/j.cpet.2021.07.001.

[21] Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K. et al. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *CoRR*. 2021, abs/2101.04808. Available at: https://arxiv.org/abs/2101.04808.

[22] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P. et al. Graph Attention Networks. october 2017. DOI: 10.48550/arXiv.1710.10903.

[23] Wimmer, C., Stancu, C., Hofer, P., Jovanovic, V., Wögerer, P. et al. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. oct 2019, vol. 3, OOPSLA. DOI: 10.1145/3360610.

[24] Würthinger, T., Wimmer, C., Wöss, A., Stadler, L., Duboscq, G. et al. One VM to Rule Them All. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. New York, NY, USA: Association for Computing Machinery, 2013, p. 187–204. Onward! 2013. DOI: 10.1145/2509578.2509581. ISBN 9781450324724.

# Appendix A

# Supported node types

| | |
|---|---|
| InvokeNode | IfNode |
| LogicNegationNode | LoopExitNode |
| ConstantNode | PiArrayNode |
| ValuePhiNode | IntegerTestNode |
| NewMultiArrayNode | SqrtNode |
| LoadFieldNode | FloatConvertNode |
| LoweredAtomicReadAndWriteNode | XorNode |
| UnsafeCompareAndSwapNode | NarrowNode |
| DynamicNewInstanceWithExceptionNode | IntegerBelowNode |
| TypeSwitchNode | FloatDivNode |
| ArrayLengthNode | ConditionalNode |
| UnsafeCompareAndExchangeNode | NotNode |
| AtomicReadAndAddNode | IntegerLessThanNode |
| LoadIndexedNode | ZeroExtendNode |
| NewInstanceWithExceptionNode | SignedRemNode |
| NewArrayNode | RightShiftNode |
| InstanceOfNode | RoundFloatToIntegerNode |
| StoreFieldNode | AbsNode |
| LoadExceptionObjectNode | CopySignNode |
| StoreIndexedNode | SignumNode |
| NewArrayWithExceptionNode | SignedDivNode |
| LogicCompareAndSwapNode | UnsignedRightShiftNode |
| InstanceOfDynamicNode | NegateNode |
| DynamicNewArrayNode | SubNode |
| AtomicReadAndWriteNode | SignExtendNode |
| ValueCompareAndSwapNode | ObjectEqualsNode |
| MethodCallTargetNode | AndNode |
| NewMultiArrayWithExceptionNode | FloatLessThanNode |
| ReachabilityFenceNode | FloatNormalizeCompareNode |
| DynamicNewArrayWithExceptionNode | ExpandBitsNode |
| DynamicNewInstanceNode | SignedFloatingIntegerRemNode |
| NewInstanceNode | UnsignedDivNode |
| LoweredAtomicReadAndAddNode | UnsignedMaxNode |
| ExceptionObjectNode | OrNode |
| DynamicCounterNode | FloatTypeTestNode |
| SideEffectNode | FloatEqualsNode |
| StringToBytesNode | IntegerEqualsNode |
| UnreachableNode | IsNullNode |
| DynamicPiNode | MinNode |

| | |
|---|---|
| MaxNode | UnsafeCopyNode |
| OpMaskTestNode | ObjectIsArrayNode |
| AddNode | StateSplitProxyNode |
| MulNode | OpaqueValueNode |
| IntegerNormalizeCompareNode | SpeculationFenceNode |
| PointerEqualsNode | GuardedUnsafeLoadNode |
| ReinterpretNode | RawLoadNode |
| LeftShiftNode | JavaReadNode |
| RoundNode | BranchProbabilityNode |
| SignedFloatingIntegerDivNode | GetClassNode |
| UnsignedMinNode | BytecodeExceptionNode |
| ShortCircuitOrNode | LoadMethodNode |
| MergeNode | JavaWriteNode |
| PauseNode | CaptureStateBeginNode |
| ConditionAnchorNode | UnboxNode |
| SpinWaitNode | ForeignCallWithExceptionNode |
| StartNode | MemoryMapNode |
| BeginNode | WriteNode |
| DeadEndNode | MemoryPhiNode |
| ComputeObjectAddressNode | ReadNode |
| UnwindNode | OffsetAddressNode |
| AllocatedObjectNode | IndexAddressNode |
| CommitAllocationNode | FloatingReadNode |
| VirtualBoxingNode | SideEffectFreeWriteNode |
| VirtualInstanceNode | MemoryAnchorNode |
| VirtualArrayNode | UnreachableControlSinkNode |
| EnsureVirtualizedNode | IndirectCallTargetNode |
| ValueProxyNode | DirectCallTargetNode |
| GetObjectAddressNode | LoopBeginNode |
| SwitchCaseProbabilityNode | ReturnNode |
| LoadArrayComponentHubNode | InvokeWithExceptionNode |
| IntegerSwitchNode | GuardedValueNode |
| ValueAnchorNode | ParameterNode |
| UnsafeMemoryLoadNode | FixedGuardNode |
| ClassIsArrayNode | PiNode |
| ForeignCallNode | LogicConstantNode |
| NullCheckNode | EndNode |
| OpaqueLogicNode | LoopEndNode |
| LoadHubOrNullNode | FrameState |
| UnsafeMemoryStoreNode | |

# Appendix B

# Evolution of training for the non-graph model

### B.0.1 Reachable Methods

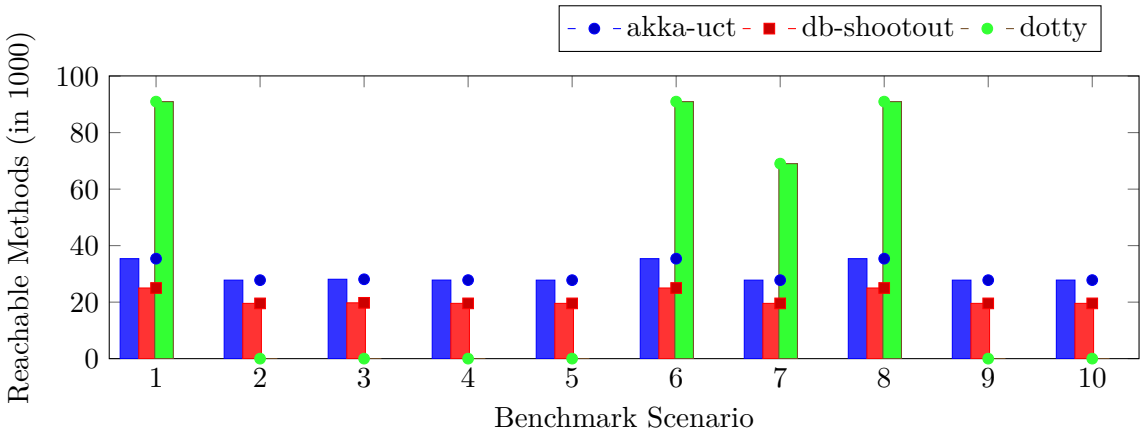


Figure B.1: 1st generation

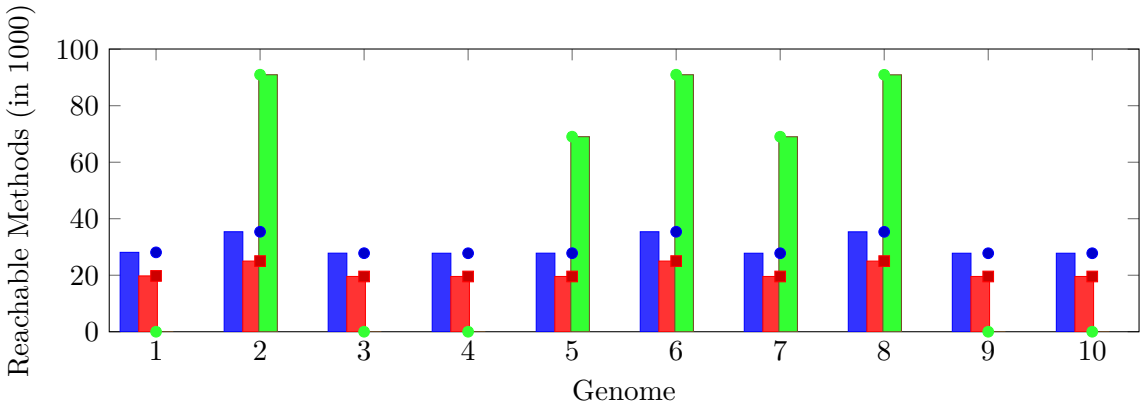


Figure B.2: 2nd generation

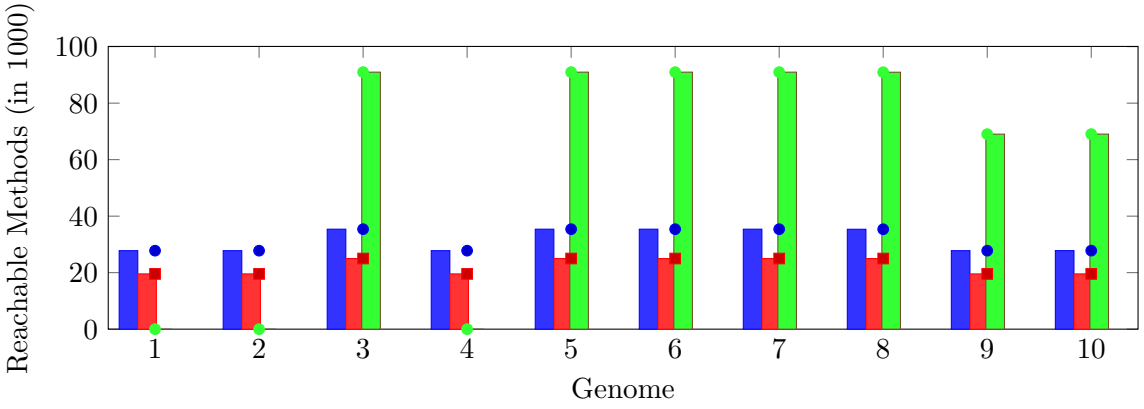


Figure B.3: 3rd generation

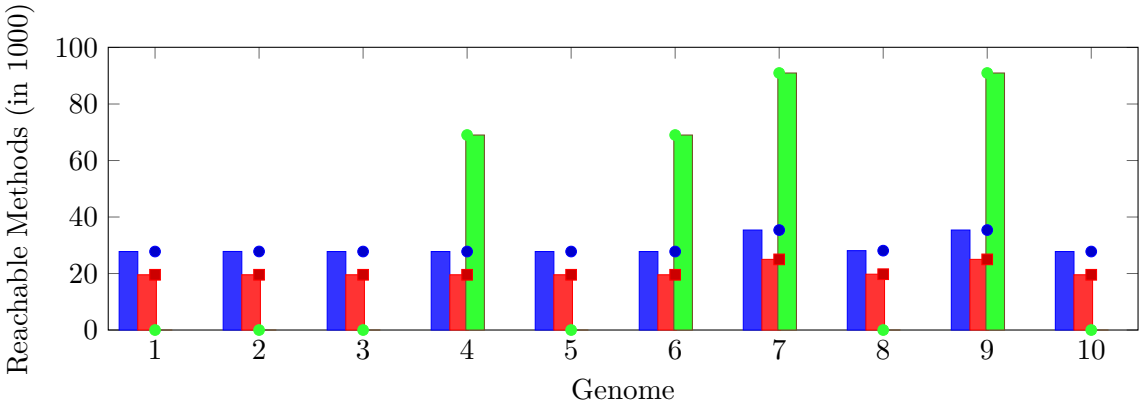


Figure B.4: 4th generation

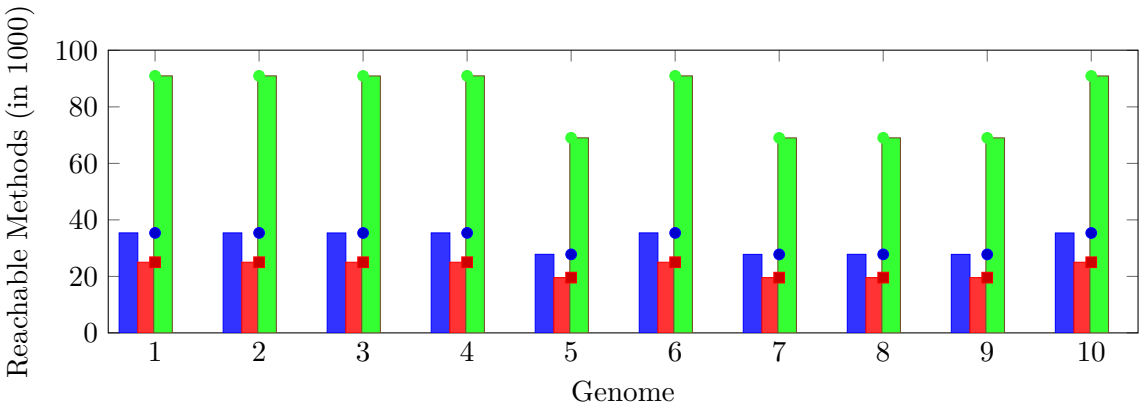


Figure B.5: 5th generation

### B.0.2 Binary Size

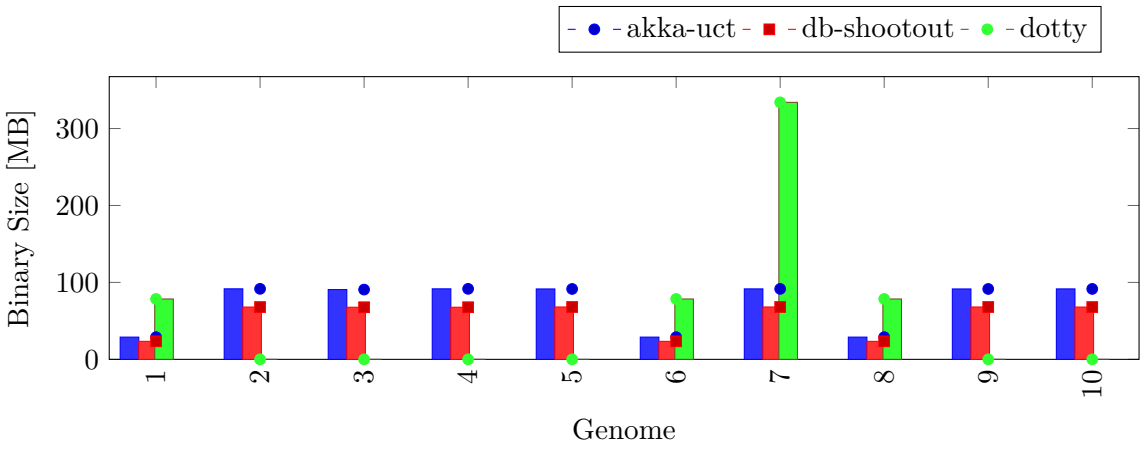


Figure B.6: 1st generation

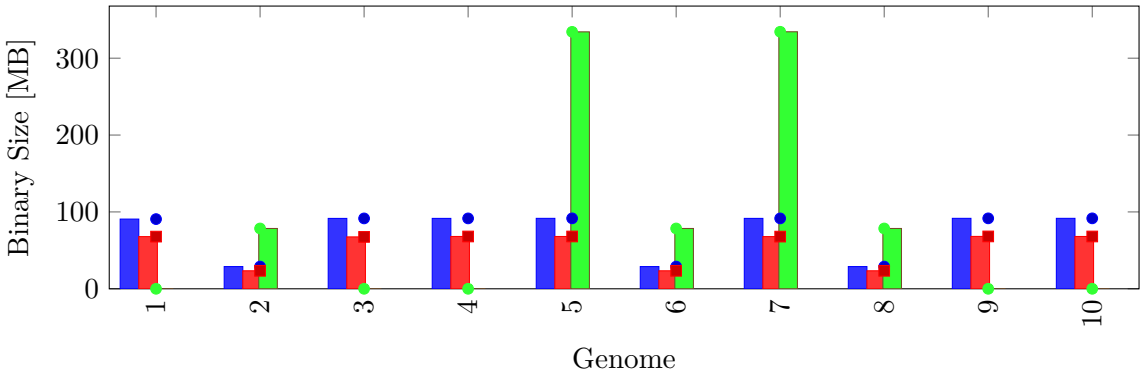


Figure B.7: 2nd generation

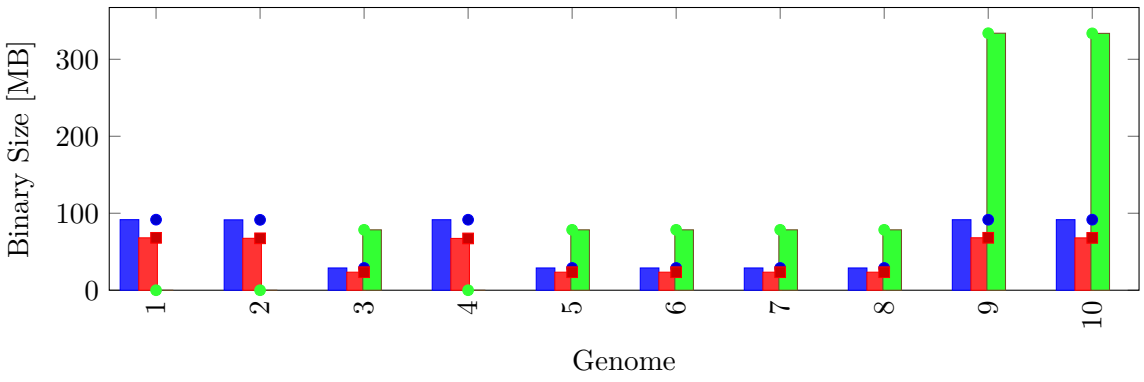


Figure B.8: 3rd generation

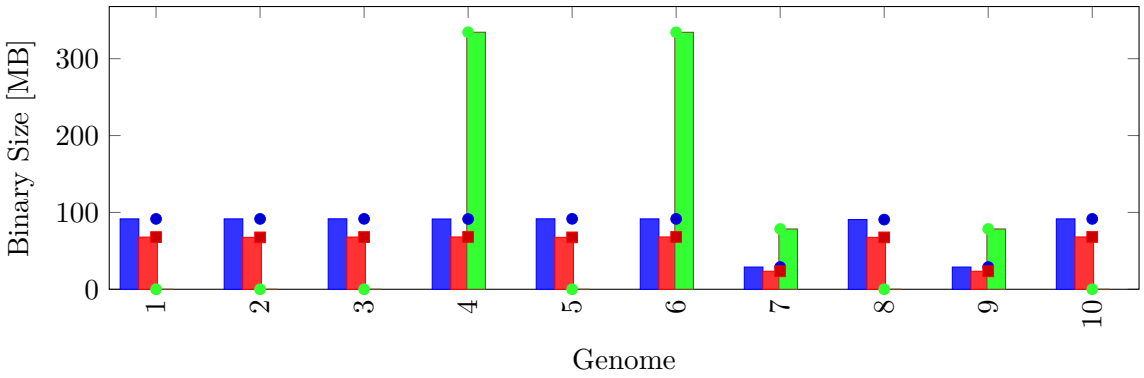


Figure B.9: 4th generation

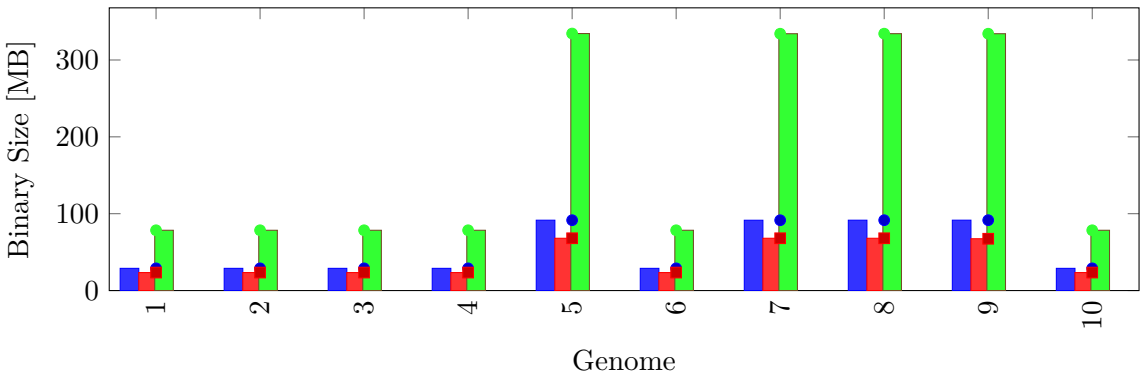


Figure B.10: 5th generation

# Appendix C

# Evolution of training for the graph model

### C.0.1 Reachable Methods

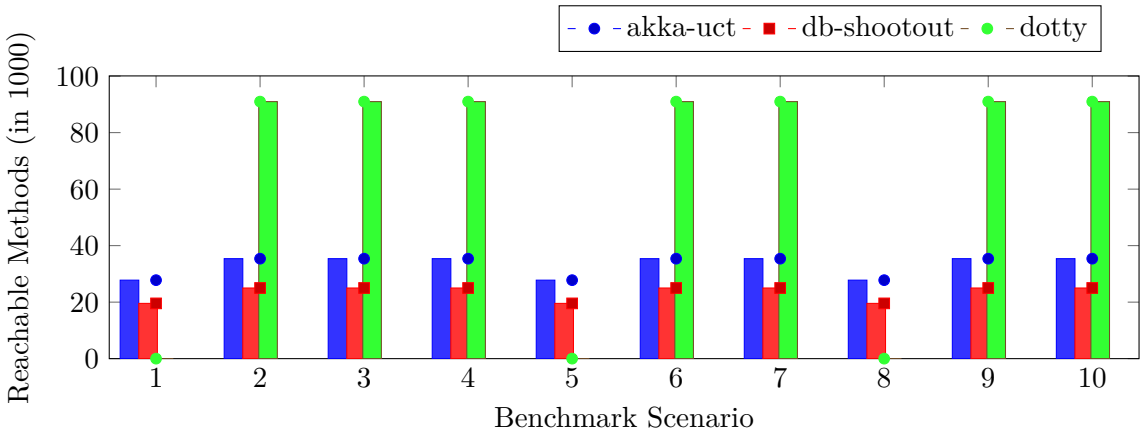


Figure C.1: 1st generation

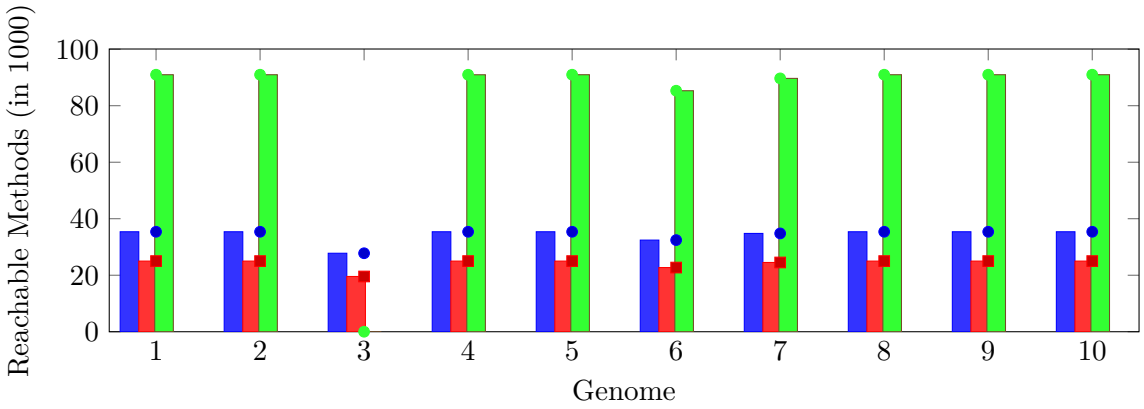


Figure C.2: 2nd generation

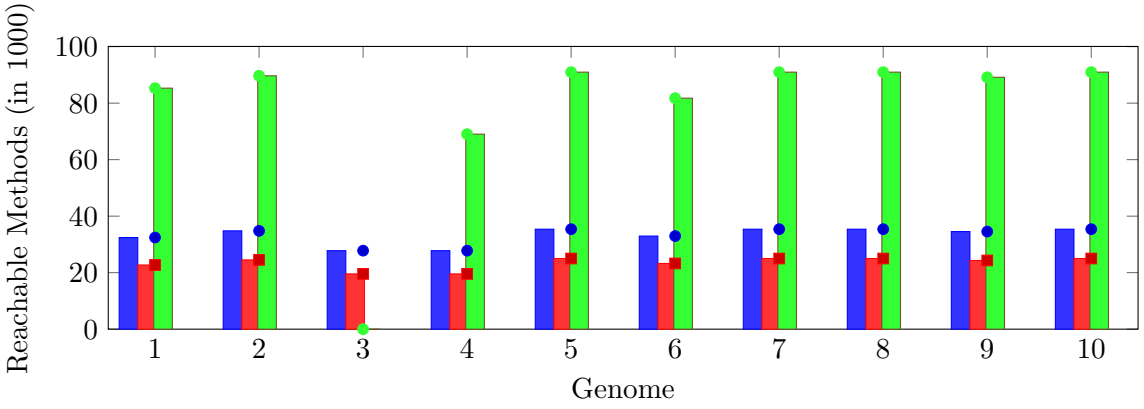


Figure C.3: 3rd generation

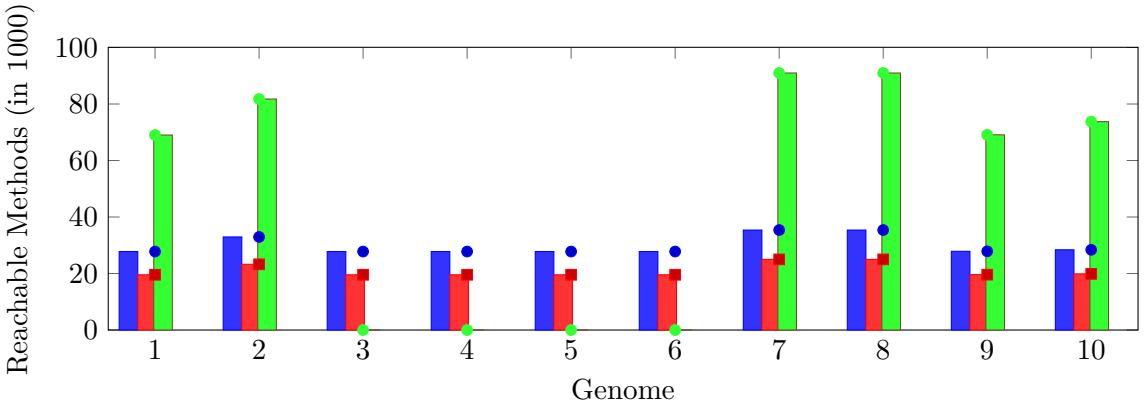


Figure C.4: 4th generation

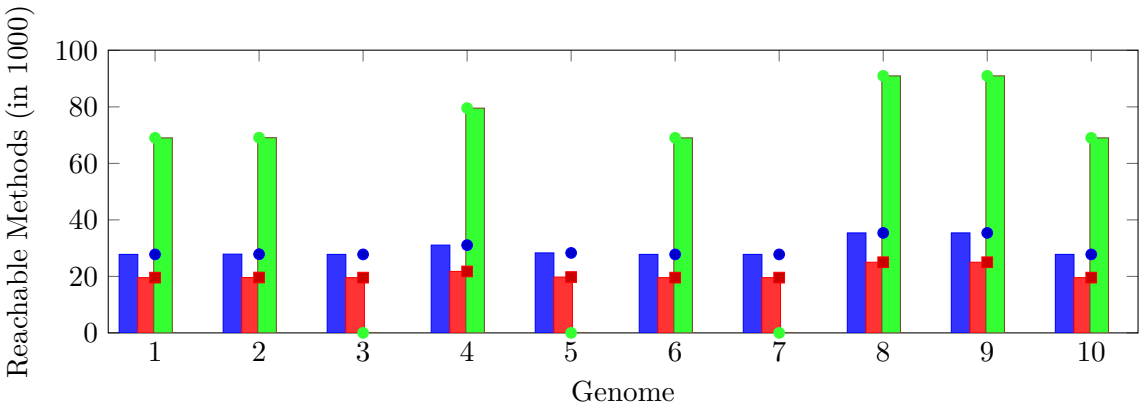


Figure C.5: 5th generation

### C.0.2 Binary Size

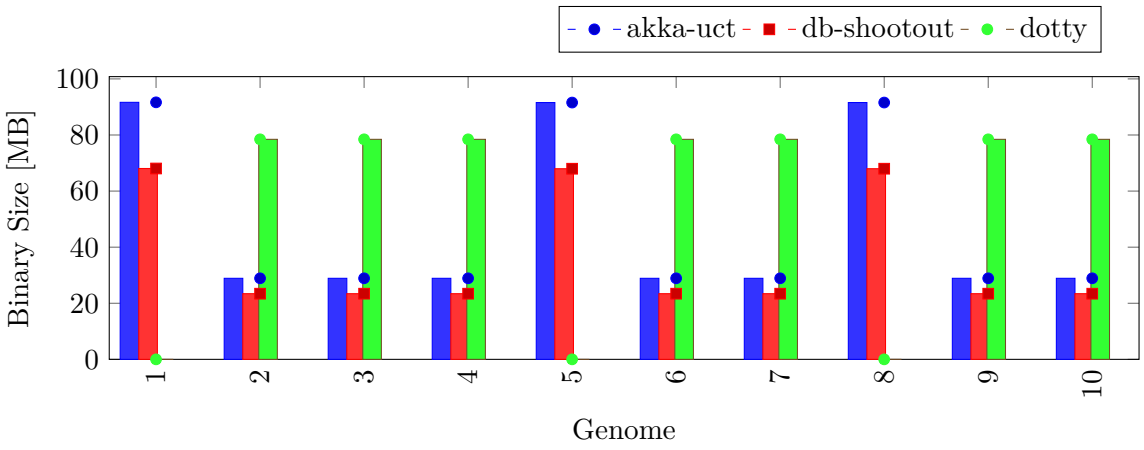


Figure C.6: 1st generation

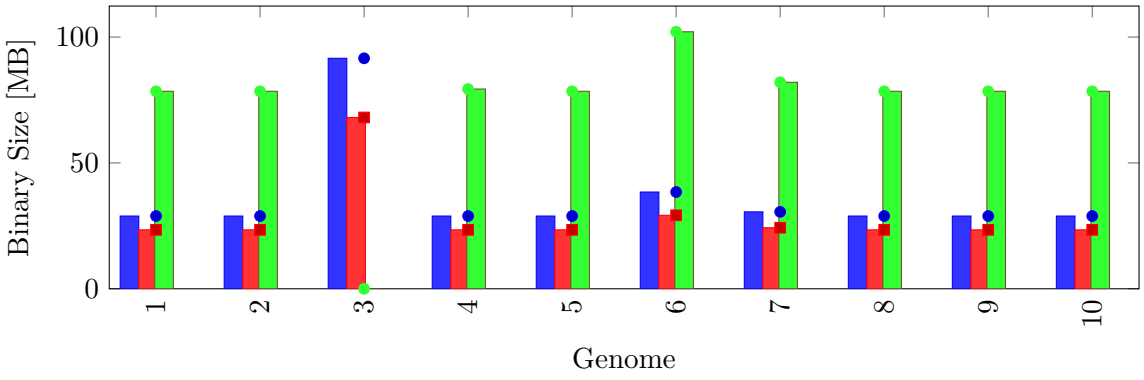


Figure C.7: 2nd generation

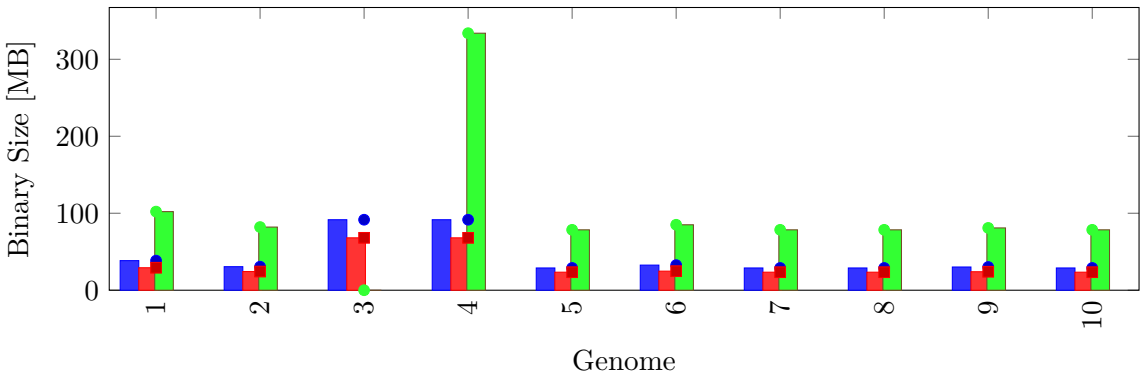


Figure C.8: 3rd generation

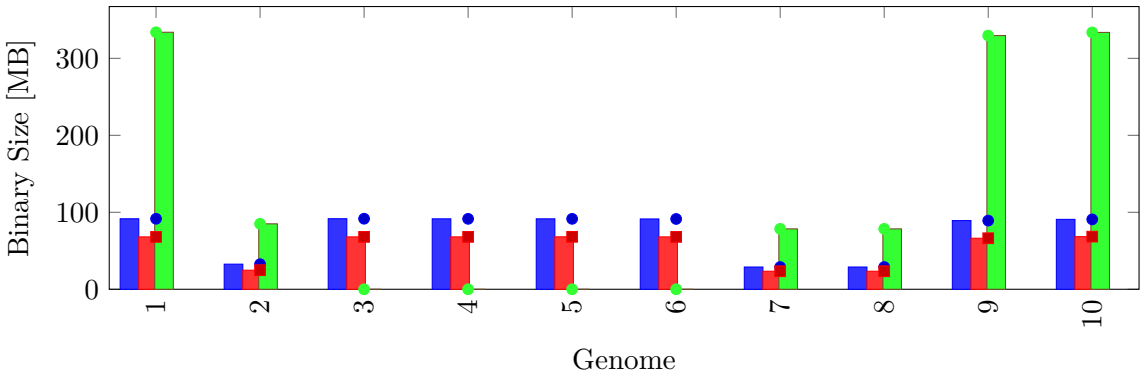


Figure C.9: 4th generation

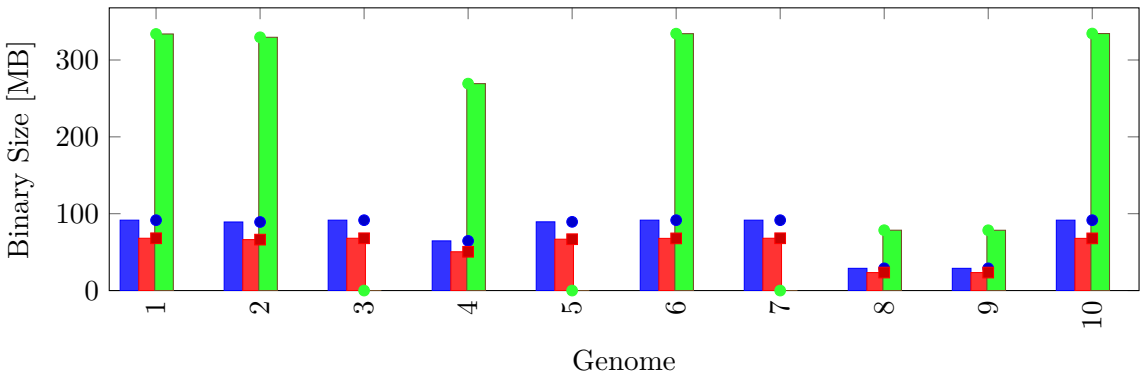


Figure C.10: 5th generation

# Appendix D

# Content of the submitted archive

The documentation and source files are submitted in a zip archive. In the root of the archive, `xkende01.pdf` contains the text of the thesis and `xkende01-latex.zip` contains its LaTeX source files. The `neat` folder contains the training and deployment pipeline of the networks, `README.md` a short guide on how to train and deploy networks, and `requirements.txt` a list of Python dependencies. `configs` contains the JSON configuration files for the best networks of both graph and non-graph networks. `config-xor` contains the original example configuration for the NEAT library, `config-feedforward` and `config-gnn` are its modified variants used to train non-graph and graph models. `evolve.py` is used to launch a training instance, `infere.py` is used to deploy an existing configuration and `api.py` for the deployment of a mock network. From the side of GraalVM's source code located in `graal`, only `InlineBeforeAnalysis.java` has been modified with an extra loop over all invokes in the IR graph, which calls the locally deployed non-graph or graph model for inlining guidance.

```
xkende01.pdf
xkende01-latex.zip
neat/
 configs/
 gnn/
    gcn.py
    graph_network.py
 .env_template
 api.py
 config-xor
 config-feedforward
 config-gnn
 evolve.py
 export_data.py
 infere.py
 requirements.txt
 README.md
graal/
 substratevm/src/
     com.oracle.graal.pointsto/src/
         com/oracle/graal/pointsto/phases/
             InlineBeforeAnalysis.java
```