PES
UNIVERSITY

Dissertation on

"Code Semantic Detection and Optimization"

Submitted in partial fulfilment of the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science & Engineering

UE17CS490B – Capstone Project Phase - 2

Submitted by:

Bhavna Arora	**PES1201700062**
G R Dheemanth	**PES1201700229**
Skanda VC	**PES1201700987**
Mehul Thakral	**PES1201701122**

Under the guidance of

Prof. N.S. Kumar
Professor
PES University

January - May 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)

100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

'Code Semantic Detection and Optimization '

is a bonafide work carried out by

Bhavna Arora	**PES1201700062**
G R Dheemanth	**PES1201700229**
Skanda VC	**PES1201700987**
Mehul Thakral	**PES1201701122**

in partial fulfilment for the completion of seventh-semester Capstone Project Phase - 2 (UE17CS490B) in the Program of Study - Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period January 2021 – May 2021. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 8th-semester academic requirements in respect of project work.

Signature	Signature	Signature
Prof. N.S. Kumar	Dr Shylaja S S	Dr B K Keshavan
Faculty	Chairperson	Dean of Faculty

DECLARATION

We hereby declare that the Capstone Project Phase - 2 entitled **“Code Semantic Detection and Optimization”** has been carried out by us under the guidance of N S Kumar, Professor and submitted in partial fulfilment of the course requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester January – May 2021. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

PES1201700062	**Bhavna Arora**	Bhavna
PES1201700229	**G R Dheemanth**	[illegible]
PES1201700987	**Skanda VC**	Skanda
PES1201701122	**Mehul Thakral**	Mehul

ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. N S Kumar, Department of Computer Science and Engineering, PES University, for his continuous guidance, assistance, and encouragement throughout the development of this UE17CS490B - Capstone Project Phase –2.

We are grateful to the project coordinators, Prof. Silviya Nancy J and Prof. Sunitha R, for organizing, managing, and helping with the entire process.

We take this opportunity to thank Dr Shylaja S S, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support we have received from the department. We would like to thank Dr B.K. Keshavan, Dean of Faculty, PES University for his help.

We are deeply grateful to Dr M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro-Chancellor – PES University, Dr Suryaprasad J, Vice-Chancellor, PES University for providing us various opportunities and enlightenment every step of the way. Finally, this project could not have been completed without the continual support and encouragement we have received from our family and friends.

ABSTRACT

Capstone phase 1 covers the Prediction of code semantics for an unknown code snippet. Multiple approaches and implementations are possible for a given semantic label. For example searching in a sorted array can be linear or binary, but binary search is more efficient in comparison to the former. The different implementations used for a particular code semantic can be distinguished based on their performance and maintenance metrics in various circumstances. An optimal implementation of the algorithm is always preferred.

Code Optimization occurs at different levels of abstraction like source code level, design level, algorithm level, compiler level, assembly level and machine level. Capstone Phase 2 covers the Algorithm level optimization for a given snippet. The program is evaluated on various metrics calculated both dynamically and statically. The time and space complexity use a dynamic model that calculates the metric values by executing the code snippet. Cyclomatic complexity and halstead metric are calculated statically from the code snippet. These metric values are then combined into a composite metric based on user given weights. The programs are compared based on this composite metric and the most optimal version from the dataset is returned. The model works for both Python and C++ programs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER-1

INTRODUCTION

Program/Code optimization is the process of modifying a software system or a piece of code to make it run efficiently or to make it use less resources.

Program optimization can be done at various levels like design level, algorithm level, data structure level, source code level, build level, compile level, assembly level, machine dependent binary level and run time level. We performed optimization at the algorithm level. We take a function as an input and suggest the most optimal alternative which is present in the dataset of known optimized programs.

Optimization can be based on various parameters like time, space, code complexity, etc. We propose a composite metric which is calculated using time, space, cyclomatic complexity and halstead difficulty metric. The weightage or importance for each of those individual metrics can be tuned by the user in order to get the most optimal program depending on what metric is preferred the most.

Time and memory are dynamically calculated metrics. They are calculated by running the program using various sizes of input and finding the complexities using polynomial regression. Later integration is done over the domain of inputs to get a value which would further be used for finding the composite metric.

Cyclomatic complexity and halstead difficulty metric are calculated statically using the given function snippet.

CHAPTER-2

PROBLEM STATEMENT

Given a code snippet in the form of a function as the input, the problem is to find the code semantics of the function and use it to suggest the most optimal solution for that function. The optimal function suggested is in the form of a code snippet.

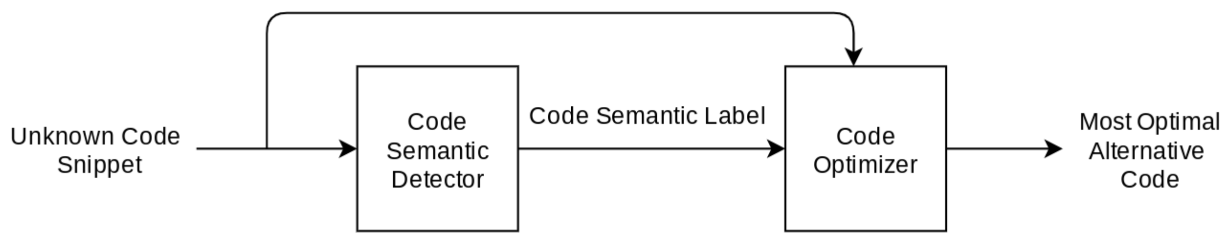


Fig 1. Block diagram of problem statement

The input has to be a valid python or C++ function. The function must be free of syntax errors and semantic errors. Global variables cannot be used.

The most optimal alternative for the given function is suggested from the dataset depending on the weight combination given by the user. The weights signify the importance/weightage which must be given to the 4 components of the composite metric time, memory, cyclomatic complexity and halstead difficulty metric.

CHAPTER-3

LITERATURE SURVEY

3.1 Dynamic vs. Static Optimization Techniques for Object-Oriented Languages [1]

Urs Hölzle

3.1.1 Introduction

This paper highlights the importance of combining static and dynamic analysis in code optimization. The data from previous runs of the program is analysed by the type feedback model to determine the set of possible receiver classes. Concrete type inference analyses the source code to compute the set of receiver classes. Type feedback accounts for relative frequency of the receiver classes unlike type inference that treats them equal and reduces the number of the sends.
The type information computed by type inference is moderated using type feedback to achieve the combined model. Thus optimization is done for methods that are frequently executed.

3.1.2 Proposed Model

Type feedback (also called profile-guided receiver class prediction) extracts type information from previous executions and feeds it back. Each call site in the code along with the list of receiver classes is recorded as program's type profile. Compiler predicts the type of the receiver classes using the type feedback.

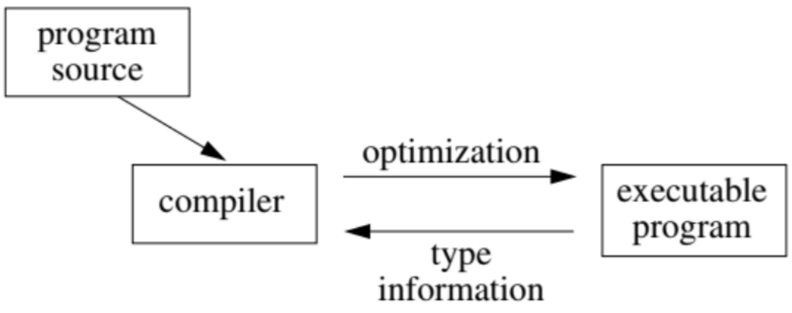


Fig 2. Type FeedBack

Type Inference: to capture control-flow and data-flow from a program. The analysis is necessary to infer which sends are invoked by a dynamically dispatched method and the receiver of the sends along with type or class is also known. The control and data flow are represented using a template. Methods containing sends yield connections to other templates. Incase of send connected to a template, the actual arguments are propagated as formal arguments . with The type is returned by propagating to the output from formal arguments through the template.

3.1.3 Evaluation

The proposed solution is specific to SELF and SELF like OO languages. Type inference is able to reduce the number of dispatches to be executed, along with reducing the size of the application through program extraction. Type feedback provides consistently high performance by scaling well to large and extensible systems, but removes fewer dispatches along with dependence on the run-time info using static feedback.

3.2 Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression [2]

Ling Huang et Al

3.2.1 Introduction

The paper highlights the relationship between execution time of computer programs and features of the programs.

3.2.2 Proposed Model

Prediction model based on program features' data collected from program execution on sample inputs. The main focus is on feature selection and model building with the goal to predict the performance metrics (e.g. execution time) on a particular input.

1.Feature Instrumentation: obtain values of features of the program (e.g. loop counts, branch counts, variable values)

2.Profiling: execute program with sample input and collect values

3.Slicing: a part or subset of a program that contributes to the feature's value. If the whole program is used, the feature is expensive and if a subset is used, the feature is cheap.

4.Modelling: use the feature value and feature costs to build a predictive model on the subset of features

3.2.3 Evaluation

There are two algorithms to learn a SPORE (sparse polynomial regression) model, and showed that both algorithms can predict execution time with more than 93% accuracy for the applications they tested.

3.3 Automatic Algorithmic Complexity Determination Using Dynamic Program Analysis [3]

Istvan Gergely Czibula et Al

3.3.1 Introduction

This paper presents a novel method for automatically determining algorithmic complexity based on runtime measurements.

The performance of a software application is one of the most important aspects for any real life software. But simulation, profiling and measurements performed are usually misleading or incomplete. So an automated tool based on a theoretical model was created to overcome this problem.

The main advantage of knowing the complexity of a method is that it gives an insight into the performance of an operation for large input data sizes.

The main contributions of this paper is to propose and experimentally evaluate a deterministic approach to find the asymptotic algorithmic complexity for a method.

3.3.2 Proposed Model

The steps involved in the proposed model are as follows:

i. Analyse input code to find out the input and output parameters

ii.. Executes program multiple times to record time taken

iii. Use a nonlinear least square data fitting method to find a curve that fits the measured data

iv. Compute RMSE metric (root mean square error) for choosing the best matching function

3.3.3 Evaluation

Method	C++ data set	Java data set
Benchmark 10 complexity classes	76%	23%
Benchmark 6 complexity classes	81%	48%
Implemented approach	90%	97%

Table 1. Accuracies for C++ and Java data sets compared with benchmark implementations

The implemented tool has a higher accuracy of time complexity prediction for both C++ and Java datasets. For C++ dataset the accuracy of the benchmarks are comparable to the implemented tool but for the java dataset the implemented tool out performs the benchmark by around 50%.

3.4 Cisco EIGRP Metric [4]

Cisco Systems

Cisco's EIGRP metric calculation serves as a way to combine different attribute values based on weights assigned to each to get a single composite value.

3.4.1 Introduction

EIGRP is a Cisco Proprietary routing protocol created in the 1980s for finding out the minimum cost path to reach a destination router considering several network parameters like Bandwidth, Load, Delay and Reliability. The value representing the minimum cost was obtained by comparing the composite value (which is the composition of above four attributes) for each available path from source to destination router.

3.4.2 Proposed Model

The weightage given to each of the above attributes while composing their values is controlled by K-values. The mapping of K-values to the attribute is mentioned below:

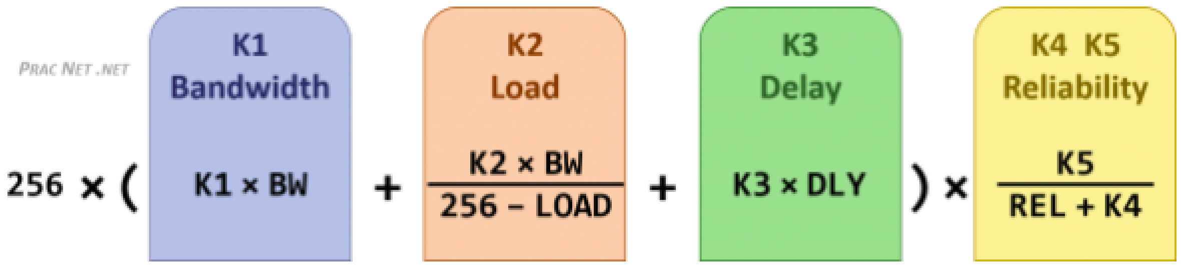
K1 = Bandwidth

K2 = Load

K3 = Delay

K4 & K5 = Reliability

Using these values they obtain EIGRP Composite Metric Formula. That formula is as follows:



PRAC NET .NET
K1
Bandwidth
K2
Load
K3
Delay
K4 K5
Reliability
256 × (K1 × BW + K2 × BW / 256 − LOAD + K3 × DLY) × K5 / REL + K4

Fig 3. EIGRP Composite Metric Formula

The K-values for each attribute can independently vary between 0-255, thus deciding the importance of each attribute in the composite value which will be used for comparing across different paths. If an attribute needs not to be considered while calculation can be given a value of 0.

3.4.3 Evaluation

Since the K-value can go upto 255 so we can decide how much consideration can be given to an attribute. For Example if Load is twice as important than Delay then K2 would be 2 and K3 as 1. Similarly if Load to Delay importance is 2:3 then K2 would be 2 and K3 be 3. Hence EIGRP composite metric formula gives flexibility in choosing which attributes to be considered and their corresponding scale or importance.

This flexibility provided by the EIGRP composite metric formula was utilized in the project to compose Time, Memory, Cyclomatic Complexity and Halstead’s Difficulty metric value to give out a single composite value for the code.

CHAPTER-4

SYSTEM DESIGN

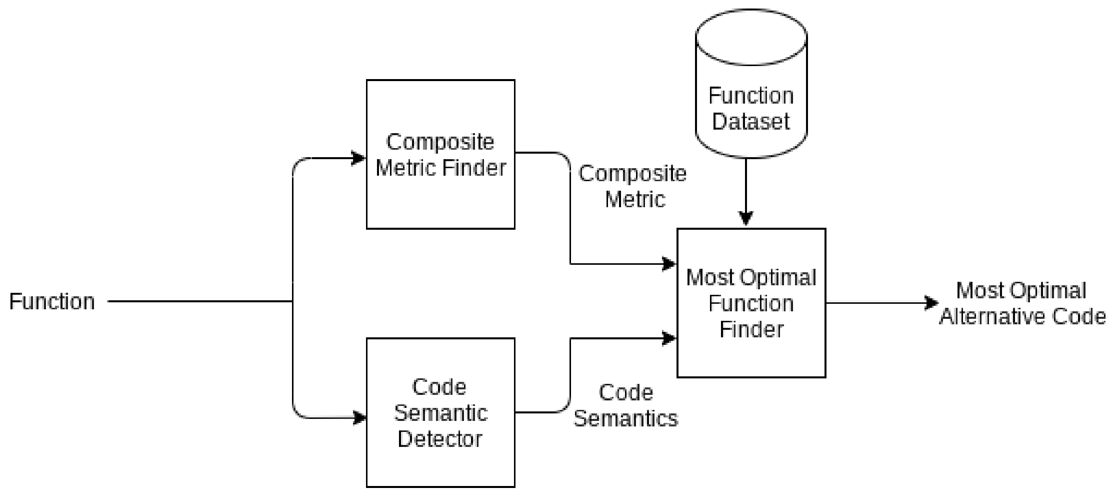


Fig 4. Overall Architecture

The above figure summarizes the overall design of the project.
The user gives a function snippet as input. The function given is then sent to the code semantic detector as well as composite metric finder.
The function dataset contains functions across multiple semantic labels along with their metrics.
The code semantic detector predicts the label i.e the main logic behind the function snippet.
The metric finder measures the function’s time, memory, cyclomatic complexity and halstead difficulty value and combines them all can be used to compare multiple functions with the same semantic label.

All programs in the dataset with the same semantic label as the input function are compared based on the composite metric obtained by accounting for user metric preference given in form of weights. The most optimal function finder decides the relevant function to return based on composite value. Lower value of the composite metric, higher optimized version of the code.

CHAPTER-5

IMPLEMENTATION

5.1 Dataset

The dataset from capstone phase 1 composing of around 500 Python programs was used in phase 2 as well. Along with that a dataset of around 1000 C++ programs was generated. These programs are using Object-Oriented paradigm spanning across different high-level data structures or algorithms. The data structures or algorithms and the corresponding labels are as follows:

1. Array:
    a. SORT
    b. ROTATE_ARRAY

2. Math:
    a. FIB
    b. PALINDROME
    c. POW
    d. UGLY_NUM
    e. SIEVE
    f. SQRT
    g. REVERSE_INTEGER
    h. CATALAN_NUMBER

3. String:
    a. ANAGRAM
    b. STR_SEARCH

4. Linked List:
    a. CYCLE_LL
    b. REVERSE_LL

5. Tree:
    a. INORDER_TRAVERSAL
    b. VALID_BST

   c. LEVELORDER_TRAVERSAL
   d. HEIGHT_BT
   e. MAX_PATH_SUM_BT

6. Graph:
   a. COUNT_ISLANDS
   b. CYCLE_GRAPH

7. Dynamic Programming:
   a. JUMP_GAME
   b. COIN_CHANGE

8. Backtracking:
   a. SUDOKU
   b. RESTORE_IP

5.1.1 Dataset Processing

Dataset was generated by scrapping code snippets from top upvoted leetcode discussion threads. This not only gives the model exposure to different ways in which the problem can be approached but also the optimized way in which that problem can be solved.

Dataset was then pre-processed to make sure programs are in the standard format, have required imports and are unit tested to ensure expected behaviour on execution.

The programs for each label from the scrapped, pre-processed and unit tested dataset were passed through a process of calculating all 4 metrics values and were stored in JSON format which comes handy when comparing all the programs with the label same as test program for the composite metric value based on the weights given by the user.

5.1.2 Generating Pre-labelled dataset

An approximate ranking of the programs is generated for each label using a novel algorithm. The algorithm uses a greedy approach to find out the range of ranks each program can take. And if the rank generated by the model lies within this range then it is considered accurate. To generate the final accuracies, the model is executed on all the programs and it's output is compared with these ranges of ranks.

5.2 Dynamically Calculated Metrics

Time and Memory Metrics are calculated dynamically. The given input function is executed on inputs of various sizes. The time taken to execute is recorded along with peak memory used by the function while executing. The max input size within which the function can execute for 1 sec is determined by the Upper Bound Algorithm. Then values till upper bound are generated and used for executing the function. Polynomial regression is used to determine the time complexity equation. Later integration is performed on the obtained equation over the domain of inputs to get the metric values.

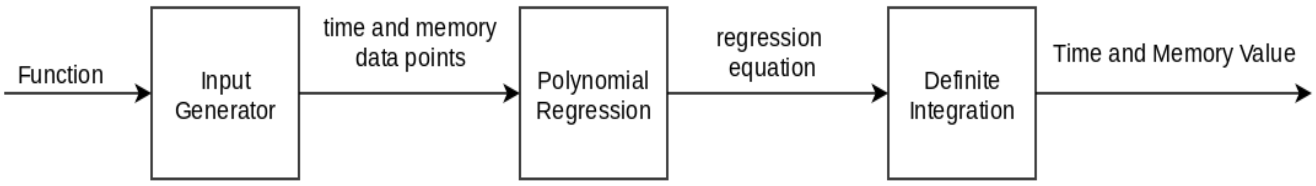


Fig 5. Flow of Operations for Dynamically Calculated Metrics

5.2.1 Input Generator

Input Generator uses the upper bound algorithm to find the max input size within which the function can execute in 1 second. The algorithm is explained in the given diagram. The algorithm is very similar to the binary search algorithm. The time taken and peak memory used are recorded for each run of the function. These values are later used by the polynomial regression algorithm.

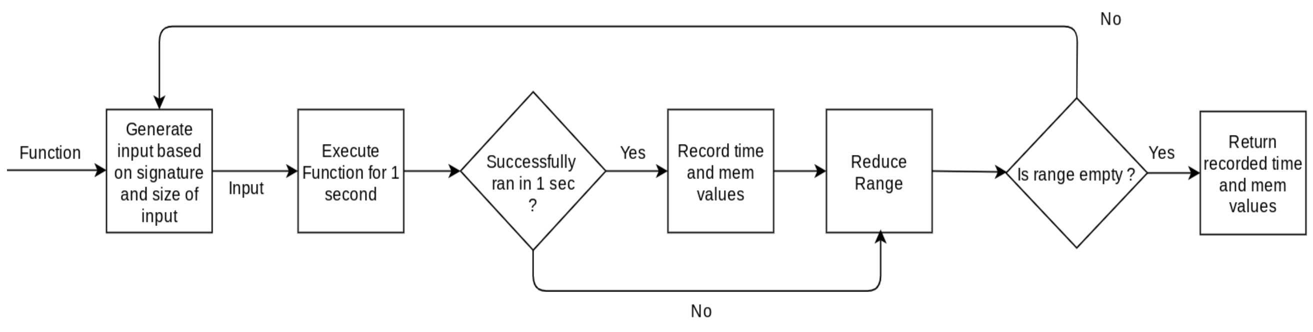

Function
Generate input based on signature and size of input
Input
Execute Function for 1 second
Successfully ran in 1 sec ?
Yes
No
Record time and mem values
Reduce Range
Is range empty ?
No
Yes
Return recorded time and mem values

Fig 6. Input Generator

5.2.2 Polynomial Regression

Polynomial Regression is a type of regression where the relation between independent and dependent variables are modelled as nth degree of polynomial of the independent variable.

Using the recorded values from the input generator, the polynomial regression algorithm is used to fit the data on various degrees. The degree varies from 1 to 9. Later the best polynomial regression model is taken. The best model is chosen based on the R^2 metric. The algorithm is shown in the below figure.

Polynomial regression is used to find the time and space complexities of the given function which are one of the most important metrics used to compare performance. For example, the curve of a O(n^2) algorithm will vary quadratically. This can be found using polynomial regression. So a polynomial regression model with degree =2 will be the model with the lowest R^2 metric.

5.2.3 Time & Memory Metric Value Calculation

The equation from polynomial regression cannot be directly used to compare performance. A scalar value is necessary in order to calculate the composite metric. So the equation obtained is integrated over the domain of inputs in order to get a single scalar value. This is done for both the time and memory equations.

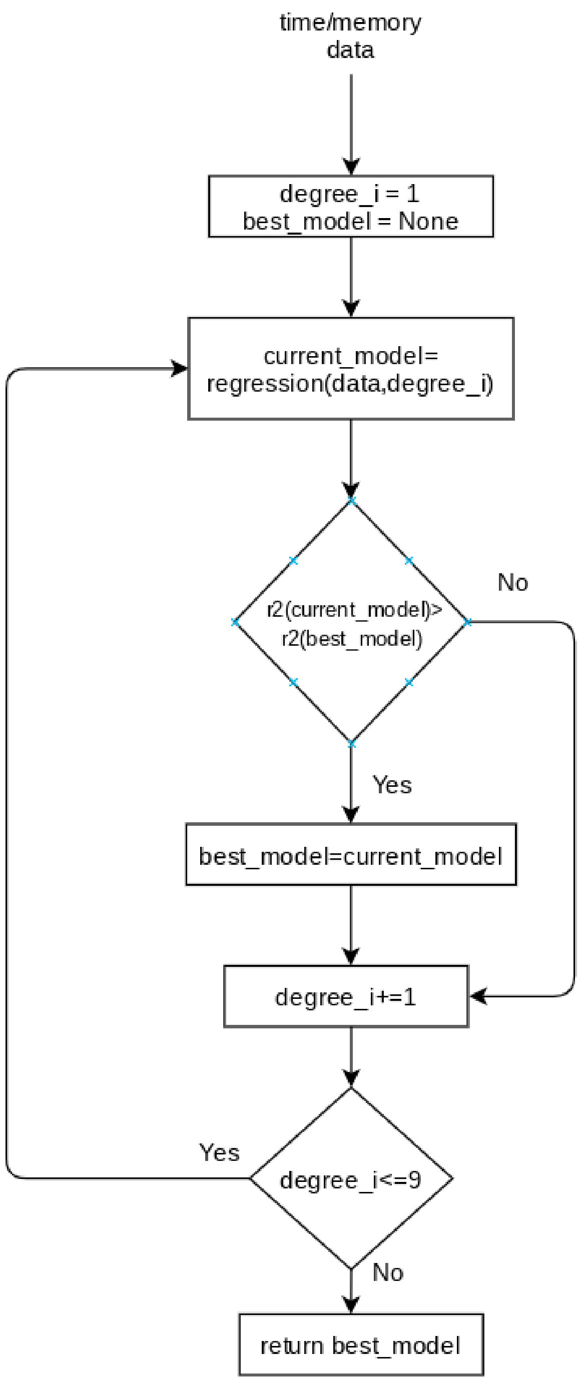


Fig 7. Polynomial Regression for finding the time and space complexities

5.3 Statically Calculated Metric

Cyclomatic Complexity and Halstead Difficulty metric are calculated statically from the code. These values are dependent on the control flow graph and structure of the code. These metrics measure the code's quality and complexity level.

5.3.1 Cyclomatic Complexity

The order of execution of statements depends on the decision making statements. The program with more decision statements is expected to be more complex due to more number of control jumps. Cyclomatic Complexity of a code snippet is the quantitative measure of the number of linearly independent paths in it.

It is calculated as M = E- N +2 P

where,

M: Cyclomatic complexity

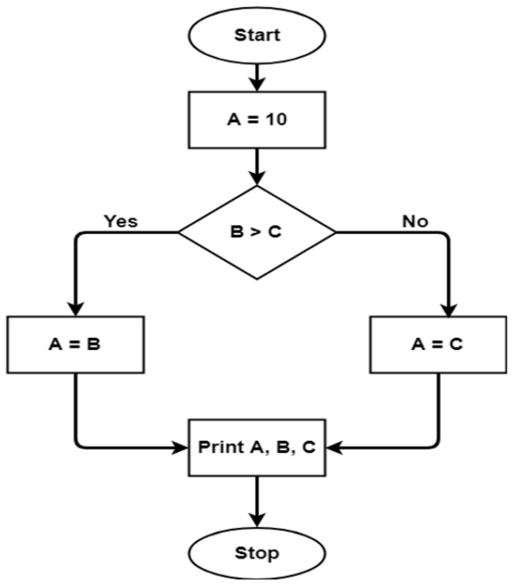
E: no. of edges in the control flow graph

N: no. of nodes in the control flow graph

P: no. of connected components

<u>**Sample Program Snippet**</u>

```
A = 10
   IF B > C THEN
      A = B
   ELSE
      A = C
   ENDIF
Print A
Print B
Print C
```



<u>**Sample Calculation**</u>

No. of Nodes = N = 7

No. of Edges = E = 7

No. of components = P = 1

Cyclomatic complexity = M
= 7 - 7 + 2*1 = 2

Fig 8. Calculation of Cyclomatic Complexity

5.3.2 Halstead Difficulty Metric

Halstead's difficulty metrics depend upon the actual implementation of the program and its measures, it is calculated directly from the operators and operands from source code, in a static manner.

Parameter	Meaning
n1	Number of unique operators
n2	Number of unique operands
N1	Number of total occurrence of operators
N2	Number of total occurrence of operands

Table 2. Indicators to check complexity of module

Metric	Meaning	Mathematical Representation
n	Vocabulary	n1 + n2
N	Size	N1 + N2
V	Volume	Length * Log2 Vocabulary
D	Difficulty	(n1/2) * (N1/n2)
E	Efforts	Difficulty * Volume

Table 3. Halstead Report generated from Indicators

5.4 Composition of metrics

The different static and dynamic metrics are combined in a similar way as paper [4]. Initially, transformation to the metrics values are applied to ensure all the metrics values have similar output range to prevent the one with higher magnitude dominating the composite value. The transformation chosen is *Robust Scaling*. It uses the equation in Fig 9 to get the new values. Since it uses Median and InterQuartile Range, hence is robust to outliers. The output values range from [-1, 1] for values between Q1 and Q3, and less than -1 and greater than 1 for values less than Q1 and greater than Q3 respectively, thus preserving the linear relationship.

$$x_{\text{new}} = \frac{x-M}{Q3-Q1} \qquad x_{new} = \begin{cases} < -1 & \text{if, } x < Q1 \\ [-1, 1] & \text{if, } Q1 \leqslant x \leqslant Q3 \\ > 1 & \text{if, } x > Q3 \end{cases}$$

Fig 9. Robust Scaling transformation

The values are scaled based on the k values which range between 0 and 100. These k values can be considered as weights given to each metric to indicate a metric's importance relative to each other. For example, if the given weights for time, memory, cyclomatic and halstead are [2,1,0,0] then it means that the time metric is twice as important as the memory metric and the other two metrics have no relevance.

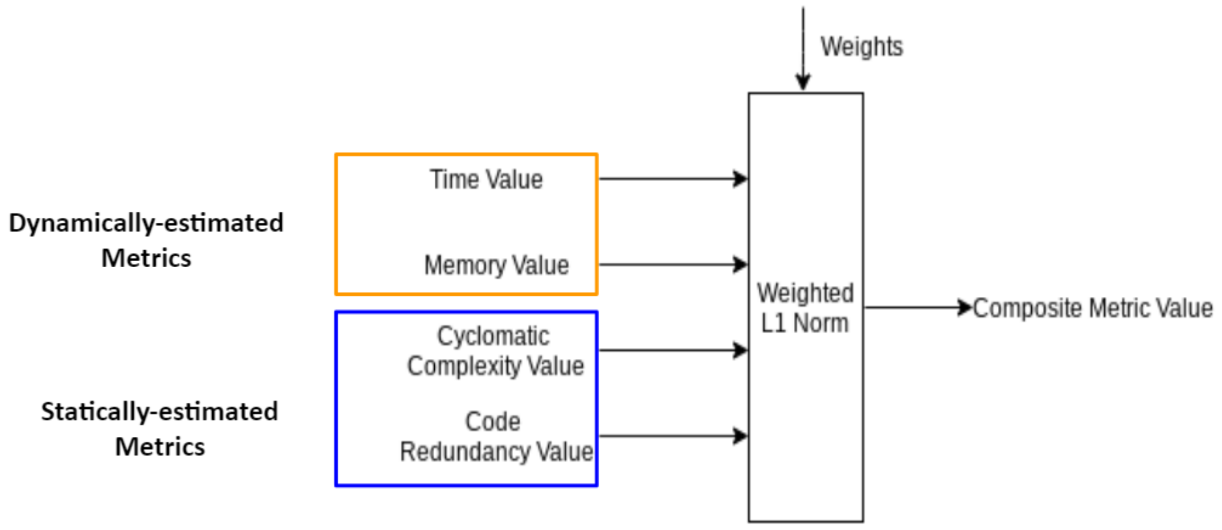


Fig 10. Composite Metric Finder

These weights and the individual metric values are then combined in the form of a weighted L1 norm(manhattan distance) to get the final composite metric value.

CHAPTER-6

RESULTS AND DISCUSSION

The data structure or algorithm wise prediction accuracies for both the models are summarized in the below table:

Data Structure/Algorithm	**Python Prediction Accuracy**	**C++ Prediction Accuracy**
Array	100%	100%
Math	99.39%	99.77%
String	95.23%	100%
Linked List	100%	100%
Tree	100%	99.61%
Graph	95.23%	100%
Dynamic Programming	100%	97.82%
Backtracking	100%	100%
Overall	**98.97%**	**99.71%**

Table 4. Table displaying DS/Algorithm wise prediction accuracy for both models

The data structure or algorithm wise optimization accuracies for both the models are summarized in the below table:

Data Structure/Algorithm	**Python Optimization Accuracy**	**C++ Optimization Accuracy**
Array	85.8%	69.1%
Math	93.05%	86.11%
String	95.06%	73.45%
Linked List	88.88%	41.97%
Tree	78.51%	94.96%
Graph	85.18%	62.08%
Dynamic Programming	91.35%	98.76%
Backtracking	100%	95.06%

Overall	**88.84%**	**87.33%**

Table 5. Table displaying DS/Algorithm wise optimization accuracy for both models

Following summarizes the obtained results and experimentations:

- Obtained optimization and prediction accuracies of both Python and C++ above 80%
- Checked efficacy of different static metrics and memory profilers based on model requirements and implementability
- Tried different scaling techniques namely min-max, standard, etc. but found Robust scaling to be robust to outliers
- Code Vectors for labels are independent of program language => Code Semantic Detection is language agnostic

CHAPTER-7

CONCLUSION AND FUTURE WORK

As part of Capstone Phase 2, an optimized code is suggested for an unknown input code snippet. The unknown code snippet is first identified using the capstone phase 1 model. After which, the optimization for the code is done at the design level by suggesting code implemented using alternate algorithms.

The optimization is done based on four metrics which are calculated for each program in the dataset. In order to compare algorithms, a composite metric was developed to compare the overall performance of algorithms using which we rank the algorithms and output the most optimal one.
A labelled dataset of programs of C++ and python languages are web scraped from common online coding platforms, covering the most commonly used programming algorithms. A novel algorithm was developed to provide an approximate rank which was compared with the output rank to calculate the accuracy.

Following are proposed future works for the project:

- Can be extended to develop as a software for accessing code submission quality in student evaluation systems, online coding platforms, interview settings, etc.
- Can use more relevant metrics particularly on statically evaluated part and refining memory profiling on dynamically evaluated part.
- Setting up a pipeline for automated addition of famous program labels from LeetCode and fetching new implementation ways for existing labels.

REFERENCES

[1] Hölzle, U. and Agesen, O., 1995. Dynamic versus Static Optimization Techniques for Object‐Oriented Languages. Theory and Practice of Object Systems, 1(3), pp.167-188.

[2] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. 2010. Predicting execution time of computer programs using sparse polynomial regression. In Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 1 (NIPS'10). Curran Associates Inc., Red Hook, NY, USA, 883–891.

[3] I. Czibula, Z. Oneţ-Marian, and R.-F. Vida, “Automatic Algorithmic Complexity Determination Using Dynamic Program Analysis,” Proceedings of the 14th International Conference on Software Technologies, 2019.

[4] “EIGRP Metric,” Practical Networking.net, 07-Feb-2017. [Online]. Available: https://www.practicalnetworking.net/stand-alone/eigrp-metric/.[Accessed: 18-Feb-2021].

[5] “Halstead Complexity Metrics” [Online]. Available: https://en.wikipedia.org/wiki/Halstead_complexity_measures .[Accessed 14-April-2021]

[6] B. Roy, “All about Feature Scaling,” Medium, 07-Apr-2020. [Online]. Available: https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35. [Accessed: 14-Apr-2021].

APPENDIX A - USER MANUAL

Steps to Install:

1. Install docker
2. Install docker-compose
3. Clone the repo
4. sudo docker-compose up --build

Steps to Train/Learn new Functions via UI :

1. Type localhost:8090 in browser to access UI
2. Select language
3. Type code/upload file. The Function name will be taken as label
4. Click on Learn button

Steps to Test/Predict Functions via UI:

1. Type localhost:8090 in browser to access UI
2. Select language and model
3. Type code/upload file.
4. Click on Test Button

Steps to Optimize Functions via UI:

1. Type localhost:8090 in browser to access UI
2. Select language
3. Type code/upload file.
4. Modify the weights if necessary by selecting "Add custom weights" checkbox
5. Click on Optimize Button

Steps to Rank Function with respect to functions in Dataset via UI:

1. Type localhost:8090 in browser to access UI
2. Select language
3. Type code/upload file.
4. Modify the weights if necessary by selecting "Add custom weights" checkbox
5. Click on Rank Button

Steps to Compare 2 or more functions via UI:

1. Type localhost:8090 in browser to access UI
2. Select language
3. Type the functions
4. Modify the weights if necessary by selecting "Add custom weights" checkbox
5. Click on Compare Button

Steps to Learn new Functions via Library function call:

1. import model1.dataset as g1
2. g1.csv_dataset.add((function_name,function))
3. g1.json_dataset.add((function_name,function))

Steps to Test/Predict Functions via Library function call:

1. import model1.predict as mp1
2. m = mp1.predict(function)

Steps to Optimize Functions via Library function call:

1. import model1.optimize as op1
2. m = op1.optimize(function)

Steps to Rank Functions via Library function call:

1. import model1.optimize as op1
2. m = op1.rank(function_arr,"python",[1,0,0,0]) #where function_arr is an array of functions

Steps to Compare Functions via Library function call:

1. import model1.optimize as op1
2. m = op1.compare(function_arr,"python",[1,0,0,0]) #where function_arr is an array of functions

Note :

1. function above can be any callable. To call C/C++ functions, use the python module cppyy.

```
import model1.predict as mp1

import cppyy

c_func_str="int test(int a){ return a+1;}"

cppyy.cppdef(c_func_str)

m = mp1.predict(cppyy.gbl.test)
```

2. giving functions with special pre conditions like strings having only digits or only special characters or functions having types which are not supported by dynamic analysis model, then, will have to specify the pre conditions in dictionary passed as default arg for that parameter.

For example : In case of a function validating IP address string, the string must not contain upper case, lower case and special characters. So in that case, the function signature will be

```
def IP_VAL(s:str={'upper_count':0,'lower_count':0, 'special_count':0, 'digits_count': 8}): pass
```

So here by making upper case, lower case and special character count as 0 and digits count as 8, you are specifying the precondition as string having 8 digits with no upper case, lower case and special characters.

3. Default Precondition when no precondition is specified is

```
{"start":0,"end":11,"len_list":8,"upper_count":3,"lower_count":3,"digits_count":3,"special_count":
3 ,"wspace_count":3,"generator": lambda : random.random}
```

So when function signature is like this,

```
def IP_VAL(s:str): pass
```

Then, random strings of length 15 are created which have 3 upper case, 3 lower case, 3 digits and 3 special characters and 3 whitespaces. If s was list, then random list of length 8 is generated.

4. start and end specify the range of random numbers to be generated. Ex : if start=1, end=1000, then numbers in the range [1,1000] are generated.

5. generator specifies a custom random data generator for an argument. It is a callable which takes no arguments. It can also be a string which has a function inside. The function must not take any arguments.
6. Optimization, Rank and Compare by default give weightage of 1 to time and 0 for the memory, cyclomatic complexity and halstead's difficulty. Can modify the weights to give more/less importance to various parameters

To determine prediction/optimization accuracy label wise for dynamic model for Python and C++:

1. cd backend
2. Add the labels for which accuracy has to be determined in the labels list in dataset_server.py. Available labels can be referred from mapping.json
3. Run python dataset_server.py for Python and the results can be seen in results.txt
4. Run python cdataset_server.py for C++ and the results can be seen in results.txt