Media Engineering and Technology Faculty

German University in Cairo

Imperative Programming Languages Capturing P and NP

Bachelor Thesis

Author: Adham Mohamed Ahmed

Supervisors: Haythem O. Ismail

Submission Date: 1 June, 2023

Media Engineering and Technology Faculty

German University in Cairo

Imperative Programming Languages Capturing P and NP

Bachelor Thesis

Author: Adham Mohamed Ahmed

Supervisors: Haythem O. Ismail

Submission Date: 1 June, 2023

This is to certify that:

(i) The thesis comprises only my original work toward the Bachelor's Degree

(ii) Due acknowledgements has been made in the text to all other material used

---

Adham Mohamed Ahmed

1 June, 2023

Acknowledgments

First and foremost, I would like to praise Allah the Almighty, the Most Gracious, and the Most Merciful for His blessing given to me during my study, in completing this thesis, and in my life, and peace be upon Muhammad his servant and messenger.

I want to express my deepest gratitude to Professor Haythem Ismail for his invaluable guidance, support, and mentorship throughout this thesis. His expertise and insightful feedback were instrumental in shaping the direction and quality of this work.

I am also indebted to my friends who accompanied me on this journey and provided unwavering support. Specifically, I would like to thank Mahmoud Hossam and Yahia Mostafa for their relentless efforts in testing the project and offering valuable feedback. Furthermore, I am grateful to Mahmoud Jobeel, Ahmed Hatem, and Ibrahim Abou Elenein for reviewing my thesis and giving me feedback on how to improve it. and for their various advice throughout the thesis that helped me overcome a lot of challenges that I faced.

Lastly, I would like to express my deepest thanks to my family, especially my mother and father, for their unwavering love, encouragement, and understanding throughout my academic pursuit. Their constant support and belief in my abilities have been a driving force behind my achievements.

I am genuinely grateful to everyone who has contributed to the completion of this thesis, directly or indirectly. Your support has been invaluable, I am indebted to you all.

Abstract

Computational complexity is a fundamental field of study in computer science that focuses on understanding the level of difficulty involved in solving computational problems. Complexity classes, such as P and NP, provide a framework for classifying problems based on their computational complexity. This thesis explores the capture of these complexity classes within imperative programming languages, specifically fragments of Java.

The thesis begins by introducing the language A as the foundation of the two fragments of Java. A fragment called $\mathcal{A}_{\mathcal{P}}$ is presented, which ensures that the code written in it belongs to class P. The expansion and coding flow within $\mathcal{A}_{\mathcal{P}}$ are discussed, highlighting the steps taken to verify compliance with the rules specified by Lotfallah[2].

Next, the focus shifts to capturing the complexity class NP. A fragment is known as $\mathcal{A}_{\mathcal{NP}}$ is introduced, built upon the formal definition of NP problems. $\mathcal{A}_{\mathcal{NP}}$ is implemented upon the implementation of $\mathcal{A}_{\mathcal{P}}$, and is designed to verify solutions correctness in polynomial time, by iterating over possible permutations of the solution which might be exponential and then checking for acceptance by a verifier algorithm.

Implementation details for both $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{NP}}$ are presented, including constraints, structure, parsing, and storing mechanisms. The code structure and techniques used in capturing P and NP are outlined, providing a practical understanding of how these fragments can be utilized in code development.

Through this research, a deeper understanding of computational complexity and its relationship with imperative programming languages has been achieved. This work contributes to the advancement of computational complexity theory, providing practical tools that can be used to increase the understanding of complexity classes.

Contents

Acknowledgments **V**

Abstract **VII**

Contents **X**

1 Introduction **1**

2 Capturing P **3**

2.1 The Language $\mathcal{A}$ 3

2.2 Fragment for P ($\mathcal{A}_{\mathcal{P}}$) 6

2.3 Proof that $\mathcal{A}_{\mathcal{P}}$=$P$ 8

2.4 Expanding $\mathcal{A}_{\mathcal{P}}$ 12

2.5 Coding flow in $\mathcal{A}_{\mathcal{P}}$ 13

3 Implementation of $\mathcal{A}_{\mathcal{P}}$ **15**

3.1 Constrains 15

3.2 Structure 16

3.3 Parsing and Storing 23

4 Capturing NP **27**

4.1 Fragment for NP ($\mathcal{A}_{\mathcal{NP}}$) 27

4.2 Proof that $\mathcal{A}_{\mathcal{NP}}$=$NP$ 28

4.3 Implementation of $\mathcal{A}_{\mathcal{NP}}$ 30

5 Testing **35**

6 Conclusion **41**

7 Future Work **43**

References **45**

Chapter 1

Introduction

Computational complexity is an important area of study in computer science focusing on the level of difficulty of solving computational problems. The theory of complexity classes is a fundamental tool for understanding the limits of computation and is used to classify problems based on their computational difficulty.

One of the most famous complexity classes is P, which consists of problems that can be solved in polynomial time by a deterministic Turing machine. On the other hand, the class NP consists of problems for which a solution can be verified in polynomial time by a non-deterministic Turing machine. programmers need to understand these two complexity classes and the field of computational theory in general, but this is not so common since these aspects are theoretical and are not faced a lot while programming in most programming languages like C++, Java, or Python.

In this thesis, we present imperative programming languages that capture the complexity classes P and NP. The languages are fragments of Java and are designed to verify that the code written in it satisfies certain rules specified by Lotfallah[2] that ensure that the written code belongs to class P or NP respectively. We then use the Java compiler to check if the code follows the rules of Java, and at the end, the code is executed by the Java virtual machine, our approach is based on expanding the fragment of code presented in the paper by Lotfallah[2] to obtain the largest possible fragment of Java that still ensures that the code written does belong to P or NP.

The thesis is split into seven chapters. Chapter one is an introductory chapter that overviews the research problem and the aim of the thesis. Chapter two presents a fragment of Java that captures the complexity class P and goes into deeper details on how it captures it and the development of this fragment. Chapter three goes through the implementation of the fragment described in chapter two and explains the structure of the code written. Chapter four describes a fragment of Java that captures the complexity class NP and it provides the implementation outline of this fragment. The fifth chapter shows how the testing was done for both fragments and the debugging of the code until it reached its current state. Chapter Six concludes and sums up the thesis, and the last chapter, chapter seven opens the door for future work that could be done based on this thesis.

Chapter 2

Capturing P

Lotfallah[2] described general rules to ensure that the code belongs to P or NP respectively, we now map these rules to Java language with some modifications in these rules to make it more generalized to appropriately map to the high-level style of code in Java language, while maintaining that these rules capture the P class, the proof is also done on the modified set of rules instead of the original rules specified by Lotfallah.

2.1 The Language $\mathcal{A}$

$\mathcal{A}$ is a fragment of Java that contains the following:

- Three data Types boolean, 32-bit integer, and 64-bit integer

  Example:

```
boolean isTrue;
int counter1;
long counter2;
```

- Arrays of the given data types whose elements can be accessed by index and whose length can be accessed via attribute length (e.g. Array[0], Array.length)

  Example:

```
boolean[] valid=new boolen[5];
long[] counterArray=new long[5];
boolean[][]visited=new boolean[10][10];
```

- Simple Assignment Statements

  Example:

```
int a=5;
boolean b=false;
int c;
int[] d={1,2,3};
c=d.length+d[0];
```

- Boolean, arithmetic operations {+, −, *, /, <, >, etc..}

  Example:

```
int a=5;
int b=6;
int sum=a+b;
boolen lessThan=(a<b);
```

- If conditions

  Example:

```
int x=5;
int y=5;
if((x==y)&&(x==5)){
    x=y+1;
}
```

- Methods definitions and method calls

  Example:

```
static boolean equals(long leftHandSide, long rightHandSide){
    return leftHandSide==rightHandSide;
}

static boolean arrayEquals(long[] leftHandSide, long[] rightHandSide
        , long index){
    if(!equals(leftHandSide.length=rightHandSide.length)) return false;
    if(equals(index,leftHandSide.length))return true;
    return (equals(leftHandSide[index],rightHandSide[index]))&&
        arrayEquals(leftHandSide,rightHandSide,index+1);
}
```

- Recursive Assignment Statements

  Example:

```
int a=5;
int b=5;
boolean eq=equals(a,b);
```

- For loops

  Example:

```
int a=0;
for(int i=0;i<5;i++){
    a=a+1;
}
```

- While loops

  Example:

```
int a=0;
while(a<5){
    a=a+1;
}
```

2.2 Fragment for P ($\mathcal{A}_{\mathcal{P}}$)

$\mathcal{A}_{\mathcal{P}}$ is a fragment of $\mathcal{A}$ which captures P languages ($\mathcal{A}_{\mathcal{P}}$=P), This fragment has the following limitations:

1. A global variable N must be defined in the code, it will be used to limit the complexity of the code.

2. The value of N can never be changed.

3. A variable Lim will be kept in all function calls to limit the depth of the function call, it will be the first parameter of all functions.

4. The value of Lim should not be changed inside the function.

5. Any for loop must count up to a polynomial in Lim or a polynomial in N.

```
for(int i=0;i<P(Lim);i++){

}
for(int i=0;i<P(N);i++){

}
```

   where P(N) is a polynomial in N

6. The value of the loop counter should not be changed inside the loop

7. Any while loop condition must be of the form $(i \leq P(N))\&\&C$ where P(N) is polynomial in N and C is other conditions the loop must also contain an increment of the counter in each iteration and the value of the counter is changed nowhere else in the loop

```
int i=0;
while(i<P(n)&&(C)){
    //some code that doesn't change i
    i=i+1
}
```

```
where P(n) is a polynomial in n
```

8. Function calls can't occur inside any loop
9. The limiter variable Lim should be passed as $Lim/2$ in the function call
10. If there is only one function call inside the function the limiter variable n can be passed as n-1

2.3 Proof that $\mathcal{A}_{\mathcal{P}}$=$P$

There are three steps to prove that $\mathcal{A}_{\mathcal{P}}$=$P$

- Prove that $\mathcal{A}_{\mathcal{P}} \subseteq P$.
- Solve the circuit value problem (CVP) with $\mathcal{A}_{\mathcal{P}}$ which is a P-complete problem[1]
- Prove that $\mathcal{A}_{\mathcal{P}}$ is closed under first-order reduction

1. Proving that $\mathcal{A}_{\mathcal{P}} \subseteq P$

   We can see that the complexity of the conditions and assignments is constant O(1).

   We can also see that the complexity of the loops is a polynomial time of n because every single loop runs up to a polynomial of N so the nesting of the loops is the multiplication of the complexities of all the loops which results in a polynomial time of N as well O(P(N)).

   Now we will be left with only the function calls, we can see that each function call will run in polynomial time in all parts of the code that are not another

function call because the code consists of loops, conditions, and assignments which are all polynomial except the other function calls so the complexity can be written as f(N)*p(N) where p(N) is a polynomial in n and the f(N) is the number of function calls in the entire code, now we only have to prove that f(N) is also a polynomial in N.

Each function in $\mathcal{A}_\mathcal{P}$ has a constant number of function calls inside it because a function can't be called in loops so all of the function calls are written by hand independent of N, now let c be the maximum occurrence of function calls in any function in the written code, we can also see that the deepest call can be O(log(N)) deep because in each call the first parameter is reduced by half and we can divide a number in half only log(N) times, so if we have an O(log(N)) layers and in each layer, we have c calls for each call in this layer which is the maximum number of calls in a function, so the number of calls per layer will be 1, c, c^2, c^3, ..., $c^{O(log(N))} \leq c^{O(log(N))}$*O(log(N)) , and $c^{O(log(N))}$*O(log(N))=$e^{O(log(N))*log(c)}$*O(log(N))=O($N^{log(c)}$)*O(log(N)=O($n^{log(c)}$) and since c is constant we can see that f(n)=O($n^{log(c)}$) and is polynomial in n so $\mathcal{A}_\mathcal{P} \subseteq P$

2. Solving the circuit value problem (CVP) with $\mathcal{A}_\mathcal{P}$. The circuit value problem is described as follows given a circuit consisting of *AND*, *OR*, and *NOT* gates, compute the value of the output of the circuit given the initial inputs. The following is a solution to the CVP problem which was written and tested in $\mathcal{A}_\mathcal{P}$. We will represent the CVP as follows:

   - An integer n which represents the number of gates in the circuit
   - An array of n+1 booleans that represents the stat of the variables ON or OFF (note that a binary circuit with n gates can have at most n+1 used variables)
   - an array of n integers of 0, 1, or 2 which represses that type of gate 0 for OR gate 1 for AND gate and 3 for NOT gate

- two arrays of n integers representing the 2 inputs of each gate each value at index i will be between 0 and n+i if the value is from 0 to n then this input is the variable at that position and if it is more than n then the input of this gate is the result of the gate at that value minus n

the following is an $\mathcal{A}_{\mathcal{P}}$ function that solves the circuit value problem

```
static boolean CV(int n, boolen[] variables, int[] gates
        , int[] firstInputs, int[] secondInputs){
    boolean[] results=new boolean[2*n+1];
    for(int i=0;i<=n;i++){
        results[i]=variables[n];
    }

    for(int i=0;i<n;i++){
        if(gates[i]==0){
            results[i+n+1]=result[firstInputs[i]]|result[secondInputs[i]];
        }else if(gates[i]==1){
            results[i+n+1]=result[firstInputs[i]]&result[secondInputs[i]];
        }else{
            results[i+n+1]=!result[firstInputs[i]];
        }
    }

    return results[2*n];
}
```

we can see that the above function is in $\mathcal{A}_{\mathcal{P}}$ and since the CVP is complete in P via first-order reduction then any problem in P can be solved using $\mathcal{A}_{\mathcal{P}}$ if $\mathcal{A}_{\mathcal{P}}$ is closed under first-order reduction

3. Proving that $\mathcal{A}_\mathcal{P}$ is closed under first-order reduction

   We can show that $\mathcal{A}_\mathcal{P}$ is closed under first-order reduction by implementing all the first-order queries using $\mathcal{A}_\mathcal{P}$

   - the following are implementations for $\neg$, $\wedge, and \vee as \mathrm{A}_P$ functions respectively

```
static boolean not (int n, boolean a){
    return !a;
}
static boolean and (int n, boolean a, boolean b){
    return a&b;
}
static boolean or (int n, boolean a, boolean b){
    return a|b;
}
static boolean implies (int n, boolean a, boolean b){
    return !a|b;
}
```

   - any relation can be computed in polynomial time by looping over all tuples of the relation and checking if the given tuple exists in them.

```
static boolean parent(int n, int father, int son, int[][]parent){
    for(int i=0;i<n;i++){
        if(parent[i][0]==father&&parent[i][1]==son)return true;
    }
    return false;
}
```

```
the above code is an example of the parent relation
```

- now we can see that using these operations without $\forall$ and $\exists$ can be represented as one function with variable inputs (e.g. $FO(X_0, X_1...., X_n)$)
- so using that we can implement both quantifiers $\forall$ and $\exists$ in polynomial time by looping over the alphabet for each quantifier and substituting that variable by each value in the alphabet in case of $\exists$ only one substitution must satisfy to return true and for $\forall$ all the substitutions must satisfy

```
static boolean is isFatherForAll (int n, int father
        , int[]alphabet, int[][]parent){
    for(int x=0;x<alphabet.length;x++){
        for(int i=0;i<parent.length;i++){
            if(!(parent[i][0]==father&&parent[i][1]==x))return false;
        }
    }
    return true;
}
```

the above code is equivalent to $isFatherForAll(X) \equiv \forall(Y)(Parent(X, Y))$, and similarly all quantifiers can be computed the same way in polynomial time.

2.4 Expanding $\mathcal{A}_\mathcal{P}$

The fragment of $\mathcal{A}_\mathcal{P}$ can be expanded to cover more Java constructs and decrease the constraints that already exist to allow for an easier and more convenient way of coding in $\mathcal{A}_\mathcal{P}$, we expanded $\mathcal{A}_\mathcal{P}$ in the following way.

- More data types could be added since all data types can be constructed from *int* and *boolean* in the first place and will be in class P, so we added *String*, *char*, and

double, also other data types and data structures can be added like *short*, *float*, *stack*, *queue*, ...etc.

- The bound of the loops can be polynomial in N, polynomial in Lim, or the minimum of one of them and anything else, this gives more freedom in the bounds of the loops.
- The first parameter of the method called inside the main method can be a polynomial in N or the minimum of it and anything else. That removes the restriction that only N is the depth bound of calls and allows for more choices in the bound as long as it is overall bounded by a polynomial in N

2.5 Coding flow in $\mathcal{A}_\mathcal{P}$

The fragment $\mathcal{A}_\mathcal{P}$ is a high-level programming language that is similar to most programming languages. However, there are two steps to learn coding in any programming language, learning the syntax and learning how to think in that language. The syntax is similar to Java but thinking in $\mathcal{A}_\mathcal{P}$ is determined by its rules, so it is important to see how an $\mathcal{A}_\mathcal{P}$ programmer would think.

Like any other imperative programming language $\mathcal{A}_\mathcal{P}$ can solve all problems in P. But in some cases, it requires a special way of approaching the problem. It is always possible to solve any problem by applying the first-order reduction of the problem to the circuit value problem and then solving it as per the proof. But this is impractically a lot of work, especially the more complicated problems. Hence the target of any programming language is to make solving the problem easier without going through the theoretical way of solving it.

In $\mathcal{A}_\mathcal{P}$ all loops must be pounded by a polynomial in N, where N is the size of the input, which might cause some trouble while coding. For example, if the programmer wanted to loop over an array of size M, it will not be possible unless we can find some polynomial in N that is always exactly equal to M. To solve this problem we allowed for

minimization between that polynomial in N and any other expression. Now it is only required to find a polynomial in N that is always greater than M. This will be one of the programmer's tasks.

For example, if the size of the array is always constant regardless of the input then it is possible to use that exact value while looping because any constant is also a polynomial. If the array size was variable, which is given in the input then we know that N is bigger than the size of the array since N is the size of the entire input, so if we want to loop the square of the size of the array, we can minimize that value with N^2 in the code, and we know that N^2 will always be larger so the first value will be used.

The same approach is done in calling inside the main method. It is allowed to pass the first parameter (depth limiter) as a polynomial in N or a minimization with it. That allows the programmer to compute the appropriate depth required in the recursive calls, not only depending on N.

Chapter 3

Implementation of $\mathcal{A}_\mathcal{P}$

In this chapter, we will go through the implementation of $\mathcal{A}_\mathcal{P}$ and the structure of the code. It will show each method and its usage, and the way the $\mathcal{A}_\mathcal{P}$ code is stored to be compiled and executed.

3.1 Constrains

There are a few constraints on the code that will be written in $\mathcal{A}_\mathcal{P}$:

1. There must be a static variable N. The code will run in time polynomial in that variable N.
2. The loops must be bounded by a polynomial in N.
3. The first variable of all method calls inside the main method must be a polynomial in N.
4. All methods must have their first parameter a variable called Lim.
5. The loops inside a method can be bounded by a polynomial in Lim because Lim is always less than or equal to N.

6. The first variable in method calls inside all methods except *main* must be *Lim* divided by an integer constant greater than 1.

7. If there is only one method call inside the body of the method then the first variable can be *Lim* minus an integer constant greater than 0.

8. Method calls are not allowed inside loops

9. Loop counters must be only incremented or decremented in the direction that guarantees termination

10. We can not use a continue statement before the counter increment of a loop

11. Scanner can be only called *sc* (Scanner is an object in Java that is used to take the input).

3.2 Structure

The code structuring is one important step in the implementation since it organizes the code in a way that is easy to trace, The code consists of the following. *In the sequel the term "limiter variables" is used to refer to Lim and N*

- An Expression class. It is used to store all expression like conditions of the while, for, and if, or the right-hand side of the assignments or any other expression

```
static class Expression {
    String expression;
    public void checkVariableChange(int line);
    public boolean isLessThan();
    public int countMethodCalls()
    public void checkMethodCalls(boolean isMainMethod
        , boolean hasSingleMethodCall, int line)
```

```
        public static int getExpressionMagnitude(String expression
            , String limiter)
    }
```

expression: It is the string containing the expression

Method checkVariableChange: It is used to check if this expression contains any change of variables values like"x++"

Method isLessThan: In case the expression is a boolean expression it is used to check if the first variable in this condition is bounded by lessThan or moreThan, it is used to check if the loop variables should be incremented or decremented

Method countMethodCalls: Counts the number of method calls inside the expression

Method checkMethodCalls: Checks if the method calls inside the expression satisfy the constraints (e.g. checking if the first variable in the method call is Lim divided by constant)

Method getExpressionMagnitude: Checks if the expression is a constant or a polynomial in some variable or is something else, it is used to check if the loops are only polynomial

- An abstract class called Instruction which is inherited in four classes {$ForLoop$, $WhileLoop$, $IfCondition$, $Statement$}

```
    static abstract class Instruction {
```

```
        int line;

        abstract public String toString(int depth);

        abstract public void check(boolean isMainMethod
            , boolean hasSingleMethodCall);

        abstract public void variableChanged(String variableName
            , boolean increase);

        abstract public int countMethodCalls();
    }
```

line: represents the line in which the instruction starts, and is used to indicate the position of the error if it exists

Method toString: is used to print the instruction and is called recursively to print all it's sub-instructions, it is only used for tracing

Method check: is used to check if the code has any error, and is called recursively to all the sub-instructions

Method variableChange : is used to check if some variable was changed inside this instruction or in one of its sub-instructions, it takes a boolean to indicate if this variable can be incremented or decremented, it is mainly used to check any change in loop counters along with checking if the limiter variables were changed

- The ForLoop class

```
static class ForLoop extends Instruction {
    ArrayList<Instruction> inside;
    String counterName;
    int counterValue;
    Expression condition;
    long inc;
}
```

inside: an ArrayList of instructions that are inside the body of the for it may contain other ForLoops or any other instructions

counterName: it shows the name of the counter and is used with VariableChange() method to check if the counter was changed inside the loop body

counterValue: it shows the initial value of the counter

condition: it is an Expression object, it is used to store the condition of the for loop

inc: it is the value of increment of the loop counter

- The WhileLoop class

```
static class WhileLoop extends Instruction {
    ArrayList<Instruction> inside;
    Expression condition;
    String counterName;
}
```

All WhileLoop variables are similar to that of the ForLoop class, the counterName is assumed to be the first variable present in the condition of the while loop

- The IfCondition class

```
static class IfCondition extends Instruction {
    ArrayList<Instruction> thenPart;
    ArrayList<Instruction> elsePart;
    Expression condition;
}
```

thenPart: an ArrayList of Instructions that are inside the then part of the if condition

elsePart: an ArrayList of Instructions that are inside the else part of the if condition, if there is "else if" it will be stored as one IfCondition instruction inside the elsePart it is equivalent to nested if

condition: it stores the condition of the IfCondition

- The Statement class

```
static class Statement extends Instruction {
    String leftHandSide;
    Expression rightHandSide;
}
```

leftHandSide: it stores the left-hand side of the Statement

rightHandSide: it stores the right-hand side of the Statement

The statement class is used to store a variety of things the way in which they are stored will be shown later

- A Method class

```
static class Method {
    int line;
    String name;
    String returnType;
    ArrayList<String> paramNames = new ArrayList<>();
    ArrayList<String> paramTypes = new ArrayList<>();
    ArrayList<Instruction> instructions;

    public String toString();
    public void check(boolean skipFirst);
    public int countMethodCalls();
}
```

line: The line in which the method starts

name: The name of the method start

returnType: The return type of the method

paramNames, paramTypes: The parameters of the method

instructions: An ArrayList of instructions that are inside the method

Method toString: Used to prints the method

Method check: Used to check if the method and all its sub-instructions are valid

Method countMethodCalls: Counts the method calls inside the method, it is used to determine if it's allowed to pass "Lim-constant" in the first variable in the method call

- The Code class, it is the one used to store the entire code, everything is stored as Method or instruction inside of it

```
public class Code {
    int line;
    ArrayList<Method> methods;
    ArrayList<Statement> imports;
    ArrayList<Statement> staticVariables;

    public String toString();
    public void check();

}
```

line: The line in which the class of the code starts

methods: An ArrayList storing all the methods in the code including main method

imports: An ArrayList storing all the imports in the code

staticVariables: An ArrayList storing all the static variables in the code

Method toString used to print the entire code

Method check used to check if the entire code is valid

3.3 Parsing and Storing

Parsing and storing the code is done in the following way

1. All the unneeded white spaces are removed. A group of spaces will be either removed or replaced by one space it will be totally removed if one of its two ends is not a letter nor a number
2. All the new line characters "\n" will be removed and treated as space characters, while removing we will keep a counter to tell us in which line of code are we
3. All comments are completely removed from the code by checking if there are double "\"
4. All strings and characters will be removed except the quote and double quote (e.g. "akaljsndjns" => "", 'a' => '')
5. Methods, for loops, while loops, and if conditions will be stored normally in their corresponding class

6. Everything else is stored as a Statement, consider the Statment as a pair **(leftHandSide, rightHandSide)** so they will be stored as folows

   - Initialization will be stored as **(variableType+" "+variableName, null)** if there was no value or **(variableType+" "+variableName, variableValue)** if the value was given
   - Assignment will be stored as **(variableName, variableValue)**
   - return statement will be stored as **("return", null)** if it was a void method or **("return", someExpression)** otherwise
   - break and continue will be stored as **("break"/"continue", null)** if it was direct or **("break"/"continue", label)** if it was indirect
   - Method calls will be stored as a plain string in the left hand side **(theMethodCall, null)**
   - increment and decrement will be stored as **(variableName+"++", null)** or **(variableName+"−", null)**
   - The following operations "+=", "-=", "*=", "/=",..... will be stored as **(variableName+"+", theRightHandSideValue)**, the "+" can be replaced by any other operation
   - System.out.print() and System.out.println() will be stored as **("System.out.print", whatWillBePrinted)** or **("System.out.println", whatWillBePrinted)**

Take for example the following code.

```
import java.util.Scanner;

public class Program{
    static int N = 1;
    public static final void main(String[]args) throws Exception {
        N = System.in.available();
```

```
        String s="djnsdn,dsdsd";
        char c='p';
        int x=5;
        for
        (int i=0;i<N;i++)
            x++;

        x-=10;

    }

    static Scanner sc=new Scanner(System.in);
}
```

This will be parsed to the following.

```
import java.util.Scanner;public class Program{static int N=1;public static final
void main(String[]args)throws Exception{N=System.in.available();String s="";
char c='';int x=5;for(int i=0;i<N;i++)x++;x-=10;}static Scanner sc=new Scanner
(System.in);}
```

Note there are no new lines in the above parsing the entire code is one tall line, it was split into multiple parts to fit in the paper.

Now we can see that all unnecessary spaces were removed also the content inside the String and char variables were also removed, and only the important data is left.

Now when we pass this code to be stored it will be stored as follows.

```
1    java.util.Scanner null


4    int N 1
17   Scanner sc new Scanner(System.in)


5    void main (String[] args){
6        N System.in.available()
7        String s ""
8        char c ''
9        int x 5
10       for(i=0;i<N;1){
11           x++ null
         }
13       x- 10
     }
```

We can see in the above code that each important line has a number behind it, this number indicates the number of the line in which this instruction starts in the original code. We can see that these numbers are not sorted due to the rearranging of the code, for example, we can see that the scanner was moved to the top above the *main* method.

We can also see that all statements are stored as 2 parts right-hand side and left-hand side, for example, *int* x=*5* is stored as left-hand side=*int* x and right-hand side=*5*. another example is $x{+}{+}$ it is stored as left-hand side=$x{+}{+}$ and right-hand side=*null*.

In the for loop we can see that the increment part is removed and only the value of the increment is left, we can also see that brackets were added in the for loop to indicate that $x{+}{+}$ is actually inside the for loop.

Chapter 4

Capturing NP

In this chapter, we describe another fragment of java which is $\mathcal{A}_{\mathcal{NP}}$, and it captures the complexity class NP we will go through the fragment, the proof that it captures NP, and implementation.

4.1 Fragment for NP ($\mathcal{A}_{\mathcal{NP}}$)

$\mathcal{A}_{\mathcal{NP}}$ is the language capturing the NP class, it is built upon the language $\mathcal{A}_{\mathcal{P}}$, and it is built on the formal definition of an NP problem, Formally, a problem is classified as an NP problem if, given a proposed solution, it can be verified as correct or incorrect in polynomial time, and a correct solution for an NP problem can be found by iterating over all possible permutations of the solution and check if one of them was accepted by the verifier algorithm, based on this here is a description for $\mathcal{A}_{\mathcal{NP}}$.

- The main method of $\mathcal{A}_{\mathcal{NP}}$ consists of multiple nested loops in the beginning that iterate exponentially many times.
- Inside the outer loops there exist an $\mathcal{A}_{\mathcal{P}}$ code.

- No information is kept between iterations other than the counters because in the non-deterministic Turing machine, all the verification is done in parallel, and we are only simulating the same effect since non-deterministic machines are impossible.
- All other methods that are not main method are $\mathcal{A}_{\mathcal{P}}$ methods.

4.2 Proof that $\mathcal{A}_{\mathcal{NP}}$=$NP$

There are three steps to prove that $\mathcal{A}_{\mathcal{NP}}$=$NP$

- Prove that $\mathcal{A}_{\mathcal{NP}} \subseteq NP$.
- Solve the clique problem with $\mathcal{A}_{\mathcal{NP}}$ which is an NP-complete problem[1]
- Proof that $\mathcal{A}_{\mathcal{NP}}$ is closed under first-order reduction

1. Proving that $\mathcal{A}_{\mathcal{NP}} \subseteq NP$.

   As per the description of $\mathcal{A}_{\mathcal{NP}}$ it is built on the definition of an NP problem. Any NP problem can be solved by iterating over all possible solutions and verifying if the solution is correct in polynomial time. Since the language $\mathcal{A}_{\mathcal{NP}}$ does exactly that, then all the problems solved with $\mathcal{A}_{\mathcal{NP}}$ are by definition NP problems.

2. Solving the clique problem with $\mathcal{A}_{\mathcal{NP}}$. The clique problem is described as follows given a graph and an integer k check if there exist k nodes in the graph such that there is an edge between each two pairs of nodes in those k nodes.

```
import java.util.Scanner;

public class clique{
    static  int N = 1;
```

```
public static void main(String[]args) throws Exception {
    N=System.in.available();
    final int n=sc.nextInt();
    final int m=sc.nextInt();
    final int k=sc.nextInt();
    final boolean[][] adj=getGraph(N,n,m);

    for(int i=0;i<Math.min(Math.pow(2,N),Math.pow(2,n));i++) {
        boolean found=true;
        if(Integer.bitCount(i)!=k)found=false;
        for(int j=0;j<Math.min(N,n);j++){
            for(int kk=1;kk<Math.min(N,n-j);kk++){
                if(((1<<j)&i)==0)continue;
                if(((1<<(kk+j))&i)==0)continue;
                if(!adj[j][kk+j]) found=false;
            }
        }
        if(found){
            System.out.println("Yes");
            for(int j=0;j<Math.min(N,n);j++){
                    if(((1<<j)&i)!=0) System.out.print(j+1+" ");;

            }
            return;
        }
        if(i==Math.pow(2,n)-1) System.out.println("No");
    }
}
```

```
    static boolean[][] getGraph(int Lim,int n,int m) {
        boolean[][] ret = new boolean[n][n];
        for (int i = 0; i < Math.min(Lim, m); i++) {
            int x = sc.nextInt()-1;
            int y = sc.nextInt()-1;
            ret[x][y] = true;
            ret[y][x] = true;
        }
        return ret;
    }
    static Scanner sc=new Scanner(System.in);
}
```

3. Proving that $\mathcal{A}_{\mathcal{NP}}$ is closed under first-order reduction

   Since the language $\mathcal{A}_{\mathcal{NP}}$ is built upon the language $\mathcal{A}_{\mathcal{P}}$ and we can write any $\mathcal{A}_{\mathcal{P}}$ code with $\mathcal{A}_{\mathcal{NP}}$, then $\mathcal{A}_{\mathcal{P}} \subseteq \mathcal{A}_{\mathcal{NP}}$, and since we have already proved that $\mathcal{A}_{\mathcal{P}}$ is closed under first order reduction then similarly $\mathcal{A}_{\mathcal{NP}}$ is closed under first order reduction

4.3 Implementation of $\mathcal{A}_{\mathcal{NP}}$

The implementation of $\mathcal{A}_{\mathcal{NP}}$ is done as follows.

1. Unlike the implementation of $\mathcal{A}_{\mathcal{P}}$ there is no specific code structure for implementing $\mathcal{A}_{\mathcal{NP}}$, instead we will use the already existing structure of $\mathcal{A}_{\mathcal{P}}$ and use it to fit $\mathcal{A}_{\mathcal{NP}}$ in it.

2. The code is split into two parts the exponential loops and the rest of the code.

3. The exponential loops are nested loops that are allowed to run exponentially many times, and there are one of them in the code inside the main method.

4. The inner body of the exponential loops is extracted and added outside the loops, and then the loops are removed from the code.

5. The inner loop is then checked to see if they verify the requirements and that non of them iterates more than exponentially many times.

6. The rest of the code is then checked via the polynomial compiler implemented for $\mathcal{A}_{\mathcal{P}}$, and they will be treated as normal $\mathcal{A}_{\mathcal{P}}$ code.

Take for example the following code.

```
public class test{
    static  int N = 1;
    public static void main(String[]args) throws Exception {
        N=System.in.available();
        for(int j=0;j<-N*(1+N);j++) {
            for(int i=1;i<N;i++){
                for(int k=0;k<5-9*2/3;k++){
                    for(int l=0;l<Math.pow(k,j);l++){
                        int z=1;
                        z++;
                    }
                }
            }
        }
    }
```

```
    static boolean check(int Lim){
        if(Lim<=0)return true;
        return check(Lim/2)&(Lim%2==1);
    }
}
```

It will be stored as follows.

```
3    int N 1

4    void main (String[] args){
5        N System.in.available()
10       int z 1
11       z++ null
     }

17   boolean check (int Lim){
18       if(Lim<=0){
18           return true
         }
19       return check(Lim/2)&(Lim%2==1)
     }
```

```
*****************************************
```

```
6    for(j=0;j<-N*(1+N);1){
7        for(i=1;i<N;1){
```

```
8             for(k=0;k<5-9*2/3;1){
9                 for(l=0;l<Math.pow(k,j);1){
                  }
              }
          }
      }
```

As we can see the first is the code without the exponential loops. We can also see that the inner body of loops was kept in the *main* method. After that, that part is treated as a normal $\mathcal{A}_{\mathcal{P}}$ code.

In the second part, we can see the loops alone without anything else, this part is used to check the validity of these loops, and whether they specify the requirements or not, this part is stored as $\mathcal{A}_{\mathcal{P}}$ code but are not checked in the $\mathcal{A}_{\mathcal{P}}$ compiler.

Chapter 5

Testing

Testing was done to ensure the correctness of both fragments and to debug the errors that occurred throughout the code. testing in the beginning was done by some of my GUC colleagues they went out and tried different codes trying to stress test all the possible corner cases, and thanks to their efforts a lot of the corner cases were found that helped to improve the code.

later Java unit tests were introduced to allow for more speed in testing and allowing to run the same test multiple times to ensure that non of them fail upon adding new things to the code.

A method was created to compile and run both $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\mathcal{NP}}$ codes that method returns true if the code compiled regardless of the output of the run even if a run-time error occurred. Then an assertion was made to ensure that the output of this function is true, and if we wanted to check that this code doesn't compile in one of the fragments we assert it as false.

all the codes presented in this were added to the Unit tests of the code, here is an example of the CVP problem after writing it in the expanded $\mathcal{A}_{\mathcal{P}}$

```
import java.util.Scanner;
```

```
public class test4{
    static  int N = 1;
    public static void main(String[]args) throws Exception {
        N = System.in.available();
        int n=sc.nextInt();
        int m=sc.nextInt();

        boolean[]variables=new boolean[n];
        int[]gates=new int[m];
        int[]first=new int[m];
        int[]second=new int[m];



        for(int i=0;i<Math.min(N,n);i++){
            variables[i]=sc.nextInt()==1;
        }
        for(int i=0;i<Math.min(N,m);i++){
            gates[i]=sc.nextInt();
            first[i]=sc.nextInt();
            if(gates[i]!=2)
                second[i]=sc.nextInt();
        }
        System.out.println(CV(N,variables,gates,first,second));
    }
    static boolean CV(int Lim, boolean[] variables, int[] gates
            , int[] firstInputs, int[] secondInputs){
        int n=variables.length;
        int m=gates.length;
```

```
        boolean[] results=new boolean[n+m];
        for(int i=0;i<Math.min(Lim,n);i++){
            results[i]=variables[i];
        }

        for(int i=0;i<Math.min(Lim,m);i++){
            if(gates[i]==0){
                results[i+n]=results[firstInputs[i]]|results[secondInputs[i]];
            }else if(gates[i]==1){
                results[i+n]=results[firstInputs[i]]&results[secondInputs[i]];
            }else{
                results[i+n]=!results[firstInputs[i]];
            }
        }

        return results[n+m-1];
    }
    static Scanner sc=new Scanner(System.in);
}
```

And here are some other examples to check corner cases.

checking for multiple nested double quotes symbols.

```
public class test2{
    static int N;
    public static void main(String[]args) throws Exception {
        N=System.in.available();
```

```
        String a="abcdef\"\"\"\"\\\"gh";
        char c='\"';
        for(int i=0;i<8;i++)
            System.out.println(a.charAt(i));

    }
}
```

Checking for the case where there is a fragment of code inside a string

```
public class test1{
    static  int N = 1;
    public static void main(String[]args) throws Exception {
        N=System.in.available();
        for(int i=0;i<N;i++) {
            String S = "for(int i=0;i<N;i++){int x=0;}";
        }
    }
}
```

Checking for exponential loops.

```
public class test3{
    static  int N = 1;
    public static void main(String[]args) throws Exception {
        N=System.in.available();
        for(int j=0;j<-N*(1+N);j++) {
            for(int i=1;i<N;i++){
                for(int k=0;k<5-9*2/3;k++){
                    for(int l=0;l<Math.pow(k,j);l++){
                        int z=1;
                        z++;
                    }
                }
            }
        }
    static boolean check(int Lim){
        if(Lim<=0)return true;
        return check(Lim/2)&(Lim%2==1);
    }
}
```

Chapter 6

Conclusion

In this thesis, we have explored the capture of computational complexity classes P and NP using imperative programming languages, specifically fragments of the Java language. We introduced the language $\mathcal{A}$ as the foundation for the two fragments. we defined $\mathcal{A}_{\mathcal{P}}$ for capturing class P, proving its equivalence to P. The expansion and coding flow within AP were discussed, providing insights into its implementation.

Furthermore, we presented the fragment $\mathcal{A}_{\mathcal{NP}}$, which captures class NP, building upon the formal definition of an NP problem. We demonstrated that $\mathcal{A}_{\mathcal{P}}$ is capable of solving NP problems by iterating over all possible solutions and verifying their correctness in polynomial time. The implementation details of $\mathcal{A}_{\mathcal{P}}$ were also discussed, including the splitting of code into exponential loops and the polynomial compiler for $\mathcal{A}_{\mathcal{P}}$ code.

Overall, this research contributes to the advancement of computational complexity theory, by providing two fragments of Java that capture the two complexity class P and NP. and this could help increase the understanding of this topic by providing this practical tools that only allow the outcome to be either P or NP codes and thus forces the user of these tools to start thinking more instinctively in terms of these complexity classes.

Chapter 7

Future Work

The thesis highlights the potential for future research, here are some of them.

One potential direction for future work is to expand the fragments of Java used to capture complexity classes, by adding more Java constructs that do not violate the rules specified by doctor Lotfallah[2], Such as more data structures and data types, like stack, queue, short, ... etc.

Another possible future work is performance optimization to further improve the efficiency of the fragments capturing complexity classes, future research can search into performance optimization techniques. Exploring strategies to enhance runtime performance, memory utilization, and code execution speed within these fragments would be valuable. Techniques such as parallelization, compiler optimizations, or algorithmic improvements tailored to the specific complexity class can be investigated. These optimizations would not only enhance the usability of the captured complexity classes but also unlock their full potential in terms of computational efficiency.

This fragment was based on Java, one possible future work is to try spreading similar fragments from other programming languages like C++ or Python these will open the door for more choices regarding capturing complexity class, and will allow the users to have more freedom to choose the language they like.

Bibliography

[1] Neil Immerman. Descriptive Complexity. *University of Massachusetts*, 1953.

[2] Wafik Boulos Lotfallah. Descriptive Complexities of Syntactical Fragments of Imperative Languages. *Unpublished manuscript*.