Dynaplex: Inferring Asymptotic Runtime Complexity of Recursive Programs

Didier Ishimwe
University of Nebraska-Lincoln
USA

ThanhVu Nguyen
George Mason University
USA

KimHao Nguyen
University of Nebraska-Lincoln
USA

ABSTRACT

Automated runtime complexity analysis can help developers detect egregious performance issues. Existing runtime complexity analysis are often done for imperative programs using static analyses. In this demo paper, we demonstrate the implementation and usage of Dynaplex, a dynamic analysis tool that computes the asymptotic runtime complexity of recursive programs. Dynaplex infers *recurrence relations* from execution traces and solve them for a closed-form complexity bound. Experimental results show that Dynaplex can infer a wide range of complexity bounds (eg: logarithmic, polynomial, exponential, non-polynomial) with great precision (eg: $O(n^{\log_2 3})$ for karatsuba). A video demonstration of Dynaplex usage is available at https://youtu.be/t7dhwZ7fbVs

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; **• Security and privacy → Software security engineering**.

KEYWORDS

dynamic invariant generation, complexity analysis, recurrence relations

ACM Reference Format:
Didier Ishimwe, ThanhVu Nguyen, and KimHao Nguyen. 2022. Dynaplex: Inferring Asymptotic Runtime Complexity of Recursive Programs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program invariants describe properties that always hold at a program location. Invariants are useful in many programming tasks including verification, documentation, testing, debugging and code generation. One of the benefits of automated invariant discovery is that they help characterize program runtime complexity [14]. Runtime complexity analysis is topic of both theoretical and practical interest. First, hard real-time systems may require some guarantees about their worst-case behaviour. Second, worst-case runtime bounds are can help in early detection of eggregious performance issues. Third, manual runtime complexity analysis can require a lot of mathematical ingenuity from developers.

Given the importance of computing runtime complexity bounds, researchers have studied the problem in different ways. Dynamic analysis aproaches [13, 14] compute nonlinear numerical invariants from which complexity bounds can be inferred. Several static analyses techniques have been developed for resource analysis. Hofmann and Jost [8] predict time and space usage of functional programs by extending their type systems to prove these resource bounds. SPEED [5, 6] uses abstract interpretation to compute non-linear bounds. The NPWCARP [3] uses ranking functions and invariant templates; ICRA [11] uses recurrence-based invariant generation whereas CHORA [2] combines both template-based and recurrence-based techniques to compute runtime complexity bounds.

While existing runtime complexity analysis techniques are useful, they have limitations. In general, static analyses can reason about all program paths soundly, but doing so is often expensive and is only possible for a restricted class of programs. For example, NPWCARP analyzes cost models instead of the original source code. The polynomial invariant discovered by SymInfer [14], a dynamic invariant generator, are too strict to capture the complexity of most programs. Several works focus on analyzing loops therefore there is less emphasis on recursive program as reflected in the Termination Competition [17] where all programs are recursion-free.

Inspired by dynamic invariant generation, in [9] we developed Dynaplex, a new dynamic inference technique and tool for learning recurrence relations (or simply *recurrences*) to capture the asymptotic complexity bounds of *recursive programs*. Briefly, a recurrence defines the complexity to solve a problem in terms of the work to solve its subproblems [4]. Dynaplex learns the relationship between a problem size and its subproblems to compute recurrences. Moreover, Dynaplex solves recurrences using pattern-matching techniques to obtain a closed-form solution that describes the asymptotic complexity.

The Dynaplex approach has several benefits. By using dynamic analysis, Dynaplex is language-agnostic and supports complex programs semantics that might be difficult for static analyses. By considering only program runs over a small number of randomly generated inputs, Dynaplex efficiently learns divide-and-conquer and linear recurrences. By using linear recurrences, Dynaplex can compute simple linear terms (over input size) to describe non-trivial complexity bounds such as logarithmic and exponential. Finally, by using pattern matching techniques optimized for common recurrences, Dynaplex quickly identifies correct worst-case complexity bounds for non-trivial recursive programs.

The envisioned users for Dynaplex include users and developers of perfomance `profiling` tools as well as runtime complexity researchers. Students learning Big-O complexity analysis can also benefit from Dynaplex and its approach. It simplifies the complex manual process involved in asymptotic complexity analysis. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

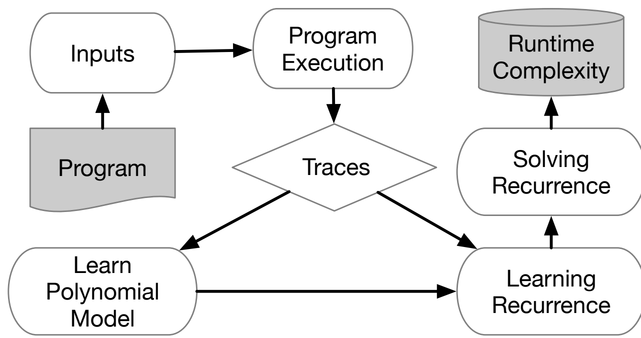

Figure 1: Dynaplex overview

source code and benchmark programs of Dynaplex are publicly available at [10]. The full details of Dynaplex approach are available in the research paper [9]. In this tool paper, we present more in-depth technical details about the design, implementation and usage of Dynaplex.

2 DYNAPLEX

Dynaplex is a command-line tool which computes the recurrence relation and complexity bound of a recursive program. Figure 1 shows the workflow of Dynaplex, which is divided in three phases:

- **trace collection:** Dynaplex relies on the user to instrument the method under analysis to capture program execution traces at runtime.
- **inferring recurrences:** by analyzing the execution traces, Dynaplex computes the recursive terms as well as the polynomial term of a programs recurrence.
- **solving recurrences:** Dynaplex generates linear recurrences as well as divide-and-conquer recurrences which can be solved by pattern matching techniques such as Master Theorem.

Figure 2 shows a running example of Dynaplex computing the complexity of karatsuba programs. This program implements an efficient multiplication algorithm for two n-digits integers using single digit multiplication [?]. Figure 2 shows an instrumented version of karatsuba program to collect the execution traces. It also shows the execution trace tree produced by running karatsuba multiplication on two 5-digit integers. The following subsections describe the three phases in detail using karatsuba program as an example.

2.1 Trace Collection

Instrumentation:

Dynaplex relies on manual instrumentation to collect execution traces. Users manually instrument a function under analysis to collect the depth of recursion, problem size and the number of loop iterations per recursive step. Figure 3 shows an example of bubblesort program with instrumentation added (in red). The trace function added at the function entry tracks the problem size n and depth of recursion id. The variable k was added to track the number of loop iterations. Dynaplex only needs the number of loop iterations in the first recursive call ($id == 0$).

Traces: In the karatsuba example, Figure 2 shows the execution trace tree where each node corresponds to a recursive call, its depth represent the corresponding depth of recursion and its value represents the corresponding problem size. Since manual instrumentation can be tedious, Dynaplex is equipped with a C++ library to help automate the process for C++ programs.

The execution traces are collected by running the instrumented program on randomly generated inputs. Random inputs are sufficient to compute the complexity of several recursive programs. However, users can generate better quality of input using pathological input fuzzers such as Perfuzz [12].

2.2 Inferring Recurrences

A recurrence relation is a recursive description of a function in terms of itself. Most recurrences can be divided into two components: the recursive terms and non-recursive terms. Dynaplex computes both components of a recurrence from the traces collected.

Inferring recursive terms: Dynaplex learns the recursive terms of karatsuba's recurrence by learning the ratio between each node and its child nodes. From the tree in Figure 2, Dynaplex infers that karatsuba function makes 3 recursive calls as each node have three child nodes. Dynaplex computes the first recursive term of karatsuba's recurrence by learning the ratio between each node and its first (rightmost) child node. Dynaplex learns the relation $t_1 = \frac{1}{2}t_0$ between each node t_0 and its first child node t_1 from the data $[(5,3),(3,2),(2,1),(3,2),(2,1),(2,1)]$.

Similarly, Dynaplex learns the ratio between each node and its second (middle) and third (leftmost) child nodes to be $t_1 = \frac{1}{2}t_0$. Dynaplex relies on regression to compute the recursive terms therefore, it requires more traces (i.e. bigger trace trees) than shown in Figure 2. Running karatsuba on 13-digit integers would produce sufficient traces for Dynaplex to infer karatsuba's recurrence. However, the trace tree produced would be too big to fit in Figure 2.

Inferring non-recursive term: the non-recursive term of the recurrence corresponds to the work than outside of recursive calls. For instance, the work than by the merging of results in mergesort program. Dynaplex learns the non-recursive term using polynomial regression between the problem size (n) and the number of loop iterations (k) per recursive step. Using polynomial regression, Dynaplex can also learn non-polynomial terms such as $t = \log n$. For instance, a linear relation between k and t implies a logarithmic relation between k and n (i.e. $k = ct \implies k = c\log n$). The karatsuba program in Figure 2 doesn't have loop; therefore Dynaplex learns that the work done outside the three recursive call is constant thus the non-recursive term is 1.

2.3 Solving Recurrences

Dynaplex combines the recursive and non-recursive terms into one recurrence relation which is then solved using pattern matching apprach from [9]. For instance, using Master Theorem, Dynaplex can map divide-and-conquer recurrences of the form $T(n) = aT(\frac{n}{b}) + f(n)$ to their corresponding complexity bounds. There are more

```
def karatsuba(x, y, id):
    trace(len(str(x)), id)
    if len(str(x)) == 1 or len(str(y)) == 1:
        return x*y
    else:
        n = max(len(str(x)),len(str(y)))
        nby2 = 10**(n / 2)

        a = int(x / nby2)
        b = int(x % nby2)
        c = int(y / nby2)
        d = int(y % nby2)

        ac = karatsuba(a, c, id++)
        bd = karatsuba(b, d, id++)
        ad_bc = karatsuba(a+b, c+d, id++)
        ad_bc = ac - bd

        prod = ac * (nby2**2) + ad_bc * nby2 + bd

        return prod
```

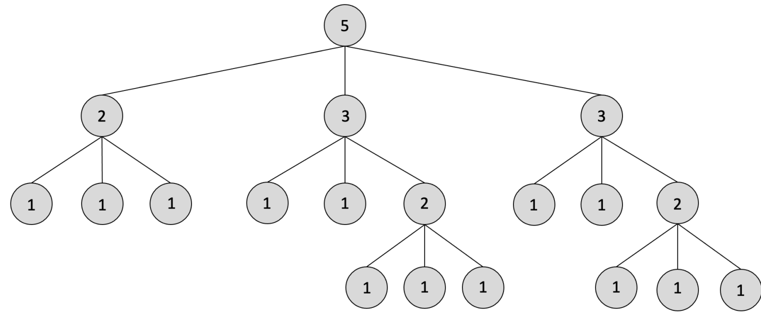

Figure 2: A karatsuba program and its execution traces.

```
def bubblesort(arr, n, id):
    trace(n, id)
    if n==1:
        return arr

    for i in range(n-1):
        if arr[i]>arr[i+1]:
            arr[i], arr[i+1] = arr[i+1], arr[i]
            if id == 0:
                k++
    bubble_sort(arr, n-1, id++)
    return
```

Figure 3: Instrumented bubblesort program

general recurrence solving technique (ex: Akra-Bazzi [1]); however, Dynaplex can solve divide-and-conquer and other linear recurrences instantaneously and soundly using the aforementioned pattern matching.

In our running example, Dynaplex combines the results from the above subsection to form the recurrence of karatsuba program $T(n) = 3T(\frac{n}{2}) + 1$. Dynaplex applies Master theorem to compute the complexity bound $O(n^{\log_2 3})$ for karatsuba. Note that the use of recurrences permit Dynaplex to compute non-polynomial complexity n^r where $r \notin \mathbb{N}$ ($log_2 3$ is irrational).

3 TOOL USAGE

3.1 Design

Dynaplex is implemented in Python and uses polynomial regression from numpy [7] to learn polynomial relations required to infer recurrence relations. Dynaplex is designed to work with any programming language as it performs the analysis on execution traces instead of source code. Dynaplex can be configured by the user to control how the tool works. These configuration settings, which can be changed from the command-line or settings.py file, include: -maxdeg d (generate polynomial models up to degree d); -nlog (generate logarithmic terms for non-recursive term of the recurrence).

Dynaplex can be used as a command line tool for Unix based operating systems that supports Docker (tested on Ubuntu 20.04 and Debian 10.7 with Docker 20.10.7). Users can try Dynaplex by cloning its Github repository https://github.com/unsat/dynaplex and installing dependencies as instructed by the README file in the repository. An easier way to try Dynaplex is by following detailed instructions for obtaining the artifact and running experiments found in [10].

In summary, to get started with Dynaplex, users need to follow three steps:

(1) docker pull unsatx/dynaplex:oopsla21 pull the up-to-date a Docker image of Dynaplex
(2) docker run -it unsatx/dynaplex:oopsla21 run the Docker container to access Dynaplex source code and pre-installed dependencies
(3) run Dynaplex /dynaplex/analyzer.py -trace karatsuba/ analyzer on the traces to analyze. Traces are collected by executing an instrumented program (karatsuba) to analyze.

Dynaplex uses naming convention to identify the type of traces and perform proper analysis. For instance, the karatsuba folder contain multiple files named output-<s>, where <s> is the original problem size, containing traces about depth of recursion and problem size. There is also a file named traces that contains problem size and loop iteration count. Dynaplex uses output-<s> to infer recursive terms, and traces to infer non-recursive term of the recurrence relation.

3.2 Dynaplex Output

Figure 4 shows the result of running Dynaplex on the karatsuba program on an AMD Ryzen 16-core Debian Buster system with 32 GB of RAM. Dynaplex's output is divided into three main parts that reflects its architecture. First, Dynaplex computes the recursive terms of recurrence relation: $T(\frac{n}{2})$ for each recursive call in the

```
# /dynaplex/analyzer.py -trace karatsuba/
Computing the recurrence relation terms
T(n/2)
T(n/2)
T(n/2)
Computing polynomial relation term
Command:  /dynaplex/dig.py -trace karatsuba/traces -maxdeg 5
Polynomial relation:  1
Recurrence relation:
T(n) = T(n/2) +  T(n/2) +  T(n/2) +  (n^0(logn)^0)
Solving the recurrence relation
Complexity is O(n^1.5849625007211563)
Analysis complete in 0.874 seconds
```

Figure 4: Running Dynaplex

karatsuba example. Second, it computes the non-recursive term of the recurrences; 1 (constant) since karatsuba have no loops. Third, it combines the results into a recurrence which is then solved to output the complexity bound of karatsuba.

For recursive programs with non-constant non-recursive terms such as the bubblesort example in Figure 3, Dynaplex computes and outputs polynomial models before and after applying selection heuristics (as described in [9]). We run Dynaplex on traces collected by running bubblesort 100 times on random inputs of size $1-500$ with -maxdeg 2. Dynaplex outputs three models before heuristics: $m_0 = 232.5$, $m_1 = 0.9n - 4.5$, $m_2 = 9.8 * 10^{-6}n^2 + 0.9901n - 4.066$. It only keeps m_1 after applying heuristics. Since Dynaplex computes asymptotic bounds it discards the coefficients of the polynomial model and only keeps the highest order term of the model (n).

4 EVALUATION

We evaluated Dynaplex on a benchmark of 37 programs. Although these programs are small ($\approx$ 100 LOC) classical recursive algorithms (ex: Fibonacci), they contain non-trivial data structures and a wide range of complexity bounds.

Our experimental evaluation showed that Dynaplex can infer correct complexity bounds for 32/37 programs. Dynaplex is able to compute a wide range of complexity including logarithimic, polynomial and non-polynomial bounds. For instance, Dynaplex computes the precise complexity of strassen matrix multiplication $O(n^{\log_2 7})$. This level of precision would not be possible in numerical invariant-based approaches [3, 14] as $\log_2 7$ is an irrational degree. Dynaplex failed to generate correct recurrences for five programs. These programs either make recursive calls on a random fraction of the problem size (ex: quicksort) or guard the recursive call with a condition (ex: heapsort) resulting in inconsistent traces that Dynaplex fails to analyze. Complete evaluation details are given in [9].

Limitations: Dynaplex, or dynamic analysis tools in general, does not guarantee the *soundness* of its results. Dynaplex analysis depends on the quality of traces which can be insufficient as explained in the above paragraph. Dynaplex can be improved by integrating with static analyses to validate its candidate recurrence relations and/or complexity bounds. Using pathological input instead of random inputs can lead to better quality of traces and thus improving Dynaplex's output.

5 CONCLUSION

We present the design, implementation details and usage of Dynaplex, a dynamic analysis tool to analyze the asymptotic runtime complexity of recursive programs. Dynaplex learns recurrence relations describing recursive programs and solves them for a wide range of asymptotic complexity bounds. Dynaplex is the first dynamic analysis technique for complexity analysis, opening a new research area. In future work, we plan to integrate Dynaplex with static analysis to validate its results. The source code of Dynaplex, its benchmark programs, and experimental results are publicly available at [10].

REFERENCES

[1] Mohamad Akra and Louay Bazzi. 1998. On the solution of linear recurrence equations. *Computational Optimization and Applications* 10, 2 (1998), 195–210.
[2] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. 2020. Templates and Recurrences: Better Together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 688–702.
[3] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems* 41, 4 (2019), 1–52.
[4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
[5] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verification*. Springer-Verlag, 51–62.
[6] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages*. ACM, 127–139.
[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernàndez del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2
[8] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. *ACM SIGPLAN Notices* 38, 1 (2003), 185–197.
[9] Didier Ishimwe, KimHao Nguyen, and NGUYEN THANHVU. 2021. Dynaplex: analyzing program complexity using dynamically inferred recurrence relations. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–23.
[10] Didier Ishimwe, Kim Hao Nguyen, and Thanhvu Nguyen. 2021. *Software Artifact for the OOPSLA'21 Paper Titled "Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations"*. https://doi.org/10.5281/zenodo.5421762
[11] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices* 52, 6 (2017), 248–262.
[12] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 254–265.
[13] ThanhVu Nguyen, Timos Antopoulos, Andrew Ruef, and Michael Hicks. 2017. A Counterexample-guided Approach to Finding Numerical Invariants. In *Foundations of Software Engineering*. ACM, 605–615.
[14] ThanhVu Nguyen, Matthew Dwyer, and William Visser. 2017. SymInfer: Inferring Program Invariants using Symbolic States. In *Automated Software Engineering*. IEEE, 804–814.
[15] Ocaml.org. 2021. 99 Problems in OCaml. https://ocaml.org/learn/tutorials/99problems.html, accessed on May 21, 2022.
[16] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conference on Computer and Communications Security*. ACM, 2155–2168.
[17] Termination Competitions. 2021. Complexity Benchmarks. https://termcomp.github.io/, accessed on May 21, 2022.