pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller

pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs *

Clément Aubert[1 [0000−0001−6346−3043]], Thomas Rubiano[2], Neea Rusch[1 [0000−0002−7354−5330]], and Thomas Seiller[2,3 [0000−0001−6313−0898]]

[1] School of Computer and Cyber Sciences, Augusta University
[2] LIPN—UMR 7030 Université Sorbonne Paris Nord
[3] CNRS

Abstract. Complexity analysis offers assurance of program's runtime behavior, but large classes of programs remain unanalyzable by existing automated techniques. The *mwp*-flow analysis sidesteps many difficulties shared by existing approaches, and offers interesting features, such as compositionality, multivariate bounds, and applicability to non-terminating programs. It analyzes resource usage and determines if a program's variables growth rates are no more than polynomially related to their inputs sizes. This sound calculus, however, is computationally expensive to manipulate, and provides no feedback if the program does not have polynomial bounds. Those two defaults were addressed in a previous work, and prepared for the tool we present here: `pymwp`, a static complexity analyzer for `C` programs based on our improved *mwp*-flow analysis.

Keywords: Static Program Analysis · Automatic Complexity Analysis · Program Verification

1 Introduction

Verifying program's resource usage is particularly important for safety-critical applications: if usage exceeds available capacity, program will fail at runtime. Although automatic complexity analysis is an active research area, no mainstream tools exist for this task. Prior attempts have been successful at e.g., analyzing various programming languages and obtaining tight bounds [8,9,10], but are limited in scalability and often lack compositionality [6]. Implicit Computational Complexity (ICC) [7] aims at finding syntactic criteria to guarantee program's runtime behavior. Since ICC systems offer some of the properties missed by other analysis techniques, it is conjectured that it could bridge the gap in achieving realistic complexity analysis. This prompted a series of work [2,5] that culminates with the tool we present in this paper. Our tool is one of the first ICC-inspired applications and the first mechanization of the specific technique it implements.

* This research is supported by the Transatlantic Research Partnership. Rubiano and Seiller are supported by the Île-de-France through the DIM RFSI project "CoHOp".

It gives early insight of the advancements ICC can provide in automatic complexity analysis and program verification.

The contribution we present in this paper, `pymwp` [4], is a static analyzer for `C` programs that computes sound worst-case complexity bounds for final values of input variables. It provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space so that if it terminates, it will do so in polynomial time. It offers several useful features not commonly found in alternative complexity analyzers: applicability to non-terminating programs—e.g., iteration bounds are abstracted—, compositionality, multivariate result, and full automation, because it requires no manual configuration or annotations. These features produce fast analysis, detailed feedback, and makes `pymwp` suitable for integration in larger compilation toolchains.

In this paper we demonstrate `pymwp` from three perspectives. We start with functionality, in Sect. 2, with brief theoretical foundations and system design. This section gives sufficient background to the theoretical framework to understand how the tool operates, computes results, and how to interpret those results. We also highlight selected theoretical adjustments and system design approaches that were essential to obtaining a practical application. The documentation, e.g., at `https://statycc.github.io/pymwp/relation/` further explains and exemplifies the modules presented in this section.

Next, we present a specific implementation challenge related to evaluation of the analysis result. This is relevant because the theory underlying `pymwp` does not address this potentially exponential-time problem, however it is necessary for implementing the analysis. In Sect. 3 we present the algorithm we developed for `pymwp` to solve this problem efficiently.

Lastly, we discuss user interaction. `pymwp` is built to support multiple use cases. While standard user interaction occurs over command-line interface, the tool can easily be reused and integrated with other systems or pipelined, as we explain in Sect. 4. We envision the developments presented in `pymwp` can lead to future improvements in static analysis tools that should be usable in interactive environments (e.g., IDEs) with nearly immediate feedback to the programmer.

2 Overview of `pymwp`

2.1 Foundations of the *mwp*-flow analysis – Briefly

The *mwp-flow analysis* [11] certifies polynomial bounds on the size of the values manipulated by an imperative program. It computes the polynomial bound—if it exists—by computing for each variable a vector tracking how it depends on other variables. The vector values are determined by applying the rules of the calculus to the commands of the program. A program is assigned a matrix collecting those vectors. While this does not ensure termination, it provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space, and as a consequence that if it terminates, it will do so in polynomial time in the size of its inputs.

Flows characterize controls from one variable to another. In increasing growth rate, they can be of type 0—the absence of any dependency—*m*aximum, *w*eak polynomial and *p*olynomial. They form a semi-ring [3, A.1 and A.2], and we use $+$ to denote max. However, the derivation may fail—some programs may not be assigned a matrix—if at least one of the variables used in the body of a loop depends "too strongly" on another, making it impossible to ensure polynomial bounds. In its first declension, the rules of the calculus were non-deterministic: multiple matrices could be assigned to the same program, to maximize the opportunities of finding bounds if they existed. The derivation stops if no bounds could be inferred, leaving the program only partially analyzed.

We modified those two latter aspects of the theory [2]: having multiple matrices was space- and time-consuming and stopping the derivation as soon as no bound could be found was depriving the programmer from precious feedback. As a result, the upgraded analysis now always provides a single matrix capturing all the possible derivations as different *choices* that must be made. This required to design the additional mechanism that decides whenever choices not leading to an ∞ flow (that represents failure) exist. We return to the example below—that uses the dummy condition `while(0)` but could have used an arbitrarily complex condition—throughout this paper and in the demonstration.

Example 1. Analysis assigns to program `foo` the *mwp*-matrix on right.

```
int foo(int x, int y){
 while(0){x=y+y;}
}
```

$$\begin{array}{c} \\ \texttt{x} \\ \texttt{y} \end{array} \begin{array}{c} \begin{array}{cc} \texttt{x} & \texttt{y} \end{array} \\ \begin{pmatrix} m + \infty\delta(0,0) + \infty\delta(1,0) & 0 \\ \infty\delta(0,0) + \infty\delta(1,0) + w\delta(2,0) & m \end{pmatrix} \end{array}$$

The matrix captures the dependencies between variables `x` and `y`, from source variable (row) to target variable (column). The complex coefficients in the `x` column express that based on the 0^{th} choice, one will get either ∞ and ∞ (if the 0^{th} or 1^{st} option is picked) or m and w (for the last option). It is easy, here, to see that the only option that yields non-∞ coefficients is the last option, $(2, 0)$.

2.2 Concretely implementing the abstract analysis

The `pymwp` tool takes as input a path to a `C` program, and returns the corresponding matrix and the valid derivation choices if the program passes the analysis, an indication of failure and, if requested, the corresponding matrix if not. We decided to implement the tool on a subset of `C` programming language,[4] because it naturally maps to the syntax of the analysis, but the technique could be applied similarly to any imperative language. The name of the tool alludes to its implementation language, Python, selected because of its flexibility and use in previous related implementations [1,12,13].

Structures of representation We placed considerable attention in the tool design to choosing suitable data structures. The internalization of the choices

[4] List of supported features: `https://statycc.github.io/pymwp/features/`

pushes inside the matrices the non-determinism and introduce the need to decide if a series of choices leading to non-∞ coefficients exist. In this section we introduce selected representations and their roles in realizing the analysis.

In `pymwp`, the *mwp*-matrix is represented as a **relation**, whose properties are the input variables of the program under analysis, and a matrix collecting variable dependencies. We represent the matrix using native lists, with defined basic operations e.g., sum, product, resize and fixpoint. We omit use of robust matrix libraries to keep the tool dependencies as light as possible. Note that since the analysis can produce dense matrices,[5] it is not possible to use algorithms optimized for sparse matrices.

For each variable pair, the matrix contains a **polynomial**, an ordered list of ordered monomials. A **monomial** is a pair containing a coefficient value and a list of deltas. Each delta captures a choice of derivation rules that internalizes the nondeterminism. A **delta** is a pair (i, j) where i is the value and j is the index in the domain (or, command at which a choice was made). To ease the analysis, the deltas are sorted— $\delta(i, j)$ is smaller than $\delta(m, n)$ iff either $j < n$ or $(j = n)$ and $(i < m)$— and no two deltas can have the same index.

Example 2. Consider the dependency flow $\infty\delta(0,0) + \infty\delta(1,0) + w\delta(2,0)$, introduced in Example 1. In `pymwp`, its representation is

```
Polynomial(Monomial("i",(0,0)), Monomial("i",(1,0)),
    ↪ Monomial("w",(2,0)))
```

where coefficients `i` and `w` are the ∞- and w-flows, respectively, and the tuples are the deltas. To represent the complete *mwp*-matrix from the example, we would create a relation with 2 variables and a matrix of 4 polynomials.

These design decisions allow us to capture *mwp*-matrices in a singular— although complex—relation. Performing the analysis then becomes a matter of iteratively mapping commands to vectors, collecting those vectors in matrices, then composing the relations.

Workflow We describe next, at a high level, the general procedure of performing the *mwp* analysis. Recall, the input is a path to a C file. The file may contain multiple functions. Each function is treated as a program under analysis, thus we refer to "program" in the remainder of this description.

1. Parse the input file to obtain an abstract syntax tree (AST).
2. For each program in the AST:
   (a) Create an initial relation, R, whose matrix is an identity matrix.
   (b) Sequentially for each statement in program body:
      i. Recursively apply derivation rules to obtain R_i.
      ii. Compose the R_i with previous relation: $R = R \circ R_i$.
      iii. If no valid choice remains,[6] terminate analysis.

[5] For example, `https://statycc.github.io/pymwp/demo/#other_dense.c`.
[6] We omit the details here; see [2, Section 4.4] on how this determination is made.

(c) Evaluate matrix to find valid derivation choices.
(d) Append to result: (relation R, valid choices, success flag). The success flag is true when a polynomial bound can be derived and false otherwise.

3. Return result.

3 Efficiently evaluating the matrix

The final step of the analysis is the evaluation, which finds a series of choices not leading to ∞-coefficients. While only one matrix needs to be searched, the task is challenging because polynomials can represent arbitrarily complex decision surfaces. A naive strategy is to iterate all choices, observe the matrix obtained for each series of choices, and retain those that yield ∞-free matrices.

Example 3. A naive evaluation of Example 1's matrix enumerates all choices, and observe that only one choice, (2,0), produces a matrix without ∞ coefficients:

$$\begin{pmatrix} m+\infty\delta(0,0)+\infty\delta(1,0) & 0 \\ \infty\delta(0,0)+\infty\delta(1,0)+w\delta(2,0) & m \end{pmatrix} \Rightarrow \quad \begin{array}{lccc} \text{choice:} & (0,0) & (1,0) & (2,0) \\ & \begin{pmatrix} \infty & 0 \\ \infty & m \end{pmatrix} & \begin{pmatrix} \infty & 0 \\ \infty & m \end{pmatrix} & \begin{pmatrix} m & 0 \\ w & m \end{pmatrix} \end{array}$$

Unfortunately, this approach is exponential. For a matrix, whose size depends on the number of variables V, and an index i that captures the number of choices introduced during analysis, the complexity is $V^2 \times 3^i$. Since variable pairs are represented in the matrix as polynomials containing deltas, the evaluation of both terms is dependent on i. For larger programs, that introduce many choices, the i increases rapidly, and makes exhaustive search computationally prohibitive. A more sophisticated solution was necessary to achieve efficient analysis.

Efficient evaluation Our efficient evaluation starts by constructing a set S of all the δ values attached to an ∞ coefficient present in the matrix. The inputs to the procedure are S, i, and allowed choices e.g., $(0, 1, 2)$ in our case.

Step 1. Simplify S in two ways, iteratively until convergence: replace elements that can be represented by a single shorter sequence, then remove supersets.

Step 2. We now need to negate the remaining elements of S, because they represent the choices that yield ∞ coefficients, and the desired output is a representation of valid choices. We proceed by initially considering all choices as valid, then eliminating those that lead to failure.
(a) Compute the cross product of the remaining elements in S.
(b) Create a *choice vector* of size i, whose elements represent the allowed choices.
(c) Eliminate those choices that lead to infinity.
(d) Discard invalid and redundant choice vectors.

The result is a disjunction of the remaining choice vectors.

Example 4. An example of an efficient evaluation could be:

```
S = ((0,0),(2,1),(1,2)),((1,0),(2,1)),((0,0))          (Initial δ-set)
                    S = ((1,0),(2,1),((0,0))           (Post-simplification)
          (0,0),(1,0) => [[2],[0,1,2],[0,1,2]]         (Choice vector 1)
          (0,0),(2,1) => [[1,2],[0,1],[0,1,2]]         (Choice vector 2)
[[[2],[0,1,2],[0,1,2]], [[1,2],[0,1],[0,1,2]]]         (Result)
```

To apply the result, we select a choice vector, then choose one value at each vector index. This yields a bounded derivation result. The result captures how e.g., sequences of choices $(2,0),(0,1),(1,2)$ and $(1,0),(1,1),(2,2)$ are valid. However, any choice containing $(0,0)$ is always invalid because it is not allowed by the result. Since this is a positional representation, it can be compacted further by omitting the index, e.g., sequence $(2,0),(0,1),(1,2)$ equal to [201].

Note that unlike the naive approach, this evaluation is independent of the number of variables or the maximum value of i. Its efficiency is only concerned with the longest unique sequence of derivation choices that leads to infinity. In practice these sequences are short after applying the described simplifications. This makes the remaining steps computationally trivial and evaluation result can be obtained nearly instantly. It also provides a compact representation of valid choices, even in cases where the representation is arbitrarily complex.

4 User Interaction

There are multiple ways to use and interact with `pymwp`. It can be used as a standalone command-line tool, or as imported Python modules. It can be integrated into other services as a Python package, as we show with the `pymwp` online demo, which is a web application with `pymwp` as a package dependency. `pymwp` does not modify the input program—it is read-only—so it can be run in parallel or independent of other processes. Therefore, it could be integrated into more sophisticated compilation toolchains. The development version is available as open-source software [4], but the easiest installation is through the Python Package Index (PyPI): `pip install pymwp`. The default interaction command is

```
pymwp /path/to/file.c [args]
```

where the first positional argument, path to a `C` file, is required. Optional arguments can be added in place of `[args]`. The current list of supported arguments is defined in `pymwp --help`. By default, `pymwp` displays a log of debugging information and analysis result on the screen and saves the result to a file. These default behaviors are customizable by specifying optional arguments.

Example 5. Analysis of Example 1 program with `pymwp`.[7] Observe that the obtained matrix and available choices match with the original example.

[7] `https://statycc.github.io/pymwp/demo/#basics_while_2.c`

```
$ pymwp basics/while_2.c
...
MATRIX
------------------------------------------------------------------------
x  |  +m+i.delta(0,0)+i.delta(1,0)  +o
y  |  +i.delta(0,0)+i.delta(1,0)+w.delta(2,0)  +m
------------------------------------------------------------------------
[12:46:00] INFO (analysis): CHOICES: [[[2]]]
...
```

References

1. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: LQICM On C Toy Parser (3 2021), https://github.com/statycc/LQICM_On_C_Toy_Parser
2. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: mwp-analysis improvement and implementation: Realizing implicit computational complexity. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics, vol. 228, pp. 26:1–26:23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.26
3. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: mwp-analysis improvement and implementation: Realizing implicit computational complexity (Mar 2022), https://hal.archives-ouvertes.fr/hal-03596285, preliminary technical report
4. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: pymwp: MWP analysis in Python (10 2022), https://github.com/statycc/pymwp/
5. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: Realizing Implicit Computational Complexity (Mar 2022), https://hal.archives-ouvertes.fr/hal-03603510, presented at the 28th International Conference on Types for Proofs and Programs (Recording)
6. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 467–478. Association for Computing Machinery (2015). https://doi.org/10.1145/2737924.2737955
7. Dal Lago, U.: A short introduction to implicit computational complexity. In: Bezhanishvili, N., Goranko, V. (eds.) ESSLLI. Lecture Notes in Computer Science, vol. 7388, pp. 89–109. Springer (2011). https://doi.org/10.1007/978-3-642-31485-8_3
8. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, Carstenand Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with aprove. Journal of Automated Reasoning **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
9. Hainry, E., Jeandel, E., Péchoux, R., Zeyen, O.: Complexityparser: An automatic tool for certifying poly-time complexity of Java programs. In: Cerone, A., Ölveczky, P.C. (eds.) Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12819, pp. 357–365. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_20
10. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012. LNCS, vol. 7358, pp. 781–786. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_64
11. Jones, N.D., Kristiansen, L.: A flow calculus of *mwp*-bounds for complexity analysis. ACM Transactions on Computational Logic **10**(4), 28:1–28:41 (2009). https://doi.org/10.1145/1555746.1555752
12. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk detection. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10482. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_7

13. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk motion by peeling with statement composition. In: Bonfante, G., Moser, G. (eds.) Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017. Electronic Proceedings in Theoretical Computer Science, vol. 248, pp. 47–59 (2017). https://doi.org/10.4204/EPTCS.248.9, http://arxiv.org/abs/1704.05169

A Detailed demonstration of pymwp

This demo installs pymwp on the local system, then uses it to analyze C programs. We assume Unix-like system with gcc and wget installed. Lines starting with $ are commands to run in a terminal, backslash \ is a line wrap for long commands, and other lines are output. Long or irrelevant output is omitted using ....

A.1 Setup

Installation First check minimum system requirements. The Python version must be 3.7 or higher, and pip must match Python version. If the output indicates otherwise, update the system before proceeding.

```
$ python3 --version && pip --version
Python 3.10.5
pip 22.2.2 from ../site-packages/pip (python 3.10)
```

Next, install the specified version of pymwp from Python Package Index.

```
$ pip install pymwp==0.2.1
...
Successfully installed pymwp-0.2.1
```

Double-check that pymwp was added to path.

```
$ pymwp --version
pymwp 0.2.1
```

This means installation completed successfully.

Obtain programs for analysis We can use pymwp to analyze any program constructed using the supported C language syntax.[8] For a quick start, we download a set of suitable programs from the pymwp development repository.

Download a pymwp release as a zip file. It includes the example programs we will analyze during this demo.

```
$ wget -O pymwp.zip \
    https://github.com/statycc/pymwp/archive/refs/tags/0.2.1.zip
```

Extract contents of the zip file to a directory, then change to that directory. For the remainder of this demo, we consider pymwp_demo as the working directory.

```
$ unzip pymwp.zip -d pymwp_demo && cd pymwp_demo
```

Copy the example programs to the working directory.

```
$ cp -R pymwp-0.2.1/c_files/* ./
```

[8] See documentation: https://statycc.github.io/pymwp/features/.

Check that example programs were copied successfully. The programs are categorized into subdirectories and `readme.md` includes their descriptions.

```
$ ls
basics  implementation_paper  infinite  not_infinite  original_paper
    ↪ other  pymwp-0.2.1  readme.md
```

This completes the setup. We are ready to start using `pymwp`.

A.2 Default behavior and arguments

`pymwp` requires one positional argument as input: a path to a `C` file. That file is preprocessed using a system `C` compiler then converted to an abstract syntax tree. Debugging information and analysis result are logged to the screen. The analysis result is also written to a file. This behavior is customizable to accommodate various runtime scenarios. For a list of all available options, specify `--help` flag.

```
$ pymwp --help
usage: pymwp [-h] [-o OUT] [--logfile LOGFILE] [--cpp_path CPP_PATH]
             [--cpp_args CPP_ARGS] [--headers HEADERS] [--no_cpp]
             [--no_eval] [--no_save] [-s] [--version] [input_file]

Implementation of MWP analysis on C code in Python.

positional arguments:
  input_file           C source code file to analyze

optional arguments:
  -h, --help           show this help message and exit
  -o OUT, --out OUT    file where to store analysis result
  --cpp_path CPP_PATH  C pre-processor [default: gcc]
  --cpp_args CPP_ARGS  C pre-processor arguments [default: -E]
  --headers HEADERS    C headers dir paths, separate by comma
  --no_cpp             disable C pre-processor
  --no_save            do not write analysis result to a file
  --no_eval            skip evaluation
  --fin                ensure completion even on failure
  --logfile LOGFILE    write console output to a file
  -s, --silent         disable console output
  --version            show program's version number and exit
```

A.3 Program analysis

Polynomially-bounded examples

Example 6. Consider a program that contains a `while` loop with a binary operation. It should seem familiar because it is the same example introduced in the paper (cf. Example 1). We re-introduce it in this demo to see it in action. This program is polynomially bounded in inputs.

```
$ cat basics/while_2.c
/*
 * This program tests that a simple while program results in the correct
    ↪ analysis.
 */

int foo(int x, int y){
    while (0) {x = y + y;}
}
```

Analyzing `while_2.c` with `pymwp` we obtain, as expected, a polynomial bound—note that the returned choice is 2, as discussed in Example 1.

```
$ pymwp basics/while_2.c
...
[12:46:00] INFO (analysis):
MATRIX
------------------------------------------------------------------------
x  |  +m+i.delta(0,0)+i.delta(1,0)  +o
y  |  +i.delta(0,0)+i.delta(1,0)+w.delta(2,0)  +m
------------------------------------------------------------------------
[12:46:00] INFO (analysis): CHOICES: [[[2]]]
[12:46:00] INFO (file_io): saved result in output/while_2.json
[12:46:00] INFO (analysis): Total time: 0.1 s (83 ms)
```

Example 7. Next an example with a conditional statement and 4 input variables.

```
$ cat not_infinite/notinfinite_3.c
int foo(int X0, int X1, int X2, int X3){
    if (X1 == 1){
        X1 = X2+X1;
        X2 = X3+X2;
    }
    while(X0<10){
        X0 = X1+X2;
    }
}
```

```
$ pymwp not_infinite/notinfinite_3.c
...
------------------------------------------------------------------------
X0  |  +m+i.delta(0,2)+i.delta(1,2)  +o  +o  +o
X1  |  +p.delta(1,0).delta(2,2)+i.delta(0,2)+i.delta(1,2)+w.delta(2,2)..
X2  |  +i.delta(0,0).delta(1,2)+p.delta(0,0).delta(2,2)+i.delta(1,0)....
X3  |  +i.delta(0,1).delta(0,2)+p.delta(0,1).delta(2,2)+i.delta(1,1)....
------------------------------------------------------------------------
[13:05:50] INFO (analysis): CHOICES: [[[0, 1, 2], [0, 1, 2], [2]]]
...
```

Note the increased size of the output matrix (concatenated for brevity). This is expected: matrix size increases with variable count. Also observe the `i`-coefficients (∞) in the matrix: they indicate failure along respective derivation paths. The output of `CHOICES` shows that the program is polynomially bounded in inputs, but not all derivation choices yield that bound. Every derivation choice is valid for the `if` statement, but only one choice is valid for the `while` loop. For any sequence of choices that meet these constraints, a polynomial bound is guaranteed.

Examples without polynomial bound

Example 8. Consider an exponential program

```
$ cat infinite/exponent_1.c
/*
 * This program tests that a simple program computing the
 * exponentiation results in matrix with infinite coefficient in them.
 * Inspired from https://stackoverflow.com/a/213897
 */

int main(int x, int n, int p, int r){
    p = x;
    while (n > 0)
    {
        if (n % 2 == 1)
            r = p * r;
        p = p * p;
        n = n / 2;
    }
}
```

We confirm with `pymwp` that no polynomial bound exists for this program.

```
$ pymwp infinite/exponent_1.c
...
[13:09:03] INFO (analysis): RESULT: main is infinite
[13:09:03] INFO (file_io): saved result in output/exponent_1.json
[13:09:03] INFO (analysis): Total time: 0.1 s (83 ms)
```

We can obtain more details about this failure by repeating the analysis with specific instructions to compute the final matrix. This allows locating the source(s) of failure: here, it originates from variables `p` and `r`, i.e., the 3rd and 4th rows.

```
$ pymwp infinite/exponent_1.c --fin
...
MATRIX
------------------------------------------------------------------------
x  |  +m  +o  +m+i.delta(0,2)+i.delta(1,2)+i.delta(2,2) +i.delta(0,1)...
n  |  +o  +m  +i.delta(0,2)+i.delta(1,2)+i.delta(2,2)  +i.delta(0,1)+...
p  |  +o  +o  +i.delta(0,2)+i.delta(1,2)+i.delta(2,2)  +i.delta(0,1)+...
```

```
r  |  +o  +o  +i.delta(0,2)+i.delta(1,2)+i.delta(2,2)  +m+i.delta(0,1)..
-------------------------------------------------------------------------
```

Example 9. Increasing the number of input variables and program statements makes it difficult to determine if a polynomial bound exists.

```
$ cat infinite/infinite_6.c
int foo(int X1, int X2, int X3, int X4){
    if (X3 == 0){
        X1 = X2+X1;
    }
    else{
        X2 = X3+X1;
    }
    while(X4<100){
        X1 = X1+X3;
        X2 = X3+X4;
        X3 = X4+X2;
        X4 = X1+X2;
    }
}
```

With `pymwp` we can easily determine the result. We omit the matrix here for brevity, but a detailed analysis, with the `--fin` flag, reveals that failure occurs at all program variables inside the `while` loop.

```
$ pymwp infinite/infinite_6.c
...
[13:10:19] INFO (analysis): RESULT: foo is infinite
[13:10:19] INFO (file_io): saved result in output/infinite_6.json
[13:10:19] INFO (analysis): Total time: 1.0 s (1035 ms)
```

Analysis challenge After seeing the examples of programs with and without polynomial bounds, we present the following challenge. The task is to determine if this program is polynomially bounded in inputs. Note that it is unknown whether the `while` loop will terminate, however this is not a problem for our analysis.

```
$ cat other/dense_loop.c
...
int foo(int X0, int X1, int X2){
    if (X0) {
        X2 = X0 + X1;
    }
    else
    {
        X2 = X2 + X1;
    }
```

```
    X0 = X2 + X1;
    X1 = X0 + X2;
    while(X2){X2 = X1 + X0;}
}
```

Challenge solution The full matrix must be omitted here for brevity, but can be inspected as an online demo.[9] The matrix contains following information.

- For any source variable, where either `X0` or `X1` is the target, the dependencies are consistently polynomially bounded, for all choices.
- The situation is different between any source variable and `X2` as the target. Multiple deviation choices fail. From the matrix, we can also observe that failures occur at the `while` loop, independent of the which branch of the conditional statement was selected.

There are however multiple sequences of choices that still allow completing the derivation. The generated choice vector captures the valid choices we can apply to complete the derivation. Therefore, the solution is yes, the program is polynomially bounded in inputs.

```
$ pymwp other/dense_loop.c
...
[23:12:13] INFO (analysis):
MATRIX
------------------------------------------------------------------------
X0  |  +m.delta(0,0).delta(0,2)+p.delta(0,0).delta(1,2)+w.delta(0,0)....
X1  |  +p.delta(0,0).delta(1,2)+p.delta(0,0).delta(2,2)+p.delta(1,0)....
X2  |  +m.delta(0,1).delta(0,2)+p.delta(0,1).delta(1,2)+w.delta(0,1)....
------------------------------------------------------------------------
[23:12:13] INFO (analysis): CHOICES: [[[0, 1, 2], [0, 1, 2], [0, 1, 2],
    ↪ [0, 1, 2], [2]]]
[23:12:13] INFO (file_io): saved result in output/dense_loop.json
[23:12:13] INFO (analysis): Total time: 0.1 s (132 ms)
```

Explore further We suggest reading the `readme.md` file, in the working directory, that describes the examples. A formatted version of the readme is available online.[10] We then suggest analyzing more examples independently. After developing sufficient familiarity with `pymwp`, create custom programs for analysis.

Clean up and exit Remove temporary files and examples.

```
$ cd .. && rm -rf pymwp.zip pymwp_demo
```

If you wish to remove `pymwp` from host system, run

[9] `https://statycc.github.io/pymwp/demo/#other_dense_loop.c`
[10] `https://statycc.github.io/pymwp/examples/`

```
$ pip uninstall -y pymwp
Found existing installation: pymwp 0.2.1
Uninstalling pymwp-0.2.1:
  Successfully uninstalled pymwp-0.2.1
```

This completes the demonstration.