

Exploring the Effectiveness of Using Promising Subprograms in Program Synthesis with Budgeted Search

Alexander Jelev

Supervisor(s): Sebastijan Dumančić

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 25, 2026

Name of the student: Alexander Jelev
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Jesper Cockx

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Program synthesis is a computationally difficult task. Budgeted search seems to be a relatively unexplored method of performing program synthesis tasks. This paper describes a budgeted program synthesis implementation that makes use of subprograms it discovers during attempts and leverages them in future attempts in hopes of finding an optimal solution faster. The approach performed slightly worse than most of the control synthesis task runs during experiments. However, some specific cases were discovered for which the budgeted synthesis ran faster than the control synthesis task.

1 Introduction

This project is in the area of program synthesis. Program Synthesis is a powerful way of solving any problem that can be represented as a programming language using input and output examples. Some examples of domains where this is useful are data wrangling, code suggestions and code repair [5].

"In its most general formulation (for a Turing-complete programming language and an arbitrary constraint) program synthesis is undecidable," Gulwani et al.[5]. To find a program that satisfies the given examples, a search over a program space can be performed. Budgeted search in particular is a new approach that has shown promise, but the research that has been done using it is mostly in isolation and the results are not compared with other research. The goal of this project will be to compare these approaches by looking at previous work in the field and then centralizing our group's ideas.

The main research question that this paper will attempt to answer is:

How do we find frequently occurring subprograms in different attempts and effectively leverage them, following the idea of FrAngel?

To explore this question, a budgeted search program synthesis program will be implemented. Since the goal will be to evaluate the effectiveness of the FrAngel program synthesis approach in a budgeted search context, the main elements of its proposed algorithm will be included in the implementation. These include the ideas of Shi et al. on fragment mining and program selection [11].

The implementation will be done in the Herb.jl framework, which also provides benchmarks that will be used to evaluate the performance of the approach.

2 Background

2.1 The FrAngel Algorithm

The FrAngel algorithm is a program synthesis algorithm designed for synthesizing java programs using a "library of components (functions)"[11]. The paper also has a focus on generating programs with control structures. As for my research, this part was omitted due to the complexity that it would introduce to my problem. Instead, the focus was put on fragment mining and program selection.

Before explaining the algorithm, there are some definitions to lay down. The term *program* is used to refer to a function with the required signiature. *Fragments*, as defined by Shi et al. [11] in their paper are "any complete subtree of a program's abstract syntax tree (AST)". Fragment mining refers to the extraction of fragments from a subprogram that was deemed useful. A test case is an input/output example of how the desired program is meant to behave. Angelic conditions are a mechanism by which the algorithm models control structures in order to allow for fragment mining. As mentioned before, this will not be explored in this paper.

Onto the algorithm itself. As input, it takes the target program signature, a set of available libraries and a set of test cases. It proceeds in three general steps, which are program generation, angelic condition resolution and fragment mining. The algorithm only proceeds from step one to two if it passes at least one test case. Also, the algorithm only proceeds to step three from step two if the candidate program's angelic conditions can all be resolved.

Another notable aspect is the logic for remembering programs between each candidate program evaluation. A generated program is remembered on the basis of being the simplest program that passes a subset of the passed test cases. If a simpler program is encountered during the execution of the algorithm, the old program is forgotten and the new one is remembered.

2.2 Special-Case Similarity and Exposure

Special-case similarity is a property of programs that is explained in the FrAngel paper. There, it is used to justify the usefulness of fragment mining by giving intuition as to why it should improve the program synthesis process. This same concept is relevant to the work done in this paper, since I make use of a restricted type of fragment mining, namely remembering full subprograms.

To describe this property, we can take the example of having a set of input/output examples C and a target (or optimal) program P_t that satisfies those examples. Define a (sub)program P_s which is able to satisfy a subset of the test cases $C|_{P_s} \subseteq C$ (and is also the simplest program to satisfy that subset). P_s is said to be special-case-similar to P_t. "We say that C exposes the special case S if $C|_{P_s} \neq \emptyset$ and P_s is the simplest program consistent with $C|_{P_s}$" [11].

With these definitions in mind, the rationale behind fragment mining can be explained. Programs (P_s) are remembered on the basis of being the simplest program satisfying a subset of the set of input/output examples. Moreover, two more assumptions are made optimistically when a program is remembered. It is assumed that P_t is special-case similar with a special case S exposed by C and that $C|_{P_s}$ exposes S [11].

2.3 Herb.jl

As mentioned previously, Herb.jl is a program synthesis framework that provides the tools to structure a program synthesis problem and then solve it. Its goal is to be a general-purpose program synthesis library to provide the building blocks for program synthesis. The reason for this is that most program synthesizer implementations are domain specific, do not provide easy reuse of modules, and are hard to benchmark and compare with each other [6].

To perform a program synthesis attempt in the Herb.jl multiple components have to be defined. Firstly, there must be a *grammar* defined, which defines the "language" that programs will be generated in. This grammar is used by the *iterator*, which enumerates programs in the search space. We refer to these programs as *candidate (sub)programs*. Next, there is the *problem*, which contains input/output examples. Finally, the programs are evaluated using an *interpreter*.

3 Related Work

There are a number of recent papers that have explored some promising approaches for program synthesis. These approaches can also be applied in a budgeted search context, which was the goal of each member of my research group.

The paper on Probe makes use of a probablilistic grammar to guide the program synthesis search procedure[1]. By making use of successful programs found during the synthesis, the grammar probabilities are updated to favour rules that are used by those programs. This was explored more in depth in [9].

The paper on Knorf has an interesting approach to improving efficiency as they perform *knowledge refactoring* in order to "restructure a learner's knowledge base to reduce its size and to minimise redundancy in it."[3]

DreamCoder is an approach that is in a way similar to mine in that it learns new programs components based on solutions to synthesis attempts. These components can then be used to solve similar problems more efficiently[4]. This approach was explored more in-depth in [10].

4 Methods

4.1 Algorithm Description

The algorithm that I implemented is divided into two parts. On the top-most level there is the budgeted-search skeleton. Then there is the specific implementation which draws on the fragment mining ideas of FrAngel and implements them within the Herb.jl framework.

Budgeted Search Framework

The pseudocode for the budgeted search skeleton is shown in **Algorithm 1**. The basic premise is that a synthesis attempt is performed with a budget and then something is learned from this synthesis attempt.

Each attempt begins with the actual synthesis, which makes use of a passed grammar and iterator to generate programs and evaluate them based on input/output examples. The solution of the synthesis is returned, which can be optimal if it passes all examples, or suboptimal if it does not.

Next, the StopChecker decides if the budgeted search should be halted and a result returned. After that, the Selector function searches through the solution and extracts the information that is relevant to the budgeted search. Finally, the Updater alters the parameters of the budgeted search to make use of the results extracted by the Selector function.

Algorithm 1 Budgeted Search Skeleton

Input: BSC - Budgeted search controller with parameters and functions defined for a specific budgeted search implementation, as well as the selected iterator and problem.

1: **function** RUNBUDGETEDSEARCH(BSC)
2: $results \leftarrow \emptyset$
3: $times \leftarrow \emptyset$
4: $bank \leftarrow \emptyset$
5: **for** $attempt \leftarrow 1$ **to** $BSC.budget$ **do**
6: $solution \leftarrow$ SYNTHESIZE($BSC.iterator, BSC.problem$)
7: $time_count$ += $solution.time$
8: $times \leftarrow times \cup solution.time$
9: $results \leftarrow results \cup solution.value$ STOPCHECKER(solution)
10: $selected =$ SELECTOR($solution.value, bank$)
11: $BSC.iterator =$ UPDATER($selected, bank$)
12: **end for**
13: **return** {results, times, time_count}
14: **end function**

FrAngel Implementation

As shown in the previous subsection, there are a number of methods that have to be defined, namely the Synthesis, Updater, Selector and stop checker functions.

The Synthesis function makes use of the iterator and problem that are passed to enumerate programs from a grammar. While doing so, it also keeps track of programs that pass a subset of the examples defined in the problem. After completing all enumerations, or reaching predefined maximum amount of enumerations, the best found program is returned (the one that passed the greatest number of examples).

Next, the Selector function serves to populate the bank with the simplest found programs for each subset of test cases during the synthesis attempt.

Finally, the Updater function serves to alter the grammar of the iterator in order to include newfound fragments that seem promising or update existing rules if a simpler alternative has been found. After a grammar update, the iterator gets reconstructed and begins enumerating programs anew. This was done because updating the grammar causes problems with some internal constraints of the iterator, and so it is best to reconstruct it instead. A consequence to beginning the iteration anew is shown later on in section 4.1.

The StopChecker function simply checks if an optimal program has been found, meaning it passes all examples from the problem definition.

5 Experimental Setup and Results

5.1 Experimental Setup

The goal of the experiments will be to contrast the different budgeted search approaches explored by my peers and I. As such, they are set up in such a way that the results can be easily compared.

For experimental runs, my colleagues and I used the DelftBlue supercomputer, making use of their memory partition compute nodes [2]. They were run on Intel Xeon Gold 6248R 24C 3.0GHz cores with memory per core of 15G. We each set a 4 hour timeout on our jobs.

We set four common parameters for the comparative experimental runs: A maximum tree depth of 10, a maximum program size of 15, a maximum enumeration count of 10000 and a maximum number of budgeted search attempts of 100.

For the benchmarks, we made use of the PBE_SLIA_Track_2019 [8] benchmarks offered by HerbBenchmarks. These are a collection of string-manipulation problems from the SyGuS Challenge 2019. They provide a number of input/output examples, each accompanied by a grammar.

There were some individual differences between the main experimental runs, due to the different nature of our approaches. The following are what I used for my run. I made use of the SizeBasedBottomUpIterator from HerbSearch and the interpret_sygus interpreter included in HerbBenchmarks. Experiments were performed in the budgeted search framework as described previously and made use of the FrAngel-inspired synth, selector and updater functions.

Lastly, I paralellized the experiment such that ten OpenMPI processes would each run 20 of the benchmark problems, and four out of those would run one extra benchmark to cover all 204 benchmarks. I also introduced a timeout of 30 minutes similar to the FrAngel paper [11], since I noticed that some benchmarks were taking very long to run.

Approach	**Total Benchmarks Evaluated**
Sketches	204
Useful Subprograms	148
Proabilistic Grammar	204
	Optimal Solutions
Sketches	51
Useful Subprograms	34
Proabilistic Grammar	33

Table 1: Total benchmarks evaluated for each budgeted search approach and number of optimal solutions (score = 1.0) found

5.2 Comparison with other Budgeted Search Methods

To compare the results between the other members we decided to simply look at the number of benchmarks that the budgeted search approach is able to find an optimal solution for. Constrained by the same budget, we obtained the results that can be seen in Table 1. The results for the two other approaches were provided by my colleagues[7][9].

The reason for the Useful Subprograms implementation having a lower total evaluated benchmarks count is that the timeout of 4 hours of the job on the supercomputer was reached before all the benchmarks were attempted.

5.3 Individual Experiments

The rest of the analysis will be performed for specifically budgeted search with useful subprograms and will be guided by a set of questions derived from the research question from the Introduction section:

How do we find frequently occurring subprograms in different attempts and effectively leverage them, following the idea of FrAngel?

- **Does budgeted search with useful subprograms improve the effectiveness of the synthesis task?**
- **Does budgeted search with useful subprograms find better intermediate solutions than a standard synthesis task?**

Each of these questions is covered by one of the sections that follow. One thing to note about the data is that due to the premature timeout of the job, one of the benchmarks had a data point for the control run but no corresponding point for the budgeted run. As such, it has been excluded from the results analysis and discussion.

5.4 Effectiveness

Does budgeted search with useful subprograms improve the effectiveness of the synthesis task?

Effectiveness here refers to finding solutions or programs in as low a number of synthesis iterations and runtime as possible. The question itself aims to answer whether a program synthesis attempt with budgeted search is able to find an optimal solution in a lower total number of synthesis iterations or runtime compared to a program synthesis attempt without budgeted search.

Another way to look at it is whether for a problem-grammar pair, the budgeted search approach is able to find a solution that is better than a synthesis attempt without budgeted search. A better solution simply means that the program is able to evaluate more input/output examples correctly.

The data I deemed most relevant is summarized on 5 graphs. As can be seen on Figure 1, the final scores were almost identical for both the budgeted search and the control runs.

Figure 2 and Figure 3 show how runs obtain optimal solutions with time and number of iterations (only the ones which had a final score of 1.0). The iterations graph has stepwise increments since there were 10000 iterations per attempt and the graph shows the change with each attempt. As can be seen from those graphs, budgeted search seems to slightly under-perform in comparison to the control runs.

Finally, Figure 4 and Figure 5 show the instances where budgeted runs outperformed control runs and vice versa. The corresponding benchmarks for which these cases occurred are also shown. One thing to note though is that the programs found here were not optimal.

When comparing the relative discrepancies in runtime, the benchmarks where the budgeted search performs best seem to far outperform control. Notably, they are all phone problems. It is possible that this is something to do with the search space for the problem. For example, for problem_phone_5_long_repeat, the grammar is updated after the first attempt and adds 5 new rules (and the ultimate best-performing solution is also found and added to the grammar). This could have given the iterator the opportunity to enumerate a large amount of new programs more easily, and so would have allowed for the budgeted search attempt to reach the iterator cap quicker.

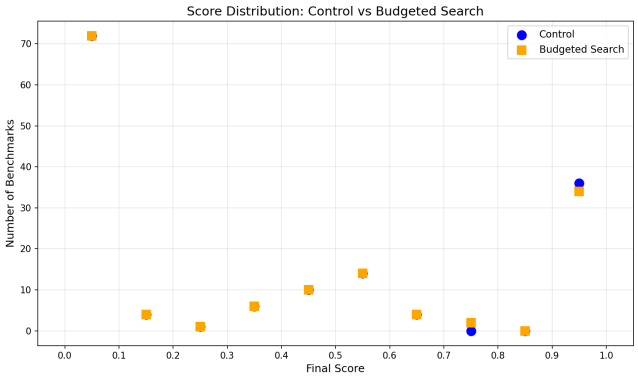

Figure 1: Score distribution

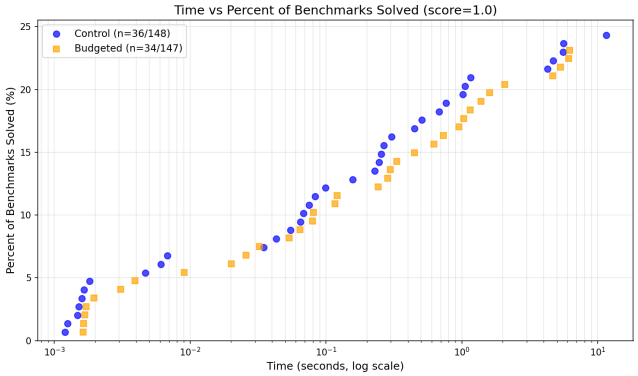

Figure 2: Solutions vs Time

5.5 Intermediate Solutions

Does budgeted search with useful subprograms find better intermediate solutions than a standard synthesis task?

Here, the analysis will look at solutions obtained after each budgeted synthesis attempt and compare the results obtained by the two approaches. These will be compared based on the metric of program score, which is simply the number of passed examples divided by the total.

After this analysis was performed with a simple python script, the result was that in 9 of the evaluated benchmarks, the control run had one or more better intermediate programs that yielded a positive score when compared to the budgeted search ones. This could be a result of the iterator restarting after a grammar rule is updated and having to iterate through programs that were already enumerated previously, causing unnecessary overhead and resulting in not finding more programs that pass test cases.

The answer to the question is no, budgeted search with useful subprograms does not seem to find better intermediate solutions (for this benchmark).

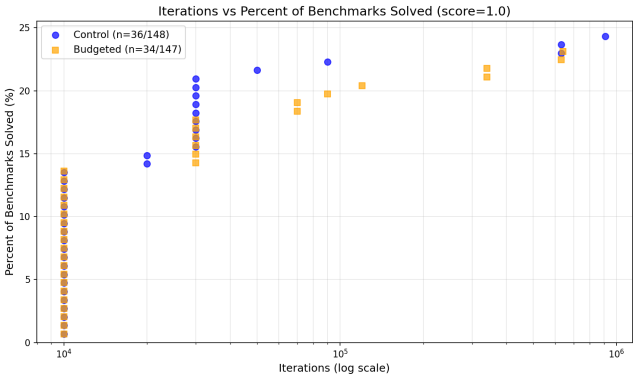

Figure 3: Solutions vs Iterations

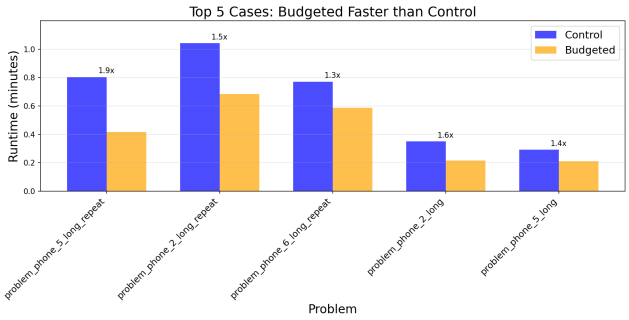

Figure 4: Budgeted Faster Times

6 Responsible Research

Since the data that we use comes from pre-made string manipulation benchmarks, and does not influence or involve other people directly, there aren't many ethical aspects to consider in this area. Where I put my focus instead was on scientific integrity and reproducibility.

I made sure to document thoroughly the experimental setup, such that the findings could be easily reproduced by anyone willing to check the work just by following the steps laid out in section 4.1.

7 Conclusions and Future Work

This paper has explored the effectiveness of budgeted search with useful subprograms in the field of program synthesis. The algorithm that was implemented made use of a mechanism of remembering the simplest subprogram for each subset of input/output examples.

Experiments were performed to compare benchmarks being ran with a simple iteration and compared to the implemented budgeted search based on the amount of input/output examples they solved, the number of iterations it took them to solve said benchmarks and also the length of time.

After analyzing the results, it seems that in the majority of cases, the budgeted search approach

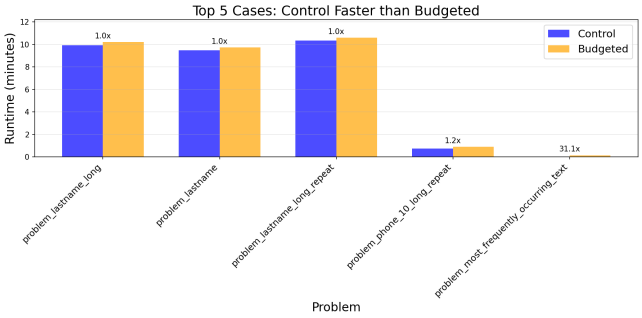

Figure 5: Control Faster Times

seems to under-perform compared to the control runs. Although there were a few cases where budgeted search performed better, it seemed as though they were a result of some properties of the search space changing when adding new rules, rather than being more useful for the program synthesis task. Of course, these conclusions were drawn from one set of benchmarks for string transformations.

Future Work

One suggestion for future improvement is testing the budgeted search approach on a larger variety and quantity of benchmarks. Due to the nature of fragment mining, input/output examples that cover different aspects of the program should in theory expose more useful subprograms and boost the effectiveness of budgeted search.

Another suggestion for improvement is figuring out how to update or add grammar rules within HerbSearch without violating iterator constraints, such that previously enumerated programs do not need to be re-explored. As mentioned earlier in the paper, the budgeted search procedure runs up until it has to update the grammar, and then it reconstructs the iterator and begins enumerating programs anew. This means that there is overhead in enumerating programs that were already considered before.

A LLM Usage

An LLM was used to help with writing the scripts for the experimental setup. Since CostBasedBottomUpIterator was used initially, I made use of the tool to create a new setup using the SizeBasedBottomUpIterator based on my old one, as well as for writing the Control run portion of the experiment, based on my budgeted search synth function.

Also, an LLM was used for writing the data analysis scripts that read out of generated experimental CSV files and generated graphs and simple statistics.

References

[1] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-

up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[2] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[3] Sebastijan Dumancic, Tias Guns, and Andrew Cropper. Knowledge refactoring for inductive program synthesis. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(8):7271–7278, May 2021.

[4] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pages 835–850, 2021.

[5] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[6] Tilman Hinnerichs, Reuben Gardos Reid, Jaap de Jong, Bart Swinkels, Pamela Wochner, Nicolae Filat, Tudor Magurescu, Issa Hanou, and Sebastijan Dumancic. Herb.jl: A unifying program synthesis library, 2025.

[7] Una Jaćimović. Identifying and leveraging structural sketches in budgeted program synthesis. Bachelor's thesis, Delft University of Technology, 2026.

[8] Saswat Padhi, Udupa Abhishek, Andi Fu, Elizabeth Polgreen, and Andrew Reynolds. Benchmarks for sygus competition. https://github.com/SyGuS-Org/benchmarks, 2019.

[9] Chris Preda. Exploring probe-style grammar updates in budgeted search program synthesis. Bachelor's thesis, Delft University of Technology, 2026.

[10] Jelle Römer. Utilizing frequently occurring subprograms in program synthesis. Bachelor's thesis, Delft University of Technology, 2026.

[11] Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.