Improving on Very Large Neighborhood Search techniques in Program Synthesis

Rixt Hellinga[1]
Supervisor: Sebastijan Dumančić[1]
[1]**EEMCS, Delft University of Technology, The Netherlands**
23-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Inductive Program Synthesis (IPS) has been implemented by a two-stage search algorithm, Brute, and consequently improved upon with a Large Neighborhood Search (LNS) technique, in an algorithm named Vlute. Unmotivated values and design choices within Vlute caused limitations on the performance of IPS tasks. This research improves upon several of these limitations through experiments. Most significant improvements are found in the robot-planning domain through the implementation of a stochastic accept method and a best improvement neighbor search.

1 Introduction

For a user wanting to produce a program, it is in general easier to define what a program is supposed to do than to define the program's steps in detail. Program Synthesis concerns the matter of automatically finding programs from simple instructions and user-specified constraints. Within program synthesis there are several levels of detail in which the constraints can be specified. Fewer specifications are simpler to define yet risk a less efficient program [14]. Finding that program through a set of input and output examples is called Inductive Program Synthesis (IPS). It is considered to be an important part of the Artificial Intelligence domain, as many processes in many domains can be automated through the involvement of program synthesis. Such domains include Intelligent Tutoring Systems, embedded systems like digital controllers in cars, and functions in Microsoft Excel for end-users [6; 11; 15].

The search space of all programs to be explored in Program Synthesis can be expressed as a search tree. This search tree can take on enormous sizes and thus exploring the entire search tree can take unrealistic amounts of time. Search algorithms can reduce the amount of time needed to traverse the search space, which is why search algorithms play an important role in inductive program synthesis [29; 4]. Improvements on the search over the tree can be made by using local search algorithms. Within these local search algorithms there are many variables that can be altered in order to make the algorithm more suited to each problem.

The Very Large Neighborhood Search (VLNS) is an example of such a local search algorithm. In previous work a two-stage IPS system, Brute, has been developed that invents programs consisting of very few instructions, called tokens, and guides the search through the tree by combining these tokens into larger programs. Brute then analyzes a distance measure of the output of the program to assess the correctness of the program [1]. An adaptation in the algorithm of Brute, Vlute, that uses a VLNS algorithm with a Variable-Depth Invent (VDI) stage [24], has shown to perform better than Brute in some domains.

However, some gaps are left in the aforementioned research. Meant here, is that a number of values are left to be randomly selected and some important decisions remain unmotivated. Other research gaps include determining the usefulness of tokens and varying on the search depth increment. This leaves the question of whether the performance of a VLNS algorithm with VDI could possibly be improved. The aim of this research is thus to find improvements in several aspects of the Vlute algorithm, through answering the following research question:

"How can we improve on the Variable-Depth Invent variation of the Very Large Neighborhood Search algorithm for Inductive Program Synthesis?"

In order to more precisely answer this, the research question can be divided into several sub-questions:

- What is the effect of a varying depth-increment on the search algorithm?
- What is the effect of different weights for tokens on the search algorithm?
- How can randomly-chosen values of variables be chosen to improve on the current destroy and repair algorithms?
- What is the effect of pruning programs on the search algorithm?
- What is the effect of applying a best improvement strategy?
- How can we improve the acceptance strategy?

In order to answer these question this research paper will adhere to the following structure: first, important concepts of VLNS, Brute, and Vlute will be explained in background and related work. Then the methodology will be explained, where the practical contribution to the code will become clear. After this the experimental setup and the domains will be laid out. Then the results of the experiments will be discussed and followed by the conclusion and discussion.

2 Background and Related Work

This section will introduce and go more into depth on relevant related concepts to this research such as Brute, VLNS, and six important components of Vlute.

2.1 Very Large Neighborhood Search

Very Large Neighborhood Search (VLNS) is a local search algorithm that takes a solution and repeatedly explores solutions that are similar to it in order to improve on the solution [3; 26]. The exploration of the neighborhood with similar solutions often follows some heuristic. "*The success of search-based optimisation algorithms depends on appropriately balancing exploration and exploitation mechanisms during the course of the search.*" [27], which also applies to VLNS. Leaning towards exploration can cause the algorithm to take a long time to converge, and leaning towards exploitation is prone to get stuck in local optima.

Several variations of this algorithm exist, in which the size or structure of the neighborhood is changed throughout the algorithm [26]. Changing neighborhood sizes and structures allows the algorithm to be applied to different problems. Another use is that changing neighborhoods can be employed to avoid large sizes of neighborhoods, as VLNS can be prone

to explodingly increasing sizes of neighborhoods: "*Learning large programs could be difficult since the size of the neighborhood grows as the size of the sequence grows since there are more ways to destroy the sequence.*" [24].

2.2 Brute

Brute is a two stage IPS system. In the first stage, the invent stage, several tokens are invented. These are categorized as transition tokens that change a given state, control tokens that determine the flow of a program (i.e. *if* and *loop*), and boolean tokens that always return true or false when applied to a state. The search stage follows the invent stage. Here the tokens that were found in the invent stage are combined to create a more complex program, of which the cost is then calculated. Brute repeatedly goes through this process, saving best-found programs until time runs out or a program is found that matches all the user specified constraints and examples [1].

2.3 Vlute

Vlute is an adaptation on Brute, where both the invent stage and the search stage are different. In the invent stage Vlute uses a Variable-Depth Invent (VDI) stage, where tokens of different complexities are invented. In Brute each search iteration uses the same tokens as the invent stage is static. In Vlute, just as in Brute, several of the simplest tokens are combined to form a little more complex tokens. These more complex tokens are limited to some depth, that determines the number of transition tokens and control tokens that the new more complex tokens consist of. The difference lies in that, in Vlute, the depth of the invented tokens can increase when the search stage yields no acceptable results, thus being a variable invent stage. The maximum depth in the invent limited to three depth levels in order to avoid an explosion of the number of tokens.

In the search stage these tokens are combined, but the difference of this with Brute is that Vlute uses a VLNS algorithm to explore the new possibly created programs through destroy and repair methods. Vlute takes the current solution and destroys it by taking some random index I in the program and removing N random tokens from the program at that index. Then the tokens that were invented by the VDI stage can be inserted back into the program in the repair method of the search stage [24].

There are some components of Vlute that are interesting and will be highlighted in sections 2.3.1-2.3.6.

2.3.1 N_i increment

The complexity, or depth, of tokens determines the balance between exploration and exploitation in the algorithm. The depth of the invented tokens, N, is determined by some set variable, which can change over iterations when no good solutions are found. When this variable should change is determined by the variable N_i where i is the number of iterations in which no better solution has been found. Throughout each run this i remains constant, meaning that after each i iterations N is increased. In Vlute VDI has only shown to improve results in a single domain. However, options to vary this i within experiments have not yet been explored.

2.3.2 Token weights

The types of tokens that are invented determine the flow of the program. Weights of the tokens determine the probability of that token being invented. The weights of the tokens were unmotivated in Vlute, and thus the only assumption can be that they were (semi-)random. However, since they are applied to a relatively uncommon approach of program synthesis research on which values might perform well is not yet available, which might explain the choice for this randomness. Some research exists on the synthesis of loop-free programs, so with the weight for loop-tokens set to 0. Applications include optimizing the core of computationally intensive loops, bitvector algorithms, and geometry constructions [12; 15].

2.3.3 Degree of destruction

A large degree of destruction results in more exploration, while a small degree of destruction result in more exploitation. The randomness of the destroy method in Vlute raises some questions. "*The most important choice when implementing the destroy method is the degree of destruction.*" [26]. In Vlute a random N is picked, bound by some maximum value (the length of the program or N_{max}, corresponding to the number of tokens that are removed from the current solution. These removed tokens are sequential and start from some random index I. N and I thus determine the size and shape of the neighborhood that we explore in that iteration. Randomness in the destroy method of VLNS has actually been used in previous researches, as well as other methods such as increasing, decreasing, and examplesize-dependent degrees of destruction [22; 28; 23; 19]. However, many research papers describe program synthesis in the domain of a Travelling Salesman Problem or some other routing problem, so the question of which destroy method works best in the domains used in this paper remains unanswered.

2.3.4 Best improvement strategy

There are several strategies to determine what the next solution should be to explore from. Brute and Vlute use a first-improvement strategy, which accepts a new solution directly when it improves on the previous solution [24; 1]. Applying a best-improvement strategy will allow for better improvements, as it explores a neighborhood and picks the best neighbor to continue the search with. However, as stated in an earlier paragraph, the size of a neighborhood can be quite large. This means that a best-improvement strategy could take a long time to conclude [20]. To avoid these long computation times a middle ground can be found, in which a set number of neighbors within the neighborhood, and not the entire neighborhood, is explored.

2.3.5 Pruning

As mentioned in [24] pruning tokens has not been employed in the original implementation of Vlute, but could possibly improve the performance of the algorithm. Pruning programs has been employed in local search algorithms in many different variants [21]. One such variant is pruning on observational equivalence. Observational equivalence is when multi-

ple solutions have the same output state for the each test case [7; 13].

2.3.6 Stochasticity

Stochasticity can help avoid algorithms getting stuck in local optima by occasionally accepting a worse solution than the current one and thus apply exploration. This stochasticity has often been used to combined with local search techniques [5; 25; 2]. Set by an initial temperature and a cooling factor, an algorithm with a stochastic acceptance method is more likely to accept a worse solution in the beginning of the experiment and when the worse solution is only a little worse than the current solution. This means that if Vlute were to use this stochastic accept method, it would favour exploration at the start of the run, and exploitation towards the end of the end of the run.

3 Methodology

This section will describe the methods in which the five of the six aspects of Vlute discussed in 2.3.1-2.3.6 are changed or implemented. The exception are the token weights, which are left out as they are easily changed in the existing code and thus require no extra implementation or explanation.

3.1 N_i increment

In order to adjust N_i throughout a run, the original algorithm of Vlute is changed to that of algorithm 1, in which the added code is seen in red. When the number of iterations without improving on the best solution has passed the current N_i, we do not only increase the search depth as Vlute does, but also replace N_i by N_{i+x}. Thus x=*0* corresponds to the original implementation of Vlute. i is of course bound by 0 and some $i_{\_max}$.

Algorithm 1 Vlute with changing N_i

```
input : Ni
output: solution
repeat
    new program = repair(destroy(current program))
     cost = cost(new program)
     if cost = 0 then
        return new program
    if cost < current cost then
        current program = new program
          i = 0
    else
        i = i + 1
    if i > Ni then
        Increase search depth
          i = 0
          Ni = Ni + x
until timeout;
return program
```

3.2 Degree of destruction

Destroying and repairing the found solutions is, as described in previous sections, currently implemented through a random method. This strategy will be changed through a passed parameter, that can be set as "*random*" (by default, as in Vlute), "*increasing*", or "*decreasing*". In the increasing and decreasing strategies the degree of destruction and reparation is proportional to the number of iterations that have passed since the last improvement (i), and the number of iterations needed to increase the search depth (N_i). This method can be seen in algorithm 2.

Algorithm 2 Destroy and repair strategies

```
input : Program, i, Ni, strategy
output: n
mn = min(NMax, program_length)
 n = random(0, mn+1)
 if strategy = increasing then
    n = max(0, mn*i/Ni)
else
    if strategy = decreasing then
        n = max(0, mn - mn*i/Ni)
return n
```

3.3 Best improvement strategy

For the best improvement strategy we iterate over the destroy and repair methods k times. For each iteration the cost of the new neighbor is saved. If the cost improvement is better than any improvement found thus far, we replace the best-found neighbor with the new neighbor. After k iterations he search algorithm resumes with the resulting best neighbor.

Algorithm 3 Best-improvement algorithm

```
input : k
output: best neighbor
best neighbor = current sol best neighbor cost = current cost
 repeat
    destroy and repair old solution
     if cost is lower than best neighbor then
        set best neighbor
until k times;
return best neighbor
```

3.4 Pruning

The pruning will get a separate method, shown in algorithm 4, which will take a dictionary of equivalence classes, a program, and a certain test case. This method is inspired by a log with thus far explored solutions, as employed in [21]. The algorithm for pruning counts the number of times that a cost has been found for that test case, and computes a probability of pruning it. Higher counts result in a higher probability of being pruned. This probability, based on the count, can also be adapted to prune fewer or more programs. This allows the algorithm to be adapted when it appears to prune too much or too little.

3.5 Stochasticity

The stochastic acceptance method was already present, but unused in the original research. Algorithm 5 shows the implementation of the algorithm, which takes a temperature, a

Algorithm 4 Pruning

input : eq_classes, input_state, cost
output: boolean
if *cost in eq_classes* **then**
  count = eq_classes.get(input_state).get(cost)
else
  eq_classes.get(input_state)[cost] = 1
  count = 1
if *random(0,1)* $> \frac{1}{count}$ **then**
  return True
else
  return False

cooling factor, and the cost of both the current program and the neighbor. The probability of a program being accepted is determined by the difference in costs and the temperature at that iteration. The algorithm thus has a higher probability of accepting a new program with a cost close to the current cost than one with a much worse cost. The algorithm also is less likely to accept poor solutions as the experiment progresses, as the temperature lowers each iteration.

Algorithm 5 Stochastic accept method

input : $temp$, $cool\_fac$, $curr\_cost$, n_cost
output: boolean
cost_diff = n_cost - curr_cost
prob = $\exp(-cost\_diff temp)$
temp = temp * cool_fac
if *random(0,1) ¡ prob* **then**
  return True
else
  return False

4 Experimental Setup

This chapter describes the experimental setup, consisting of the values chosen for variables, the domains on which the algorithms are tested, and the hardware environment the experiments are run in.

4.1 Variable values

The N_i increment value in the experiments is set at 100 and 500. This allows N_i to reach values not yet covered in the original research of Vlute, and is therefore most probable to provide new insights. The values for the token weights are originally set as 1 for w_{loop} and 0 for w_{if}. In the experiments I will use two other weight sets: the first being the inverted weights, so *0* for w_{loop} and 1 for w_{if}, and the other being both weights set to 1. This allows analysis on the separate and combined influence of the tokens.

The values used for the degree of destruction are inspired by the current implementation of Vlute, which varies the degree of destruction between 0 and the length of the program or some N_{Max}. The increasing and decreasing strategies will thus vary between the same values. The only variable in pruning would be the probability with which an observationally equivalent program is pruned. This has to be negatively proportional to the number of observationally equivalent programs that have been encountered, resulting in a value of $\frac{1}{count}$ as seen in algorithm 4.

The values chosen for the size of the neighborhood in the best improvement tactic are 10 and 100, and were chosen such that they are much larger than the original size 1. The values chosen for the stochastic acceptance method were chosen through some simple experiments. The initial temperature value in the original code for Vlute was 0.1, however this resulted in acceptance probabilities as small as 10^{-5} which accepted very few worse programs. Some testing showed that an initial temperature of 1.0 resulted in probabilities throughout a run of roughly 0.3 to 0.1, which would likely produce a more discernible effect of the stochastic method. The cooling factor, set at either 0.997 or 0.993, having been chosen because the simulated annealing technique this stochastic method is based on has shown to require slow cooling [16; 17].

For the remaining parameters I have chosen to use the same values as used in the original experiments for Vlute. This allows easy comparison of the results. These are the following settings:

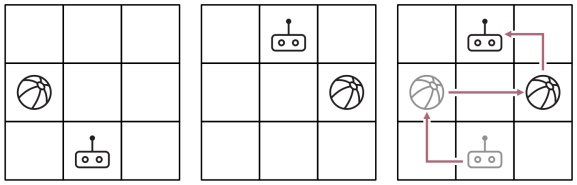
- The depth levels are [(1, 1),(2, 1),(2, 2)], where the first number in each tuple is the number of transition tokens and the second is the number of control tokens.
- The maximum degree of destruction, N_{max} , is set at 3
- The weight for transition tokens, w_{trans} is set at 1.
- Experiments are done with N_i chosen from {1000, 3000, 5000, 10000, 15000, 30000}

4.2 Domains

The domains used to experiment on are the as used in the original research on Vlute [24].

4.2.1 Robot-planning

This domain concerns a robot and a ball in a grid-world of size *nxn*. The goal is to reach an output state with the robot and the ball in a certain cell of the grid. To reach this goal the robot can move and pick up the ball through the following transition tokens: *MoveUp*, *MoveRight*, *MoveDown*, *MoveLeft*, *GrabBall*, and *DropBall*. The boolean tokens in this domain are *AtTop*, *AtRight*, *AtBottom*, *AtLeft*. These tokens can be combined through *if* and *loop* tokens. An example of this domain is shown in figure 1. The test cases for this domain are generated for $n \in 2, 4, 6, 8, 10$. For each *n* there are 110 test cases [1].

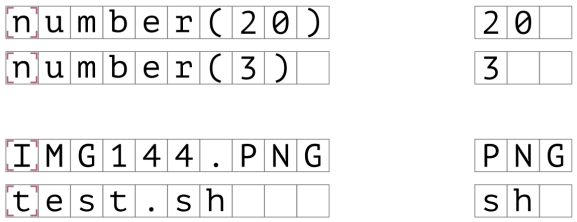


(a) Input state (b) Output state (c) Valid solution

Figure 1: Robot-planning domain visualisation [24]

4.2.2 Real-world string-transformations

Real-world string-transformation takes an input string and a pointer, and has the pointer move and change characters in such a way that the output string is reached. The transition tokens are *MoveRight*, *MoveLeft*, *MakeUppercase*, *MakeLowercase*, and *DropCharacter*, and the boolean tokens are *AtEnd*, *AtStart*, *IsUppercase*, *IsLowercase*, *IsLetter*, *IsNumber*, *IsSpace*. An example of an input string and an output string is shown in figure 2. For testing there are 327 real-world strings to be transformed to output strings, where each string has 10 in/output examples. These examples are divided in $n \in 1, 2, ..., 9$ training samples and $10n$ test samples. Each test case, consisting of training and test samples, is repeated 10 times with different train samples.

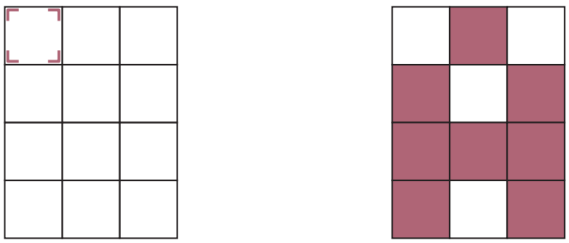


(a) Input states (b) Outputs

Figure 2: String-transformation domain visualisation [24]

4.2.3 Drawing ASCII-art

This domain takes a grid of *w*x*h* blank pixels and a pointer, and has to execute the task of colouring the right pixels. An example with an input grid and an output grid can be seen in figure 3. The transition tokens in this domain are *MoveUp*, *MoveRight*, *MoveDown*, *MoveLeft*, and *Draw*. The boolean tokens are *AtTop*, *AtRight*, *AtBottom* and *AtLeft*. The ASCII characters that are to drawn are generated from strings of length $n \in 1, 2, 3, 4, 5$, for each of which a 100 tasks are generated [1]. The characters are selected through a uniform random distribution and translated to pixel art with *text2art*. The width, *w*, and height, *h*, for each character are 4 and 6, respectively [24].



(a) Input state (b) Output state

Figure 3: ASCII-art domain visualisation [24]

4.3 Experimental environment

The experiments are run on the TU Delft DHPC supercomputer [9], with one node per experiment and using 32 of the 48 available cores per node. The number of variants of the algorithm ran per experiment varies from 2 to 16. The more lightweight experiments were ran sequentially on the same node, while heavier experiments were ran with fewer variants sequentially.

5 Results and discussion

This section will discuss the results of the adapted components described in previous sections for each of the domains. Because the hardware setup of this research differs somewhat from the setup used in the previous research on Vlute, results are compared not directly to that of [24], but rather the same base algorithms for Vlute and Brute are run on the hardware described in this research.

5.1 N_i increment

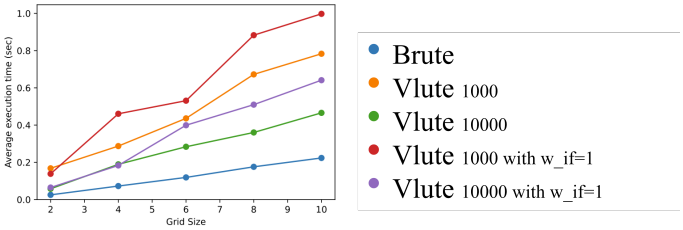
Having Ni increment with 100 or 500 each time N_i is reached does not seem to improve or deteriorate the performance of Vlute much in any of the three domains. The accuracy and the execution time both stay roughly the same regardless of whether N_i is increased or not. Further insight into Vlute provides an explanation into the minor effect of incrementing N_i; Even though higher values for N_i in Vlute perform better than lower values, and incrementing N_i should thus improve the performance of Vlute, the number of times that N_i is reached before finding a better solution is very low. As a result N_i is not often increased.

5.2 Token weights

Setting the weights w_{if} and w_{loop} both to 1 has none to a slight negative effect of Vlute in the three domains. The robot-planning domain shows worse execution time, as seen in Figure 4. This weight assignment as no effect on accuracy in the robot-planning domain. The string-transformation domain shows no decrease or increase in execution time and accuracy. The ASCII-art domain also shows no improvement in either execution time or accuracy, and even an overall small decrease in performance.

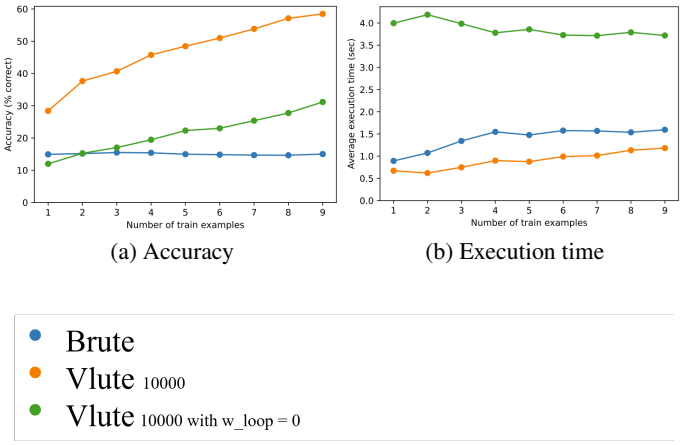
Setting the weights w_{if} and w_{loop} to 1 and 0, respectively, has different effects throughout all three domains. The accuracy of Vlute in the robot-planning domain is not affected, but the execution time is slightly lower. The ASCII-art domain shows some slight improvement in both accuracy and execution time. Another observation is that this assignment of weights to tokens causes slightly more of a spread in the performance of Vlute with different values for N_i. This is noticeable in both the robot-planning and the ASCII-art domain. However, the difference spread is relatively small, possibly because of the limited range of N_i values, so not much can be concluded from this.

The string-transformation domain however shows a much worse performance in both the execution time and the accuracy of Vlute, as seen in Figure 5. A possible explanation of the difference in performance in the string-transformation domain is that string-transformations might often be 'cuts' out of a string, meaning that it loops until a certain starting character has been found, and then loops again until a stopping character has been found. This would result in a domain that heavily depends on loop tokens.



(a) Execution time (b) Legend

Figure 4: Performance of Vlute with w_{if}=1 in the robot-planning domain



(a) Accuracy (b) Execution time

(c) Legend

Figure 5: Performance of Vlute with w_{loop}=0 in the string-transformation domain

5.3 Degree of destruction

Having the degree of destruction in Vlute increase or decrease throughout the iterations does not improve performance in any domain. Both the increasing and decreasing destroy and repair strategies seem to perform worse in execution time and accuracy than Vlute in the robot-planning domain. The execution time and accuracy for the increasing strategy however approach the performance of a random-destroy strategy Vlute for higher N_i's. Both the ASCII-art and the string-transformation domains show a much worse execution time for both the increase and decrease strategy. The accuracy in both domains can be seen in Figure 6. An explanation for this performance of increasing and decreasing degrees of destruction could be that the exploration and exploitation are not performed properly. An increasing degree of destruction could take too long in the beginning of the algorithm to destroy enough to explore the search space. A decreasing degree of destruction could take too long in the end of the algorithm to destroy little enough to exploit the search space. A random degree of destruction might just be the correct balance for this.

5.4 Best improvement

The best improvement strategy for searching through a neighborhood seems most promising in performance throughout the domains. In the robot-planning and the string-transformation domains the execution time is more than halved for Vlute with N_i-value 1000 while receding nothing in accuracy. Noticeable in both domains is that Vlute with a best improvement strategy improves a lot on regular Vlute, but stays somewhat consistent with a growing N_i, as seen in Figure 7. This fading improvement suggests that the value chosen for the number of neighbors to explore should grow with N_i. The ASCII-art domain also shows some improvement, although not as much as the other domains, in both execution time and accuracy, with its performance having the same interesting relation to N_i as the robot-planning domain.

5.5 Pruning

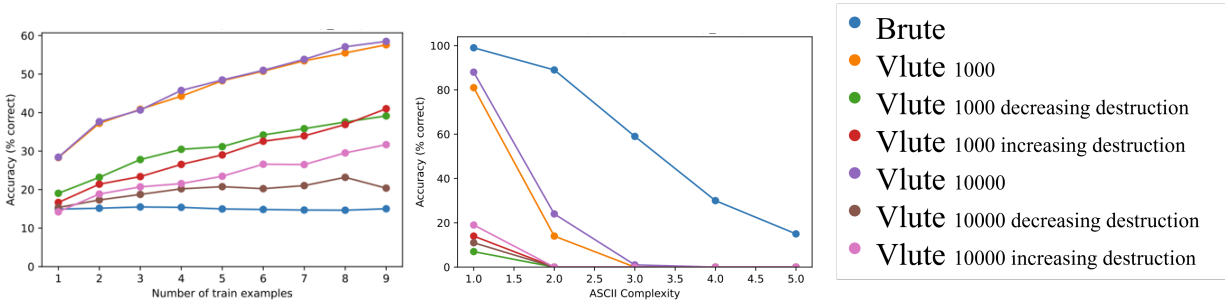
Pruning by itself does not seem to improve the performance of Vlute in any of the three domains. This can mostly be attributed to the way Vlute was implemented; Because the cost of a program is determined by its output state and the desired output state, any solution with the same output state has the same cost. As Vlute would only accepts neighbors with a strictly better cost, no observationally equivalent solutions will be explored by the algorithm and pruning on it becomes somewhat useless. Then again, logically, allowing worse solutions to explored, with the only goal being to prune them later in the algorithm does not seem like a useful strategy.

However, pruning in combination with a best improvement tactic that does not discriminate between worse or better neighbors sounds useful. Pruning an observationally equivalent program and only counting those steps in which the neighbor was not pruned could make the steps more meaningful. This combination however did not yield any significant improvements in the experiments, which could be explained by that pruning still remains relatively rare, thus preventing any real improvement.

5.6 Stochasticity

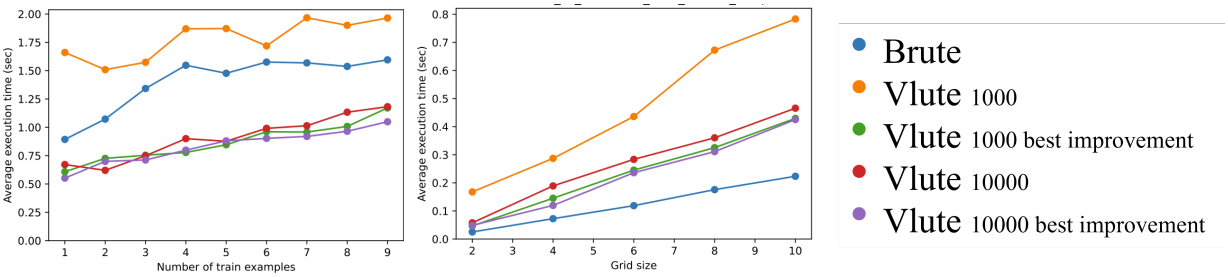
Accepting non-improving solutions by means of a stochastic accept function significantly improved the performance of Vlute in the robot-planning domain. It roughly halves the execution time, as seen in figure 8a, while receding no performance in accuracy. Stochastic accept methods combined with high values for N_i even approach the performance of Brute in this domain. In the ASCII-art domain stochasticity results in nearly the same improvement. These results could be explained by the nature of the accept method, which performs exploration first and exploitation second.

Interestingly, the string-transformation domain shows a strong decrease in performance. Both the execution time and accuracy are much worse for a stochastic Vlute than for a deterministic Vlute. The results for the robot-planning and the string-transformation domain are shown in Figure 8. A possible cause of the poor performance in the string-transformation domain compared to the other two domains is the length of the resulting programs. In general, domains that require large programs as solutions benefit from stochastic program synthesis [30]. Further analysis of all three domains



(a) string-transformation domain (b) ASCII-art domain (c) Legend

Figure 6: Accuracy of Vlute with increasing and decreasing degrees of destruction in the string-transformation and ASCII-art domain



(a) string-transformation domain (b) robot-planning domain (c) Legend

Figure 7: Execution time of Vlute with best improvement tactic in the robot-planning and string-transformation domains

shows that program synthesis the string-transformation domain results in the smallest programs out of all domains, thus explaining why it benefits least from this adaptation on Vlute. This theory, however, contradicts the performance difference in the robot-planning and ASCII-art domain. That is to say, the ASCII-art domain results in the longest programs, indicating that its performance should be improved most of all three domains, which is not the case.

6 Conclusions and Future Work

The inductive program synthesis (IPS) system Vlute has shown in previous work to improve upon another IPS system, Brute, in the program synthesis task of string-transformations through the use of a Very Large Neighborhood Search (VLNS) with a Variable Depth Invent (VDI) stage. It has not shown any improvement in robot-planning or drawing ASCII-art. This research in turn improved upon Vlute through various tactics. Some improvement tactics even approach Brute's performance where Vlute could not. Of these tactics using a best improvement strategy for exploring neighbors improved the most on Vlute, by halving execution time while receding no accuracy. Including a stochastic accept method and exploring only loop-free programs also showed improvement in both execution time and accuracy in the robot-planning and ASCII-art domain. Overall most improvement was found in the robot-planning domain, and the worst deterioration was found in the string-transformation domain. My experiments thus show that, even though Vlute is an inventive way of improving on Brute, there are still tactics to explore that could decrease execution time and increase accuracy.

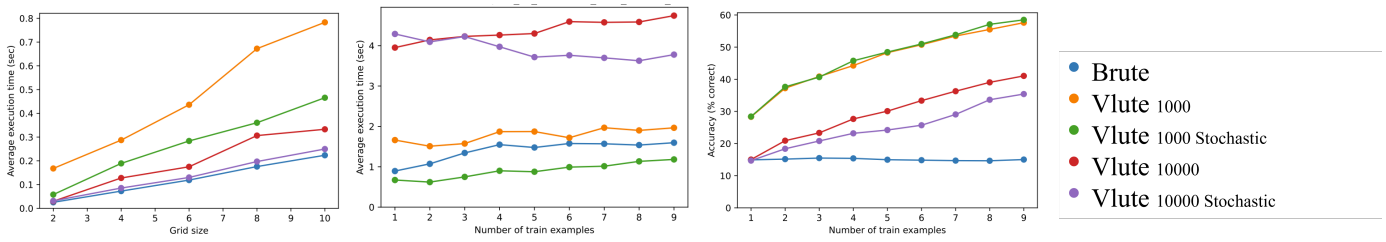
For future research, the parameters of each of the described tactics could be explored in further detail. Possibly linking the value of N_i to parameters in the best-improvement tactic, or any other, could lead to interesting results.

Also, while pruning on observational equivalence might lead to more interesting results with more suited search algorithms, other types of pruning might still be applicable to Vlute. Here, pruning algorithms that take into account the length of the program or the types of tokens might be more useful. Further interesting research could be conducted on the role of loop-free programs in the execution time of Vlute. The relation between the spread of the Vlute algorithms with different N_i values and the weight of the *if*-token might also yield interesting results.

7 Responsible Research

7.1 Data usage

The data has been received from the supervisor of the project, and has been used by [1] and [24]. However, the data has not been checked for bias or correctness because of time constraints on this project. The data for the robot-planning domain and the ASCII-art domain were generated randomly, so the test cases for these domains can be considered to be



(a) Execution time robot-planning domain

(b) Execution time string-transformation domain

(c) Accuracy time string-transformation domain

(d) Legend

Figure 8: Performance of Vlute with stochastic accept method in the robot-planning and string-transformation domains

mostly unbiased. The real-world string transformations, however, were taken from and used in previous research, where part of the data seems to originate from online Microsoft Excel user forums, and other data was handcrafted [8; 18; 10]. The re-use of the data throughout previous research suggests some reliability, but cannot ensure it.

The ethical aspect of the data can be categorized as not at risk of being manipulated with malicious intent. This is because there is no privacy sensitive data stored. Furthermore, the stored data has also not been published along this research, and is therefore not easy to access unless someone is authorized to do so.

7.2 Reproducibility

Reproducibility of the research is subject to the hardware on which the code is run. The hardware setup can be reproduced with relatively high certainty, as the TU Delft supercomputer (DHPC) provides an unchanging environment for the experiments to be run in. A cause for different results in some of the experiments however can be the stochasticity of some of the experiments. The simulated annealing acceptance strategy that replaces a deterministic strategy in some experiments may have different outcomes in several runs.

Another flaw in the reproducibility of the research is the code itself. As the code has not been produced in a release form, meaning that no definitively 'correct' version has been made, it is hard for a random user to find the exact configurations in which the code was run for the experiments. In general structure and implementation the released code conforms to the setup used for the experiments, however variables used in the experiments have to be changed by the user in order reproduce the experiments. This is subject to human error, and allows for mistakes in reproductions.

7.3 Credibility

The credibility of the results can be questioned, as the domains on which the experiments are tested are limited. However, since the goal of the research was to improve upon Vlute, no option for the expansion of the domains or additional domains was possible. So, as far as possible, the results are credible.

References

[1] S. Dumancic A. Cropper. Learning large logic programs by going beyond entailment. 2020.

[2] B. Abbasi, S. T. A. Niaki, M. A. Khalife, and Y. Faize. A hybrid variable neighborhood search and simulated annealing algorithm to estimate the three parameters of the weibull distribution. *Expert Systems with Applications*, 38(1):700–708, 2011.

[3] R. K. Ahuja, O. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[4] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61:84–93, 11 2018.

[5] P. Borges, T. Eid, and E. Bergseng. Applying simulated annealing using different methods for the neighborhood search in forest planning problems. *European Journal of Operational Research*, 233(3):700–710, 2014.

[6] D. Kroening C. David. Program synthesis: challenges and opportunities. 2017.

[7] K. D. Cooper and L. Torczon. Chapter 8 - introduction to optimization. In Keith D. Cooper and Linda Torczon, editors, *Engineering a Compiler (Second Edition)*, pages 405–474. Morgan Kaufmann, Boston, second edition edition, 2012.

[8] A. Cropper. Playgol: Learning programs through play. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 6074–6080. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[9] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. volume 46, pages 317–330, 01 2011.

[11] S. Gulwani. Applications of program synthesis to end-user programming and intelligent tutoring systems. 2014.

[12] S. Gulwani, Jha S, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 62–73. Association for Computing Machinery, 2011.

[13] S. Shoham E. Yahav H. Peleg, S. Itzhaky. Programming by predicates: a formal model for interactive synthesis. *Acta Informatica*, 57, 2020.

[14] S. Jha, S. Gulwani, A. S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224, 2010.

[15] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. page 215–224, New York, NY, USA, 2010. Association for Computing Machinery.

[16] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation. part i, graph partitioning. *Operations Research*, 37:865–892, 12 1989.

[17] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science (New York, N.Y.)*, 220:671–80, 06 1983.

[18] D. Lin, E. Dechter, K. Ellis, J. Tenenbaum, and S. Muggleton. Bias reformulation for one-shot function induction. *Frontiers in Artificial Intelligence and Applications*, 263:525–530, 01 2014.

[19] R. Lutz. Adaptive large neighborhood search. 2015.

[20] R. Todosijević S. Hanafi P. Hansen, N. Mladenovic. Variable neighborhood search: basics and variants. *EURO Journal on Computational Optimization*, 5, 08 2016.

[21] H. Peleg and N. Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. In *ECOOP*, 2020.

[22] D. Pisinger and S. Ropke. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40:455–472, 11 2006.

[23] D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers Operations Research*, 34(8):2403–2435, 2007.

[24] S. Rasing. Improving inductive program synthesis by using very large neighborhood search and variable-depth neighborhood search. 2021.

[25] P. Román-Román and F. Torres-Ruiz. A stochastic model related to the richards-type growth curve. estimation by means of simulated annealing and variable neighborhood search. *Applied Mathematics and Computation*, 266:579–598, 2015.

[26] D. Pisinger S. Ropke. Large neighborhood search. pages 399–419, 2010.

[27] E. Segredo, E. Lalla-Ruiz, E. Hart, and S. Voß. A similarity-based neighbourhood search for enhancing the balance exploration–exploitation of differential evolution. *Computers Operations Research*, 117:104871, 2020.

[28] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher and J. Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[29] S. Tupailo. General methods in inductive program synthesis. 1996.

[30] Y. Yuan and W. Banzhaf. Iterative genetic improvement: Scaling stochastic program synthesis, 2022.