Improving Inductive Program Synthesis by using Very Large Neighborhood Search and Variable-Depth Neighborhood Search

Bachelor Thesis

Stef Rasing

January 2022

CSE3000: Research Project
EEMCS

Improving Inductive Program Synthesis by using Very Large Neighborhood Search and Variable-Depth Neighborhood Search

Stef Rasing
Supervisor: Sebastijan Dumanĉić
EEMCS, Delft University of Technology, The Netherlands

Abstract

Brute, A state-of-the-art inductive program synthesis (IPS) system, introduced a two-phase algorithm; first, complex program instructions are invented from basic instructions. Second, a best-first search algorithm finds a sequence of invented instructions to solve an IPS task. This method is limited because invented instructions are always of the same complexity, also when less or more complexity is needed. Also, best-first search falls into local optima easily. In this paper, I describe Vlute, an IPS system using Large Neighborhood Search (LNS), in which a solution is gradually improved by exploring neighboring solutions, and Variable-Depth Invent (VDI), in which instruction complexity is increased dynamically. Vlute is tested on three IPS domains (robot-planning, string transformations, and drawing ASCII-art). Results show that using VDI improves Vlute's performance only for string transformation. Vlute can outperform Brute and escape local optima encountered by Brute also only for string transformation. A limitation of Vlute is finding large programs.

1 Introduction

Program synthesis is the automation of writing programs [5]. In a program synthesis task, constraints of some form that define how a program should behave are given by the user. The task in program synthesis is finding what program satisfies those constraints. Inductive program synthesis (IPS) is a type of program synthesis where a program should be synthesized using input and output examples [7]. Program synthesis is considered the holy grail of artificial intelligence [5]. It has powerful applications in fields like software engineering, computer-aided education, and end-user programming. One can imagine more impactful applications of program synthesis if it gets developed into a more powerful tool.

However, program synthesis is not a simple task. Difficulties come from two aspects of the problems: the size of the search space and user intent [5]. Since there are an infinite number of programs that can be created, the search space is infinite as well. Moreover, the search space grows exponentially with the size of the program. Therefore it is difficult to solve tasks that require large programs. The second challenge is finding a program that not only satisfies the given constraints, but also the intent of the user. A lot of different programs will satisfy the given constraints but might be overfitted to those constraints and not adhere to the users' intent.

In [4] Cropper and Dumanĉić argued that a key limitation of existing IPS systems is that entailment is used to guide the search. Using entailment limits the potential since it is a binary decision. A program entails an example or it does not; intermediate states are non-existent. Therefore they introduced an example-dependent loss function depending on the domain of the task. To evaluate a program, the program's output, given an input example, is compared to the desired output. This gives a distance measure, or cost, of the program, which is used to guide the search.

Brute, designed by Cropper and Dumanĉić, is a state-of-the-art system designed to solve IPS problems using an example-dependent loss function [4]. One of Brute's intents is to learn larger programs than other state-of-the-art IPS systems could. Simply stated, Brute works in two phases; an invent stage, where more complex program instructions are created out of basic ones. The second stage, the search stage, is a best-first search, where a sequence of the previously invented instructions needs to be found.

By using an invent stage, the search problem is reduced. Without an invent stage instructions can be nested infinitely deep (think about if and loop statements in programming languages). By using an invent stage, the branching factor of possible programs is reduced and the search space is significantly smaller. To demonstrate the power of this concept, Brute uses a greedy, best-first search and still shows promising results. However, a different search technique is desired because a best-first search is prone to get stuck in local optima.

This research *aims* to explore the use of a different search metaheuristic: Very Large Neighborhood Search (VLNS). Metaheuristics are used to search combinatorial space and find a decent solution within a reasonable time [2]. In short, VLNS is a class of improvement algorithms, where each algorithm improves a feasible solution by exploring solutions in a "very large" neighborhood around it [1]. A metaheuristic belonging to this class is Large Neighborhood Search (LNS) [6]. LNS explores neighbors that can be reached by partially destroying the best-found solution and thereafter repairing it. A subclass of VLNS is Variable-Depth Neighborhood Search (VDNS) [6]. The idea behind VDNS is to extend the neighborhood from which neighbors can be found whenever the search fails to yield improving results. This research explores using a combination of VDNS and the invent stage of Brute, called Variable-Depth Invent (VDI). Initially in the search less complex tokens are invented, but if the search cannot find a better program, more complex tokens will be invented. In this paper, I describe Vlute, an IPS system that combines VDI with a simple LNS algorithm.

I claim that Vlute can perform better in certain IPS domains than Brute. To do so, I have devised the following research questions that will be answered in this paper:

Q1. Can Vlute, with VDI, outperform Vlute without VDI, both guided by an example-dependent loss function?

Q2. Can Vlute outperform Brute, both guided by an example-dependent loss function?

Q3. Can Vlute, guided by an example-dependent loss function, solve problems where Brute encountered local optima?

To answer these questions and support the claim, the following contributions are described in this paper:

- An implementation of VDI and Vlute is described. This technique combines the invent stage from Brute with VDNS while using LNS to search for a solution.
- Vlute is evaluated on three different IPS domains. Vlute will be compared to a version of itself without VDI and Brute. Results show that Vlute outperforms Brute in one of the tested domains; real-world string transformations.

2 Related Work

Brute

As stated before, Brute is a state-of-the-art IPS system [4]. Brute works in two stages: invent and search. In the **invent** stage, basic program statements are combined to create more complex statements. These complex statements are used in the search stage, where a sequence of complex statements needs to be found. Since there is an infinite number of complex statements that can be created, Brute follows a common convention to limit the number of complex statements created. The effectiveness of the invent stage is based on the assumption that most IPS challenges are solvable using a sequence of these generated complex statements, which significantly reduces the search space.

Brute uses a best-first **search**. The search initially starts out by only having discovered a program containing no statements. For every statement, created in the invent stage, a new program is created by appending that statement to the best-discovered, lowest cost, program. It then computes the cost and adds the newly discovered program to the set of discovered programs. This process is repeated until a zero-cost solution has been found or another termination condition is reached.

Very Large Neighborhood Search

Very Large Neighborhood Search (VLNS) is a class of improvement algorithms. An improvement algorithm in general starts with a feasible solution and tries to improve this solution each iteration [1]. Neighborhood search, also called local search, is a class of improvement algorithms where each iteration it tries to find a better solution by exploring neighbors in a neighborhood around the current solution. How a neighborhood is defined differs widely among implementations. VLNS is the subclass where the neighborhood is considered to be ”very large”.

A subclass of VLNS is Variable-Depth Neighborhood Search (VDNS) [6]. The idea of VDNS is to gradually extend the neighborhood as the search progresses. Whenever the search gets trapped in a local optimum, the size of the explorable neighborhood is increased and the search can possibly escape the local optimum.

Large Neighborhood Search

A more concrete VLNS algorithm is Large Neighborhood Search (LNS) [6]. In LNS, neighbors are reached by first partially destroying a solution and thereafter repairing it. Usually, the destroy method contains randomness or stochasticity to make sure that every part of a solution can be destroyed. How exactly the destroy and repair methods work, is specific to a problem and how one designs both methods.

Adaptive Large Neighborhood Search is an extension of the LNS algorithm; during the search different destroy and repair methods can be used. Selection of the destroy and repair method is according to weights, which will be updated after every iteration.

3 Methodology

Program syntax

A program is a sequence of tokens. When a program interprets an input environment, every token in its sequence is applied to the input environment sequentially. These tokens can consist of three types: *transition*, *boolean*, and *control* tokens. **Transition tokens** are tokens that directly alter a given state. When this token is applied to a state, a new state will be returned. **Boolean tokens** are tokens that when applied to a state always yield either true or false. Transition and boolean tokens are different for each IPS domain. **Control tokens** are tokens that determine the flow of a program. There are two control tokens: *If* and *Loop*. An *If* token takes in one boolean and two sequences of tokens. When applied, the boolean token is evaluated first. If that yields true, the first sequence is applied. Otherwise, the second sequence is applied. A *Loop* token takes in one boolean and one sequence of tokens. First, the boolean is evaluated. As long as the boolean token yields true, the sequence keeps being applied.

Vlute

Like Brute, Vlute has two components; invent and search. Below both components are detailed, accompanied by pseudocode and examples.

Invent

Similar to Brute, in the invent stage basic tokens are combined to create more complex ones. However, in Brute the library of invented tokens is static; for every search, the same invented tokens are created. This can be inefficient when a certain search does not require the complexity of the invented tokens and this method can be insufficient when a more complex token is required to solve the problem. Therefore, I introduce **Variable-Depth Invent** (**VDI**).

Variable-Depth Invent

The main idea behind VDI is to increase the complexity of the invented tokens if the search fails to yield improving results. Initially, the invented tokens are simple, simpler than the ones created by Brute. Whenever a certain condition indicating that the search is stuck in a local optimum is met, the depth of VDI is increased. New, more complex, tokens will be invented and are available to use in the search. For simplicity VDI only invents *Loop* or *If* tokens. Brute also invents sequences of transition tokens, however, the same sequences can also be found in the search stage.

But what exactly does depth or complexity mean in the context of invented tokens? I have identified two dimensions of depth for an invented token; the total number of transition tokens and the total number of control tokens. Depth can therefore be represented by a 2-tuple where the first entry is the number of transition tokens and the second the total number of control tokens. The lowest sensible depth is $(1, 1)$, where control tokens containing only a single transition token are invented. Because the number of tokens that can be invented explodes quickly when the depth is increased, I have limited it to three depth levels: $(1, 1)$, $(2, 1)$, and $(2, 2)$. For example, the following two tokens will be invented at depths $(1, 1)$ and $(2, 2)$ respectively:

Loop(IsNotSpace, [Drop])

Loop(NotAtEnd, [If(IsNumber, [MoveRight], [Drop])])

To search more efficiently, some pointless invented tokens are pruned. The following types of tokens are pruned:

- *If* tokens with negations as a condition; a version with the non-negated condition and flipped branches will be invented too, making this token redundant.
- *If* tokens with equal branches.
- *Loop* tokens with an empty body.

Random tokens can be retrieved from the VDI object. Weights can be set that determine the probability of an *If*, *Loop*, or *transition* token being returned. The weights are denoted by w_{if}, w_{loop}, and w_{trans} respectively.

Search

The core of the **search** stage is variable-depth LNS. As stated before, LNS gradually improves the best-found solution by exploring its neighbors. For depth variability, the search keeps track of how many iterations have passed since a new best solution was found. When this amount exceeds a threshold, the depth is increased.

Pseudocode for the search stage can be found in algorithm 1. Inputted is N_i, denoting the depth increase threshold. The search starts with an empty list as the best-found solution. Also, a counter, i_b, denoting the number of iterations since a new best-solution was found, is initialized to zero. Each iteration, a temporary solution is created by destroying and repairing the best-found solution. If this solution is a zero-cost solution, meaning it has solved the task, it will be returned.

```
Inputs: $N_i : int$
Output: $list$
$x^b = []$                         ▷ Init. best solution with empty list
$i_b = 0$                          ▷ Init. iterations since best found
repeat
    $x^t = r(d(x^b))$              ▷ Destroy and repair current solution

    if $c(x^t) = 0$ then           ▷ Return if solved
        return $x^t$

    if $c(x^t) < c(x^b)$ then      ▷ New best solution found
        $x^b = x^t$
        $i_b = 0$
    else
        $i_b = i_b + 1$

    if $i_b > N_i$ then            ▷ Increase search depth
        Invent.increase_depth()
        $i_b = 0$

until timeout
return $x^b$                       ▷ Return best solution
```

Algorithm 1: Variable-Depth Large Neighborhood Search.

If the temporary solution is better than the best-found solution, it will become the new best-found solution, and i_b is set to zero. If i_b exceeds the threshold N_i, the search depth will be increased. If no zero-cost solution was found in time, the best-found solution will be returned.

Destroy method: remove-N

The destroy method, called **remove-N**, is simple; split a sequence by removing N sequential tokens from it. The pseudocode can be found in algorithm 2. Inputted is an integer N_{max}, the maximum number of tokens that can be removed, and a list of tokens P, the program. Outputted is a 2-tuple containing the leading and trailing part of the program, split by the deletion of N sequential tokens.

When a sequence gets destroyed, first a random N, the number of tokens to be removed, is chosen (line 2). N cannot exceed N_{max} and the length of the program. Thereafter, a random I, the index from where tokens are destroyed is chosen (line 3). In line 5 two splices are returned: $P[: I]$ are the tokens before the destroyed ones and $P[I + N :]$ are the tokens after. For clarification, two examples are given below. For the first example, N was 0 and I was 2. In the second example, N was 2 and I was 0.

$$[t_1, t_2, t_3] \rightarrow [t_1, t_2], [t_3]$$
$$[t_1, t_2, t_3, t_4] \rightarrow [], [t_3, t_4]$$

Repair method: insert-N

The repaired method, **insert-N**, works the other way around; stitch two token sequences together while adding N tokens in the middle. Inputs are a 2-tuple D containing two token sequences, N_{max}, the maximum number of tokens that can be inserted, and I, a reference to an invent object from which random tokens can be retrieved.

In algorithm 3 pseudocode for insert-N can be found.

```
Inputs: $N_{max} : int, P : list$
Output: $(list, list)$
$N'_{max} \leftarrow min(N_{max}, |P|)$
$N \leftarrow$ random int from $[0, N'_{max}]$
$I \leftarrow$ random int from $[0, |P| - N]$

return $(P[: I], P[I + N :])$
```

Algorithm 2: Destroy method: Remove-N.

```
   Inputs: $D : (list, list), N_{max} : int, I : Invent$
   Output: $list$
1: $P = []$
2: $N =$ random int from $[0, N_{max}]$
3:
4: while $N > 0$ do
5:     $t = I$.random_token()
6:     $P = P + t$
7:     $N = N - 1$
8:
9: return $D[0] + P + D[1]$
```

Algorithm 3: Repair method: Insert-N.

On line 2 a random N is chosen. Then on lines 4 through 7, N random tokens will be inserted in a list, P. Random tokens are retrieved through some invent object. On line 9, the N random tokens are put in between the two input sequences and returned. To clarify, two examples are stated below. In the first example N was 0 and for the second example N was 2.

$$[], [t_3, t_4] \rightarrow [t_3, t_4]$$
$$[t_1, t_2], [t_3] \rightarrow [t_1, t_2, t_a, t_b, t_3]$$

4 Results

4.1 Experimental procedure

Question 1

To investigate if using variable-depth invent (VDI) yields better results than using static-depth invent, my experiments aim to answer the question:

Q1. Can Vlute, with VDI, outperform Vlute without VDI, both guided by an example-dependent loss function?

To answer this question two versions of Vlute are created; one using VDI and one using a static-depth invent stage. VDI increases depth if no better solution was found after N_i iterations. To fully explore VDI's potential, multiple values for N_i are tested. To easily denote the value of N_i, it is subscripted. For example, Vlute with $N_i = 1000$ is called Vlute$_{1000}$. Vlute without VDI is named Vlute$_{\infty}$ since its depth is never increased.

Question 2

To investigate if Vlute can outperform a state-of-the-art IPS system like Brute, my experiments try to answer the following question:

Q2. Can Vlute outperform Brute, both guided by an example-dependent loss function?

To answer this question, Vlute is compared against Brute. The same experiments can be used as before, only Brute also needs to be tested.

Question 3

A fallacy of Brute is that it falls into local optima easily. To investigate if Vlute can escape local optima encountered by Brute, my experiments aim to answer the following question:

Q3. Can Vlute, guided by an example-dependent loss function, escape local optima encountered by Brute?

To answer this question Brute is run on each test case. If Brute fails to solve the test case, two versions of Vlute are run: one normal Vlute with an empty program as the initial solution and one Vlute with Brute's best-found solution as the initial solution. The normal Vlute is called the same as before and the other version is called Vlute$_{BN_i}$ (Vlute$_{B1000}$ for example). Results of both versions of Vlute are compared and if they yield the same predictive accuracy, it means that Vlute was able to escape the local optima in which Brute fell.

Experimental settings

For each test case, the system has 10 seconds to solve the problem. The created IPS system has been evaluated on three domains: robot-planning, real-world string transformations, and drawing ASCII-art.

Vlute with VDI

Vlute with VDI is tested with the following values for N_i: 1000, 3000, 5000, 10000, 15000, and 30000. Only two N_is are shown in the results; the other values followed the same trend as one of the shown values. The following system parameters were used:

- Depth levels are $[(1, 1), (2, 1), (2, 2)]$.
- N_{max} for destroy and repair is 3.
- $w_{trans} = w_{loop} = 1$ and $w_{if} = 0$.

Vlute without VDI

For Vlute without VDI the following system parameters were used:

- N_{max} for destroy and repair is 3.
- $w_{trans} = w_{loop} = 1$ and $w_{if} = 0$.

In the invent stage *If* tokens are invented containing two transition tokens and *Loop* tokens are invented that contain two transition tokens and can contain and *If* token.

Brute

Brute will generate the following tokens during the invent stage:

- All sequences up to 3 tokens.
- All *If* tokens with 1 token in each branch.
- All *Loop* tokens with 1 token in the loop body.

4.2 Robot-planning

Materials

In the first domain, displayed in figure 1, a robot starts at a location in an n by n grid. It has to pick up a ball and deliver it at some location, whereafter the robot has to walk to

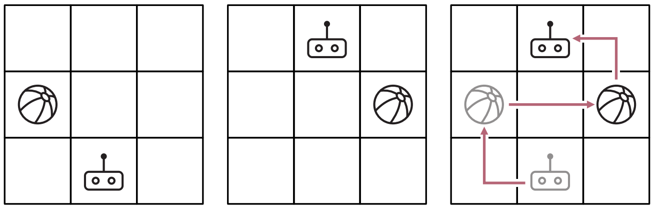

(a) Input state (b) Output state (c) Valid solution

Figure 1: Robot-planning domain visualisation.

$$d = \begin{cases} D_{r,b} + D_{b,b'} + D_{b',r'} + 2 & \text{if } D_{r,b} > 0 \wedge D_{b,b'} > 0 \\ D_{r,b'} + D_{b',r'} + 1 & \text{if } D_{b,b'} > 0 \\ D_{r,r'} & \text{otherwise} \end{cases}$$

Figure 2: Robot-planning distance heuristic.

some final location. The input state (figure 1a) is the start location of the robot and the ball and the output state (figure 1b) is the end location of the robot and the ball. Transition tokens for this domain are *MoveUp*, *MoveRight*, *MoveDown*, *MoveLeft*, *GrabBall*, and *DropBall*. For boolean tokens, *AtTop*, *AtRight*, *AtBottom*, *AtLeft*, and their negations are available. The IPS system has these tokens available together with the *If* and *Loop* tokens and has to devise a program that solves the problem (figure 1c). The used heuristic for this domain is the number of actions the robot needs to take to complete the goal (walking, pickup ball, drop ball). This is given by the formula in figure 2. Inputted are r, r', b, and b' which are the robot's position, the robot's goal position, the ball's position, and the ball's goal position respectively. $D_{x,y}$ means the Manhattan distance between positions x and y.

Method

Example test cases are from a set of generated cases with five different world sizes, n, where n is from $\{2, 4, 6, 8, 10\}$ [4]. For each world size 110, test cases were generated. Measured are the predictive accuracy (percentage of tasks solved) and average execution time, both per n. For both, the average is plotted and the 90% confidence interval.

Results

The used heuristic in this domain is almost equal to the number of single transition tokens needed to complete the tasks. This makes solving the tasks easy; no local optima exist. Figure 3b confirms this; all tested IPS systems solve robot-planning with 100% accuracy. However, looking at figure 3c, execution time is not equal for all systems. Brute performs the best, while Vlute$_{1000}$ has a longer execution time. A possible explanation is that VDI depth is increased too fast and the search has to deal with more complicated tokens. The not-shown Vlutes follow the trend as Vlute$_{10000}$, which has an execution time between Brute and Vlute$_{1000}$.

An example program found by Vlute$_{5000}$ can be found in figure 4. The program that has been found is stated in figure 4a and a corresponding visualization of the path taken can be found in figure 4b.

4.3 Real-world string transformations

Materials

In the second domain, shown in figure 5, an input string needs to be transformed into an output string. The input state is the input string and the position of a pointer, which points to a certain position in the string. The output state is the output string, the position of the pointer does not matter. Transition tokens act on one character; the one at the position of the pointer. Transition tokens for this domain are *MoveRight*, *MoveLeft*, *MakeUppercase*, *MakeLowercase*, and *DropCharacter*. It has *AtEnd*, *AtStart*, *IsUppercase*, *IsLowercase*, *IsLetter*, *IsNumber*, *IsSpace*, and their negations as boolean tokens. The distance heuristic used for this domain is the Levenshtein distance between the produced output and the desired output. Levenshtein distance is the number of changes (insertions, deletions, substitutions) that need to be made to transform one string into another.

Method

327 real-world string transformation tasks are taken from [3], where each task has 10 in-/output examples. For each task and n from $\{1, 2, ..., 9\}$, n in-/output examples are taken as training data. The other $10-n$ examples are used as test data, together they create one test case. Each test case is repeated 10 times with different train examples. Measured are the predictive accuracy (percentage of test cases where *all* examples are solved) and average execution time of solved tasks, both per n. For both the 90% confidence interval has been plotted. Another plot that will be created, is a scatter plot in which per n a dot is plotted with the corresponding predictive accuracy and average execution time as coordinates. In such a plot, one can compare the number of input examples needed to reach a certain accuracy.

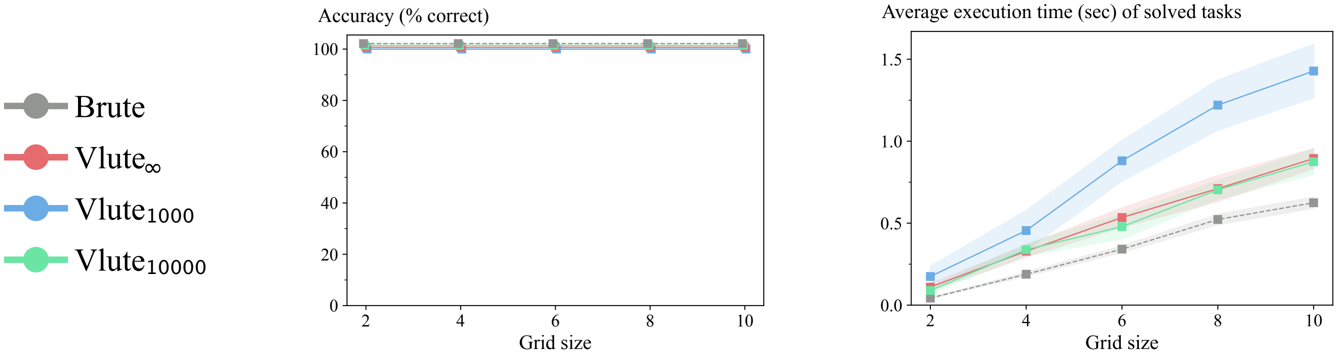
Results

Figure 6b shows that Vlute has a higher accuracy than Brute for each number of examples given. Vlute$_{10000}$ has a better performance than Vlute$_{\infty}$. All other Vlutes are almost equal to Vlute$_{1000}$, which is still a bit better than Vlute$_{\infty}$. This confirms the belief that VDI can improve performance over not using VDI.

Figure 6c shows that Brute still has the shortest execution time when the number of given examples is low. It can also be seen that Vlute$_{10000}$ has the shortest execution time of all Vlutes, shorter than Vlute$_{\infty}$, indicating that using VDI can also improve execution time. All not-shown Vlutes have execution times between that of Vlute$_{10000}$ and Vlute$_{\infty}$. However, Vlute$_{1000}$ has a longer execution time than Vlute$_{\infty}$. As before, a possible explanation is that the search depth is increased too fast and the search has to deal with more complicated tokens.

In figure 6d it can be seen that Vlute$_{10000}$ needs fewer training examples than Vlute$_{1000}$ and Vlute$_{\infty}$ to reach the same accuracy. For example, to reach an accuracy of 50% Vlute$_{10000}$ needs 2 examples, while Vlute$_{1000}$ and Vlute$_{\infty}$ need 3 and 5 respectively.

An example program that Vlute$_{10000}$ found can be found in figure 7. The task is displayed in figure 7a and the found program can be seen in figure 7b.



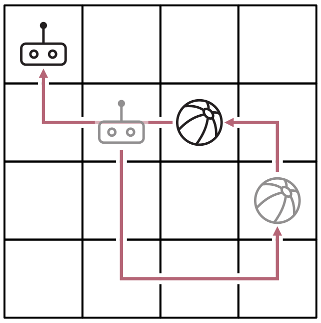
(a) Legend

(b) Predictive accuracy vs. grid size

(c) Average execution time vs. grid size

Figure 3: Results of robot-planning experiments.

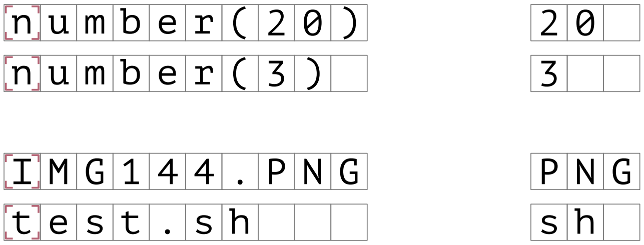
1. Loop(NotBottom, [Down])
2. Loop(NotRight, [Right])
3. Up
4. Grab
5. Up
6. Left
7. Drop
8. Left
9. Left
10. Up



(a) Found solution

(b) Solution visualisation

Figure 4: Program found by Vlute$_{5000}$ solving a robot-planning task.



(a) Input states

(b) Outputs

Figure 5: Visualisation of the real-world string transformation domain showing two example test cases.

4.4 Drawing ASCII-art

Materials

For the third and last domain, shown in figure 8, the objective is to draw pixel art where each pixel can either be colored or not. The input state is a blank canvas of h by w pixels accompanied by the location of the pointer. The output state is a pixelated canvas, the position of the pointer does not matter. The pointer denotes the pixel on which transition tokens can act. Available transition tokens are *MoveUp*, *MoveRight*, *MoveDown*, *MoveLeft*, and *Draw*. Available booleans tokens are *AtTop*, *AtRight*, *AtBottom*, *AtLeft*, and their negations. In this domain, the used heuristic is the hamming distance (number of incorrect pixels) between the created output and the desired output.

Method

ASCII strings are from a dataset in which strings with n characters were generated, where n is from $1, 2, ..., 5$ [4]. For each n a total of 100 tasks are generated. Characters are sampled uniformly at random with replacement and text2art was used to transform the string into pixels. Each character is 4 pixels wide and 6 pixels high. The input environment is an empty matrix. Measured are the predictive accuracy and the average execution time, both per n.

Results

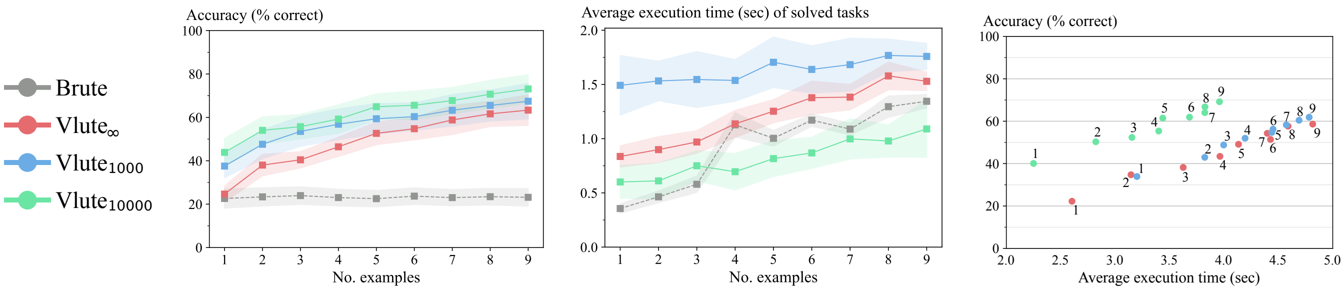
Figure 9b shows that Brute outperforms all Vlutes. Between the versions of Vlute, no significant difference can be found. This domain requires large programs, especially when the number of characters is higher. As expected Brute performs fine for large programs, but Vlute struggles to find large programs. At 2 characters almost no problems are solved anymore by Vlute, while Brute can solve almost all problems with 2 characters. Figure 9c shows that for the cases that are solved, Brute has a lower execution time than all Vlutes. Between the versions of Vlute, there is again no significant difference in execution time.

4.5 Escaping local optima

Performance of the two different versions of Vlute described before needs to be compared for cases where Brute failed to find a zero-cost solution. Since Brute solved all robot-planning cases and Vlute could not solve many cases for drawing ASCII-art, the different versions will not be compared in these domains, only in the string transformation domain.

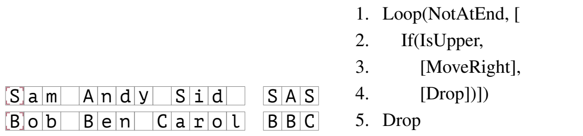
Results

Figure 10 shows that performance of both versions are similar. The difference never gets larger than 10%, indicating that Vlute is able to escape some local optima, but not all. A possible explanation is that when Brute is trapped in a local optima, it keeps adding tokens that do nothing, creating



(a) Legend (b) Accuracy vs no. examples (c) Execution time vs. no. examples (d) Accuracy and execution time

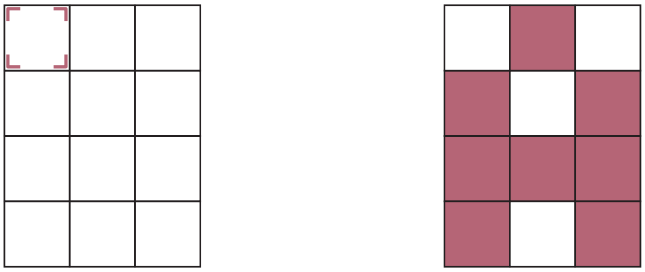
Figure 6: Results of real-world string transformation experiments.



```
1. Loop(NotAtEnd, [
2.     If(IsUpper,
3.         [MoveRight],
4.         [Drop])])
5. Drop
```

(a) Test case visualisation (b) Found solution

Figure 7: Solution found by Vlute$_{10000}$ solving a string transformation task.



(a) Input state (b) Output state

Figure 8: Visualisation of the drawing ASCII-art domain.

an extremely large program. For Vlute it can be hard dealing with large programs, since there are more ways to reach neighbors, making it harder finding a good one.

5 Discussion and Limitations

Question 1

From the results, the conclusion can be drawn that using VDI only significantly improves results in the real-world string transformation domain. In this domain, each Vlute version with VDI outperforms Vlute without VDI based on accuracy. Execution time is also shorter or equal for almost every Vlute using VDI, only Vlute$_{1000}$ has a longer execution time than Vlute without. However, VDI does not improve results as much for the other two domains. For robot-planning accuracy is 100% for each method. Execution times are similar for Vlute with VDI and without, only Vlute$_{1000}$ has a higher execution time, showing no improvement using VDI. In the drawing ASCII-art domain accuracy and execution time are similar for each Vlute version, showing no improvement in this domain either. A possible explanation for the fact that VDI only yields better results in the string transformation domain, is that more complex tokens are not that useful in the other domains. All robot-planning and drawing ASCII-art cases can be solved with a sequence of transition tokens. However, in the string transformation domain more complex tokens are often needed to solve the problem.

Question 2

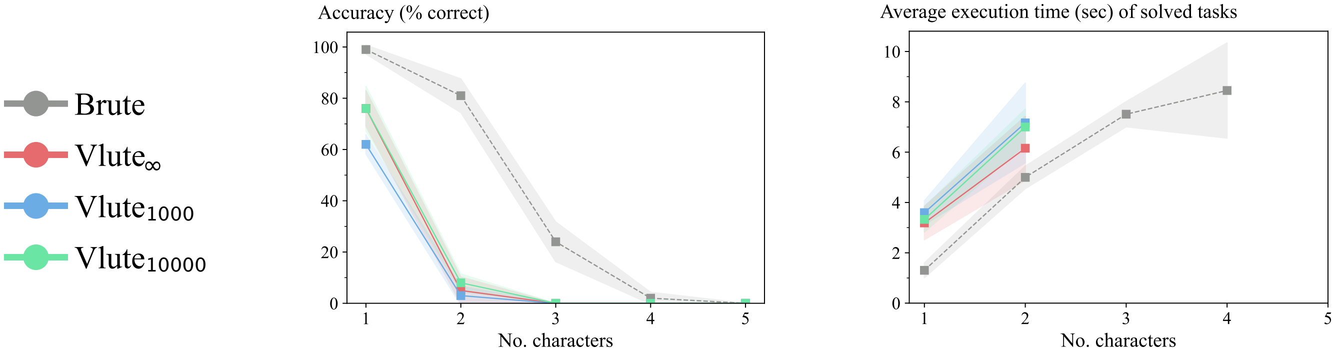
From the results, it can also be concluded that Vlute outperforms Brute only in the real-world string transformation domain. In the real-world string transformation domain, every tested Vlute method has higher accuracy. However, for the problems Brute solved, execution times were in general better than Vlute. Only one version of Vlute executed faster than Brute, but only for a larger number of examples. Also, Vlute needs significantly fewer training examples to reach the same accuracy as Brute. However, in the other two domains, Vlute does not outperform Brute. In robot-planning, all methods reach 100% accuracy, but Brute has a lower average execution time at each grid size. For drawing ASCII-art, Vlute does not perform well. Only when one character needs to be drawn can it solve the majority of the problems. Brute has a significantly higher accuracy for the first three matrix sizes while having a lower execution time than Vlute.

Question 3

The last conclusion that can be drawn, is that Vlute can escape some of the local optima encountered by Brute, but not all. Vlute that starts with Brute's best-found solution performs a bit worse than normal Vlute, but not much worse. This indicates that Vlute has escaped some of the local optima, but not all.

Limitations

A limitation of using Vlute in the manner described is finding large programs. For solving drawing ASCII-art examples, large programs are needed when the number of symbols is higher, which explains why results are not promising in that domain. 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.

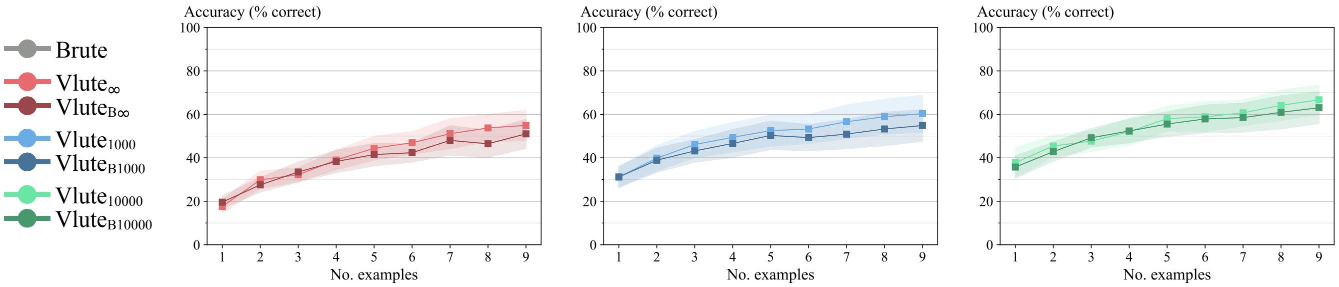


(a) Legend

(b) Predictive accuracy vs. no. characters

(c) Execution time vs. no. characters

Figure 9: Results of drawing ASCII-art experiments.



(a) Legend

(b) Predictive accuracy vs. no. examples for Vlute$_{\infty}$

(c) Predictive accuracy vs. no. examples for Vlute$_{1000}$

(d) Predictive accuracy vs. no. examples for Vlute$_{10000}$

Figure 10: Results of escaping local optima experiments.

6 Conclusion and Future Work

Conclusion

Brute has shown great potential for using an invent stage before searching for a solution. However, the search used in Brute is a best-first search, which is prone to falling into local optima. Therefore another search method was proposed, Very Large Neighborhood Search (VLNS), which can escape local optima as long as the searched neighborhood is large enough. Another proposed improvement is the use of Variable-Depth Invent (VDI), in which more complex tokens are invented if the search does not yield improving results with the currently available tokens. My experiments show that Vlute using VDI outperforms Vlute without VDI for real-world string transformations. For the other two domains, no improvement can be seen when using VDI. Experiments also show that Vlute outperforms Brute for real-world string transformations. The experiments show that even the worst-performing Vlute version outperforms Brute in this domain. However, for the other two domains, Brute outperforms Vlute. The last experiment shows that Vlute is able to escape some, but not all, of the local optima encountered by Brute in the real-world string transformation domain.

Future work

Variable-depth invent

The current implemented VDI is simple; almost all tokens of a certain depth can be used in the search and the depth is increased when no improving solution is found after a fixed amount of iterations. Something worth looking into is to automatically determine the usefulness of tokens. There will be a lot of tokens that are useless or do not make sense. The search can be sped-up if those tokens are not used in the search as often as others.

Another possible improvement is changing the depth increment criterion. Currently, this is after a fixed amount of iterations. However, knowing what value will yield the best results is hard and will most likely be different for each domain or test case. Using a different criterion might improve results.

Search

In the implemented LNS the destroy and repair methods are designed to be simple and efficient. Using more sophisticated destroy and repair methods might improve results. For example, alternating the current random repair with a best-first repair could exploit Brute's ability to find large programs, while still being able to escape local optima. Also, research can be done in using different VLNS algorithms.

7 Responsible Research

In this section concerns regarding reproducibility and credibility are stated. First, I will talk about the data sets; where did it come from, and is it to be trusted? Thereafter, reproducibility corners are stated. Last but not least, I will say something about the credibility of the conclusions drawn from my research.

Data sets

The data sets were obtained through my responsible professor, Sebastijan Dumanĉić. In Dumanĉićs paper it was stated that the data for robot-planning and drawing ASCII-art was generated randomly like described in this paper. String transformation data was obtained from another paper, also cited in this paper. The data sets used in this research are blindly trusted; no full check-up of the data sets was done. The data has been used by Andrew Cropper and Sebastijan Dumanĉić in their research, so it probably is valid data, but a check-up might not be a bad idea. String transformation data has another problem; is it representable data. The data is based on real-world string transformation examples, but I am not aware of how this data was obtained and what possible biases it could have. It could be the case that accuracy is different if data was collected from a different source.

Reproducibility

I have identified two concerns for the reproducibility of this research; availability of source code and the machine on which experiments are run. To make this research as reproducible as possible, the code source should be published. "data.4tu.nl" has been recommended to publish and cite source code. Another thing that could affect reproducibility is the machine on which the code is run. My experiments were run at the TU Delft's HPC (High Performance Cluster), which is a powerful cluster of computers. If the same experiments were run on a different machine the results could be different. For example, running a test case in which the maximum depth level needs to be reached can take very long. When such a test is run on a less powerful machine, it could be the case that the time limit is reached before the maximum depth level is reached.

Credibility

Important is to consider how credible the results and conclusions of this research are. A few credibility concerns are explained. Right now the system is only tested on three domains, but to fully explore its performance, it should be tested on more domains. Other interesting domains could be arithmetic, list transformations, software development, and many more. Another factor influencing credibility is the available syntax. Currently, the syntax is limited, especially for string transformations. For example, no characters can be added. However, some test cases need characters addition to be solved and are therefore unsolvable right now. Choosing available syntax can influence results; with more instructions more problems are solvable, but searching for a solution takes longer.

At last I want to thank my peers: Farhad Azimzade, Bas Jenneboer, Nadia Matulewicz, Victor van Wieringen. Together we created a codebase from which we all could diverge into our research.

References

[1] Ravindra K Ahuja, Ozlem Ergun, James B Orlin, and Abraham P Punnen. A survey of very large-scale neighborhood search techniques, 2002.

[2] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8:239–287, 2009.

[3] Andrew Cropper. Playgol: Learning programs through play, 2019.

[4] Andrew Cropper and Sebastijan Dumanĉić. Learning large logic programs by going beyond entailment, 2020.

[5] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, and Now Publishers. *Program synthesis*, volume 4. now Publishers Inc., 2017.

[6] D. Pisinger and S. Røpke. Large neighborhood search, 2010.

[7] Armando Solar-Lezama. Lecture 2: Introduction to inductive synthesis, 2018.