How to train your dragon: on the application of the Metropolis-Hastings algorithm for program synthesis

BO HOFSTEDE
Supervisor(s): SEBASTIJAN DUMANČIĆ, LEONARD VOLARIĆ HORVAT
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

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

This paper addresses the problem of Inductive Synthesis by analysing the Metropolis-Hastings stochastic search algorithm. The goal of Inductive Synthesis is to generate programs whose intended behaviour is established through the use of input and output examples. The Metropolis-Hastings algorithm searches the set of all possible programs and finds possible solutions. Our experiments show that while optimization can be done under certain conditions, it does not improve the algorithm's success rate in synthesizing programs on complex domains compared to more randomized but domain-specific approaches.

1 Introduction

Program Synthesis is a field in Computer Science where the goal is to create algorithms that generate other programs to solve a variety of problems. The types of programs they create are based on a variety of semantic and syntactic input requirements. This paper focuses on Inductive Synthesis (IS), where example input-output sequences are used to define those requirements. The problem is that there are an infinite amount of programs that can be created and the algorithm needs to be able to establish which programs would be suitable solutions. The input-output sequences allow the algorithm to decide whether a program matches the solution requirements. The difficulty, and the focus of this paper, is to efficiently search the set of all possible programs to find a suitable one for the problem. To be specific, this paper focuses on a single stochastic search method: Metropolis-Hastings (MH). Research suggests that the combination of IS and stochastic search methods can help generate larger more complex programs [12].

This paper builds on the recently released paper by van Wieringen [12], by using the same code base as a starting point. Van Wieringen's work and subsequently this paper focus on finding an algorithm once the user's intent has been established through the use of input-output examples. Van Wieringen focused on developing a version of the MH algorithm and comparing it to other stochastic search algorithms on three different domains with their own Domain Specific Language (DSL). The Metropolis-Hastings Algorithm stochastically explores the search space of all possible programs. At each step, it either improves the program or takes a non-improving step with a certain acceptance probability. Its focus was to compare the performance of a Metropolis-Hastings-based algorithm to a variety of other search techniques.

While Van Wieringen's work applied the MH algorithm to the domain of program synthesis, it was limited in the degree to which tried to optimize and analyse the algorithm because it focused on the comparison with other algorithms. The problem that this paper address is that there is a lack of clear, systematic analysis of the effects of the different components of the MH search algorithm, on the domain of IS. This provides insight into the limitations as well as the potential of the MH algorithm in the context of IS. As such, this paper explores the effects that the program mutations and acceptance probability function have on the ability to create working programs on various complex domains.

2 Background and Related Work

2.1 Program Synthesis

Program Synthesis is the field of research focused on the creation of algorithms that can generate programs to solve problems autonomously. They only require a minimal amount of semantic and/or syntactic constraints provided by a user that defines the task. This means that a user does not need significant technical knowledge of the problem. For example, a user without knowledge of how sorting algorithms work could still define an ordering of elements. If they were to pass that ordering to a Program Synthesis algorithm, it should be able to create a suitable sorting algorithm.

Gottschlich *et al.* defined *Three Pillars of Machine Programming*: Intention, Invention and Adaptation [3]. Intention describes the area of research dedicated to understanding the best way to convey a user's goals to an algorithm. In the previous example of sorting this would describe the method by which the user can describe the ordering of the algorithm. The invention covers the process of finding a suitable program to solve the problem once the objective is clear. How would the algorithm come up with a working and efficient program such as *QuickSort*? Finally, Adaptation concerns itself with the maintenance and evolution of the programs that were found to deal with changes in hardware or discovery of edge cases. If more elements are added to the ordering, then the algorithm should be able to adapt to these changes.

While this paper specifically focuses on the Invention part of the process, the Intention part of the algorithm is a prerequisite of the search for possible programs. In this paper, Intention will be defined by examples of inputs and their desired outputs (IO). The algorithms that use this method are referred to as *Inductive Synthesis* algorithms. To be specific these IO examples allow the algorithm to define a cost function that describes the degree of error between the output of the generated program and the requested output. For the sorting problem specifically, the minimal number of inversions to get to a sorted list would be a suitable cost function. The cost function that is used must be specific to the domain of the problem.

The Invention stage is conducted by considering an extensive search tree of all possible programs where the goal is to locate the optimal one for the given problem. It is the goal of the search algorithm to find that program within a reasonable amount of time. As the search tree is incredibly large a few measures can be taken to prune it. Similar to the cost function, the language used to write the programs generally is a simplified language specific to the domain, i.e a *Domain Specific Language* (DSL). A DSL is generally split into three types of tokens: *Transition* that change the program state, *Boolean* that return a bool based on the state, and Control tokens that affect the actions of a program, e.g. by adding a loop. These define the operations which can be carried out, the The DSL can also be used to put restrictions on the algorithm's ability to create recursion in the

Algorithm 1 BubbleSort in a DSL

```
[
  LoopWhile(NotSorted,
    [
      LoopWhile(GreaterThanRightNeighbour,
        [
          SwapNeighbours,
          MoveRight
        ]
      ),
      IfThen(AtEnd,
        [
          LoopWhile(NotAtStart,
          MoveLeft)
        ]
      )
    )
  ]
]
```

programs it writes, this cuts down significantly on the size of the search tree. The sorting example could have a DSL composed only of: *MoveLeft*, *MoveRight*, *SwapNeighbours*, *NotSorted*, *GreaterThanRightNeighbour*, *AtEnd*, *NotAtStart*, *LoopWhile*, and *IfThen* operations as well as a with which it would theoretically be able to create a BubbleSort program (see Algorithm 1).

Program Synthesis has been used to successfully recreate several noteworthy and fairly complex algorithms such as Karatsuba's Integer Multiplication algorithm, and Strassen's Matrix-Multiplication [11]. It has its applications in a variety of fields, e.g the reverse engineering of code [6], bit-string manipulation and providing programmers with feedback [10]. Microsoft Excel's FlashFill add-on was the first example of a commercially available Program Synthesis tool [4]. It allows people, without any particular background in the field of data wrangling, to systematically manipulate data by providing the add-on with example manipulations. Program Synthesis is still a field that has considerable room for research and potential commercial applications, hence this paper.

2.2 Metropolis-Hastings

The Metropolis-Hastings method is a Markov Chain Monte Carlo method where the goal is to approximate a distribution, π, which can be evaluated but not sampled from directly. As a Markov Chain based method, this algorithm is applied to time-series, i.e. sequential data, where the $(i+1)^{th}$ iteration is only dependent on the iteration before it: i. At the start of the algorithm the Markov-Chain, *K*, is empty. The Monte Carlo part means that at each iteration a random sample is drawn from a proposal distribution, J. It is then evaluated using π. If it is an improvement the algorithm accepts the sample and adds it to the Markov Chain. If it is not, then the algorithm might still accept the sample but only with a certain acceptance probability $A(x, y)$.

$$A(x,y) = \frac{\pi(y) * J(y,x)}{\pi(x) * J(x,y)} \tag{1}$$

where x is the current Markov Chain, and y is the Markov Chain which includes the new sample. This means that the Markov Chain can be defined as:

$$K(x,y) = J(x,y) * min(1, A(x,y)) \tag{2}$$

This acceptance probability allows the algorithm to escape local optima. It is defined in such a way that after a certain number of iterations, referred to as the *burn-in period*, the Markov-Chain will converge to the stationary distribution π. This is done by upholding the *condition of detailed balance*.

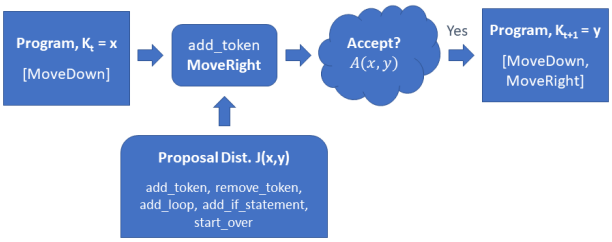

Figure 1: Example iteration of a MH Program Synthesis algorithm. An add_token mutation is selected from the proposal distribution that randomly selects a *MoveRight* token from the list of transition tokens.

As an example take the simple problem of a character in an empty grid world trying to move to the bottom of the grid as quickly as possible. The algorithm starts off with an empty program denoted as an empty list: $K_0 = []$. During the first iteration of the algorithm, an *add_token* mutation is randomly drawn based on a set of default probabilities from the proposal distribution $J(x, y)$. The *add_token* then randomly selects the transition token *MoveDown*. This reduces the cost, as the character gets closer to the bottom of the grid. Therefore the program is updated to $K_1 = [MoveDown]$. This process is then repeated for K_2. Again, an *add_token* mutation is drawn from $J(x, y)$, which this time results in a *MoveRight* transition token. This transition token does not reduce the cost and hence the acceptance function, $A(x, y)$ is used to determine the probability that it is still accepted. If it is then $K_2 = [MoveDown, MoveRight]$, otherwise it simply stays the same and the algorithm moves to the next iteration. Figure 4 shows this second iteration.

In 1953, Metropolis *et al.* published their new Monte Carlo method in the Journal of Chemical Physics [8]. The *Metropolis* method was created so that computers could more efficiently describe the properties of "substances consisting of interacting individual molecules". It relies on symmetric proposal distributions to determine whether a sampled variable is accepted, i.e. $J(x, y) = J(y, x)$. This general method was particularly useful for sampling from high dimensional distributions. In 1970, Hastings extended the method to permit the sampling of non-symmetric distributions, given that the forward and backward transition probabilities were known [5]. This method was thereafter known as the Metropolis-Hastings method.

As mentioned, this paper's algorithms are extended versions of the one presented by Van Wieringen [12]. His ap-

Mutation	Default Weights	Description
add_token_end (add_token)	10	Adds a random trans_token to the end of the program
remove_token_random (remove_token)	20	Removes a random trans_token from the program
add_loop_end (add_loop)	10	Adds a loop with a single bool_token as a condition and single trans_token to the end of the program
add_if_statement_end (add_if_statement)	10	Adds an if-else statement with a single bool_token and two trans_tokens to the end of the program
start_over	2	Return an empty program

Table 1: Mutations and weights used by van Wieringen [12]. Tokens parts of the DSL, where trans_tokens can specify an action and bool_tokens can specify a condition. Note: the original names have been enclosed in parentheses, while new names have been given in order to be more descriptive and avoid confusion with other mutations that are added in this paper, see Table 2.

Mutation	Description
add_token_random	Adds a trans_token at a random position in the program
remove_token_end	Removes a trans_token from the end of the program
add_loop_random	Adds a loop with a single bool_token as a condition and single trans_token at random position in the program
add_if_statement_random	Adds an if-else statement with a single bool_token two trans_tokens at a random position in the program

Table 2: New mutations that will be tested

proach to creating a program synthesis algorithm was to view the search tree of all possible programs as a Markov-chain, K, which starts off as an empty program, P_0. For simplicity, we assume the same notation as used by Van Wieringen. At each iteration of the stochastic algorithm, the algorithm makes a random but weighted selection from the proposal distribution, i.e. the set of possible mutations which can be applied to the last program on the Markov Chain. See Table 1 for details on the mutations used by Van Wieringen. Note that the probability of selecting a mutation can change over time based on a cost function specific to the problem domain. The mutation is then applied to the program, P', to create a new program, P. This is repeated till the best program is found. This, the set of possible mutations and their weights are the first independent variable that is analysed in this paper.

Once created, a new program is evaluated using a domain-specific cost function $\pi(P)$, which runs P on a list of input examples and compares its outputs with the corresponding output examples. If it is an improvement then the algorithm goes to the next iteration, otherwise, it accepts with an acceptance probability, $A(P, P')$. Van Wieringen used the method outlined by Metropolis

$$A(x, y) = \frac{\pi(y)}{\pi(x)} \tag{3}$$

Other variations, including the main MH method, rely on other factors to change this acceptance function. E.g. MH uses multiplies the right-hand side by the ratio of the forward-transition, $J(y, x)$ to the backwards-transition, $J(x, y)$. Where J(y, x) is the probability of obtaining y starting from x. Assuming that there is a significant discrepancy between $J(y, x)$ and $J(x, y)$ then this term would become necessary for the Markov Chain to eventually reach a stationary, optimal distribution/program. The choice of acceptance probability function is the second independent variable that is analysed in this paper.

2.3 Application: Metropolis-Hastings & Program Synthesis

Schkufza *et al.* applied the MH algorithm to the domain of loop-free program synthesis specifically with the goal of optimizing binaries in terms of execution time [9]. The binaries produced by their algorithm were generally at least as good as those of state-of-the-art compilers as well as human experts. This highlights the importance of taking the performance of the synthesised programs into consideration when comparing a MH-based Program Synthesis algorithm.

Van Wieringen describes his process of applying the Metropolis method to the problem of Program Synthesis with a focus on its ability to generalise and its success ratio on three different domains [12]. In particular, he compares his results with other types of stochastic search algorithms developed by his peers and supervisor, Dumančić whose own paper served as a basis for Van Wieringen's work. Cropper and Dumančić's paper focused on creating larger programs [1]. Their algorithm *Brute*, a best-first search algorithm, managed to synthesise programs that were "20 times larger than (previous) state-of-the-art systems". The work of Van Wieringen and subsequently this paper is follow-up research on the work of Cropper and Dumančić. We maintain the three domains first outlined in their paper, for details see Section **??**, and the use of an example-based loss-function.

3 Methodology

Note: to evaluate each variable, this paper will refer to their effects on the algorithm's ability to solve problems in complex domains. Additionally, the execution time and program length of the generated programs are taken into consideration. As such when there is a reference to the search algorithm's performance, what the best variation is, etc. it will refer to these criteria. See Section 4 for details on the specific metrics.

3.1 Mutations

As mentioned, the first component of the MH algorithm that can be altered is the selection of mutations. Notable in van Wieringen's work was the absence of a rationale for his choice of mutations, especially with regards to where in the

program the mutation would take place. Hence, we will examine the effect of applying the mutations at the end of the program or in a random position. To accommodate for this, new mutation functions are used, see Table 2. This also has an impact on the possibilities for acceptance functions. Phrased as a question: is it better to apply mutations at random locations in the program or at the end?

The second part of the mutations that can be altered is the default weights they are given. Van Wieringen's work alluded to the selection of weights, but no systematic process was discussed to analyse the effects of these weights. No clear rationale was given other than that his selection lead to approximately the same forward- and backwards-transition functions so that he could use the simpler Metropolis method. Hence the questions: 1) Does the configuration of default mutation weights have a significant impact on the performance of the algorithm? 2) What is the best performance that can be achieved by the selection of the default mutation weights?

3.2 Acceptance Function, $A(x, y)$

The first part of the acceptance function that can be adjusted is the cost function π. A problem that arises with IS is that a cost function inherently will inherently have a range from 0 to positive infinity as it describes an absolute error. This means that it has to be normalised to the range of [0, 1]. Van Wieringen proposed the following normalization method:

$$\pi(x) = e^{-\alpha * Cost(x)} \tag{4}$$

which ensures that the cost is normalised and that the higher the cost the lower the value for π. Recall that if a sample has a higher cost then, rather than immediately accepting, the acceptance function will be used. In that situation, one would expect a lower numerator than the denominator for the ratio to be between 0 and 1. In the equation, α is the discount factor. This raises two questions: 1) Does the discount factor $alpha$ have a significant impact on the performance of the algorithm? 2) What is the optimal value of the discount factor α?

The second part of the acceptance function that can be varied is the function itself. In his paper, Van Wieringen used the simpler Metropolis method rather than the more general MH method. As he discussed, the MH methods acceptance function requires the definition of explicit inverses. He noted two problems with this approach. First, he mentions that some mutations' inverses may be difficult to define. Second, there may be multiple paths that lead to the same result. The here proposed hypothesis is that both of these issues can be largely addressed by changing the type of mutations made. To be specific, they can be resolved by fixing the location in the program of where the mutation. Hence, we use the full MH method by only applying mutations to the end of the program, for details on the mutations see 3.1.

Van Wieringen's solution to this was to assume that $J(x, y) \approx J(y, x)$, though no statement was made as to how that resulted in the provided default weights for his mutations. This same logic can be applied to inverses, it can be hypothesised that within a certain margin of error the algorithm will still be able to get to the stationary distribution. This also resolves the "start_over" problem, where it is difficult to define an inverse for the "start_over" operation. As a "start_over" never yields an improvement in the program, it serves as a mechanism to encourage exploration, and an assumption can be made that with equal forward- and backward transition probabilities it would fit within the hypothetical margin of error. As such the final question arises: is there a benefit to using the full Metropolis-Hastings acceptance function over the one of the simpler Metropolis method?

4 Experimental Setup

4.1 Domain 1: Robot

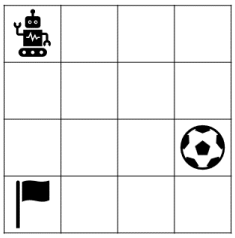

Figure 2: Example problem on the robot domain. Task: move the robot to the ball, pick it up and drop it at the flag.

The first domain, *Robot*, is the task of creating programs for a robot on a 2-dimensional grid world to find a ball and bring it to a goal flag. Note that there are no obstacles in the world. The DSL consists of the following actions: *MoveRight, MoveDown, MoveLeft, MoveUp, Drop, Grab*, and condition checks: *AtTop, AtBottom, AtLeft, AtRight, NotAtTop, NotAtBottom, NotAtLeft, NotAtRight*. The number of steps taken by the robot is used as a cost function to evaluate the programs. This domain contains 550 input-output examples to test the algorithm's performance.

4.2 Domain 2: Pixel

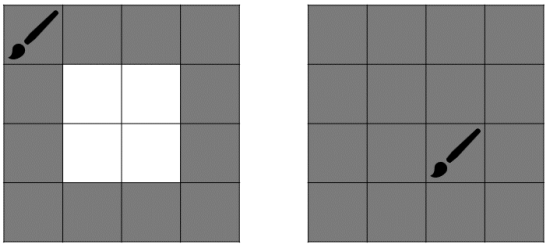

Figure 3: Example problem on the pixel domain. Task: fill in the outline, by moving the brush/cursor and drawing in necessary pixels. Left is the initial state, right is the goal state.

The second domain, *Pixel*, consists of a black/white pixel-grid (i.e. an image), where a cursor needs to manipulate

the image by moving the cursor to the right cells and drawing them in. Note that the domain is limited in that there is no mention of other colours nor that the algorithm is expected to erase previously drawn pixels. The DSL consists of the following actions:*MoveRight, MoveDown, MoveLeft, MoveUp, Draw*, and conditions checks: *AtTop, AtBottom, AtLeft, AtRight, NotAtTop, NotAtBottom, NotAtLeft, NotAtRight*. As the pixels are either black or white, Hamming Distance is used as a cost function to evaluate the programs. This domain contains 500 input-output examples to test the algorithm's performance.

4.3 Domain 3: String

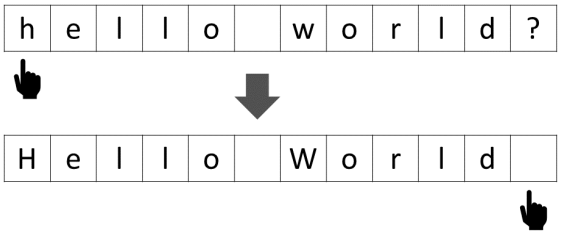

Figure 4: Example problem on the string domain. Task: capitalize the first letter of each word and remove any non-alphabetical characters. Top is the initial state, bottom is the goal state.

The third domain, *String*, is the problem of sting manipulation and is arguably the most realistic, but also complex domain. The goal is to manipulate a string by moving a pointer to the necessary characters and altering them. For example: capitalize the first letter, remove the first word etc. Due to its complexity, a sub-problem of the domain is taken where the algorithm is only involved in capitalization and removal of letters but does not need to consider adding them. The DSL consists of the following actions:*MoveRight, MoveLeft, MakeUppercase, MakeLowercase, Drop*, and conditions checks: *AtEnd, NotAtEnd, AtStart, NotAtStart, IsLetter, IsNotLetter, IsUppercase, IsNotUppercase, IsLowercase, IsNotLowercase, IsNumber, IsNotNumber, IsSpace, IsNotSpace*. As the operations are made on a string, the Levenshtein distance is used as a cost function to evaluate the programs. This domain contains 19,295 input-output examples to test the algorithm's performance.

4.4 Metrics

The main metric that will be used to evaluate the performance of the variations of the algorithm is the success rate that the algorithm has in synthesizing the 550, 500 and 19,295 tests for the *Robot*, *Pixel* and *String* domains respectively. Additionally, the execution time and program length of each test will be recorded to provide additional insight into the characteristics of the algorithm and how the parameters affect the search. Note that execution time represents the time it takes for the algorithm to generate the programs, during which it also runs the program on the example IO, it, therefore, encompasses both the run-time of the algorithm as well as of the created programs.

5 Results and Discussion

5.1 Acceptance Function: Discount Factor (α)

	cases_solved	execution_time	program_length
alpha	-0.038462	-0.333835	0.379566

Table 3: Shows the Kendall correlation coefficients for alpha compared to the various metrics across all three domains. Other parameters: {'type': 'metropolis', 'add_token_end': 10, 'add_token_random': 0, 'remove_token_end': 0, 'remove_token_random': 20, 'add_loop_end': 10, 'add_loop_random': 0, 'add_if_statement_end': 10, 'add_if_statement_random': 0, 'start_over': 2}

The first question to be addressed is whether the discount factor α has a significant impact on the performance of the algorithm. To answer this question, we can examine whether there is a correlation between the value alpha and any of the three metrics if all other variables remain fixed. To be specific we obtain the Kendall correlation coefficient because it means that we do not need to assume that the data is distributed normally, especially because we do not need to assume that the trend is linear.

The data shows that alpha can be selected to strike a balance between execution time and program length, but that it does have a significant effect on the algorithm's ability to solve problems. There was a significant but approximately opposite correlation coefficient found with the execution time as well as program length, see Figure 3. There was, however, no observable correlation with the percentage of cases solved. As such, it is therefore difficult to answer the second question and point out an optimal value for alpha.

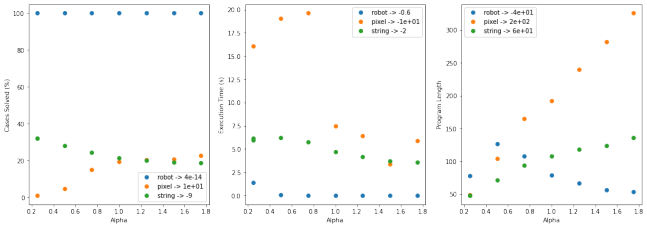

Figure 5: Shows the number of test cases solved, and the execution time as well as the program length of the programs generated for different values of alpha for each of the three domains. Other parameters: {'type': 'metropolis', 'add_token_end': 10, 'add_token_random': 0, 'remove_token_end': 0, 'remove_token_random': 20, 'add_loop_end': 10, 'add_loop_random': 0, 'add_if_statement_end': 10, 'add_if_statement_random': 0, 'start_over': 2}

Further examination of the exact data points shows however that this correlation is not unanimous amongst the different domains, see Figure. First, alpha values below 1 showed opposite trends for the pixel and string domains with a significant amount of impact on the number of problems solved especially for the pixel domain which drops to 0 as alpha decreases. Second, alpha values below 1 also irregular behaviour with a jump to significantly longer execution times

on the pixel domain. This is also true for an alpha of 0.25 for the robot domain. Third, the trends in program length are also opposite for the robot domain compared to the others. As a whole, the behaviour of the algorithm is more erratic and unpredictable below alpha values of 1. The conclusion is that it is difficult to balance an alpha for multiple different domains.

Overall, a higher discount factor should reduce the probability of accepting a mutation which negatively affects the evaluation of the program. Although none of the data is sufficient together they suggest that a higher alpha could mean that the programs get stuck in local optima more frequently due to their reluctance to accept mutations which reduce the score.

5.2 Mutation Weights

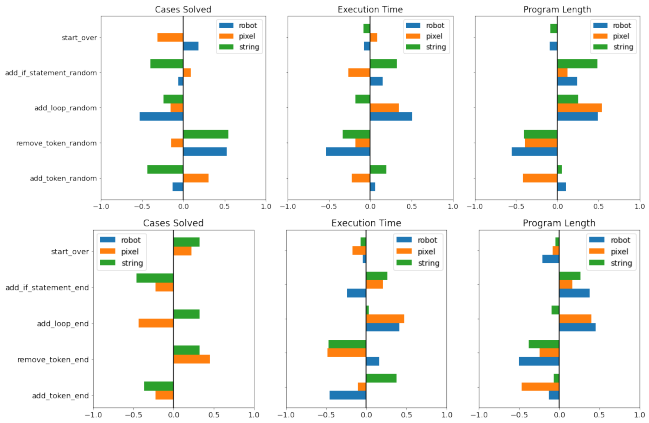

Figure 6: Shows the absolute Kendall correlation between the various mutations and the three metrics for each domain for randomized locality (Top) and fixed locality (Bottom). Other parameters: {'type': 'metropolis', 'alpha': 1}

The configuration of weights has a significant impact on the performance of the algorithm, but it is very domain dependent on how it affects the score. This was true for both the mutations with randomized as well as those with a fixed locality.

Two sets of experiments were done: one for randomized locality on the mutations and conversely one with each mutation being applied to the end of the program. Five different values were tested for each mutation weight with all others kept at a constant value as control variables to systematically test the effect of each mutation. The weights used by Van Wieringen were scaled by a factor of 3 to make it easier to test smaller weights relative to the control variables. Note that the *remove_token* mutation is also being assigned a weight of 30 so that it would be equal to the other mutations in the control set. For the control variables, the *start_over* mutation was kept at a quarter of the weight of the other mutations, the same as in the experiments run by Van Wieringen. This was done because other research suggests that lower probabilities for restarts are better for performance since higher probabilities result in an algorithm more akin to random-walk rather than an applied method [7]. Finally, while the weights scale linearly the probability that a mutation is selected is dependent on the weights of all possible mutations. As such the relative probability of each mutation was taken into account rather than their weight when comparing the variables.

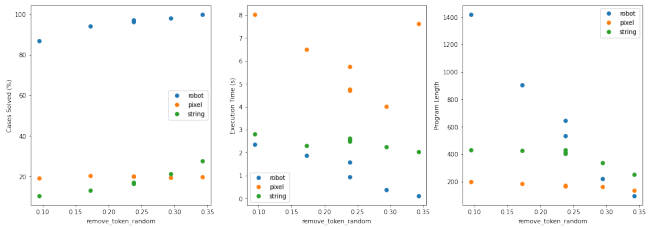

Figure 7: Shows metrics for different probabilities of the *remove_token_random* mutation. Other parameters: {'type': 'metropolis', 'add_token_random': 30, 'add_loop_random': 30, 'add_if_statement_random': 30, 'start_over': 6}

Similar to how alpha's impact on the performance of the algorithm was tested, Kendall correlation coefficients were obtained to see if there were any, including non-linear, correlations between the weights and the metrics. While the coefficients show that there is a significant correlation, there is no unanimous agreement in terms of correlation between the domains that are in the same direction with a significant degree for the mutations with randomized locality, see Figure 6. One possible explanation is that since the algorithm has less control over where to apply a mutation, the algorithm has to try to over-fit to the average distribution of the domain rather than being able to recognize a more logical distribution that is generally applicable.

The one possible exception is that there was some agreement that a *remove_token_random* mutation should be slightly higher than the other parameters, although this correlation was only moderately present for the robot and string domains and negligible for pixel. This is better illustrated in Figure 7. It is also notable that for the *remove_token_random* mutation, there are also negative correlations across all three domains for the generated programs execution time and program length. This supports the notion that slightly higher probabilities of removing tokens improve performance. This improvement could be because it enables the algorithm to make more corrections since it cannot control exactly which transition tokens will be selected. This could also explain the shorter program lengths as the algorithm is less likely to generate identity transitions in the program.

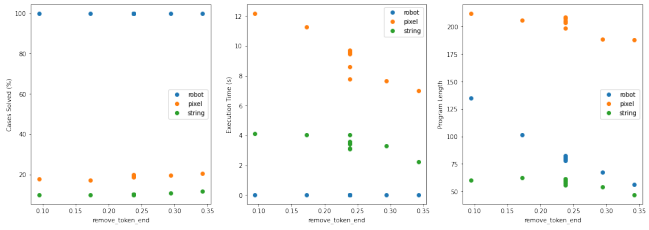

Figure 8: Shows metrics for different probabilities of the *remove_token_end* mutation. Other parameters: {'type': 'metropolis', 'add_token_end': 30, 'add_loop_end': 30, 'add_if_statement_end': 30, 'start_over': 6}

The experiments run with mutations solely applied to the end of the program show more correlated behaviour. That said they are also more difficult to judge given the relatively low strength of the correlation values and that the robot domain had a 100% success rate. With the exception of the *add_token_end* mutation, all other mutations showed a slight to moderate degree of correlation in the same direction for the pixel and string domains, see Figure 6. Again the *remove_token_end* mutation stands out, because it is the only mutation where the correlation is significant and in the same direction for almost domains for execution time and program length. The exception is that on average the robot domain's execution time increases rather than decreases, but the correlation is almost insignificant. Furthermore, because the robot domain had a 100% success rate removing more tokens on a simple domain with very low execution times and short programs could cause the algorithm to have to add slightly more tokens overall causing the delay, without experiencing the benefits from corrections. Apart from that, the same hypothesis proposed for the randomized locality equivalent applies here. That said, the performance graphs show the actual difference in performance is small, see Figure 8. The maximum difference in solving percentage of 1.4% and 0.8% for the pixel and string domains respectively. This is especially noticeable compared to the randomized locality version with monotonic increases of 13.1%, to 17.1% for the robot and string domains, recall that the pixel domain was the exception to the trend there. This could be because the algorithm already has less randomness in where it can alter the program, which would make errors relatively easier to fix.

Another noteworthy point is that the *add_loop_end* mutation had opposite correlations on the pixel and string domains, which is the only exception in the end locality experiments. This could be explained by the different lists of available Boolean tokens and the types of example problems. Where the pixel domain only has checks related to the location of the cursor the String domain also had checks on the state, e.g. *IsUppercase*. These could be significantly more useful in solving certain problems. For example, if the goal was to capitalize the first lower case letter then the algorithm could benefit significantly from using a loop to create a segment such as: [*LoopWhile(IsUppercase, MoveRight), MakeUpperCase*] which would require only two tokens in the program. The question is also whether state checks would be a significant benefit to the pixel domain, also since the types of tasks on the pixel domain are presumably less state-dependent given that each cell can only have one of two values as opposed to any character. This highlights the importance of the selection of tokens in the DSL.

5.3 Mutation Locality

The collected data does not provide sufficient evidence to suggest that locality has a significant impact on the algorithm's ability to successfully synthesize programs. Given that there is no guarantee that optimal hyper-parameters were found for each category, the best results, mean and median were examined in each category to try and highlight the potential strengths and weaknesses of each approach, see 9.

Although randomized locality for mutations had better success rates on the more complex pixel and string domains, unlike the algorithm which applied mutations to the end of the program, it did not have a 100% success rate on the simpler robot domain, within the allotted 60-second time limit. As mentioned in Subsection 5.2, one idea is that the randomized locality mutation overfits the average distribution of the domain, this could also explain why the results are similar and why randomized locality might work better on the more complex domains.

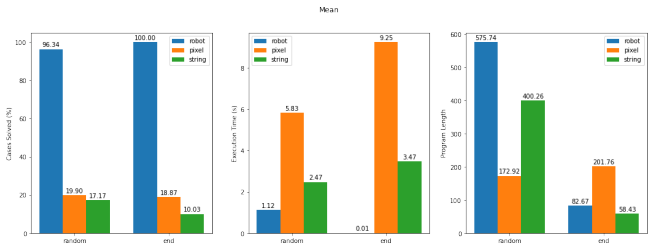

Figure 9: Shows the mean success rate, execution time and program length for all of the tests which varied the locality on the three domains. Due to similar findings, the median and best results have been included in Appendix 7 instead. Other parameters: {'type': 'metropolis', 'alpha': 1}

Apart from the success rate, on average the randomized locality experiments had shorter execution times when successful but their programs were significantly longer. This suggests that it was more likely to create an identity segment in the program where the sum of the tokens would be an empty program. For example, [*MoveRight*, *MoveLeft*] on the robot domain. This could be because the algorithm has more difficulties in trying to associate such a transition with a specific mutation.

5.4 Acceptance Function: Metropolis-vs-Hastings

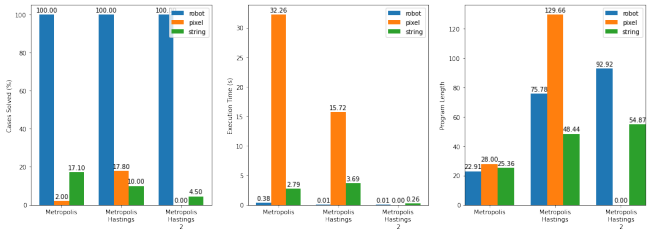

Figure 10: Shows the mean success rate, execution time and program length for all of the tests which varied the acceptance function on the three domains. The methods are described in Subsection 5.4. Other parameters: {'alpha': 1, 'add_token_end': 30, 'remove_token_end': 50, 'add_loop_end': 10, 'add_if_statement_end': 10, 'start_over': 10}

In addition to the original Metropolis method, two versions of the Metropolis-Hastings acceptance function were created. The forward and backward transition probability is calculated using the probability of the given mutation and the one which would undo it. For example, the remove_token could be undone by applying the respective mutation that placed the last token in the program, hence end-locality was used for these

tests. The difference between the two methods is start_over mutation. The first made a symmetry assumption and would return an equal forward and backward transition probability. The idea was that this would not disrupt the condition of detailed balance since the Metropolis method could also be applied successfully. The second calculated the backward probability by taking into account the probability of each token in the latest program.

The collected data supports Van Wieringen's original hypothesis that inverses would result in an underestimate of $J(x, y)$. While the full Metropolis-Hastings versions of the algorithm managed to keep up on the simpler robot domain, they showed significantly worse results in the other domains to the point that the 2nd version had a 0% success rate on the pixel domain, see Figure 10. Although the 1st method won in a few instances of specific domain-metric combinations, the version of the Metropolis-based algorithm was worse compared to when it was run with different weight configurations. The weights of each mutation were purposefully shifted so that the forward and backward probabilities would be different. Random values were selected that seemed feasible based on the results for the analysis of the mutation weights. As such, it has to be stated that hyper-parameter tuning or more broadly parameter selection has not been thoroughly explored within the given time. That said, based on our experiments, there seems to be no benefit to using the full Metropolis-Hastings acceptance function over the one of the simpler Metropolis method based on the current parameters without accurately being able to define explicit inverses.

6 Conclusions and Future Work

Systematic evaluation of the components of the Metropolis-Hastings stochastic search algorithm suggests that with certain conditions tuning can be done to improve the performance but that this fails to significantly improve the performance compared to more randomized approaches focused on more specific domains.

The first component to be altered was the mutation applied to the program during each iteration. . The mutation weights and therefore probabilities, showed that they could have a significant impact on the algorithm's success rate for synthesising programs. In all experiments where the location of a mutation was randomized, the results varied significantly in the degree and direction of the correlation depending on the domain it was tested on. Contrasting, the experiments where the locality of a mutation was fixed at the end of the program did show noticeable trends across domains. That said, the impact they had on the algorithm's success rate was significantly less than that of the randomized alternative. Furthermore, the randomized and fixed locality variations had similar performance overall. Together this suggests that the algorithm with randomized mutations overfits the domain and manages to even outperform the fixed locality version on the more complex domains. It also has to be stated that the DSL played an observable role in how domains responded to the algorithm, although this does not change the previous conclusion.

The second component was the acceptance function which is used to determine whether a non-improving mutation would be applied. None of the variations of the acceptance function was found to significantly improve the success rate of the algorithm. The discount factor used to calculate the normalized cost of a program for the acceptance function could be used to balance the programs found in terms of algorithm execution time and program length, but showed no impact on the success rate. Furthermore, alternative acceptance functions based on the full Metropolis-Hastings method showed increasingly poor success rates on more complex domains. This supported Van Wieringen's original hypothesis that it would be difficult to define explicit inverse probabilities for each of the mutation types, due to the problem of underestimating. Unless a method is found to accurately define explicit inverses there seems to be no benefit to adjusting this part of the acceptance function.

Overall, independent of adjustments, the Metropolis-Hastings algorithm does not appear to be the most suitable for generating programs on the relatively complex domains with the current set of mutations. Van Wieringen's prior research showed that other stochastic techniques showed better performances [12]. That said While improvements can be made, these do not seem to significantly improve the algorithm's performance sufficiently to outperform more domain-specific search methods.

Despite the underwhelming results with regards to the Metropolis algorithm's ability to efficiently search the space of programs, there are still several suggestions that can be made regarding future research. Primarily, there is still a lot of research that can be done on the mutations used by the search algorithm. Currently, there is no true recursion that can be created by the algorithm since the loop mutations create a loop dependent on a single condition token which repeats a single transition token, rather than enabling the algorithm to create a full program. This means that even certain relatively simple algorithms like bubble sort cannot be synthesized with these mutations. Similarly, the If mutations always only take a single condition and result in an if-then-else. This means that the generated programs cannot have an equivalent to switch conditions. This could be another direction for research that is also not limited solely to the Metropolis-Hastings method. Subsequently, more research can also be done on the effects of the selection of DSL tokens. Apart from the generation procedure, attempts could be made to combine the Metropolis-Hastings algorithm with other search techniques, to see if it can overcome some of its rigidity in search technique or to add elements to train it on a specific domain. Additionally, a more detailed analysis of the specific programs generated by the Metropolis-Hastings algorithms can be done to analyse its behaviour and to come up with strategies to overcome the weaknesses of the algorithm which could help with understanding how to combine different techniques.

7 Responsible Research

To be transparent about the origin of the code, it must be stated that this paper adapted the code base used by van Wieringen [12]. That code base was already a continuation of the code produced by Cropper and Dumančic [1]. Their

work laid the foundation for the code-base used in this paper As such, credit has to be given to these previous authors for their contribution to making this paper possible.

To ensure that the results are reproducible, there are several technical aspects of the experiments with regard to hardware and randomization that need to be mentioned. All of the experiments were conducted with a Mersenne-Twister pseudo-random number generator with a seed of 5099404. Each experiment was run on the DelftBlue supercomputer, hosted by Delft High-Performance Computing Centre (DHPC), in a Python 3.9 environment [2]. The individual runs were done in parallel using pools, separate experiments for each domain and run configuration were split into SLURM script tasks which were carried out with 32 CPUs each with 4G of memory. The data was then processed using Jupyter Notebooks. In the spirit of open data, all of the relevant information has been made available online. The algorithm, instructions on how to run the code and the notebooks are available online on GitHub, but for any questions feel free to email the author directly, the email is listed at the top of the paper.

A Appendix I: Results Locality

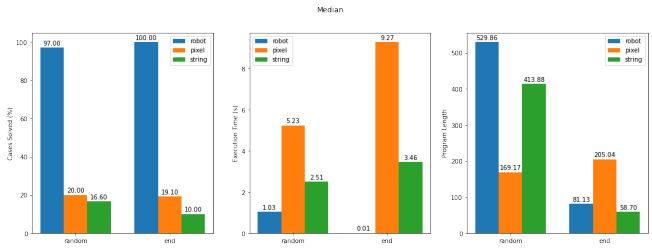

Figure 11: Shows the median success rate, execution time and program length for all of the tests which varied the locality on the three domains. Other parameters: {'type': 'metropolis', 'alpha': 1}

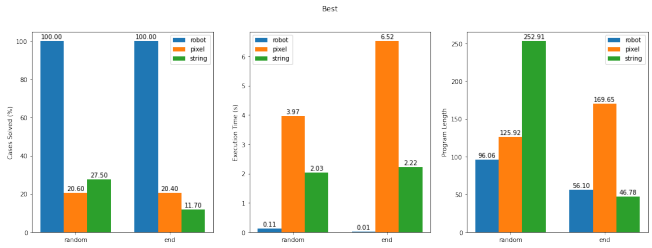

Figure 12: Shows the mean success rate, execution time and program length for all of the tests which varied the locality on the three domains. Other parameters: {'type': 'metropolis', 'alpha': 1}

References

[1] Andrew Cropper and Sebastijan Dumanči´ Dumanči´c. Learning large logic programs by going beyond entailment. 2020.

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

[3] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. The three pillars of machine programming publisher association for computing machinery (acm) terms of use creative commons attribution-noncommercial-share alike the three pillars of machine programming. 2018.

[4] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. pages 317–330. Association for Computing Machinery, 2011.

[5] W K Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97, 1970.

[6] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. pages 215–224. Association for Computing Machinery, 2010.

[7] Jason R Koenig, Oded Padon, and Alex Aiken. Adaptive restarts for stochastic synthesis, 2021.

[8] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:2384, 1953.

[9] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. pages 305–316. Association for Computing Machinery, 2013.

[10] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. pages 15–26. Association for Computing Machinery, 2013.

[11] Armando Solar-Lezama. A. lecture 1: Introduction and definitions., 2018. [Online; accessed 22-Apr-2022].

[12] Victor van Wieringen, S (mentor) with contributions from Dumančić, and C B Poulsen. Comparative analysis of the metropolis-hastings algorithm as applied to the domain of program synthesis, 1 2022.