Bachelor Thesis

-

Comparative analysis of the Metropolis-Hastings algorithm as applied to the domain of program synthesis

Victor van Wieringen, *V.J.vanWieringen@student.tudelft.nl*
Supervised by: Sebastijan Dumancic

January 23th, 2021

Abstract

In this research the Metropolis-Hastings algorithm is implemented for the problem of program synthesis and compared with Brute[1], a best-first search, together with multiple other different search algorithms. The implementation and choices of the Metrolpolis-Hastings algorithm are discussed in detail. The algorithms are tested for three different domains, each with their own associated DSL. Finally, comparisons are drawn between the search algorithms by analyzing the results of these experiments. It is found that the performance of any search algorithm depends very heavily on the specific domain and cost function used and the Metropolis-Hastings algorithm falls short in terms of performance when compared with other conventional methods.

1 Introduction

1.1 Program Synthesis

Program synthesis is an area of ongoing research focusing on the automatic generation of explicit programs that match a given specification. It allows us to automate various parts of the software development process, which has large implications, since software development is a difficult, time consuming and costly task[2]. By being able to automate different parts of the software development process, progress in program synthesis might lay the foundation for new programming paradigms such as natural language programming [3].

Program synthesis can be split up into a combination of the following three sub problems[4]:

1. Intention, Deriving the specification for the program.
2. Invention, Deriving the program itself.
3. Adaptation, Adapting the program to changing needs.

For this research, the intention part of our program synthesis problems will consist of input/output-examples, the research will be focused on the invention phase in the above three-component program synthesis pipeline.

Figure 1: Example input/output pair

1.1.1 Cost heuristic

The total set of all possible programs (program space) is too large to explicitly enumerate for any meaningful domain, yet it is the goal of the invention phase to find, within this vast space, a program that matches a certain behaviour. One of the main challenges of the invention phase therefore, is finding an efficient means of searching this vast program space[5].

A general property of programs that we can utilize when searching, is that similar programs tend to create similar outputs when ran on the same input. Because of this property, a strategy that can be used is to define an example-dependent loss function for every domain. The cost function simply compares the candidate program's output with the expected output, and serves, within the search algorithm, as a measure of how close this candidate program is to finding a solution. The cost function is therefore used as an important guiding heuristic in the program synthesis search effort.

1.1.2 Research focus

This research will build forward on a paper by Andrew Cropper and Sebastijan Dumancic[1], in which they describe a novel ILP system called "Brute" which makes use of an example-dependent loss function. They show that Brute performs relatively well in comparison with other classical ILP systems because of the use of such an example-based loss function. However, Brute had a tendency to get stuck in local minima and failed to find a correct program in these cases.

In this research, different search algorithms are implemented for the problem of program synthesis, these algorithms are then evaluated in terms of their performance and learning time. We work with a group of 5 students, each of us responsible for implementing a different algorithm to search the program space. The different algorithms are then ran on the same domains as Brute was in the original paper in order to draw comparisons between them. The report places an emphasis specifically on the development and

adaptation of the Metropolis-Hastings algorithm, since the author of this paper was responsible for adapting it to the domain of program synthesis.

2 Related work

2.1 Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm is based off the concept of Markov-chains, which are state models with a set of probabilistic transitions between them[6]. The Fundamental Theorem of Markov-chains states that for Markov-chains with certain properties, a *stationary probability* of a Markov-chain can be calculated [7]. This stationary probability will be an eigenvector of the transition matrix like so.

$$\pi(x) = \pi(x) * K(x, y) \tag{1}$$

Where $K(x, y)$ corresponds to the probability of a state change from state x to state y. The stationary probability of a state - $\pi(x)$ - is the chance of the Markov-chain being in state x after starting at a random state and running for a long time.

We can construct a Markov-chain with a given stationary distribution by defining a proposal distribution $J(x, y)$ and using it in our definition of $K(x, y)$ like so:

$$K(x,y) = \begin{cases} J(x,y) & \text{if } A(x,y) \geq 1 \\ J(x,y) * A(x,y) & \text{if } A(x,y) < 1 \end{cases} \tag{2}$$

Or more concisely as:

$$K(x, y) = J(x, y) * min(1, A(x, y)) \tag{3}$$

Where $A(x, y)$ is defined as:

$$A(x, y) = \frac{\pi(y) * J(y, x)}{\pi(x) * J(x, y)} \tag{4}$$

The function A(x,y) is known as the acceptance ratio because it essentially determines whether we accept a transition from x to y or not based on the ratio between the two solutions' costs.

2.2 Search methods

Below is an overview of the different search methods that are compared in this research.

2.2.1 Brute (best-first search)

The deterministic algorithm as described in the original paper[1]. Brute consists of two stages, the first stage (invent), functions simply as a way to increase the possible step size for program construction. This happens by generating all permutations of the simple atomic tokens up until a maximum length, these permutations form the library that the second stage (search) can use. The second stage then is simply a best-first search using this library: a list of candidate programs is maintained and every iteration the candidate with the lowest cost is expanded with all tokens in this library.

2.2.2 Metropolis-Hastings

A stochastic search technique, well suited for large spaces from which direct sampling is difficult. The algorithm models the search tree as a Markov-chain, this Markov-chain is explicitly constructed first and then weighted by using the cost function as a guiding heuristic. By starting at a random state and then simulating the constructed Markov-process for this state, progressively better programs will be generated.

2.2.3 Genetic Algorithms

Genetic Algorithms are a class of algorithms rather than a specific implementation. However, a genetic algorithm takes inspiration from the natural process of evolution. By defining operators on one or multiple programs, we can simulate the natural evolution process. A large 'population' of programs is kept and updated, and through selective breeding, random mutations and removal of bad programs, the population of programs is slowly evolved to minimize this cost.

2.2.4 Very large-scale neighborhood

Another class of largely stochastic search techniques, enveloping a whole array of different implementations. The common principle is the following: as the algorithm searches through the space, a copy of the best candidate(s) is kept, when expanding the search to more candidates the algorithm looks at the neighbours of these candidates and evaluates those. However, when the neighbourhood of the search is small, a basic neighbourhood search will suffer from getting

stuck in local minima. The principle of a very large-scale neighborhood search is then to expand this space and allow for programs 'further away' from the best candidate to also be evaluated. By simply making the space of candidates broader, the algorithm hopes to escape local minima.

2.2.5 A*

This deterministic search technique in principle works very much like Brute does, only it extends the cost heuristic to include a cost function based on the path to the solution. By including this new heuristic, A* hopes to imbue the best-first search with more sense of direction. Hence speeding up the search effort. For this search technique then, the difficulty lies in defining this path heuristic. If it is possible and done right however, A* should yield significant improvements as compared with the best-first search.

2.2.6 Monte Carlo tree

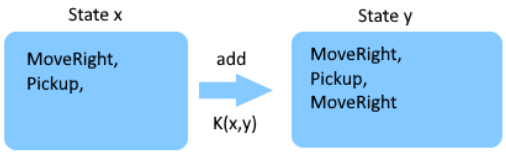
This stochastic search technique works by modeling the syntax tree of the DSL and associating a weight with every node. The weights will start out randomized, then when the algorithm is ran it will pick a token from the syntax tree based on these weights. Based on the induced change in cost, the algorithm can determine if the action was beneficial or not and through back-propagation, the weights are updated. In this way the Monte Carlo tree search trains itself and develops a predisposition over time for actions that result more often in a decrease of the cost.

3 Method

3.1 Adapting the Metropolis-Hastings algorithm

In applying the Metropolis-Hastings algorithm to the domain of program synthesis, we model the search tree as a Markov-chain K and simulate it starting at a random state P_0, which corresponds in our case to the empty program. By simulating the state changes in K for P_0, we are able to repeatedly mutate the initial program and eventually hope to reach a state in the Markov-chain that corresponds to a program that solves our task.

Figure 2: Example of a state change (add instruction) in the Markov chain.



The difficulty in implementing the Metropolis-Hastings algorithm for the specific case of program synthesis, is in how to best construct this Markov-chain K.

3.1.1 Proposal distribution J(x,y)

Since the program space is infinitely large, it is infeasible to define unique transitions for every program in the space. Instead, we start with a random program and define a transition function, which will take program as input and apply a mutation to it. Since the usage of Markov-chains makes it infeasible for our algorithm to backtrack, it is important that our algorithm defines plenty of smaller mutations if we want to have any hope of our algorithm ever converging.

To construct Markov-chain K, we start out with a proposal distribution J, which we explicitly defined as a set of mutations with associated (unweighted) probabilities. The proposal distribution is then weighted with the cost to form K. The state transitions in J and K are the same, and are explicitly defined as a set of mutations, these mutations can be applied to any program to change it to a new program.

The set of mutations and associated probabilities forms the basis of our algorithm: the proposal distribution J. Because we associated a probability with our mutations, we can sample mutations from J.

Figure 3: The mutations and weights that were used.

Mutation	Unweighted probability
add_token	10
remove_token	20
add_loop	10
add_if	10
start_over	2

3.1.2 Calculating the acceptance ratio

Looking at equation 4, it becomes clear that in calculating the acceptance ratio for a mutation sampled from $J(x, y)$, we also need to calculate the value of $J(y, x)$, the inverse probability. It would seem since we have such easy access to $J(x, y)$, that the value of $J(y, x)$ would be just as easy to calculate. In practice however, it's not that simple.

The way that the proposal distribution is defined allows for easy sampling, but it doesn't allow us to calculate the chance of a transition between arbitrary programs. There are multiple ways around this.

1. Make every operation have an explicit inverse.
2. Make it so that $J(x, y) \approx J(y, x)$ for all mutations.

On first sight, the first option might seem most logical, however, it poses some additional challenges. The biggest challenge that comes with trying to define an explicit inverse for every mutation, is that for some mutations this might be difficult or even impossible to do. Take for instance the mutation "start_over", which turns the program into P_0, the empty program: it is easy to see that for longer programs this inverse chance will be much lower than for shorter programs. Additionally, even with properly defined inverses for all mutations, there will still be inconsistencies: in defining an explicit inverse for every mutation, we implicitly assume that this is the only way the algorithm can make this inverse transition. This isn't true: many of these explicitly defined operations could be modeled by using multiple other different transitions, thus leaving us with an underestimate of $J(y, x)$.

Since the first option has its flaws, the usual way of dealing with the problem of calculating inverse probabilities is the second option: Make it so that $J(x, y) \approx J(y, x)$ for all mutations. This also has the following benefit: it reduces the calculation of the acceptance ratio to the much simpler Metropolis-ratio[8].

$$A(x, y) = \frac{\pi(y)}{\pi(x)} \tag{5}$$

3.1.3 Defining the stationary distribution

It is clear that the cost should map to the stationary distribution. However, since the cost can be arbitrarily high, it needs to be normalized first:

$$\pi(x) = e^{-Cost(x)} \tag{6}$$

By normalizing the cost in this manner, the resulting probability $\pi(x)$ will have a value between 0 and 1.

3.1.4 Final algorithm

After constructing the proposal distribution J, which we can use to obtain a mutated version of the input program, the algorithm is rather simple:

$P \leftarrow \{\}$
while $Cost(P) \neq 0$ **do**
  $P' \leftarrow J(P)$
  $C \leftarrow A(P, P')$
  $R \leftarrow rand(0, 1)$
  if $R < C$:
    $P \leftarrow P'$
end while

P will contain the solution program

Where $rand(0, 1)$ returns a real number between 0 and 1 and the *Cost* function corresponds to the example-based cost function as described in the original paper[1].

3.2 Domains

We define three different domains, each with their own associated DSL. Every DSL will consist of Loop- and If-tokens, these specify program control flow and are hence not unique to the domain. All the other tokens that are used in the search however, are unique to the domains. The complete set of instructions for every one of the DSL's can be found in Appendix A.

Domain	Description	Cost function
String	Manipulate strings in specific ways.	Levenshtein
Pixel	Paint the right pixels in a grid.	Hamming
Robot	Pickup and move a ball to the right spot.	#Steps

4 Experiments

Seeing as there are substantial differences between the different domains, the methodology and

results pertaining to each domain will be discussed separately.

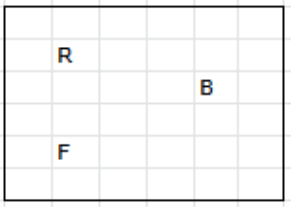
4.1 Robot

The first domain is one on giving directions to a robot. The goal of this robot is to move a ball to a designated cell in the grid (the flag). In order to achieve this the robot has to move to the cell containing the ball, pick it up, and drop it in the cell where the flag is placed.

4.1.1 Domain

In this domain, a robot, ball, and flag are placed in a grid. Below is a visualization of this domain. The robot, ball, and flag are marked with R, B and F respectively.

Figure 4: Simple representation of robot domain



In the above example, the robot is expected to move to the right three times and then down once, pick up the ball, and drop the ball after moving in a similar fashion to the flag. The DSL for this domain consists of few basic instructions to move the robot about the grid, pickup and drop the ball, as well as simple loop and conditional constructs.

4.1.2 Method

The different algorithms were ran for increasing grid sizes to compare the relative performance for increasing complexity.

The cost function as described in the original paper consisted of the Manhattan distance between the current and goal location of the Ball added with the Manhattan distance of the old and new location of the Robot[1]:

$OCost{=}MD(B_C,B_G){+}MD(R_C,R_G)$

However, an improvement to this heuristic was found: by recognizing the problem consists of stages, the cost function was split up for every stage.

Robot moves to ball:

$Cost1{=}MD(B_C,R_C){+}MD(B_C,B_G){+}MD(B_G,R_G){+}2$

Robot moves ball to flag:

$Cost2{=}MD(R_C,B_G){+}MD(B_G,R_G){+}1$

Robot moves to goal:

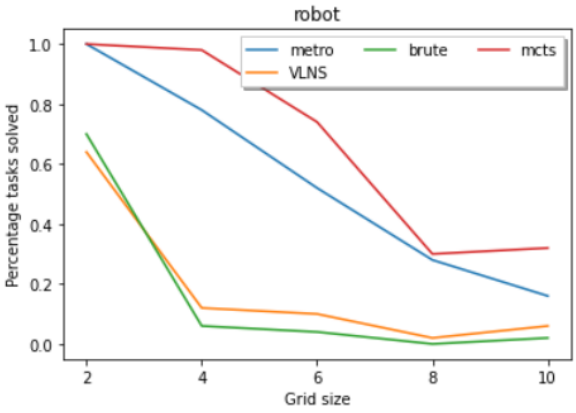
$Cost3{=}MD(R_C,R_G)$

This improved cost heuristic is continuous as well as consistent: it exactly matches the amount of steps the robot should minimally take. The overestimate in this case only being due to the fact that in certain cases, a single instruction (loop) can be used to take multiple steps.

4.1.3 Results

Due to the old heuristic not being consistent, a lot of local optima were introduced, for example, there were cases in which the robot prioritized moving to the end spot without ever picking up the ball.

Figure 5: Robot domain performance with old heuristic



As seen in the above figure, using the old heuristic makes performance drastically decrease as the grid gets larger because these local optima get deeper, e.g. the robot needs to take more steps to reach a certain waypoint.

After improving the cost function, all the algorithms except for the genetic algorithm were able to solve all of the test cases. The reason for this is because there is no more possibility for local optima to occur: any decrease in the cost function means the program is actually closer to solving the task.

Figure 6: New heuristic, cases solved

Search Algorithm	Robot
Brute	250/250
Metropolis-Hastings	250/250
VLNS	250/250
MCTS	250/250
GP	19/250
A*	250/250

With this improved cost function, the robot domain turned into what is essentially a sanity check for the search algorithms.

Seeing as the performance for the genetic algorithm was extremely low here, the choice was made not to include it in the other domains, where it similarly underperformed.

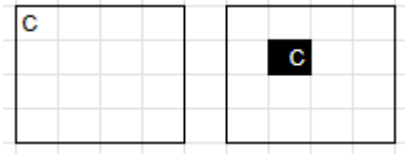
4.2 Pixel

The second domain is about painting pixels in a grid. The DSL of this domain deals with moving a cursor through a grid, being able to paint cells that are under the cursor. Starting from an empty grid (input), the task is to paint the grid so it corresponds to the output example.

4.2.1 Domain

In the below figure, an example of a problem in the pixel domain is shown, the cursor is labeled C.

Figure 7: Visualization of robot domain in/out



The DSL for this domain consists only of instructions to move the cursor up, down, left or right, together with instructions to colour or uncolour the grid. A possible solution for the problem shown in Figure 7 would be the following generated program:
"[MoveRight, MoveDown, Draw]". Of course, most of the actual testing examples consisted of more than just one pixel.

4.2.2 Method

The pixel domain is ran for increasing grid sizes, all consisting of a random combination of coloured and uncoloured pixels. The cost function for this domain consists of the number of differing pixels between the two domains being compared: the hamming distance. The results are plotted for both the percentage of the tasks solved as well as the mean learning time in seconds, both with respect to the increasing grid sizes.

4.2.3 Results

Figure 8: Pixel domain performance

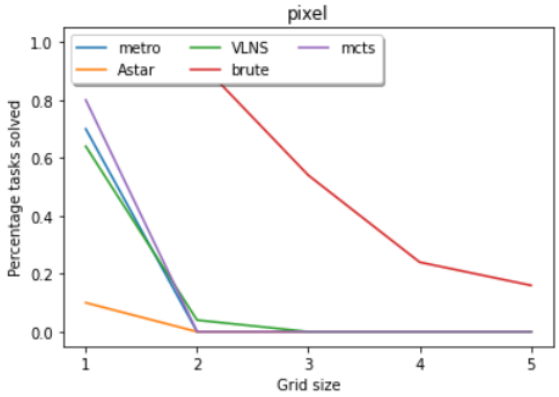
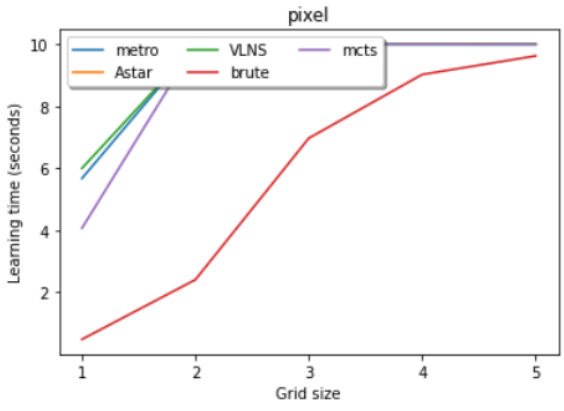


Figure 9: Pixel domain learning times



It can be seen that compared to the other algorithms, brute performs really well. This is because for the pixel domain, the cost only decreases if a pixel is coloured in the right place: it in no way depends on the location of the cursor.

In practice this means that often a large number of steps have to be taken before any decrease in the cost is induced. Furthermore, it should be noted that the cost function of the pixel domain in no way allows for local minima to occur. It is purely this characteristic of minimal distance that makes the pixel domain very difficult to deal with for the stochastic methods (Metropolis-Hastings, VLNS, MCTS), the cursor is moved randomly about the grid without any sense of direction, depending on blind luck to draw the right pixels. In contrast, Brute takes much larger steps due to making use of the library "invent" step: these algorithms have a big chance to get somewhere within only a few iterations and decrease the cost.
Note: in principle, A-star should function largely like brute, it is unclear why it underperformed this much here.

4.3 String

The last domain is the domain of real-world string manipulations, these vary from simple tasks: *(e.g. Capitalize the first letter of a word)*, to more difficult ones: *(e.g. Remove the middle word in a sentence)*.

4.3.1 Domain

The string domain provides a simple DSL for string manipulations, it consists of a pointer that points to a character in the current string. The pointer can then be moved and used to manipulate the character below.

An input/output example set is shown below:

In	Out
amelia!	amelia
robert!	robert

Similar to the other domains, we can come up with a program that solves this particular task: "[LoopWhile(IsLowercase [MoveRight]), Drop]"

The cost function that is used for the String domain is the Levenshtein distance: the minimal amount of atomic operations to change one string into another.

4.3.2 Method

For the string domain, there are 10 examples that can be solved generally by a single program. These examples are then split up into test and training groups in such a way that some cases will have only one training example, and some cases will be trained on multiple training examples. By using multiple examples in a set and adding up the cost over all of them, we incentivise the search algorithms to find general programs, rather than programs that solve a single example. We plot the results of the string experiments against the number of examples that were given.

4.3.3 Results

Figure 10: String domain performance

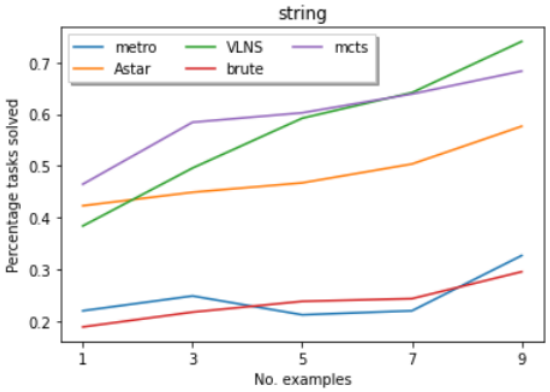
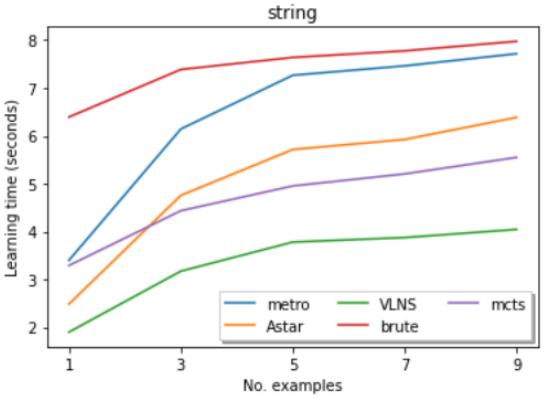


Figure 11: String domain learning times



As expected, the more examples are given, the better the performance of the algorithms gets. Whereas the Metropolis-Hastings and Brute approaches perform similarly, great improvements are seen in the other search algorithms: especially the VLNS boasts an impressive learning time as compared with the rest, being at least twice as fast as Brute in most cases. Whether it is for learning time or performance, all of the algorithms outperform Brute for the string domain.

5 Conclusion and Limitations

Brute is a best-first search approach defined for program synthesis, it performs really well when compared to classic methods because of the inclusion of a cost heuristic. In this research the best-first search was compared with multiple different other search techniques, all of them making use of a similar cost-heuristic too. These different search algorithms were then compared with one another.

5.1 Cost-heuristic

As seen in the case of the Robot domain, the exact definition of the cost-heuristic used can determine how well the search algorithms are able to find solutions. By making sure to pick a cost function that doesn't allow for local minima, we can make sure to create a synthesis problem that is trivial to solve. Additionally, in the case of the Pixel environment, a bad heuristic can make 'smarter' search algorithms perform worse than simpler, enumerative approaches.

In practice, a cost function is good if it ensures that the following properties hold:

- Small minimal amount of steps to take to see a decrease in cost.
- Cost function should not allow local optima to occur.

How to best ensure these properties depends highly on the specific domain. But in general, the more the cost-heuristic 'assumes' about a problem domain, the better it will be for the specific problem. This means there's a tradeoff between performance and generality. Lastly, in the extreme case, writing a 'perfect' cost-function for a domain will be akin to solving the general problem yourself, voiding the need for program synthesis in the first place.

5.2 Local minima

For some domains it is impossible to define a cost function with the properties listed above, these domains will have local minima even in the best case. One of the concerns around Brute was its inability to escape these minima.

It was found that while using Metropolis-Hastings provided some benefit for some domains, it performed similar to or worse than the other approaches that were evaluated.

Especially VLNS and MCTS proved to be reliable solutions when applied to the string domain. One thing to be noted however, is the fact that for these search algorithms, explicit training was done on the different domains, while the Metropolis-Hastings and Brute approaches are non-assuming: exactly the same version of these algorithms are used for all domains, down to the weights and biases.

The benefit of training a search algorithm to a specific domain is clear: performance is increased by quite a lot, as can be seen in Figure 6. However, there can be drawbacks too: specializing a search algorithm to a certain domain can mean a loss of generality, much like is true for the cost function. In Figure 5 this effect can be seen at play for the VLNS, suddenly, when using a heuristic that is different than the one that was trained on, the performance drops tremendously.

6 Responsible Science

This is the responsible research section of the report, it deals with ethical remarks about the integrity and reproducibility of this research.

6.1 Integrity and Responsibility

Seeing as the development of the codebase was a joint effort, there are some points to make here. As written in the Code of Conduct, section 3.3, all students should be critical of the material they receive, be respectful of the work of their peers and give credit where credit is due. During a large project like this, all people working on a codebase together are mutually dependent, especially for comparative analysis like this, where each student was responsible for a different algorithm. I made sure to check the validity of the results of the different algorithms as well as to understand the code. The other side of this mutual-dependency is the fact that my group members wrote large parts of the codebase that this research is built on, and for that reason they deserve credit too.

6.2 Open data

The results that were acquired for this paper were generated using a custom codebase written in

Python. In the light of open data and reproducible results, this codebase is made open-source and can be found on Github at the following URL: https://github.com/victorvwier/BEP_project_synthesis. Instructions on how to install and run can be found here as well.

References

[1] A. Cropper and S. Dumancic, "Learning large logic programs by going beyond entailment," *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, Jun 2019.

[2] C. David and D. Kroening, "Program synthesis: Challenges and opportunities," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, p. 20150403, 2017.

[3] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, "Program synthesis using natural language," *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[4] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson, "The three pillars of machine programming," *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018.

[5] "Introduction to program synthesis," *MIT lecture series (6.S084/6.887 2020)*.

[6] Merriam-Webster, "Markov chain.." `https://www.merriam-webster.com/dictionary/Markov\chain`, 2021. Accessed 14 Nov. 2021.

[7] Megan Goldman, "Markov chains - stationary distributions." `https://www.stat.berkeley.edu/~mgoldman/Section0220.pdf`, Spring 2008. Accessed 14 Nov. 2021.

[8] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pp. 305–316, 2013.

A Appendix

A.1 DSL's

As well as featuring If and While tokens, every DSL consists of the following tokens:

A.1.1 String

Boolean Tokens:
$AtEnd, NotAtEnd, AtStart, NotAtStart, IsLetter, IsNotLetter, IsUppercase, IsNotUppercase, IsLowercase, IsNotLowercase, IsNumber, IsNotNumber, IsSpace, IsNotSpace$
Transition Tokens:
$MoveRight, MoveLeft, MakeUppercase, MakeLowercase, Drop$

A.1.2 Pixel

Boolean Tokens:
$AtTop, AtBottom, AtLeft, AtRight, NotAtTop, NotAtBottom, NotAtLeft, NotAtRight$
Transition Tokens:
$MoveRight, MoveDown, MoveLeft, MoveUp, Draw$

A.1.3 Robot

Boolean Tokens:
$AtTop, AtBottom, AtLeft, AtRight, NotAtTop, NotAtBottom, NotAtLeft, NotAtRight$
Transition Tokens:
$MoveRight, MoveDown, MoveLeft, MoveUp, Drop, Grab$