Inductive Program Synthesis through using Monte Carlo Tree Search guided by a Heuristic-Based Loss Function

Bachelor's thesis

Nadia Matulewicz

Supervisor: sebastijan Dumančić
Examinor: Casper Poulsen

Computer Science and Engineering
Faculty of Electrical Engineering, Mathematics & Computer Science
Delft University of Technology

January, 2022

Inductive Program Synthesis through using Monte Carlo Tree Search guided by a heuristic-based loss function

Nadia Matulewicz
Supervisor: Sebastijan Dumančič
EEMCS, Delft University of Technology, The Netherlands
n.i.matulewicz@student.tudelft.nl, s.dumancic@tudelft.nl

Abstract

Recently, a new and promising Inductive Program Synthesis (IPS) system, Brute, showed the potential of using a heuristic-based loss function. However, Brute also has its limitations and struggles with escaping local optima. The Monte Carlo Tree Search might offer a solution to this problem since it balances between exploitation and exploration. I design MUTE, a new IPS system which uses MCTS guided by a heuristic-based loss function. MUTE's performance is tested and compared to other IPS systems in three diverse domains, namely robot planning, ASCII art and string transformations. MUTE's performance for string transformations is a first indication that MUTE can outperform Brute and other IPS systems. Manual analysis of the results shows that MUTE can indeed escape local optima. Two branch reducing enhancements, namely the removal of similar programs and the removal of tokens that show no potential, are essential for the success of MUTE.

1 Introduction

Program Synthesis is the automation of writing programs. In Inductive Program Synthesis (IPS) examples are used to specify what this program is supposed to do. Program Synthesis can be used by people without a programming background to create programs, as well as by developers to automate part of their job. It is useful for a variety of tasks including the discovery of new algorithms, optimization of existing programs, automation of debugging and even for teaching (Gulwani, 2010). Program Synthesis can be viewed as a search problem in which a space of possible programs is searched for a program that solves the task.

The main motivation behind this research is a recent study by Cropper and Dumančić (2020) that introduces a new IPS system called Brute which uses a promising search technique. They researched IPS in Inductive Logic Programming (ILP) which is a specific branch of machine learning. They argued that one of the limitations of most existing ILP systems is that an entailment based loss function is used to guide the search algorithm. When using an entailment-based loss function, a program is scored based on binary decisions saying that the program either satisfies or does not satisfy a certain condition or constraint. This way, a program that almost satisfies a constraint is scored equally bad as a program that does not even come close to satisfying that constraint. To tackle this problem, Brute is guided by a **heuristic-based loss function** which scores a program based on how close the returned output is to the wanted output. Brute's good performance on three domains of IPS problems showed the potential of using a heuristic based loss function.

However, Brute struggles with tasks where local optima occur. Brute tends to only explore the programs that immediately decrease the loss function, also in case these programs will never lead to the optimal program. This is why it could be beneficial to use a search algorithm that balances between exploitation and exploration: the exploitation of programs that look the most promising and the exploration of programs that have not been properly evaluated yet.

Monte Carlo Tree Search (MCTS) is such a search method that balances between exploitation and exploration. MCTS techniques have mainly been used for game AI, but have also proven to be useful in non-game applications like procedural content generation and planning (Browne et al., 2012). Altogether, this leads to the belief that a new IPS system which combines the use of a heuristic-based loss function with Monte Carlo Tree Search, has the potential to solve the aforementioned problem and outperform Brute and other IPS systems.

This has resulted in the following research questions:

Q1 Can a new IPS system that uses MCTS guided by a heuristic-based loss function, escape local optima?

Q2 Can a new IPS system that uses MCTS guided by a heuristic-based loss function, outperform other IPS systems?

To answer these questions, I made the following contributions. First of all, I did a literature search to get a better understanding of the topic and to find out what research has already been done. Section 3 gives an overview of the most relevant work. Secondly, I designed and implemented MUTE, a new IPS system that uses MCTS guided by a heuristic-based loss function. In section 4, the process of developing MUTE is described. Last of all, I test and analyse MUTE's performance and compare it with the performance of Brute and other IPS systems. Section 5 describes the experiments that were performed. Their results can be found in section 6.

2 Used terms

Before diving into the conducted research, this section will clarify some of the terms used in this paper.

In general, a **program** is a piece of code that solves a certain task. Common programming languages like Java or Python are very extensive and therefore it is very inefficient to search the space of all programs that can be written in these languages. This is why for IPS usually, a custom language is created that consists of a limited number of statements. All the IPS systems in this research, create programs that are simply sequences of transitional tokens. A **transitional token** is a function that takes a state as an input and returns a new state. Since the programs that we consider are sequences of transitional tokens, these programs also expect a state as input and return a state as output.

The IPS systems described in this paper take a list of transitional tokens and a list of boolean tokens as input. **Boolean tokens** take a state as input and return *True* or *False*. There are two special transitional tokens, namely *If* and *While* tokens, that can be formed by the IPS systems by using the given boolean and transitional tokens.

The IPS systems in this paper are tested in three different domains of problems, namely robot planning, string transformations and ASCII art. A custom language was created for each domain as well as a domain-specific loss function.

For **robot planning**, the IPS systems have to find a program that transforms the initial state to the wanted output state by moving a robot around on a squared grid and making it pick up and drop a ball. Each state contains the size of the grid and the positions of the robot and the ball. The provided lists of transitional and boolean tokens for this domain are respectively *[MoveLeft, MoveRight, MoveUp, MoveDown, Grab, Drop]* and *[AtTop, NotAtTop, AtBottom, NotAtBottom, AtLeft, NotAtLeft, AtRight, NotAtRight]*. The loss function is equal to the minimum number of moves that the robot needs to pick up the ball and drop it at the right position plus the number of moves that the robot needs to move to its own final position.

For **ASCII art**, the IPS systems have to find a program that transforms an empty grid into the wanted output image which represents an ASCII string. Each state consists of a binary image and a cursor which points to one of the pixels in the binary image. The provided lists of transitional and boolean tokens are *[MoveLeft, MoveRight, MoveUp, MoveDown, Draw]* and *[AtTop, NotAtTop, AtBottom, NotAtBottom, AtLeft, NotAtLeft, AtRight, NotAtRight]*. The loss function is equal to the binary distance.

For **string transformations**, a program has to be found that can transform a string into another string based on one or multiple example pairs of the initial state and wanted output state. Each state consists of a string and a cursor that points to a position in the string. The lists of transitional and boolean tokens are *[MoveRight, MoveLeft, MakeUppercase, MakeLowercase, Drop]* and *[AtEnd, NotAtEnd, AtStart, NotAtStart, IsLetter, IsNotLetter, IsUppercase, IsNotUppercase, IsLowercase, IsNotLowercase, IsNumber, IsNotNumber, IsSpace, IsNotSpace]*. The loss function is equal to the Levenshtein distance.

The provided loss functions for ASCII art and string transformations are not a perfect indication of how close a program is to solving the provided example(s). This allows **local optima** to occur. This happens when the IPS system keeps extending the same program which has a small loss even though this program will never lead to the program that solves the task.

3 Related Work

As part of this research, I did a literature search to get a better understanding of the topic and to find out what research has already been done. Due to space limitations, this section only provides the literature that is most relevant to my research.

3.1 Brute

Brute is an ILP system implemented in prolog. Brute's search for the wanted program can be divided into two stages. In the first stage, Brute invents new transitional tokens from the provided tokens. Up to a certain length, all possible combinations of tokens are included. In the second stage, a best-first search is performed using the invented tokens. Brute starts by creating a priority queue that only contains the empty program. The greatest priority is given to the programs with the smallest loss associated. Each iteration, a program is popped from the queue. For each invented token, a new program is added to the queue which is equal to the just popped program extended with this invented token. The iterations stop when a program is found that solves all the provided I/O examples or when a given time limit is reached.

Brute was tested in three different domains of problems, namely robot planning, string transformation and ASCII art. Brute was both faster and able to find larger programs than an existing ILP system in all three of these domains. In two out of the three domains, Brute also outperformed a version of Brute that used an entailment-based loss function. This shows the great potential of using a heuristic-based loss.

However, Brute was also struggling when local optima occur. E.g. consider the string transformation task of transforming `"james"` into `"J"`. After the first iteration, Brute finds the program *[While(NotAtEnd, [Drop])]* which deletes all but the last character and from there it cannot find the correct program.

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a search method that balances between exploitation and exploration. MCTS techniques have mainly been used for game AI. It has achieved great successes, also for games with a large branching factor, like Go, or when little domain knowledge is available, like in General Game Play (Browne et al., 2012; Chaslot, Bakkes, Szita, & Spronck, 2008). MCTS algorithms have also proven to be useful in non-game applications like procedural content generation and planning.

MCTS method

MCTS is a search method that repeats four steps until a time or space limit is reached (see fig. 1). The four steps are:

1. **Selection:** The search tree is traversed based on a score that balances between exploration and exploitation until a node is reached with an unexplored child node.

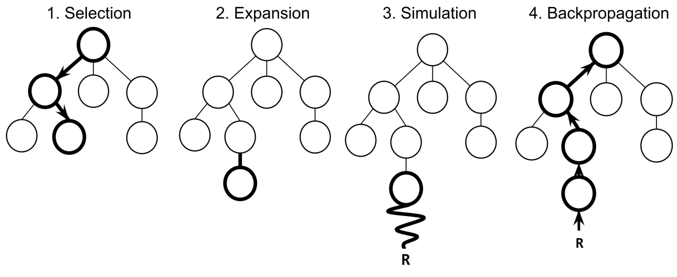


Figure 1: Four steps of MCTS

2. **Expansion:** A child node is explored.
3. **Simulation:** A simulation happens from the expanded node until a terminal state is reached.
4. **Backpropagation:** The obtained result from the simulation step is used to update all the traversed nodes.

There are many different variations and enhancements of MCTS. A selection of these is discussed in section 3.3.

Upper Confidence Bounds for Trees

The Upper Confidence Bound for Trees (UCT) algorithm (Kocsis & Szepesvári, 2006) is the most popular MCTS algorithm. *UCT* uses UCB1 to decide which child node to select during the selection step of MCTS. For each child node i, the UCT_i is computed:

$$UTC_i = \left(\overline{X_i} + 2C_p \sqrt{\frac{2 \ln n}{n_i}} \right) \tag{1}$$

where $\overline{X_i}$ is the average reward received in simulations branching from node i, n is the number of visits of the parent of i, n_i is the number of visits of i and C_p is the exploration constant. The child node with the greatest UTC is selected.

3.3 Reaching terminal states

When applying MCTS for program synthesis it is important that a terminal state (in this case: an executable program) is found before the given time or space limit is reached. This paragraph highlights research into possible enhancements that help to ensure this. The studies in this paragraph were found in a survey (Browne et al., 2012) which provides an overview of 5 years of MCTS research.

One of the non-game domains in which MCTS has been applied is planning. In game domains, MCTS is used to decide what move to do next, so it is not required that one of the explored leave nodes represents a terminal state. However, in both planning and program synthesis, the goal is to reach a terminal state which represents the solution to the problem. This is why research done on applying MCTS in the domain of planning is of great relevance to research of applying MCTS in the domain of program synthesis. One example is the study by Pellier, Bouzy, and Métivier (2010). In their search algorithm, MHSP, they combined the "the principles of UCT [...] with heuristic search in order to obtain an anytime planner that provides partial plans before finding a solution plan" (p.211) They argue for using a heuristic call to replace the random simulation in the UCT. On classic planning problems, MHSP performed comparably to classical planners.

Another interesting application of UCT is shown in FUSE. FUSE is an extension of UCT designed "to deal with i) a finite unknown horizon [...]; ii) the huge branching factor of the search tree" (Gaudel & Sebag, 2010, p.359). To tackle these problems, a couple of enhancements were made. One enhancement that was made is increasing the probability of selecting an action that leads to a terminal state as the node depth increases.

A limitation of the two above approaches is that they only help for reaching terminal states that can be found at a small depth in the search tree. However, for program synthesis, there is a particular interest in finding greater and more complex programs which would typically appear at greater depth in the search tree. First-play urgency might offer a solution. In UCT all child nodes of the current node will be explored at least once before further exploiting an already visited child node. This means that even promising nodes that occur at a high depth are very unlikely to be exploited if the tree has a high branching factor. First-play Urgency (FPU) was introduced by Gelly and Wang (2006) to tackle this problem. A certain value replaces the UCT of child nodes until they are first visited. If this value is set properly, child nodes that have shown good potential in a simulation will be revisited before visiting their sibling nodes for the first time.

3.4 MCTS in Program Synthesis

Only one publication was found on the application of MCTS in the field of Program Synthesis. In an exploratory study, Lim and Yoo (2016) implemented a program synthesis system based on MCTS and evaluated this system on six benchmarks.

In their implementation, programs were represented by program trees. A node called 'concat' was used to be able to combine multiple instructions and an 'if' node with three child branches was used to represent an if-then-else construct. The UCT algorithm (see 3.2), guided by a heuristic fitness function, was used to search the space of programs. To limit the search space, typing was applied and a maximum program length was set.

To be able to evaluate the performance, Lim and Yoo also implemented a genetic programming (GP) based synthesis system. They compared the results of the two programs on six benchmarks. Both systems were able to generate good programs on the simple benchmarks, but could not produce correct programs on the more complex tasks.

4 Development of MUTE

As section 3 indicates, there are many possible implementations for an MCTS-based algorithm. Each implementation will lead to different results and is suitable for different applications. In this section, the different steps taken and decisions made when developing MUTE are discussed. After this, the final implementation of MUTE is described. MUTE was implemented in Python and the implementation code was made public (Azimzade, Jenneboer, Matulewicz, Rasing, & van Wieringen, 2022).

4.1 Initial version

It is not possible to determine upfront what specific implementation is of MCTS is most suitable for IPS. This is why the initial implementation of MUTE was based on making logical and intuitive decisions. In this section, this initial version and its observed behaviour are described.

Implementation specifications

The first version of MUTE was based on the UCT algorithm (see section 3.2). Browne et al. (2012) stated that "The value $C_p = 1/\sqrt{2}$ was shown by Kocsis and Szepesvari (2006) to satisfy the Hoeffding inequality with rewards in the range $[0, 1]$" (p.8). This is why the exploration constant C_p was set to $1/\sqrt{2}$ and the reward was calculated using the following formula:

$$reward = \frac{maximum\_expected\_loss - loss}{maximum\_expected\_loss} \quad (2)$$

The $maximum\_expected\_loss$ is the loss that is obtained when no program is applied to the input of the provided training examples. In this way, the reward was expected to be between 0 and 1.

In the used data structure for the search tree, each node contained a possibly incomplete program. Each program was a sequence of the given transitional tokens and possibly incomplete *If* and *While* tokens. The *If* and *While* tokens had to be completed by expanding them with a boolean token as their condition and a subprogram as their body. In this way, it would be possible to build all sorts of programs, including programs with complex *If* and *While* statements.

The algorithm worked as follows. First, the search tree is initialized with an empty program at its root. Each edge either expands or completes a part of the program contained in the node it originates from. In the simulation step, random tokens are chosen to finish the program. However, corresponding to the idea of Gaudel and Sebag (see section 3.3), preference is given to actions that lead to a terminal state. As a result, the simulation step takes less time and the obtained loss is more likely to represent the actual value of the node from which the simulation started.

Observed behaviour

I used a selection of 100 examples sets (50 for string transformations and 25 for each of the two other domains) to observe the behaviour of this initial implementation.

First of all, as expected, the search tree grew asymmetrically. Secondly, this algorithm also struggled with local optima. Like Brute, for string transformations, MUTE often found deleting everything to be the best program. Due to the large branching factor, the branch containing the optimal program was not exploited enough to find this program. The depths of the leaves of the search tree were between 3 and 6 after the time limit of 10 seconds was reached. This means no programs containing more than 6 tokens were considered.

4.2 Variations

Starting from this initial version, different variations were implemented and tested. In this subsection, I will discuss the observations made when testing these different variations. Since string transformations are most suitable for observing local optima, only the example sets of this domain were used for analyzing the behaviour of the different versions of MUTE.

Deleting statistics of exploited nodes

One problem in performance was caused by programs that had low loss associated, but could not be expanded any further. For example, the program *[While [NotAtEnd] [Drop], Drop]* could not be expanded since it deletes all the characters of the given input string. However, due to its low loss, it was the cause of many visits to its parent node.

Therefore, as a first alteration, MUTE deleted statistics of nodes that could not be expanded any further. If a node had been fully exploited, the node was removed and its parent node was updated such that it no longer contained the number of visits and the reward obtained through this removed child node.

This relatively small alteration already showed some improvements in the results. This was the first version of MUTE that was able to solve tasks where Brute got stuck because of local optima, but only for a very limited number of cases.

Using complete tokens only

The second problem that was tackled, was that of having a lot of different nodes that contain similar programs. Two programs are similar if they produce the same output for the given input examples. E.g. the program *[Drop]* is similar to the program *[MakeUpperCase, Drop]*. To be able to keep track of similar programs, it is more practical to work with complete tokens only. In this way, there are no incomplete programs and thus for each node the resulting states can be computed. These resulting states can then be used to recognize similar programs.

I decided to introduce an invent function to be able to find programs that contain *If* and *While* statements without having to deal with the complexity of incomplete tokens and programs. To be able to compare the performances of new IPS algorithms with Brute's performance, I implemented a Python version of Brute together with a group of peer students. I reused the invent function of this implementation for my newer version of MUTE. In this newer version, a program is simply a sequence of invented tokens.

Now, in the expansion step, the program in the new node is applied to the given examples immediately. If the program turns out to be invalid, or similar to a program that was found before, it is removed immediately. In the simulations step, it is no longer necessary to add tokens to be able to compute a reward. Similar to the anytime planner built by Pellier et al. (2010) (see section 3.3), the reward is computed, without doing any random simulation, using the loss associated with the program in the new node. The obtained reward is more likely to match the actual value of the program contained in that node and it will save computation time.

Against expectations, using the invented tokens, MUTE still solved the exact same string transformation tasks as before. However, by analyzing the algorithm in debug mode, some notable observations were made. First of all, the invent function resulted in 2780 invented tokens. This means that for each node, first 2779 sibling nodes had to be explored before exploitation of this node would happen. This huge branching factor resulted in a search tree that only had a height of 3 after

10 seconds of execution time in debug mode. This means the longest program found existed of only three invented tokens.

New invent function

To decrease the branching factor, a new invent function was introduced. This invent function returned the following invented tokens:

1. Invented tokens containing just a single transitional token.
2. Invented *If* and *While* tokens consisting of a single boolean token as its condition and one or two distinct transitional tokens as its body.

For string transformations, this new implementation of the invent function led to only 705 invented tokens instead of 2780.

As a result of the reduced branching factor, the search tree had a height of 4 instead of 3 after 10 seconds of execution time in debug mode. However, still the same number of string transformation problems got solved.

Tracking token scores

To use a more informed selection policy, a dictionary was introduced to keep track of token scores. By collecting statistics on each invented token, the UCT could be extended to also include the token score.

The token score was updated after each time this invented token was selected for the new node in the expansion step. A token was rewarded a score of -1 if the addition of the token leads to an invalid program, a program similar to a program that was found before, or a program with a greater loss than the program without the token. The token was rewarded a score of +1 if the token leads to a program with a smaller loss than the program without the token. A token score of 0 was rewarded in other cases. The average token score was added to the UCT.

This alteration did not give the wanted effect. Tokens that led to local optima had the greatest average token score and again this led to the preference of the wrong tokens in the selection step. However, the analysis of the computed token scores provided a valuable insight. A lot of tokens had an average token score of -1, meaning the addition of this token had always led to either invalid or similar programs or an increased loss. Examples of such tokens are *If(isNumber [MakeLowerCase])* and *While(NotAtStart [MoveRight])*. Removing these tokens as options to expand the program could save a lot of computation time. This time can be used to exploit tokens that show more potential instead.

Removing tokens that show no potential

To be able to remove the tokens that show no potential, a new constant, C_t, is introduced. When the token score has been updated C_t times, the average token score is computed. If this equals -1, the token is removed from the set of invented tokens, and that token will no longer be used to expand a program. Also, the token score is no longer used in computing the UCT since this only worked counterproductively.

When analyzing the new implementation in debug mode, some interesting observations were made. After 10 seconds of execution time with $C_t = 5$, 669 out of the 705 invented tokens were removed and the search tree had a height of 6.

Domain	C_p	C_t
Robot planning	0	∞
String transformation	$2 \cdot \frac{1}{\sqrt{2}}$	7
ASCII art	$0.25 \cdot \frac{1}{\sqrt{2}}$	11

Table 1: Optimised constants

When analyzing the 36 invented tokens that were left, I noted that some useful tokens had been removed. These were tokens that were not useful at the start of the program, but that would have been useful later in the program . When I set $C_t = 10$, this led to promising results. 52 invented tokens were kept and the search tree had a height of 5. Some of the string transformation tasks that could not be solved before were now solved completely or the best-found program came close to solving it.

Fine-tuning of constants

The last alteration was fine-tuning the exploration constant C_p in the UCT and the constant C_t. The fine-tuning was done per domain since I expected there to be significant differences between the optimal values for the different domains. For robot planning, I expected that MUTE would perform best with the minimal exploration constant of $C_p = 0$ since the used heuristic is a perfect indication of how close a program is to the wanted program. However, this is not the case for the loss functions of ASCII art and string transformations and therefore my expectation was that they would perform better for values of C_p greater than 0.

For each of the domains, 75 example sets were used to fine-tune the constants. Table 1 shows the final values of the constants per domain. For robot planning, the value $C_t = \infty$ was chosen to make sure two essential tokens, namely *Grab* and *Drop*, would not be removed just because they lead to invalid programs when the robot has not reached the ball yet.

4.3 Final version

For clarity, this subsection describes the final implementation of MUTE.

Data structure

MUTE is a UCT algorithm in which the root node of the search tree represents the empty program and is the only node that does not contain a token. All other nodes contain an invented token and represent the program that you get by adding this token to the program represented by their parent node.

Setup

Invented tokens are created from the input tokens. After this, the search tree is initialized which contains only a root node. Other required variables are also initialized, for example variables for tracking statistics or for keeping track of the programs found so far.

Iteration

In the selection step, a node is selected by traversing the search tree. The UCT with the domain-dependent exploration constant C_p (Table 1) is used for this.

In the expansion step, a child node is added to the selected node. This child node contains an invented token that had not yet been chosen for any other child nodes of the selected node. In the simulation step, no random simulation is done. Instead, the loss corresponding to the program represented by the new node is used to compute the reward (equation 2).

In the backpropagation step, the values of the traversed nodes are updated based on the obtained reward. Also, the token score is updated for the token that was selected when creating a new child node. If the token token score indicates no potential after the token has been selected C_t times, the token is removed from the list of invented tokens and will no longer be selected in the expansion step.

The selection, expansion, simulation and backpropagation steps are repeated until a program is found that results in a loss of zero for all the given examples, or until a given time limit is reached.

5 Experiments

To answer the research questions, MUTE's performance was tested and compared to the performance of other IPS systems. This section describes the setup of the experiments conducted and the analysis that was done.

5.1 Setup

To find the answer to **Q2**, MUTE's performance was compared to the performance of Brute and three other IPS systems. My research is one of five, in which specific search algorithms were combined with a heuristic to see if this can lead to better performing IPS systems. The other four studies were executed by my peer students, F. Azimzade, B. Jenneboer, S. Rasing and V. van Wieringen. The specific search algorithms that were researched by them are respectively genetic algorithms (GA), A* search (AS), very large-scale neighbourhood search (VLNS) and Metropolis-Hastings (MH). Each research resulted in a new IPS system which will be described in forthcoming papers. Like MUTE, the new IPS systems were all implemented in Python and with the group of peer students, we also implemented a Python version of Brute (Azimzade et al., 2022). Since the implementation of the GA system was only finished after this paper was finalized, the performance of this system will not be part of the experiments.

Each of the remaining algorithms was tested on the domains of robot planning, ASCII art and string transformations (see section 2). These are the same domains that were used by Cropper and Dumančić (2020) to test the performance of Brute. The test data was received from Dumančić.

For ASCII art, for each integer $1 \leq n \leq 5$, 50 tasks were selected for which the wanted output represented a string consisting of two ASCII characters.

For robot planning, for each grid size a $n \in 2, 4, 6, 8, 10$, 55 tasks were selected which all consisted of a randomly generated input and output state. 55 tasks were selected for each grid size.

For string transformations, 50 different tasks were selected from the provided test data. Each task was defined by 10 examples that demonstrated the wanted behaviour. For each task and for each integer $1 \leq n \leq 9$, n examples were randomly chosen and given as training examples for the search algorithms. By choosing n different examples each time, 5 to 10 trials were available for each task and each n. All 10 examples were used to test whether the returned program was correct. Only if the program returned the correct output for each of the 10 outputs, the task was considered to be solved.

5.2 Analysis

To be able to answer **Q1** and get a better understanding of the behaviour of both MUTE and Brute, an analysis was done on a selection of their results of the string transformation tasks.

For each of the 50 string transformation tasks that were used in testing the performances, a single trial from each of complexity 1, 5 and 9 was selected and used for this analysis.

For both MUTE and Brute, the following data was obtained for each of the selected trials: input, expected output and obtained output for each test example; whether the task was solved successfully; execution time; the found program and it's length; the number of explored programs and the number of iterations.

For MUTE, for each iteration in which a new best program was found, the program, its loss, and the iteration number were also saved. For Brute, each iteration, the loss and the length of the current program were saved.

The obtained data were manually analyzed.

6 Results

This section provides the results of the experiments and analysis that are described in section 5.

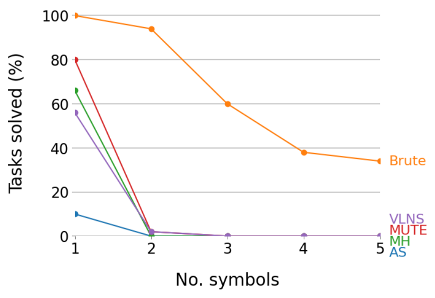
6.1 Experiments

The resulting plots of the experiments can be found in Figures 2 and 3.

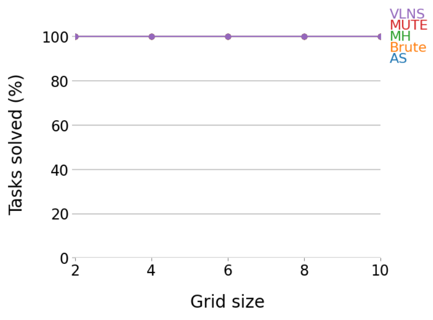
For ASCII art, MUTE performs worse than Brute. The percentages of solved tasks for MUTE are lower and the execution time per solved is longer. For the tasks with only one ASCII symbol in the wanted output image, MUTE outperforms the three other systems. However, for tasks with more than one ASCII symbol in the wanted output image, the percentage of solved tasks is close or equal to zero for all systems except Brute. To come back to **Q2**, the results for ASCII art do not show support for the hypothesis that MUTE is able to outperform Brute and other IPS systems.

For robot planning, all systems perform very well and solve all tasks for each grid size. MUTE is slower than three of the other systems, but MUTE's average execution time for solved tasks is still far below the given time limit of 10 seconds. It is hard to answer **Q2** based on the results for robot planning since all 5 IPS systems have a perfect performance.

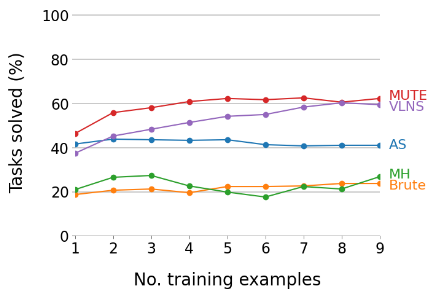
For the string transformations, overall, MUTE has a greater percentage of solved tasks than the other systems. For the tasks with a smaller number of training examples, MUTE outperforms the other systems. As the given number of training examples per task increases, the percentages of solved tasks of MUTE and the VLNS approach each other. All the systems have a smaller average execution time per solved task than MUTE. However, MUTE's average execution time for



(a) ASCII art

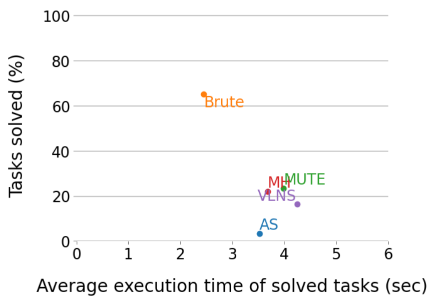


(b) Robot planning

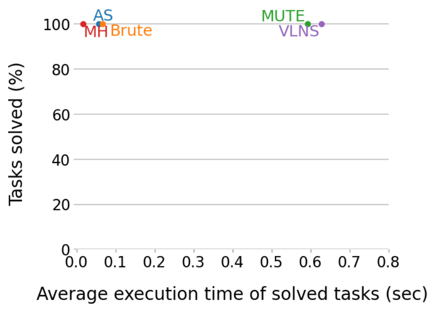


(c) String transformations

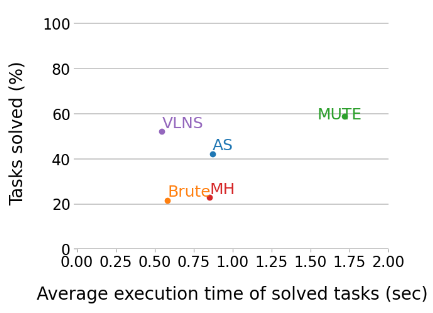
Figure 2: Success rates



(a) ASCII art



(b) Robot planning



(c) String transformations

Figure 3: Success rate versus execution time

the solved string tasks is still much below the given time limit of 10 seconds.

Altogether, even though the results of ASCII art and robot planning do not show any clear support, MUTE's performance for string transformations suggests that the answer to **Q2** is yes. Mainly examples for string transformations were used when designing MUTE and doing intermediate testing. This might explain why MUTE performs so well for this domain, but not for ASCII art. Further research is required to get a better understanding of why MUTE was outperformed by Brute on the latter domain and to find out if MUTE can be useful in other domains.

6.2 Analysis

The analysis gave greater insight into the results obtained for MUTE and Brute for string transformations,

Brute

For Brute, one main cause for failing to solve a task was that it got stuck in local optima. This could be observed by analyzing the length of the program per iteration and its costs.

For each task, solved or not, each iteration the program length either increased or stayed the same compared to the program that was considered in the iteration before. The cost of the selected programs would decrease in the first few iterations and after this stay the save for all the iterations that followed. This indicates that programs will never be extended if they have greater costs than the best program found so far. This would mean that even if you would run Brute for an infinite amount of time, it would still not be able to find the wanted the program, if this program is not formed by extending the best program found so far each time.

An example that demonstrated this was Task 3. For this task, the wanted program had to make the first letter uppercase and delete the remaining letters. E.g. `"charles"` had to be transformed to `"C"`. An example of a program that does this is *[MakeUppercase, While(NotAtEnd, [Drop])]*. This program could be formed by Brute by simply combining two invented tokens. However, Brute is unable to do this. Instead Brute finds the program which deletes all but the character in the string and cannot find a better program after this.

There are tasks that Brute is unable to solve because the wanted program cannot be formed using only the tokens invented by Brute. There were also a few tasks for which only a small number of training examples were given and Brute found a program that produced the correct output for the training examples, but not for all the test examples. In all the other cases, Brute gets stuck because of a local optimum.

MUTE

There are quite some tasks that MUTE was able to solve even though Brute got stuck in a local optimum.

One example of a task where MUTE is able to escape a local optimum is Task 5 in which only the last four digits need to be kept from a string of digits. E.g. `"22022002"` has to be transformed to `"2002"` and `"1252010"` has to be transformed into `"2010"`. After iteration 4, the best program found is *[Drop]*, after iterations 709, 3593 and 18566 the newly found best programs are respectively *[Drop, Drop]*, *[Drop, Drop, Drop]* and *[Drop, Drop, Drop, Drop]* and in the 41288th iteration the final program, *[While(NotAtEnd [MoveRight]), MoveLeft, MoveLeft, MoveLeft, MoveLeft, While(NotAtStart [Drop, MoveLeft]), Drop]*, is found.

Another task that demonstrates the capabilities of MUTE is Task 29 in which Mute needs to keep only the name of the month in a string representing a date which also contains some numbers and other characters. E.g. `"June 20 - 2002"` should be transformed into `"June"` and `"2007 (September)"` into `"September"`. MUTE is able to escape multiple local optima after which it finds a program that solves the task.

The made observations clearly demonstrate that the answer to **Q1** is yes.

I also analyzed the results of the tasks that MUTE was not able to solve. I found four causes of these failures. First of all, there were three tasks for which the tokens invented by MUTE were insufficient to be able to solve the task. One task needed the addition of a space character and the other two required more complex *While* tokens that contained *If* of other *While* statements in their bodies. Secondly, there were some tasks for which MUTE found a program that solved the training examples, but not the test examples. These were mainly trials that provided only one I/O example as training data and one trial that provided 5 I/O examples as training data. Thirdly, there were a few tasks that could not be solved because the invented tokens required to solve them were removed since they showed no potential in the first few iterations they were used in. Lastly, there were tasks for which MUTE simply needed more time to be able to solve them. The latter was the case for the majority of the trials that could not be solved, especially for complexities 5 and 9.

Even though this analysis shows the potential of using an MCTS-based algorithm, it should also be noted that the initial version of MUTE did not show this potential. MUTE was able to perform so well, only after different enhancements were made. This means that other IPS algorithms might also benefit from similar enhancements like the removal of invented tokens that show no potential and the removal of programs that are similar to programs that were found before. This would probably also increase the performance of Brute because local optima would be fully exploited after a certain number of iterations. Therefore, if similar programs were to be removed, Brute would eventually move on to exploit the next best program.

7 Conclusion

This paper searches for an answer to the questions **Can a new IPS system that uses MCTS guided by a heuristic-based loss function, escape local optima?** (**Q1**) and **Can a new IPS system that uses MCTS guided by a heuristic-based loss function, outperform other IPS systems?** (**Q2**). For this purpose MUTE, a new IPS system that uses MCTS guided by a heuristic-based loss function, was designed and its performance was tested and analyzed.

MUTE's performance was tested on three diverse domains, namely ASCII art, robot planning, and string transformation. In ASCII art, MUTE was outperformed by Brute. In robot planning, no local optima occurred and MUTE was a 100 percent successful in solving the tasks just like the four other IPS systems. On the string domain, MUTE showed its potential and gave a first indication that the answer to **Q2** is yes.

Two enhancements seemed essential for this success of MUTE. The first of these enhancements is the removal of tokens that show no potential. The second enhancement is the removal of similar programs. Both enhancements reduce the branching factor a lot, allowing MUTE to spend more time exploiting the remaining branches. Other IPS systems might also benefit from similar enhancements.

To get more insight into the behaviour of MUTE and Brute, an analysis was done on their data on a selection of string transformation tasks. This data showed that MUTE is able to escape local optima, where Brute gets stuck. This means that the answer to **Q1** is yes.

8 Future work

This study offers a lot of starting points for further research.

First of all, MUTE's performance can be further tested and analyzed. A manual analysis of its results for ASCII art could be conducted, similar to the analysis that was conducted for string transformations. This might lead to ideas on how to further improve MUTE, or reveal properties of the type of domains that an MCTS-based IPS system simply will not be successful for.

Secondly, a lot of enhancements can be found in the MCTS research field of which some might lead to improvements of MUTE and other IPS systems. For example, parallelization (Browne et al., 2012) would allow MUTE to do more iterations in the same amount of time, which could lead to solving more tasks without needing more time.

It would also be interesting to see if the token score enhancement can be used to improve other IPS systems. Using the token score in other smart ways, for example by using it in the selection policy might also be worth researching.

The last idea I would like to offer is to go back to using the original version of MUTE, which (in theory) was able to form complex programs thanks to the use of incomplete tokens. When this version would be enhanced with the removal of similar programs and the removal of tokens that show no potential, the reduced branching factor might lead to increased performance.

9 Responsible Research

As part of this research, I followed lectures on conducting research in a responsible way. During these lectures different good as well as bad research practices were discussed. In this section, I will discuss my efforts for conducting responsible research based on three topics that were discussed during these lectures.

One of the topics I kept in mind while writing my paper was the bias in science towards positive results. In the lectures about responsible research, it was discussed that there is a preference in science to producing positive results. This is unfortunate since it is very important that negative results are also published because these also contain important insights. Mehta (2019) advocates the publication of negative results and states that “When negative results aren’t published in high-impact journals, other scientists can’t learn from them and end up repeating failed experiments, leading to a waste of public funds and a delay in genuine progress.” This is why in my paper, I describe the whole process of designing MUTE and not just the final implementation. The first version of MUTE seemed unsuccessful in escaping local optima and this can be considered a negative result. In my paper, I still describe this initial version as well as the observations that I made while testing it. This way others can also use the insights that I gained by building and testing this initial version of MUTE. After building the initial version, I did different attempts to improve MUTE. Some of these attempts were successful, others were not. Again, I described both the successful and the not-so-successful attempts as well as the insights I gained.

During my research, I also highly valued the principle of scrupulousness. Results and observations made during research are most valuable if they are obtained using correct methods. Experiments need to be designed, set up and conducted with care. In designing and setting up my experiments, I wanted to make sure the results would provide answers to my research questions. To ensure this, I defined my research questions before deciding what experiments I would conduct. With my peer group and supervisor, we discussed and wrote down different possibilities for doing experiments and collecting data. Once I decided on my research questions, I decided what experiments and data would be most useful to answer these questions. Another action I took to ensure the quality of my results was that, together with the students in my peer group, I made a selection of data that would be used for testing and comparing our different algorithms. I made sure I used different data when training MUTE or doing intermediate testing. This way overfitting could be prevented.

The last topic I will discuss regarding responsible research is transparency and reproducibility. In my paper, I tried to make my reasoning and methods as clear as possible. I do this by chronologically taking the reader with me in the whole process. In the introduction, I give the motivation and outline for the research. After this, I give an overview of the literature that formed the base for my reasoning in the rest of the paper. Next, I describe the process of designing MUTE. I try to provide the reader with all the relevant implementation details so that anyone interested would be able to implement MUTE themselves. I also published the implementation code and referenced it in my paper, so anyone interested can access and use it. In the experiment section, I describe how I test my final version of MUTE. I clearly state the source of the test data as well as what data exactly I retrieve and analyze. Together with the implementation code, this allows others to reproduce the results I obtained.

References

Azimzade, F., Jenneboer, B., Matulewicz, N., Rasing, S., & van Wieringen, V. (2022). *Bep_project_synthesis.* `https://github.com/victorvwier/BEP_project_synthesis`. GitHub.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1–43.

Chaslot, G., Bakkes, S., Szita, I., & Spronck, P. (2008). Monte-carlo tree search: A new framework for game ai. *AIIDE*, *8*, 216–217.

Cropper, A., & Dumančić, S. (2020). Learning large logic programs by going beyond entailment. *arXiv preprint arXiv:2004.09855*.

Gaudel, R., & Sebag, M. (2010). Feature selection as a one-player game. In *International conference on machine learning* (pp. 359–366).

Gelly, S., & Wang, Y. (2006). Exploration exploitation in go: Uct for monte-carlo go. In *Nips: Neural information processing systems conference on-line trading of exploration and exploitation workshop.*

Gulwani, S. (2010). Dimensions in program synthesis. In *Proceedings of the 12th international acm sigplan symposium on principles and practice of declarative programming* (pp. 13–24).

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning* (pp. 282–293).

Lim, J., & Yoo, S. (2016). Field report: Applying monte carlo tree search for program synthesis. In *International symposium on search based software engineering* (pp. 304–310).

Mehta, D. (2019). Highlight negative results to improve science. *Nature*. doi: https://doi.org/10.1038/d41586-019-02960-3

Pellier, D., Bouzy, B., & Métivier, M. (2010). An uct approach for anytime agent-based planning. In *Advances in practical applications of agents and multiagent systems* (pp. 211–220). Springer.