Improvements in Monte Carlo Tree Search for Inductive Program Synthesis

Nathalie van de Werken
Supervisor: Sebastijan Dumančić
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

A recent development in program synthesis is using Monte Carlo Tree Search to traverse the search tree of possible programs in order to efficiently find a program that will successfully transform the given input to the desired output. Previous research has shown promising results as Monte Carlo Tree Search is able to escape local optima that occur during the search. I have continued this previous research by changing some components of Monte Carlo Tree Search and testing them on three different domains; robot planning, string transformations and ASCII art.

Most notable, I have found that by changing the exploration constant C_p to slowly decrease during the running of the algorithm, you can improve the algorithm's accuracy. This makes it easier for the algorithm to escape local optima, however here it is crucial the parameters are tuned well. Furthermore, I have also found that improvements can still be made in the expansion step of MCTS. However, changes to which values are backpropagated have not shown an improvement in the accuracy of MCTS in program synthesis.

1 Introduction

Program synthesis is a process in which code is being automatically generated for the user given some high-level specification [Gulwani *et al.*, 2017]. This process makes it possible for people without programming knowledge to create computer programs and can also speed up the job of developers. It is currently already used in software engineering, biological discovery and data cleaning [David and Kroening, 2017]. Although many improvements have been made over the past years, there is still a lot of progress that needs to be made in order for program synthesis to be feasible for most programs, especially more complicated ones [Cropper and Dumančić, 2020].

Many different versions of program synthesis exist, but what is at the core of each of them is the way it searches. Here program synthesis looks through all the different programs possible given the syntax and executes it on the given input to check if it indeed gives the expected output. All these different programs can be represented in a tree data structure, of which at the root an empty program and following are the children with similar programs, just one operation added. Usually, this tree is very big or even infinite, even when we define a very limited language. This means we cannot easily just search through the entire tree using a simple tree traversal algorithm such as Breadth-First Search or Depth-First search. Because of this, it is very important a good algorithm is used for this search, as a bad decision early on can lead you to completely go down the wrong branch and therefore it taking very long for you to find a program that can indeed solve this task. Here is where more complicated searching algorithms come in such as Monte Carlo Tree Search. This algorithm traverses through this tree in such a way that it balances exploration and exploitation such that we can quickly find a program that solves all the input cases.

There has already been done some research in the field of program synthesis in combination with Monte Carlo Tree Search. Most notable is the research by Matulewicz, where Monte Carlo Tree Search was implemented to work on program synthesis, called MUTE, and tested on three domains [Matulewicz, 2022]. This will be the basis of this research paper and in this research paper I will change several components in this algorithm to answer the question: **"How can we improve the different components of Monte Carlo Tree Search to make it perform better on program synthesis?"** In order to answer this main research question, I will answer the following sub-questions:

- **Q1** What does it mean to perform better on program synthesis?
- **Q2** For each component that has been researched, what can we change to it to make it run better?
- **Q3** What is the optimal combination of the different components?

To answer these questions, the paper will be structured in the following way. In section 2, I will explain the most relevant background and related work on which I will be continuing my research. Following this, in section 3, I will go through the methodology and explain all the different components I have changed. In section 4, I will explain how I ran the experiments, of which I will describe the results in section 5. In section 6, I will give the conclusions and also do some recommendations for future work. Finally, in chapter 7, I will give some final remarks on how I have taken responsible research principles into account for this paper.

2 Background and Related Work

In this section, I will explain the most important terms you need to know to fully understand this paper. Furthermore, I will also explain the most important past work that has been done in program synthesis, Monte Carlo Tree Search and combining the two.

2.1 Program synthesis

In the simplest of words, in program synthesis we want to automatically generate a program that can translate the given input into the desired output. This means that someone then does not need to write code or anything but from a higher level define the input and output and get a final program [Solar-Lezama, 2008]. A concrete example in the domain of string transformations is for instance if for the input you give Harry Styles and your desired outcome is H. Styles, you want to generate a program that does this automatically and can also generalise to a case such as Taylor Swift becoming T. Swift.

Now that we have an understanding of the main goal, many different algorithms can solve the problem of program synthesis. The field is still rapidly expanding with improvements to older algorithms, as well as completely new topics and algorithms [Polozov, 2018]. Some notable algorithms that have been experimented with are genetic programming [Helmuth

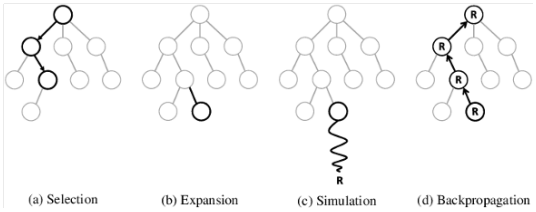

Figure 1: The four components of Monte Carlo Tree Search [James *et al.*, 2017]

et al., 2018] and functional approaches [Kitzelmann, 2009]. In this paper, I will focus on Monte Carlo Tree Search, a searching algorithm that balances exploration and exploitation.

BRUTE

I would like to especially highlight the work of Cropper and Dumančić, who used a best-first search algorithm, called Brute, for program synthesis [Cropper and Dumančić, 2020]. This algorithm works in two phases, an invention phase and a search stage. In the first stage, all sequences of tokens up until a certain length are constructed. Then in the second phase of the algorithm, we start by making a priority queue and adding the empty program to it. Then every iteration, we pop the first element in the priority queue with the best score until now, altering the program by adding one of the tokens found in stage 1 and adding it again to the queue.

This method has proved quite effective in the three domains of ASCII art, string transformations and robot planning. However, the main struggle of Brute is local optima, due to the fact that using best-first search is very prone to local optima.

2.2 Terminology used

Since doing program synthesis on a programming language such as Python or Java would lead to a very big search space, of which the majority will not lead to a correct result, we run program synthesis on a limited language. In this limited language, what will be referred to as **program** is known as a sequence of transitional tokens.

There exist two different kinds of tokens, a transitional token and a boolean token. A **transitional token** takes as input the state and returns the state after the specified operation is executed. A **boolean token** also takes as input the state but instead returns a boolean which describes whether the specified predicate is true for that current state. All of these tokens are defined specifically for each domain and will be explained in section 4.3. There also exist two general transitional tokens, and *If* and *While* token, which uses the boolean tokens and transitional tokens to synthesize more complicated programs. These two tokens are available for all the domains.

A big challenge in programming synthesis is that of **local optima**. This occurs when the algorithm is stuck searching in an area which performs very well on the heuristic, but it is very difficult or even impossible to turn that program into a program that actually solves the task.

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search is an algorithm that in the past has mainly been used in games, where the search space expands with a large factor on each level of the tree [Chaslot *et al.*, 2008]. Here its main strength is how it is possible to balance exploration and exploitation.

MCTS is an algorithm that works in iterations, where each iteration has four stages. For an overall overview of how a single iteration works, see figure 1. In the first stage **selection**, we select which node we will want to explore in the next iteration. There are many algorithms that can be used to select it, but the one most commonly used is known as UTC [Kocsis and Szepesvári, 2006]. This means the following formula is used for selecting the node:

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

Here $\bar{X}_i$ is the average reward received in node i, C_p the exploration constant, n the number of visits to the parent of node i and n_i is the number of visits to i.

This value is calculated for each node and the node with the maximum value is chosen to explore next. Then in the next stage, **expansion**, we take the current node and explore it by taking another move. In the third stage, **simulation**, the node is expanded until a terminal state is reached, such that we can calculate the performance of the done expansion. Finally, in the last stage of **backpropagation**, the information is reflected back up in the tree all the way to the root.

2.4 Monte Carlo Tree Search in Program Synthesis

The combination of Monte Carlo Tree Search and Program Synthesis has been a recent development and therefore not a lot of research has been done. I have been able to find two previous times that Monte Carlo Tree Search has been used in combination with Program Synthesis, which I will describe in the following section.

First of all, a field report has been implemented as to see if it is possible to use Monte Carlo Tree Search in the domain of Program Synthesis [Lim and Yoo, 2016]. Here they have implemented MCTS using UTC for Java Bytecode and have also implemented a genetic programming algorithm to compare the performance. They found Monte Carlo Tree Search to be effective in the smaller domains, however, it does struggle with more difficult domains and longer problems. They also found that "It appears that MCTS commits to an instruction that yields moderate rewards and keeps exploiting it, when in fact its rewards are suboptimal." [Lim and Yoo, 2016]

Furthermore, there has also been done a study by Matulewicz, where MCTS is implemented for program synthesis in Python. UTC is used for selecting the next node, and in the expansion step a complete token is added to the program, which can consist of multiple tokens if for instance when an *If* or a While is added. Then in the simulation step, the loss of the current program is computed on the example program, and finally, this loss is then propagated back into the tree. Two big improvements were made that significantly reduced the branching factor and therefore enhanced the performance,

namely the removal of tokens that did not show any potential as well as the removal of similar programs. Analysis has shown that this version of program synthesis can indeed escape local optima and it has shown promising results for using Monte Carlo Tree Search in the future of program synthesis.

3 Methodology

In this chapter, I will explain for each component which alterations have been done and why these alterations show the potential to perform better in the domain of program synthesis.

3.1 Balancing exploration and exploitation

The previous version of Monte Carlo Tree Search had a single constant that was used to balance exploration versus exploitation. This constant, C_p, is known as the exploration constant and is hypertuned for each domain to lead to optimal results. This comes with quite some downsides, as first of all, you need to tune this parameter with a big number of example programs before you can actually use program synthesis in the real life. Furthermore, at the beginning of the search, you want more exploration, so you do not get stuck in some local optima early on and struggle to escape it. This means having more exploration in the first couple of iterations and after a while focus more on getting a final program and therefore exploit the best nodes. Therefore I have implemented multiple ways in which the C_p is changed throughout the process as to see if it will lead to more optimal results.

Linear change

First of all, I tried out a way of slowly decrementing C_p using a linear term throughout the process. This means that after every iteration, I decrement it with a set amount. The way this is implemented is that we define a beginning value, known as $C_{p,maximum}$ and ending value $C_{p,minimum}$. Initially, C_p is set to $C_{p,maximum}$ and then slowly it will per iteration change by a set amount. It has been implemented in such a way that C_p cannot go under $C_{p,minimum}$.

I have first of all chosen this implementation since it is very simple and you can easily see the effects of these measurements. Despite the simplicity, it does give you a lot of freedom as you can manually decide the beginning and end value. This allows for us to easily change this implementation and also make its influence on the overall performance larger or smaller. This should mean we can modify the algorithm so it works well for each domain. It does come with the downside of there being more hyperparameters to tune, however, I have tried to minimize this impact by using the values from Matulewicz as the basis of my search.

Less frequent linear change

Such little changes from constant linear change might have little impact when the algorithm is running, so to combat this I have also implemented a way in which C_p is not changed after each iteration but instead after each X iterations by a bigger amount. I have chosen that we will change C_p a 100 times, so we will bridge the gap between $C_{p,minimum}$ and $C_{p,maximum}$ in a 100 steps, as this will still allow enough changes and flexibility, but has much contrast with the first alternative of constant linear change.

Exponential change

As linear change might have the problem that we are stuck with too high of a C_p for too long and therefore do not reach a lot of progress early on, we can also try an exponential change over time. This means that first of all the constant starts high but it lowers very quickly and stays low for the rest of the iterations. This can in theory perform better, but also makes the tuning of the parameters more tricky as you also have to tune how fast the exponential should lower. In order to implement this, I have used the following formulas:

$$C_{current} = \log((C_{p,maximum} - C_{p,maximum})^{current\_level}) \tag{2}$$

with current_level defined as

$$current\_level = 1 - min(\frac{i}{i_{tot}}, 1) \tag{3}$$

where i is defined by at which iteration we currently are and i_{tot} is the amount of total expected iterations. There is a minimum function in there in case we go over this number such that we do not reach a negative value of $current\_level$.

Getting out of local optima

A big issue in program synthesis is that of plateaus in which we get stuck in local optima. In order to combat this, we would like to first of all notice that we are in a plateau and then try to explore more as to come out of it. The way I have tried to find a way to easily escape these plateaus without having to completely reset the entire search is by increasing C_p a lot when we have been stuck in a plateau for a long time and bringing it back down again when we are out of it. You can notice that you are in a plateau when for X iterations, you have had the same accuracy without any improvements, where this X has to be set for each domain since for some domains plateaus are more frequent and we do not want to change C_p too soon as we would want sufficient time to explore all the good nodes first.

I have decided that instead of increasing C_p by a set value, it will increase it by 1/100 of the exploration constant each iteration more than 5000 that we have not had a small improvement. This way it is more domain-specific but it does not require another hyperparameter to be tuned, as in a domain where C_p is set to a lower amount, it will then also grow slower. Also this way it is gradually increased and does not spike all of a sudden too much, which could cause all the previous progress to be lost. Once we are out of the local optima, C_p will be reset back to the original value.

3.2 Expansion and Simulation

In the unaltered state of Monte Carlo Tree Search after selecting which node to expand on, all possible tokens that can still be added are enumerated and the one that is first in the list is chosen. Then the loss of this current program is computed and we will continue the final stage of the Monte Carlo Tree Search, backpropagation. For this stage, there are many different ways to alter how to expand the tree. However, it is also necessary to not overfit on the domain and choose two specific ways to compute the way for the expansion, as this can allow the probability of getting into local optima to grow.

Randomly choosing which operation
In the unaltered implementation, always the first element in the list of possible tokens is chosen. Due to how the list is composed, which is always in the same order, it might be possible that a necessary token for that program to succeed is only at the end of the list. Because of this, it is possible that this token is not explored until very late in the process. In order to combat this, I have implemented a variation of the expansion algorithm that first shuffles the list of possible extensions and then removes the first one.

Using token's past success
It is a reasonable assumption that the tokens that are used in the up until now best current program are often useful, and therefore tokens we want to use to expand on. Something that one should be careful of however in this case is that this means that local optima can be more likely to occur. However, I still deemed this an interesting alteration to look into.

In order to implement this, I take into account the token's past scores which are stored in a map and choose the tokens based on such a that the token with the highest score is chosen to expand with. Although this can lead to local optima, as it might overfit on some tokens that seem to generate good results according to the heuristic, it will also speed up some obvious steps where one token is the right choice so can still improve performance.

3.3 Backpropagation

As explained in section 2.3, backpropagation is a set algorithm in which we need to reflect the result of the final rollout in the rest of the tree, so in the next iteration, the new best leaf is chosen. This algorithm is constant in all different versions of Monte Carlo Tree Search since it is crucial for the algorithm to succeed that the value is backpropagated in the entire tree. The only degree of freedom you have is which value you backpropagate, so I have come up with two different alternatives.

Loss
Instead of just backpropagating the reward, defined by the current loss divided by the maximum loss, you can also simply backpropagate the current loss itself. This has as benefit that it is easier to compute, so therefore should give a small speedup in theory as well as there being a bigger range in which the values can be. Therefore, there might be bigger differences which will lead for the algorithm to sometimes choose to visit different nodes, which can be beneficial for the search.

Normalised between -1 and 1
Monte Carlo Tree Search has been developed originally for games, as can be seen in some of the original papers [Coulom, 2006] [Gelly *et al.*, 2006]. Here it is customary that it is implemented using a zero-sum game, which means that one play's win is the other player's loss[Blakely, 2021]. This means that either the value +1, 0 or -1 is backpropagated in the tree.

Since this is how Monte Carlo Tree Search is often used like this, it is interesting to see how it will perform in Program Synthesis if it has similar constraints. It might perform then better because a lot of the research and optimisation done in Monte Carlo Tree Search is based on this. The way I have implemented this is by comparing the new loss score with the current best from the parent, if it has improved, you backpropagate +1, if the loss is the same, you propagate 0 and else you propagate -1.

3.4 Combination of factors

After I have run my experiments to find the optimal performance of each of the components, I will do a final round of experiments where I combine each of these optimal components to see the final potential of Monte Carlo Tree Search in Program Synthesis. Although I do expect to see some improvements when I am running all these optimal factors combined, this may be less than expected since these factors will not be tuned to each other.

4 Experimental Setup

In order to answer the research questions, I have run many experiments using the different components explained in the previous section. In this section, I will further explain how these experiments are set up, on which domains these were run and also on which metrics I have compared the different alternatives.

4.1 Set up experiments

As a basis for this research, the code from Matulewicz was used [Matulewicz, 2022]. Here Monte Carlo Tree Search for Program Synthesis has been implemented in Python. I have altered parts of this code to try out the different variations described in section 3. To debug the code, I ran it locally on my computer on the robot domain. For the final tests, a case is considered solved if both the train and test cost are 0.

DelftBlue
In order to run the final experiments, I have been granted access to the DelftBlue, a high-performance computer [Delft High Performance Computing Centre (DHPC), 2022]. On this computer, there are 218 nodes available with 48 cores. The CPU available is 2 times the Intel XEON E5-6248R 24C 3.0GHz, with 192 GB memory and 480 GB SSD.

To run the experiments, you need to have a job script that contains how to set up the environment, the parameters of what you're going to run, such as how many CPUs you need and the maximum time allowed, as well as which script you want to run. The main script I used to run the experiments can be found together with the code, and here you see that I allowed the job to take a maximum of 5 hours, a maximum of 4 GB memory per CPU, and used 46 CPUs as to make the computation go as quickly as possible. A single program synthesis task is allowed to take a maximum of 60 seconds after which it will time out.

4.2 Metrics

As to answer Q1 ("What does it mean to perform better on program synthesis?") I have spent a significant amount of time looking at other research on program synthesis and which metrics they used there to compare the outcome of different parameters or algorithms. Here I found that the most

important metric to compare on is the accuracy, meaning how many of the tasks have been solved successfully. You can also compare the average loss of the examples that could not have been solved. However, I do not think this is a very useful metric, because first of all, it is possible for this loss to be a low number but still has completely the wrong result, as well as even if the result is off by a little bit, it is still off and therefore you still cannot use this program that has been synthesised.

Some other studies have looked a lot into the different plateaus that can occur during program synthesis [Koenig *et al.*, 2021]. Although this is very interesting to look at and can give you some insights into the process of program synthesis, it is outside of the scope of this current research, as I prioritized getting data on a lot of different examples in the three different domains compared to looking more into depth in how specific examples are synthesised.

Lastly, it is very important we also look at the speed of how fast the program is found that solves that task. If this takes a very long time, even though the accuracy of the algorithm is higher, it might never be used in practice because the resources to run that algorithm are simply not there. I only look at the average time of the cases it actually managed to solve, as otherwise most of the cases that were never solved timed out and this will mean that the average time is dependent on the accuracy, which is not what we want to analyse.

For the final experiments of running the optimal combination of parameters, I chose to also look at how the accuracy fluctuates over the difficulty of the domain. For the robot planning domain, this difficulty is denoted by the grid size, for string transformations this is the number of training examples and for the ASCII art domain, it is the number of symbols.

4.3 Different domains

I have run these experiments on three different domains. This has many advantages, as it gives us a more accurate picture of the performance of the component since simply there was more data to test on. Since there is randomness involved in some of the components, running more experiments will reduce the standard deviation. It also shows us how Monte Carlo Tree Search will perform on both less and more complex domains. Furthermore, it also allows us to not completely overfit on a single domain and it will give us more of an idea of how these techniques will generalise to program synthesis as a whole.

Robot planning

In this domain, for an example program, you have been given a grid, where the goal is to move the robot around in such a way that it picks up and drops the ball in the desired place. The loss function used in this domain is the minimum number of moves needed for the robot to navigate to the ball, pick it up and drop it off. This is a very informative heuristic, which makes the chance of local optima very unlikely to happen since it is impossible to score well on this heuristic and not be close to the final solution.

The boolean tokens for this domain are *[AtRight, NotAtRight, AtLeft, NotAtLeft, AtTop, NotAtTop, AtBottom, NotAtBottom]* and transitional tokens are *[Grab, Drop, MoveRight, MoveLeft, MoveUp, MoveDown]*.

String transformations

For string transformations, the goal is to take the input string and transform it into the correct output string. For a single program, between one and nine example strings have been given as input. The heuristic that is used in this domain is Levenshtein's distance. In this metric, we define the difference between two strings to be the minimum amount of single-character edits needed to transform one string into the other [Yujian and Bo, 2007]. Although this is already a very informative heuristic, it does allow for strings that cannot easily be transformed from one to the other to score well, for example turning the string ABCDEFG into A, a program that transforms it into the empty string, such as While(NotAtEnd, drop) will still perform very well according to this metric. This can allow for local optima to occur.

For this domain, the transitional tokens are *[Drop, MakeUppercase, MakeLowercase, MoveRight, MoveLeft]* and the boolean tokens are *[AtStart, NotAtStart, AtEnd, NotAtEnd, IsLetter, IsNotLetter, IsUppercase, IsNotUppercase,IsLowercase, IsNotLowercase, IsNumber, IsNotNumber, IsSpace, IsNotSpace]*.

ASCII art

In the ASCII art domain, the goal is to turn an empty grid into a filled grid which will have one or several ASCII characters encoded in it. The heuristic that is used in this domain is the binary distance between the current output and the desired output. The main challenges of this domain are the size of the grid, as well as the heuristic not always being very informative, as it is possible the heuristic gives a high score to a program but expending on this program will never lead to a correct program since that one is impossible to reach from this node.

The transitional tokens for this domain are *[Draw, MoveRight, MoveLeft, MoveUp, MoveDown]* and the boolean tokens are *[AtRight, AtLeft, NotAtRight, NotAtLeft, AtTop, AtBottom, NotAtTop, NotAtBottom]*.

5 Results and Discussion

This section contains the results from the experiments described in section 4. Per changed component, I will analyse its performance and offer possible explanations as to why these results were reached. In figures 2, 3 and 4 you can see the performance of each of the components in each of the domains respectively. Note that some of the different components in the graphs fall together, and in tables 1, 2 and 3 you can see the final scores for each of the domains.

In general, I noticed that some solutions have a low accuracy as well as a low average time. This is because only the easier tasks in the domain have been solved, which takes less time to solve as the length of the programs is lower. This means that you should also take into account when you are comparing times which of the tasks have been solved.

5.1 Balancing exploration and exploitation

As can be seen from the graphs, many different setups have been run for each component to test its performance. Notable

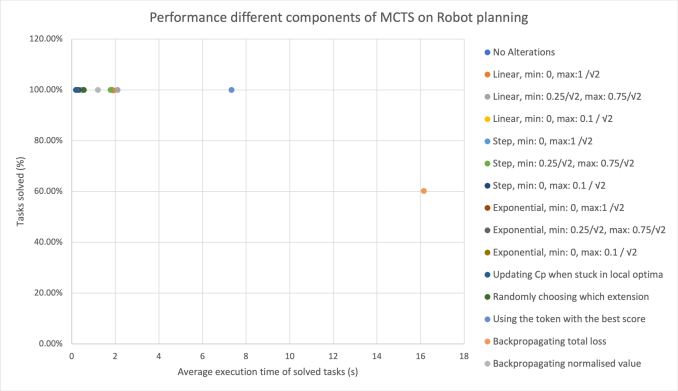

Figure 2: Accuracy versus time in the robot planning domain for all tried out components

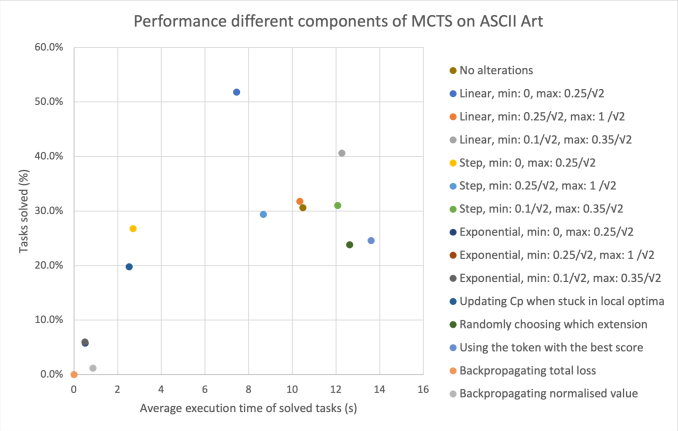

Figure 3: Accuracy versus time in the ASCII art domain for all tried out components

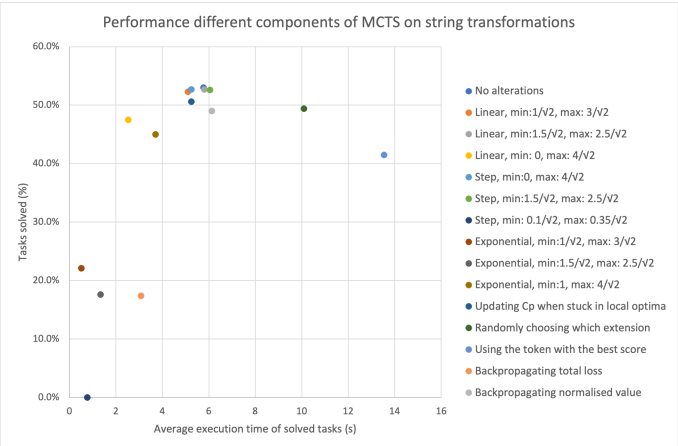

Figure 4: Accuracy versus time in the string transformations domain for all tried out components

is how much the different parameters influence the overall accuracy, as when you have well-tuned parameters this can really elevate the performance of the algorithm. This really highlights the importance of tuning your parameters well.

You can see that in all three domains, the best performing result is the linear function. I believe the constant updating indeed gives it the edge over the step function, as this additional freedom allows for even more exploration.

The exponential formula had the opposite of the desired result and instead really hindered the performance. I believe this is due to the fact that the parameters were set too high and therefore too long was spent with too little exploration. Furthermore, the decay is very fast, while having some exploration, later on, is still useful since this is when you run into local optima. Therefore I believe it is useful for in the future to look into other exponential formulas, where decay is slower.

Something that had a lesser effect than expected is checking for plateaus and then increasing C_p in order to get out of local optima. I believe this could be a case of starting the incrementing too early or too late, such that it has too limited influence. Furthermore, if you are really stuck in a local optimum, increasing this constant might be too late and you will remain stuck in the local optima. Instead, there might be more useful techniques at that point to get out of this local optima, for instance pruning that part from the search space or completely resetting the search.

5.2 Expansion and Simulation

First of all, I saw that randomly choosing which token to choose instead of orderly going through the list did not improve performance, but it also did not hinder it. Therefore I believe this is not an issue that Monte Carlo Tree Search is running into. In the ASCII art domain it even decreased the accuracy by a little bit, which I believe is due to the fact that there in the way the tokens are sorted, the simpler tokens are first, which you want to use more often in the final solution since they are more general, and therefore using a regular ordering will improve the accuracy of the algorithm.

Finally, using the token with the best score has not increased the accuracy. This is mainly due to the fact that this way it is easier to get stuck in local optima as well as overfitting on past data, which might not be accurate for this current example and therefore might decrease performance.

5.3 Backpropagation

Using a different value to propagate has significantly decreased the performance of Monte Carlo Tree Search. This is due to when you do not normalise the rewards, the values get too big very quickly and therefore you will not visit any node if it was not visited in the first few iterations, as after that the difference in score between an already visited node and an unvisited node is too big and this gap will never be bridged.

By just returning -1, or +1, we did not run into this same problem, but as a result, way fewer data is backpropagated through the tree and there no longer is a distinction between a marginally better expansion and a way better, so therefore

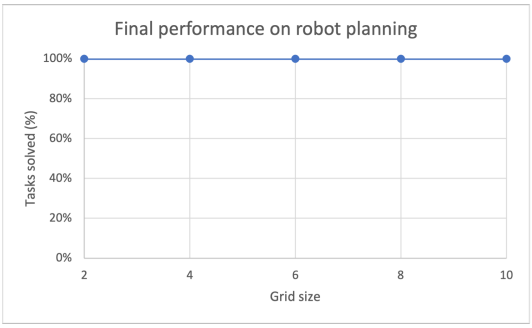

Figure 5: Accuracy versus grid in the robot planning domain for optimal combination

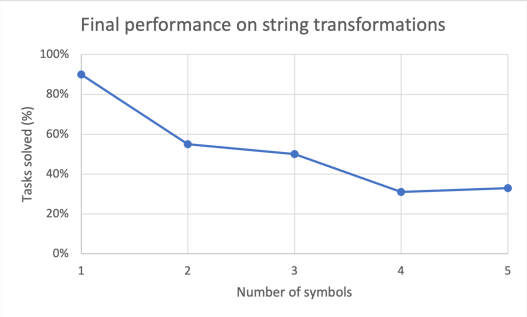

Figure 6: Accuracy versus grid in the string transformations domain for optimal combination

the algorithm has to make way more uninformed decisions. Herefore, the accuracy of the algorithm suffers.

5.4 Optimal combination

Finally, for the optimal combination, you can see that the accuracy has improved. For our first domain, robot planning, as the original version of MCTS could already solve all the examples, therefore no big improvements have been made. Something that has improved however is the time it takes to solve the cases, as allowing for more exploration early on, allows for a speed up in the computation.

For our second domain, string transformations, we have improved the performance of MCTS marginally. Something notable is that as the number of examples increases, we can also see the accuracy go up. This means that Monte Carlo Tree Search does not struggle with having more examples and overfitting on just one and then not being able to generalise to the other cases.

In the final domain, ASCII art, we have made bigger improvements to the final accuracy. I believe this is because we have shown in this domain that by using a different C_p throughout the exploration, you have a lot more freedom and it really is able to get out of local optima and solve way more cases. Due to this, now it is also possible to solve problems that have more than 2 symbols in there, which was not possible in the past.

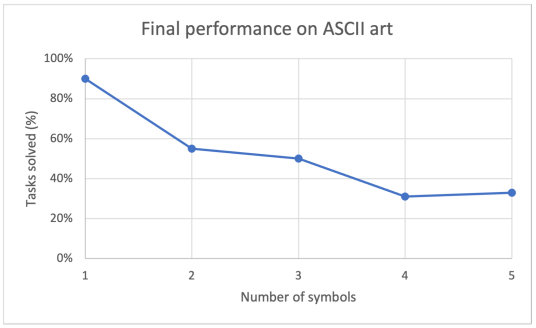

Figure 7: Accuracy versus grid in the ASCII art domain for optimal combination

6 Conclusions and Future Work

In this paper, we aimed to answer the question **"How can we improve the different components of Monte Carlo Tree Search as to make it perform better on program synthesis?"**. In order to answer this question, I, first of all, answered "What does it mean to perform better on program synthesis?" I have found that the most crucial aspect to compare the different components on is the accuracy of the program synthesis, as this gives you an overall understanding of how well the program synthesis performs, in combination with time as this way you know how fast it will have found the solution.

For our second research question, where I tested "For each component that has been researched, what can we change to it to make it run better?", I have made several changes to the selection component by trying out different ways to balance exploration and exploitation. Here I found that big improvements can be made to the overall accuracy, especially when you are using an algorithm that allows C_p to decrease linearly, where the parameters of $C_{p,mimumum}$ and $C_{p,maximum}$ are hypertuned to the domain. Furthermore, many improvements can still be made to the way we select how to expand the problem, but my alterations did not make a significant difference in the accuracy of the program synthesis, as they often lead to local optima. Lastly, by changing which value we backpropagate no improvements were made, as this only decreased the amount of data we allow the algorithm to learn from, which did not improve the accuracy.

When I combined all of these optimal parameters, I found that the accuracy has improved compared to no alterations. This means that I believe Monte Carlo Tree Search can be a feasible technique to use for program synthesis in the future, and it is definitely worth it to investigate it more since many more alterations can be made.

6.1 Future work

As there are many different variations in Monte Carlo Tree Search, there are still many things that can be experimented with in combination with program synthesis to see if it can improve the accuracy and speed of the creation of programs.

Most notably, I believe that we can make a big improvement in both accuracy and most importantly time to get there if more research is done in parallelizing the search such that multiple simulations can be run at the same time. Previous

research has shown that multiple versions of parallelization can be beneficial in the domain of GO [Chaslot *et al.*, 2008]. I believe this to be a similar case for program synthesis, since different kinds of parallelization will make it less likely for the algorithm to get stuck in local optima and explore more nodes at the same time.

Furthermore, for the selecting step, I think there are still big improvements that can be made to more effectively select a good node. I believe that maybe we can add some data structure that keeps track of how many times certain tokens have been used in the most successful programs and use this data in deciding how we will modify the program, by choosing to use those tokens with a higher probability than the rest or the opposite.

Especially since the field of Monte Carlo Tree Search in program syntesis is so new, not a lot of research has been done until now. There are still many other alternatives of Monte Carlo Tree Search [Świechowski *et al.*, 2021] which have not yet been tested out in Program Synthesis. For future research, we can try to more vastly alter the algorithm of Monte Carlo Tree Search and then see how this will impact the performance of the algorithm.

Lastly, although I believe that Monte Carlo Tree Search has shown very promising results for program synthesis, it will not always be the solution to each domain and give optimal results. There are still many different algorithms that have shown promising results when it comes to program synthesis. I believe it would be very useful if more research is put into combining different methods for finding the optimal program synthesis algorithm. This could include using Monte Carlo Tree Search for some promising programs and then using genetic algorithms to combine these and find an optimal result. This would make a more robust algorithm that can solve many different kinds of problems in many different domains, but it does require more tuning.

7 Responsible Research

In research, it is very important that we take into account different ways to do the experiments and the reporting of it as ethically as possible. Because of this, I have attended some lectures about responsible research as part of the research project. In this section, I will be highlighting the different ways I have taken responsible research into account while conducting my research.

7.1 Transparency

In research, it is very important that you are transparent in your research and the way you have conducted it. Therefore, in this article, I have included all the different experiments I ran, as well as tried to write this paper in such a way that people can easily understand why certain decisions have been made and also recreate some of the experiments if they see fit.

7.2 Reproducibility

One of the most troubling aspects in my research is the ability to reproduce these results. This is mainly due to the fact that these experiments are run on the supercomputer of TU Delft, which other people might not have access to. Because of this, it is difficult to recreate these experiments in order to verify these results. It is very difficult to change this, but by being open about the specifications used to run these experiments, I hope that people might find a way to run these experiments on similar hardware or otherwise be able to account for this change in some way in their own experiments.

Furthermore, the code is available publicly on GitHub, together with the data and script I wrote to execute the code such that people can rerun these experiments.

7.3 Bias for positive results

The third aspect I have taken into account is that I want to only show the positive results of the research. To combat this, I have also included the components I altered that have instead decreased the performance or did not change the performance, since this data is equally as useful for people that want to work further on this topic such that they do not have to do the same research twice.

References

[Blakely, 2021] Sarah Blakely. Zero-sum game meaning: Examples of zero-sum games - 2022, Aug 2021.

[Chaslot *et al.*, 2008] Guillaume MJ-B Chaslot, Mark HM Winands, and HJVD Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[Cropper and Dumančić, 2020] Andrew Cropper and Sebastijan Dumančić. Learning large logic programs by going beyond entailment. *arXiv preprint arXiv:2004.09855*, 2020.

[David and Kroening, 2017] Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150403, 2017.

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

[Gelly *et al.*, 2006] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. *Modification of UCT with patterns in Monte-Carlo Go*. PhD thesis, INRIA, 2006.

[Gulwani *et al.*, 2017] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[Helmuth *et al.*, 2018] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1127–1134, 2018.

[James *et al.*, 2017] Steven James, George Konidaris, and Benjamin Rosman. An analysis of monte carlo tree search. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[Kitzelmann, 2009] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*, pages 50–73. Springer, 2009.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[Koenig *et al.*, 2021] Jason R Koenig, Oded Padon, and Alex Aiken. Adaptive restarts for stochastic synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 696–709, 2021.

[Lim and Yoo, 2016] Jinsuk Lim and Shin Yoo. Field report: Applying monte carlo tree search for program synthesis. In *International Symposium on Search Based Software Engineering*, pages 304–310. Springer, 2016.

[Matulewicz, 2022] Nadia Matulewicz. Inductive program synthesis through using monte carlo tree search guided by a heuristic-based loss function, 2022.

[Polozov, 2018] Alex Polozov. Program synthesis in 2017-18, Jul 2018.

[Solar-Lezama, 2008] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[Świechowski *et al.*, 2021] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. *arXiv preprint arXiv:2103.04931*, 2021.

[Yujian and Bo, 2007] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

Which alteration	Average execution time of solved tasks (s)	Tasks solved (%)
No Alterations	1.840901868	100%
Linear, min: 0, max:1 /$\sqrt{2}$	1.91674279	99.8%
Linear, min: 0.25/$\sqrt{2}$, max: 0.75/$\sqrt{2}$	2.108145797	100%
Linear, min: 0, max: 0.1 / $\sqrt{2}$	0.477891603	100%
Step, min: 0, max:1 /$\sqrt{2}$	0.440659823	100%
Step, min: 0.25/$\sqrt{2}$, max: 0.75/$\sqrt{2}$	1.785644034	100%
Step, min: 0, max: 0.1 / $\sqrt{2}$	0.302154068	100%
Exponential, min: 0, max:1 /$\sqrt{2}$	0.218142068	100%
Exponential, min: 0.25/$\sqrt{2}$, max: 0.75/$\sqrt{2}$	0.217072345	100%
Exponential, min: 0, max: 0.1 / $\sqrt{2}$	0.216285541	100%
Updating C_p when stuck in local optima	0.199532467	100%
Randomly choosing which extension	0.553799426	100%
Using the token with the best score	7.327209766	100%
Backpropagating total loss	16.14946309	60.2%
Backpropagating normalised value	1.20469277	100%

Table 1: Results of running the experiments on robot planning

Which alteration	Average execution time of solved tasks (s)	Tasks solved (%)
No alterations	10.4785756	30.6%
Linear, min: 0, max: $0.25/\sqrt{2}$	7.45168004	51.8%
Linear, min: $0.25/\sqrt{2}$, max: $1/\sqrt{2}$	10.3391489	31.8%
Linear, min: $0.1/\sqrt{2}$, max: $0.35/\sqrt{2}$	12.2702627	40.6%
Step, min: 0, max: $0.25/\sqrt{2}$	2.70556951	26.8%
Step, min: $0.25/\sqrt{2}$, max: $1/\sqrt{2}$	8.670615	29.4%
Step, min: $0.1/\sqrt{2}$, max: $0.35/\sqrt{2}$	12.08174	31.0%
Exponential, min: 0, max: $0.25/\sqrt{2}$	0.5098	5.8%
Exponential, min: $0.25/\sqrt{2}$, max: $1/\sqrt{2}$	0.481673	7.4%
Exponential, min: $0.1/\sqrt{2}$, max: $0.35/\sqrt{2}$	0.493446	6.0%
Updating C_p when stuck in local optima	2.5209962	19.80%
Randomly choosing which extension	12.6235951	23.80%
Using the token with the best score	13.5992838	24.60%
Backpropagating total loss	0	0.0%
Backpropagating normalised value	0.86927706	1.2%

Table 2: Results of running the experiments on ASCII art

Which alteration	Average execution time of solved tasks (s)	Tasks solved (%)
No alterations	10.4785756	30.6%
Linear, min: 0, max: $0.25/\sqrt{2}$	7.45168004	51.8%
Linear, min: $0.25/\sqrt{2}$, max: $1/\sqrt{2}$	10.3391489	31.8%
Linear, min: $0.1/\sqrt{2}$, max: $0.35/\sqrt{2}$	12.2702627	40.6%
Step, min: 0, max: $0.25/\sqrt{2}$	2.70556951	26.8%
Step, min: $0.25/\sqrt{2}$, max: $1/\sqrt{2}$	8.670615	29.4%
Step, min: $0.1/\sqrt{2}$, max: $0.35/\sqrt{2}$	12.08174	31.0%
Exponential, min: 0, max: $0.25/\sqrt{2}$	0.5098	5.8%
Exponential, min: $0.25/\sqrt{2}$, max: $1/\sqrt{2}$	0.481673	7.4%
Exponential, min: $0.1/\sqrt{2}$, max: $0.35/\sqrt{2}$	0.493446	6.0%
Updating C_p when stuck in local optima	2.5209962	19.80%
Randomly choosing which extension	12.6235951	23.80%
Using the token with the best score	13.5992838	24.60%
Backpropagating total loss	0%	0.0%
Backpropagating normalised value	0.86927706	1.2%

Table 3: Results of running the experiments on string transformations