Division of Computing Science and Mathematics
Faculty of Natural Sciences
University of Stirling

Enhancing Large Language Model Performance on Complex Reasoning Tasks using Tree of Thoughts Prompting and Search Algorithms

Rodolfo Jacinto Croes

Dissertation submitted in partial fulfilment for the degree of
Master of Science in Artificial Intelligence

September 2024

Abstract

This study aims to address whether tree of thought prompting, combined with different search algorithms, enhances the performance of large language models on complex reasoning tasks. Previous work has shown that different prompting strategies tested on various datasets has led to improved performance from the language models, but tree of thought was recently introduced thus more testing with different search algorithms on different datasets is warranted.

Public datasets that address different reasoning tasks were selected along with three different search algorithms. The tree of thought framework was then adapted to accommodate for these datasets and search algorithms. Finally, the large language model was assessed and has shown improvements in performance in accuracy, suggesting that the tree of thought combined with different search algorithms can enhanced the performance of large language models on complex reasoning tasks.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this dissertation reports original work by me during my University project except for the following:

- The technology reviewed in Section 2.4 was largely taken from [18].
- The code that was used in Section 3 was taken from [18] and adapted.
- The dataset that was used in Section 3.3.1 was taken from [9].
- The code that was used for Section 3.3.2 was taken from [32] and modified.
- The code that was used for Section 4.1 was taken from [33] and modified.
- The code that was used for Section 4.3 was taken from [34] and modified.
- The dataset that was used in Section 5.1 was taken from [24].
- The dataset that was used in Section 5.2 was taken from [30].

Signature: [illegible] **Date: 30th August 2024**

Acknowledgements

I would like to start by thanking my supervisor, Professor Marc Cavazza, for his guidance and support during this dissertation. His experience with LLMs and the advice that he provided on successfully carrying out this project was valuable in terms of a workflow that is feasible but still challenging.

Secondly, I send my regards to the researchers who introduced the Tree of Thought concept to the world. This prompting technique shows a lot of potential, and I hope it continues being developed.

I am also grateful to the peers that gave me the opportunity to discuss our separate works among each other, especially my roommate Obinna Irrechukwu. Sharing the thoughts and progress of our projects was a strong motivator for myself to get as much done as possible and I hope my peers feel the same way.

To the department's staff, I thank you again for allowing me to show this work at the poster session organized. Being able to verbally communicate with staff, industry and likeminded individuals about this project has enlightened me on how valuable this project is.

Finally, I send my appreciation and love to my family and friends for their unconditional support during the work that was done for this dissertation. To the ones I have met in Stirling and to the ones that stayed in touch, thank you for everything. Especially to my mother Damia, Peace, my siblings Dione, Reu and Betto, and to my father Rudy who is no longer with us physically but is always with me every step of the way.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Context

Large Language Models (LLMs) are extremely powerful Artificial Intelligent (AI) tools that can process natural language and generate human-like text for a variety of uses. But these models tend to struggle with tasks that require complex reasoning, planning, and multi-step problem solving. Because of such limitations, LLMs are unlikely to be adopted for problems where difficult decisions-making and long-term consequence evaluations are required.

1.2 Scope and Objectives

This dissertation explores an innovative approach to potentially enhance LLM performance on complex reasoning tasks: the Tree of Thoughts (ToT) prompting framework, introduced by S. Yao *et al.* [18], combined with different search algorithms. The scope of this research encompasses:

- Implementation of the ToT framework for multiple reasoning and planning tasks
- Integration of different search algorithms with ToT
- Evaluation across multiple LLM benchmarks

The main goals and research objectives of this work are:

1. To apply and modify the ToT framework to fit a variety of reasoning tasks, with an emphasis on multi-step reasoning tasks, commonsense reasoning tasks, and mathematical word problems.
2. To implement and assess various search algorithms' performance within the ToT framework, focusing on their effects on the runtime, accuracy, and expenses.
3. To determine whether this approach enhances the performance of LLMs on reasoning tasks by conducting a thorough analysis of these modified ToT implementations across multiple LLM benchmarks.

1.3 Significance and Impact

As AI become more integrated into our everyday lives, the ability of LLMs to effectively tackle complex multi-step problems could have major implications on how people could potentially approach various decision-making tasks across different industries and fields of study. Enhanced planning and reasoning abilities in LLMs could also address challenges in various domains, such as optimising delivery routes for logistics firms or aiding in financial planning through scenario analysis.

1.4 Technologies and Tools

To carry out this dissertation, it was necessary to learn and implement several technologies, strategies and tools:

- Google Colab for Python development
- OpenAI’s API calling method
- ToT Framework

- Tree Search Algorithms (BFS, DFS, A*)
- Benchmark Datasets (GSM8K, GSM-HARD, StoryCloze)
- Performance Metrics for LLM evaluation

These tools and technologies form the foundation of our research methodology and experimental approach. The installation instructions for this work can be found in Appendix 3 – User Guide.

2 State-of-The-Art

2.1 Large Language Models (LLMs)

Large Language Models (LLMs), as described by W. X. Zhao *et al.* [1], is large-sized pre-trained language models that use the context of previously generated text, and deep learning methods, to predict the next word in the generated text sequence. Early language models aimed to generated text data using statistical learning methods and neural networks. Recently, LLMs are commonly based on the transformer architecture and employ the self-attention mechanism introduced by A. Vaswani *et al.*[2]. These two features are used for efficient training on large datasets and allow for better capture of dependencies in text which result in improved performance compared to previous language models.

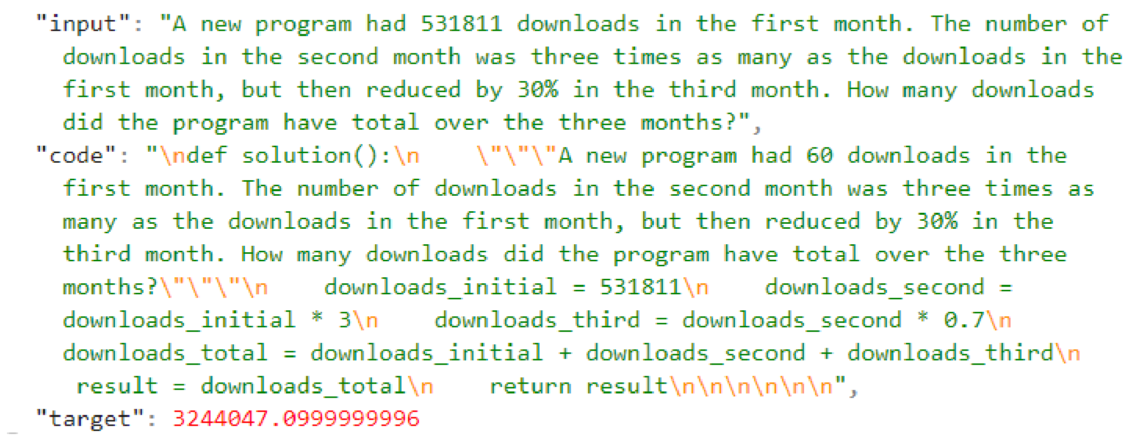
Some applications for LLMs include text generation, code generation, translation and question-answering. The performance of the LLMs on said applications is shown by T. Brown *et al.* [3] via experiments that used different datasets to evaluate different aspects of LLMs. These datasets are commonly known as LLM benchmarks.

2.2 Benchmarking LLM Performance

LLM benchmarks are standardized tests designed to evaluate the performance of LLMs on specific tasks or capabilities. These benchmarks range from general language understanding (GLUE)[5] to specialized areas like code generation (HumanEval)[6] and multimodal tasks (MME benchmark)[7]. For evaluating reasoning, most tests in the past did not run tests on LLMs. Instead, researchers focussed on testing different algorithms on their question-answering abilities that required reasoning over scientific facts. An example of this is the AI2 Reasoning Challenge (ARC)[11] which in its report did not test the dataset on LLMs.

One of the earliest benchmarks used on LLMs for evaluating reasoning abilities is GSM8K [9]. This dataset consists of 8,500 high-quality math word problems split into 7,500 training set and 1,000 test set of problems. These problems require multi-step reasoning and basic arithmetic operations, challenging LLMs to demonstrate both language understanding and mathematical problem-solving skills.

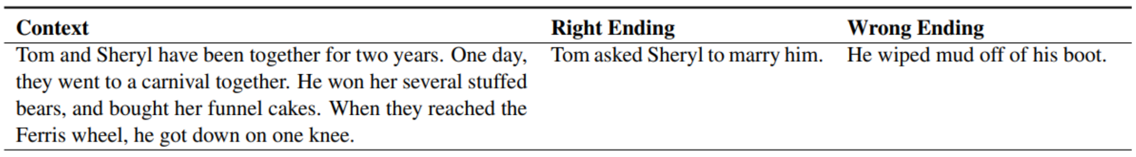
GSM8K is still used today to test LLMs, but there are some limitations with this dataset. One limitation identified by Madaan and Yazdanbakhsh [35] is the numbers in the problems are mostly low integers, which recent models excel at solving. To combat this, researchers have created the GSM-HARD[24] benchmark which introduces larger numbers and non-integers to the existing GSM8K questions to test how the model performs with mathematical reasoning. An example of a question from this benchmark can be found in Figure 1.



```
"input": "A new program had 531811 downloads in the first month. The number of
  downloads in the second month was three times as many as the downloads in the
  first month, but then reduced by 30% in the third month. How many downloads
  did the program have total over the three months?",
"code": "\ndef solution():\n    \"\"\"A new program had 60 downloads in the
  first month. The number of downloads in the second month was three times as
  many as the downloads in the first month, but then reduced by 30% in the
  third month. How many downloads did the program have total over the three
  months?\"\"\"\n    downloads_initial = 531811\n    downloads_second =
  downloads_initial * 3\n    downloads_third = downloads_second * 0.7\n
  downloads_total = downloads_initial + downloads_second + downloads_third\n
   result = downloads_total\n    return result\n\n\n\n\n\n",
"target": 3244047.0999999996
```

Figure 1. Example of GSM-HARD Benchmark [24].

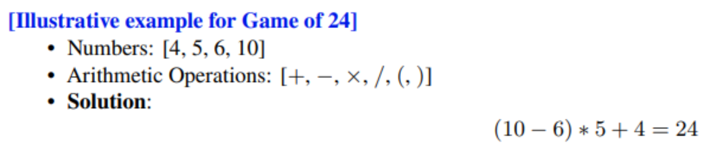
In the area of commonsense reasoning benchmarks, the StoryCloze[30] and HellaSwag[8] datasets assesses a language model's ability to understand and reason about various situations using commonsense knowledge. Both benchmarks evaluate how the LLM finishes a sentence by choosing between multiple options. Where HellaSwag looks between four options on how to finish a body of text, StoryCloze presents a four-sentence story followed by two alternative endings. Both benchmarks challenge the model’s ability to select the most plausible conclusion. An updated dataset of StoryCloze was introduced in 2018 to mitigate some limitations from the original dataset [31]. An example from the StoryCloze dataset can be seen in Figure 2, where the model must choose between two possible endings to complete a short narrative about a character's actions and motivations.



Context	Right Ending	Wrong Ending
Tom and Sheryl have been together for two years. One day, they went to a carnival together. He won her several stuffed bears, and bought her funnel cakes. When they reached the Ferris wheel, he got down on one knee.	Tom asked Sheryl to marry him.	He wiped mud off of his boot.

Figure 2. Example of StoryCloze Benchmark from the original report [30].

Outside of sentence completion and mathematic word problems, LLMs can also be evaluated on their ability to explore different solutions to a partial problem with the goal of finding a solution to a general problem. Used in the original Tree of Thought report, Game of 24 benchmark [18] is a collection of integers combinations that can generate the number 24 by using basic mathematical operations. A solution example from this benchmark is seen in Figure 3 as an equation.



[Illustrative example for Game of 24]
- Numbers: [4, 5, 6, 10]
- Arithmetic Operations: $[+, -, \times, /, (,)]$
- **Solution**:

$$(10-6)*5+4=24$$

Figure 3. Example of Game of 24 Benchmark from [10].

Recently, a benchmark that evaluates the planning ability of LLMs include PlanBench [12]. This dataset includes multiple tasks, such as stacking blocks on a table or moving packaged between different locations, that require generating, evaluating, and executing plans. This benchmark also tests the models' ability to break down complex problems into manageable steps.

From the discussed benchmarks, the GSM8K[9], GSM-HARD[24] and StoryCloze[30] benchmarks will be implemented to evaluate the mathematical reasoning and commonsense reasoning capabilities of the LLMs. These were chosen to show different reasoning methods that must be considered when evaluating LLMs, while being feasible to implement given the amount of time allocated for this project.

While these benchmarks have been instrumental in assessing and improving LLM performance, they also highlight areas where models still fall short of human-level reasoning. This gap between benchmark performance and real-world application has led researchers to explore various techniques to enhance LLM capabilities. An example of a technique developed into its own discipline, this being prompt engineering.

2.3 Prompt Engineering Techniques

Prompt engineering is an ever-evolving skill that introduces systematic approaches on how to design prompts with the goal of improving and optimizing said prompts for better model performance [13]. The simplest form of prompt design is input-output or zero-shot prompting which involves querying an LLM with minimal instructions, relying solely on the model's pre-trained knowledge to solve a given task.

Based on zero-shot prompting, few-shot prompting provides the model with a small number of examples to demonstrate the task [3]. While this technique has shown to lead to improved performance from the LLMs, it does not show how LLMs gets to the solution of the task. One technique that does show a step-by-step breakdown of a problem and how it gets solved is Chain-of-thought (CoT). Introduced by J. Wei *et al.* [14] , CoT encourages LLMs to break down complex problems into intermediate steps by giving the model few-shot examples. A modified CoT approach by T. Kojima *et al.* [15], zero-shot CoT is done by prompting the model with the sentence "think step by step" which can be as effective as few-shot CoT.

While CoT has shown improvements in some reasoning tasks, there are still some limitations with the LLMs using CoT. Some of these limitations include complex proofs and multi-step planning tasks. Researchers have also identified an issue with "faithfulness" of the model's predictions where the chain is intervened by a change, leading to unfaithful results [16]. Furthermore, LLMs with CoT usually take a greedy selection approach when decoding their solution thus limiting the search space for other solutions. A workaround for this, introduced by X. Wang *et al.* [17], is to extend CoT with self-consistency which generates multiple reasoning paths and selects the most consistent answer.

With the demand for better LLMs increasing, the development of prompting techniques that advance the capabilities of LLMs with reasoning tasks has become a focal point of research. One method that is built upon the discussed prompting techniques is the Tree of Thoughts (ToT) prompting technique which can be seen in Figure 4.

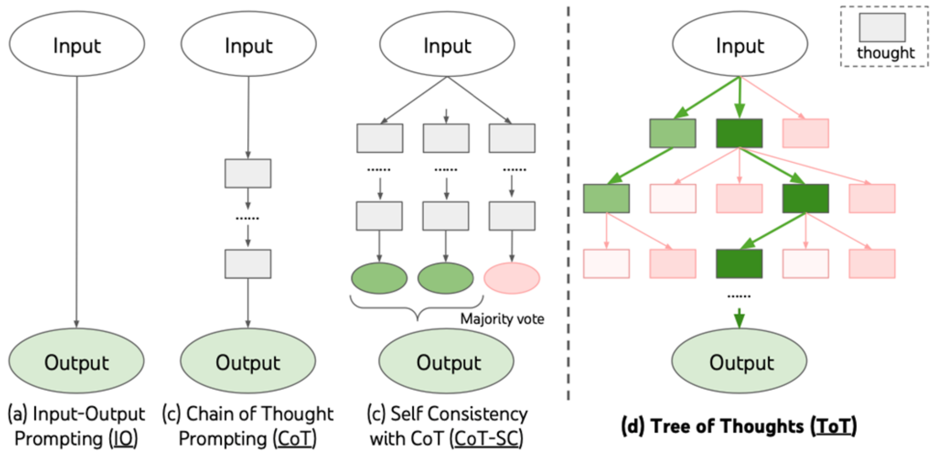

Input
Input
Input
Input
thought
Majority vote
Output
Output
Output
Output
(a) Input-Output Prompting (IO)
(c) Chain of Thought Prompting (CoT)
(c) Self Consistency with CoT (CoT-SC)
(d) Tree of Thoughts (ToT)

Figure 4. Visual Representation of various prompting techniques (IO, CoT, Cot-SC and ToT) From [18].

2.4 Tree of Thoughts (ToT) Framework

Unlike previous prompting techniques, the ToT framework offers a more comprehensive approach to enhancing LLM reasoning capabilities. Introduced by S. Yao *et al.* [18], ToT expands on CoT prompting by reimagining the reasoning process as a tree-like structure rather than a linear chain. This allows LLMs to explore multiple reasoning paths simultaneously, mirroring human problem-solving strategies more closely. The framework is made up from four key components: thought decomposition, thought generator, state evaluator, and search algorithm.

The thought decomposition aims to break the problem into intermediate steps. These steps or “thoughts” should be small enough to allow for promising and diverse solutions to be generated, yet large enough that the model can evaluate various solutions generated based on the potential of the solution leading towards solving the problem.

Next, the thought generator creates multiple solutions at each reasoning step of the tree. These solutions are generated by either a sampling via CoT prompt if the thought space of the solution is rich (e.g. paragraphs), or by a “propose prompt” where the thought space is more constrained (e.g. word or number). The sampling generation allows for more diversity while the proposal generation avoids duplication in generated solutions.

Following this, the state evaluator assesses each generated thought by how likely they are to lead towards a possible overall solution to the problem. This is done by using the LLM to reason about possible states that lead towards said overall solution. This is a more flexible heuristic compared to when it is programmed in, and it is more sample-efficient than when the heuristic is based on neural networks learning method. The strategies that are used in quantifying the evaluations are either by valuing each state independently or voting across states by comparing partial solutions with other partial solutions at the same tree level. Both strategies allow the LLM to be prompted multiple times to trade time and cost for more faithful heuristics.

Finally, within this framework, one can implement different search algorithms depending on the tree structure. These search algorithms determine the selection method of which states to continue exploring. In the original work, the two implementations of search algorithms are Breadth-fist search (BFS)[32] and Depth-first search (DFS)[33] visualized in Figure 5. BFS is used when the depth of the tree structure is limited, and the thought steps can be filtered to create subsets of states with the highest potential to produce an overall solution. DFS explores the most promising state first until a final solution is reached, or the evaluator deems the problem as impossible to solve in the current state. In this case, the subtree explored is thrown out. Both cases of DFS use backtracking to get back to the parent state to continue exploration.

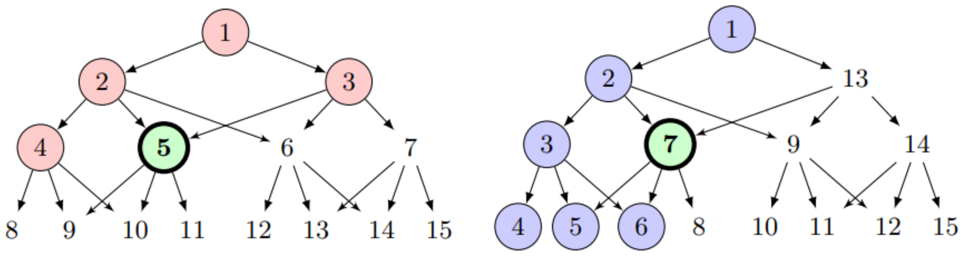
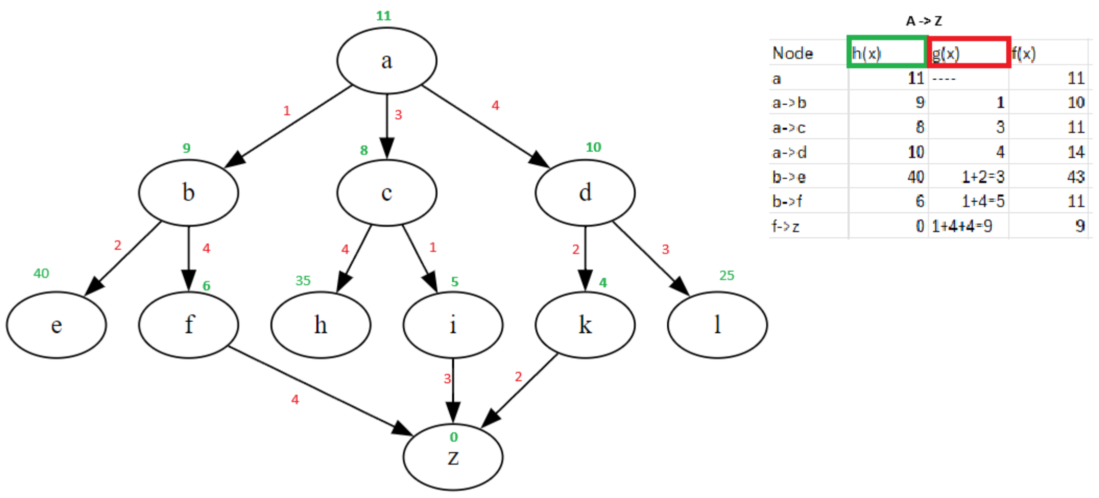


Figure 5. Visual Representation of BFS (left) against DFS (right) from [20].

While the original ToT framework implements BFS and DFS as its primary search algorithms, the flexibility of the ToT approach allows for the integration of more sophisticated search strategies. The choice of search algorithm can potentially impact the efficiency and effectiveness of the reasoning process, particularly for complex problems with large solution spaces.

2.5 Search Algorithms for Tree Structures

Building upon the foundation laid by BFS and DFS in the Tree of Thoughts (ToT) framework, more advanced search algorithms can potentially enhance the efficiency and effectiveness of LLM reasoning. One algorithm that could potentially lead to enhanced performance is the A* (A-star) search[19].



A -> Z

Node	h(x)	g(x)	f(x)
a	11	----	11
a->b	9	1	10
a->c	8	3	11
a->d	10	4	14
b->e	40	1+2=3	43
b->f	6	1+4=5	11
f->z	0	1+4+4=9	9

Figure 6. Example of A* tree search with values.

A* is an informed best-first search algorithm that combines the benefits of both BFS and DFS by using the cost-heuristic function *f(x)* to estimate the cost from the current node to the goal. From this function, the order of nodes can prioritize by lowest value of *f(x)*. The function is usually determined by taking the sum of the cost function *g(x)* and the heuristic estimate of the goal *h(x).* An example of the A* search algorithm can be seen in Figure 6. In terms of ToT, *g(x)* relates to how many thoughts are generated in the past via the LLM. The *h(x)* function in ToT relates to how close the current state is to a final solution to the tasks. Both functions effect the cost and runtime of the ToT depending on the depth and size of the tree. Even so, the A* could lead to more efficient traversal of the solution space, particularly in problems where certain thought paths are clearly more promising than others.

While it would be ideal to implement more than one complex search algorithms in the experiments, given the time constraints of this project, the decision was made to only implement the A* search algorithm because it allows for exploration with various implementations of the heuristic and cost functions. One other search algorithm that was considered was the Monte Carlo Tree Search (MCTS) [21], but the cost of running simulations for MCTS is higher than the cost of running A* search.

2.6 Recent Developments in LLM Reasoning Abilities

Ongoing research in LLMs include improving reasoning capabilities, enhancing factual consistency, and developing more efficient training methods to reduce cost and computational requirements [4]. With the goal of enhancing reasoning abilities from LLMs using prompting techniques, researchers have proposed to build upon the existing ToT framework in different ways. One of these is Cross-lingual Tree of Thoughts (Cross-ToT) introduced by L. Ranaldi *et al.* [22], this method reasons across languages by stating to do so in the prompt, leading to different reasoning paths and a more diverse search space.

While the previous prompting technique is inspired by ToT, another approach to enhance reasoning abilities is to allow the model to choose its own prompting strategy when solving a problem. Meta-Reasoning Prompting (MRP)[23] allows the LLM to initially identify the most appropriate reasoning plan based on various existing prompting techniques, and from there apply the chosen method to complete a task. This approach addresses the task-specific limitation of ToT by using a meta-reasoning prompt to select a strategy from the possible methods implemented, based on the inputted task.

Another approach to handling reasoning tasks using LLMs is by altering how the reasoning is outputted. Instead of natural language, Program-Aided Language models (PAL)[24] propose to output the generated reasonings as programs that are offloaded to an interpreter that can run the program. This allows the LLM to only generate strategies for complex tasks in the form of code instead of generating solutions which saves computational cost.

Outside of text and code, the LLMs reasoning capabilities have also been implemented in visualization tasks such as segmentation. An example of this is with LISA, a large language instructed segmentation assistant, that uses a multimodal LLM to produce segmentation masks on images given an ambiguous set of instructions [25]. This is done by jointly considering text and an image as input and using the LLM to reason about how and where to produce a segmentation mask on the input image. This example demonstrates that with further enhancement in reasoning capabilities of LLMs, it is possible that these prompting techniques are not limited to only text and code, giving further reason to continue the exploration of enhancing reasoning capabilities in LLMs.

3 Implementation of Tree of Thoughts

3.1 Background on Model Selection and Usage

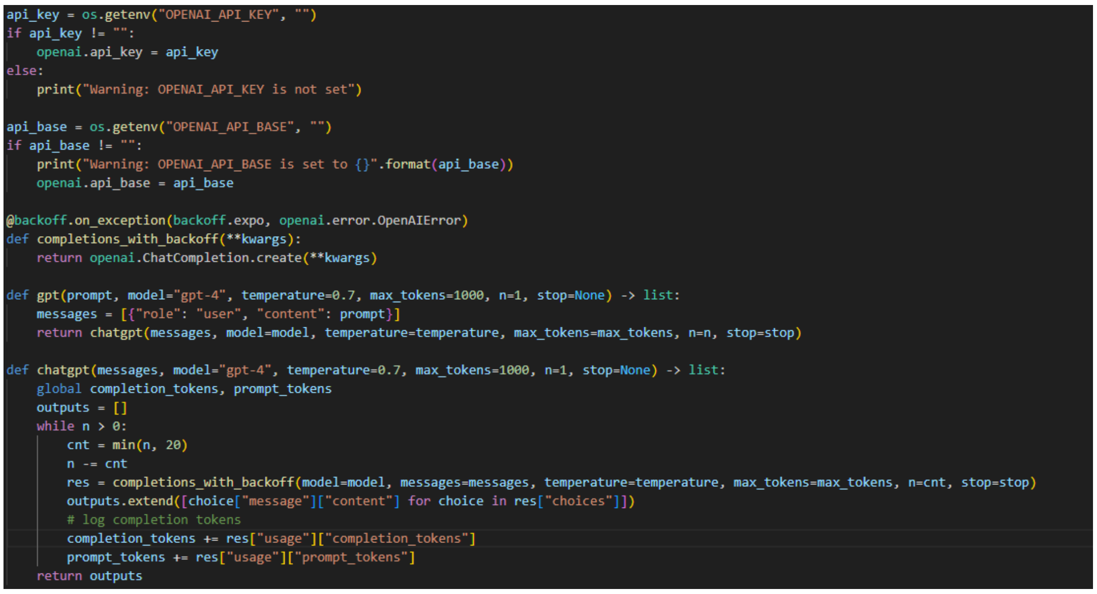
S. Yao *et al.* [18] originally experiments utilized GPT-4, which at the time represented the state-of-the-art in LLMs [26]. GPT-4 demonstrated impressive capabilities in handling complex reasoning tasks, making it an ideal choice for exploring the ToT framework. The temperature used in the original experiment is 0.7 which will be retained during experimentations.

However, since then there have been significant developments in the field of language models. Notably, OpenAI released GPT-4 Omni (GPT-4o), a more cost-efficient variant of GPT-4 that maintains strong performance while reducing the operational costs [27]. This development presented an opportunity to replicate and extend the ToT experiments on GPT-4 with a more economically viable model.

It is worth noting that for the purposes of this research project, a new OpenAI API key was created. This decision was made to accurately track and isolate the costs associated specifically with this project, ensuring transparency and precision in our resource utilization analysis.

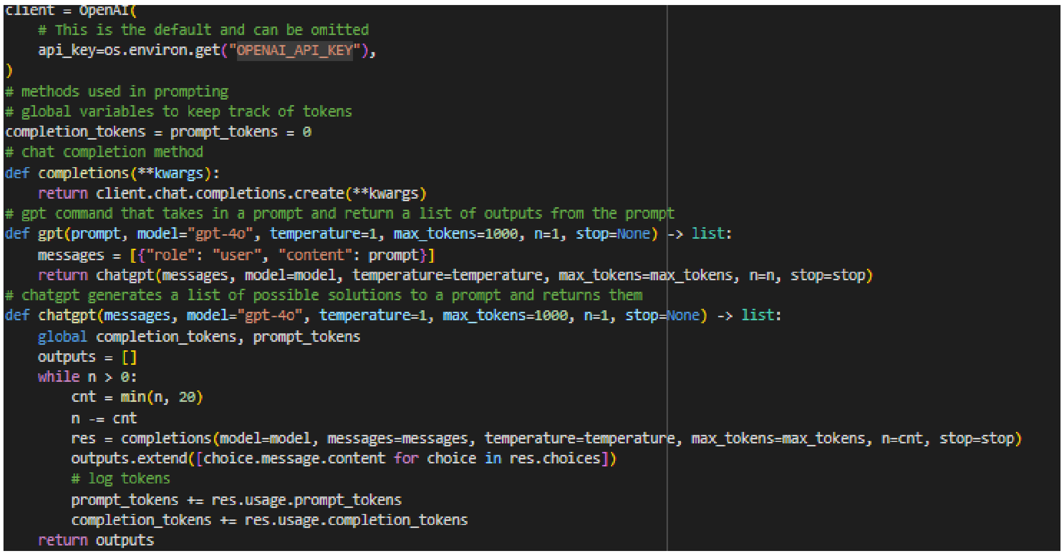
3.2 Adapting ToT for updated API calls

To implement the ToT framework with GPT-4o, several modifications were made to the original code. The primary change involved transitioning from the OpenAI Completion API calls to the Client Chat Completion API calls. The reason for this is because with GPT-4o, the old API calling methods do not work given that OpenAI has shifted from User API keys to Project API keys. Luckily, this does not affect the method of calling older models, such as GPT-3.5, because the new API calling method works with them. This shift required a change to the completions function from the original code. Figure 7 illustrates the original API call, while Figure 8 demonstrates the updated API call for GPT-4o:



```python
api_key = os.getenv("OPENAI_API_KEY", "")
if api_key != "":
    openai.api_key = api_key
else:
    print("Warning: OPENAI_API_KEY is not set")

api_base = os.getenv("OPENAI_API_BASE", "")
if api_base != "":
    print("Warning: OPENAI_API_BASE is set to {}".format(api_base))
    openai.api_base = api_base

@backoff.on_exception(backoff.expo, openai.error.OpenAIError)
def completions_with_backoff(**kwargs):
    return openai.ChatCompletion.create(**kwargs)

def gpt(prompt, model="gpt-4", temperature=0.7, max_tokens=1000, n=1, stop=None) -> list:
    messages = [{"role": "user", "content": prompt}]
    return chatgpt(messages, model=model, temperature=temperature, max_tokens=max_tokens, n=n, stop=stop)

def chatgpt(messages, model="gpt-4", temperature=0.7, max_tokens=1000, n=1, stop=None) -> list:
    global completion_tokens, prompt_tokens
    outputs = []
    while n > 0:
        cnt = min(n, 20)
        n -= cnt
        res = completions_with_backoff(model=model, messages=messages, temperature=temperature, max_tokens=max_tokens, n=cnt, stop=stop)
        outputs.extend([choice["message"]["content"] for choice in res["choices"]])
        # log completion tokens
        completion_tokens += res["usage"]["completion_tokens"]
        prompt_tokens += res["usage"]["prompt_tokens"]
    return outputs
```

Figure 7. **API Call Functions from original paper [18].**



```python
client = OpenAI(
    # This is the default and can be omitted
    api_key=os.environ.get("OPENAI_API_KEY"),
)
# methods used in prompting
# global variables to keep track of tokens
completion_tokens = prompt_tokens = 0
# chat completion method
def completions(**kwargs):
    return client.chat.completions.create(**kwargs)
# gpt command that takes in a prompt and return a list of outputs from the prompt
def gpt(prompt, model="gpt-4o", temperature=1, max_tokens=1000, n=1, stop=None) -> list:
    messages = [{"role": "user", "content": prompt}]
    return chatgpt(messages, model=model, temperature=temperature, max_tokens=max_tokens, n=n, stop=stop)
# chatgpt generates a list of possible solutions to a prompt and returns them
def chatgpt(messages, model="gpt-4o", temperature=1, max_tokens=1000, n=1, stop=None) -> list:
    global completion_tokens, prompt_tokens
    outputs = []
    while n > 0:
        cnt = min(n, 20)
        n -= cnt
        res = completions(model=model, messages=messages, temperature=temperature, max_tokens=max_tokens, n=cnt, stop=stop)
        outputs.extend([choice.message.content for choice in res.choices])
        # log tokens
        prompt_tokens += res.usage.prompt_tokens
        completion_tokens += res.usage.completion_tokens
    return outputs
```

Figure 8. Updated API Call Functions.

3.3 Initial Experiment Setup

3.3.1 Integrating GSM8K Benchmark

To evaluate the initial implementation of ToT, the GSM8K Benchmark is implemented by loading in the test data containing 1319 questions and answers [9]. Then, a tasks selection mechanism based on either random selection or ranked selection was implemented. The ranked selection scored each task by difficulty based on question length, number of numerical values and number of mathematical operations in the solution. This allows for more controlled evaluations when testing different prompting techniques.

3.3.2 ToT + BFS Solver Function

For the implementation of the GSM8K Benchmark, some modifications were made to ToT Solver developed by S. Yao *et al.* [18]. The method that was employed to motivate these modifications was by remaking the code with the purpose of better understanding the structure of the framework. This was done by first implementing a BFS structure in Python, acquired from S. Bhadaniya[32], and afterwards adding the workflow of the algorithm in the structure.
Initially, instead of considering the tree depth as a limit on when to stop exploration, a solution detection system was employed that would indicate to the solver when to stop. This was done by searching in the output for the answer format of GSM8K. However, this was not the ideal approach because it had the chance of cutting the exploration off too early. So, this was reverted to the maximum tree depth parameter which would stop exploration once a solution was generated, given that the solution layer is known, and evaluated as the end of said task.

After this, the thought generation was chosen to be handled by the proposing method instead of the sampling method because of the constrained search space. As mentioned before, the state evaluator chosen is the voting method and from this the extractions were done to assign values to the generated solutions. Once this is completed, the selection of states to continue exploring is possible by two methods:

- Greedy: sorts the list of evaluations from largest to smallest and selects the top states to continue exploring. This ensures that only the best initial solutions are selected throughout exploration. One limitation of this method is the lack of randomness which is crucial for exploring diverse solutions and escaping local optima solutions.
- Sample: Using the normalized probabilities and a stochastic selection method to select the states to continue exploring. This ensures an exploration and exploitation balance in the search space.

Once the states have been selected, they are saved in a data structure and afterwards the current outputs are set to the new selected outputs for further exploration. The function will then terminate if the maximum depth of the tree is reached, and the number of steps and selected solution will be logged as the final solution.

1. Determine the exploration order depending on the selection method (greedy, sample)
2. Explore the solutions in the determined order while taking account for impossible states, deepest state explored so far, and if the next state can be explored

Once the maximum steps or maximum depth has been reached, the resulting output is the deepest explored state. If there is a tie in level, then the most recent explored deepest state is returned. From how the GSM8K benchmark is setup, the maximum depth is known to be two: One for strategy selection and one for answer generation.

3.3.3 Prompting for GSM8K

The format prompt that defines how the output is structured, the zero-shot IO prompt and zero-shot ToT prompt were all acquired from the implementation of ToT on GSM8K done by S. Yao *et al.* [18]. Following this, the strategy format and answer format prompts were designed with the intention of allowing the ToT to have more than one layer of depth for a more robust search space. For the evaluation prompt of this implementation, the voting prompt was taken from S. Yao *et al.* [18]. Seen in Figure 9, this prompt would allow for each evaluation to only select one state as the best and the state with the most selections will be prioritized for exploration.

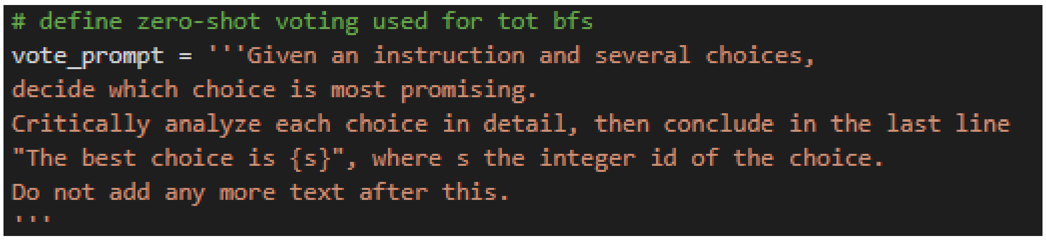
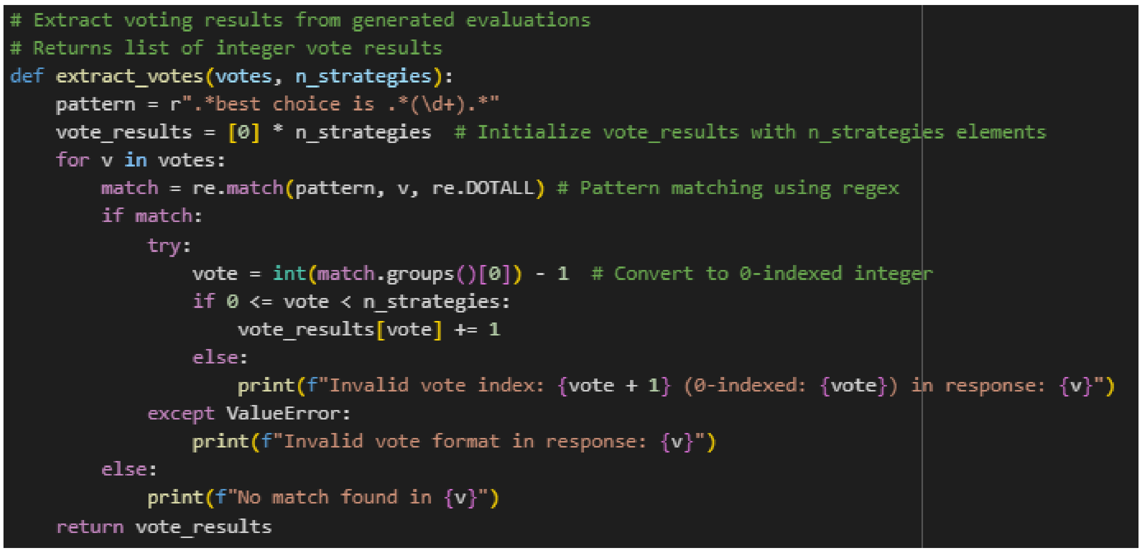

```
# define zero-shot voting used for tot bfs
vote_prompt = '''Given an instruction and several choices,
decide which choice is most promising.
Critically analyze each choice in detail, then conclude in the last line
"The best choice is {s}", where s the integer id of the choice.
Do not add any more text after this.
'''
```

Figure 9. Vote prompt for ToT on GSM8K.

Along with the evaluation prompt, there is also an extraction function implemented to translate the evaluations to numerical values that are less complex to work with during the state selection of the ToT framework. The voting extraction function in Figure 10 is straightforward given that for each evaluation, only one generated thought is chosen as the best by voting. The detection of a vote outcome is done by implementing a regular expression that extracts the selected vote from the format that is defined in the vote prompt.



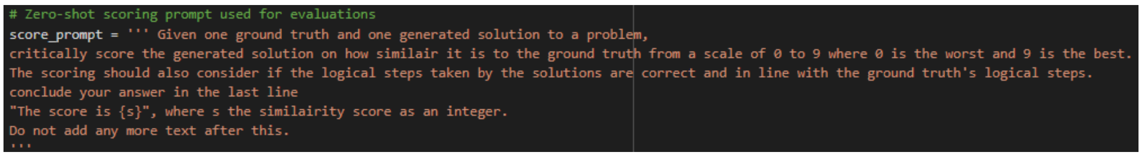
```python
# Extract voting results from generated evaluations
# Returns list of integer vote results
def extract_votes(votes, n_strategies):
    pattern = r".*best choice is .*(\d+).*"
    vote_results = [0] * n_strategies  # Initialize vote_results with n_strategies elements
    for v in votes:
        match = re.match(pattern, v, re.DOTALL) # Pattern matching using regex
        if match:
            try:
                vote = int(match.groups()[0]) - 1  # Convert to 0-indexed integer
                if 0 <= vote < n_strategies:
                    vote_results[vote] += 1
                else:
                    print(f"Invalid vote index: {vote + 1} (0-indexed: {vote}) in response: {v}")
            except ValueError:
                print(f"Invalid vote format in response: {v}")
        else:
            print(f"No match found in {v}")
    return vote_results
```

Figure 10. Vote extraction function.

3.3.4 Performance Metrics

Assessing the performance of the prompt implementations is done by three key metrics:

1. Runtime: Measured using Python's time library to capture the execution time of each solver.
2. GPT Usage Cost: Calculated based on OpenAI's pricing for GPT-4-o, considering both prompt and completion tokens [28].
3. Self-Evaluation Accuracy: Determined through a self-evaluation method, inspired by LLM-based evaluation [29], that compares the generated solution against the provided solution in the GSM8K dataset using a score prompt seen in Figure 11. The calls to the LLM during the evaluation are not considered under the GPT Usage Cost, instead it is considered as an independent cost (Analysis cost) because it is not the only possible metric for accuracy. The self-evaluation accuracy was used when the ground truth from the data is an explanation of the solving process, along with a result of said process.



```python
# Zero-shot scoring prompt used for evaluations
score_prompt = ''' Given one ground truth and one generated solution to a problem,
critically score the generated solution on how similair it is to the ground truth from a scale of 0 to 9 where 0 is the worst and 9 is the best.
The scoring should also consider if the logical steps taken by the solutions are correct and in line with the ground truth's logical steps.
conclude your answer in the last line
"The score is {s}", where s the similairity score as an integer.
Do not add any more text after this.
'''
```

Figure 11. Scoring prompt used for determining accuracy via self-evaluation.

3.4 Initial Evaluation of GPT-4o

When assessing the effectiveness of ToT using GPT-4o after conducting the experiments, a comparison of the results from the ToT solver using BFS and Greedy selection to the IO solver results is employed because these are the first two successfully implemented prompting techniques. The ToT used 5 generations, 5 evaluations and 20% of the evaluated states were kept for further exploration. The parameter of when to stop the ToT in this run was implemented with a solution detection system that checks when a solution contains the pattern of the answer format. The aim for this data is to check whether implementing ToT shows better performance when solving multi-step mathematical word problems compared to the less costly IO prompting. Table 1 summarizes the performance from a set of 10 randomly chosen tasks, and Table 2 summarizes the performance from a set of the most challenging set of tasked from GSM8K using the ranked selection. In Appendix 1 and Appendix 2 there is example outputs that were logged from running the solvers.

Table 1. Results from prompting techniques on random GSM8K tasks using GPT-4o

Random 10 Tasks	**Prompting Techniques:**	
	IO	ToT + Greedy BFS
Runtime *(s)*	49	504
Cost *($)*	0.02	0.72
Self-Accuracy *(%)*	93	100
Self-Analysis Cost ($)	0.08	0.10

Table 2. Results from prompting techniques on challenging GSM8K tasks using GPT-4o

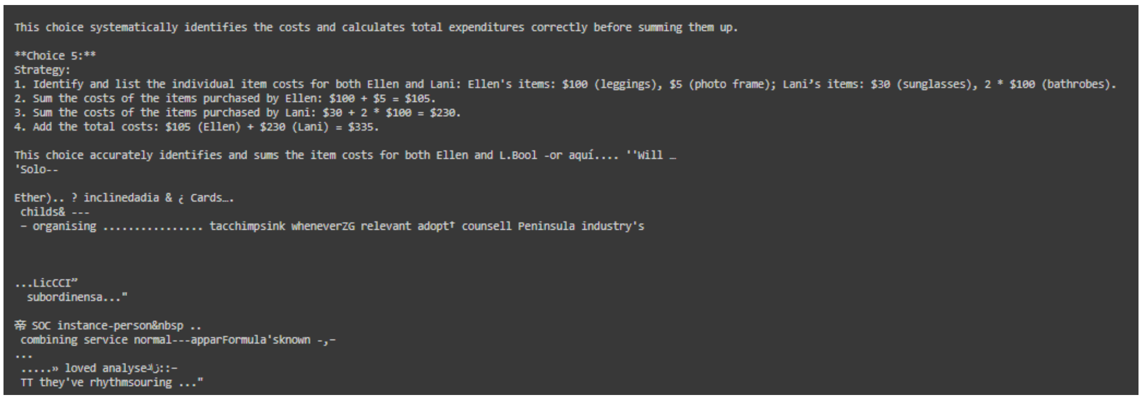
Ranked 20 Tasks	**Prompting Techniques:**	
	IO	ToT + Greedy BFS
Runtime *(s)*	91	734
Cost *($)*	0.08	2.07
Self-Accuracy *(%)*	96	96
Self-Analysis Cost ($)	0.27	0.30

At first glance, the results in terms of accuracy on GPT-4o suggest that the ToT using BFS, and greedy selection is marginally better than IO prompting. However, the cost and runtime are significantly more when using the ToT here. Because of this, the cost to performance ratio favors the IO prompting technique on GPT-4o.

3.5 Challenges and Limitations of Initial Experiment

Despite the promising results, several challenges were identified during the implementation and testing of the ToT solver:

1. Hallucinations and Incompleteness: The voting system implemented for state evaluation occasionally produced incomplete or hallucinated outputs, as demonstrated in Figure 12.
2. Cost-Effectiveness: The significant difference in runtime and cost between the IO and ToT solvers raises questions about the practical applicability of ToT for simpler reasoning tasks. For the GSM8K problems tested, the current cost-to-performance ratio does not justify using ToT with greedy BFS over IO with GPT-4-o.
3. Only one metric: The metric implemented to evaluate the model's performance is not enough to draw general conclusions about the performance of the LLMs reasoning abilities. The self-evaluation accuracy can also be inconsistent with how it scores a solution depending on what the LLM deems is a good solution compared to a bad solution.
4. Only one search algorithm implemented: The initial tests were only ran on one ToT search algorithm combination. This limits the assumptions that can be made of the effectiveness of ToT.
5. Only one benchmark: The initial test was only performed on one dataset thus showing only one type of reasoning skill. Also, the GSM8K dataset could have been used to train the newer model thus making this test relatively easier for GPT-4o.



```
This choice systematically identifies the costs and calculates total expenditures correctly before summing them up.

**Choice 5:**
Strategy:
1. Identify and list the individual item costs for both Ellen and Lani: Ellen's items: $100 (leggings), $5 (photo frame); Lani’s items: $30 (sunglasses), 2 * $100 (bathrobes).
2. Sum the costs of the items purchased by Ellen: $100 + $5 = $105.
3. Sum the costs of the items purchased by Lani: $30 + 2 * $100 = $230.
4. Add the total costs: $105 (Ellen) + $230 (Lani) = $335.

This choice accurately identifies and sums the item costs for both Ellen and L.Bool -or aquí.... ''Will …
'Solo--

Ether).. ? inclinedadia & ¿ Cards….
 childs& ---
 - organising ............... tacchimpsink wheneverZG relevant adopt† counsell Peninsula industry's



...LicCCI”
  subordinensa..."

帝 SOC instance-person&nbsp ..
 combining service normal---apparFormula'sknown -,-
...
 .....» loved analyseڑ::-
 TT they've rhythmsouring ..."
```

Figure 12. **Hallucinations and Incompleteness present in ToT solver output**

3.6 Modifications to Mitigate Initial Experiment Limitations

Mitigating the previously mentioned limitations consisted of checking potential errors made in the first implementation of ToT. From further analysis, the following changes were made to the original code:

1. Token size: The problem discovered that caused the incompleteness was the size of the token window allotted for outputs. Increasing the token window (from 1024 to 4096) lead to this issue being solved for the future tests.

2. Cost-effective model selection: With how well GPT-4o performed on the initial experiment using IO prompting compared to ToT prompting, the cost to performance ratio on GPT-4o does not seem financially viable for this project. For this reason, the decision to use a more cost-effective offering from OpenAI was made. The GPT-3.5-turbo model was selected with hopes of showing that ToT can improve the reasoning performance of an older model at a lower cost.
3. Selecting new metric: While the LLM self-evaluation excels at matching reasoning steps between the optimal solution and the generated solution, it is not considered as a standard and simple metric. The metric that will be added to the evaluation of the LLM performance is an accuracy score which compares the respective values of the outputs of the dataset and the generated solution. This is a more standard and simple accuracy score compared to the self-evaluated accuracy score from before.

This section has shown that implementing ToT on a frequently used LLM benchmarks with a newer model is possible. However, the results that were gathered are not convincing enough to state that ToT improves the performance of LLMs on reasoning tasks. In the following sections, there are descriptions on how various search algorithms are implemented in the ToT framework.

4 More Search Algorithms

4.1 Implementing DFS into ToT

In the ToT, the DFS has already been implemented by S. Yao *et al.* [18] for the Crossword puzzle task. However, this implementation is not ideal for the GSM8K benchmark because the algorithm continues exploring without considering when it has already visited the deepest possible node in the solution. This is where the introduction of a maximum steps parameter is valuable. This parameter allows the DFS to continue searching until the maximum allocated steps have been exhausted, or until a solution at the deepest state is returned.

The implementation of the DFS search algorithm is based on the Python implementation by S. Bhadaniya[33]. To adapt the code from the BFS to the DFS algorithm, the functionality of the ToT method was maintained within a recursive function, with the modified order of operations in the recursive function being:

1. Early return if the maximum depth is reached, if the maximum steps are exhausted or if a unanimous solution is found.
2. Update the prompt to reflect what state is currently being explored.
3. Thought generation previously implemented.
4. State Evaluation using values instead of votes.
5. Record the current state at each iteration to show the entire search process afterwards.
6. Selection process previously implemented.
7. Explore the next thought in the order of exploration after skipping the impossible states, updating the deepest state when one is found, and checking if the next thought can be explored.

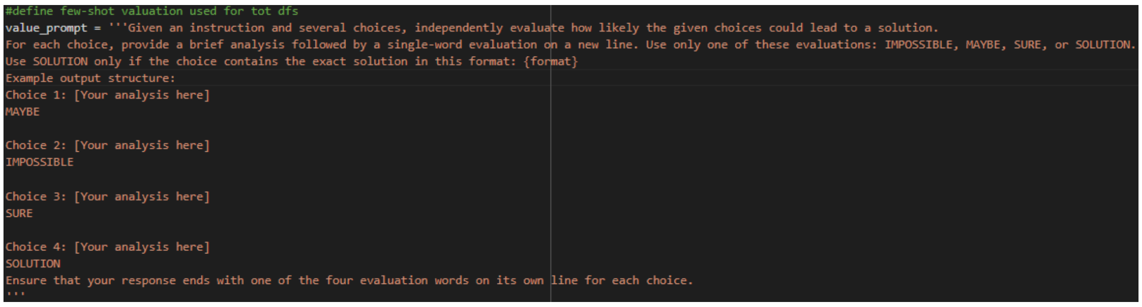
4.2 Valuation System for State Evaluation

In the original ToT framework by S. Yao *et al.* [18], there is another option outside of voting for how the state evaluator determines a best solution. The valuation system uses a prompt to evaluate each thought individually and produces a classification or a number that represents the quality of the solution. The change from the voting system to the value for DFS was warranted because the range given to each LLM to evaluate a solution on allows a unanimous solution to be detected earlier. This is beneficial for terminating the loop early if all the state evaluators determine that a solution is perfect.

The prompt that was developed for this implementation uses a classification valuation system where the LLM chooses a rank for each thought based on how likely the state can lead to a solution to the complete problem. The prompt, seen in Figure 13, uses few-shot to ensure that the output format remains consistent throughout the runs. The values that the model can choose are:

- IMPOSSIBLE: the state cannot generate an answer to the task

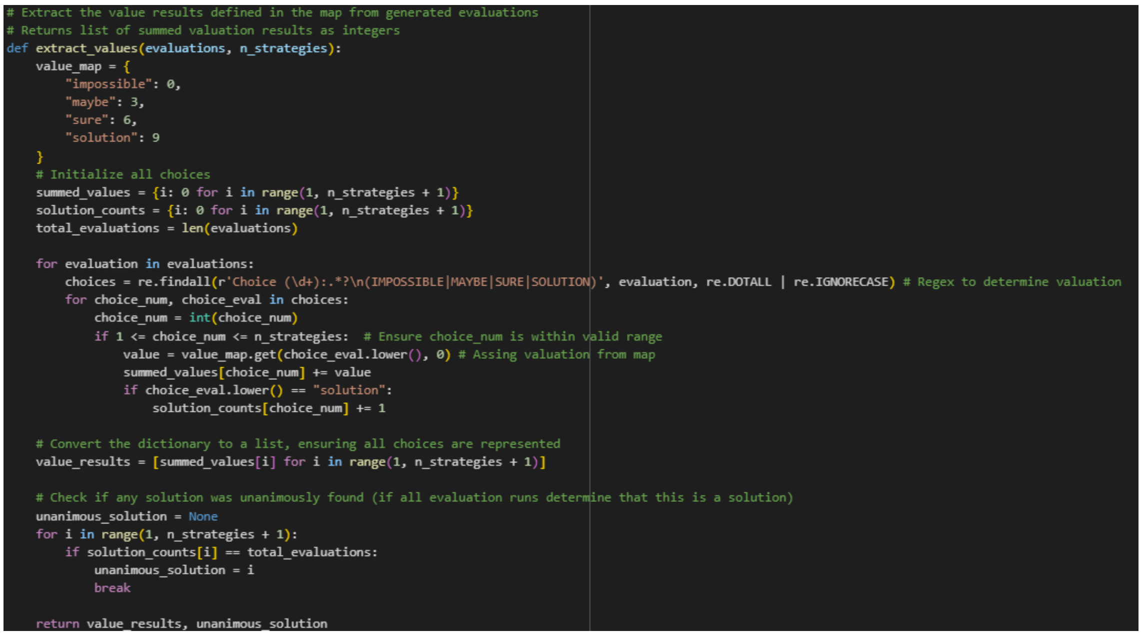
- MAYBE: the state could potentially generate an answer to the task
- SURE: the state is likely to generate an answer to the task
- SOLUTION: the state is an answer to the task



```
#define few-shot valuation used for tot dfs
value_prompt = '''Given an instruction and several choices, independently evaluate how likely the given choices could lead to a solution.
For each choice, provide a brief analysis followed by a single-word evaluation on a new line. Use only one of these evaluations: IMPOSSIBLE, MAYBE, SURE, or SOLUTION.
Use SOLUTION only if the choice contains the exact solution in this format: {format}
Example output structure:
Choice 1: [Your analysis here]
MAYBE

Choice 2: [Your analysis here]
IMPOSSIBLE

Choice 3: [Your analysis here]
SURE

Choice 4: [Your analysis here]
SOLUTION
Ensure that your response ends with one of the four evaluation words on its own line for each choice.
'''
```

Figure 13. Value prompt for ToT with DFS on GSM8K.

To accompany this, a value extraction function was developed. Seen in Figure 14, this function uses a value map to assign the classifications to their respective numbers. After this, a regular expression is used to extract the classification from the generated evaluation and the sum of all evaluations are returned as the total valuation for each solution. This function also determines whether a unanimous solution is found thus leading to an early exit of the DFS.



```python
# Extract the value results defined in the map from generated evaluations
# Returns list of summed valuation results as integers
def extract_values(evaluations, n_strategies):
    value_map = {
        "impossible": 0,
        "maybe": 3,
        "sure": 6,
        "solution": 9
    }
    # Initialize all choices
    summed_values = {i: 0 for i in range(1, n_strategies + 1)}
    solution_counts = {i: 0 for i in range(1, n_strategies + 1)}
    total_evaluations = len(evaluations)

    for evaluation in evaluations:
        choices = re.findall(r'Choice (\d+):.*?\n(IMPOSSIBLE|MAYBE|SURE|SOLUTION)', evaluation, re.DOTALL | re.IGNORECASE) # Regex to determine valuation
        for choice_num, choice_eval in choices:
            choice_num = int(choice_num)
            if 1 <= choice_num <= n_strategies:  # Ensure choice_num is within valid range
                value = value_map.get(choice_eval.lower(), 0) # Assing valuation from map
                summed_values[choice_num] += value
                if choice_eval.lower() == "solution":
                    solution_counts[choice_num] += 1

    # Convert the dictionary to a list, ensuring all choices are represented
    value_results = [summed_values[i] for i in range(1, n_strategies + 1)]

    # Check if any solution was unanimously found (if all evaluation runs determine that this is a solution)
    unanimous_solution = None
    for i in range(1, n_strategies + 1):
        if solution_counts[i] == total_evaluations:
            unanimous_solution = i
            break

    return value_results, unanimous_solution
```

Figure 14. Value extraction function.

Seeing that both search algorithms implemented were already done in some manor by S. Yao *et al.* [18], there is still a demand to implement a new search algorithm to the ToT framework. As discussed in Section 2.5, the A* search is the most appealing algorithm to implement.

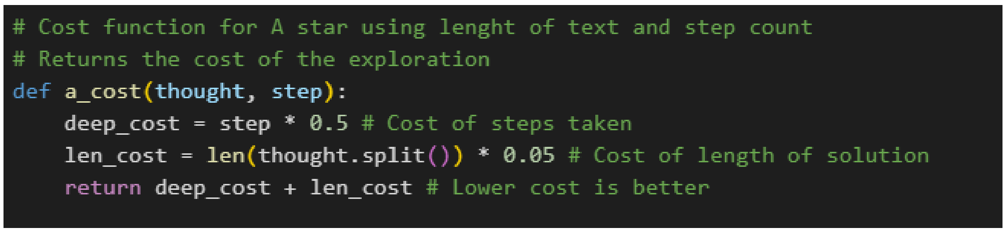
4.3 Implementing A* Search into ToT

The A* Search algorithm implementation uses the BFS' method of exploring the search space in combination with a greedy selection method to gather information on what state to explore next. The code for this is based on the Python implementation on GeeksforGeeks[34] which uses a heap to keep track of exploration order of the unvisited nodes. The unvisited nodes are kept in an open list and the visited nodes are kept in a closed set. The search continues until a solution is found, the maximum depth of the tree is reached, or when the open list is empty. The order of the heap is determined by the lowest value of *f(x)* which is the sum of our heuristic function *h(x)* and cost function *g(x)*. These values are calculated and assigned when a new node is discovered and is followed by adding said node to the open list. The selected state evaluator to use for this implementation is the voting system from the BFS implementation because the comparison evaluation method seems more suited for this implementation.

When discussing what cost and heuristic functions to implement for the ToT, there are a few key points that are considered. Firstly, the cost function should be an estimation of how much resources were used to get to the current state. Secondly, the heuristic function should be implemented as an estimation of how far the current state is to the final solution of the problem. The priority of this estimation is to not overshoot the distance from the current state to the final solution. For these reasons, the chosen cost and heuristic functions are shown in Figure 15 and Figure 16.

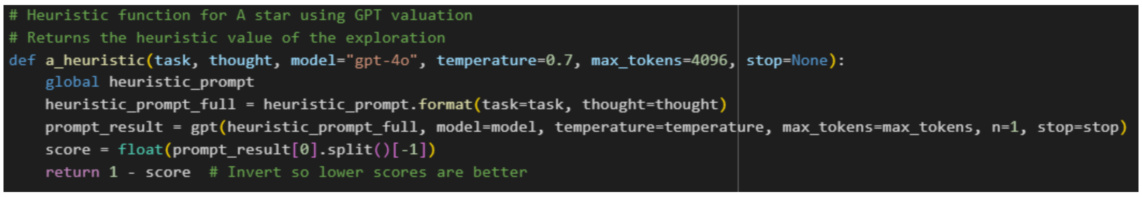
The cost function uses the steps taken so far, and the length of the current generated solution to evaluate how much computational resources are used for the current solution. The length of the solution is used with the goal of prioritizing simpler solutions compared to more complex solutions. These are weighed differently by scalars to mitigate any one factor outweighing the other.

The heuristic function implemented uses a technique inspired by the state evaluators from prior implementations. A prompt was developed for this heuristic to be used by the LLM to give a value from 0 to 1 on how close the current thought is to a complete solution of the task. As shown in Figure 17, the few-shot prompt is formatted in a way that the value extraction can be done without using regular expressions.



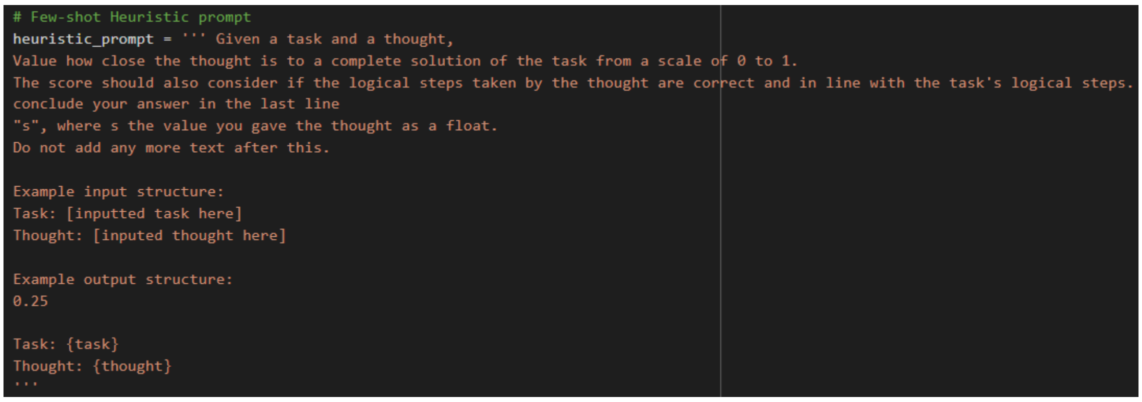
```python
# Cost function for A star using lenght of text and step count
# Returns the cost of the exploration
def a_cost(thought, step):
    deep_cost = step * 0.5 # Cost of steps taken
    len_cost = len(thought.split()) * 0.05 # Cost of length of solution
    return deep_cost + len_cost # Lower cost is better
```

Figure 15. Cost function for ToT with A* Search.



```python
# Heuristic function for A star using GPT valuation
# Returns the heuristic value of the exploration
def a_heuristic(task, thought, model="gpt-4o", temperature=0.7, max_tokens=4096, stop=None):
    global heuristic_prompt
    heuristic_prompt_full = heuristic_prompt.format(task=task, thought=thought)
    prompt_result = gpt(heuristic_prompt_full, model=model, temperature=temperature, max_tokens=max_tokens, n=1, stop=stop)
    score = float(prompt_result[0].split()[-1])
    return 1 - score  # Invert so lower scores are better
```

Figure 16. **Heuristic function for ToT with A* Search.**



```python
# Few-shot Heuristic prompt
heuristic_prompt = ''' Given a task and a thought,
Value how close the thought is to a complete solution of the task from a scale of 0 to 1.
The score should also consider if the logical steps taken by the thought are correct and in line with the task's logical steps.
conclude your answer in the last line
"s", where s the value you gave the thought as a float.
Do not add any more text after this.

Example input structure:
Task: [inputted task here]
Thought: [inputed thought here]

Example output structure:
0.25

Task: {task}
Thought: {thought}
'''
```

Figure 17. **Evaluation prompt for heuristic function.**

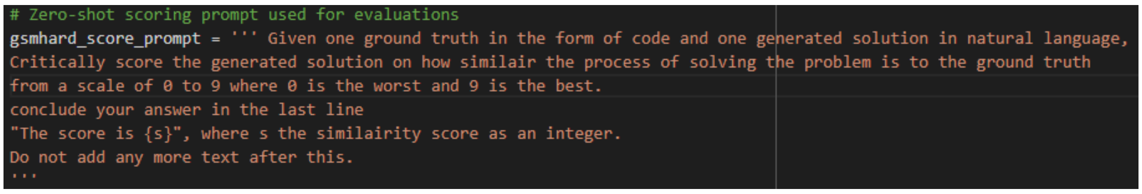
Using these various implementations of the ToT framework ensures that there is enough variety in the results collected from further experiments thus allowing for better assumptions to be made on how well the LLM performs using ToT. To showcase multiple reasoning capabilities of LLMs, additional benchmarks shall be considered that cover different types of reasoning and enhanced difficulty from the initial experiment.

5 Additional LLM Benchmarks

5.1 GSM-HARD

In terms of implementation, the GSM-HARD dataset is very similar to the GSM8K dataset. Both are saved as JSON files with similar input values. The difference between them is noticeable when analysing the output format. While GSM8K has its output as the reasoning steps along with the answer under one variable, GSM-HARD has two output variables: One is for the code that shows the reasoning steps in code form, and the other is the value of the solution as a number. Adapting the existing GSM8K code to work with the GSM-HARD dataset was accomplished in the following changes:

1. The structure of the chosen tasks for a run considers the logic steps and solution separately
2. The accuracy score function extracts the ground truth differently for GSM-HARD and the margin of error for what is considered a correct answer has changed from exact value matching to values that are at most 0.1 different from each other
3. The self-evaluation accuracy function uses a different score prompt for GSM-HARD that only considers the process of solving a solution given the truth as code and the generated solution as natural language. Along with the prompt in Figure 18, the ground truth is initialized by concatenating the solution to the end of the code.



```
# Zero-shot scoring prompt used for evaluations
gsmhard_score_prompt = ''' Given one ground truth in the form of code and one generated solution in natural language,
Critically score the generated solution on how similair the process of solving the problem is to the ground truth
from a scale of 0 to 9 where 0 is the worst and 9 is the best.
conclude your answer in the last line
"The score is {s}", where s the similairity score as an integer.
Do not add any more text after this.
'''
```

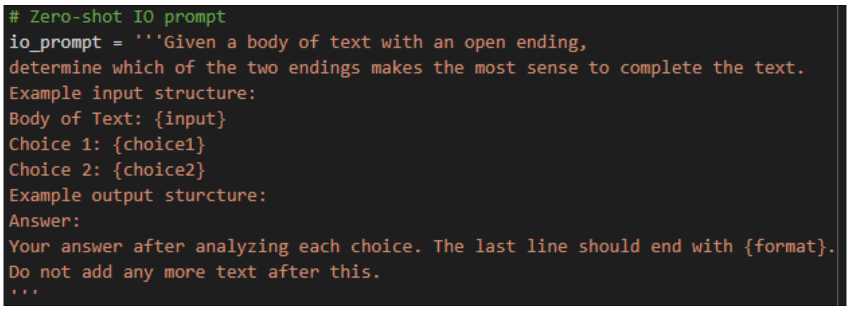
Figure 18. **Score prompt for GSM-HARD Self Evaluation Accuracy.**

Because this benchmark is testing the mathematical reasoning ability of the LLM on more difficult numbers, this is not sufficient to make assumptions on the general reasoning abilities of the LLM. For this reason, another benchmark is selected that tests a different type of reasoning task. As discussed in Section 2.2, the benchmark chosen is the StoryCloze dataset which will test the commonsense reasoning ability of the LLM.

5.2 StoryCloze

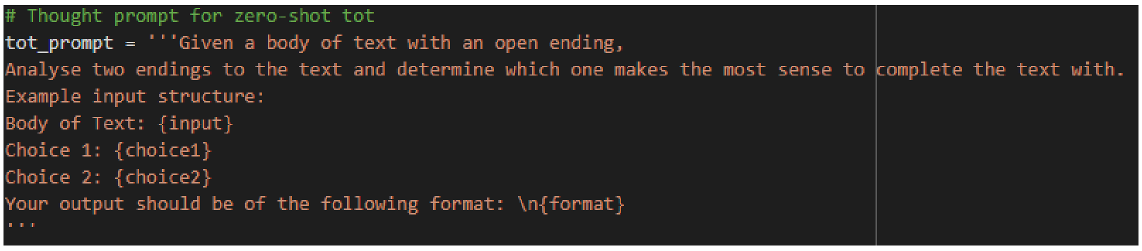
The StoryCloze dataset has more noticeable differences compared to both the GSM8K and GSM-HARD datasets because of how different the objectives are. To start, the StoryCloze data that is being implemented is the validation set because the test set provided does not have outputs to compare the generated answers to. Following this, the file is a csv with multiple datatypes per task in different columns. This required modifications to accommodate how the data is read in and processed before being passed onto the different solvers for the prompting techniques. The prompts that were developed for this dataset took account for the binary answer format and the task at hand. These altered prompts can be seen in Figure 19 and Figure 20.

Along with these changes, the state evaluator implemented for all ToT techniques implemented is the vote system. The reason for this is because the search space for this task is given as two possible answers, thus the creativity of the generated solutions is limited compared to the search space for GSM8K and GSM-HARD. Even so, the maximum depth chosen for this benchmark is also two because it employs a strategy layer and an answer layer. The search space limit also affects the self-evaluation accuracy given that no reasoning process is provided in the dataset. Because of this, only the comparison accuracy will be considered as the evaluation metric for StoryCloze.



```
# Zero-shot IO prompt
io_prompt = '''Given a body of text with an open ending,
determine which of the two endings makes the most sense to complete the text.
Example input structure:
Body of Text: {input}
Choice 1: {choice1}
Choice 2: {choice2}
Example output sturcture:
Answer:
Your answer after analyzing each choice. The last line should end with {format}.
Do not add any more text after this.
'''
```

Figure 19. IO prompt for StoryCloze.



```
# Thought prompt for zero-shot tot
tot_prompt = '''Given a body of text with an open ending,
Analyse two endings to the text and determine which one makes the most sense to complete the text with.
Example input structure:
Body of Text: {input}
Choice 1: {choice1}
Choice 2: {choice2}
Your output should be of the following format: \n{format}
'''
```

Figure 20. ToT prompt for StoryCloze.

6 Results

6.1 Overview of Results

From the adjustments made throughout chapters 4 and 5, the final experiments were executed on GPT-3.5-turbo using the parameters of 20 randomly selected tasks for all benchmarks, as well as, 20 ranked tasks for GSM8K. The number of generations and evaluations was 5 and the maximum depth was 2. For BFS and A*, the percentage of selected states to continue exploring was 20% and the maximum steps taken by the DFS was 10. The temperature of the LLM is set to 0.7 to match the parameter S. Yao *et al.*[18] used. With these parameters, the results from 20 Random GSM8K tasks are seen in Table 3 , 20 ranked GSM8K tasks in Table 4, 20 random GSM-HARD tasks in Table 5 and 20 random StoryCloze tasks in Table 6.

Table 3. Results from prompting techniques on random GSM8K tasks using GPT-3.5-turbo

Random 20 Tasks	**Prompting Techniques:**						
	IO	ToT + Greedy BFS	ToT + Sample BFS	ToT + Greedy DFS	ToT + Sample DFS	ToT + Greedy A*	ToT + Sample A*
Runtime *(s)*	32	119	114	512	455	116	113
Cost *($)*	0.05	0.63	0.62	2.29	2.23	0.80	0.80
Accuracy *(%)*	55	50	55	30	50	60	50
Self-Accuracy *(%)*	66	69	67	62	74	73	71
Self-Analysis Cost ($)	0.12	0.14	0.14	0.14	0.14	0.14	0.14

Table 4. Results from prompting techniques on challenging GSM8K tasks using GPT-3.5-Turbo

Ranked 20 Tasks	Prompting Techniques:						
	IO	ToT + Greedy BFS	ToT + Sample BFS	ToT + Greedy DFS	ToT + Sample DFS	ToT + Greedy A*	ToT + Sample A*
Runtime *(s)*	39	149	144	510	488	129	141
Cost *($)*	0.07	0.78	0.79	2.87	2.83	1.01	1.00
Accuracy *(%)*	40	50	40	15	15	40	50
Self-Accuracy *(%)*	67	70	68	79	64	66	72
Self-Analysis Cost ($)	0.18	0.21	0.21	0.19	0.19	0.21	0.21

Table 5. Results from prompting techniques on random GSM-HARD tasks using GPT-3.5-Turbo

Random 20 Tasks	Prompting Techniques:						
	IO	ToT + Greedy BFS	ToT + Sample BFS	ToT + Greedy DFS	ToT + Sample DFS	ToT + Greedy A*	ToT + Sample A*
Runtime *(s)*	28	101	109	392	327	117	124
Cost *($)*	0.05	0.64	0.65	2.26	1.88	0.83	0.83
Accuracy *(%)*	25	50	30	25	20	35	30
Self-Accuracy *(%)*	68	85	77	74	78	77	77
Self-Analysis Cost ($)	0.12	0.14	0.14	0.14	0.13	0.14	0.14

Table 6. Results from prompting techniques on random StoryCloze tasks using GPT-3.5-Turbo

Random 20 Tasks	**Prompting Techniques:**						
	IO	ToT + Greedy BFS	ToT + Sample BFS	ToT + Greedy DFS	ToT + Sample DFS	ToT + Greedy A*	ToT + Sample A*
Runtime *(s)*	35	121	133	125	136	125	139
Cost *($)*	0.06	0.76	0.77	0.80	0.79	0.94	0.95
Accuracy *(%)*	70	80	80	85	90	90	85

6.2 Analysis of Results

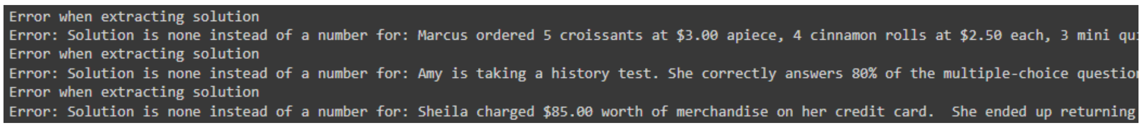
Analysing the results across multiple benchmarks shows how the performance of the IO prompting decreases when the complexity of the problem increases. This is an indication that a different prompting strategy should be implemented on GPT-3.5-turbo for more complex tasks. In each benchmark, at least one of the various ToT methods is shown to equal or outperform the IO prompting when only considering accuracy.

Starting with the random GSM8K tasks, the results suggests that the ToT is not preferred over IO prompting given that the best performing ToT method, ToT with Greedy selection and A*, only surpasses the performance on accuracy of the IO prompt by 5%, but for a larger cost. While there shows some promise with the self-evaluation accuracy scores, this implies that the reasoning steps are evaluated as better than the IO prompt, but the calculation result hinders the overall performance.

Within the same benchmark, the ranked GSM8K tasks shows more promise with four out of six ToT methods equalling or outperforming the IO prompt. The two methods that outperformed the IO prompt by 10% are the ToT + Greedy BFS and ToT + Sample A*. With both accuracy metrics surpassing the IO prompt, the cost of the Greedy BFS being less than the Sample A* would suggest that the ToT + Greedy BFS would be the preferred prompting strategy for this task. The DFS methods here should also be discussed given their low accuracy score but high self-evaluation accuracy. This could be a fault of the extraction method or the generated solution not following the format it was given. An example of this fault being detected in the output is shown in Figure 21.

Following this, the GSM-HARD benchmark experiment demonstrates the cause of increased complexity in the results. Five out of the six ToT methods outperformed the IO prompt by accuracy and all the ToT implementations outperformed the IO prompt by self-evaluating accuracy. The most impressive of the ToT methods on this benchmark is the ToT + Greedy BFS with a relative high accuracy, and self-evaluating accuracy score. Along with a low-cost relative to the other ToT methods, this justifies the cost of ToT for this benchmark.

Finally, the StoryCloze benchmark results show how all the ToT methods outperform the IO prompt. One interesting note from these results is how the DFS methods outperform the BFS methods for this task in accuracy score. This shows that depending on the task, the DFS could be better than the other two algorithms when taking cost and accuracy into consideration. But for overall performance on each benchmark, the greedy A* and the greedy BFS show the most promise for search algorithms to implement with ToT.



```
Error when extracting solution
Error: Solution is none instead of a number for: Marcus ordered 5 croissants at $3.00 apiece, 4 cinnamon rolls at $2.50 each, 3 mini qu
Error when extracting solution
Error: Solution is none instead of a number for: Amy is taking a history test. She correctly answers 80% of the multiple-choice questio
Error when extracting solution
Error: Solution is none instead of a number for: Sheila charged $85.00 worth of merchandise on her credit card.  She ended up returning
```

Figure 21. **Example of Extraction Error from the Ranked GSM8K tasks.**

7 Conclusion

7.1 Summary

This dissertation provides evidence that Tree of Thought prompting with various search algorithms can enhance the performance of large language models on complex reasoning tasks. By updating the existing code from S. Yao *et al.* [18], the implementation of different benchmarks and introducing a new informed search algorithm to the tree of thought framework was achieved. The analysis of the results from the multiple experiments has shown improvements in the models reasoning capabilities when using tree of thought on more complex tasks.

7.2 Evaluation

Looking back at the main goals set for this work in Section 1.2, the implementation and modification of the tree of thought framework was achieved. The emphasis of the type of problems was met for mathematical word problems and commonsense reasoning, but the multi-step problem was limited to multi step mathematical word problems. Multiple search algorithms were implemented and assessed for accuracy, cost and runtime thus the second goal was met successfully. Finally, the assessment of the model's performance using multiple benchmarks was met, but the variety in the types of reasoning tasks is not properly represented in the evaluations. Overall, the achievements are satisfactory for the allotted time for this research.

7.3 Future Work

Even with these results, there are limitations in how these experiments was performed. Some of them include:

1. Small number of runs: While these experiments has shown that the implementation of different search algorithms works on newer and older LLMs, the number of trials performed do not give a clear answer of if this prompting method gives better performance overall. Increasing this could lead to different and more accurate results.
2. Parameter Tuning: The experiments performed did not take various generations, evaluations, and number of states selected to continue searching into account. Tuning these parameters could lead to different results on performance.
3. More than two tree layers: The benchmarks that were used for experimentation had the same maximum depth. This limits the exploration on how different search algorithms perform with deeper tree structures.
4. Limited Search Space Complexity: From the experiments ran, the search space complexity is not rich enough for more complex search algorithms to make a major difference in performance. A* Search works best when the depth of the tree and the number of possible branches is many. For this reason, it can be assumed that for these experiments, A* was not at it's full potential to show a big difference in performance compared to BFS and DFS.

5. Natural Language Metrics: The metrics chosen for these experiments are not standardized for evaluating natural language. The self-evaluating accuracy is close to a natural language interpreter, but this is not commonly used by other researchers.

With more time and available budget, these limitations can be mitigated with further exploration of ToT prompting. But for the time being, the results are promising for the further development of ToT as a LLM performance enhancement. Some of the potential further development based on this research include parameter finetuning, expanding the benchmarks allotted for testing to show performance on deeper tree structures, more evaluation metrics to better gauge the natural language produced by the language model, combining the tree of thought prompting technique with other systems that use language models to reason or minimize the cost of the tree of thought further with various search algorithms that prune branches sooner. With these potential areas of research for Tree of Thought prompting, and more prompting techniques being developed, the directions for future research are limitless.

References

[1] W. X. Zhao *et al.*, “A survey of large language models,” 2023. Link: https://arxiv.org/abs/2303.18223

[2] A. Vaswani *et al.*, “Attention is all you need,” vol. 30, 2017. Link: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[3] T. Brown *et al.*, “Language models are few-shot learners,” vol. 33, pp. 1877–1901, 2020. Link: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] B. Goertzel, “Generative ai vs. agi: The cognitive strengths and weaknesses of modern llms,” 2023. Link: https://arxiv.org/abs/2309.10371

[5] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “GLUE: A multi-task benchmark and analysis platform for natural language understanding,” 2018. Link: https://arxiv.org/abs/1804.07461

[6] M. Chen *et al.*, “Evaluating large language models trained on code,” 2021. Link: https://arxiv.org/abs/2107.03374

[7] S. Yin *et al.*, “A survey on multimodal large language models,” 2023. Link: https://arxiv.org/abs/2306.13394v4

[8] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, “Hellaswag: Can a machine really finish your sentence?,” 2019. Link: https://arxiv.org/abs/1905.07830

[9] K. Cobbe *et al.*, “Training verifiers to solve math word problems,” 2021. Link: https://arxiv.org/abs/2110.14168

[10] Y. Zhang, J. Yang, Y. Yuan, and A. C.-C. Yao, “Cumulative reasoning with large language models,” 2023. Link: https://arxiv.org/abs/2308.04371

[11] P. Clark *et al.*, “Think you have solved question answering? try arc, the ai2 reasoning challenge,” 2018. Link: https://arxiv.org/abs/1803.05457

[12] K. Valmeekam, M. Marquez, A. Olmo, S. Sreedharan, and S. Kambhampati, “Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change,” vol. 36, 2024. Link: https://arxiv.org/abs/2206.10498

[13] E. Savaria, “Prompt Engineering Guide,” https://www.promptingguide.ai/ , December 2022

[14] J. Wei *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” vol. 35, pp. 24824–24837, 2022. Link: https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[15] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” vol. 35, pp. 22199–22213, 2022. Link: https://proceedings.neurips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[16] T. Lanham *et al.*, “Measuring faithfulness in chain-of-thought reasoning,” 2023. Link: https://arxiv.org/abs/2307.13702

[17] X. Wang *et al.*, “Self-consistency improves chain of thought reasoning in language models,” 2022. Link: https://arxiv.org/abs/2203.11171

[18] S. Yao *et al.*, “Tree of thoughts: Deliberate problem solving with large language models,” vol. 36, 2024. Link: https://arxiv.org/abs/2305.10601

[19] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” vol. 4, no. 2, pp. 100–107, 1968. Link: https://ieeexplore.ieee.org/abstract/document/4082128/

[20] T. Everitt and M. Hutter, “Analytical results on the BFS vs. DFS algorithm selection problem: Part II: Graph search,” 2015, pp. 166–178. Link: https://link.springer.com/chapter/10.1007/978-3-319-26350-2_15
[21] C. B. Browne et al., “A Survey of Monte Carlo Tree Search Methods,” vol. 4, no. 1, pp. 1–43, 2012, doi: 10.1109/TCIAIG.2012.2186810.Jacobson, J. and Andersen, O., editors. Software Controlled Medical Devices. SP Report 1997:11, Swedish National Testing and Research Institute, Sweden, 1997. Link: https://ieeexplore.ieee.org/abstract/document/6145622/
[22] L. Ranaldi, G. Pucci, F. Ranaldi, E. S. Ruzzetti, and F. M. Zanzotto, “A Tree-of-Thoughts to Broaden Multi-step Reasoning across Languages,” 2024, pp. 1229–1241. Link: https://aclanthology.org/2024.findings-naacl.78/
[23] P. Gao *et al.*, “Meta Reasoning for Large Language Models,” 2024. Link: https://arxiv.org/abs/2406.11698
[24] L. Gao *et al.*, “Pal: Program-aided language models,” 2023, pp. 10764–10799. Link: https://proceedings.mlr.press/v202/gao23f
[25] X. Lai *et al.*, “Lisa: Reasoning segmentation via large language model,” 2024, pp. 9579–9589. Link: https://openaccess.thecvf.com/content/CVPR2024/html/Lai_LISA_Reasoning_Segmentation_via_Large_Language_Model_CVPR_2024_paper.html
[26] J. Achiam *et al.*, “Gpt-4 technical report,” 2023. Link: https://arxiv.org/abs/2303.08774
[27] S. M. Kerner, “GPT-4o explained: Everything you need to know,” 2024. Link: https://www.techtarget.com/whatis/feature/GPT-4o-explained-Everything-you-need-to-know
[28] OpenAI, “Pricing,” 2024. Link: https://openai.com/api/pricing/
[29] B. Keum, J. Sun, W. Lee, S. Park, and H. Kim, “Persona-Identified Chatbot through Small-Scale Modeling and Data Transformation,” vol. 13, no. 8, p. 1409, 2024. Link: https://www.mdpi.com/2079-9292/13/8/1409
[30] N. Mostafazadeh *et al.*, “A corpus and evaluation framework for deeper understanding of commonsense stories,” 2016. Link: https://arxiv.org/abs/1604.01696
[31] R. Sharma, J. Allen, O. Bakhshandeh, and N. Mostafazadeh, “Tackling the story ending biases in the story cloze test,” 2018, pp. 752–757. Link: https://aclanthology.org/P18-2119/
[32] S. Bhadaniya, “Breadth First Search in Python (with code) | BFS Algorithm”, 2023, Link: https://favtutor.com/blogs/breadth-first-search-python
[33] S. Bhadaniya, “Depth First Search in Python (with code) | DFS Algorithm”, 2023, Link: https://favtutor.com/blogs/depth-first-search-python
[34] GeeksforGeeks, “A* search algorithm”, 2023, Link: https://www.geeksforgeeks.org/a-search-algorithm/ (Accessed Aug 9, 2024)
[35] Madaan, A. and Yazdanbakhsh, A. “Text and patterns: For effective chain of thought, it takes two to tango.” 2022. Link: https://arxiv.org/abs/2209.07686

Appendix 1

Sample GSM8K IO output:

```
Number of tasks: 20

Number of solved tasks: 20

run time: 104.0194640159607 seconds

GPT usage cost so far: {'completion_tokens': 4249, 'prompt_tokens': 2642, 'cost': 0.060875}

Task 27: 1

Ground Truth: 16

Generated Solution: 16.0

Score: 1

Ground Truth: Each container of ice cream has 15 servings and she eats 1 serving a night so
after 60 days she will need 60/15 = 4 containers of ice cream

If each carton costs $4.00 and she needs 4 containers then it will cost her 4*4 =
$<<4*4=16.00>>16.00

#### 16

Generated Solution: To determine how much Cynthia spends on ice cream after 60 days, we
first need to calculate how many servings of ice cream she consumes in that period.

Cynthia eats one serving of ice cream per day:

Number of servings in 60 days = 60 servings

Each carton contains 15 servings:

Number of cartons needed = 60 servings / 15 servings per carton = 4 cartons

Each carton costs $4.00:

Total cost = 4 cartons * $4.00 per carton = $16.00

Answer:

The answer is 16

Self evaluation Score: 1.00
```

Appendix 2

Sample GSM8K ToT Output:

```
Number of tasks: 20

Number of solved tasks: 20

run time: 624.483728647232 seconds

GPT usage cost so far: {'completion_tokens': 97589, 'prompt_tokens': 49547, 'cost': 1.23115}

Task 27: 1

Ground Truth: 16

Generated Solution: 16.0

Score: 1

Ground Truth: Each container of ice cream has 15 servings and she eats 1 serving a night so
after 60 days she will need 60/15 = 4 containers of ice cream

If each carton costs $4.00 and she needs 4 containers then it will cost her 4*4 =
$<<4*4=16.00>>16.00

#### 16

Generated Solution: Strategy:

1. Determine the total number of servings Cynthia will consume over 60 days.

2. Calculate how many cartons of ice cream are needed to provide the total servings.

3. Multiply the number of cartons by the cost per carton to find the total amount spent on ice
cream after 60 days.


Answer:

1. Total servings consumed in 60 days = 60 servings

2. Number of cartons needed = 60 servings / 15 servings per carton = 4 cartons

3. Total amount spent = 4 cartons * $4.00 per carton = $16.00


The answer is 16

Self evaluation Score: 1.00
```

Appendix 3 – User Guide

For GSM8K and GSMHARD:

- Download the ToT-GSM8K.ipynb, along with the data files, test.jsonl from the GSM8K data page on GitHub and gsmhardv2.jsonl from the PAL data folder from GitHub.
- Rename the test.jsonl to gsm8k.jsonl and the gsmhardv2.jsonl to gsmhard.jsonl
- Import both json files to the ToT-GSM8K.ipynb notebook
- Insert OpenAI Project API Key
- Run all cells until the runGSM8K and runGSMHARD functions are found
- After running this, you use one of these functions in a new cel with a number inputted as the number of tasks you want to be solved from the respective json file. By default, it is randomly selected tasks.
- Optional parameters include: ranked=selects n tasks by their difficulty from most to least, methods= for the solver you want to employ, task_ids=if you want to manually input the ids, max_depth=to adjust the max depth of the problem that you are implementing, max_steps=for the amount of steps the DFS will explore, model=for OpenAI model selection, return_results=if you want the results printed in the command line.

For StoryCloze:

- Download ToT-StoryCloze.ipynb and the data file cloze_test_val__winter2018-cloze_test_ALL_val - 1 – 1.csv
- Import the csv file to the ToT-StoryCloze.ipynb notebook
- Insert OpenAI Project API Key
- Run all cells until the runStoryCloze functions are found
- After running this, you use one of these functions in a new cel with a number inputted as the number of randomly selected tasks you want to be solved from the respective json file.
- Optional parameters include: methods= for the solver you want to employ, task_ids=if you want to manually input the ids, max_depth=to adjust the max depth of the problem that you are implementing, max_steps=for the amount of steps the DFS will explore, model=for OpenAI model selection, return_results=if you want the results printed in the command line.