1 Workflow Illustration

Problem Solving Using Tree_of_Thoughts: Initialize TreeOfThoughts() instance and call **solve(question)**. Executes the Tree-of-Thoughts reasoning framework by iterating through multiple steps, generating, evaluating, selecting, and refining thoughts to produce the most confident solution:

- **generate_thoughts()**: Creates multiple reasoning steps using a structured step-by-step prompt and calls chat_with_gpt to generate possible thought processes.
- **evaluate_thought()**: Assigns a confidence score (0 to 1) to each generated thought, caching evaluations to avoid redundant API calls.
- **select_best_thoughts()**: Uses a heap-based selection to retain the top k highest-scoring thoughts for further processing.
- **self_correct()**: Refines the final solution by reviewing logical consistency and correcting errors through chat_with_gpt().

Validation Process:

- **Load DeepInfra api key and validation dataset**: Load validation set from a JSON file.
- **Process data in batches**: Solve each question using solve(). Run validation and test datasets in parallel, processing 30 JSONL files per batch with 10 threads.
- **Evaluate performance and get results**: Display ToT and CoT accuracy and total token usage statistics. Will print the error answers to analysis but it's commented out for the sake of the beauty of the code results.

Testing and Submission:

- **Load test dataset and process questions in batches**: Load test set from a JSON file. Use solve() to generate answers and store results in a structured format.
- **Save submission file and rerify submission format**: Convert results into a CSV format and ensure it meet requirements. Validate column names, question ids, and numeric answers.

2 Core Featurs

BFS strategy: Employed a BFS strategy to investigate several solution path for each question. Use select_best_thoughts() before it determines which solution is worth exploring in greater depth. This method prevents the issue of being misleading by an invalid approach early in the process.

Greedy algorithm: Create an effective way of identifying the optimum ideas by implementing a greedy selection approach. Although this technique will not always yield the optimal solution, it is a pragmatic way of generating satisfactory results without letting the processing time get out of hand.

Structured Reasoning Prompt in both CoT and ToT: Solve the problem by following these 5 structured steps to get formatted results: Understand the Problem -> Plan the Solution -> Execute the Plan -> Verify Each Step -> Reflect on the Solution.

3 Experimental Analysis

Ablation Studies: Evaluate the influence of different hyperparameters on the accuracy in ToT and CoT reasoning frameworks. When adjusting hyperparameters, control variables to keep other parameters unchanged. In my experiments I just randomly sample 100 questions to save api cost and speed up:

- **temperature**: [0.1, 0.3, 0.5, 0.7, 0.9] ToT is stable (84%–87%) across all temperatures, whereas CoT fluctuates, peaking at 90% at 0.5 but declining afterward.
- **max_steps**: [4, 8, 12, 16, 20]. CoT achieves peak accuracy at 8 steps (82%) but declines with further increases. ToT remains stable between 78%-84%.

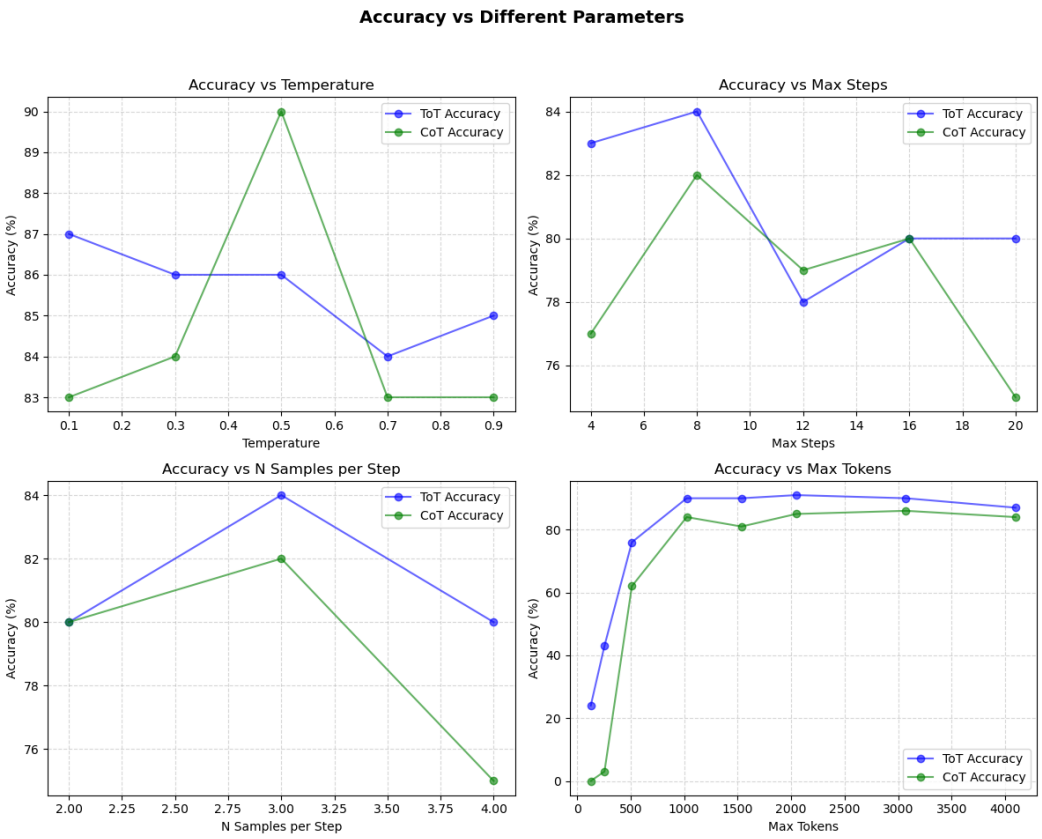

Figure 1: The influence of different hyperparameters on the accuracy of ToT and CoT

- **n_samples_per_step**: [2, 3, 4] Optimal performance is achieved at 3 samples per step (ToT: 84%, CoT: 82%), with additional samples reducing CoT accuracy.
- **max_tokens**: [128, 256, 512, 1024, 1536, 2048, 3072, 4096] Accuracy improves with longer sequences up to 2048 tokens. CoT starts at 0% at 128 tokens, improving to 86% at 3072 tokens. ToT peaks at 91% at 2048 tokens, but drops to 87% at 4096 tokens, indicating diminishing returns.

Concolusion: ToT demonstrates greater stability across hyperparameters, while CoT is more sensitive but can reach higher peak performance under optimal settings. Increasing test-time compute improves accuracy but exhibits diminishing returns at high resource levels.

4 Failure Cases Studies and Observations

- **One-shot examples decrease accuracy**: Providing a single correct example results in poor performance. Try to feed multiple correct and incorrect examples together or separately does not improve accuracy. The model might be struggle to learn in context if it's only seeing a single example. Large language models tend to perform better if they see sufficient diverse examples to identify patterns. Providing both correct and incorrect examples can cause confusion, which can result in confused learning.
- **Stop conditions might be unnecessary**: The structured format of the output may already be robust, reducing the need for strict stop conditions, and using multithreading significantly reduces runtime.