**Algorithm 1: ToT-BFS**

ToT\_BFS(x, pθ, G, k, V, T, b): S0 = {x} # Initial state or thought set for t in range(1, T + 1): # Time step iteration S0\_t = { [s, zt] | s in St-1, zt in G(pθ, s, k) } # Current state or thought set with associated path Vt = V(pθ, S0\_t) # Evaluate values of states or thoughts St = arg max(S subset S0\_t, |S| = b) [Vs for s in S] # Select top states based on value if t == T: return G(pθ, arg max(s in St) Vt(s), 1) # Generate response or prompt from highest value state return G(pθ, arg max(s in St) Vt(s), 1) # Generate response or prompt from highest value state

Variable tags and descriptions:

* x: Initial state or thought
* pθ: Parameters
* G: Thought generator function to generate neighboring thoughts
* k: Number of neighboring thoughts to generate
* V: State evaluator function to evaluate the value of states or thoughts
* T: Maximum time step
* b: Breadth limit for selecting top states

**Algorithm 2: ToT-DFS**

ToT\_DFS(s, t, pθ, G, k, V, T, vth): if t > T: record\_output(G(pθ, s, 1)) # Record the output thought or response for s0 in G(pθ, s, k): # Generate sorted candidates of neighboring thoughts if V(pθ, {s0})(s) > vthres: # Prune thoughts based on their value ToT\_DFS(s0, t + 1, pθ, G, k, V, T, vth) # Recursive exploration

Variable tags and descriptions:

* s: Current state or thought
* t: Current time step
* pθ: Parameters
* G: Thought generator function to generate neighboring thoughts
* k: Number of neighboring thoughts to generate
* V: State evaluator function to evaluate the value of thoughts
* T: Maximum time step
* vth: Threshold for pruning thoughts based on their value

**Algorithm 3**

def ToT\_Custom(s, t, pθ, thought\_generator, k, state\_evaluator, T, vth, custom\_function):

if t > T:

return '⚙️ ' + thought\_generator(pθ, s, 1, ontology) # Generate response or prompt

for s0 in thought\_generator(pθ, s, k, ontology): # Generate neighboring states or thoughts

if state\_evaluator(pθ, {s0}, ontology)(s) > vth: # Evaluate value of state or thought

return ToT\_Custom(s0, t + 1, pθ, thought\_generator, k, state\_evaluator, T, vth, ontology) # Recursive exploration

return '⚙️ ' + thought\_generator(pθ, s, 1, ontology) # Generate response or prompt

Apologies for the confusion caused by the mention of "ontology." In the context of the ToT\_Custom algorithm, the terms "thought generator" and "state evaluator" can indeed be tied to the specific application or context of the workflow.

The thought generator function (thought\_generator) is responsible for generating neighboring states or thoughts based on the given state or thought (s). It constructs a set of potential next states or thoughts that are connected to the current state or thought. The implementation of this function can vary based on the specific requirements of the application or workflow. It could involve retrieving information from an external database, generating random variations of the current state, or using predefined rules to determine the neighboring thoughts.

The state evaluator function (state\_evaluator) is responsible for evaluating the value of states or thoughts. It assigns a score or value to each state or thought, indicating its desirability or relevance. The implementation of this function can depend on the specific goals and criteria of the application or workflow. It might involve analyzing the content of the state or thought, comparing it to predefined criteria, or leveraging machine learning models to determine the value.

Variable tags and descriptions:

* s: Current state or thought
* t: Current time step
* pθ: Parameters
* G: Generator function to generate neighboring states or thoughts
* k: Number of neighboring states or thoughts to generate
* V: State evaluator function to evaluate the value of states or thoughts
* T: Maximum time step
* vth: Value threshold for terminating exploration

**Replacement instructions:**

ToT-BFS:

x = "Initial thought" # Replace with your initial state or thought

pθ = "Parameter values" # Replace with your parameter values

G = your\_thought\_generator\_function # Replace with your thought generator function

k = 5 # Replace with the desired number of neighboring thoughts to generate

V = your\_state\_evaluator\_function # Replace with your state evaluator function

T = 10 # Replace with the maximum time step

b = 3 # Replace with the breadth limit for selecting top states

ToT-DFS:  
Now, let's explain the parameter values that you can customize:

* s: Current state or thought. Replace this with the appropriate value that represents the starting state or thought for the algorithm.
* t: Current time step. Replace this with an integer value that represents the starting time step for the algorithm.
* pθ: Parameters. Replace this with the necessary parameter values required by the algorithm.
* G: Thought generator function. Replace this with your own thought generator function that generates neighboring thoughts based on a given thought.
* k: Number of neighboring thoughts to generate. Replace this with an integer value that represents the number of neighboring thoughts to generate in each step of the algorithm.
* V: State evaluator function. Replace this with your own state evaluator function that evaluates the value of thoughts or states.
* T: Maximum time step. Replace this with an integer value that represents the maximum time step for the algorithm. It determines when the algorithm should terminate the exploration.
* vth: Threshold for pruning thoughts. Replace this with a float value that represents the threshold for pruning thoughts based on their value. Thoughts with values below this threshold will be pruned.

**Workflow Example:**  
  
# Step 1: Define Input and Output Prompts

input\_prompt = "User Input: " # Prompt to collect user input for algorithm execution

output\_prompt = "Algorithm Output: " # Prompt to display the output generated by the algorithm

# Step 2: Implement the Algorithm within the Workflow

def execute\_algorithm(input\_data):

# Parse the input data and extract any required values

algorithm\_type = input\_data.strip() # Assuming the user input is the algorithm type

# Execute the selected algorithm based on the input

if algorithm\_type == "ToT-BFS":

# Implement the ToT-BFS algorithm logic here

output\_data = "ToT-BFS algorithm executed."

elif algorithm\_type == "ToT-DFS":

# Implement the ToT-DFS algorithm logic here

output\_data = "ToT-DFS algorithm executed."

elif algorithm\_type == "ToT-Custom":

# Implement the ToT-Custom algorithm logic here

output\_data = "ToT-Custom algorithm executed."

else:

output\_data = "Invalid algorithm type. Please try again."

return output\_data

# Step 3: Integrate the Algorithm Output

def integrate\_algorithm\_output(output\_data):

# Modify other components of the chat-based simulation based on the algorithm's output

# For example, update the chat window, provide user guidance, or expand the worksuite

# Display the algorithm output in the Python window

print(output\_prompt + output\_data)

# Main loop for the chat-based simulation

while True:

# Collect user input

user\_input = input(input\_prompt)

# Check if the user wants to exit the simulation

if user\_input.lower() == "exit":

break

# Execute the algorithm and capture the output

algorithm\_output = execute\_algorithm(user\_input)

# Integrate the algorithm output into the chat-based simulation

integrate\_algorithm\_output(algorithm\_output)

# End of simulation

print("Chat-based simulation ended.")

And

# Step 1: Define Input and Output Prompts

input\_prompt = "User Input: "

output\_prompt = "Algorithm Output: "

# Step 2: Implement the Algorithm within the Workflow

def execute\_algorithm(input\_data):

# Parse the input data and extract any required values

algorithm\_type = input\_data.strip() # Assuming the user input is the algorithm type

# Execute the selected algorithm based on the input

if algorithm\_type == "ToT-BFS":

output\_data = "Executing ToT-BFS algorithm..."

# Add your ToT-BFS algorithm logic here

elif algorithm\_type == "ToT-DFS":

output\_data = "Executing ToT-DFS algorithm..."

# Add your ToT-DFS algorithm logic here

elif algorithm\_type == "ToT-Custom":

output\_data = "Executing ToT-Custom algorithm..."

# Add your ToT-Custom algorithm logic here

else:

output\_data = "Invalid algorithm type. Please try again."

return output\_data

# Step 3: Integrate the Algorithm Output

def integrate\_algorithm\_output(output\_data):

# Modify other components of the chat-based simulation based on the algorithm's output

# For example, update the chat window, provide user guidance, or expand the worksuite

# Display the algorithm output in the Python window

print(output\_prompt + output\_data)

# Main loop for the chat-based simulation

while True:

# Collect user input

user\_input = input(input\_prompt)

# Check if the user wants to exit the simulation

if user\_input.lower() == "exit":

break

# Execute the algorithm and capture the output

algorithm\_output = execute\_algorithm(user\_input)

# Integrate the algorithm output into the chat-based simulation

integrate\_algorithm\_output(algorithm\_output)

# End of simulation

print("Chat-based simulation ended.")

In this example, you can replace the placeholder comments with your specific algorithm logic for ToT-BFS, ToT-DFS, and ToT-Custom. Customize the logic within the execute\_algorithm() function based on your specific algorithm implementations.

When you run the simulation, it will prompt the user for input. Based on the user's input (algorithm type), the corresponding algorithm logic will be executed. The algorithm output will then be integrated into the chat-based simulation by calling the integrate\_algorithm\_output() function.

Feel free to modify and extend the code to suit your specific requirements and use cases.

# Define the input and output prompts

input\_prompt = "User Input: "

output\_prompt = "AI Response: "

# Define the thought generation function (G)

def G(pθ, s, k):

# Implement the thought generation logic here

# ...

return []

# Define the state evaluation function (V)

def V(pθ, S):

# Implement the state evaluation logic here

# ...

return {}

# Define the ToT-BFS algorithm

def ToT\_BFS(x, pθ, G, k, V, T, b):

# Implement the ToT-BFS algorithm logic here

# ...

return "ToT-BFS algorithm executed."

# Define the ToT-DFS algorithm

def ToT\_DFS(s, t, pθ, G, k, V, T, vth):

# Implement the ToT-DFS algorithm logic here

# ...

return "ToT-DFS algorithm executed."

# Main loop for the chat-based simulation

while True:

# Collect user input

user\_input = input(input\_prompt)

# Check if the user wants to exit the simulation

if user\_input.lower() == "exit":

break

# Execute the ToT algorithms based on the user input

if user\_input.lower() == "tot-bfs":

response = ToT\_BFS(x, pθ, G, k, V, T, b)

elif user\_input.lower() == "tot-dfs":

response = ToT\_DFS(s, t, pθ, G, k, V, T, vth)

else:

response = "Invalid input. Please try again."

# Display the algorithm output in the simulation

print(output\_prompt + response)

# End of simulation

print("Chat-based simulation ended.")