Knowledge Retrieval for Task Tree Extraction in Functional Object-Oriented Networks

Varun Sai Raigir
Department of Computer Science
University of South Florida
Tampa, Florida, USA
varunsair@usf.edu

Abstract— **In the realm of robotics and artificial intelligence research, the development of intelligent agents capable of understanding human intentions and executing tasks in diverse human-centered domains has been a significant challenge. This report delves into the creation and evaluation of algorithms for generating task trees in a Functional Object-Oriented Network (FOON), focusing on the goal of preparing various recipes. Two search algorithms, Iterative Deepening Search and Greedy Best-First Search, with two distinct heuristics, are analyzed and compared. The evaluation considers factors such as the number of functional units in the resulting task trees, memory usage, and computational complexity. The findings demonstrate that Greedy Best-First Search with the heuristic based on motion success rates is the most efficient approach, leading to more streamlined task trees with a reduced need for human intervention. This research contributes to the advancement of AI systems capable of creatively adapting to novel situations, particularly in the domain of AI-driven cooking.**

I. Introduction

In the realm of robotics and artificial intelligence (AI) research, significant efforts have been dedicated to creating intelligent agents capable of understanding human intentions and executing tasks in human-centered domains. These domains encompass a wide spectrum of applications, from assisting the elderly and disabled to food preparation and delivery. Developing AI systems for such applications is challenging due to the diversity of tasks and dynamic environments in which they operate.

In particular, the field of AI-driven cooking presents unique challenges. Ingredients and objects can vary greatly in form, shape, size, and state, and there are situations where an AI system may face difficulty executing a complete recipe due to the unavailability of specific ingredients or objects. This complexity is further heightened when a robot needs to adapt its knowledge to accommodate or exclude ingredients it might not have encountered previously.
Inspired by humans' ability to adapt creatively to novel situations, this research delves into the task of generating plans for unfamiliar scenarios based on the limited knowledge available to AI systems, represented through the Functional Object-Oriented Network (FOON).

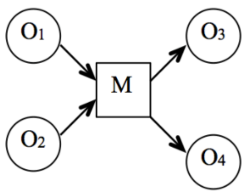


Fig. 1: A basic functional unit with two input nodes and two output nodes connected by an intermediary single motion node.[2]

FOON is a key knowledge representation in our AI system, enabling the creation of task trees for a wide range of recipes and ingredient variations. We aim to overcome the limitations by exploring ways to empower AI systems to handle novel recipes that may involve ingredient combinations not previously encountered within the network.
In our exploration, we will delve into the development of a recipe generation pipeline, the design of heuristic-based search algorithms, comparing the results to know which works better, and an evaluation of the generalizability of FOON for a wide array of recipes.

II. Functional Object-Oriented Network

Functional Object-Oriented Network (FOON), a graphical knowledge representation that captures high-level concepts related to human manipulation for service robots. FOON is a bipartite network comprising two fundamental types of nodes: object nodes and motion nodes. Object nodes represent physical entities, typically objects that are relevant to a particular task or action. These nodes can describe various attributes or states of objects, such as their size, shape, location, and condition. In our case of cooking they can represent vegetables, kitchen utensils, containers and their states such as vegetables being in cut state and etc. Motion nodes represent specific actions or motions that are part of a manipulation task. These nodes capture the movements, gestures, or manipulations required to interact with object nodes. For example, in the context of stirring tea, a motion node could represent the action of "stir". The connection between objects and actions is represented by edges, signifying both the relationship between objects and the order of actions within the network. A critical component of FOON is the concept of a "functional unit," where object nodes and motion nodes combine to describe a single action.

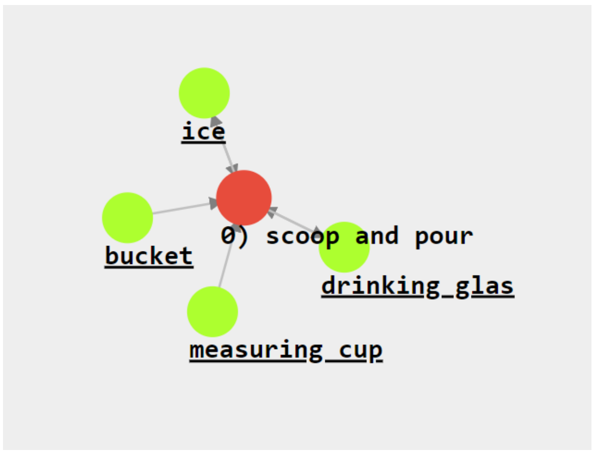


Fig. 2: A basic functional unit in FOON. The green nodes denote objects, while the red nodes denote motions.

A. *Creating a FOON*

To create a FOON, we define a critical concept known as a "functional unit." A functional unit represents an individual action in an activity and serves as a unit of description for the state changes of objects before and after execution. Input object nodes detail the required states of objects essential for a specific task, while output object nodes depict the outcomes of these inputs. In some cases, the execution of an action may not lead to changes in all input object states, resulting in a lesser number of output object nodes than inputs.

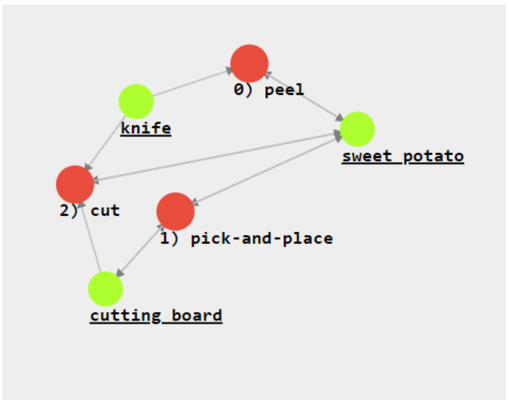


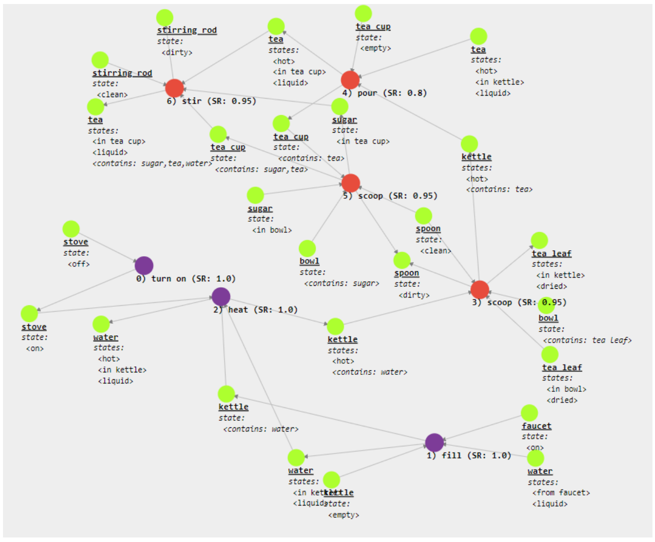
Fig. 3: A subgraph showing how to get sweet potato ready to use after going through the functional units which cut and peel the sweet potato.

The process of generating FOON involves annotating video demonstrations, converting them into the FOON graph structure, and documenting the actions, objects, and functional units leading to the creation of a particular meal or product. Each FOON that represents a single activity is referred to as a "subgraph." Subgraphs consist of functional units organized in a sequence to describe the states of objects before and after each action, the corresponding time-stamps in the source demonstration, and the manipulated objects. As of now, this annotation process is largely manual. However, on-going efforts aim to explore semi-automatic approaches [23]. To construct a "universal FOON," two or more subgraphs are merged by performing a union operation on all functional units. This process effectively eliminates duplicate functional units.

B. *Integrating Weights into FOON*

In previous iterations of FOON [5], [6], all motions were assigned equal weights within the network. This essentially implied that all manipulations could be executed by any robot without considering the robot's likelihood of successfully performing each action. However, this approach lacks realism, as different robots are designed with varying capabilities, precision, and dexterity.

To address this limitation, we introduce the concept of "weights" into FOON. These weights serve as indicators of the difficulty level associated with each manipulation, and they are based on the robot's success rate in executing a given action. Success rates are assigned as percentages to the motion nodes of each functional unit and consider factors like the robot's physical capabilities, past experiences, and ability to execute actions. Tools or objects used in manipulation also influence these values.



stirring_rod state: <dirty>
tea states: <hot> <in tea cup> <liquid>
tea_cup state: <empty>
tea states: <hot> <in kettle> <liquid>
stirring_rod state: <clean>
6) stir (SR: 0.95)
4) pour (SR: 0.8)
tea states: <in tea cup> <liquid> <contains: sugar,tea,water>
tea_cup state: <contains: sugar,tea>
tea_cup state: <contains: tea>
sugar state: <in tea cup>
kettle states: <hot> <contains: tea>
5) scoop (SR: 0.95)
sugar state: <in bowl>
spoon state: <clean>
stove state: <off>
0) turn on (SR: 1.0)
bowl state: <contains: sugar>
spoon state: <dirty>
tea_leaf states: <in kettle> <dried>
3) scoop (SR: 0.95)
bowl state: <contains: tea leaf>
2) heat (SR: 1.0)
stove state: <on>
water states: <hot> <in kettle> <liquid>
kettle states: <hot> <contains: water>
tea_leaf states: <in bowl> <dried>
kettle state: <contains: water>
faucet state: <on>
1) fill (SR: 1.0)
water states: <in kettle> <liquid>
kettle state: <empty>
water states: <from faucet> <liquid>

Fig. 4: A weighted FOON depicting the preparation of tea. Here the SR is the success rate of robot in performing each motion mentioned.

C. *Task Trees*

Task trees are a vital aspect of FOON and describe the sequential execution of actions within an activity. These trees are composed of functional units in sequence, each detailing the changes in object states before and after the action. The timestamps of action occurrences and the manipulated objects are also recorded within a subgraph. This knowledge structure simplifies the understanding of the workflow within a specific task. When combining multiple task trees, you can visualize the entire process for a complex task or activity.

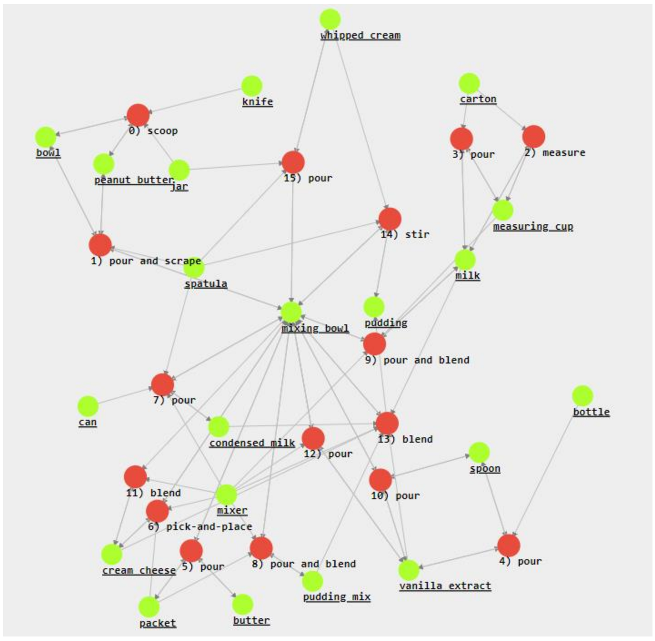


Fig. 5: An example of a Task Tree for getting whipped cream by using the input nodes which are readily available, using them and performing actions on them as functional units to create the final product.

FOON isn't just a knowledge representation; it's a tool for robots to solve problems. Imagine the robot has a goal. It uses a process called task tree retrieval to find a bunch of actions (functional units) it needs to reach that goal. This collection of actions is what we call a task tree. Unlike a standard collection of actions, a task tree doesn't stick to one example from a human - it learns from various sources to create a fresh plan.

To find the right actions, the robot needs a list of stuff in its environment (like what's in the kitchen). It checks which actions it can do based on what's available. This searching process is inspired by common graph-based methods, like the depth-first search (DFS) and best-first search (BFS). Starting from the goal, it looks for possible actions step by step.

Task Trees are retrieved using many methods and researchers have covered these methods in more detail in [5]. However, these methods don't include the new idea of using different search algorithms we are implementing on FOON in this work. So, we'll explore a different way to make task trees by considering how well actions work this way, we look at all the possible combinations of actions that could solve the problem.

III. Methodology

This project aims to develop a program for searching and retrieving task trees from the available Functional Object-Oriented Network (FOON) functional units of recipes and ingredients already available. For the products or the ingredients, it doesn’t know already we are implementing search algorithms to find other ingredients which will make up the required one. We are trying to implement two search algorithms, namely Iterative Deepening Search and Greedy Best-First Search, for finding the task trees that lead to the achievement of specific goals within the FOON. In the Greedy Best-First Search method we are using different heuristics to see which gives us better results.

A. *Input Data*

We have different input files which include FOON file, a text file which includes all the functional units from the recipes we already have. Each functional unit in the FOON file is a basic recipe or another action done to get output objects which are final products, or which can be directly used in other recipes. The next input file is a Json file containing goal nodes. These are the final products for which we need to find the task tree or way for them to be prepared. The other input file is also a Json file, the kitchen file which contains all the items or ingredients already present in the kitchen and we do not have to search or do anything else for them. There are other text files such as utensils.txt which have all the utensils available for us to work on the recipe. There is a motion.txt file which has the success rates of all the motions which a robot does in the process.

B. *Search Algorithms*

This project implements two search algorithms which are used to retrieve the task trees which give the functional nodes for getting the goal node. We will compare the working of the algorithms and see the results.

1. Iterative Deepening Search (IDS):

In this algorithm, the program explores the FOON nodes in a depth-wise manner, starting from the goal node. It keeps increasing the depth of exploration until a task tree is found that can lead to the desired goal object's state. The program aims to identify a solution task tree where the leaf nodes are objects available in the kitchen. We have used the following algorithm to implement the Iterative Deepening Search algorithm.

Pseudocode:

```
function search_IDS(kitchen_items, goal_node):
    max_depth = 0
    while True:
        task_tree_units = dfs_with_depth_limit(kitchen_items, goal_node, max_depth)
        if task_tree_units is not empty:
            return task_tree_units
        else:
            if max_depth > 9999:
                return task_tree_units
        max_depth += 1


function dfs_with_depth_limit(kitchen_items, goal_node, depth_limit):
    reference_task_tree = empty list
    items_to_search = empty list
    items_to_search.append(goal_node)
    items_already_searched = empty list


    while items_to_search is not empty:
        current_item = items_to_search.pop(0)
        if current_item is in items_already_searched:
            continue
        else:
            items_already_searched.append(current_item)
        if not check_if_exist_in_kitchen(kitchen_items, current_item):
            candidate_units = foon_object_to_FU_map[current_item.id]
            for selected_candidate in candidate_units:
                if selected_candidate is in reference_task_tree:
                    continue
                fu = foon_functional_units[selected_candidate]
                reference_task_tree.append(selected_candidate)
                for node in fu.input_nodes:
                    node_index = node.id
                    if node_index not in items_to_search:
```

```
            flag = True
            if node.label is in utensils and len(node.ingredients) is 1:
                for node2 in fu.input_nodes:
                    if node2.label is node.ingredients[0] and node2.container is node.label:
                        flag = False
                        break
            if node is not in items_to_search and node is not in items_already_searched:
                items_to_search.append(node)
    reference_task_tree.reverse()
    result = empty list
    for i in reference_task_tree:
        result.append(foon_functional_units[i])
return result
```

The Iterative Deepening Search (IDS) algorithm begins with the search_IDS function, which takes the available kitchen items and a target goal object as inputs. IDS progressively explores the search tree, increasing the maximum depth of the search iteratively. Within this iterative process, the dfs_with_depth_limit function delves into the search tree to discover a task tree, which represents a sequence of functional units needed to achieve the goal. The core of the algorithm involves managing lists of items to search and items already explored. It selects candidate functional units from the Functional Object-Oriented Network (FOON) and evaluates their relevance by checking if they are already part of the reference task tree. If a candidate unit is a potential addition, it is incorporated into the reference task tree, and its input nodes are assessed for further exploration. The algorithm proceeds until all items have been searched, and the resulting task tree is constructed, outlining the steps required to transform available objects into the desired goal. This iterative and depth-bound search process provides a systematic way to find an optimal solution in situations where multiple paths may lead to the goal.

2. Greedy Best-First Search

It is a pivotal search algorithm employed in this study to navigate the Functional Object-Oriented Network (FOON) efficiently. This algorithm is specifically tailored to guide the robot in achieving predefined goals within the context of complex, dynamic human-centered domains. The approach revolves around heuristic evaluation, where two distinct heuristics are utilized. The first heuristic, denoted as h(n), factors in the success rate of associated motion for each functional unit, allowing the algorithm to prioritize actions with the highest likelihood of successful execution. The second heuristic, h(n) = number of input objects in the functional unit, optimizes pathfinding by emphasizing paths that require fewer input objects.

Greedy Best-First Search strategically selects nodes within the FOON, continually assessing their heuristic values and choosing the node with the most favorable heuristic estimation. This prioritization facilitates the construction of task trees, which serve as blueprints for task execution. These task trees may draw knowledge from multiple sources, culminating in a task sequence optimized for efficiency and practicality. The algorithm plays a vital role in enhancing the adaptability and problem-solving capabilities of the robot, particularly in human-centered domains that demand adaptability and efficiency in task execution.

We have used two heuristic functions while implementing Greedy Best-First Search. They are as follows:

2.1. Success rate of motion.

The heuristic function for this algorithm is h(n) = success rate of the motion. We extract the data from motion.txt file and save it in a dictionary called motion_success_rate. We have used the following algorithm to implement this heuristic.

Psuedocode:

```
function BFS_Search(kitchen_items, goal_node)
reference_task_tree = []
items_to_search = [goal_node.id]
items_already_searched = []

while items_to_search is not empty:
    current_item_index = Dequeue(items_to_search)
    if current_item_index is in items_already_searched:
    Continue

    current_item = foon_object_nodes[current_item_index]

    if not check_if_exist_in_kitchen(kitchen_items, current_item):
    candidate_units = foon_object_to_FU_map[current_item_index]

    if the length of candidate_units is 1:
        selected_candidate_idx = candidate_units[0]
    else:
        success_rate = 0
        for each candidate in candidate_units:
        candidate_success_rate = motion_success_rate[foon_functional_units[candidate].motion_node]
        if candidate_success_rate > success_rate:
                success_rate=
                candidate_success_rate
                selected_candidate_idx = candidate

        if selected_candidate_idx is in
        reference_task_tree:
                Continue

        Append selected_candidate_idx to
        reference_task_tree
        for each node in
        foon_functional_units[selected_candidate_id
        x].input_nodes:
                node_idx = node.id
                if node_idx is not in
                items_to_search:
                flag = True

                if node.label is in utensils and the
                length of node.ingredients is 1:
                for each node2 in
                foon_functional_units[selected_can
                didate_idx].input_nodes:
                if node2.label is node.ingredients[0]
                and node2.container is node.label:
                        flag = False
                        Break

                if flag is True:
                Enqueue(node_idx to
                items_to_search)

    Reverse the reference_task_tree

    task_tree_units = []
    for each item in reference_task_tree:
        Append foon_functional_units[item] to
        task_tree_units

    Return task_tree_units
```

This pseudocode outlines the Greedy Best-First Search (BFS) algorithm applied in the context of constructing a task tree for completing kitchen tasks based on a given goal. The algorithm begins with an empty list to represent the reference task tree, a list for items to search, and a list of already explored items. It dequeues the goal node to initiate the search process. The algorithm explores the potential functional units (candidate_units) that can lead to the goal node and selects the best candidate using a heuristic. This heuristic aims to maximize the chance of successfully executing the motion required for the task. Once a candidate is chosen which has the maximum motion success rate among all the candidates available at the point, it is appended to the reference task tree. The algorithm then iteratively explores the input nodes of the selected functional unit and adds them to the list of items to search. Special consideration is given to whether specific items need to be combined for certain actions. After exploring all possible paths, the reference task tree is reversed to establish the correct sequence of actions. The resulting task tree comprises functional units that collectively accomplish the goal, representing a coherent plan for completing the specified kitchen task.

2.2. Number of input objects in the function unit.

The heuristic function for this algorithm is h(n) = number of input objects in the function unit. This selects the candidate with the least number of input objects and count the least number of ingredients in the input objects. We have used the following algorithm to implement this heuristic.

Psuedocode:

```
function search_BFS(kitchen_items, goal_node)
  reference_task_tree = []
items_to_search = []
  items_to_search.append(goal_node.id)
  items_already_searched = []

  while items_to_search is not empty
    current_item_index = items_to_search.pop(0)
    if current_item_index is in
items_already_searched
      continue

    else
items_already_searched.append(current_item_index)

    current_item =
foon_object_nodes[current_item_index]

    if not check_if_exist_in_kitchen(kitchen_items,
current_item)
      candidate_units =
foon_object_to_FU_map[current_item_index]

      if len(candidate_units) == 1
        selected_candidate_idx = candidate_units[0]
      else
        min_input_nodes = 999
        for candidate in candidate_units
          number_of_input_nodes = 0
```

```
                for input_node in foon_functional_units[candidate].input_nodes
                    if len(input_node.ingredients) == 0
                        number_of_input_nodes += 1
                    else
                        number_of_input_nodes += len(input_node.ingredients)
                if number_of_input_nodes < min_input_nodes
                    min_input_nodes = number_of_input_nodes
                    selected_candidate_idx = candidate
            if selected_candidate_idx is not in reference_task_tree
                reference_task_tree.append(selected_candidate_idx)

            for node in foon_functional_units[selected_candidate_idx].input_nodes
                node_idx = node.id
                if node_idx is not in items_to_search
                    flag = True
                    if node.label is in utensils and len(node.ingredients) == 1
                        for node2 in foon_functional_units[selected_candidate_idx].input_nodes
                            if node2.label == node.ingredients[0] and node2.container == node.label
                                flag = False
                                break
                    if flag
                        items_to_search.append(node_idx)

    reverse(reference_task_tree)

    task_tree_units = []
    for i in reference_task_tree
        task_tree_units.append(foon_functional_units[i])

    return task_tree_units
```

The provided pseudocode outlines a Breadth-First Search (BFS) algorithm used to search for a sequence of functional units in a knowledge graph called FOON (Functional Object-Oriented Network). This algorithm aims to find a path from a given goal node to a starting node, seeking to construct a task tree that outlines the necessary actions to achieve the goal. The BFS starts with the goal node and iteratively explores candidate functional units, aiming to select the one with the least input objects and ingredients among available options. This selection process prioritizes functional units that require fewer prerequisites, making it more efficient for the robot to execute the desired actions. The algorithm continues to explore and expand the task tree until the goal is reached, and the resulting task tree represents a sequence of functional units for achieving the specified task.

IV. EXPERIMENT

The experiment is done on the given FOON text file which has all the previously defined functional units ready to use in our recipe to prepare the goal node. We have 5 goal nodes which are whipped cream, greek salad, macaroni, sweet potato, and ice. Let's compare the performance of the search algorithms Iterative Deepening Search (IDS) and the Greedy Best-First Search for different heuristics on the basis of the number of functional nodes created in the task tree, the complexity at which the algorithm worked and the memory it used.

A. *Functional units in task trees*

The number of functional units in the task tree is the best way to measure the performance as this is not just for the algorithm running but the robot performing these tasks. So, the smaller number of functional units present in the task tree makes it easier for the goal to be reached in real-time.

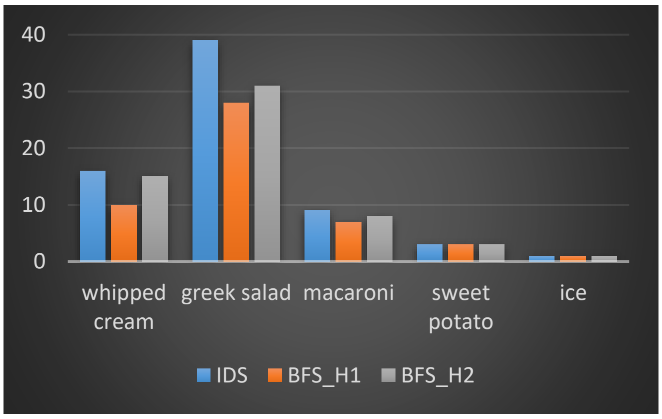


Fig. 6: Chart representing the number of functional units.

The number of functional units created by the Iterative Deepening Search is high compared to the Greedy Best-First search having high number of them for whipped cream, Greek salad and macaroni. This shows that it can cost us more while considering the robot working on preparing the goal node.

The number of functional units created by the Greedy Best-First Search with the heuristic of selecting the candidate who has the best success rate for its motion are least in number when compared to other algorithms for whipped cream, Greek salad and macaroni. This algorithm can give robots a higher success rate in completing the process requiring less help from humans.

The number of functional units created by the Greedy Best-First Search with the heuristic of selecting the candidate who has the least number of input nodes and having required least number of ingredients gave a few more functional nodes when compared to the heuristic with success rate for preparing whipped cream, Greek salad and macaroni. Its results are still better than IDS algorithm for the three goal nodes.

The number of functional units for goal nodes sweet potato and ice are the same for all the algorithms because their ingredients are mostly already present in the inputs and don't have to create a new way to prepare.

B. Memory Usage

Comparing the memory usage of the three provided pseudocodes, namely Iterative Deepening Search (IDS), Greedy Best-First Search with heuristic function based on motion success rate (GBFS-H1), and Greedy Best-First Search with heuristic function considering the number of input objects and their ingredients (GBFS-H2), is crucial for understanding their efficiency and performance.

Iterative Deepening Search (IDS): IDS is a memory-intensive algorithm as it relies on a depth-first search with depth limits. It maintains a reference task tree and a list of items to search. During its execution, the depth limit increases incrementally, which can lead to a considerable amount of memory usage, especially if the search space is extensive. IDS explores all possible paths up to a certain depth, potentially causing memory constraints.

Greedy Best-First Search with Heuristic H1 : GBFS with the motion success rate heuristic is relatively memory-efficient compared to IDS. It maintains a reference task tree and items to search but relies on a heuristic to select the best path. This method optimizes memory usage by selecting the candidate with the highest success rate for motion execution, reducing the number of explored paths and, consequently, memory overhead.

Greedy Best-First Search with Heuristic H2 : GBFS with the heuristic based on the number of input objects and their ingredients is also memory-efficient. It evaluates the input complexity of functional units to select the most efficient path. This approach can save memory by avoiding the exploration of overly complex paths.

C. Complexity

Comparing the computational complexity of the three provided algorithms, Iterative Deepening Search (IDS), Greedy Best-First Search with heuristic function based on motion success rate (GBFS-H1), and Greedy Best-First Search with heuristic function considering the number of input objects and their ingredients (GBFS-H2), is essential for understanding their efficiency and suitability for different scenarios.

Iterative Deepening Search (IDS): IDS is known for its simplicity and ease of implementation. It follows a depth-first search approach with increasing depth limits, making it a straightforward and complete algorithm. However, IDS can be computationally expensive, especially for large search spaces, as it explores all possible paths up to a certain depth. This approach results in exponential time complexity, O(b^d), where "b" is the branching factor and "d" is the depth limit. The memory complexity is relatively high, making IDS less suitable for resource-constrained environments.

Greedy Best-First Search with Heuristic H1 : GBFS-H1 offers a more computationally efficient approach by incorporating a heuristic based on motion success rates. This heuristic guides the algorithm to prioritize paths with higher success rates for executing actions. GBFS-H1 has a better time complexity compared to IDS, as it often converges to solutions more quickly by exploring promising paths first. However, the efficiency heavily depends on the quality of the heuristic. The memory complexity is also significantly lower than IDS, making GBFS-H1 suitable for situations where memory resources are limited.

Greedy Best-First Search with Heuristic H2 : GBFS-H2 further optimizes computational efficiency by considering the number of input objects and their ingredients when selecting paths. By minimizing the input complexity of functional units, it reduces the number of potential paths to explore. GBFS-H2 also offers a favourable time complexity, similar to GBFS-H1, and excels in terms of memory efficiency. This makes it a practical choice for resource-constrained environments or cases where memory usage needs to be minimized.

CONCLUSION

In summary, IDS is simple and complete but computationally expensive in terms of both time and memory. GBFS-H1 and GBFS-H2 provide more efficient alternatives, with GBFS-H2 being particularly well-suited for scenarios with stringent memory constraints. The choice of algorithm depends on the specific requirements of the problem at hand, considering factors like available computational resources and the quality of the heuristics.

IDS tends to consume more memory due to its depth-first nature, while both versions of GBFS make use of heuristics to make more memory-efficient decisions. GBFS-H1 considers motion success rates, optimizing the execution of actions, and GBFS-H2 focuses on reducing the number of input objects and their ingredients. The memory comparison suggests that the two GBFS versions are more memory-efficient alternatives when dealing with large search spaces, making them suitable for resource-constrained environments.

IDS tends to use more functional units compared to the GBFS algorithms. The GBFS-H1, which considers the success rate of motion uses the least number of functional units. By this, we can conclude that using Greedy Best-First Search-H1 considering motion success rate the best algorithm as it also requires less human help and higher success rate.

REFERENCES

[1] Md Sadman Sakib, Hailey Baez, David Paulius, and Yu Sun. Eval uating Recipes Generated from Functional Object-Oriented Network. arXiv preprint arXiv:2106.00728, 2021. (Featured in 18th Interna tional Conference on Ubiquitous Robots (UR 2021)).

[2] Task Planning with a Weighted Functional Object-Oriented Network.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. ArXiv, abs/2005.14165, 2020

[4] Md. Sadman Sakib, David Paulius, and Yu Sun. Approximate task tree retrieval in a knowledge network for robotic cooking. IEEE Robotics and Automation Letters, 7:11492–11499, 2022.

[5] [15] M. Beetz, D. Beßler, Andrei Haidu, M. Pomarlan, A. Bozcuoglu, ˘ and Georg Bartels. KnowRob 2.0 — A 2nd Generation Knowledge Processing Framework for Cognition-Enabled Robotic Agents. 2018

IEEE International Conference on Robotics and Automation (ICRA), pages 512–519, 2018

[6] Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2motion: From natural language instructions to feasible plans. arXiv preprint arXiv:2303.12153, 2023.

[7] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating situated robot task plans using large language models. In International Conference on Robotics and Automation (ICRA), 2023.

[8] OpenAI. Chatgpt. https://openai.com/, 2021. Accessed on 8th June 2023.

[9] Yongqiang Huang, Juan Wilches, and Yu Sun. Robot gaining accurate pouring skills through self-supervised learning and generalization. Robotics and Autonomous Systems, 136:103692, 2021.

[10] Javier Marin, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Sal vador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba. Recipe1M+: A Dataset for Learning Cross-Modal Embeddings for Cooking Recipes and Food Images. IEEE Trans. Pattern Anal. Mach. Intell., 2019.