Sequential Discrete Action Selection via Blocking Conditions and Resolutions

Liam Merz Hoffmeister[1], Brian Scassellati[1], Daniel Rakita[1]

***Abstract*— In this work, we introduce a strategy that frames the sequential action selection problem for robots in terms of resolving *blocking conditions*, i.e., situations that impede progress on an action en route to a goal. This strategy allows a robot to make one-at-a-time decisions that take in pertinent contextual information and swiftly adapt and react to current situations. We present a first instantiation of this strategy that combines a state-transition graph and a zero-shot Large Language Model (LLM). The state-transition graph tracks which previously attempted actions are currently blocked and which candidate actions may resolve existing blocking conditions. This information from the state-transition graph is used to automatically generate a prompt for the LLM, which then uses the given context and set of possible actions to select a single action to try next. This selection process is iterative, with each chosen and executed action further refining the state-transition graph, continuing until the agent either fulfills the goal or encounters a termination condition. We demonstrate the effectiveness of our approach by comparing it to various LLM and traditional task-planning methods in a testbed of simulation experiments. We discuss the implications of our work based on our results.**

I. Introduction

Robots offer the promising potential to assist people across a variety of domains, including home care, healthcare, agriculture, and manufacturing. In these scenarios, robots are tasked with determining the most beneficial action to take at any given moment to achieve specific goals or objectives.

Much previous work has investigated the action selection problem posed above [8, 9, 10, 11, 15]. At a high level, previously proposed algorithms often reason over the broad space of which actions are *viable* at a certain state. For instance, task planners often consider *prerequisites* that define whether a given action is viable, then use modeled effects to define how the environment would change if a certain action is executed. Using this information, a task planner considers many possible roll-outs of legal action trajectories, stopping when a sequence of actions reaches a given goal.

The action viability-centered view of task selection has afforded several useful algorithms (§II); however, the strategy as a whole can often lead to computational and representational challenges. For instance, if there are many viable actions that beget many viable actions to consider at a next step, and so on, the number of possible action branches scale exponentially with respect to the number of planning steps, often overwhelming computational resources. While heuristics can help alleviate this issue [9, 11], it may still be difficult to prune large segments of or find shortcuts through these dense branches of actions as it is often unclear which current actions will be necessary to achieve future goals. Also, given the uncertainty of the real-world, it can often be impossible to perfectly model which actions will be viable at what time. For instance, it is impossible to know if the robot will know where a target object is or if it will have the physical strength to lift a certain object ten steps into the future. Thus, if any modeled assumption is broken in the real world, a full re-plan often has to be triggered, exacerbating the computational challenges outlined above.

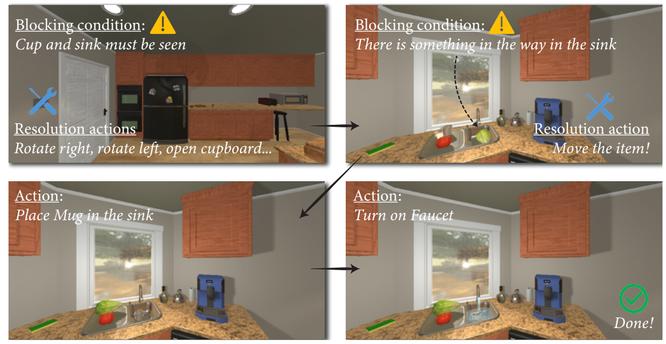

Fig. 1: We present a strategy that frames action selection in terms of resolving *blocking conditions*, i.e., situations that impede progress on an action en route to a goal. This example illustrates a robot using blocking conditions and resolutions to complete the task "wash the cup in the sink" in the AI2Thor simulation environment. This task is part of our evaluation.

In this work, we introduce a strategy that frames the action selection problem for robots in terms of resolving *blocking conditions*, i.e., situations that impede progress on an action en route to a goal. This strategy allows a robot to make one-at-a-time decisions that take in pertinent contextual information and swiftly adapt and react to current situations.

Our proposed strategy follows three high-level steps: (1) The robot *selects an action* from a set of candidate actions; (2) The robot assesses if any conditions are *blocking progress* on the current action; and (3) If progress is blocked on its current action, the robot evaluates all attempts to *resolve* the situation, incorporating possible resolutions into its set of candidate actions (circling back to Step 1). At this juncture, the robot may either choose an action that resolves the issue and resume the previously blocked action or prioritize a new action, reflecting an adapted strategy. This loop continues until the agent fulfills the goal criteria or encounters a termination condition.

Our central premise is that centering action selection de-

[1]Liam Merz Hoffmeister, Brian Scassellati, and Daniel Rakita are with the Department of Computer Science, Yale University, New Haven, CT 06520, USA `liam.merzhoffmeister@yale.edu`

This work was supported by Office of Naval Research award N00014-24-1-2124

cisions around blocking conditions and resolutions will ease the computational and representational challenges associated with discrete action selection, ultimately enabling more successful robot task execution. For instance, the exponential action branching challenge discussed above is eased by only considering actions that can either directly achieve the given goal or resolve a blocking condition currently impeding progress toward the goal. Our empirical results (§V) suggest that this set of candidate actions will be much smaller than the set of all viable actions at a particular state. Also, because action decisions are made one at a time in response to blocking conditions, it is straightforward to quickly adapt strategies when needed in the face of physical obstructions or lack of information that could not be modeled *a priori*.

We present a first instantiation of our strategy that combines a state-transition graph and a zero-shot Large Language Model (LLM). The state-transition graph tracks which previously attempted actions are currently blocked and which candidate actions may resolve existing blocking conditions (Steps 2 and 3 in the strategy above). This information from the state-transition graph is used to automatically generate a prompt for the LLM, which then uses the given context and set of candidate actions to select a single action to try next (Step 1 in the strategy above). This selection process is iterative, with each chosen and executed action further refining the state-transition graph. The algorithm continues iterating until the agent fulfills the goal criteria or encounters a termination condition.

We demonstrate the effectiveness of our approach by comparing it to various LLM and traditional task-planning methods in a testbed of simulation experiments (§V). We conclude by discussing the implications and limitations of our work based on our results. We provide open-source code for an implementation of our approach.[1]

II. Related Works

A. Task Planning

The goal of task planning is to compute a sequence of actions that achieve some given goal. Task planners typically reason over some logic-based domain language, such as STRIPS [8] or PDDL [15]. At a high level, these domain languages specify a start state, a goal state, and what constitutes legal, viable actions at any given state.

Over the years, several highly efficient task planning algorithms have been developed to find feasible sequences of actions from start to goal, including Fast Forward (FF) [11] and Fast Downward (FD) [9]. FF uses a graph search strategy coupled with a carefully designed heuristic that estimates the cost to reach the goal from the current state. FD decomposes planning tasks into more manageable sub-problems, allowing for significant speedups on a hierarchical representation.

Several task planners were subsequently presented that address the challenges of non-observability and uncertainties. For instance, FF-Replan [18], which modified the FF planner to address non-deterministic outcomes through a "determinize-and-replan" strategy, marked a significant shift towards managing unpredictable elements in task planning, although it did not completely resolve the issue of state uncertainties.

Further advancement was achieved through planning over belief states, i.e., some representation of many possible world states [3, 4]. For instance, Hoffmann and Brafman [10] presents implicit belief state planning where actions can have non-deterministic outcomes. Handling implicit belief states allows the planner to consider various contingencies and develop plans that are robust to uncertainties.

Our proposed strategy draws on many concepts from task planning. For instance, our strategy uses a logic-based domain language to specify goals and track information about the world and possible actions. However, our work differs in that instead of framing search in terms of what actions are viable from a certain state (or belief state), our strategy frames action selection in terms of which actions may resolve blocking conditions that have been observed. This strategy enables a robot to make one-at-a-time decisions that take in contextual information and adapt and react to current situations.

B. LLMs for Planning

Recent developments have demonstrated remarkable progress in natural language processing models, particularly Large Language Models (LLMs) [19]. One key area of exploration is the potential for these models to understand a planning query in natural language and generate a clear, step-by-step response to achieve a specific goal.

Modern LLMs are known to struggle with various planning-related tasks [5]. However, recent work has investigated this problem. For instance, work from Ahn et al. [2] and Chalvatzaki et al. [6] have demonstrated how LLMs can bridge the gap between abstract instructions and actionable tasks, leveraging their vast semantic understanding. These works resonate with our emphasis on adaptability and dynamic action selection in response to evolving environments. Also, the work by Song et al. [17] and Singh et al. [16] emphasizes the role of LLMs in generating dynamic plans, grounding them in the physical environment, and employing programmatic prompting to aid in plan formation.

Frameworks combining LLMs with structured planning, such as the work by Liu et al. [14] and Ding et al. [7], showcase the benefits of integrating LLM-derived insights to navigate unforeseen scenarios and improve flexibility in changing settings. Our first instantiation of our proposed strategy (outlined in §IV) fits well in this category. Our idea involves merging the contextual understanding provided by LLMs with the clarity and precision of a state-action graph representation. This integration aims to create a flexible and robust method for action selection, combining the best of both to enhance decision-making processes.

III. Strategy Overview

In this section, we describe the high level strategy proposed in this work: framing discrete action selection around

[1] https://github.com/Apollo-Lab-Yale/llm_task_planning

blocking conditions and resolutions. In §IV, we describe in detail one instantiation of this strategy which we subsequently use in our evaluation.

A. Strategy Components

Our strategy builds on the following components. Several of these components are also used in standard task planners, and are named the same, accordingly.

- A set of distinct nouns in the world called *instances*.
- A set of binary-valued functions of one or more instances, called *predicates*. Each application of a predicate to a specific set of instances can return true (called a *positive literal*) or false (called a *negative literal*). For example, suppose we have a predicate `Under` that takes two instances as arguments `Under(dirt, rug)`. This predicate would return true and be a positive literal if dirt is under the rug.
- The current setting of all literals over all instances will be called the *world state*.
- A set of *actions* which each provide a functional specification (through predicates) for how the robot can change the world state.
- Each action has a set of *effects*, which each specify the changes to the world state resulting from the execution of its action. Importantly, our strategy allows for effects with truth values of `possibly_true` or `possibly_false` to accommodate uncertain results.
- Each action has a set of *blocking conditions*, which each provide reasons for why the action may not succeed. For instance, example blocking modes for a "pick up item" action may be "target object not in view", "item is out of reach", or "item is too heavy to lift".
- Each blocking condition has a set of *resolutions*, which are literals that must be satisfied for the blocking mode to be considered resolved.
- Each blocking condition has a corresponding set of *resolution actions*, which are actions that have effects that would adjust the literals of the world to satisfy its resolutions.

B. Strategy Steps

The strategy starts by representing some goals as a set of literals. Actions with effects that directly achieve some aspect of these goals are added as roots of their own tree structures. Then, our task execution via blocking conditions and resolutions strategy proceeds as follows:

(1) The robot *selects an action* from a set of candidate actions. The set of candidate actions is all leaf nodes in the tree structures mentioned above. The module that makes this decision is called the *selection engine*; (2) The robot attempts executing the selected action and assesses if any *blocking conditions* are present on the current action; and (3) If the action executes successfully, it and all of its sibling nodes (i.e., nodes with the same parent) are removed from the tree structure. If progress is blocked on the current action, the robot evaluates all attempts to *resolve* the situation, adding all resolution actions for the blocking condition at hand as children nodes of the blocked action in the tree structure (circling back to Step 1).

At this juncture, the selection engine may either choose an action that resolves the issue in service of resuming the previously blocked action or prioritize a new action, reflecting an adapted strategy. These three steps iterate until the robot either satisfies the goal literals or encounters a termination condition.

C. Strategy Example

Here, we outline how our strategy may apply to a robot tasked with retrieving milk from a refrigerator. The goal literal for this task is `On(milk, counter) = true`.

There is one action that has an effect that achieves the given goal: `Place(milk, counter)`. The approach adds this action as a root node in a tree structure. The selection engine chooses this action (it is the only option) and attempts to execute it. However, a blocking condition is encountered: for an object to be placed, it must first be currently held. The resolution literal for this blocking condition is `isHolding(milk) = true`.

The robot has one possible way to elicit the desired effect: `grasp(milk)`. This action is added as a child node to the `place` action node. The action selection engine chooses the `grasp(milk)` action (again, the only option). The robot attempts to execute this action, but another blocking condition is met: for an object to be grasped, it must currently be visible. The resolution literal for this blocking condition is `isVisible(milk) = true`.

While no actions can guarantee the milk becoming visible, several actions *may* lead toward achieving the desired outcome. For example, the actions `visualSearch(direction_bias)`, `open(refrigerator)`, `open(freezer)`, `open(oven)` all have effects that include `isVisible(milk) = possibly_true`.

The selection engine chooses `open(refrigerator)`. The tree structure at this point in the task is visualized in Figure 2. Upon successfully opening the refrigerator, new instances become visible inside, such as yogurt, juice, etc. However, the milk is not immediately detected.

The selection engine opts for the `visualSearch(direction_bias)` action, this time applying a `direction_bias` toward the refrigerator. The robot maneuvers its viewpoint and locates the milk in the back of the refrigerator. The blocking condition regarding the milk's visibility is thereby resolved, and all other resolution actions related to this issue are consequently removed from the tree structure.

The action `grasp(milk)` is no longer obstructed, and the selection engine elects to pursue this action next. After the `grasp` action is executed, the blocking condition affecting the `place` action has been cleared. Consequently, the robot positions the milk on the counter, successfully accomplishing its given goal.

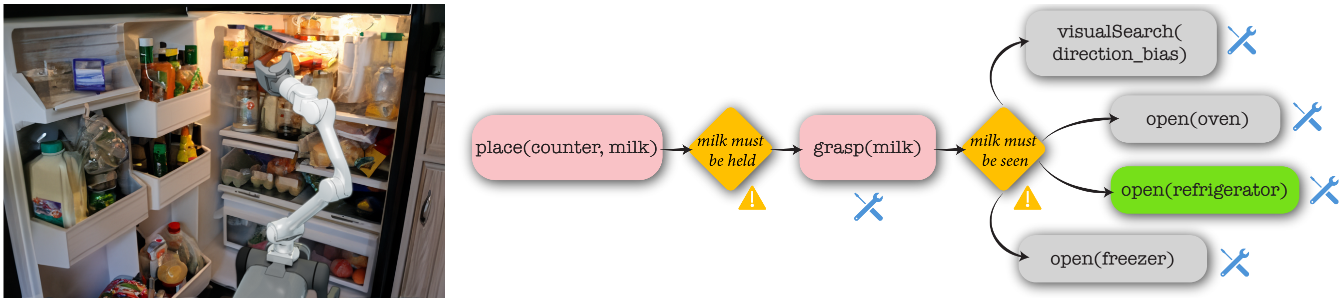

Fig. 2: The tree structure during the retrieving milk from the refrigerator example. The red nodes signify blocked actions, the orange diamonds signify blocking conditions, gray nodes signify available actions, the green node is the currently selected action, and the blue symbols signify resolution actions that may fix the parent blocking condition.

IV. Initial Instantiation of Strategy

In this section, we present an initial instance of the strategy discussed in §III. We also employ this implementation in our evaluation in §V.

A. Selection Engine using LLM

As outlined in Section III-B, an integral component of our proposed strategy involves a selection engine tasked with choosing an action from a set of candidate actions. In this work, we use an LLM to make this selection, aiming to use its understanding of context to effectively advance the robot toward its goal.

At each step, the algorithm first aggregates all candidate actions from which the selection action can be chosen. This set of candidate actions is comprised of all leaf nodes in the state-action graph, which represent all possible actions that have effects that may either directly achieve a specified goal or resolve a current blocking condition. For instance, in Figure 2, the set of feasible actions would be `[visualSearch(direction_bias), open(oven), open(refrigerator), open(freezer)]`. All of these actions may resolve the blocking condition "milk must be seen".

The set of all candidate actions is converted into a text string. This string is then incorporated into an expansive prompt, the specifics of which are elaborated upon in Section IV-B. The fully constructed prompt is subsequently dispatched to the LLM. Following the LLM's analysis of the prompt, it produces a response articulated in natural language. The action chosen by the LLM is identified and extracted from this output.

Steps 2 and 3 in §III-B progress with the action selected by the LLM module. When the approach circles back to Step 1 (the selection engine step), a new prompt is generated given the current context. Every decision the LLM makes regarding action selection is conducted afresh without leveraging any knowledge from its prior interactions or choices. While we speculate that maintaining continuous interaction with the LLM throughout all decisions may significantly impact the observed results, these investigations are beyond the scope of our current work. We discuss this point further in §VI.

B. LLM Prompt Generation

In our approach, generating prompts for the LLM involves populating a predefined template with specific details at each decision point. Given that the LLM does not maintain a history of prior interactions or decisions, it's crucial to repeatedly clarify its role and provide ample context within our framework. We outline below the eight sequential components that make up the prompt, with complete input and output examples in Appendix A and B:

(1) The prompt begins by stating the LLM's role, indicating that the LLM is assisting the user in selecting their next action. This sets a clear context and defines the scope of tasks expected to be addressed; (2) The prompt next incorporates instructions for the LLM, advising it to take its time and reason methodically, aligning with recommendations from the literature [12]; (3) An array of previous actions taken by the robot is added to the prompt. This summary gives a sense of history in the interaction to contextualize the current decision; (4) An array of sub-goals already completed by the robot is added to the prompt. Again, this list provides important context for the current decision; (5) The array of candidate actions is added to the prompt; (6) An array of remaining goal predicates are added to the prompt, focusing the robot's effort towards achieving its given goals; (7) Next, the prompt includes descriptions of any errors or failures encountered during previous actions. These descriptions may include relevant errors returned by the simulation in action execution, any unforeseen blocking condition encountered, or if the action returned by the LLM was not part of the list of candidate actions provided; and (8) Finally, the prompt again emphasizes the necessity of selecting actions from the provided list and asks the LLM output its response in a specified output format using the sentence `provide your selected action in the format 'format '$$ <selected action> $$`. This format string makes it easier to extract the selected action from the output.

These ordered components were selected based on extensive trial-and-error testing. While a goal of this work is to show that these prompt components are sufficient for effective action selection, demonstrating the necessity or optimality of these (or any) prompt components is beyond the scope of our current investigation.

V. Evaluation

In this section, we present an evaluation of our sequential action selection approach, comparing its performance against various LLM and traditional task-planning alternatives.

A. Implementation Details

Our experimental implementation is programmed in Python. Experiments were run on an Asus Vivobook laptop with a 2.4 GHz Intel Core i7 processor and 16GB RAM. While our selection engine approach posed in §IV can with any off-the-shelf LLM, our current implementation is integrated with OpenAI LLMs. Because our approach needs to make fast, real-time decisions, our evaluation uses the efficient GPT-3.5 Turbo model [1].

B. Experimental Testbed

Our experimental testbed is set within the AI2Thor simulation environment [13], designed to simulate realistic home scenarios for AI agents.

The experimental testbed is structured as follows:

(1) An experimental condition (i.e., some action selection approach) is provided a goal literal. The process of trying to achieve a goal will be called a "task". The approach selects a sequence of actions, trying to achieve the provided goal within the simulation environment. Each approach only interfaces with the simulator through action selection, all other rendering and object manipulation are handled by the simulator. We record information and metrics along the way. If the goal is not reached within 100 actions, this trial is considered a failure for the given approach; (2) Step 1 is repeated for 50 trials, with information and metrics recorded for all trials; (3) Steps 1 – 2 are repeated for four tasks, each characterized by their own goal literal (outlined below); and (4) Steps 1 – 3 are repeated for five action selection conditions (outlined below).

C. Baseline Comparisons

We compare our approach (Blocking-conditions and Resolutions Action Selection, abbreviated as BCR) to two baselines: (1) ProgPrompt, a few-shot Large Language Model (LLM) planner [16]; and (2) FF-Replan [18]. We chose these comparisons because they represent two common yet disparate strategies for discrete action selection.

For both ProgPrompt and FF-Replan, we consider two conditions each. We use a publicly available implementation of ProgPrompt.[2] The algorithm presented by Singh et al. [16] includes an action `find ?obj`, where `?obj` is some stand-in variable for an object in the environment. As the name implies, this action results in the robot automatically finding a certain object in the environment. The two conditions for ProgPrompt are (1) the exact version proposed by [16] (referred to as *ProgPrompt*; and (2) a version that excludes the `find ?obj` action (referred to as *ProgPrompt-no-find*). This distinction allows us to assess ProgPrompt's performance under optimal conditions as well as in scenarios, akin to our approach, where this automatic locating of objects is absent and objects must be located manually using actions like `rotate_right` or `rotate_left`.

We also use a publicly available implementation of FF-Replan.[3] The conditions are (1) the standard version presented by Yoon et al. [18] (referred to as *FF-Replan*), and (2) a version that removes all room location information for relevant goal objects before attempting to generate plans (referred to as *FF-Replan-limited*). Similar to the case above, this distinction allows us to assess FF-Replan's performance under optimal conditions as well as in scenarios, akin to our approach, where the planner has to locate pertinent objects on the fly.

D. Experimental Tasks

Each action selection condition (outlined below) was tasked with executing four distinct tasks within the experimental testbed:

1) *Making Coffee* (abbreviated as *Coffee*): involves locating a mug and a coffee maker and turning the coffee maker on while the mug is in the output area.
2) *Putting the Apple in the Fridge* (abbreviated as *Apple*): Involves the robot locating the apple and fridge, and storing an apple in the fridge.
3) *Washing the Mug in the Sink* (abbreviated as *Mug*): Involves the robot locating the mug and sink, putting the mug in the sink, and turning the faucet on.
4) *Making Toast* (abbreviated as *Toast*): involves locating a loaf of bread and a toaster, slicing the bread, and toasting it in the toaster.

The tasks above are specified by goal literals, and provided to each condition at the beginning of each trial.

For all tasks, necessary components are factored ahead of time to be compatible with each condition in §V-C. For instance, actions are manually associated with blocking conditions and effects for our approach, actions are manually associated with preconditions and effects for FF-Replan, and literals are reformatted into Python function templates for ProgPrompt. All possible actions are also associated with their own AI2Thor wrapper function. AI2Thor then knows how to execute each of these wrapper functions at run-time to correctly update the simulation environment.

Importantly, certain actions within these tasks are subject to points of failure. For example, the "navigate to object" action does not take into account the optimal point of interaction with the object, so if the robot navigates to the fridge and then opens it, the fridge door may block the robot's view not allowing it to interact with objects in the fridge. Thus, a key aspect of our evaluation is assessing the resilience of various action selection approaches in the face of such obstacles or uncertainties.

E. Evaluation Metrics

We report on three metrics in our evaluation: (1) success rate, the number of trials out of 50 total trials where the

[2] https://github.com/NVlabs/progprompt-vh

[3] https://fai.cs.uni-saarland.de/hoffmann/cff.html

Table 1: Success Rate

	Coffee	Apple	Mug	Toast
BCR (ours)	50/50	39/50	39/50	49/50
ProgPrompt	19/50	40/50	18/50	0/50
FF-Replan	14/50	13/50	11/50	0/50
ProgPrompt-no-find	14/50	12/50	0/50	0/50
FF-Replan-limited	0/50	0/50	0/50	0/50

Table 2: Mean Runtime per Trial

	Coffee	Apple	Mug	Toast
BCR (ours)	10.6 ± 16	46.8 ± 50	46.4 ± 44	11.6 ± 16
ProgPrompt	20.3 ± 6.7	8.77 ± 8.4	73.8 ± 75	24.1 ± 3.1
FF-Replan	0.52 ± 0.3	0.97 ± 0.4	6.90 ± 5	2.74 ± 5.6
ProgPrompt-no-find	20.2 ± 5.5	20.7 ± 9.2	117 ± 59	27.9 ± 41
FF-Replan-limited	4.23 ± 0.55	7.81 ± 1.7	8.8 ± 2.6	5.4 ± 1.8

(*measured in seconds and range values are standard deviation*)

approach was able to reach the goal; (2) The average runtime per trial; and (3) the average number of actions considered per action selection decision (not applicable for ProgPrompt).

F. Results

As shown in Table 1, our approach has a higher success rate than the alternative conditions. Even when ProgPrompt and FF-Replan are provided additional information or enhanced actions not available to our approach, we see that our approach still has a comparable or higher success rate. Moreover, as demonstrated in Table 2, the runtime of our approach is comparable to ProgPrompt, although it is not as fast as FF-Replan. Each decision in our approach takes about one second, which is certainly fast enough to make one-at-a-time sequential decisions in real time.

Additionally, Table 3 shows that our approach achieves these results while considering many fewer actions than the FF-Replan conditions. This suggests that our approach is more data efficient, helping to mitigate the exponential scaling of action branches typically exhibited by task planners.

VI. Discussion

In this work, we have presented a strategy that frames the action selection problem for robots in terms of resolving blocking conditions. This strategy allows a robot to make one-at-a-time decisions that take context and adapt and react to current situations. We also presented the first implementation of this strategy that uses a natural language processing model (LLM) as the selection engine. Our tests in a simulated environment show that our approach is often more effective than alternative methods, leading to a higher success rate in completing various tasks.

A. Limitations and Future Work

We note several limitations of our work that suggest future extensions and investigations. First, the LLM selection engine approach in §IV and prompt components in §IV-B reflect just one possible instance of the strategy specified in §III. Our current work does not claim that any of these choices are "best", only sufficient within the context of our evaluation. We will continue to explore this broad design space going forward, assessing the necessity and optimality of these components.

Table 3: Mean Number of Actions Considered per Decision

	Coffee	Apple	Mug	Toast
BCR (ours)	11.2 ± 7.6	12.0 ± 4.5	13.6 ± 4.5	11.0 ± 5.4
FF-Replan	49 ± 56	117 ± 118	139 ± 135	135 ± 117
FF-Replan-lim.	661 ± 74	649 ± 112	721 ± 213	669 ± 106

(*range values are standard deviation*)

The results in §V suggest that our approach scales well to tens of candidate actions (i.e., < 100) at a time. However, the LLM would likely get overwhelmed with decisions on the order of hundreds or thousands of candidate actions. Additional investigation is needed to characterize and address these possible scaling challenges.

Our approach outlined in §IV exhibits several occasional errors. For instance, even when directly instructed to only select an action from the set of candidate actions, the LLM may still hallucinate and select an action from outside of this set. Our current implementation detects this error and simply re-sends the prompt with an additional note urging the LLM to carefully read this instruction, which often fixes this issue. Also, it is possible for the LLM to get caught in a loop of repeated actions. Many of the unsuccessful trials in our evaluation were due to this issue, wherein our approach reached the 100-action maximum. We speculate that this issue may be mitigated by maintaining continuous interaction with the LLM instead of starting a fresh interaction at every decision. We plan to directly address this challenge in future work.

Our approach currently requires all actions to be manually populated with blocking conditions and resolutions. While this process is analogous to manually specifying preconditions and effects for standard task planners, the process may still be tedious. We plan to investigate methods to automatically generate these connections between actions, blocking conditions, and resolutions, both with offline preprocessing as well as creative online inference. A promising future direction is to incorporate Visual Language Models (VLMs) capable of detecting and adapting to previously unknown blocking modes encountered during runtime.

B. LLMs for Sequential Task Reasoning

Our work demonstrates a promising application of LLMs. We show that, if used in a particular way, even LLM models that are not the largest or newest at the time of writing (e.g., GPT-3.5) can enable effective sequential discrete action selection. This observation is in contrast to other works that suggest that even more modern LLM models (e.g., GPT-4) still struggle with sequential task reasoning [5].

We suggest the following hypothesis: *current LLM models may struggle with robustly formulating a sequence of actions to reach a goal all in one output. However, these models may excel at making one-at-a-time action selection decisions to reach a specified goal if given choices and enough context at each decision point*. While our current work does not provide enough evidence to prove this point, our findings offer preliminary evidence suggesting some validity of this phenomenon. We suggest this hypothesis as an exciting avenue of research going forward.

ACKNOWLEDGMENT

The authors thank Rachel Holladay for interesting and insightful conversations regarding this work.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances, 2022.

[3] Piergiorgio Bertoli, Alessandro Cimatti, et al. Improving heuristics for planning as search in belief space. In *AIPS*, pages 143–152, 2002.

[4] Blai Bonet and Hector Geffner. Planning with incomplete information as heuristic search in belief space. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 52–61, 2000.

[5] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[6] Georgia Chalvatzaki, Ali Younes, Daljeet Nandha, An Le, Leonardo F. R. Ribeiro, and Iryna Gurevych. Learning to reason over scene graphs: A case study of finetuning gpt-2 into a robot language model for grounded task planning, 2023.

[7] Yan Ding, Xiaohan Zhang, Saeid Amiri, Nieqing Cao, Hao Yang, Chad Esselink, and Shiqi Zhang. Robot task planning and situation handling in open worlds, 2022.

[8] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[9] Malte Helmert. The fast downward planning system. In *Journal of Artificial Intelligence Research*, pages 191–246, 2006.

[10] Jörg Hoffmann and Ronen I Brafman. Contingent planning via heuristic forward search with implicit belief states. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 15, pages 71–80, 2005.

[11] Jörg Hoffmann and Bernhard Nebel. Ff: The fast-forward planning system. *AI Magazine*, 22(3):57, 2001.

[12] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023.

[13] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, et al. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*, 2017.

[14] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+p: Empowering large language models with optimal planning proficiency, 2023.

[15] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl - the planning domain definition language. In *AIPS98 Planning Committee*. Yale Center for Computational Vision and Control, 1998.

[16] 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, 2022.

[17] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M. Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models, 2023.

[18] Sungwook Yoon, Alan Fern, and Robert Givan. Ff-replan: A baseline for probabilistic planning. *Artificial Intelligence*, 14:255–279, 2007.

[19] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

APPENDIX

A. Example LLM Prompt

Here, we provide a full example prompt dispatched to the LLM in our approach. (Note that tags like `Cabinet|-01.14|+00.39|+03.52` are object IDs for instances in AI2Thor).

```
['role': 'system', 'content': 'You are
   helping me select my next action,
   take your time and verify that the
   action you select is part of the list
   I provide. Take your time and go step
   by step.','role': 'user', 'content':
   'I am in the kitchen. I am not
   holding anything.','role': 'user',
   'content': 'The action scanroom if
   available allows me to visually scan
   the room I am currently in to see if
```

```
an object is visible. All actions
require the target object to be
visible other than scanroom.’,’role’:
’user’, ’content’: ’Some objects may
be contained within others that need
to be opened and wont be found by
scanroom.’,’role’: ’user’, ’content’:
"select the the best action from this
list: [’$$ turnleft character $$’,
’$$ turnright character $$’, ’$$
moveforward $$’, ’$$ movebackward
$$’, ’$$ turnaround $$’, ’$$ lookup
$$’, ’$$ lookdown $$’, ’$$
walk_to_room kitchen $$’, ’$$ open
Cabinet|-01.14|+00.39|+03.52 $$’, ’$$
walk_to_object
Drawer|-02.11|+00.71|+03.66 $$’, ’$$
walk_to_object
Cabinet|-02.38|+02.01|+01.69 $$’, ’$$
walk_to_object
Cabinet|-01.98|+01.78|+03.73 $$’, ’$$
walk_to_object
Fridge|-02.64|+00.00|+02.13 $$’, ’$$
walk_to_object
Cabinet|-02.38|+00.64|+01.68 $$’, ’$$
walk_to_object
Cabinet|-02.38|+02.01|+01.06 $$’, ’$$
walk_to_object
Cabinet|-01.96|+01.93|+03.73 $$’, ’$$
walk_to_object
Cabinet|-02.95|+01.93|+03.73 $$’, ’$$
walk_to_object
Cabinet|-02.43|+00.44|+03.42 $$’, ’$$
walk_to_object
Cabinet|-02.38|+01.69|+02.57 $$’, ’$$
walk_to_object
Cabinet|-02.43|+00.44|+03.17 $$’, ’$$
walk_to_object
Cabinet|-02.38|+02.01|+02.56 $$’, ’$$
walk_to_object
Cabinet|-01.99|+00.39|+03.51 $$’, ’$$
walk_to_object
Cabinet|-02.38|+02.01|+03.18 $$’, ’$$
walk_to_object
Cabinet|-02.38|+01.93|+01.69 $$’, ’$$
walk_to_object
Cabinet|-02.38|+01.41|+01.68 $$’, ’$$
walk_to_object
Cabinet|-02.62|+01.93|+03.73 $$’, ’$$
walk_to_object
Cabinet|-01.97|+00.39|+03.52 $$’, ’$$
scanroom Apple|-00.74|+00.77|+00.99
kitchen $$’, ’$$ turnleft character
$$’, ’$$ turnright character $$’, ’$$
moveforward $$’, ’$$ movebackward
$$’, ’$$ turnaround $$’, ’$$ lookup
$$’, ’$$ lookdown $$’] that is most
    likely to help me achieve my goal:
    [’INSIDE Apple|-00.74|+00.77|+00.99
    Fridge|-02.64|+00.00|+02.13’]?
    include an explaination for your
    action selection, only use the object
    name in your consideration ie
    ’<objectname>|#|#|#’. Please refrain
    from getting stuck in action loops
    and provide your selected action in
    the format ’format ’$$ <selected
    action> $$.",’role’: ’user’,
    ’content’: ’Please only select
    actions in the list I provided.’]
```

B. Example LLM Output

Here, we provide a full example output returned by the LLM in our approach:

```
$$walk_to_objectFridge|-02.64|+00.00|+02.
    13$$.ThisactionwillhelpyoureachtheFri
    dge,whichiswheretheAppleislocatedbase
    dontheinformationyouprovided.
```

C. Example Action Sequence

Here, we provide a full sequence of actions generated by our approach for the task “putting the apple in the fridge”:

```
[’walk_to_objectFridge|-02.64|+00.00|+02.
    13’,’openFridge|-02.64|+00.00|+02.13’
    ,’scanroomApple|-00.74|+00.77|+00.99k
    itchen’,’grabApple|-00.74|+00.77|+00.
    99’,’putinApple|-00.74|+00.77|+00.99F
    ridge|-02.64|+00.00|+02.13’,’putApple
    |-00.74|+00.77|+00.99DiningTable|-00.
    92|00.00|+01.20’,’openFridge|-02.64|+
    00.00|+02.13’,’walk_to_objectCabinet|-
    02.38|+02.01|+01.06’,’openCabinet|-02
    .38|+02.01|+01.06’,’scanroomEgg|-02.5
    3|+01.18|+02.22kitchen’,’grabEgg|-02.
    53|+01.18|+02.22’,’putEgg|-02.53|+01.
    18|+02.22Fridge|-02.64|+00.00|+02.13’
    ,’walk_to_roomkitchen’,’openCabinet|-0
    2.38|+01.69|+02.57’,’scanroomApple|-0
    0.74|+00.77|+00.99kitchen’,’grabApple
    |-00.74|+00.77|+00.99’,’walk_to_object
    Fridge|-02.64|+00.00|+02.13’,’walk_to_
    objectCabinet|+00.58|+00.39|+02.36’,’
    walk_to_roomkitchen’,’openCabinet|+00.
    58|+00.39|+02.36’,’moveforward’,’move
    backward’,’openDrawer|+00.73|+00.70|+
    02.20’,’scanroomFridge|-02.64|+00.00|
    +02.13kitchen’,’walk_to_objectFridge|-
    02.64|+00.00|+02.13’,’putinApple|-00.
    74|+00.77|+00.99Fridge|-02.64|+00.00|
    +02.13’]
```

(Note that actions like `grabEgg` *and* `placeEgg` *here are resolution actions needed because these objects are in the way, hindering progress on the task at hand)*