Stepwise Large-Scale Multi-Agent Task Planning Using Neighborhood Search

Fan Zeng, Shouhei Shirafuji, Changxiang Fan, Masahiro Nishio, and Jun Ota, *Member, IEEE*

***Abstract*—This letter presents a novel stepwise multi-agent task planning method that incorporates neighborhood search to address large-scale problems, thereby reducing computation time. With an increasing number of agents, the search space for task planning expands exponentially. Hence, conventional methods aiming to find globally optimal solutions, especially for some large-scale problems, incur extremely high computational costs and may even fail. In this letter, the proposed method easily achieves the goals of multi-agent task planning by solving an initial problem using a minimal number of agents. Subsequently, tasks are reallocated among all agents based on this solution and the solutions are iteratively optimized using a neighborhood search. While aiming to find a near-optimal solution rather than an optimal one, the method substantially reduces the time complexity of searching to a polynomial level. Moreover, the effectiveness of the proposed method is demonstrated by solving some benchmark problems and comparing the results obtained using the proposed method with those obtained using other state-of-the-art methods.**

***Index Terms*—Heuristic search, multi-agent system, neighborhood search, plangraph, task planning.**

I. Introduction

MULTI-AGENT task planning aims to obtain a solution that can achieve goals using a group of agents by maximizing the overall performance of a system. This is a fundamental problem common in various fields, including robotics, transportation, logistics, and manufacturing. In terms of domain description, the STRIPS-style language [1] is often utilized because it allows the world to be described using predicates, which are statements that can be either true or false. Although this language facilitates scaling up to solve highly complex problems by adding more states and actions, the search space for finding solutions grows exponentially with the number of predicates. Bylander [2] reported that such planning problems are PSPACE-complete and more difficult to solve compared to the NP-complete problems.

In recent decades, remarkable progress has been made in developing algorithms to solve multi-agent temporal task planning problems. Researchers [3], [4] have extended earlier methods, or integrated with motion planning [5] to solve multi-agent temporal planning problems. However, in large-scale problems, the number of instances and the search space grow exponentially, resulting in substantial time and memory related complexities, particularly when aiming to find optimal solutions. To solve this problem, pruning techniques [6] have been proposed to eliminate "bad" actions during a heuristic search. Hierarchical Task Network (HTN) planning [7] decomposes tasks into subtasks, reducing the search space by grouping similar tasks and addressing them simultaneously. Moreover, fast heuristic functions such as Fast Forward (FF) [8] and Fast Downward (FD) [9] have been developed and adopted by some widely used planners such as Partial-Order Planning Forward (POPF2) [10], Optimizing Preferences and time-dependent Costs (OPTIC) [11], Simultaneous Temporal Planning (STP) [12], Temporal Fast Downward (TFD) [13], and satisfiability modulo theories based planning (SMTPlan+) [14]. Although these methods contribute to speeding up the search to some extent, their performance in solving large-scale problems remains unsatisfactory. Some articles proposed methods to solve specific large-scale problems including underwater missions [15], warehouses [16], logistics [17], but these methods are highly dependent on the cases and can not be extended to a wide range of problems. Recently, some researchers applied learning methods to solving large-scale problems, like learning to plan [18], [19], Large Language Models [20], [21], and so on. However, they require a huge amount of datasets to be prepared in advance and extremely high computation costs, or will be poorly portable for different planning problems.

In this study, we present a stepwise method for solving multi-agent task planning problems in large-scale STRIPS-style problems described by the Planning Domain Definition Language (PDDL) [22] within a realistic time frame. While existing planners can promptly solve problems containing only a small number of agents, addressing large-scale problems efficiently remains a challenge. Our method solves this problem by initially achieving the goals of the given problem using a minimum

Manuscript received 7 June 2023; accepted 2 November 2023. Date of publication 10 November 2023; date of current version 21 November 2023. This letter was recommended for publication by Associate Editor F. Ju and Editor C.-B. Yan upon evaluation of the reviewers' comments. *(Corresponding author: Fan Zeng.)*

Fan Zeng is with the Department of Precision Engineering, Graduate School of Engineering, University of Tokyo, Tokyo 113-8656, Japan (e-mail: zeng@race.t.u-tokyo.ac.jp).

Shouhei Shirafuji is with the Department of Mechanical Engineering, Faculty of Engineering Science, Kansai University, Osaka 565-0842, Japan (e-mail: srfj@kansai-u.ac.jp).

Changxiang Fan is with the Institute of Facility Agriculture, Guangdong Academy of Agricultural Sciences, Guangzhou 510640, China (e-mail: fanchangxiang@gdaas.cn).

Masahiro Nishio is with the Strategic R&D Planning Department, R&D and Engineering Management Division, Advanced R&D and Engineering Company, Toyota Motor Corporation, Toyota 471-0826, Japan (e-mail: mnishio@mail.toyota.co.jp).

Jun Ota is with the Research into Artifacts, Center for Engineering (RACE), School of Engineering, University of Tokyo, Tokyo 113-8656, Japan (e-mail: ota@race.t.u-tokyo.ac.jp).

Digital Object Identifier 10.1109/LRA.2023.3331900

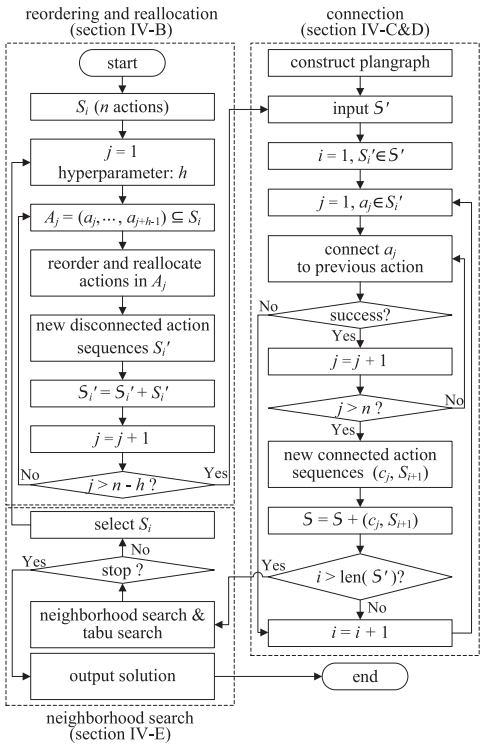

Fig. 1. Overview of the proposed task planning method.

number of agents and then iteratively refines the solution through reordering and partially reallocating actions to other agents. During the local refining process, the reordering and reallocating may disrupt the original logical connections between adjacent actions, so our proposed method reconnects them by searching for optimal connections using a plangraph. The time complexity of obtaining a new solution using the connection of adjacent actions is linearly related to the length of the solution, which reduces the complexity to a polynomial level. The pursuit of realizing an optimal solution is abandoned since developing scalable and quick algorithms to realize optimality is not plausible. The refining process adopts a neighborhood search approach, treating reallocated and reordered solutions as neighbors. Additionally, tabu search is employed to iteratively escape the local optimal solution and ultimately obtain a near-optimal solution.

The remaining sections of this letter are organized as follows: Section II provides an overview of the proposed methods. Section III discusses the modeling of the task planning problem using multi-agent PDDL and a plangraph. Section IV explains the refinement of the solution through task reallocation. To demonstrate the advantages of our method, we present an application using benchmark problems and provide a comparison in Section V. Finally, Section VI concludes this letter.

II. Overview

Fig. 1 provides an overview of our method. To solve a large-scale multi-agent problem, we obtain an initial solution (denoted as S_i) that achieves the same goals using minimum possible agents at first, it is an action sequence and consists of n actions. We select h (a predefined hyperparameter) consecutive actions from the beginning of S_i. By reordering these h actions and reallocating them to all agents, we generate new combinations of these selected h actions. Moreover, actions in each new combination are reallocated to all agents by partially replacing their arguments. Using the actions before and after the selected h actions in S_i, we can create multiple new disconnected action sequences denoted as S_i'. These sequences lack logical connections between adjacent actions owing to the reordering and reallocation process. We repeat this process by selecting the next h consecutive actions in S_i and repeat the above steps until we reach the end of S_i. The resulting set of action sequences is denoted as $\mathcal{S}' = (S_1', S_2', \ldots, S_i', \ldots)$, encompassing all possible task reallocations among all agents. The next problem to be solved is connecting the adjacent actions in S_i'.

The adjacent actions are connected sequentially using a plangraph. We construct the plangraph with mutually exclusive (mutex) pairs in each layer. We attempt to find a connection between actions $a_j (1 \leq j \leq n)$ to their previous actions in the action sequence S_i'. If a connection is not found, we skip the current S_i' and move on to the next S_{i+1}'. The connection scheme found within each action sequence is optimal. We transform the disconnected action sequence S_i' into a connected action sequence S_{i+1}. Next, we repeat this process until we have attempted to find connections among all the new action sequences in $\mathcal{S}'$, resulting in multiple logically connected action sequences $\mathcal{S}$ that are also local optimal solutions and are considered as neighbors. Finally, we refine the solution using neighborhood search and tabu search to iteratively escape local optimal solutions. This process continues until the stopping criteria are met, ultimately yielding a near-optimal solution.

III. Problem Description

A. Multi-Agent PDDL

This study adopts the PDDL to describe the multi-agent planning problem. The problem is represented via a five-tuple $\mathcal{T} = (\mathcal{O}, \mathcal{P}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, where the purpose of solving $\mathcal{T}$ is finding a solution S.

- $\mathcal{O}$ represents a set of objects, including agents and other instances in the working environment.
- $\mathcal{P}$ is a finite set of predicates defined with parameter $params(p) \subseteq \mathcal{O}$ for $p \in \mathcal{P}$. Predicates describe the relationships among different instances. This set includes static predicates $\mathcal{P}_\mathrm{s} \subseteq \mathcal{P}$ that retain their truth values throughout the planning procedure and dynamic predicates $\mathcal{P}_\mathrm{d} \subseteq \mathcal{P}$, whose truth values can be changed during the planning procedure.
- $\mathcal{A}$ is a finite set of actions with arguments. These actions are used to change a state $s \subseteq \mathcal{P}$. An action $a \in \mathcal{A}$ is defined by a tuple $(params, pre^+, pre^-, e^+, e^-, d)$, where $params(a) \subseteq \mathcal{O}$ represents the parameters of action a, $pre^+(a) \subseteq \mathcal{P}$ and $pre^-(a) \subseteq \mathcal{P}$ are positive preconditions and negative preconditions that are conjunctions of predicates describing what predicates must be true in a state before action a can be executed, $e^+(a) \subseteq \mathcal{P}$ and $e^-(a) \subseteq \mathcal{P}$ represent the add and delete effects, respectively, which

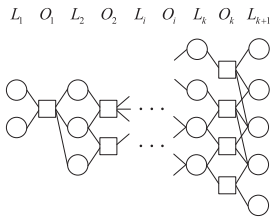

Fig. 2. A simple example of plangraph.

are conjunctions of predicates describing how the state changes when the action a is executed, and $dur(a) \in \mathbb{R}^+$ represents the duration of action a. The total time cost that evaluates the performance of the solution can be calculated by summing the durations of individual actions, considering some concurrent events. The execution of action a in state s can generates a state s' such that $s' = s \setminus e^-(a) \cup e^+(a)$.

- $\mathcal{I} \subseteq \mathcal{P}$ denotes the initial state of $\mathcal{T}$.
- $\mathcal{G} \subseteq \mathcal{P}$ denotes the goal state that the solution must achieve in $\mathcal{T}$.
- S is an action sequence $(a_1, a_2, \dots, a_i, \dots)$ that can transfer the state from $\mathcal{I}$ to $\mathcal{G}$, where $a_i \in \mathcal{A}$.

Here, we assume that agents included in $\mathcal{O}$ are homogeneous, implying that an agent can be in the same state and perform the same action as other agents, and $o \in \mathcal{O}$ is equally included in the parameter of predicates and actions.

B. Plangraph

This study converts a given multi-agent PDDL problem to a plangraph representation for effective local search and heuristic calculation. Unlike some methods such as UNPOP [23] or HSP [24] develop heuristics based on the assumption that all the actions are independent, which makes them less suitable for solving problems involving concurrent actions such as multi-agent planning problems. By contrast, the plangraph [25] is adept at handling such scenarios by providing concurrent information regarding actions and predicates. This letter adept the plangraph [25] to handle such scenarios by providing concurrent actions and predicates.

Plangraph classifies a predicate in a state $p \in \mathcal{P}$ as a literal and a predicate not included in a state $\neg p$ where $p \in \mathcal{P}$ as a negated literal. For a plangraph with k layers, like Fig. 2, the nodes are represented by $\mathcal{N} = (L_1, O_1, L_2, O_2, \dots, L_i, O_i, \dots, L_k, O_k, L_{k+1})$, where O_i is the set of actions whose preconditions appear in L_i. O_i also includes trivial actions whose precondition and effect are the same single predicate. L_{i+1} consists of the effects of all actions in O_i and all predicates in L_i, L_1 and L_{k+1} include all the initial state and goal states of the planning problem.

In the i-th literal layer L_i, the set M_{L_i} includes mutex pairs $\{p_1, p_2\}, \{p_1, \neg p_2\}, \{\neg p_1, \neg p_2\} \in M_{L_i}$ for two dynamic predicates p_1 and p_2. These pairs are the predicates that cannot be realized simultaneously in the i-th layer. Similarly, in the i-th operator layer O_i, the mutex pair $\{a_1, a_2\} \in M_{O_i}$ for two actions a_1 and a_2 cannot be executed simultaneously in that layer. If predicates and actions are not mutex pairs in one layer, they cannot be mutex pairs in the subsequent layer. This advantageous property allows us to list every mutex pair beforehand in polynomial time, which is the decisive advantage of the plangraph.

IV. Multi-Agent Task Planning via Reallocation

This section presents an efficient method for realizing multi-agent task planning using an initial plan with respect to a given PDDL problem. Prior to initiating the planning process, two key specifications need to be specified: the hyperparameter h and object sets $\mathcal{O}_\mathrm{a} \subseteq \mathcal{O}$ and $\mathcal{O}_\mathrm{f} \subseteq \mathcal{O}$. The hyperparameter h is the number of actions that are locally reallocated to escape from the local optimum in refinement. A large h value can deviate more from the local optimum but increase computational time, an intuitive relationship between them can be found in Fig. 4. h must be tuned carefully based on the specific problem at hand. $\mathcal{O}_\mathrm{a}$ represents the set of objects that correspond to agents involved in the planning process. In the subsequent subsection, the method for obtaining the initial solution by reducing these objects is presented. $\mathcal{O}_\mathrm{f}$ represents the set of objects that remain fixed during the reallocation process, generally it includes agent-independent arguments. The selection of reallocated arguments in addition to $\mathcal{O}_\mathrm{f}$ is a heuristic process aimed at reducing computational time.

A. Initialization

As finding optimal or near-optimal solutions of large-scale problems involving multiple agents within a reasonable time limit can be challenging due to the expansive search space, our method initially solves a simplified version of the problem with a single agent to achieve the given goals. The simplified problem, denoted as $\mathcal{T}_1$, is derived from the original problem $\mathcal{T}$ by omitting all agents except for a randomly chosen single agent. $\mathcal{T}_1$ is defined as $\mathcal{T}_1 = (\mathcal{O}_1, \mathcal{P}_1, \mathcal{A}_1, \mathcal{I} \cap \mathcal{P}_1, \mathcal{G} \cap \mathcal{P}_1)$, where $\mathcal{O}_1$ represents the set of instances consisting only of the chosen agent, and $\mathcal{P}_1$ and $\mathcal{A}_1$ are the subsets of predicates and actions, respectively, that are relevant to the single-agent problem, they are $\mathcal{P}_1 = \{p \mid p \in \mathcal{P},\ params(p) \subseteq \mathcal{O}_1\}$ and $\mathcal{A}_1 = \{a \mid a \in \mathcal{A},\ params(a) \subseteq \mathcal{O}_1\}$. Having transformed the problem into a single-agent setting, we can apply state-of-the-art methods, as discussed in Section I, to solve $\mathcal{T}_1$. If no solution is obtained for $\mathcal{T}_1$, the method proceeds by gradually adding agents and defining new problems $\mathcal{T}_i$ until a solution is obtained.

Once a solution is obtained for problem $\mathcal{T}_i$, our method converts it into the initial solution for the original problem $\mathcal{T}$. This conversion is necessary because the original multi-agent problem may require additional actions to accommodate the presence of multiple agents. The procedure described in Section IV-C, which involves connecting actions, facilitates this conversion. Importantly, this conversion step typically incurs minimal computational overhead. Consequently, an initial solution, denoted as $S_0 = (a_1, a_2, \dots, a_n)$, is obtained for the original problem.

B. Local Reordering and Reallocation

A given solution S_i, such as the initial solution S_0 obtained in the previous section, is performed by fewer agents to achieve the desired goal state. Our method aims to improve the efficiency of the solution by reallocating these actions among different agents, through a process of reordering actions and replacing the arguments of certain actions.

Initially, by choosing h consecutive actions starting from the j-th action of S_i, the algorithm selects subsets $A_j = (a_j, a_{j+1}, \ldots, a_{j+h-1}) \subseteq S_i$, where $1 \leq j \leq n-h$. All possible permutations A'_j of A_j are generated through reordering using a permutation function σ, resulting in

$$A'_j = (a_{\sigma(j)}, a_{\sigma(j+1)}, \ldots, a_{\sigma(j+h-1)}). \tag{1}$$

Next, in addition to the fixed arguments defined by $\mathcal{O}_\mathrm{f}$, the other arguments will be replaced by options under the same instance type. For example, the action "*human pick an apple by righthand*" is represented as `[pick, human, apple, righthand]`, then arguments about agents like `human` and `righthand` can be replaced by other instances under same types, such as `human` is replaced by `robot`, `righthand` is replaced by `gripper`.

Let $a' \in \mathcal{A}$ be a possible reallocation for an action $a \in \mathcal{A}$, of which at least one argument is reallocated. The reordered set of actions after reallocation is denoted as

$$A''_j = (a'_{\sigma(j)}, a'_{\sigma(j+1)}, \ldots, a'_{\sigma(j+h-1)}). \tag{2}$$

An action a' is considered a possible reallocation of another action a if it satisfies the condition params $params(a) \cap \mathcal{O}_\mathrm{f} = params(a') \cap \mathcal{O}_\mathrm{f}$. The reordering and reallocation process replaces the original series of actions $S_i = (a_1, a_2, \ldots, a_n)$ with the new actions as $S'_i = (a_1, \ldots, a_{j-1}, a'_{\sigma(j)}, a'_{\sigma(j+1)}, \ldots, a'_{\sigma(j+h-1)}, a_{j+h}, \ldots, a_n)$.

Through the reordering and reallocation process of j from 1 to $n-h$, a set of action sequences $\mathcal{S}_i$ is obtained. However, the reordering and replacing process disrupt the original logical connections between actions, resulting in action sequences $S'_i \in \mathcal{S'}_i$ that lack logical connections. The subsequent step involves connecting these disjoint action sequences to form a feasible and complete solution.

C. Connection of Adjacent Actions

Our algorithm, as described in Algorithm 1, obtains a locally reordered and reallocated planning solution S_{i+1} from the action sequence without logical connection S'_i. The algorithm focuses on finding the best actions to connect adjacent actions based on the plangraph.

Algorithm 1 begins by starting from the first layer of the plangraph (line 1), indicating that the search level l is set to 1. It proceeds to logically connect the actions in the action sequence S'_i one by one (line 2). The open list denoted as $open$ and the close list denoted as $close$ are defined in line 3 and represent lists of states that have the possibility of being optimal and states that are excluded from the dynamic programming of A*, respectively. The dictionary structures S and C serve as mappings to associate states with their corresponding action sets and costs, respectively (line 4). Initially, the open list contains only the initial state s with a cost of zero (line 5). In the algorithm, the next step is to select the state s with the smallest cost from the open list $open$ as the current state and move it to the close list $close$ (lines 7–8). The cost denoted as c is obtained from the dictionary C (line 9). The condition $(pre^+(a_j) \subseteq s) \wedge (pre^-(a_j) \subseteq \mathcal{P} \setminus s)$ being true indicates that the action $a_j \in S'_i$ has a logical connection to the previous action a_{j-1} because the current state satisfies its preconditions (line 10). In this case, the algorithm proceeds to the next action a_{j+1}. The objective of A* is to find optimal actions that can connect to the target action, continuing until a state satisfying this condition is found or the open list $open$ becomes empty.

A* searches for states to connect the action a_j using the candidate actions set A and the mutex pairs M obtained from Algorithm 2 described in Section IV-D (line 12). If the candidate action set A is empty, the current state s cannot connect to the target action, and the algorithm skips to the next level of the search (lines 13–15). Each set $A_\lambda \in A$ contains actions that can change the preconditions of a_j or their mutex predicates (line

Algorithm 1: Connecting Procedure for Adjacent Actions.

Input: Action sequence without logical connections S'_i
Output: Connected action sequence (new solution) S_{i+1}

1: $S_{i+1} \leftarrow S'_i,\ s \leftarrow \mathcal{I},\ l = 1$
2: **for** $a_j \in S'_i = (a_1, a_2, \ldots, a_n)$ **do**
3: $open \leftarrow$ Heap, $close \leftarrow \emptyset$
4: $S[s] \leftarrow \emptyset,\ C[s] \leftarrow 0$
5: $open$.heappush$(s, C[s])$
6: **while** $open$ **do**
7: $s \leftarrow open$.heappop
8: $close \leftarrow close \cup \{s\}$
9: $c \leftarrow C[s]$
10: **if** $(pre^+(a_j) \subseteq s) \wedge (pre^-(a_j) \subseteq \mathcal{P} \setminus s)$ **then**
11: **break**
12: $A, M \leftarrow$ GetCandidates(s, a_j, l) (Algorithm 2)
13: **if** $A = \emptyset$ **then**
14: $l \leftarrow l + 1$
15: **continue**
16: **for** $A_\lambda = (a'_1, a'_2, \ldots, a'_m) \in A$ **do**
17: **for** a' **in** A_λ **do**
28: $s \leftarrow (s \setminus e^-(a')) \cup e^+(a')$
19: $c \leftarrow c +$ cost(a')
20: $d^+ = \{d(m) \mid m \in M,\ p \in e^+(a'),\ p \in m\}$
21: $d^- = \{d(m) \mid m \in M,\ p \in e^-(a'),\ \neg p \in m\}$
22: $D[a'] = d^+ \cup d^-$
23: $h \leftarrow c + H(D, A_\lambda)$
24: **if** $(s \notin close) \wedge (h \leq C[s])$ **then**
25: $S[s] \leftarrow A_\lambda = (a'_1, a'_2, \ldots, a'_m)$
26: $C[s] \leftarrow h$
27: $open$.heappush$(s, C[s])$
28: $A_\lambda = (a'_1, a'_2, \ldots, a'_m) \leftarrow S[s]$
29: $S_{i+1} \leftarrow (a_1, \ldots, a_{j-1}, a'_1, a'_2, \ldots, a'_m, a_j, \ldots, a_n)$
30: $l \leftarrow l + 1$
31: **return** S_{i+1}

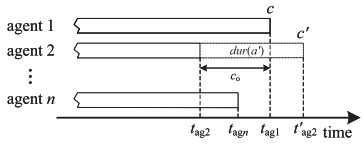

Fig. 3. Time cost calculation for concurrent actions.

16). An action $a' \in A_\lambda$ represents a possible change to one of the preconditions of a_j or its mutex predicate (line 17). These actions are applied to update the state, approaching a state that satisfies all the preconditions of a_j (line 18). The cost function $cost$ calculates the cost of executing these actions, and the variable c represents the total time cost after the action is applied (line 19). The cost $\text{cost}(a')$ to apply the action a' by agent $ag_k \in \mathcal{O}_\text{a}$, $(k = 1, 2, \ldots, n)$ is determined by considering the overlapping time c_o with actions already applied by different agents. Let $t_{\text{ag}_1}, t_{\text{ag}_2}, \ldots, t_{\text{ag}_n}$ represent the end times of actions already applied by different agents, respectively, as illustrated in Fig. 3.

In this scenario, the cost of applying action a' by agent ag_k considers the overlapping time c_o and is calculated as follows:

$$cost(a') = \begin{cases} dur(a') - c_\text{o} & (dur(a') - c_\text{o} > 0) \\ 0 & (dur(a') - c_\text{o} \leq 0) \end{cases}, \tag{3}$$

where c_o is the overwrapped time that is expressed as

$$c_\text{o} = \max(t_{\text{ag}_1}, t_{\text{ag}_2}, \ldots, t_{\text{ag}_n}) - t_{\text{ag}_\text{k}}. \tag{4}$$

The level distance d between the unsatisfied precondition predicate p_p and the blocking predicate p_s, which are mutex pairs in the plangraph, is used to represent the distance from the current state to the target precondition in the heuristics for executing A_λ. The heuristics estimates how much the chosen actions A_λ make the state closer to the target action's precondition. Algorithm 2 calculates the mutex pairs M consisting of pairs of unsatisfied preconditions p_p and blocking predicates p_s (line 12). The level distance d between these predicates is expressed as

$$d(\{p_\text{p}, p_\text{s}\}) = \|l_\text{p} - l_\text{s}\|, \tag{5}$$

where l_p is the lowest level where p_p becomes true or the lowest level where p_p becomes false in case of the negated predicate. Similarly, l_s is the lowest level where p_s becomes true or the lowest level where p_s becomes false in case of the negated predicate. An action that changes several mutex predicates and the distances it shortens are recorded in D, as shown in lines 20–22. Therefore, D represents the extent to which the state approximates the preconditions of a_j owing to the action.

Furthermore, the heuristics estimate the lowest cost corresponding to the chosen actions by many agents (this pertains to the parallel conduction of actions). First, for the level distances of an action $D[a']$, its smallest level distance implies the most effective way for the action to approximate the state to the target, $\min(D[a'])$. Considering an agent cannot execute more than one action simultaneously, we define the heuristics of part of action $A'_\lambda \subset A_\lambda$ performed by an agent ag_k as the accumulation of the above smallest level distance as follows:

$$H_{ag_k} = \sum_{a' \in A'_\lambda} \min(D[a']). \tag{6}$$

As a result, heuristics obtained for all ω agents in A_λ is

$$H_\text{ag} = (H_{\text{ag}_1}, H_{\text{ag}_2}, \ldots, H_{\text{ag}_\omega}) \tag{7}$$

The overall heuristics for A_λ not only relies on the largest heuristics of H_ag but also relies on the smaller heuristics. The following provides the resultant heuristics to choose actions that efficiently render the state close to the target state by the parallel execution of actions of many agents:

$$H = \max(H_\text{ag}) + \tanh\left(\frac{\text{sum}(H_\text{ag}) - \max(H_\text{ag})}{\text{len}(H_\text{ag}) - 1}\right). \tag{8}$$

Using heuristics H, the algorithm sums up the time cost c and heuristic cost H as the total cost h after applying A_λ for A* search (line 23).

If s is not sufficiently in $close$ and its total cost h is smaller than the optimal cost, the algorithm will update the state cost in C and actions set in S. Furthermore, the new state s and its cost c are also added to the open list $open$ (lines 24–27). After these procedures, if the open list $open$ becomes empty or the state satisfies the precondition of a_j, S_{i+1} is renewed by inserting the best action sequences A* found (lines 28–29) and l is increased by 1 (line 30). Finally, the solution S_{i+1} for local reordering and reallocation is returned (line 31).

Our proposed algorithm obtains a logically connected action sequence by connecting adjacent actions, while considering only the relevant candidate actions using the plangraph. The process of obtaining a new solution by connecting adjacent actions has a linear relationship with the length of the sequence, thereby reducing the complexity to a polynomial level. This feature makes our algorithm well-suited for solving task planning problems with a large number of instances and complex actions.

D. Action Candidates Extraction Based on Mutex

In line 12 of Algorithm 1, a function GetCandidates is used to find executable actions that affect unsatisfied preconditions or the predicates blocking these preconditions. The details of the GetCandidates function are shown in Algorithm 2. Let the negated predicates not included in the current state be $\bar{s}$, the unsatisfied positive preconditions of a_j be P^+, and the unsatisfied negative preconditions of a_j be P^-(lines 1–3).

First, action set A_p satisfying the unsatisfied preconditions P^+ and P^- and M including many pairs of unsatisfied preconditions and a predicate that blocks it are initialized as an empty set (line 4). All the candidate actions in A_p should be applicable in the current state, thus the searching actions for A_p should be within the actions set O'_l that contains actions whose preconditions $pre^+(a)$ and $pre^-(a)$ are satisfied in the current state s (line 5). Pertaining to the unsatisfied positive preconditions P^+, A_{p_p} containing candidate actions for P^+ is initialized by actions whose effects include predicates in P^+ directly (line 7). Furthermore, a mutex pair of the plangraph in the current level l shows the predicate pairs that cannot simultaneously appear

in the state. That implies a dynamic predicate p_{s}^{+} or p_{s}^{-} in the current state s, which is mutex with an unsatisfied precondition $p_{\mathrm{p}}^{+} \in P^{+}$, preventing the satisfaction of the preconditions of a_j. Therefore, the method checks all these combinations to determine whether they are mutex pairs. If yes, it adds candidate actions that include p_{s}^{+} or p_{s}^{-} in their effects to A_{p_p} (lines 8–10 and 12–14) because the action must change p_{s}^{+} or p_{s}^{-} to satisfy the precondition of a_j. Simultaneously, the algorithm retains the mutex pairs in M for the latter heuristics calculation (lines 11 and 15). Finally, the algorithm lists the actions that affect each predicate in the unsatisfied precondition as A_{p} in the line 16. In case of the unsatisfied negative preconditions P^{-}, similar procedures are executed (lines 17–27). Every combination of actions in which each action affects each unsatisfied predicate directly or through mutex pair is generated as A (line 28). Finally, the algorithm returns A and M for the calculation in Algorithm 1 (line 29).

Algorithm 2: Function to Obtain Actions to Change Predicates.

Input: Current state s, Target action to be connected a_j, Current plangraph level l,
Output: Candidate actions A, Mutex predicate pairs M

1: $\bar{s} \leftarrow \mathcal{P} \setminus s$
2: $P^{+} \leftarrow pre^{+}(a_j) - s$
3: $P^{-} \leftarrow pre^{-}(a_j) - \bar{s}$
4: $A_{\mathrm{p}} \leftarrow \emptyset,\ M \leftarrow \emptyset$
5: $O'_l \leftarrow \{a \mid a \in O_l,\ pre^{+}(a) \in s,\ pre^{-}(a) \in \bar{s}\}$
6: **for** $p_{\mathrm{p}} \in P^{+}$ **do**
7: $\quad A_{p_{\mathrm{p}}} \leftarrow \{a \mid a \in O'_l,\ p_{\mathrm{p}} \in e^{+}(a)\}$
8: $\quad$ **for** $p_{\mathrm{s}}^{+} \in s \cap \mathcal{P}_{\mathrm{d}}$ **do**
9: $\quad\quad$ **if** $\{p_{\mathrm{p}}, p_{\mathrm{s}}^{+}\} \in M_{L_l}$ **then**
10: $\quad\quad\quad A_{p_{\mathrm{p}}} \leftarrow A_{p_{\mathrm{p}}} \cap \{a \mid a \in O'_l,\ p_{\mathrm{s}}^{+} \in e^{-}(a)\}$
11: $\quad\quad\quad M \leftarrow M \cap \{\{p_{\mathrm{p}}, p_{\mathrm{s}}^{+}\}\}$
12: $\quad$ **for** $p_{\mathrm{s}}^{-} \in \bar{s} \cap \mathcal{P}_{\mathrm{d}}$ **do**
13: $\quad\quad$ **if** $\{p_{\mathrm{p}}, \neg p_{\mathrm{s}}^{-}\} \in M_{L_l}$ **then**
14: $\quad\quad\quad A_{p_{\mathrm{p}}} \leftarrow A_{p_{\mathrm{p}}} \cap \{a \mid a \in O'_l,\ p_{\mathrm{s}}^{-} \in e^{+}(a)\}$
15: $\quad\quad\quad M \leftarrow M \cap \{\{p_{\mathrm{p}}, \neg p_{\mathrm{s}}^{-}\}\}$
16: $\quad A_{\mathrm{p}} \leftarrow A_{\mathrm{p}} \cup \{A_{p_{\mathrm{p}}}\}$
17: **for** $p_{\mathrm{p}} \in P^{-}$ **do**
18: $\quad A_{p_{\mathrm{p}}} \leftarrow \{a \mid a \in O'_l,\ p_{\mathrm{p}} \in e^{-}(a)\}$
19: $\quad$ **for** $p_{\mathrm{s}}^{+} \in s \cap \mathcal{P}_{\mathrm{d}}$ **do**
20: $\quad\quad$ **if** $\{\neg p_{\mathrm{p}}, p_{\mathrm{s}}^{+}\} \in M_{L_l}$ **then**
21: $\quad\quad\quad A_{p_{\mathrm{p}}} \leftarrow A_{p_{\mathrm{p}}} \cap \{a \mid a \in O'_l,\ p_{\mathrm{s}}^{+} \in e^{-}(a)\}$
22: $\quad\quad\quad M \leftarrow M \cap \{\{\neg p_{\mathrm{p}}, p_{\mathrm{s}}^{+}\}\}$
23: $\quad$ **for** $p_{\mathrm{s}}^{-} \in \bar{s} \cap \mathcal{P}_{\mathrm{d}}$ **do**
24: $\quad\quad$ **if** $\{\neg p_{\mathrm{p}}, \neg p_{\mathrm{s}}^{-}\} \in M_{L_l}$ **then**
25: $\quad\quad\quad A_{p_{\mathrm{p}}} \leftarrow A_{p_{\mathrm{p}}} \cap \{a \mid a \in O'_l,\ p_{\mathrm{s}}^{-} \in e^{+}(a)\}$
26: $\quad\quad\quad M \leftarrow M \cap \{\{\neg p_{\mathrm{p}}, \neg p_{\mathrm{s}}^{-}\}\}$
27: $\quad A_{\mathrm{p}} \leftarrow A_{\mathrm{p}} \cup \{A_{p_{\mathrm{p}}}\}$
28: $A \leftarrow A_1 \times A_2 \times \cdots \times A_i \times \cdots \times A_n,\ A_i \in A_{\mathrm{p}}$
29: **return** $A,\ M$

E. Neighborhood Search for Solution With Multiple-Agents

After the process of locally reallocating and reordering actions followed by logically connecting these actions, we obtain a new solution S_{i+1}. We consider the new action sequence S_{i+1} as a neighbor of the solution S_i. For example, starting with the initial solution S_0 that involves a single agent, the above process generates a new solution S_1 where other agents contribute in performing the actions. The number of neighbors generated in this process is proportional to the length of the action sequence, resulting in a substantial number of neighbors for an initial solution. Therefore, efficient selection of neighbors is essential to find a better solution with multiple agents.

This type of problem can be categorized as a typical neighborhood search problem. We adopted a basic neighborhood search algorithm [26] with steepest descent and tabu search for selecting best neighbors. In each iteration, it selects the action sequence with the smallest cost as the initial solution and repeats this process until the stopping condition that no solution with a smaller cost can be found is met. In addition, it prevents the search algorithm from revisiting previously visited solutions during the connection process.

V. Experiment

In this section, we evaluate the effectiveness of our proposed method in terms of computation time and the obtained solution scores. For the sake of fairness, the proposed method is compared with 4 baseline algorithms that can effectively generate solutions for problems described by PDDL, including POPF2 [10], TFD [13], STP [12], and SMTPlan+ [14].

POPF2 combines the flexibility of partial-order planning with the fast forward search, TFD uses context-enhanced additive heuristics to efficiently guide the search toward the goals, STP can propagate constraints through the temporal network, SMTPlan+ allows us to perform modeling including more complex logic and constraints. These allow them to quickly solve problems, even in complex scenarios with intricate forms of concurrency. So they are used as baselines for comparison with our proposed approach called stepwise large-scale multi-agent task planning (SLMTP). Because the problems involve multiple simultaneous actions with different agents, we set the maximum number of simultaneous actions handled by STP to 3. All the result values were obtained by running the experiments on a computer installed with Ubuntu 20.04 and equipped with a 2.5 GHz AMD Epyc 7513 processor and 512 GB RAM. The computation time for each method was limited to 1000 seconds.

A. Experimental Setup

Our algorithm is specifically designed to address the task planning problem involving multiple agents on a large scale. To evaluate its effectiveness, we have chosen several benchmark problems that encompass multi-agent scenarios and simultaneous events. These problems include Depots, Storage, Floortile, and Zenotravel.

In the Depots problem, it is crucial to strike a balance between optimal routes and efficient loading strategies. Storage requires spatial reasoning to manage different storage areas effectively. Floortile involves the determination of the best path for painting robots, taking into account location constraints.

TABLE I
DESCRIPTION OF BENCHMARK PROBLEMS

Benchmark Problems	Actions(duration)	Initial Problem	Reallocation
DEPOTS	drive(10), lift(1), drop(1), load(3), unload(4)	3 depots, 3 distributors, 1 truck, 6 pallets, 6 crates, 6 hoists	load: truck, unload: truck
STORAGE	lift(2), drop(2), move(1), go-out(1), go-in(1)	18 store-areas, 1 hoist, 6 crates, 2 containers, 2 depots, 2 transit-areas	lift: hoist, drop: hoist
FLOORTILE	change-color(5), paint-up(2), paint-down(2), up(3), down(1), right(1), left(1)	20 tiles, 1 robot, 2 colors	change-color: robot, paint-up: robot, paint-down: robot
ZENOTRAVEL	board(20), debark(30), fly(180), zoom(100), refuel(73)	1 aircraft, 13 persons, 10 cities, 5 fuel-levels	board: aircraft, debark: aircraft, fly: aircraft, zoom: aircraft

TABLE II
COMPARISON BETWEEN OUR METHOD AND THE OTHER METHODS BASED ON SOLVING DIFFERENT BENCHMARK PROBLEMS

Problems	Initial Solution		SLMTP(1)		SLMTP(2)		SLMTP(3)		POPF2		STP		TFD	
	CPU Time	Score	CPU Time	Score	CPU Time	Score	CPU Time	Score	CPU Time	Score	CPU Time	Score	CPU Time	Score
DEPOTS(1)	1.17s	111	70.65s	101	76.99s	88	92.98s	69	424.51s	88	-	-	-	-
DEPOTS(2)	1.17s	111	134.04s	101	164.80s	84	274.74s	62	814.42s	68	-	-	-	-
DEPOTS(3)	1.17s	111	209.46s	101	279.55s	84	580.49s	62	-	-	-	-	-	-
STORAGE(1)	19.62s	39	45.73s	39	49.00s	31	72.29s	29	253.11s	31	-	-	175.23s	31
STORAGE(2)	19.62s	39	113.05s	39	141.81s	29	231.66s	25	557.75s	23	-	-	427.49s	23
STORAGE(3)	19.62s	39	202.92s	39	273.74s	29	620.45s	23	730.43s	25	-	-	991.77s	25
FLOORTILE(1)	13.97s	99	13.64s	77	18.67s	69	45.45s	60	633.04s	60	432.84s	71	12.91s	63
FLOORTILE(2)	13.97s	99	16.01s	77	33.88s	68	224.68s	48	-	-	-	-	-	-
FLOORTILE(3)	13.97s	99	22.91s	89	130.51s	68	743.72s	48	-	-	-	-	-	-
ZENOTRAVEL(1)	1.69s	3080	59.27s	3073	69.58s	3066	84.61s	2763	223.74s	2677	-	-	13.69s	3010
ZENOTRAVEL(2)	1.69s	3080	249.92s	3073	401.45s	2770	892.01s	2271	-	-	-	-	19.28s	2218
ZENOTRAVEL(3)	1.69s	3080	263.77s	3073	653.39s	2770	*942.47s	*2456	-	-	-	-	824.36s	1780

* It did not stop searching within the time limit.

ZENOTRAVEL represents a complex variation of the well-known TSP, where planes can fly at varying speeds with limited fuel. These problems pose considerable challenges due to the presence of multiple variables, constraints, and interdependencies that must be taken into consideration simultaneously. Furthermore, the search space of these benchmark problems can be vast and can be regarded as large-scale problems.

To select the arguments for reallocation, it should be first checked if the arguments are related to agents or not. For example, in DEPOTS, "truck" is the agent executing the action and is thus reallocated, while the other arguments such as "distributors" remain fixed. Table I provides the definition of actions and durations for various benchmark problems, as well as the instances included in the initial problem and the arguments for reallocation. The initial solutions for these benchmark domains were obtained using the POPF2 planner.

B. Results

The evaluation results presented in Table II demonstrate the effectiveness of the proposed method in terms of computation time (CPU time) and solution scores with respect to different benchmark problems. The scores represent the total time needed for all agents to reach their goals for the given problems, the smaller its value, the better. The initial solutions refer to the solutions obtained for the initial problems, "-" indicates that no solution can be obtained within 1000 seconds, the number after each benchmark problem name (in parenthesis) is the number of new agents added to the initial problem. In the case of column headings related to SLMTP, the number in parenthesis indicates the value of the hyperparameter h, which is equal to the selected actions as described in Section IV. SLMTP can stop searching within the described time limit, while the other methods cannot even after exceeding the time limit. SMTPlan+ can solve none of these problems, so we eliminated it from the results.

Results listed in Table II demonstrate that for a given problem, the time required to obtain a solution increases with the number of agents owing to the increased complexity of the problem. Additionally, increasing h in SLMTP increases the total time required to obtain the final solution, and improve the obtained results. This is because a larger h increases replaced actions in the initial solution, and leads to a large number of possible local reallocations, resulting in increased action sequences that need to be connected, but also at the same time increases the number of local optimal solutions. Overall, SLMTP can find solutions for all these benchmark problems within the time limit, unlike the other methods.

When adding one agent to the initial problems, SLMTP realizes a better than POPF2 with respect to DEPOTS and STORAGE, achieving a reduction of 60%-80% CPU time. For the FLOORTILE problem, SLMTP finds a solution with the same score as POPF2 but with a CPU time reduction of 90%. In the case of ZENOTRAVEL, SLMTP achieves a slightly higher score than POPF2 but with a substantial reduction of 60% in the CPU time required. This is because fuel level made the plangraph more complicated and the number of reallocations larger. STP can only find a solution for FLOORTILE, while TFD can find solutions for FLOORTILE and ZENOTRAVEL quickly owing to its fast heuristics; however, the scores of STP and TFD are worse than those of POPF2 and SLMTP. When two agents are added to the initial problems, POPF2 fails to provide solutions for FLOORTILE and ZENOTRAVEL, whereas SLMTP successfully provides solutions for all problems. In the case of DEPOTS, SLMTP outperforms POPF2 with a CPU time reduction of 60% and finds a better

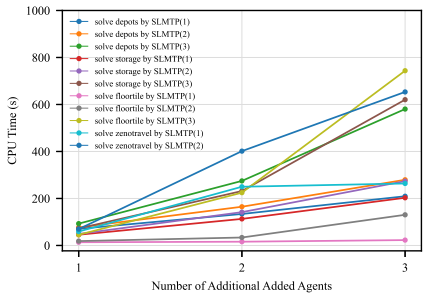

Fig. 4. Change of CPU time as the number of added agents increased with different h. Solve ZENOTRAVEL by SLMTP(3) is excluded as it did not stop searching within the time limit, making it meaningless to compare.

solution. In the case of STORAGE, SLMTP achieves a slightly higher score than POPF2 and TFD but consumes considerably lesser CPU time. When three agents are added to the initial problems, SLMTP continues to deliver near-optimal solutions for all problems, while POPF2 can only find a solution for STORAGE and TFD can only find solutions for FLOORTILE and ZENOTRAVEL.

Furthermore, as the number of agents increases, SLMTP's CPU time for these problems gradually increases at a polynomial level, which should have been $2^{n\times\Omega}$ (n is the number of new added agents, Ω is the average number of searches) times as the original one, as shown in Fig. 4. This is because the introduced adjacent actions connection algorithm transforms what used to be a long exhaustive search process into several easier sub-processes, reducing its time complexity from exponential to polynomial level. It is also evident from Fig. 4 that as h increases, CPU time also increases. These results demonstrate that SLMTP is capable of providing solutions to large-scale problems in a considerably short time. When a large number of agents are added, SLMTP takes a long time to find solutions; however, it still outperforms the other methods that struggle to find any solution at all.

VI. CONCLUSION

The letter proposes a stepwise multi-agent task planning method to solve large-scale problems. Instead of solving these large-scale problems directly, the proposed method begins by solving a simplified initial problem having a small number of agents. Based on the solution obtained by solving the initial problem, the proposed method generates multiple locally optimal solutions by reordering, reallocating, and connecting actions. To find a near-optimal solution, the proposed method combines neighborhood search with tabu search. This approach helps to refine the locally optimal solutions and improve their quality. The effectiveness of the proposed method is verified by solving four benchmark problems. The results show that the proposed method achieves near-optimal solutions, with a substantial reduction in computation cost (60%–80%) compared to the other state-of-the-art methods. The versatility of the proposed method can be increased if the method can be dynamically adjusted based on the environment change. This concept will be investigated in our future research.

REFERENCES

[1] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3/4, pp. 189–208, 1971.
[2] T. Bylander, "The computational complexity of propositional strips planning," *Artif. Intell.*, vol. 69, no. 1/2, pp. 165–204, 1994.
[3] Y. Carreno, R. P. Petrick, and Y. Petillot, "Multi-agent strategy for marine applications via temporal planning," in *Proc. IEEE 2nd Int. Conf. Artif. Intell. Knowl. Eng.*, 2019, pp. 243–250.
[4] A. Nikou, D. Boskos, J. Tumova, and D. V. Dimarogonas, "On the timed temporal logic planning of coupled multi-agent systems," *Automatica*, vol. 97, pp. 339–345, 2018.
[5] A. Messing, G. Neville, S. Chernova, S. Hutchinson, and H. Ravichandar, "GRSTAPS: Graphically recursive simultaneous task allocation, planning, and scheduling," *Int. J. Robot. Res.*, vol. 41, no. 2, pp. 232–256, 2022.
[6] M. Krajňanský, J. Hoffmann, O. Buffet, and A. Fern, "Learning pruning rules for heuristic search planning," in *Proc. 21st Eur. Conf. Artif. Intell.*,2014, pp. 483–488.
[7] B. Hayes and B. Scassellati, "Autonomously constructing hierarchical task networks for planning and human-robot collaboration," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2016, pp. 5469–5476.
[8] J. Hoffmann, "FF: The fast-forward planning system," *AI Mag.*, vol. 22, no. 3, pp. 57–57, 2001.
[9] M. Helmert, "The fast downward planning system," *J. Artif. Intell. Res.*, vol. 26, pp. 191–246, 2006.
[10] A. Coles, A. Coles, A. Clark, and S. Gilmore, "Cost-sensitive concurrent planning under duration uncertainty for service-level agreements," in *Proc. Int. Conf. Automated Plan. Scheduling*, 2011, pp. 34–41.
[11] J. Benton, A. Coles, and A. Coles, "Temporal planning with preferences and time-dependent continuous costs," in *Proc. 22nd Int. Conf. Automated Plan. Scheduling*, 2012, pp. 2–10.
[12] D. F. Blanco, A. Jonsson, H. L. P. Verdes, and S. Jiménez, "Forward-search temporal planning with simultaneous events," in *Proc. 13th Workshop Constraint Satisfaction Techn. Plan. Scheduling*, 2018, pp. 11–20.
[13] P. Eyerich, R. Mattmüller, and G. Röger, "Using the context-enhanced additive heuristic for temporal and numeric planning," in *Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*. Berlin, Germany: Springer, 2012, pp. 49–64.
[14] M. Cashmore, D. Magazzeni, and P. Zehtabi, "Planning for hybrid systems via satisfiability modulo theories," *J. Artif. Intell. Res.*, vol. 67, pp. 235–283, 2020.
[15] S. MahmoudZadeh, D. M. Powers, K. Sammut, and A. Yazdani, "Toward efficient task assignment and motion planning for large-scale underwater missions," *Int. J. Adv. Robot. Syst.*, vol. 13, no. 5, 2016, Art. no. 1729881416657974.
[16] D. Shi, Y. Tong, Z. Zhou, K. Xu, W. Tan, and H. Li, "Adaptive task planning for large-scale robotized warehouses," in *Proc. IEEE 38th Int. Conf. Data Eng.*, 2022, pp. 3327–3339.
[17] Z. Liu, H. Wei, H. Wang, H. Li, and H. Wang, "Integrated task allocation and path coordination for large-scale robot networks with uncertainties," *IEEE Trans. Automat. Sci. Eng.*, vol. 19, no. 4, pp. 2750–2761, Oct. 2021.
[18] C. Wang, D. Xu, and L. Fei-Fei, "Generalizable task planning through representation pretraining," *IEEE Robot. Autom. Lett.*, vol. 7, no. 3, pp. 8299–8306, Jul. 2022.
[19] A. Mordoch, D. Portnoy, R. Stern, and B. Juba, "Collaborative multi-agent planning with black-box agents by learning action models," in *Proc. Int. Conf. Automat. Plan. Schedul.*, 2022, pp. 56–64.
[20] T. Silver, V. Hariprasad, R. S. Shuttleworth, N. Kumar, T. Lozano-Pérez, and L. P. Kaelbling, "PDDL planning with pretrained large language models," in *Proc. Found. Models Decis. Mak. Workshop*, 2022, pp. 1–13.
[21] I. Singh et al., "ProgPrompt: Program generation for situated robot task planning using large language models," *Auton. Robots*, pp. 1–14, 2023.
[22] M. Fox and D. Long, "PDDL21: An extension to PDDL for expressing temporal planning domains," *J. Artif. Intell. Res.*, vol. 20, pp. 61–124, 2003.
[23] D. V. McDermott, "A heuristic estimator for means-ends analysis in planning," in *Proc. 3rd Int. Conf. Artif. Intell. Plan. Syst.*, 1996, pp. 142–149.
[24] B. Bonet and H. Geffner, "Planning as heuristic search: New results," in *Proc. 5th Eur. Conf. Plan. Recent Adv. AI Plan.*, 2000, pp. 360–372.
[25] D. Long and M. Fox, "Exploiting a graphplan framework in temporal planning," in *Proc. Int. Conf. Automated Plan. Scheduling*, 2003, pp. 51–62.
[26] R. F. Sproull, "Refinements to nearest-neighbor searching in K-dimensional trees," *Algorithmica*, vol. 6, pp. 579–589, 1991.