Appl Intell (2017) 47:914–934
DOI 10.1007/s10489-017-0936-x

LinGraph: a graph-based automated planner for concurrent task planning based on linear logic

Sıtar Kortik[1] · Uluç Saranlı[2]

Published online: 2 May 2017

Abstract In this paper, we introduce an automated planner for deterministic, concurrent domains, formulated as a graph-based theorem prover for a propositional fragment of intuitionistic linear logic, relying on the previously established connection between intuitionistic linear logic and planning problems. The new graph-based theorem prover we introduce improves planning performance by reducing proof permutations that are irrelevant to planning problems particularly in the presence of large numbers of objects and agents with identical properties (e.g. robots within a swarm, or parts in a large factory). We first present our graph-based automated planner, the Linear Logic Graph Planner (*LinGraph*). Subsequently we illustrate its application for planning within a concurrent manufacturing domain and provide comparisons with four existing automated planners, BlackBox, Symba-2, Metis and the Temporal Fast Downward (TFD), covering a wide range of state-of-the-art automated planning techniques and implementations. We show that even though LinGraph does not rely on any heuristics, it still outperforms these systems for concurrent domains with large numbers of identical objects and agents. These gains persist even when existing methods on symmetry reduction and numerical fluents are used, with LinGraph capable of handling problems with thousands of objects. Following these results, we also show that plan construction with LinGraph is equivalent to multiset rewriting systems, formally relating LinGraph to intuitionistic linear logic.

Keywords Automated planning · Linear logic · Assembly planning · Multiset rewriting

✉ Sıtar Kortik
sitar@cs.bilkent.edu.tr

Uluç Saranlı
saranli@ceng.metu.edu.tr

1 Department of Computer Engineering, Bilkent University, Ankara, Turkey

2 Department of Computer Engineering, Middle East Technical University, Ankara, Turkey

1 Introduction

Task planning refers to the problem of finding a sequence of *actions*, also called a *plan*, that takes an agent from a set of possible *initial states* to a particular *goal state*, using suitably chosen formalizations of all components. This framework assumes a discretized view of time, where physical actions are summarized by *pre-conditions* and *effects*, providing a discrete abstraction of system behavior in between [21, 53]. This discrete interface between continuous, "low-level" behaviors and their "high-level" compositions through discrete sequencing of actions has been at the basis of numerous languages and methods, including STRIPS and PDDL. In its general form, even for propositional representations of state, this classical view of task planning is PSPACE-complete [6]. Consequently, substantial literature is devoted to exploring practical restrictions to the general problem by reducing expressivity and providing heuristics for plan search.

In this context, our main contribution in this paper is the design, implementation and characterization of a graph-based method for automated generation of plans for deterministic, concurrent planning problems based on their encoding within a propositional fragment of Intuitionistic Linear Logic (ILL) [23]. In doing so, we rely on the previously established connection between AI planning and theorem proving for intuitionistic linear logic [8, 11, 30, 38], but introduce new ideas to achieve a practically feasible planner. Our automated planner, *LinGraph*, is particularly suitable for concurrent domains with large numbers of functionally identical objects, that present difficult challenges for existing planners due to the presence of irrelevant plan permutations and substantial concurrency. LinGraph is inspired from GraphPlan [3] and maps a planning problem into a graph encoding states, actions, goals and associated constraints, wherein plan search corresponds to finding a subgraph connecting all initial states to all goals. We formulate and perform this search as a Constraint Satisfaction Problem [32] on integer equalities.

Our approach is similar to the idea of using *bagged representations* [50] but supports an automated mechanism to identify object multiplicities and uses constraints to encode plan dependencies. It also bears similarities to symmetry reduction techniques [1, 15, 48] but does not impose any explicit symmetry constraints on initial and goal states and supports concurrency. Recent work on capturing object equivalence through numerical fluents is also closely related to our approach [20, 27]. Among important differences is LinGraph's connection to intuitionistic linear logic and its multiset rewriting semantics as a representational basis to eliminate the need to pre-specify all objects that the plan might need, as well as its automated exploitation of symmetries in problems with these characteristics. We illustrate these properties and characterize the performance of LinGraph through systematic comparisons with modern, state-of-the-art automated planners incorporating a wide range of techniques and show that it outperforms existing planners for domains featuring large numbers of functionally identical objects.

2 Related work

2.1 Automated planning systems

Automated planning has long been recognized as a central component in the synthesis of autonomous agents. In this paper, we focus on *classical planning*, which uses a deterministic and discrete *world model* of states and actions, seeking to find an appropriate ordering of actions to bring a system from an initial state to a desired final state [21]. In this context, *domain-independent* methods use separate specifications for both the domain and a specific problem given through a sufficiently expressive language [46]. STRIPS is commonly used for such specifications, succeeded by the more general PDDL language, used by the International Planning Competition (IPC) [10].

Variants of classical planning include *sequential* problems, wherein the planner seeks to find a totally ordered sequence of actions for a single agent, and *temporal* problems, wherein concurrent execution of actions need to be considered to optimize plan duration, or *make-span*. Earlier methods focusing on the latter set of problems include Partial-Order Planning (POP) [43] and its extensions, which formulated the search problem in *plan-space* rather than the state space to reduce the complexity of plan search. In contrast, GraphPlan [3] performs search in the state space, while encoding concurrent solutions as a layered graph.

These earlier approaches performed uninformed exploration of the state or plan spaces, limiting feasible problem sizes but ensuring optimality in either plan length for sequential problems, or make-span for concurrent problems. A more recent idea, used by successful planners such as the SatPlan and Blackbox [37], has been to encode planning problems as instances of boolean satisfiability (SAT) [36, 52], benefiting from existing, efficient algorithms for solving SAT problems.

On the other hand, many modern planners rely on heuristics to guide plan search. Substantial recent effort has been devoted to finding effective, efficient domain-independent and preferably admissible heuristics for optimality [4, 21]. In general, such heuristics rely on a relaxation of the original problem, allowing approximate estimation of remaining plan cost. Among these are additive and max heuristics [5, 56], as well as those relying on explicit computation of relaxed plans. For example, the Fast-Forward (FF) planner uses a GraphPlan-like structure on a relaxed problem to compute the heuristic, also using the concept of *helpful actions* [28]. More recently, planners such as the Fast Downward [24] and LAMA [49] systems improve on the performance of FF by relying on *landmarks* [29] among other improvements to guide the search. Other examples include the Madagascar planner [51], which sequentially extracts concurrent actions to find feasible plans.

Most of these heuristic methods, however, focus on sequential problems. Explicit consideration of concurrency during plan search requires the ability to explore sets of parallel actions in a single step, combinatorially increasing the branching factor for state-based search. Partial-order planning addresses this problem by switching to search in plan space but defining heuristics in this alternative formulation is much more challenging. A commonly used alternative is to parallelize an initially constructed sequential plan by considering ordering constraints between actions. However, this often impairs optimality, makes it difficult to discover and exploit symmetries within a planning problem and presents additional challenges associated with required parallelism as opposed to action commutativity [12]. Recent planners with temporal support include CPT [59], which uses Constraint Programming to reduce the search space and Temporal Fast-Downward [18, 25], which uses context-sensitive heuristics.

Intuitionistic linear logic was previously shown to be a suitable representational language for planning problems [30, 31, 33, 38]. Our work relies on these results, but proposes a new, graph-based algorithm for constructing proofs in a suitable fragment of intuitionistic linear logic that reduces the search space by eliminating irrelevantly permuted uses of identical resources within the planning problem. LinGraph structures its plan search as constructing successive "layers" of a graph, whose nodes represent multisets of linear logic atoms encoding objects and their states, combining equivalent instances of *functionally identical* objects and their states. Similar to the use of *numerical fluents* [20], LinGraph keeps track of node multiplicities as positive integers, imposing equality constraints to capture dependencies.

LinGraph's aggregation of functionally identical objects in a planning problem is also similar in spirit to bagged representations [50]. LinGraph, however, provides a fully automated mechanism to this end that can identify object equivalence even in intermediate states. Problems associated with such functionally equivalent but distinct objects in planning problems is also addressed by *symmetry reduction* techniques [1, 15, 19, 48]. These methods, however, focus primarily on sequential A* search. Our approach differs in its native support for concurrency, as well as its encoding of symmetries as part of the representation rather than as state equivalences. As we show in subsequent sections, this turns out to be very effective for problem domains wherein large numbers of functionally identical objects are manipulated, created and destroyed. As we show in our experimental results, even recent implementations of automated symmetry reduction techniques in modern planners such as Metis [1] do not achieve comparable reductions in the search space for these domains.

2.2 Automated reasoning and linear logic

The use of theorem proving for planning has not received much recent attention due to the computational complexity of proof search beyond the complexity of the planning problem itself. Reasons for this include the monotonicity of classical logic as well as the frame problem [39, 44]. In this context, *linear logic* [23] provides a strong semantic foundation to represent planning problems, addressing some of these problems with its interpretation of assumptions as consumable resources. A number of researchers have recognized these advantages of intuitionistic linear logic, exploring its connections to AI planning problems both in terms of expressivity [14, 30, 38] and concurrency [31]. Logic programming and answer sets have also shown the utility of nonmonotonic reasoning for planning [13, 16, 40, 45].

The use of linear logic for planning problems involves using simultaneous conjunction, linear implication and disjunction to respectively represent state, actions and nondeterministic effects [8, 42], with Horn fragments to address undecidability problems [33]. Certain inherent strengths of linear logic, such as its native support for concurrency have also been explored in the context of planning problems [31]. The suitability of using linear logic to represent and solve planning problems is also suggested by previous uses of Petri Nets for the same purpose [26, 55]. In this regard, our method has similarities to the PetriPlan system [55], which formulates a reachability query in an acyclic Petri Net and uses Integer Programming to find a solution. Nevertheless, practical deployment of these ideas for automated planners has been elusive due to the persistent difficulty of constructing efficient theorem provers for linear logic.

An alternative approach is to use model-based reasoning [9], wherein plan construction relies on searching through a semantic structure. In addition to SatPlan, the use of Linear Temporal Logic for planning is also among successful applications of this idea [2]. Unfortunately, strict reliance on temporal logic and model checking decreases expressivity and eliminates the possibility of deductive reasoning.

An interesting set of planning problems, for which LinGraph will be particularly well suited, is characterized by the presence of multiple, functionally identical objects [34]. Examples include job scheduling with multiple CPUs, coordination and task planning for swarm robots or parallel assembly lines. Even though linear logic can natively represent object multiplicity, existing theorem provers for it do not yet offer effective means for detecting such symmetries to be readily applicable for automated planning. In order to illustrate the advantages of LinGraph on such problems, we introduce a new manufacturing domain in Section 4, in which different types of manipulators with multiple functionally identical instances operate on a possibly large collections of components of different types. Our results in Section 6 compare the effectiveness of LinGraph on these concurrent domains to existing planners.

3 LinGraph: the language

In this section, we introduce the multiplicative exponential fragment of intuitionistic propositional linear logic we use as a basis for LinGraph, which we call the Linear Graph Planning Logic (LGPL).

Formally, LGPL syntax is defined by the grammar

$$\begin{array}{lll}
\text{Resource formulas:} & \mathcal{F}_R ::= & p \mid (p \otimes \mathcal{F}_R) \\
\text{Action formulas:} & \mathcal{F}_A ::= & (\mathcal{F}_R \multimap \mathcal{F}_R) \\
\text{Program formulas:} & \mathcal{F}_P ::= & !\mathcal{F}_A \mid \mathcal{F}_R \\
\text{Goal formulas:} & \mathcal{F}_G ::= & \mathcal{F}_R \mid (\mathcal{F}_P \multimap \mathcal{F}_G)
\end{array}$$

where p denotes atomic resources, $\mathcal{F}_R$ denotes single-use linear formulae representing states, $\mathcal{F}_A$ denotes implicative formulae representing actions, $\mathcal{F}_P$ captures program formulae that can appear as assumptions and $\mathcal{F}_G$ denotes goal formulae.

This grammar incorporates two multiplicative linear connectives, *simultaneous conjunction* $\otimes$ and *linear implication* $\multimap$, which allow simple but expressive modeling of dynamic components of the states. Even though additional linear connectives can increase expressivity, they are not necessary to express STRIPS problems. Consequently, for simplicity and efficiency, we focus on this small fragment of linear logic to develop LinGraph. The only additional component in LGPL other than the multiplicative connectives is the modal use of the "bang" operator, !, which recovers the possibility of using action definitions more than once.

For task planning, simultaneous conjunction ($\otimes$) is used to aggregate components of system state at a particular time instant, whereas linear implication ($\multimap$) is used to model actions, consuming precondition resources and creating postcondition (effect) resources. In LGPL, linear implication is also used within goal formulae to define initial states and actions by introducing them as assumptions. More specifically, the formula

$$\mathcal{F}_{P1} \multimap (\ldots \multimap (\mathcal{F}_{Pn} \multimap \mathcal{F}_G)\ldots) \tag{1}$$

corresponds to the the planning problem where resource and action formulae among $\mathcal{F}_{P1}$ through $\mathcal{F}_{Pn}$ respectively define the initial state and available actions.

4 A robotic assembly planning domain

Automated planner implementations are often characterized based on their ability to solve standardized planning problems [10]. These domains, however, focus on planning the actions of a single agent, generating either sequential or temporal plans without explicit consideration of object equivalence. LinGraph is not designed to outperform existing approaches in such small domains, but targets a less frequently studied class of problems with large numbers of functionally equivalent objects. In this section, we introduce a new class of planning domains that possess these properties to compare the performance of LinGraph to state-of-the-art planners.

The assembly planning domain we introduce consists of multiple types of *components* C_i, *manipulators* M_i and *products* P_i. Instances of each type of object are assumed to be identical in their functionality and properties despite being physically distinct. In this context, manipulators are assumed to operate on components to generate products as a result. An overview of these building blocks is given in Fig. 1.

Since a particular assembly problem will require different operational sequences on components and products, each problem will have its own set of actions. For example, a simple example domain might involve a single component type C, a single manipulator type M and a single product type P, with an action MakeP transforming a component C into a product P. Another, more complex example might have different types of components, products and manipulators and different actions for each manipulator. An important aspect of this domain will be the possibility of having a large number of these building blocks present in a particular scenario. For example, a large factory might have hundreds of manipulator robots, all of which considered as independent resources in constructing a plan. This aspect of our example domain will help illustrate distinguishing advantages of the LinGraph planner.

LGPL allows native encoding of STRIPS domains by using linear resources to represent components of the dynamic state and implicative formulas to represent actions [14]. Suppose, for example, that a robotic assembly problem has a single type of manipulator, M, that takes a component of type C and transforms it into a product of type P. This action can be represented by

$$\text{MakeP} := !(C \otimes M \multimap M \otimes P)\ . \tag{2}$$

Here, MakeP is the name of the action, and the implicative formula captures the requirement that a component C and a manipulator M are available, resulting in the production of a product P and keeping the manipulator M available for later use. Linearity of LGPL ensures that an instance of C is consumed and a new instance of P is generated. Unlike STRIPS, lack of monotonicity in LGPL requires that action formulas reintroduce resources that are required but are left unaffected.

The initial state for LGPL encodings of planning problems are given as a resource formula $\mathcal{F}_{R0}$. For example, the example above with three components and two manipulators can be encoded as

$$\mathcal{F}_{R0} := (C \otimes C \otimes C \otimes M \otimes M)\ . \tag{3}$$

Objects:

$C_1, \ldots, C_n$	:	Components of type 1 through n
$M_1, \ldots, M_m$	:	Manipulators of type 1 through m
$P_1, \ldots, P_o$	:	Products of type 1 through o

Fig. 1 Object types and notation for the robotic assembly planning domain

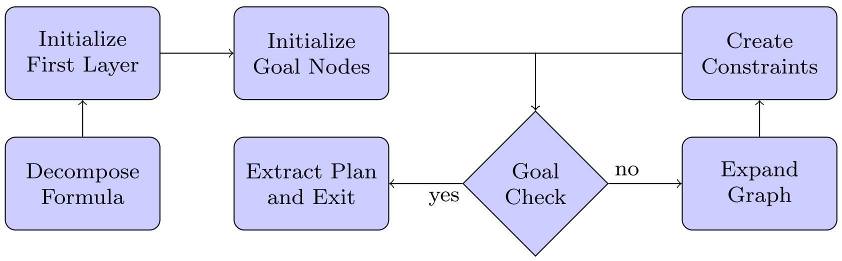

Fig. 2 Algorithm for constructing a LinGraph instance

For brevity, we will abbreviate multiple occurrences of resources with an "exponent" notation

$$(F_R)^n := \underbrace{(F_R \otimes ... \otimes F_R)}_{n \text{ times}} . \qquad (4)$$

The example initial state above now takes the form $\mathcal{F}_{R0} = (C^3 \otimes M^2)$. Such automated aggregation of identical resources is an important novelty of LinGraph.

The goal state for a problem can also be encoded as a resource formula in LGPL. For instance, a goal of producing three components would take the form

$$\mathcal{F}_{Gf} := (P^3 \otimes M^2) . \qquad (5)$$

Put together, the planning problem for this example corresponds to finding a proof for the goal formula

$$\mathcal{F}_G = !(C \otimes M \multimap P \otimes M) \multimap (C^3 \otimes M^2) \multimap (P^3 \otimes M^2) . \qquad (6)$$

Every proof of this formula corresponds to a valid plan, consisting of a partially ordered application of multiple instances of MakeP. This mapping between proofs and plans is in general not injective, with possibly multiple proofs corresponding to the same partial ordering of actions. In this paper, we focus on only the existence of plans, and leave proof equivalence for future work.

5 LinGraph: plan construction

5.1 Planning with LinGraph

LinGraph is structurally similar to GraphPlan [3] in that it incrementally constructs an approximate reachability graph consisting of alternating state and action layers. In LinGraph, state nodes encode *linear resources*, together with their multiplicities, guaranteed to be uniquely reachable from the initial states. LinGraph also keeps separate track of multiple nodes associated with the same proposition that possibly have different multiplicities and constraint structures. In this fashion, concurrent application of conflicting actions can be prevented through the use of *integer inequality constraints* that generalize the binary mutexes of GraphPlan. Overall, careful maintenance of nodes, their multiplicities and constraints enables LinGraph to find plans that minimize plan make-span, even in the presence of large numbers of functionally identical resources.

Unlike GraphPlan, LinGraph is closer to a forward chaining approach since even a partially constructed LinGraph captures all possible applications of actions. Consequently, once a propositional state layer that matches the goals and a valid solution for the cumulative set of constraints are found, plan extraction is guaranteed to succeed and an additional backchaining search is not required. This aspect of LinGraph is illustrated by the examples in Section 5.1 (see, for example, Fig. 8), and justified by its formal connection to multiset rewriting systems described in Section 7.

The overall structure of our method for constructing a LinGraph is shown in Fig. 2. The algorithm starts by initializing nodes in the first layer based on the initial plan state by decomposing the original LGPL formula into its linear assumptions (initial state), unrestricted assumptions (action definitions) and the goal formula (goal state). Subsequently, goal nodes are also initialized in preparation for the goal check that attempts to match goal states to nodes of the latest state layer. Our careful maintenance of node constraints ensures that a successful match yields a valid plan. If no match is found, the graph is extended by one layer.

In visualizing the construction of the LinGraph, we use the components illustrated in Fig. 3. In particular, state nodes are be shown with rounded rectangles, including the corresponding atomic resource type p, the count n, available number of resources with this type and a unique label s.

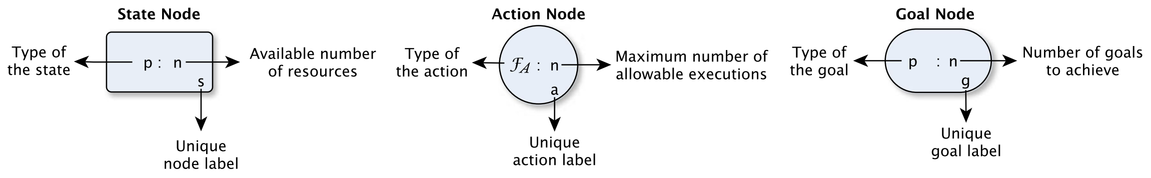

Fig. 3 LinGraph nodes for states, actions and goals

Nodes in action layers will be shown with circles, showing the corresponding implicative formula $\mathcal{F}_A$, the maximum number of times n that this action can be used and a unique action label a. Finally, goal nodes are shown with oval shapes, showing the corresponding atomic resource type p, the number of required instances n and a unique goal label g.

In the rest of the paper, we will use $n(s_i)$, $n(a_i)$ and $n(g_i)$ to denote the number of available instances for state, action and goal nodes, respectively. We will also use $lvl(s_i)$ and $lvl(a_i)$ to denote the graph level for state and action nodes, respectively. Finally, we will use $frm(s_i)$, $frm(a_i)$ and $frm(g_i)$ to denote formulae associated with state, action and goal nodes, respectively.

5.1.1 LinGraph constraints

LinGraph incorporates *constraints* on the number of instances that can be used for each state node, ensuring strict correspondence between plans extracted from LinGraph and associated LGPL proofs. Constraints in LinGraph take the form of integer linear equality and inequality conditions on the number of instances consumed from each state node, which we denote by overloading the unique labels s_i for state nodes, constrained to be integers in the range $0 \leq s_i \leq n(s_i)$. Based on this notation, LinGraph constraint C_j is formally defined as

$$C_j := \sum_{i=0}^{N} \alpha_j^i s_i = \beta_j \quad \text{or} \quad C_j := \sum_{i=0}^{N} \alpha_j^i s_i \geq \gamma_j$$

where N is the total number of state nodes, α_j^i are integer constraint coefficients and β_j, γ_j are constants. LinGraph constraints serve two purposes. First, when a match can be found between the desired goal state encoded by goal nodes, and the final reachable state encoded by state nodes in the last level of the LinGraph, constraints are used to ensure that a corresponding valid LGPL proof can be found without violating linearity properties of the underlying logic. The second use of LinGraph constraints accompanies graph expansion, wherein preconditions of available actions are matched against available state nodes in the last level. Constraints are used to prevent the creation of actions whose preconditions cannot be simultaneously realized, reducing the size of the LinGraph by. In both cases, we use an existing constraint solver, Minion, to find a feasible solution to a set of constraint equations [22], revealing the number of instances s_i consumed for all state nodes.

5.1.2 Decomposing the goal and initializing LinGraph

Consider the planning problem encoded by the formula

$$C^2 \multimap M^2 \multimap !(C \otimes M \multimap M \otimes P) \multimap (P^2 \otimes M^2)\ . \qquad (7)$$

Parsing of this goal formula allows the identification of linear resources $\Delta = \{C^2, M^2\}$, unrestricted actions $\Gamma = \{C \otimes M \multimap M \otimes P\}$ and the goal formula $\mathcal{F}_G = P^2 \otimes M^2$, leading to the LGPL sequent

$$(C \otimes M \multimap M \otimes P); (C^2, M^2) \Rightarrow P^2 \otimes M^2\ .$$

The final state is also decomposed into its atomic constituents P^2 and M^2 to initialize goal nodes.

Subsequently, initial state nodes s_i are created in the first state layer corresponding to each unique type of initial resource, with $n(s_i)$ initialized with the total number of such resources. For the example above, this results in two nodes, C and M, with two instances available for each as shown in Fig. 4. Similarly, individual goal nodes are created for each type of goal resource. In the above example, this results in two nodes, for types P and M, each having a count of two. Each new node is assigned a unique label.

LinGraph initialization is completed by adding an equality constraint for each initial state node, capturing the requirement that all initial resources must be completely consumed. For the example above, this results in two constraints, $s_1 = 2$ and $s_2 = 2$.

5.1.3 Goal check

Each incremental step in the construction of the LinGraph checks whether all desired goals can be satisfied with resources in the very last level, while also satisfying all previously recorded constraints. To this end, the goal check goes through each goal node g_i and finds all associated state nodes in the last level l_{max} of the LinGraph with matching formulae, defining the set

$$S[g_i] := \left\{ s_j \mid (lvl(s_j) = l_{max}) \wedge (frm(s_j) = frm(g_i)) \right\}.$$

For each goal, a corresponding constraint is defined, ensuring that an adequate number of states are found to satisfy the goal, taking the form

$$\left(\sum_{s_j \in S[g_i]} s_j \right) = n(g_i)\ .$$

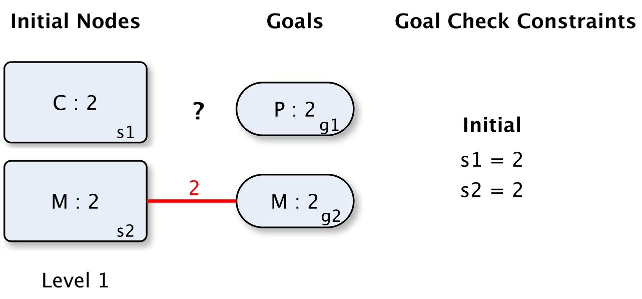

Fig. 4 LinGraph instance with initial state and goal nodes for the planning problem of (7). Initial constraints and the first failed goal check attempt are also shown

Recall that the symbol s_j is overloaded to both denote the label for the state node, as well as the number of times the corresponding resource is used in the final plan. Subsequently, a solver for these constraints together with all previously accumulated LinGraph constraints is invoked to find a feasible solution for the number of consumed instances s_i for all state nodes within the bounds $0 \leq s_i \leq n(s_i)$. By construction, constraints added during initialization and expansion ensure that this solution corresponds to a valid LGPL proof and hence a feasible plan (see Section 7).

For our example above, a match is found for g_2 with $S[g_2] = \{s_2\}$, resulting in the constraint $s_2 = n(g_2) = 2$. However, no matches are found for g_1, yielding $S[g_1] = \{\}$ corresponding to an unsatisfiable constraint $0 = n(g_1) = 2$. This requires the expansion of the LinGraph (see Section 5.1.4), followed by subsequent creation of additional constraints (see Section 5.1.5).

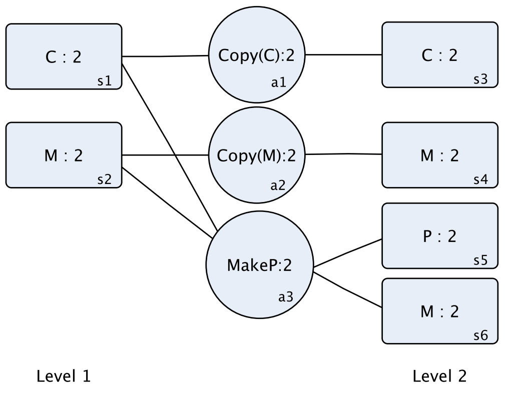

Fig. 5 LinGraph expansion example illustrating second level nodes created both through copying (s_3 and s_4) as well as the application of the MakeP action (s_5 and s_6)

5.1.4 LinGraph expansion: creating new nodes

The failure of the goal check step indicates that no feasible plans exist with the current number of graph levels and that plans with more steps should be explored. To this end, the expansion step creates a new LinGraph level, containing nodes for states reachable by actions consuming resources in the last level of the existing graph. In the rest of this section, the last LinGraph level before expansion will be denoted with k, with the expansion creating the level $(k+1)$.

In order to ensure that the new level includes complete information on all reachable states, all states in level k are copied to the level $(k+1)$ level through special "copy" actions. Formally, consider a node s_i in level k. A new copy action a_{new} is created for this state node with $frm(a_{new}) = (frm(s_i) \multimap frm(s_i))$ and $n(a_{new}) = n(s_i)$, connecting to a new state node s_{new} with $frm(s_{new}) = frm(s_i)$ and $n(s_{new}) = n(s_i)$. For convenience we will define the parameterized notation $\text{Copy}(x) := x \multimap x$ to denote formulas associated with copy actions in LinGraph illustrations throughout the rest of the paper, keeping in mind that these are not explicitly included as actions.

Figure 5 illustrates the expansion of the LinGraph for the example in Fig. 4 from the first level to the second level. Both s_1 and s_2 in the first level, having types C and M, are linked to two new copy actions a_1 and a_2, creating two new state nodes s_3 and s_4.

Once all state nodes are copied to the new level, the system considers possible applications of all available actions $F_A \in \Gamma$. Consider one such action

$$F_A := e_1^{c_1} \otimes \ldots \otimes e_m^{c_m} \multimap f_1^{d_1} \otimes \ldots \otimes f_n^{d_n}$$

with atomic preconditions and effects. For each such action, LinGraph expansion first attempts to match each precondition e_i to a corresponding state node s_{e_i} in level k, such that $lvl(s_{e_i}) = k$, $frm(s_{e_i}) = e_i$, and $n(s_{e_i}) \geq c_i$. If nodes satisfying these conditions are found for all preconditions, a new action node a_{new} is created with $n(a_{new}) = \min_{i=1}^{m}(\lfloor n(s_{e_i})/c_i \rfloor))$ and $frm(a_{new}) = F_A$. Subsequently, new state nodes are created in level $k+1$ with fresh labels $s_{new,j}$ such that $lvl(s_{new,j}) = k+1$, $lvl(s_{new,j}) = f_i$ and $n(s_{new,j}) = n(a_{new}) * d_j$ with $j = 1, \ldots, n$. Figure 5 illustrates the expansion of the LinGraph in Fig. 4.

5.1.5 LinGraph expansion: constraints

In addition to the *initial node constraints* of Section 5.1.2 and the *goal checking constraints* of Section 5.1.3, LinGraph makes use of *sibling constraints* to ensure that the effects of a single action are used in accordance with their cardinality in the action definition and *dependency constraints* to enforce limitations arising from shared preconditions between actions in the same level.

We first describe sibling constraints created for each newly created action with two or more different types of effects. Consider, for instance, an action

$$F_A := e_1^{c_1} \otimes \ldots \otimes e_m^{c_m} \multimap f_1^{d_1} \otimes \ldots \otimes f_n^{d_n}$$

created during the expansion step from level k to level $k+1$ with $n > 1$. Every instance of this action used within the final plan will create d_1 instances of the resource f_1, d_2 instances of the resource f_2 and so on, all of which must be consumed by subsequent actions. Assuming that state nodes s_1 through s_n are created in level $k+1$ after the expansion step, corresponding to the effects f_1 through f_n for this action, this requires that $s_1/d_1 = s_2/d_2 = \ldots = s_n/d_n$. Denoting the least common multiple of d_1 through d_n with

$lcm(d_1, ..., d_n)$, this expands into $n-1$ integer equality constraints

$$\begin{aligned} lcm(d_1, ..., d_n)s_1/d_1 &= lcm(d_1, ..., d_n)s_2/d_2 \\ &\dots \\ lcm(d_1, ..., d_n)s_{n-1}/d_{n-1} &= lcm(d_1, ..., d_n)s_n/d_n \end{aligned}$$

added to the current set of LinGraph constraints. For the example in Fig. 5, this results in the addition of the constraint $s_5 = s_6$ as shown in Fig. 6.

In contrast, dependency constraints are introduced following the expansion step from level k to level $k+1$ between effects of actions and their common preconditions in level k. Suppose that a state node s_{cm} with $frm(s_{cm}) = e_{cm}$ in level k appears as a precondition with different cardinalities to multiple actions (possibly including copy actions) $a_1, ..., a_j$, taking the form

$$\begin{aligned} frm(a_1) &= ... \otimes e_{cm}^{c_1} \otimes ... \multimap f_{1,1}^{d_{1,1}} \otimes ... \otimes f_{1,n_1}^{d_{1,n_1}} \\ &\dots \\ frm(a_j) &= ... \otimes e_{cm}^{c_j} \otimes ... \multimap f_{j,1}^{d_{j,1}} \otimes ... \otimes f_{j,n_j}^{d_{j,n_j}} \end{aligned}$$

The expansion step will then have created new state nodes $s_{r,t}$ in level $k+1$ corresponding to the effects $f_{r,t}$ of these actions with $r = 1, ..., j$ and $t = 1, ..., n_r$. However, there are only $n(s_{cm})$ instances available for the node s_{cm}, meaning that only a certain subset of these actions can be used in the final proof. A constraint must be introduced to ensure that these state nodes in level $k+1$ are used no more or less than what is allowed by the availability of the state node s_{cm} in level k. Since sibling constraints described above ensure consistency of $s_{r,1}$ through s_{r,n_r} in level $k+1$, it is sufficient to impose the dependency constraint on the first effect node $s_{r,1}$ for each action. This dependency constraint for s_{cm} hence takes the form $\sum_{r=1}^{j} c_r \frac{s_{r,1}}{d_{r,1}} = s_{cm}$ which can be transformed into the equality

$$\sum_{r=1}^{j} c_r \frac{lcm(d_{1,1}, ..., d_{j,1})}{d_{r,1}} s_{r,1} = lcm(d_{1,1}, ..., d_{j,1})s_{cm}$$

Satisfaction of this constraint ensures that the state node s_{cm} is used within its allowable limits.

In the example of Fig. 5, the state node s_1 is shared by a_1 and a_3, and the state node s_2 is shared by a_2 and a_3. These dependencies result in the creation of the constraints $s_3 + s_5 = s_1$ and $s_4 + s_5 = s_2$ as shown in the final LinGraph of Fig. 6.

Having created both the sibling and dependency constraints for newly created state nodes in level $k+1$, our algorithm proceeds to perform another goal check. To this end, new goal check constraints are created as $s_5 = 2$, $s_4 + s_6 = 2$ and $s_3 = 0$ as shown in Fig. 6. In this case, a valid solution is found as

$$s_1 = 2,\ s_2 = 2,\ s_3 = 0,\ s_4 = 0,\ s_5 = 2,\ s_6 = 2\ , \tag{8}$$

corresponding to a valid solution for the planning problem of (7).

5.1.6 LinGraph expansion: pruning actions

As described in Section 5.1.5, the expansion step exhaustively explores all possible matches for the preconditions of an action to state nodes in the last level of the LinGraph. This is one of the primary sources of computational complexity in the LinGraph planner, corresponding to the nondeterminism for action selection also inherent to proof search for linear logic. In this section, we introduce two mechanisms to prevent instantiating actions that are either redundant, or guaranteed to be inapplicable. These mechanisms reduce the size of the LinGraph, and substantially increase the performance of plan search. Suppose that an action

$$F_A := e_1^{c_1} \otimes ... \otimes e_m^{c_m} \multimap f_1^{d_1} \otimes ... \otimes f_n^{d_n}$$

is being considered for expansion and a particular set of matching nodes s_{e_i} have been identified in level k. If all preconditions, s_{e_i}, have been created as effects of *copy actions* in level $k-1$, we prevent the creation of a new action since

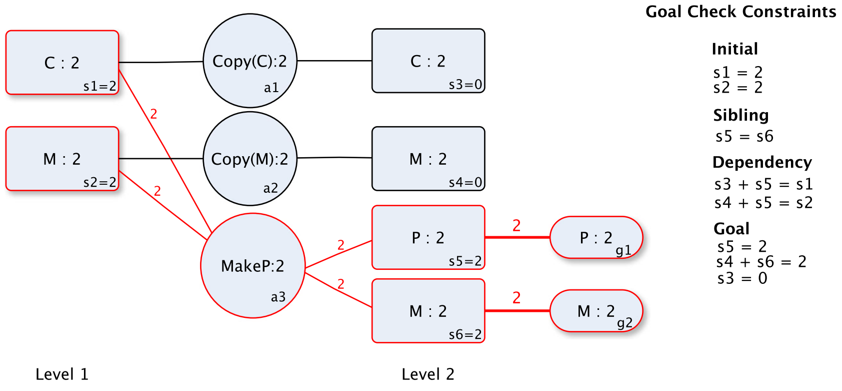

Fig. 6 Final state of the LinGraph for the planning problem of (7) with all goals satisfied

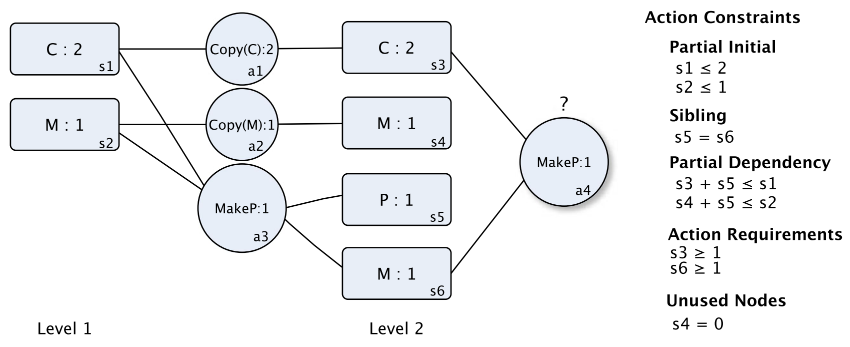

Fig. 7 LinGraph expansion step checking for feasibility of a new instance of the MakeP action through *action constraints*

all nodes s_{e_i} have, by construction, identical corresponding nodes in level k, which would have matched the preconditions of F_A, thereby creating its effects. Introducing another action node would be fully redundant, adding no new plan alternatives.

The second mechanism involves locally checking whether simultaneous use of all preconditions for a single application of an action is feasible under the current set of constraints. However, such a single, isolated action application will naturally result in a *partial consumption* of available resources. However, to preserve correspondence to the semantics of linear logic, initial, sibling and dependency constraints enforce all available resources to be entirely consumed, and hence do not allow such a partial check.

Fortunately, we can proceed by observing that checking for partial resource consumption can be implemented using *inequality constraints*. In particular, we will replace initial equality constraints of Section 5.1.2 with inequalities, placing an upper bound on the number of initial resources to be used. Similarly, dependency constraints that were previously equality conditions on state nodes in successive levels, are transformed into inequality constraints, placing a lower bound on resources to be used in level k, based on the needs of level $k+1$. Sibling constraints are kept as equality conditions. Finally, usage counts for nodes in level $k+1$ that are neither among the preconditions or for the action under consideration nor among their siblings are forced to be zero for efficiency.

To illustrate, consider the slightly modified example

$$C^2 \multimap M \multimap !(C \otimes M \multimap M \otimes P) \multimap (P^2 \otimes M)\ , \qquad (9)$$

wherein only a single manipulator is available and a successful plan requires a third level. Having only a single product available in the second level as shown in Fig. 7, the LinGraph algorithm proceeds with the next expansion stage, and considers the MakeP action, yielding the updated action constraints for this problem shown in Fig. 7. In this case, the resulting equality and inequality constraints are satisfiable, successfully instantiating the action node.

5.1.7 Plan extraction

LinGraph construction ends when the constraint solver finds a solution to the current set of constraints during the goal check. This solution identifies how many resources from each state node is used for the final plan. The final LinGraph for the example of (9) is shown in Fig. 8, where we have

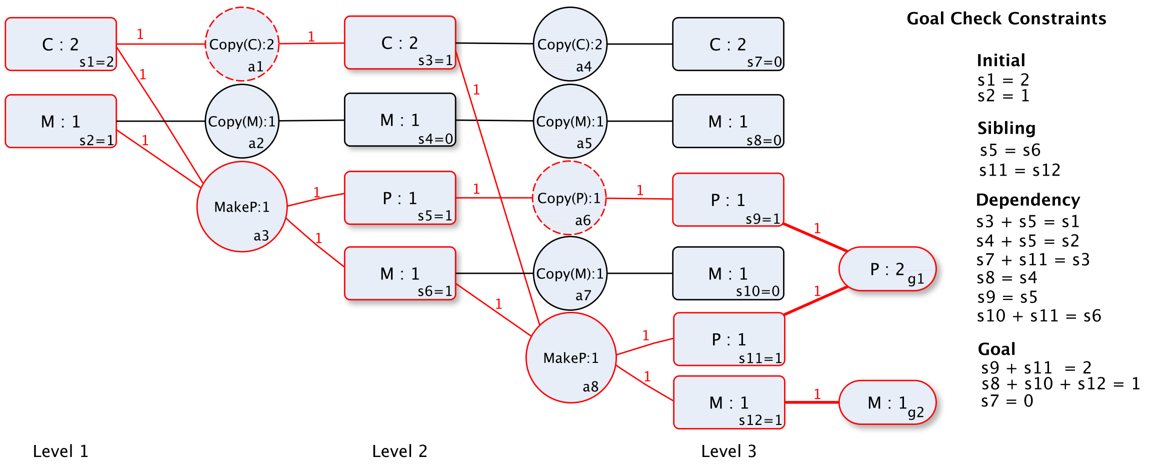

Fig. 8 Plan extraction using the solution obtained from the constraint solver. Each state node shows the number of utilized resources in the lower right corner

also included a specific solution to the constraints with the utilization counts s_i.

Plan extraction from LinGraph yields multisets of actions to be executed (possibly in parallel) for each level, ignoring copy actions. For example, the final plan for the graph of Fig. 8 consists of two successive applications of the MakeP action. Note, also, that multiple instances of the same action can be present in a particular level, capturing multiple concurrent operations on different instances of similar objects.

5.1.8 Algorithmic properties of LinGraph

The first component in the complexity of LinGraph comes from the space required to represent its graph structure. Given a LinGraph with i levels, and N_i nodes in its last level, a actions, with a maximum of e postconditions and a maximum of k different ways each action can be applied, the graph expansion phase requires the creation of $N_{i+1} \leq k * a * e * N_i$ new nodes. In the worst case, this means that the LinGraph size is exponential in the number of graph levels, with space complexity $O((kae)^i)$, compatible with the EXPSPACE-hard complexity of multiset rewriting [41]. In particularly problematic cases with many nodes having identical propositions, k may also be proportional to N_i, resulting in doubly exponential complexity. In general, however, object symmetry will reduce both e and k since identical propositions will be aggregated in the same node and our constraint based pruning in Section 5.1.6 will prevent infeasible actions from being considered.

In the face of the exponential space complexity of LinGraph, our examples focus on how performance scales with the number of functionally identical object instances, which is the axis along which our algorithm outperforms existing approaches. Naturally, as the number of graph levels (and hence plan length) increases, LinGraph in its current form becomes infeasible due to this space complexity. Nevertheless, two aspects of LinGraph help with the complexity along this axis. First and foremost, LinGraph aggregates identical propositions within the postconditions of an action into a single node, decreasing exponential buildup of space requirements for representing multisets that include identical propositions. Second, domains that encode physical planning problems, in general, include actions that transform physical resources, which inherently disallow an exponential buildup in the number of generated objects and resources. In the long term, the exponential space complexity of LinGraph can be improved on by combining multiple node instances with the same proposition into a single node without, but this requires a careful reconsideration of how node constraints are formulated to maintain semantic validity. We leave this extension for future work.

The second computationally expensive component in LinGraph is the goal check. In the worst case, with g goals, having maximum multiplicities n, each matching a maximum of k, mutually independent nodes in the last level, LinGraph will generate g constraints, each involving at most k variables with their right hand side equal to m. Assuming an uninformed, brute force constraint solver, this result in the consideration of $N_g = (k^m)^g$ combinatorial assignments to constraint variables. It is important to note that the complexity of the goal check is complementary to the complexity of graph construction in that, the worst case for graph size, with each node and goal having a multiplicity of one, corresponds to the best case for the goal check. In contrast, when node multiplicities increase, the graph size decreases but constraints have greater freedom with larger values of g and larger ranges for the variables s_i.

It is important to note that LinGraph does not yet use any heuristics. Therefore, its average performance on sequential problems is expected to be below existing planners that perform better pruning of the search space. As a result, we have limited the scope of this paper to present the basic ideas behind LinGraph and its ability to successfully and automatically recognize symmetries in a domain, and left extensions to incorporate heuristics and better search methods for future work. These extensions would require relaxing the layered structure of the graph, tighter integration with the constraint solver to reduce space requirements and work on finding heuristics in the presence of concurrency.

5.2 A more challenging assembly planning example

In this section, we use the LinGraph planner on a more complex problem instance within the domain of Section 4. Figure 9 shows the associated object types and actions, wherein two types of components are first transformed into two separate sub-products, which are then assembled in pairs into a single product. Four instances of these products are finally assembled into a final product (using the action MakeFP) to finish the process.

This example is structurally similar to our previous examples, but the presence of multiple intermediate products requires more levels for the solution. Linear logic

Objects:		
C_i, S_j	:	Component i and Sub-product j objects
M	:	Manipulator object
P, FP	:	Product and Final Product objects
Actions:		
MakeS$_i$	:	Manipulator produces S_i from C_i.
MakeP	:	Manipulator produces P from S_1 and S_2.
MakeFP	:	Manipulator produces FP from 4 of P.

Fig. 9 LGPL encodings for objects and actions for the planning example with a final product

$$\begin{array}{lcl} \mathrm{MakeS}_i & : & (C_i \otimes M) \multimap (S_i \otimes M) \\ \mathrm{MakeP} & : & (S_1 \otimes S_2 \otimes M) \multimap (P \otimes M) \\ \mathrm{MakeFP} & : & (P^4 \otimes M) \multimap (FP \otimes M) \end{array}$$

Fig. 10 Linear logical encodings of actions within the more complex assembly planning problem domain

formulae corresponding to the actions described in Fig. 9 are shown in Fig. 10.

We now describe a challenging example problem in this domain. Suppose that we initially have 32 components of type C_1, 32 components of type C_2 and 48 manipulators M. Using these components, we would like to produce 16 instances of product P and 4 instances of the product FP. This corresponds to the goal formula

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap !\mathrm{MakeFP} \multimap \left(C_1^{32} \otimes C_2^{32} \otimes M^{48}\right) \multimap (P^{16} \otimes FP^4 \otimes M^{48}) \quad (10)$$

The initial number of available components and manipulators were chosen to both illustrate parallel execution of actions, as well as the extension of the plan to several steps due to the relatively limited number of manipulators. The LinGraph generated for this example up to the level where a solution is found is partially illustrated in Fig. 11. LinGraph succeeds in finding a valid plan even for this complex example including large numbers of components and manipulators. The plan extracted from the LinGraph is shown in Fig. 12, with multiple actions executed in parallel within each step.

$$\begin{array}{ll} \text{Step 1:} & \mathrm{MakeS}_1^{32},\ \mathrm{MakeS}_2^{16} \\ \text{Step 2:} & \mathrm{MakeS}_2^{16},\ \mathrm{MakeP}^{16} \\ \text{Step 3:} & \mathrm{MakeP}^{16},\ \mathrm{MakeFP}^{4} \end{array}$$

Fig. 12 Solution to the assembly planning problem of (10)

6 Experimental results

In this section, we evaluate the performance of LinGraph on increasingly difficult, concurrent planning problems and show that it outperforms modern automated planners for problems with substantial symmetry, large numbers of functionally equivalent objects, as well as intermediate creation of objects throughout the plan. Currently, LinGraph does not use any heuristics to guide plan search. As a result, its performance on structurally simple, general planning problems, particularly sequential domains with long action sequences, cannot challenge existing planners. Even though we believe that relaxing the layered structure of LinGraph to support heuristic search and actions with non-unit durations will be possible to bring its performance on general planning problems to match modern planners for temporal problems, we have left these extensions for future work and focused the scope of the present paper to describe LinGraph's novel representation of object equivalence and its performance on domains with large numbers of functionally identical objects.

Our evaluation focuses on comparing LinGraph's execution time to several existing planners for specific problems with the characteristics described above. First, we consider

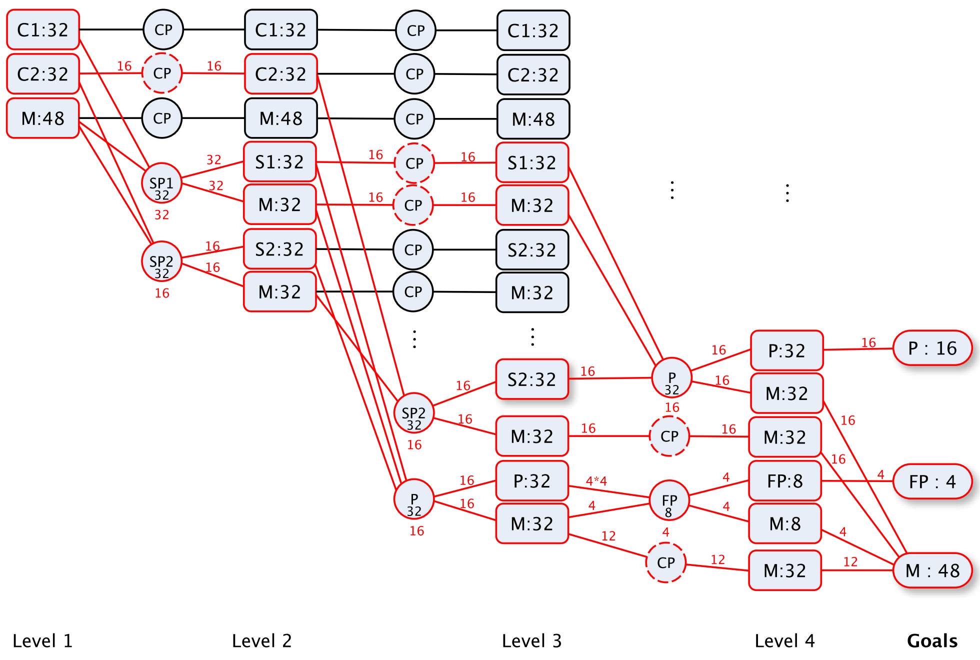

Fig. 11 Final LinGraph for (10). Some actions and nodes for levels larger than 2 were omitted for visual clarity

Blackbox, one of the earlier yet successful planners based on a combination of GraphPlan and SAT planning with support for concurrency and make-span optimality. Our comparisons also include Symba-2 [57], which is one of the recent high performance planners for sequential domains, having won the sequential optimal track for the 2014 IPC. Next, we consider the Metis planner [1] since it implements symmetry reduction techniques that are uniquely relevant to our approach. Due to their exclusive focus on sequential problems, these last two planners do not generate concurrent plans and hence can be considered to operate on easier instances of our domains. Finally, we include the Temporal Fast Downward (TFD) planner [17], which is among the best modern planners for temporal problems with explicit support for concurrency and make-span optimality, as well as numerical fluents.

Our prototype LinGraph planner was implemented in SML, without any explicit optimizations for execution efficiency. In contrast, the latest, optimized implementations were used for Blackbox, Symba-2, Metis and TFD. In all cases, experiments were performed on a 2.93GHz Intel Pentium Dual-Core CPU E6500 Processor and 2 GB of RAM.

All planning examples we describe in this section were fed to our LinGraph planner in the form of LGPL goal formulas. In contrast, Blackbox, Symba-2, Metis and TFD implementations were given problem inputs encoded in PDDL. Additional PDDL features we relied on for encoding example problems included type specifications and durative actions, as well as numerical fluents to explicitly encode object multiplicity. It is also important to remember that plans generated by the Symba-2 and Metis planners are sequential and fail to capture concurrency features of any domain.

6.1 Domain-1: assembly with two component types

The first example we investigate is an instance of the assembly planning domain described in Section 4, with two types of components, C_1 and C_2, first transformed into subproducts S_1 and S_2, which are then combined to make a product. Associated actions encoded in LGPL take the form

$$\begin{aligned} MakeS_i &: (C_i \otimes M) \multimap (S_i \otimes M) \\ MakeP &: (S_1 \otimes S_2 \otimes M) \multimap (P \otimes M) \end{aligned} \qquad (11)$$

The PDDL domain encoding of the same problem is shown in Fig. 13, wherein object types, available predicates and actions are defined together with a specification of required PDDL features. An important detail in this encoding is that the effects of each action include both $(manip\ ?m)$ and $(not(manip\ ?m))$, which was needed to ensure that the same manipulator is not used simultaneously within the same level. The need for such tricks in STRIPS encodings of planning problems are a byproduct of the informal semantics associated with PDDL, wherein implementation details for a particular planner may result in different semantics for the domain. Due to its semantic correspondence to LGPL formulae, LinGraph does not require such implementation specific details for its encodings.

```
(define (domain assembly-domain1)
        (:requirements :strips :typing)
 (:types COM1 COM2 SUB1 SUB2 MAN PRO)
 (:predicates (comp1 ?c1) (comp2 ?c2)
              (subprod1 ?s1) (subprod2 ?s2)
              (manip ?m) (prod ?p))
 (:action MakeS1
  :parameters (?c1 - COM1 ?m - MAN ?s1 - SUB1)
  :precondition (and (comp1 ?c1) (manip ?m) )
  :effect (and (subprod1 ?s1) (not (comp1 ?c1))
               (not (manip ?m)) (manip ?m) ))
 (:action MakeS2
  :parameters (?c2 - COM2 ?m - MAN ?s2 - SUB2)
  :precondition (and (comp2 ?c2) (manip ?m) )
  :effect (and (subprod2 ?s2) (not (comp2 ?c2))
               (not (manip ?m)) (manip ?m)))
 (:action MakeP
  :parameters (?s1 - SUB1 ?s2 - SUB2 ?m - MAN ?p - PRO)
  :precondition (and (subprod1 ?s1) (subprod2 ?s2)
                     (manip ?m))
  :effect (and (prod ?p) (not (subprod1 ?s1))
              (not (subprod2 ?s2))
              (not (manip ?m)) (manip ?m))))
```

Fig. 13 PDDL domain file for Domain-1 with two types of components

A specific problem instance requires the specification of initial and goal states. For instance, suppose four of each component are available and four final products are to be produced. The LGPL encoding takes the form

$$\begin{aligned} &!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \\ &\multimap \left(C_1^4 \otimes C_2^4 \otimes M^4\right) \multimap (P^4 \otimes M^4)\,. \end{aligned} \qquad (12)$$

In contrast, the PDDL encoding of the same example is given by the *problem file* shown in Fig. 14. An inconvenient requirement for the PDDL problem definition is the need to define *all* objects that may be needed throughout the plan, including any number of intermediate objects. However, it is often difficult to know these beforehand, highlighting another advantage of the LGPL encoding.

The PDDL encodings of Figs. 13 and 14 are appropriate for sequential planners. Temporal planners, including TFD, require support for *durative actions* as shown in Fig. 15.

```
(define (problem domain1-four-components)
(:domain assembly-domain1)
(:objects c11 c12 c13 c14 - COM1 c21 c22 c23 c24 - COM2
          s11 s12 s13 s14 - SUB1 s21 s22 s23 s24 - SUB2
          m1 m2 m3 m4 - MAN p1 p2 p3 p4 - PRO)
(:init (comp1 c11) (comp1 c12) (comp1 c13) (comp1 c14)
       (comp2 c21) (comp2 c22) (comp2 c23) (comp2 c24)
       (manip m1) (manip m2) (manip m3) (manip m4) )
(:goal (and (prod p1) (prod p2) (prod p3) (prod p4)
       (manip m1) (manip m2) (manip m3) (manip m4) )))
```

Fig. 14 PDDL encoding of a Domain-1 problem instance with four components

```
(:durative-action MakeS1
 :parameters (?c1 - COM1 ?m - MAN ?s1 - SUB1)
 :duration (= ?duration 1)
 :condition (and (at start (comp1 ?c1))
                 (at start (manip ?m)) )
 :effect (and (at end (subprod1 ?s1))
              (at start (not (comp1 ?c1)))
              (at start (not (manip ?m)))
              (at end (manip ?m)) ))
```

Fig. 15 PDDL encoding of the durative version of the MakeS1 action for Domain-1

The problem file for TFD is similar to Fig. 14, except a new directive to request minimization of the plan's make-span.

In addition to these two, we have also used a third PDDL encoding to support a more concise and efficient representation with numerical fluents and the PDDL feature *fluents* as shown in Fig. 16. This feature is supported by the TFD planner, allowing us provide a fair comparison with a modern planner and an encoding that is similar in spirit to LinGraph's aggregation of functionally identical resources.

In this definition, several numerical functions are defined to keep track of the available instances for the components, subcomponents and manipulators, as well as a special function `num` to allow the planner to explore using different object multiplicities. The definition for the MakeS1 action checks manipulator availability as a precondition, picks the number of manipulators to use through `(num ?u)` and adjusts the postconditions accordingly. It is assumed that the problem file will define the predicate `(num ?u)` for as many positive integers as there are manipulators. Figure 17 shows an example problem file with six manipulators and three of each component to produce three final products. We use TFD-NF to refer to this encoding in our results.

We first investigate problems in Domain-1 where there are enough manipulators to transform all components in the first level, allowing a solution in three levels. The associated LGPL encoding is

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap \left(C_1^n \otimes C_2^n \otimes M^{2n}\right) \multimap (P^n \otimes M^{2n})\,. \qquad (13)$$

```
(define (domain assembly-domain1-fluent)
 (:requirements :strips :equality :typing
                :durative-actions :fluents)
 (:functions (cnt-c1) (cnt-c2) (cnt-m) (cnt-s1)
             (cnt-s2) (cnt-p) (num ?n))
 (:durative-action MakeS1
  :parameters (?u)   :duration (= ?duration 1)
  :condition (and (at start (>= (cnt-c1) (num ?u)))
                  (at start (>= (cnt-m) (num ?u))))
  :effect (and (at start (decrease (cnt-c1) (num ?u)))
               (at start (decrease (cnt-m) (num ?u)))
               (at end (increase (cnt-s1) (num ?u)))
               (at end (increase (cnt-m) (num ?u)))))
```

Fig. 16 PDDL encoding of Domain-1 using numerical fluents to capture object multiplicity. This figure only shows the definition of the MakeS1 action

```
(define (problem prob-three-comp-six-manip)
(:domain assembly-domain1-fluent)
(:objects n1 n2 n3 n4 n5 n6)
(:init (= (num n1) 1) (= (num n2) 2) (= (num n3) 3)
       (= (num n4) 4) (= (num n5) 5) (= (num n6) 6)
       (= (cnt-c1) 3) (= (cnt-c2) 3)
       (= (cnt-s1) 0) (= (cnt-s2) 0)
       (= (cnt-m) 6) (= (cnt-p) 0) )
(:goal (and (= (cnt-p) 3)))
            (:metric minimize (total-time)) )
```

Fig. 17 An example PDDL problem definition for Domain-1 using numerical fluents

Table 1 shows execution times for all planners for different values of n. The optimized implementations for Blackbox, Symba-2, Metis, TFD and TFD-NF perform as well, and sometimes better than LinGraph for small values of n. However, the combinatorial choices for different instances of components quickly increase the search space, resulting in termination with a (possibly out-of-memory) errors for Blackbox, Symba-2, Metis, TFD and TFD-NF. In contrast, as a result of its aggregation of identical components, LinGraph can solve large instances of this problem. An important additional observation is that TFD-NF, despite its use of numerical fluents, fails to find valid plans after a certain problem size, whereas LinGraph performance stays constant independent of problem size.

The second Domain-1 example involves a smaller number of manipulators requiring plans to extend to level 4, with two new levels to produce all subproducts and a final level to produce the final products. The general form of the LGPL encoding for this problem is

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap \left(C_1^n \otimes C_2^n \otimes M^{n}\right) \multimap (P^n \otimes M^{n})\,. \qquad (14)$$

Results from this second problem instance are shown in Table 2 and exhibit scalability properties similar to the previous problem, showing that LinGraph can successfully handle large numbers of functionally identical objects without increasing the search space. Sequential planners,

Table 1 Execution times (in seconds) for Domain-1 with twice the number of initially available manipulators as the number of initial components n

n	BB	S-2	M	T	T-N	LG
1	0.004	0.292	0.144	0.022	0.096	0.056
2	0.008	0.672	0.204	7.788	0.112	0.055
3	0.021	1.104	0.604	Error	0.220	0.058
4	3.136	1.544	1.756	–	0.896	0.059
8	Error	11.88	166.2	–	257.7	0.059
32	–	Error	Error	–	Error	0.055
1000	–	–	–	–	–	0.061

LinGraph solutions have 4 levels

Table 2 Execution times (in seconds) for Domain-1 with the same number n of initially available components and manipulators

n	BB	S-2	M	T	T-N	LG
1	0.004	0.284	0.124	0.076	0.080	0.198
2	0.008	0.688	0.146	0.218	0.092	0.678
3	0.556	1.098	0.348	99m	0.156	0.775
4	43m	1.348	1.016	Error	0.520	0.788
5	>1d	1.964	2.616	–	2.716	0.752
1000	–	Error	Error	–	Error	0.794
10000	–	–	–	–	–	2.497

LinGraph solutions have 4 levels

Symba-2 and Metis, perform better in this second problem instance than the first since the smaller number of manipulators decrease the available number of actions in each step. TFD experiences similar performance gains since its underlying search relies on intermediate parallelization of sequential plans. In contrast, LinGraph and Blackbox natively support concurrency, exhibiting degraded performance due to the longer make-span required for the optimal solution.

Extending on the examples above, Table 3 shows results for the same domain, with the initial number of manipulators chosen to be a non-integer multiple of components. The associated LGPL encoding is

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \\ \multimap \left(C_1^n \otimes C_2^n \otimes M^m\right) \multimap (P^n \otimes M^m) \;, \tag{15}$$

which is also successfully solved by LinGraph.

In order to illustrate the internal operation of LinGraph, Table 4 shows node and constraint statistics for LinGraph solutions to different instances of (15). For all instances of Domain-1 problems, the same LinGraph structure was obtained beyond a certain problem size. This results from having a fixed number of possible combinations for different object types and actions, wherein only the multiplicities

Table 3 Execution times (in seconds) for Domain-1 with the number m of initially available manipulators a non-integer multiple of the number n of initially available components

n/m	BB	S-2	M	T	T-N	LG
2/3	0.012	0.696	0.192	1.936	0.120	0.774
3/5	1.580	1.148	0.548	7.248	0.240	0.739
4/6	47 m	1.512	1.348	13.855	2.016	0.783
5/8	Error	2.112	3.912	46.272	9.300	0.777
32/48	–	Error	Error	Error	Error	0.742
1000/1500	–	–	–	–	–	0.781

LinGraph solutions have 4 levels

Table 4 LinGraph level, node and constraint statistics for different instances of (15), showing number of levels, last level and total number of nodes, pruned nodes as well as the number of dependency and sibling constraints

n/m	Lvls	(Last/All)	Pr	Dep	Sib
1/1	4	15/36	70	21	6
2/2	4	87/116	136	29	42
3/3	4	185/216	60	31	91
4/4	4	221/252	24	31	109
5/5	4	223/254	22	31	110
1000/1000	4	223/254	22	31	110
32/48	4	223/254	22	31	110
32/64	3	21/31	4	10	9

for each node and action increase beyond a certain problem size. This also explains why the execution times for LinGraph remain the same (except differences due to constraint solver overhead and parsing of the problem) in Tables 1, 2 and 3 even when problem sizes are increased dramatically.

Finally, Table 5 shows node and constraint statistics for the expansion of each intermediate level for the Domain-1 problem instance with $n = m = 10000$. Once the LinGraph reaches level 4, there is a sharp increase in the number of created nodes due to many new combinations becoming available for how actions can be applied to nodes in level 3. As we will see in subsequent sections, however, not all problem domains entail such exponential increase in the number of nodes.

6.2 Domain-2: assembly without trivial parallelism

Domain-1 includes trivial parallelism wherein the production associated with each manipulator can be separately considered, decomposing the problem into multiple smaller subproblems that may be manageable for planners other than LinGraph. In this section, we extend Domain-1 to disallow such explicit parallelization by requiring that multiple Product instances are assembled in a final step into a so called "Final Product", extending (11) with the new action

$$\mathrm{MakeFP} : P \otimes P \otimes P \otimes P \otimes M \multimap FP \otimes M \;.$$

Table 5 Node and constraint statistics for each level of the LinGraph solution for Domain-1 with $n = m = 10000$

Lvl	Nodes	Pr	Dep	Sib
1	3	0	0	0
2	7	0	3	2
3	21	4	7	7
4	223	18	21	101

Table 6 Execution times (in seconds) for the problem of (16)

n	BB	S-2	M	T	T-N	LG
1	101.484	1.94	2.544	1.416	1.572	1.907
2	Error	78.696	27 min	> 1 d	Error	1.902
4	–	Error	Error	–	–	1.968
250	–	–	–	–	–	2.151
2500	–	–	–	–	–	8.087

LinGraph plans have 4 levels

We consider two different instances of this domain, one where all products are converted into final products, and another where there are leftover products. The first problem instance is encoded by the LGPL goal formula

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap !\mathrm{MakeFP} \multimap \left(C_1^{4n} \otimes C_2^{4n} \otimes M^{8n}\right) \multimap (FP^n \otimes M^{8n}) , \qquad (16)$$

wherein all initial $4n$ components are converted into products, which are then assembled into n final products. Table 6 shows execution times for this problem.

The second problem instance in this domain is encoded by the LGPL formula

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap !\mathrm{MakeFP} \multimap \left(C_1^{n} \otimes C_2^{n} \otimes M^{2n}\right) \multimap (P^m \otimes FP^r \otimes M^{2n}) , \qquad (17)$$

where from the initial n components, and the resulting n products, not all are converted into final products, resulting in m leftover products P such that $4r + m = n$. Table 7 shows execution times for this problem instance.

These results show that when trivial parallelism is not possible, LinGraph is still capable of generating feasible plans whereas the computational complexity of planners that consider different instances of such objects as distinct individuals quickly becomes impractical. This remains the case even when numerical fluents are used to manually aggregate identical objects.

Table 7 Execution times (in seconds) for the problem of (17)

$n/m/r$	BB	S-2	M	T	T-N	LG
5/1/1	Error	3.908	8.884	Error	13.69	1.847
6/2/1	–	8.676	49.84	–	203.1	1.910
9/1/2	–	243.7	> 1d	–	Error	1.997
19/3/4	–	Error	–	–	–	2.054
1200/400/200	–	–	–	–	–	4.804

LinGraph plans have 4 levels

6.3 Domain-3: cooperative multi-robot assembly

In this section, we introduce and investigate a new domain where wheel and body parts are transported into a central station to be assembled into bicycles. As shown in Fig. 18, there are three stations in this example, two supply stations l_1 and l_2 containing wheel and body parts respectively, and a third base station l_0 for hosting transportation robots and performing bicycle assembly. More general instances of this domain can of course be considered, but this example was chosen to illustrate distinguishing features of LinGraph.

Robots can move between locations connected with traversable paths. Objects can only be moved when they are picked up and carried by robots. As usual, our LGPL encoding for this domain begins with defining propositional atoms encoding components of the problem state, and individual actions for transforming state. Figure 19 summarizes these components, together with LGPL definitions and descriptions for all actions.

The desired final state for our example requires the base station to contain all n fully assembled bicycles and all r robots. Since LGPL currently lacks quantifiers, actions defined in Fig. 19 must be, as usual, explicitly instantiated for each location or location pair in the goal statement. In particular, $Move_{xy}$ actions are instantiated only for pairs of locations connected by traversable paths.

Choosing $n = 1$ and $r = 7$, LinGraph finds the solution shown in Fig. 20 that minimizes the plan make-span. Even though this is the simplest instance of the problem with only a small number of functionally identical objects, only BlackBox and sequential planners Symba-2 and Metis were capable of finding feasible solutions, with TFD failing with errors. As shown in Table 8, beyond $n = 3$ and $r = 21$, only LinGraph continues to find solutions as a result of its aggregation of functionally identical objects.

To go further, when the required plan length is extended to 9 steps from 5 steps by decreasing the number of available robots, the execution times detailed in Table 9 show that all temporal planners fail, whereas the sequential planners Symba-2 and Metis can only find solutions for the simplest case, with execution times substantially larger than those of LinGraph. In summary, these examples show that LinGraph is much more capable than existing automated planners in exploiting symmetries in concurrent planning problems with large numbers of functionally identical objects in ways that are not possible with current methods for symmetry reduction including the use of numerical fluents.

7 LinGraph, multiset rewriting and linear logic

In this section, we present a formalization of the connection between LinGraph and multiset rewriting, enabling us

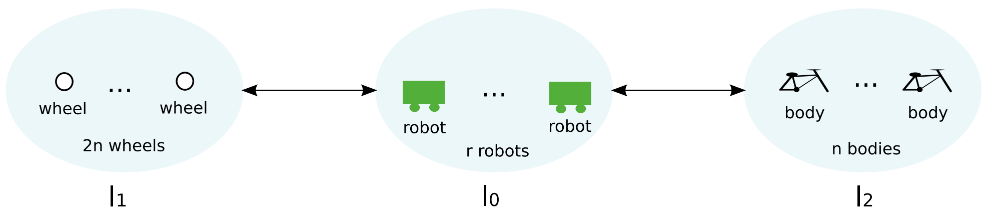

Fig. 18 The cooperative multi-robot domain with two supply stations, l_1 and l_2 and a base station l_0. This figure also illustrates the initial state for our examples, with $2n$ wheels, n body parts and r robots in their corresponding stations. Arrows indicate traversable routes

to prove soundness and completeness with respect to the LGPL fragment of linear logic. We show that LinGraph construction provides a sound and complete method for proof construction in this logic by first interpreting LinGraph as a multiset rewriting system, then using the well-known correspondence between such systems and theorem proving in linear logic [7, 41, 58].

We begin by adapting and reviewing key definitions on multiset rewriting systems from [58]. Given a set of propositions P, a finite *multiset* over P is a function $M : P \to \mathbb{N}$ such that $M(p)$ gives the multiplicity of the proposition p in the multiset. We write $p \in M$ if $M(p) \neq 0$. As noted in [35], a multiset $[p_1, ..., p_k]$ over P can also be associated with the product expression $(p_1 \otimes ... \otimes p_k)$. As such, multiplicative conjunction of two product expressions corresponds to the *additive multiset union* of two multisets M_1 and M_2 over P, which is the multiset defined by $(M_1 \uplus M_2)(p) := M_1(p) + M_2(p)$.

A *multiset rewriting rule* R is an ordered pair of multisets, $R = (M_1, M_2)$ over the set of propositions P, where M_1 and M_2 are called the *preset* and the *postset*, respectively. A rule $R = (M_1, M_2)$ is said to be *applicable* on a multiset M if $\forall p \in P, M_1(p) \leq M(p)$ encoding the requirement that the preset M_1 of R is a submultiset of M. In such cases, the application of the rule R on the multiset M *generates* a new multiset $M_{new} = (M \setminus M_1) \uplus M_2$ where $\setminus$ denotes multiset difference. Based on this definition, a *multiset rewriting system* (MRS) is simply a set of rewriting rules, $\mathcal{R} = \{R_1, ..., R_N\}$ which, in the context of problems in task planning, can be used to represent possible actions in a domain to transform system state.

Definition 1 (Multiset of a LinGraph Node) Given a LinGraph L with r nodes, and a solution vector $[s_1, ..., s_r]$ satisfying all of its constraints, each node n_i having the type encoded by the propositional atom P_i uniquely defines an associated multiset $M_{nd}(n_i) := \{P_i, ..., P_i\}$ consisting of s_i copies of the proposition P_i.

Definition 2 (Multiset of a LinGraph level) Given a LinGraph L with r nodes, l levels and a solution vector $[s_1, ..., s_r]$ satisfying all of its constraints, every level $j \leq l$ in L uniquely defines an associated multiset $M_{lvl}(j) := M_{nd}(n_{i_1}) \uplus ... \uplus M_{nd}(n_{i_m})$, where i_1 through i_m denote indices of all LinGraph nodes that belong to level j.

Since every action formula $F_a = p_1 \otimes ... \otimes p_k \multimap q_1 \otimes ... \otimes q_t$ can be associated with the multiset rewriting rule $R_{F_a} = (\{p_1, ..., p_k\}, \{q_1, ..., q_t\})$, every LinGraph instance defines a corresponding MRS, consisting of rewriting rules associated with all of its actions, together with rules corresponding to trivial "copy" actions $p \multimap p$ for all propositions p. We will denote this MRS with $\mathcal{R}_L$. The following theorem is a key step in establishing that every LinGraph, when considered together with a particular solution with its constraints, encodes a valid sequence

Objects:

$wheelat_x$	:	There is a wheel in l_x
$bodyat_x$	:	There is a body in l_x
$bicycleat_x$	:	There is a bicycle in l_x
$rbtat_x$	:	There is a robot in l_x
$wteamat_x$	:	A 2 robot team holding a wheel are in l_x
$bteamat_x$	:	A 3 robot team holding a body are in l_x

Actions:

Move$_{xy}$	:	$rbtat_x \multimap rbtat_y$ (Robot moves from l_x to l_y)
HoldWheel$_x$	:	$(rbtat_x)^2 \otimes wheelat_x \multimap (wheelteamat_x)$ (Two robots pick up a wheel in l_x)
HoldBody$_x$	:	$(rbtat_x)^3 \otimes bodyat_x \multimap (bodyteamat_x)$ (3 robots pick up a body in l_x)
RlsWheel$_x$	:	$wheelteamat_x \multimap (rbtat_x)^2 \otimes wheelat_x$ (2 robots release a wheel in l_x)
RlsBody$_x$	:	$bodyteamat_x \multimap (rbtat_x)^3 \otimes bodyat_x$ (3 robots release a body in l_x)
CarryWheel$_{xy}$	:	$wheelteamat_x \multimap wheelteamat_y$ (2 robots take a wheel from l_x to l_y)
CarryBody$_{xy}$	:	$bodyteamat_x \multimap bodyteamat_y$ (3 robots take a body from l_x to l_y)
AssemBike$_x$	:	$bodyat_x \otimes wheelat_x^2 \multimap bicycleat_x$ (A bicycle is assembled in location l_x)

Fig. 19 LGPL encoding for the cooperative multi-robot assembly example, detailing propositions encoding state components and LGPL formulae encoding actions

Step 1:	Move$^4_{01}$,	Move$^3_{02}$
Step 2:	HoldWheel$^2_1$,	HoldBody$_2$
Step 3:	CarryWheel$^2_{10}$,	CarryBody$_{20}$
Step 4:	RlsWheel$^2_0$,	RlsBody$_0$
Step 5:	AssemBicycle$_0$	

Fig. 20 Solution to the multi-robot example with $n = 1$ and $r = 7$, generated by the LinGraph planner

Table 8 Execution times (in seconds) for different planners in solving increasingly large instances of the multi-robot planning example

n/r	BB	S-2	M	T	T-N	LG
1/7	38 min	50 min	445.6	Error	Error	0.130
2/14	Error	Error	Error	–	–	0.130
3/21	–	–	–	–	–	0.131
4/28	–	–	–	–	–	0.130
128/896	–	–	–	–	–	0.130

LinGraph extends to level 6 in all cases, corresponding to 5 steps. Other planners are unable to solve any of the problem instances

of rewriting rule applications starting from its initial level, ending with its last level.

Theorem 1 *(LinGraph Expansion) Given a LinGraph L with r nodes, two successive levels i and $i+1$ and a solution vector $[s_1, ..., s_r]$ satisfying all of its constraints, there exists a (not necessarily unique) sequence of rewriting rules, $[R_1, ..., R_u]$ with $\forall j \leq u, R_j \in \mathcal{R}_L$, with an associated sequence of intermediate multisets $[M_0, ..., M_u]$ such that $M_0 = M_{lvl}(i)$, $\forall j \leq u, M_j$ is generated by R_j from M_{j-1} and $M_u = M_{lvl}(i+1)$.*

Proof The proof proceeds by iteration through action nodes connecting level i to level $i+1$ in L. Consider an ordering of these actions, $\mathcal{A} = [F_{A_1}, ..., F_{A_k}]$. As a result of sibling constraints and the structure of L, *all resources* in the final multiset $M_{lvl}(i+1)$ can be associated with their corresponding generating actions. Consequently, if a proposition $p \in M_{lvl}(i+1)$, then we have $s_p > 0$ and by construction, there exists an action $F_A \in \mathcal{A}$ has p in its postconditions. Moreover, as a result of the sibling constrains being satisfied, $M_{lvl}(i+1)$ is also guaranteed to have a consistent number of propositions corresponding to all the postconditions of this action. This ensures that $M_{lvl}(i+1) \subseteq M_u$.

On the other hand, dependency constraints between node multiplicities ensure that exactly the number of available resources in level i are consumed by the application of actions connecting level i to level $i+1$. Consequently, the multiset $M_{lvl}(i)$ is entirely replaced by the postconditions of all the actions in A. This ensures that only nodes in level $i+1$ can be in the corresponding multiset, meaning that $M_u \subseteq M_{lvl}(i+1)$. This proves that $M_u = M_{lvl}(i+1)$. The multisets M_1 through M_{u-1} are then generated by successive application of rewriting rules associated with actions in A. □

Table 9 Execution times (in seconds) for different planners on the multi-robot planning example with fewer manipulators

n/r	BB	S-2	M	T	T-N	LG
1/5	> 1 d	30.986	41.15	Error	Error	1.524
2/10	–	> 1 day	Error	–	–	3.846
4/20	–	–	–	–	–	3.846
128/640	–	–	–	–	–	4.350

LinGraph extends to level 10 in all cases (9 steps). Other planners abort with errors beyond a certain problem size

In order to establish the soundness of LinGraph with respect to LGPL semantics, we first review standard proof theoretic semantics for the $LGPL$ language, adapted from [47]. We use a sequent calculus formulation, with the corresponding sequent definition $\Gamma; \Delta \Rightarrow \mathcal{F}_G$, where Δ is a multiset of atomic resources and Γ is a multiset of action formulae. This sequent states that the goal formula $\mathcal{F}_G$ can be proven using the single-use resources in Δ and unlimited use propositions in Γ. Sequent calculus is generally used to formalize proof construction for logical languages, wherein "left" and "right" inference rules capture how occurrences of connectives as assumptions or goals, respectively, can be refined to recursively construct a valid proof. Sequent calculus rules for $LGPL$ are detailed in Fig. 21, where we also follow the conventions of [47].

All together, these inference rules enforce the requirement that all resources appearing in Δ on the left hand side of a sequent must be used exactly once in the associated proof, thereby allowing their interpretation as consumable resources [14]. This property, combined with our choice of an *intuitionistic* fragment of linear logic make it possible to associate each proof for $LGPL$ goal formulae with a valid plan.

Theorem 2 *(Soundness of LinGraph) Given a goal formula F_G in LGPL, the corresponding completed LinGraph L with r nodes, its last level l matching the decomposed goal from* Section 5.1.2 *and a valid solution vector $[s_1, ..., s_r]$ satisfying all of its constraints, there exists a proof in the system of* Fig. 21 *for F_G.*

$$\frac{}{\Gamma; p \Rightarrow p}\ init \qquad \frac{(\Gamma, \mathcal{F}_A); (\Delta, \mathcal{F}_A) \Rightarrow \mathcal{F}_G}{(\Gamma, \mathcal{F}_A); \Delta \Rightarrow \mathcal{F}_G}\ copy$$

$$\frac{(\Gamma, \mathcal{F}_A); \Delta \Rightarrow \mathcal{F}_G}{\Gamma; (\Delta, !\mathcal{F}_A) \Rightarrow \mathcal{F}_G}\ !L \qquad \frac{\Gamma; \Delta_1 \Rightarrow p \quad \Gamma; \Delta_2 \Rightarrow \mathcal{F}_R}{\Gamma; \Delta_1, \Delta_2 \Rightarrow p \otimes \mathcal{F}_R}\ \otimes R$$

$$\frac{\Gamma; \Delta, p, \mathcal{F}_R \Rightarrow \mathcal{F}_G}{\Gamma; \Delta, p \otimes \mathcal{F}_R \Rightarrow \mathcal{F}_G}\ \otimes L \qquad \frac{\Gamma; \Delta, \mathcal{F}_P \Rightarrow \mathcal{F}_G}{\Gamma; \Delta \Rightarrow \mathcal{F}_P \multimap \mathcal{F}_G}\ \multimap R$$

$$\frac{\Gamma; \Delta_1 \Rightarrow \mathcal{F}_{R1} \quad \Gamma; \Delta_2, \mathcal{F}_{R2} \Rightarrow \mathcal{F}_G}{\Gamma; \Delta_1, \Delta_2, \mathcal{F}_{R1} \multimap \mathcal{F}_{R2} \Rightarrow \mathcal{F}_G}\ \multimap L$$

Fig. 21 Sequent calculus proof rules for $LGPL$

Proof LGPL formulae have the form shown in (1). Proving the validity of this formula corresponds to proving the sequent $(F_{A_1}, ..., F_{A_a}); (F_{R_1}, ..., F_{R_b}) \Rightarrow F_{R_g}$ in the system of Fig. 21, following straightforward elimination of implications and exponentials using the rules $\multimap L$, $\otimes L$ and $!L$. In the LinGraph initialization phase, all resource formulae in the linear context, F_{R_1} through F_{R_b} are converted into nodes, forming the first layer of the graph. Together with associated initial constraints described in Section 5.1.2, these ensure that the multiset $M_{lvl}(1)$ exactly corresponds to the linear context in this sequent. Similarly, the goal formula F_{R_g} is converted into a set of goal nodes which, through the use of goal check constraints satisfied in the last stage of LinGraph construction ensure that the multiset $M_{lvl}(l)$ associated with the last level of L exactly corresponds to the desired goal formula F_{R_g}. Iterative application of Theorem 1, shows that there is a finite sequence of multiset rewriting rule applications connecting the initial multiset to the final multiset associated with the goal formula. Finally, this sequence of rules can be transformed into a proof in the multiplicative Horn fragment of linear logic, establishing soundness [58]. □

Theorem 3 *(Completeness of LinGraph) Given a goal formula F_G in LGPL, provable in the system of* Fig. 21, *the algorithm described in* Section 5.1 *is guaranteed to terminate and produce a valid LinGraph for F_G.*

Proof Without loss of generality, the proof of the formula F_G in (1), can be assumed to start with the elimination of implications and exponentials, thereby having a subproof for the sequent

$$(F_{A_1}, ..., F_{A_a}); (F_{R_1}, ..., F_{R_b}) \Rightarrow F_{R_g} \ , \qquad (18)$$

Since this sequent is free of exponentials, the corresponding proof will be in the multiplicative Horn fragment of linear logic, and hence can be transformed into a sequence $S = [R_1, ..., R_u]$ of multiset rewriting rule applications corresponding to left rules for actions, $(F_{A_1}, ..., F_{A_a})$, starting from the initial multiset associated with the initial context $I = (F_{R_1}, ..., F_{R_b})$ [58]. By construction, action nodes in a particular LinGraph level capture *all possible applications* of available multiset rewrite rules, each of which corresponding to a particular solution for the cumulative set of constraints associated with the LinGraph.

The sequence S defined above can be permuted to collect all mutually independent applications related to the initial multiset I to the beginning of the sequence due to their inherent concurrency [7]. By construction, each such rule applied to the initial multiset, will then have a corresponding action node in the LinGraph, with the subsequent level having nodes associated with the elements in the postset of the rule. Once all rules applicable to the initial set are exhausted, the second level of the LinGraph will be formed, and the construction will proceed recursively until all the rules in S are covered. Finally, since the last level in the LinGraph constructed in this fashion will have nodes corresponding to all components of F_{R_g}, the goal check is guaranteed to succeed, ensuring termination of LinGraph construction, with a valid solution for all constraints in the LinGraph. □

8 Conclusion and future work

In this paper, we introduced LinGraph, a new, model-based, domain-independent automated planner based on a fragment of propositional intuitionistic Linear Logic (LGPL), and a graph-based strategy for plan generation. Our planner benefits from the non-monotonicity of linear logic to allow simple and effective encoding of dynamic state. Our novel graph-based method implements forward proof-search within the LGPL fragment of linear logic, providing an effective means of automated plan construction. The intuitionistic nature of LGPL preserves strict correspondence between valid proofs and plans, admitting the use of this system as an automated planner.

A distinguishing feature of LinGraph is its ability to eliminate irrelevant combinatorial nondeterminism in plan search when there are multiple, functionally identical objects within a problem. Our graph-based search strategy aggregates multiple instances of such indistinguishable components of the state, admitting efficient plan search for problems that are otherwise intractable. We illustrate both the basic operation of the planner, as well as its performance on increasingly complex instances of a simple assembly planning domain incorporating multiple identical instances of different types of components and manipulators. In this context, we provide a comparison of execution times for LinGraph with four planners: Blackbox, Symba-2, Metis and the Temporal Fast Downward (TFD) system, using the TFD system first with a simple encoding treating objects as unique, and a second using numerical fluents to more efficiently encode equivalent object multiplicity. We show that even though our unoptimized implementation of LinGraph, with its current blind search strategy, does not necessarily outperform existing planners for small concurrent problems or long sequential problems, it is much more scalable and maintains its ability to identify feasible plans for increased problem sizes.

In addition to a presentation of LinGraph and these experimental results, we provided a brief analysis of algorithmic properties of LinGraph, followed by theoretical results establishing the soundness and completeness of LinGraph's plan construction with respect to a standard proof theory for the LGPL fragment of intuitionistic linear logic.

This was accomplished by interpreting LinGraph instances as multiset rewriting systems, which are known to correspond to Petri Nets and also proofs in Horn fragments of propositional linear logic. These results provide a formal connection between LinGraph generated plans and the semantics of goal formulae in the LGPL fragment of linear logic.

Possible extensions to our work can be categorized in three directions. First, different sources of complexity in LinGraph construction can be addressed. In particular, defining a concept of state node equivalence, and introducing the ability to combine equivalent nodes during graph expansion might substantially reduce LinGraph size and allow longer plans to be generated. This could be extended with the detection of loops wherein the effects of an action are completely reversed by a subsequent action. Methods in existing planning literature focusing on state equivalence and loop detection could be applicable to LinGraph to address these issues.

The second category of improvements involve the action creation and goal check stages. Currently, LinGraph expands the graph in a strict layered structure, performing blind, breadth-first search. Incorporating effective heuristics to properly guide graph generation would substantially improve performance on sequential problems. Similarly, the current LinGraph implementation uses an uninformed constraint solver during the goal check, and partial usage counts for action creation. A new instance of the constraint solver is invoked for every check, discarding previous work that might have helped eliminate assignments that are altogether infeasible. A tighter integration with graph generation and an incremental constraint solver could improve the performance of both graph generation and the goal check.

The third and final direction for future work involves extensions to the expressivity of the LGPL language, which currently only allows discrete actions with no explicit support for conditional statements or nondeterminism. On the other hand, linear logic introduces additive connectives an other components that could potentially provide richer expressivity for encoding planning problems. Such more expressive uses of linear logic for planning problems has not been sufficiently explored in previous literature, but could allow more uniform modeling of more advanced planning structures such as hierarchical plans, dynamic management of available actions and nondeterminism. In addition to the semantic formulation of the connection between these logical connectives and the planning domain, both the structure and the construction of LinGraph would need to be extended to correctly handle proof construction in the presence of new connectives. Other possible extensions include the incorporation of continuous constraint expressions into linear logic as was proposed in [54], allowing native modeling of hybrid systems.

Acknowledgments This work was supported TUBITAK projects 109E032 and 114E277. We thank Prof. Frank Pfenning for his insights and guidance as well as Prof. Ozan Kahramanoğulları for his feedback on our work.

References

1. Alkhazraji Y, Katz M, Matmüller R, Pommerening F, Shleyfman A, Wehrle M (2014) Metis: arming fast downward with pruning and incremental computation. In: Eighth international planning competition, deterministic part, pp 88–92
2. Belta C, Bicchi A, Egerstedt M, Frazzoli E, Klavins E, Pappas G (2007) Symbolic planning and control of robot motion - grand challenges of robotics. IEEE Robot Autom Mag 14(1):61–70
3. Blum AL, Furst ML (1995) Fast planning through planning graph analysis. Artif Intell 90(1):1636–1642
4. Bonet B, Geffner H (2001) Planning as heuristic search. Artif Intell 129(1-2):5–33
5. Bonet B, Loerincs G, Geffner H (1997) A robust and fast action selection mechanism for planning. In: Proceedings of the national conference on artificial intelligence, pp 714–719
6. Bylander T (1994) The computational complexity of propositional STRIPS planning. Artif Intell 69(1-2):165–204
7. Cervesato I, Scedrov A (2009) Relating state-based and process-based concurrency through linear logic. Inf Comput 207(10):1044–1077
8. Chrpa L (2006) Linear logic in planning. In: Proceedings of the international conference on automated planning and scheduling, pp 26–29
9. Cimatti A, Pistore M, Roveri M, Traverso P (2003) Weak, strong, and strong cyclic planning via symbolic model checking. Artif Intell 147(1–2):35–84
10. Coles A, Coles A, Olaya AG, Jimenez S, Lopez CL, Sanner S, Yoon S (2012) A survey of the seventh international planning competition. AI Mag 33(1)
11. Cresswell S, Smaill A, Richardson J (2000) Deductive synthesis of recursive plans in linear logic. In: Biundo S, Fox M (eds) Recent advances in AI planning, volume 1809 of lecture notes in computer science. Springer, Berlin, pp 252–264
12. Cushing W, Kambhampati S, Mausam, Weld DS (2007) When is temporal planning really temporal? In: Proceedings of the international journal of conference on artificial intelligence. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, pp 1852–1859
13. Dimopoulos Y, Nebel B, Koehler J (1997) Encoding planning problems in nonmonotonic logic programs. In: Recent advances in AI planning, vol 1348, pp 169–181
14. Dixon L, Smaill A, Tsang T (2009) Plans, actions and dialogue using linear logic. J Log Lang Inf 18:251–289
15. Domshlak C, Katz M, Shleyfman A (2012) Enhanced symmetry breaking in cost-optimal planning as forward search. In: Proceedings of the international conference on automated planning and scheduling
16. Eiter T, Faber W, Leone N, Pfeifer G, Polleres A (2003) Answer set planning under action costs. J Artif Intell Res 19:25–71
17. Eyerich P, Keller T, Aldinger J, Dornhege C (2014) Preferring preferred operators in temporal fast downward. In: Eighth international planning competition (IPC 2014), deterministic part, pp 88–92
18. Eyerich P, Mattmüller R, Röger G (2009) Using the context-enhanced additive heuristic for temporal and numeric planning. In: Proceedings of the international conference on automated planning and scheduling

19. Fox M, Long D (1999) The detection and exploitation of symmetry in planning problems. In: Proceedings of the international journal conference on artificial intelligence, pp 956–961
20. Fuentetaja R, de la Rosa T (2015) Compiling irrelevant objects to counters. Special case of creation planning. AI Commun:1–33. Preprint
21. Geffner H, Bonet B (2013) A concise introduction to models and methods for automated planning. Synthesis Lectures on Artificial Intelligence and Machine Learning 7(2):1–141
22. Gent IP, Jefferson C, Miguel I (2006) MINION: a fast, scalable, constraint solver. In: Proceedings of the european conference on artificial intelligence, pp 98–102
23. Girard J-Y (1987) Linear logic. Theor Comput Sci 50(1):1–102
24. Helmert M (2006) The fast downward planning system. J Artif Intell Res 26:191–246
25. Helmert M, Geffner H (2008) Unifying the causal graph and additive heuristics. In: Proceedings of the international conference on automated planning and scheduling, pp 140–147
26. Hickmott SL, Rintanen J, Thiébaux S, White LB (2007) Planning via petri net unfolding. In: Proceedings of the international joint conference on artificial intelligence, vol 7, pp 1904–1911
27. Hoffmann J, System Metric-FFPlanning (2003) Translating "Ignoring delete lists" to numeric state variables. J Artif Intell Res 20:291–341
28. Hoffmann J, Nebel B (2001) The FF planning system: fast plan generation through heuristic search. J Artif Intell Res 14(1):253–302
29. Hoffmann J, Porteous J, Sebastia L (2004) Ordered landmarks in planning. J Artif Intell Res 22:215–278
30. Jacopin E (1993) Classical AI planning as theorem proving: the case of a fragment of linear logic. CA, Palo Alto
31. Kahramanogullari O (2009) On linear logic planning and concurrency. Inf Comput 207(11):1229–1258
32. Kambhampati S (2000) Planning graph as a: (Dynamic) CSP: exploiting EBL, DDB and other CSP search techniques in Graph-Plan. J Artif Intell Res 12:1–34
33. Kanovich M, Vauzeilles J (2001) The classical AI planning problems in the mirror of horn linear logic: semantics, expressibility, complexity. Math Struct Comput Sci 11(6):689–716
34. Kanovich M, Vauzeilles J (2007) Strong planning under uncertainty in domains with numerous but identical elements (a generic approach). Theor Comput Sci 379(1-2):84–119
35. Kanovich MI (1994) Linear logic as a logic of computations. Ann Pure Appl Logic 67(1):183–212
36. Kautz H, Selman B (1992) Planning as satisfiability. In: Proceedings of the european conference on artificial intelligence. Wiley, pp 359–363
37. Kautz H, Selman B (1998) Blackbox: a new approach to the application of theorem proving to problem solving. In: Proceedings of the AIPS98 workshop on planning as combinatorial search, pp 58–60
38. Kungas P (2003) Linear logic for domain-independent AI planning. Trento, Italy
39. Levesque HJ, Reiter R, Lespérance Y, Lin F, Scherl RB (1997) Golog: a logic programming language for dynamic domains. J Log Program 31(1):59–83
40. Lifschitz V (1999) Answer set planning. In: Gelfond M, Leone N, Pfeifer G (eds) Logic programming and nonmonotonic reasoning, volume 1730 of lecture notes in computer science. Springer, Berlin, pp 373–374
41. Martí-Oliet N, Meseguer J (1989) From petri nets to linear logic. In: Pitt DH, Rydeheard DE, Dybjer P, Pitts A, Poigne A (eds) Category theory and computer science, volume 389 of lecture notes in computer science, pp 313–340
42. Masseron M, Tollu C, Vauzeilles J (1993) Generating plans in linear logic I. Actions as proofs. Theor Comput Sci 113(2):349–370
43. McAllester D, Rosenblitt D (1991) Systematic nonlinear planning. In: Proceedings of the national conference of the american association for artificial intelligence (AAAI-91), pp 634–639
44. McCarthy J (1969) Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence 4:463–502
45. Minker J (1993) An overview of nonmonotonic reasoning and logic programming. J Log Program, Special Issue 17:95–126
46. Nau DS (2007) Current trends in automated planning. AI Mag 28(4):43
47. Pfenning F (2002) Lecture notes on linear logic. Technical report, Carnegie Mellon University
48. Pochter N, Rosenschein JS, Zohar A (2011) Exploiting problem symmetries in state-based planners. In: Proceedings of the AAAI conference on artificial intelligence
49. Richter S, Westphal M (2010) The LAMA planner: guiding cost-based anytime planing with landmarks. J Artif Intell Res 39:127–177
50. Riddle P, Barley M, Franco S, Douglas J (2015) Analysis of bagged representations in PDDL. In: Heuristics and search for domain-independent planning, pp 71–79
51. Rintanen J (2012) Planning as satisfiability: heuristics. Artif Intell 193:45–86
52. Robinson N, Gretton C, Pham DN, Sattar A (2009) SAT-based parallel planning using a split representation of actions. In: Proceedings of the international conference on automated planning and scheduling
53. Russel S, Norvig P (1995) Artificial intelligence: a modern approach. Prentice Hall
54. Saranli U, Pfenning F (2007) Using constrained intuitionistic linear logic for hybrid robotic planning problems. In: Proceedings of the IEEE international conference on robotics and automation, pp 3705–3710
55. Silva F, Castilho MA, Künzle LA (2000) Petriplan: a new algorithm for plan generation. In: Advances in artificial intelligence. Springer, pp 86–95
56. Smith DE (2003) Choosing objectives in over-subscription planning. In: Proceedings of the international conference on automated planning and scheduling
57. Torralba A, Alcazar V, Borrajo D, Kissmann P, Edelkamp S (2014) Symba: a symbolic bidirectional a planner. In: Eighth international planning competition (IPC 2014), deterministic part, pp 105–108
58. Tzouvaras A (1998) The linear logic of multisets. Log J IGPL 6(6):901–916
59. Vidal V, Geffner H (2006) Branching and pruning: an optimal temporal POCL planner based on constraint programming. Artif Intell 170(3):298–335

Sıtar Kortik received his B.Sc. degree in Computer Engineering from Dokuz Eylül University, Izmir, Turkey in 2006 and his M.Sc. degree in Computer Engineering from Bilkent University, Ankara, Turkey in 2010. He was a visiting researcher at the Computer Science Department, Carnegie Mellon University, USA in 2012. Currently, he is a Ph.D. candidate in Computer Engineering Department, Bilkent University. His research interests include artificial intelligence, automated reasoning, robotic task planning, applications of linear logic and theorem proving to domain independent task planning and multiagent systems.

Uluç Saranlı received his B.Sc. degree in Electrical and Electronics Engineering from the Middle East Technical University, Ankara, Turkey in 1996 and his M.Sc. and Ph.D. degrees in Computer Science from the University of Michigan, Ann Arbor in 1998 and 2002, respectively. He then joined the Robotics Institute in Carnegie Mellon University as a postdoctoral fellow until 2005. Subsequently, he served as an Assistant Professor in the Department of Computer Engineering in Bilkent University, Ankara, Turkey until 2011, after which he joined the Department of Computer Engineering in the Middle East Technical University as an Associate Professor. His research interests include the analysis and control of dynamic locomotion with legged robots, nonlinear dynamical systems, embedded systems, software architectures for robot programming and control and formal methods applied to planning and robotic autonomy.